Diplomová práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Verzování v mikroslužbové architektuře

Diplomová práce

Bc. Vojtěch Svoboda

Vedoucí: Ing. Martin Chloupek, Ph.D.
Obor: Softwarové inženýrství
Studijní program: Otevřená informatika
Květen 2020

Poděkování

Chtěl bych tímto poděkovat vedoucímu práce Ing. Martinu Chloupkovi, Ph.D. za cenné rady a připomínky k práci. Dále bych chtěl poděkovat Ing. Jiřímu Novákovi za podnětné připomínky a názory v průběhu vypracovávání práce. Mé poděkování také patří všem, kteří se podíleli na korektuře a v neposlední řadě přítelkyni a rodině, kteří mě v průběhu celé práce podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze, 22. května 2020

Abstrakt

Tato diplomová práce se zabývá problematikou verzování v mikroslužbové architektuře. V rámci této práce je navržen způsob verzování jednotlivých služeb pro udržení jasných verzí včetně externích závislostí, dále je navrženo a implementováno CI/CD pro mikroslužbové projekty a pro jednotlivé služby a následně jsou tyto návrhy implementovány na projektu firmy Quanti s.r.o. Součástí této práce je také nástroj pro správu mikroslužbových aplikací s jednoduchým uživatelských rozhraním, který poskytuje přehled nasazených verzí služeb pro jednotlivá prostředí a umožňuje spouštět nasazení služeb a kontrolovat průběh testů a nasazení.

Pro potřeby této práce jsou využity nástroje GitLab, Artifactory a Graylog, pro správu kontejnerů je použit Docker. Práce se skládá z YAML, bash a Python skriptů. Pro uživatelské rozhraní je použit Python framework Flask, který využívá Jinja šablon.

Klíčová slova: mikroslužby, verzování, CD/CI, Docker, proces nasazení, verzování mikroslužeb, mikroslužbová architektura

Vedoucí: Ing. Martin Chloupek, Ph.D.
Quanti s.r.o.,
Thákurova 4,
Praha 6

Abstract

This master thesis deals with the topic of versioning in microservice architecture. A new way of versioning of individual services is proposed to maintain clear versions of services including external dependencies. Furthermore, CI/CD for microservice projects and individual services has been designed, implemented and applied to a project of company Quanti s.r.o. A microservice application management tool with simple user interface has been developed as a part of this work. The tool displays a summary of deployed versions of services for individual environments and allows the user to deploy the services and control the testing and deployment flow.

This work is based on GitLab, Artifactory and Graylog and is scripted in YAML, bash and Python. Docker is used for container management. User interface is implemented in Flask, a Python-based framework that leverages Jinja templating.

Keywords: microservices, versioning, CD/CI, Docker, deploy process, microservices versioning, microservice architecture

Title translation: Versioning in microservice architecture — Diploma thesis

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Architektura využívající mikroslužeb zažívá v posledních letech velký nárůst v popularitě. Jedná se o architekturu aplikací, ve které se aplikace jako celek skládá z menších, volně vázaných aplikací, neboli služeb[1]. Nárůst popularity této architektury je dán také příchodem agilních způsobů vývoje, který byl konkrétně definován experty na vývoj software začátkem 21. století[2], kde jsou vývojáři rozdělení do týmů a je snaha o krátké vývojové cykly s průběžným dodáním do produkce, což by bylo velmi náročné u monolitních aplikací.

Tato práce je motivována nedostatky ve verzování a nasazování služeb v mikroslužbové architektuře. Jedná se zejména o chybějící nástroje, které by usnadňovaly správu a poskytovaly přehled, jaká verze služby je na daném prostředí, dále nedostatečnost verzování pouze zdrojových kódů. Na velkých projektech se často verzování mění v průběhu vývoje i několikrát, což znesnadňuje práci zejména administrátorům.
Z hlediska provozu je velmi reálné nebezpečí chyb v aplikaci z důvodu vágně specifikovaných verzí externích závislostí, případně ztráty možnosti tyto závislosti získat (například z důvodu nečekaného výpadku). Z toho důvodu velké IT firmy často vytváří a udržují vlastní repozitáře závislostí a snižují tak závislost na externích službách.

Hlavním cílem této práce je vymyslet a navrhnout takový **způsob verzování služeb a celé mikroslužbové architektury**, který bude automatizovaný a který nebude závislý na údržbě vzdálených repozitářů závislostí.
Druhým cílem je **získat centrální přehled**, jaké verze jsou na jednotlivých prostředích nasazené, což umožní jednoduchou a správnou konfiguraci prostředí pro reprodukci chyby. Tato funkce je vhodná nejen pro vývojáře

a testery, ale zejména pro vedoucí projektů.
Třetím cílem této práce je usnadnit vývojářům na projektu replikaci vybraného prostředí pro reprodukci chyb a jejich snadnější opravu. Tato možnost dále umožní vývojářům předejít možným problémům v integraci.
Dalším cílem je zavést co možná největší míru automatizace sestavení, testování i nasazení s rychlou zpětnou vazbou vývojářům, čímž se předejde nasazování chyb a usnadní kontrola opravených chyb. V neposlední řadě je potřeba **detekce chyb při běhu aplikace** z logů aplikace a serveru a jejich následné centrální zpracování.

Součástí této práce je rešerše, ve které je popsána mikroslužbová architektura, problematika agilního vývoje, možnosti centrálního zpracování logů a možnosti kontejnerového prostředí. Dále je navržen a implementován proces pro verzování, testování a správu mikroslužbových aplikací včetně nástroje, který poskytuje centrální správu a přehled. Tento proces a nástroj byl implementován a otestován v praxi. V poslední kapitole je práce vyhodnocena.

Kapitola 2

Rešerše

Kapitola rešerše se věnuje shrnutí informací potřebných k vytvoření této práce. Bude se zabývat především definicí mikroslužbové architektury, přiblížením moderních postupů v agilním vývoji a rolí CI/CD. Dále se bude také věnovat nástrojům pro agregaci a analýzu logů a v neposlední řadě i možnosti testování aplikací.

2.1 Mikroslužbová architektura

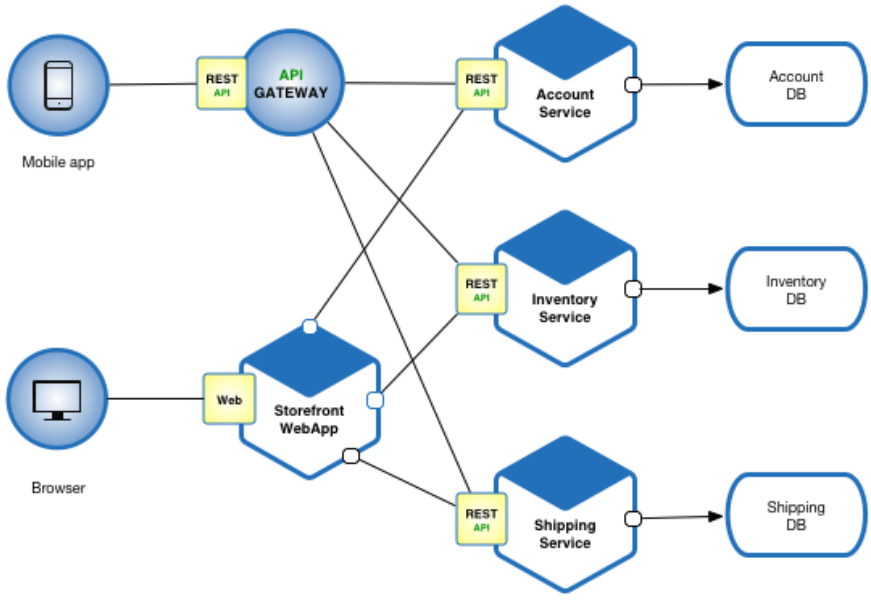
Tato kapitola se zabývá mikroslužbovou architekturou (microservices architecture). Zaměříme se zde na definici mikroslužbové architektury, typické využití a porovnání s monolitickou architekturou.

2.1.1 Úvod do problematiky

Užití mikroslužeb (microservices) je architekturální přístup ke stavbě aplikace[3]. Jde o přístup, kdy se jediná aplikace staví jako soubor malých funkcí, neboli služeb, z nichž každá běží ve vlastním procesu (nebo také případně na vlastním hostu). Tyto služby spolu navzájem komunikují jednoduchým lightweight mechanismem (nejčastěji HTTP API)[4]. Příklad aplikace s mikroslužbami je zobrazen na obrázku 2.1. Každá funkce aplikace se nazývá služba (service),

tyto služby jsou stavěné a organizované podle jejich business schopností (business capabilities). Celá architektura obsahuje minimální, nebo vůbec žádné, centralizované řízení[5].
Jednotlivé služby nemusí být napsané ve stejném jazyce a ani používat společné datové úložiště, to umožňuje vývojářům použití nejvhodnější možné technologie. Služby jsou snadno spravovatelné, udržovatelné a testovatelné. Jsou jen volně provázané a jsou nezávisle na sobě nasazovatelné (deployovatelné), nejčastěji automatickým procesem. Každá služba bývá často vyvíjena právě jedním týmem.
Takováto architektura umožňuje velmi rychlý a flexibilní vývoj s krátkými vývojovými cykly[6].

Mikroslužbová architektura je snadno škálovatelná a nasazovatelná přes různé servery a infrastruktury. Mikro v názvu bývá často spojováno s velikostí jednotlivých služeb. To ale je chyba, mikroslužba může být malý jednoduchý kus softwaru stejně jako velká služba, podle jejích business schopností[7]. Tím, že spolu služby jen komunikují, ale navzájem na sobě nezávisí, je aplikace stabilnější a odolnější vůči pádům jednotlivých služeb[8].



Obrázek 2.1: Příklad mikroslužbové architektury[9]

Snaha stavět systémy tím, že poskládáme několik komponent dohromady, je v odvětví téměř už 40 let. V posledních několika desítkách let se tímto směrem pokročilo využíváním knihoven, které jsou součástí většiny programovacích jazyků. Komponentu systému definuje Martin Fowler takto: Komponenta je jednotka softwaru, která je nezávisle nahraditelná a upgradovatelná[6]. Mikroslužbová architektura využívá knihovny, ale primárním cílem kompo-

nentizace je rozložení aplikace do služeb. Knihovny jsou tedy komponenty, které jsou linkované do programu a volané pomocí in-memory funkčních zavolání, zatímco služby jsou mimoprocesové komponenty, které komunikují pomocí mechanismů jako je HTTP požadavek (request) nebo vzdálené volání procedur (remote procedure call).
Jedním z hlavních důvodů, proč využíváme služby jako komponenty, je možnost nezávislého nasazení jednotlivých služeb, což znamená, že nemusíme přenasadit celou aplikaci kvůli změnám v jen jedné službě[10]. Samozřejmě toto tvrzení neplatí na 100 %, pokud jsou změny ve službě velké – například pokud se změní její aplikační rozhraní – bude vyžadována koordinace a pravděpodobně úprava i dalších služeb, ale cílem dobře navržené mikroslužbové architektury je minimalizovat takovéto případy.
Důsledkem využívání služeb jako komponent je explicitní rozhraní, na kterém komponenty komunikují. Většina programovacích jazyků nemá dobré mechanismy pro definování explicitního "publikovaného" rozhraní a často pouze kvalitní dokumentace a disciplína zabraňuje klientům v rozbití enkapsulace komponent, která by vedla k tzv. tight-coupling, tedy těsné vazbě mezi komponentami. Použití služeb usnadňuje prevenci těsné vazby při použití explicitních volání (remote call) mechanismů.
Nevýhodou tohoto přístupu je cena za volání mezi procesy, která je vyšší než cena volání uvnitř procesu, a velká členitost API, které pak mohou být obtížněji využitelné.

Při dělení velké aplikace je častým postupem technologické dělení, které vede k týmům jako UI týmy, týmy na logiku na straně serveru a databázové týmy. Při použití takového rozdělení ale mohou i malé změny vést k těsné mezitýmové spolupráci. Častým důsledkem je pak implementace logiky nejen na straně serveru, ale i na dalších komponentách (neboli všude), což je příkladem Conwayova zákona. Conwayův zákon[6]:

> *Jakákoli organizace, která navrhuje systém (široce definováno), vyprodukuje návrh, jehož struktura je kopií komunikační struktury organizace.*
> — *Melwyn Conway, 1967*

V mikroslužbách se používá jiný přístup k dělení. Služby jsou organizovány podle funkcí (business capabilities), což znamená, že po technologické stránce mohou obsahovat UI, logiku i datové úložiště, případně další technologie[11]. Týmy pak musí obsahovat pracovníky přes všechna tato odvětví.

V mikroslužbové architektuře je tendence uhýbat od standardního modelu, kdy je aplikace vyvinuta, předána do správy, a tím s ní vývojáři ztratí kontakt[12]. Preferovaný postup je takový, že tým má aplikaci (službu)

ve správě po celý její životní cyklus (příkladem může být iniciativa Amazonu "Build it / run it")[6]. Jedná se tedy o jakýsi kontinuální vztah s aplikací. Je tedy potřeba při vývoji aplikace myslet i na snadnou údržbu a ladění (debugging) v případě chyb, vývojáři na tom při tomto postupu mají vlastní zájem, protože budou následně i aplikaci udržovat. Budoucí údržba navíc vývojáře nutí k produkování kvalitnějšího kódu. Tento postup mikroslužbové architektury vede k decentralizaci správy.

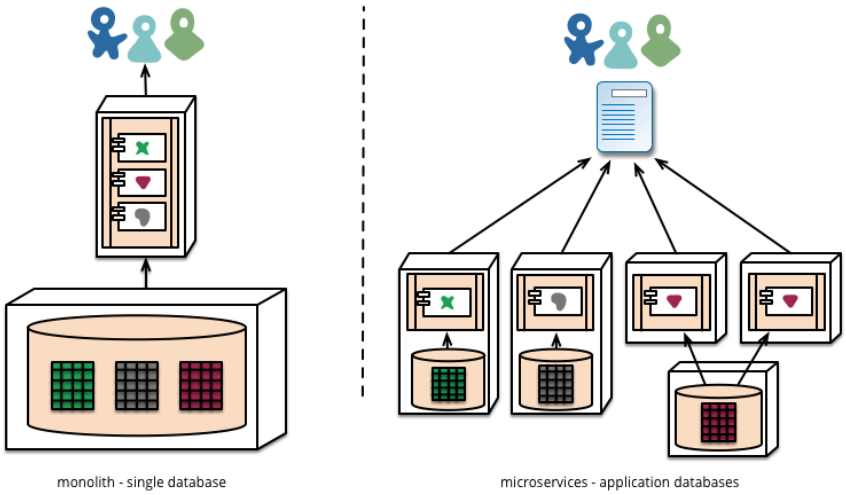
V mikroslužbách se používá ke komunikaci mezi jednotlivými službami přístupu tzv. 'smart endpoints and dumb pipes'[6]. Protože aplikace složené z mikroslužeb mají za cíl být velmi volně svázané (low-coupling)[13], jednotlivé služby fungují často na principu přijmout požadavek, zpracování logikou, odeslat odpověď. Nejčastěji používanými protokoly jsou HTTP požadavek/odpověď s danými API a lehké (lightweight) messaging protokoly[14], většinou se jedná o binární protokoly jako například protobuf od Google. Častým ulehčením síti bývá použití cache pro často používaná data, které zároveň zrychluje práci jednotlivých služeb. Infrastruktura používaná pro messaging protokoly bývá často velmi jednoduchá, až hloupá, používají se jednoduché implementace message queueing jako například RabbitMQ.

Časté využití v mikroslužbách nachází open source. To je dáno tím, že každá služba může být vytvořena použitím technologie, která je pro daný cíl nejlepší. Mnoho věcí je generických nebo se opakuje, k tomu vznikají opensourcové knihovny, které se pak využívají pro dané mikroslužby, a mohou je využívat i jiní. Díky rozšíření služby Git a GitHubu je nyní sdílení a verzování jednotlivých opensourcových knihoven velmi snadné.

Decentralizace správy dat probíhá mnoha způsoby v mikroslužbové architektuře. Každá služba, pokud to vyžaduje, má z pravidla své vlastní datové úložiště, jak je popsáno na obrázku 2.2. Typy těchto datových úložišť nemusí být pro všechny služby stejné (přístup tzv. Polyglot Persistence), opět se zde používá přístupu nejlepší možné volby technologie.

Decentralizovaná správa dat má vliv na způsob aktualizací. Běžným způsobem řešení aktualizací je použití databázových transakcí. Toto řešení pomáhá ke konzistenci dat, ale vytváří dočasné vazby (temporal coupling), proto je v mikroslužbové architektuře důraz na beztransakční koordinaci mezi službami[15] a problémy s konzistencí dat se kompenzují operačně. Takto řešit konzistenci může být často náročné, ale je to cena za rychlejší řešení požadavků. Vyplatí se to případech, kdy cena opravení chyb je menší, než cena ztrát businessu v případě konzistentnějšího řešení.

V mikroslužbové architektuře je třeba při návrhu služeb dbát na možnost



Obrázek 2.2: Rozdíl mezi centralizovaným a distribuovaným systémem[6]

selhání té, či oné služby a zohlednit to v reakci služby na negativní nebo žádnou odpověď. Týmy spravující mikroslužby by dále měly zohledňovat, jak selhání jejich služby může ovlivnit celkovou zkušenost uživatele se systémem. Je proto třeba testovat i tato selhání a reakce systému, případně i uživatelů na ně. Dále je vhodné nastavit komplexní monitorovací sestavu, který v reálném čase zachytí možné nastávající nebo nastalé problémy. Takový monitoring by měl kontrolovat architekturální problémy (například počet připojení k databázi) i businessové metriky (například počet objednávek za hodinu). Dobře nastavený monitoring může včas poskytnout informaci a vývojové týmy mohou situaci vyřešit dříve, než nastane škoda.

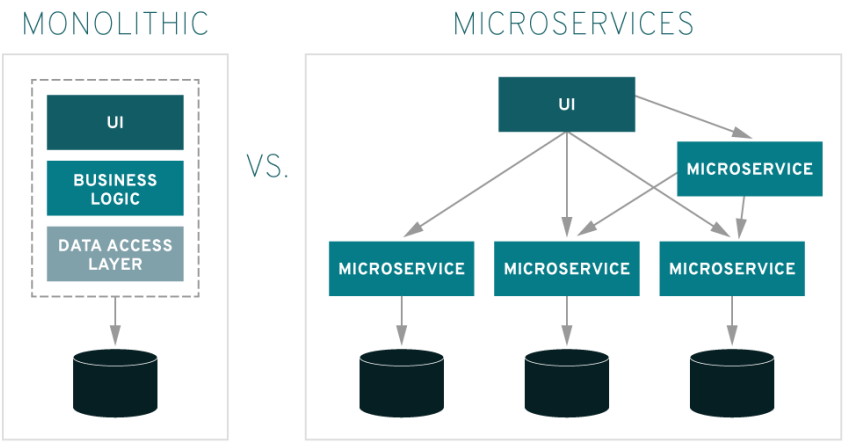
Rozkládání aplikace na komponenty (tedy služby), umožňuje vývojářům rychlejší, flexibilnější a lépe kontrolované změny v aplikaci[16]. Lepší kontrola změn v aplikaci neznamená v tomto případě zpomalení změn, ale naopak je zde možné provádět změny častěji.
Proces rozdělení aplikace na komponenty není jednoduchý. Je třeba vymyslet jak aplikaci rozdělit[7]. Klíčové parametry bývají nezávislá výměna a aktualizovatelnost. Dále části systému, které se mění méně často, by zpravidla neměly být součástí služeb, které prochází aktuálně vývojem a mají velmi časté změny. V případě, že jsou 2 služby často měněny spolu, je vhodné zvážit jejich sjednocení do jedné.

Využití mikroslužbové architektury není vždy zárukou kvalitní aplikace. Je třeba vždy před návrhem zvážit, případně i využít modelovací nástroje (například rozšíření Palladio do IDE Eclipse) pro ověření, která architektura je pro daný systém přínosnější [1].
Dále hodně záleží na tom, jak systém jako takový lze rozdělit do komponent. Často je velmi náročné určit, kde mají mít jednotlivé komponenty hranice[16].

Pokud jsou komponenty špatně určené a nekomunikují spolu přirozeně, pak se jen komplexita přesouvá z vnitřku služby do komunikace mezi službami[17]. Častým využitím mikroslužbové architektury jsou velké webové portály jako Amazon nebo Netflix [11].

2.1.2 Porovnání s monolitickou architekturou

Monolitická aplikace je na rozdíl od mikroslužbové postavena jako jediná jednotka. Enterprise monolitické aplikace jsou zpravidla složeny ze 3 částí, jak je vidět na obrázku 2.3, aplikace na straně klienta (client-side), aplikace na straně serveru (server-side) a databáze. Aplikace na straně serveru přijímá HTTP požadavky (requesty) a poté provádí logiku aplikace, čte a mění data v databázi a vytváří odpovědi HTML/JSON/XML, které odesílá prohlížeči k zobrazení, případně komunikuje s dalšími systémy[18]. Serverová aplikace je tzv. monolitní, jakákoliv změna v aplikaci bude nutně vést k nasazení nové verze na straně serveru. Veškerá logika spoléhá na jeden proces. Monolitickou aplikaci lze horizontálně škálovat pomocí spuštění více instancí za loadbalancerem.
Celý vývoj na monolitické aplikaci musí být synchronizován – malé změny vyžadují nasazení nové verze celé aplikace[18], což je náročný proces, který vede k delším vývojovým cyklům, v porovnání s mikroslužbami, kde lze nasazovat služby v zásadě bez ohledu na ostatní. V průběhu času je obtížné udržet modulární strukturu, která je u mikroslužeb dána už rozdělením na jednotlivé služby, a škálování vyžaduje škálování celé aplikace a ne jen potřebných částí, k čemuž je potřeba také více zdrojů.



Obrázek 2.3: Schéma monolitické a mikroslužbové architektury[3]

Výhodou monolitické aplikace může být například snazší nasazení, jelikož je většinou potřeba jen nasadit WAR soubor (nebo složku souborů) na správný runtime. U mikroslužbových aplikací je infrastruktura a s ní spojený nasazovací proces složitější[9].

Vývoj na monolitické aplikaci může být obtížný pro nové vývojáře, kterým trvá déle se zapojit a vyznat se v projektu[18], který má mnohem větší objem kódu (a celkově informací o projektu), než pokud se připojuje do týmu vyvíjejícím mikroslužbu, která zpravidla tolik kódu obsahovat nebude.
Dále monolitické aplikace vyžadují dlouhodobější závazek zvolené technologii[18], protože změna by vyžadovala přepracování celé aplikace a mnoho času. Mikroslužbová architektura tento problém řeší, změna technologie v rámci služby bývá jednodušší, jelikože se jedná vždy pouze o část celého systému[9].

Mezikrokem mezi monolitickou a mikroslužbovou architekturou je SOA (Service Oriented Architecture). Tato architektura se skládá z diskrétních, znovupoužitelných služeb, které spolu vzájemně komunikují přes ESB (Enterprise Service Bus). V této architektuře se každá služba, která je organizovaná okolo určitého business procesu, drží komunikačního protokolu (SOAP, ActiveMQ), přes který komunikuje s ostatními službami v rámci ESB. Tyto služby dohromady, společně s ESB, dávají celou aplikaci. Tato architektura umožňuje vývoj jednotlivých služeb nezávisle na sobě, ale obsahuje "single point of failure" v ESB, který zároveň může být i úzkým hrdlem aplikace.
Na rozdíl od zmíněných 2 aplikací, v mikroslužbové architektuře spolu služby komunikují, jak již bylo zmíněno v předchozí kapitole 2.1.1, napřímo pomocí API (standardně se tedy jedná o bezestavovou (stateless) komunikaci).

2.2 Agilní vývoj

Termín agilní vývoj pod sebou skrývá různé přístupy k vývoji softwaru. Tyto přístupy probíhají v cyklech, během nichž se vyvíjí jak požadavky na software, tak řešení[19]. Zpravidla součástí vývojových cyklů bývá i komunikace se zákazníkem. Zahrnuje adaptivní plánování, evoluční vývoj, rychlé dodání a kontinuální vylepšování[20].
Pojem agilní vývoj vznikl v roce 2001, kdy světoví odborníci na vývoj software, experti v metodikách extrémního programování, Scrum, DSDM a dalších, vymysleli a sepsali tzv. Manifest Agilního vývoje[21], ve kterém jsou hlavní body agilního vývoje SW.

Objevujeme lepší způsoby vývoje software tím, že jej tvoříme

a pomáháme při jeho tvorbě ostatním.
Při této práci jsme dospěli k těmto hodnotám:
***Jednotlivci a interakce** před procesy a nástroji*
***Fungující software** před vyčerpávající dokumentací*
***Spolupráce se zákazníkem** před vyjednáváním o smlouvě*
***Reagování na změny** před dodržováním plánu*
Jakkoliv jsou body napravo hodnotné, bodů nalevo si ceníme více.

Tento manifest byl postaven na 12 základních principech[21]:

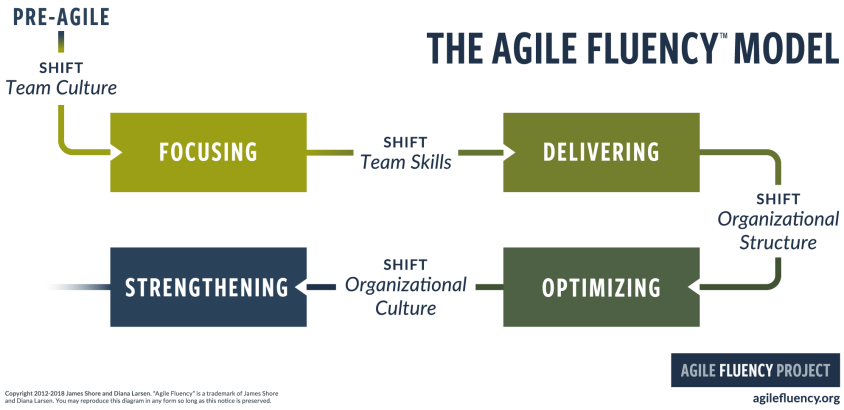
1. Naší nejvyšší prioritou je vyhovět zákazníkovi časným a průběžným dodáváním hodnotného softwaru.
2. Vítáme změny v požadavcích, a to i v pozdějších fázích vývoje. Agilní procesy podporují změny vedoucí ke zvýšení konkurenceschopnosti zákazníka.
3. Dodáváme fungující software v intervalech týdnů až měsíců, s preferencí kratší periody.
4. Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu.
5. Budujeme projekty kolem motivovaných jednotlivců. Vytváříme jim prostředí, podporujeme jejich potřeby a důvěřujeme, že odvedou dobrou práci.
6. Nejúčinnějším a nejefektivnějším způsobem sdělování informací vývojovému týmu z vnějšku i uvnitř něj je osobní konverzace.
7. Hlavním měřítkem pokroku je fungující software.
8. Agilní procesy podporují udržitelný rozvoj. Sponzoři, vývojáři i uživatelé by měli být schopni udržet stálé tempo trvale.
9. Agilitu zvyšuje neustálá pozornost věnovaná technické výjimečnosti a dobrému designu.
10. Jednoduchost – umění maximalizovat množství nevykonané práce – je klíčová.
11. Nejlepší architektury, požadavky a návrhy vzejdou ze samo-organizujících se týmů.
12. Tým se pravidelně zamýšlí nad tím, jak se stát efektivnějším, a následně koriguje a přizpůsobuje své chování a zvyklosti.

Agilní vývoj je reaktivní spíše než prediktivní. Je orientovaný spíše na lidi než na procesy[22]. V agilním vývoji slouží plán pouze jako základní čára, která pomáhá kontrolovat vývoj změn. Plánování je prováděno stejně, jako u vývoje plánovacího, ale plány jsou, na rozdíl od plánovacího, kde plán slouží k validaci čeho bylo dosaženo, konstantně upravovány, aby reflektovaly zkušenosti a znalosti získané během projektu[23]. Úspěch je poté validován podle hodnoty dodané softwarem. Na druhou stranu u plánovacího vývoje je úspěch validován podle toho, jak vývoj dodržuje plán.
Agilní vývoj je adaptivní, což znamená, že plánování je děláno jen na krátkou dobu dopředu[2]. Delší plánování obsahuje jen milníky, kterých je žádoucí dosáhnout, ale cesta k nim je stále volitelná. Proto je obtížné pro adaptivní týmy popsat, co se v budoucnu přesně udělá, jelikož s velkou pravděpodobností nastanou nějaké změny – proto se plánuje jen na velmi krátkou dobu dopředu[23].
Agilní inženýrství považuje vývoj software za primárně lidskou aktivitu[20], kde tým lidí a jejich součinnost a spolupráce je primární hnací silou úspěchu. Procesy a nástroje hrají až druhou roli. Plánovací vývoj na druhou stranu zajišťuje strukturu, která se snaží potlačit individuální variace. Takový postup je více predikovatelný a snáze se vyrovnává se ztrátou/ziskem nového člověka.
Přístup k testování je jedním z velkých rozdílů mezi tradičním vodopádovým přístupem a agilním vývojovým přístupem. Ve vodopádu (waterfall) je separátní testovací fáze, která následuje po build fázi. V agilním vývoji se testování provádí ve stejné iteraci jako programování. Což nás vede k dalšímu rozdílu, celkově se v agilním vývoji projde v každé iteraci všemi fázemi vývoje najednou (pokud je to možné), kdežto ve vodopádovém vývoji další fáze nastává až po ukončení fáze předchozí.
Dokumentace hraje v agilním vývoji až vedlejší roli. Dokumentace by podle Scotta Ambera měla být "Just Barely Good Enough" nebo "a little bit less than just enough" podle Highsmithe [2], podle něj by příliš detailní dokumentace znamenala ztrátu času a vývojáři zpravidla dokumentaci moc nevěří, protože je pravidelně pozadu oproti kódu. Příliš málo dokumentace by ale znamenalo problém pro údržbu, komunikaci a sdílení vědomostí. Alistair Cockburn, jeden ze signatářů a tvůrců Agile Manifestu, napsal metodu zvanou 'Crystal Clear', ve které přirovnává vývoj software k sérii kooperativních her a přidává, že dokumentace by měla být jen tak dobrá, aby pomohla vyhrát nadcházející (další) hru.

> *Crystal considers development a series of co-operative games, and intends that the documentation is enough to help the next win at the next game. The work products for Crystal include use cases, risk list, iteration plan, core domain models, and design notes to inform on choices...however there are no templates for these documents and descriptions are necessarily vague, but the objective is clear, just enough documentation for the next game. I always tend to characterize this to my team as: what would you want to know if*

you joined the team tomorrow.
—-Alistair Cockburn[24]

Velkou motivací v agilním vývoji je podpora lidské kolaborace. Důležitá je komunikace a zpětná vazba. Cíli jsou spokojený zákazník i spokojený programátor.
Přechod pro tým na agilní vývoj není jednoduché a plynulé a proto James Shore a Diana Larsen přišli s modelem (obrázek 2.4), který přechod usnadní. Model se skládá ze 4 základních částí / zón, kterými každý tým začínající s agilním vývojem prochází, jsou to Focusing, Delivering, Optimizing a Strengthening. Každá z těchto zón přináší nějaký užitek. Focusing – soustředěné týmy produkují business hodnotu. Delivering – týmy se soustředí na dodání a frekvenci dodání na trh. Optimizing – týmy vedou jejich relevantní trh. Strengthening – týmy které posilují jejich organizaci. Detailnější popis jednotlivých částí lze vidět na obrázku 2.5.

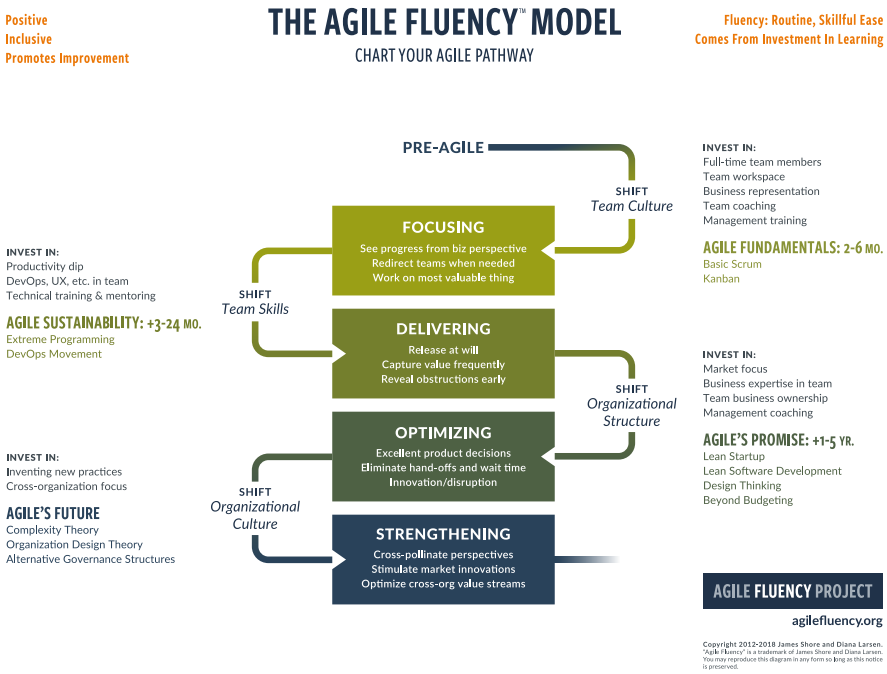


Obrázek 2.4: Schéma modelu pro usnanění přechodu na agilní vývoj – Fluency model[25]

Většina agilních metodik rozděluje vývoj produktu mezi krátké iterace, které zpravidla trvají jeden až tři týdny. Každá iterace obsahuje následující aktivity – plánování, analýza, návrh, vývoj (kódování), unit testy a akceptační testy. Na konci každé iterace je pokrok prezentován stakeholderům (zákazníkovi). Takovýto přístup minimalizuje risk a možnou chybu při oddálení se vývojem od představ zákazníka. Dále se možné změny a požadavky zákazníka promítnou rychleji a levněji do vývojového procesu. Každá iterace nemusí končit releasem do produkce, některé funkce jsou složitější a žádají si více času (a tedy i více iterací), případně zákazník nemusí být s výsledkem spokojen.

V agilním vývoji je důraz na soustředění týmu v jedné lokaci. To se dělá kvůli lepší komunikaci mezi členy týmu a celkové identitě týmu. Osobní

komunikace (face-to-face) ušetří v celku hodně času oproti jiným formám komunikace, dále se také vzniklé problémy dají diskutovat okamžitě. Každý pracovní den navíc standardně začíná tzv. Stand Upem, kde každý pracovník sděluje plány práce na další den, žádá o práci, případně o pomoc s řešením nějakého problému. Každý tým by měl obsahovat zástupce zákazníka, který jedná za zákazníka a může odpovídat na dotazy od vývojového týmu. Spe-



Obrázek 2.5: Podrobné a popsané schéma Fluency modelu[25]

ciální techniky jsou často využívané pro zlepšení kvality produktu, kódu a prostředí. Příkladem takových praktik může být například párové programování, kontinuální integrace (continuous integration, bude zmíněno v další kapitole), automatické testy, vývoj řízený testy (test driven development) nebo vývoj podle domén (domain driven design)[26]. Často se také doporučuje užívání návrhových vzorů a refactoring kódu.

Pro lepší porozumění agilnímu vývoji si představíme některé, často používané, metodiky.

2.2.1 Metodiky

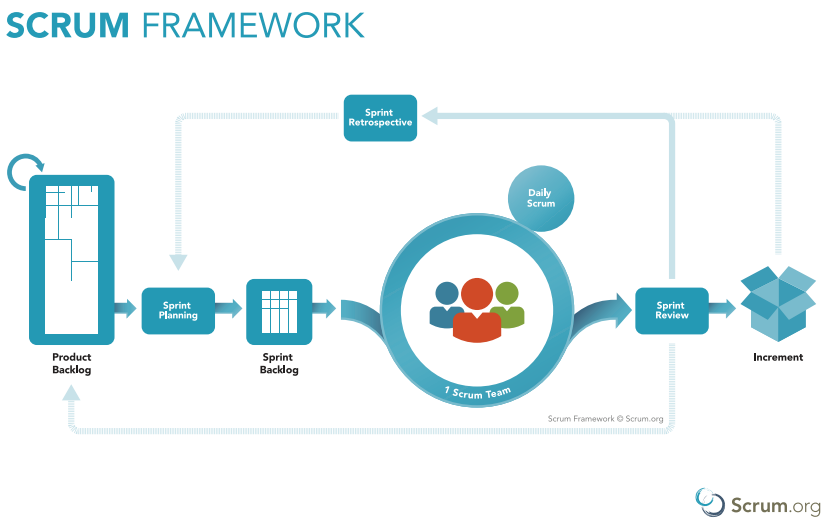
Mezi agilní vývojové metodiky se řadí například Scrum, extrémní programování, DSDM, Kanban nebo FDD. Fungování některých z nich si zde rozvedeme.

SCRUM

Scrum je framework nebo nástroj pro řízení projektu. Lidé díky němu mohou řešit komplexní adaptivní problémy a zároveň produktivně a kreativně doručovat produkt vysoké kvality. Scrum je lightweight, jednoduchý na porozumění (srozumitelný) ale obtížný na dokonalé zvládnutí. Počátky scrumu sahají až do 90. let. Scrum není samotný proces vývoje, ale spíše návod nebo rámec, v rámci kterého lze používat další procesy a techniky[27].

Scrum je založen na teorii empirických procesů, empirismus říká, že znalost přichází ze zkušenosti a rozhodování o tom, co je známo. Využívá se iterací (tzv. sprint) a inkrementální řízení pro minimalizaci rizik a získání zkušeností, což umožňuje lepší rozhodování[28]. Implementace empirického procesu má tři základní pilíře – transparentnost, kontrola a adaptace.
Transparentnost znamená, že důležité aspekty projektu musí být viditelné těm, kteří mají vliv na výsledek. Důležité je zavést společný standard, aby rozuměli všichni zúčastnění.
Kontrola stojí za časté kontrolování artefaktů a včasné odhalení odchylek od žádoucího výsledku. Frekvence kontrol musí být nastavena tak, aby byla dostatečně častá pro včasné zaznamenání problémů, ale dostatečně málo frekventovaná, aby nebránila cestě k výsledku.
Adaptace je nutná v případě, že se na základě kontroly zjistí odchylky nebo nějaké jiné problémy, které znamenají nepřijatelnosti výsledného produktu. Procesy je pak třeba adaptovat, aby se odchylka napravila, případně maximálně zmenšila.
Základními formálními činnostmi pro kontrolu a adaptaci jsou pravidelné denní schůzky (Stand Up), plánování sprintu, vyhodnocení sprintu a restrospektiva sprintu, jak je znázorněno na následujícím obrázku 2.6.

Předepsané činnosti Scrumu zajišťují pravidelnost a minimalizují potřebu přidání dalších, Scrumem nedefinovaných, schůzek. Většina těchto definovaných činností má určitý časový rámec, který by neměla přesáhnout.
Základní činností Scrumu je jedna iterace, neboli sprint[29]. Jakmile začne, nelze ho prodloužit ani zkrátit. Je souborem určitých předepsaných činností – plánování sprintu, denní schůzky, vývojové práce, vyhodnocení sprintu



Obrázek 2.6: Obecné schéma SCRUM[30]

a retrospektiva. Během sprintu je kompletně vytvořen potenciální přírůstek k produktu. Cíl sprintu je určen na schůzce k plánování sprintu. Během sprintu by se neměla snižovat kvalita cíle sprintu ani by se neměly provádět žádné aktivity ohrožující cíl sprintu. Délka sprintu by neměla přesáhnout jeden měsíc, zpravidla se používá rozsah jednoho až tří týdnů. Krátkodobost sprintů přináší předvídatelnosti dalšího vývoje.

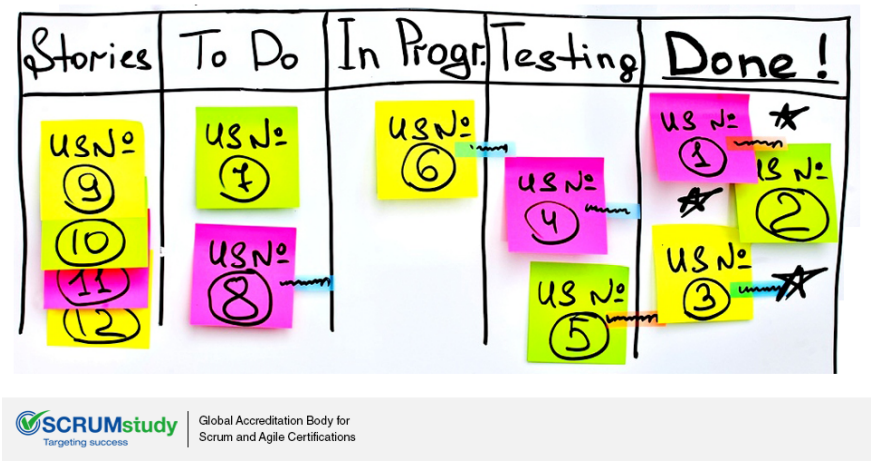
Plánovací schůzka (Sprint Planning Meeting) řeší, jaká práce bude vykonávána v příštím sprintu[28] – tj. co může být vytvořeno. Dále řeší jak bude práce provedena. Je časově omezena, pro měsíční sprint by neměla zabrat více než 8 hodin. Vstupními informacemi této schůzky jsou produktový plán (backlog), poslední přírůstek produktu, plánovaná kapacita vývojového týmu pro nadcházející sprint a výkon týmu za poslední sprint. Vlastník produktu může případně objasňovat nejasné položky v backlogu.
Denní schůzka (Stand Up nebo Daily Scrum) se koná každý den a plánuje se na ní práce na příštích 24 hodin. Délka schůzky by neměla přesáhnout 15 minut. Na schůzce by každý člen týmu měl říct co dělal předchozí den, co bude dělat den následující a případně jestli vidí nebo řeší nějaké překážky v práci.
Vyhodnocení sprintu (Sprint Review Meeting) se koná na konci sprintu. Zkoumá přírůstek k produktu za aktuální sprint, v případě potřeby adaptuje produktový backlog[28]. Všichni zúčastnění probírají možnosti jak postupovat dále na základě zjištěných skutečností. Jde o neformální schůzku, předvedení přírůstku má za cíl získat zpětnou vazbu a podpořit spolupráci. Délka schůzky pro jednoměsíční sprint by neměla přesáhnout 4 hodiny. Výsledkem schůzky by měl být revidovaný produktový backlog.
Retrospektiva sprintu slouží týmu ke kontrole sebe sama a naplánování zlep-

šení do příštího sprintu. Koná se před plánovací schůzkou – může jí předcházet. Pro měsíční sprint by neměla délkou přesáhnout 3 hodiny. Kontroluje se jak poslední sprint probíhal v ohledu na lidi, vztahy, procesy a použité nástroje.

Artefakty Scrumu reprezentují práci a její hodnoty. Jsou navržené pro maximalizaci transparentnosti klíčových informací. Mezi artefakty Scrumu patří produktový backlog, backlog sprintu a přírůstek.
Produktový backlog je prioritizovaný seznam všeho, co je třeba udělat[28]. Jedná se o jediný zdroj požadavků na změny. Za jeho obsah, dostupnost i prioritizaci je zodpovědný vlastník produktu. Je doplňován v průběhu vývoje, na začátku vývoje obsahuje jen známé a dobře definované požadavky.
Backlog sprintu je množina úkolů vybraných z produktového backlogu pro sprint. Jedná se o jakousi prognózu vývojového týmu, co za funkcionalitu bude dodáno za sprint. Mělo by se jednat o natolik podrobný seznam, že změny v obsahu a statusu jednotlivých úkolů by měly být patrné na denních schůzkách[28]. Je ve výlučné správě vývojového týmu.
Přírůstek je suma všech dokončených úkolů z produktového backlogu za průběh aktuálního sprintu. Dokončené úkoly musí splňovat podmínky určené pro stav “hotovo” (done) a také splňovat podmínky použitelnosti.

Tým ve Scrumu se skládá z vývojového týmu, vlastníka produktu a Scrum mastera. Jednotlivé týmy ve Scrumu jsou sebe-organizující a zpravidla i multifunkční, volí si tedy, jakou práci provedou a mají všechny schopnosti potřebné k tomu, aby ji dokončili, aniž by musely čekat na zásah nebo asistenci někoho stojícího mimo tým. Takovýto model je navržen pro optimalizaci flexibility, produktivity a tvořivosti.
Vlastník produktu je zodpovědný za maximalizaci hodnoty produktu a práce vývojového týmu[29]. Jedná se o jednu osobu. Stará se produktový backlog (formulaci položek, uspořádání, transparentnost), že vývojový tým rozumí položkám v backlogu a zajištění nejlepší kvality dodávané vývojovým týmem.
Vývojový tým je složený z vývojářů, kteří dodávají příspěvek k produktu na konci každého sprintu. Tým je multifunkční, tj. má všechny potřebné znalosti a zkušenosti pro vytvoření přírůstku k produktu, a nese zodpovědnost za výsledek jako celek. Velikost by měla být taková, aby tým zůstal flexibilní, ale aby byl také schopen dokončit práci v daném sprintu, jedná se zpravidla o velikost 3 až 9 členů[28].
Scrum master je odpovědný za znalost a dodržování pravidel Scrumu. Dohlíží na to, aby týmy pracovaly v duchu Scrumu, dodržovaly techniky a pravidla. Zaujímá roli vedoucího týmu a pomáhá týmu rozeznat, které interakce jsou přínosné a které nikoliv. Pomáhá udělat v týmu takové prostředí, aby mohl tým vytvářet maximální hodnotu. Dále se zapojuje do plánování produktu a vysvětluje a aplikuje pravidla agilního vývoje. Mimo jiné dále moderuje schůzky a pomáhá organizaci v aplikaci Scrumu.

Užitečným nástrojem Scrumu je také tzv. Scrum Board (někdy také Kanban board), kde je možné sledovat stav jednotlivých požadavků. Scrum Board může být digitální (například v software JIRA) i fyzický, jak lze vidět na následujícím obrázku 2.7.



Obrázek 2.7: Příklad tzv. Scrum Board – tabule se seznamem úkolů v jednotlivých fázích progresu[31]

XP – extrémní programování

Jedná se o agilní metodologii pro vývoj softwaru vzniklou v 90. letech. Cílem je vyprodukování softwaru větší kvality a také zkvalitnit pracovní život vývojového týmu. Vývoj probíhá v iteracích, které se většinou skládají z plánování, vývoje a komunikace a následné prezentace fungujícího softwaru a diskuze se zákazníkem[26].
Hlavní charakteristiky projektu, kde je vhodné XP použít, popsané Donem Wellsem[32] jsou dynamicky měnící se softwarové požadavky, projekt obsahuje rizika spojená s novou technologií a má striktní časový rámec, vývojový tým je malý a na jednom místě, použitá technologie umožňuje automatizované testy.
Extrémní programování má velmi specifická pravidla, která je třeba dodržovat. Kvůli této specificitě je často aplikována jen část XP[32].

V extrémním programování podporuje a je v něm dáván důraz zejména na jednoduchost, komunikaci, zpětnou vazbu, odvahu a respekt[32].
Komunikace mezi členy vývojového týmu je velmi důležitá. Klade se důraz na komunikaci tváří v tvář, tj. vývojový tým se musí často potkávat, doporučovanou pomůckou je i tabule. Dalším důležitým faktorem je sdílení vědomostí mezi členy týmu.

Jednoduchost je pravidlo, kterého je třeba se držet v návrhu i ve průběhu vývoje. Cílem je vyhnout se plýtvání zdroji na obtížné a složité úkony a naopak udržovat návrh a kód co nejjednodušší a nejlépe pochopitelný. Další důležitou věcí je práce pouze na známých požadavcích beze snahy připravovat si návrh nebo kód na budoucnost, která může vypadat úplně jinak.
Pravidelnou a častou zpětnou vazbou (feedback) mohou týmy snáze identifikovat a řešit problémy, případně nalézt možnosti zlepšení. Feedback dále pomáhá s udržováním jednoduchého návrhu a celého produktu produkovaného vývojovým týmem a tím zjistit odchylky od přání zákazníka dříve.
Odvaha, jak ji definoval Kent Beck[32], je efektivní akce tváří v tvář strachu. Tato definice preferuje takovou akci, která nebude mít špatné následky na celý tým. Je třeba odvahy přestat dělat něco, co nefunguje a zkusit něco jiného. Je také třeba odvahy reagovat na zpětnou vazbu, která není příjemná, nebo zavést debatu nad organizačními problémy.
Členové týmu se musí vzájemně respektovat, aby komunikace a vzájemná zpětná vazba měly kýžený (a maximální) efekt.

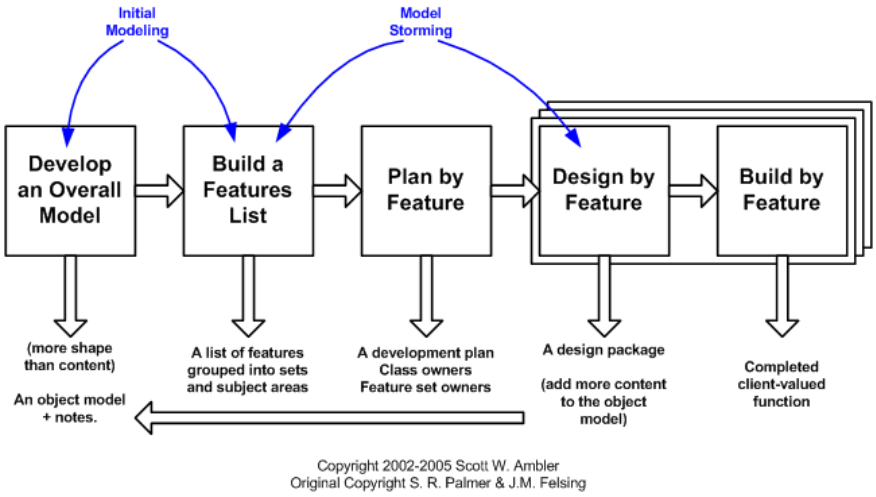
Extrémní programování se skládá z jednoduchých pravidel, která sama o sobě moc smysl nedávají, ale dohromady ano. Pod extrémní programování řadíme několik praktik, které lze implementovat izolovaně i ve vzájemné kolaboraci. Mezi tyto praktiky patří párové programování, testing, plánovací hra (planning game), jednoduchý design, malé releasy, metafory, kolektivní vlastnictví, kontinuální integrace (continuous integration, věnujeme mu zde celou kapitolu), 40-hodinový týden, zákazník na místě práce a používání kódovacích standardů (Coding Standard)[32].
Párové programování je jednou z nejznámějších technik extrémního programování. Jedná se o praktiku, kde všechny části softwaru jsou vyvíjeny dvěma programátory sedícími u stejného počítače. Hlavní myšlenkou je "více očí víc vidí" a v případě problému je možné hned získat pomocný názor někoho dalšího. Kontrola probíhá v momentě psaní kódu a ne až zpětně, což vede k lepší kvalitě kódu.

XP nespecifikuje role v týmu. Idea je, že všichni členové týmu jsou na stejné úrovni jako partneři v kolaborujícího týmu. Tým by měl fungovat na principu sebe-organizace[33]. Některé zdroje ovšem uvádí i specifické role v týmu – zákazník, vývojář, tracker a kouč. V takovém případě je zákazník zodpovědný za specifikaci funkčnosti systému, akceptačních kritérií, prostředků na vývoj a pořadí vývoje jednotlivých funkcí. Vývojář je zodpovědný za realizaci požadavků od zákazníka. Tracker je zpravidla vývojář, který část času věnuje sledování relevantních metrik, které ukazují progres týmu v dodání softwaru. Tato role nemusí být zaplněna. Role kouče je vhodná zejména pro týmy, které nemají zkušenosti s extrémním programováním nebo s ním teprve začínají. Jedná se zpravidla o externího konzultanta, který ale součástí organizace a který má zkušenosti s XP. Tento konzultant pomáhá dodržovat pravidla XP a učí ostatní členy týmu jednotlivé praktiky.

FDD – feature driven development

Feature driven development, neboli česky vývoj zaměřený na funkce, je iterativní a inkrementální vývojový proces[34]. Tato metodologie je zaměřena na zákazníka s cílem dodání jednotlivých malých částí SW často a spolehlivě. Počátky FDD sahají do konce devadesátých let dvacátého století. Cílem krátkých iterací s prezentací zákazníkovi je rychlá zpětná vazba a levnější oprava případných nedostatků.
FDD je podobné Scrumu s tím rozdílem, že zatímco ve Scrumu je důraz na dodání zákazníkovi, FDD je více zaměřený na jednotlivé funkcionality. Dále FDD dává, na rozdíl od ostatních agilních metodik, větší důraz na dokumentaci[34], na kterou je během feature driven development spoléháno jako ve Scrumu na denní schůzky.

Tato metodika je zpravidla používána pro větší projekty. Typicky se skládá z 5 základních aktivit[34], které jsou zobrazené na obrázku 2.8 – vývoj celkového modelu, vytvoření seznamu funkcí, plánování podle jednotlivých funkcí, návrh jednotlivých funkcí a vytvoření jednotlivých funkcí. Zatímco celkový model systému je zpravidla vytvořen v prvních 2 zmíněných aktivitách, většina (až tři čtvrtiny) času je stráveno v posledních dvou. Aktivity týmu jsou pak organizovány podle funkcí než podle celkového plánu.
Každá funkce ve FDD je důležitá pro zákazníka a její funkčnost je dobře prezentovatelná. Tato metodika využívá krátkých, dvou týdenních cyklů.



Obrázek 2.8: Obecné schéma životního cyklu projektu využívajícího FDD[35]

Prvním krokem (a aktivitou) je sběr dat, je důležité porozumět, co má systém dělat a na jaké uživatele je zaměřen. Je nutné pochopit, co cílový uživatel bude očekávat a potřebovat. Tento krok nelze vynechat, protože

bez porozumění cíli nelze vytvořit kvalitní model.
Krok vytvoření celkového modelu navazuje na sběr dat. V tomto kroku se vytváří detailní doménové modely, které se spojí do jednoho celkového modelu, který poté funguje jako "rozvrh" projektu. Tento celkový model je poté v průběhu projektu obohacován o další detaily.
Dalším krokem je vytvoření seznamu funkcí. Jedná se o funkce důležité pro klienta. Měly by být stihnutelné v průběhu jednoho dvoutýdenního cyklu.
Plánování podle jednotlivých funkcí je analýza obtížnosti a rozdělení funkcí na jednotlivé úkoly (tasky) pro vývojáře. Tohoto kroku by se měli účastnit všichni členové týmu. Dále by se na základě složitosti funkcí mělo určit pořadí, ve kterém budou funkce implementovány. Dále by se měli identifikovat majitelé jednotlivých tříd, kteří za ně ponesou odpovědnost a v případě potřeby změn tyto změny budou schvalovat.
V průběhu návrhu jednotlivých funkcí zpravidla hlavní programátor určí majitele jednotlivých tříd, případně celé funkční týmy. Cílem této aktivity je mít technický návrh, vybrané technologie a struktury jednotlivých funkcí. Na konci tohoto kroku by celý tým měl revidovat připravený návrh.
Vytvoření jednotlivých funkcí znamená jejich celkovou implementaci podle návrhu z předchozího kroku. Dále jsou tvořené jednotkové testy a prováděna technická review.

Podrobný přehled jednotlivých kroků je na procesním diagramu v příloze C.1.

V FDD se zpravidla rozeznává 6 hlavních rolí, jedná se o projektového manažera, hlavního architekta, vývojového manažera, hlavního programátora, vlastníka třídy a doménového experta[35]. V průběhu práce často jedinec zastává více než jednu roli.

2.3 CI/CD

Zkratka CI/CD znamená Continuous Integration a Continuous Delivery a občasně i Continuous Deployment. Tento proces je zobrazen na obrázku 2.9. Je to základním pilířem agilního vývoje a DevOps procesu[36]. Jedná se o 2 navazující aktivity, které umožňují rychlé a stabilní dodání otestované aplikace zákazníkovi díky automatizaci. Základním principem je automatická a monitorovaná integrace, testování a dodání, v některých případech i nasazení, aplikace.

Obrázek 2.9: Obecné schéma sekvence kroků CI/CD[36]

Cílem tohoto procesu je snadná integrace jednotlivých částí aplikace a také urychlení sestavení a otestování aplikace[36]. Celý tento proces se časově počítá v minutách. Tento proces řeší problém mnoha různých větví, které je často vzájemně nekompatibilní.
Prostředí, ve kterém probíhají testy, by mělo být klonem produkčního prostředí, v opačném případě hrozí, že se nezdetekují problémy, které následně mohou nastat v produkčním prostředí[6].

2.3.1 Continuous Integration

Continuous integration (průběžná integrace) je vývojový proces, který vyžaduje od vývojářů integraci kódu do sdíleného repozitáře i několikrát v průběhu dne, což umožňuje vyhnout se "integračnímu peklu" [37]. Každá změna uložená do zdrojových kódů projektu (tzv. commit) je pak kontrolován pomocí automatického sestavení aplikace, což umožňuje vývojovým týmům brzkou detekci možných problémů. Pravidelnou integrací se pak případné problémy lépe izolují a odhalují[38]. Tento přístup viditelně snižuje vývojový čas a případné řešení integračních problémů. V případě využívání CI by se měl zapojit celý vývojový tým pro viditelné zlepšení. Velmi důležité je udržet celý CI build rychlý[39], ideálně v řádu minut. V opačném případě budou vývojáři odrazeni od pravidelné integrace nutností čekat desítky minut až hodiny na evaluaci jejich kódu. V případě, že automatická integrace selže, by měla náprava této chyby být nejvyšší prioritou a měla by mít přednost před přidáváním nových funkcí[38].
Continuous integration probíhá ve 3 fázích: push (integrace kódu vývojáře do sdíleného repozitáře), test (zpravidla unit a integrační testy), fix (oprava případných chyb).
Push by měl probíhat denně, zpravidla se v tomto kroku mluví o commitu do hlavní větve, kde by se měl kód pravidelně integrovat. Nekompletní změny lze integrovat za použití tzv. feature flags. Feature flags jsou v zásadě pouze if podmínky, které říkají zda pouštět nebo nepouštět nový kód.
Testování je druhým krokem CI. V případě nutnosti kompilace programu se i kompilace počítá jako test. Dále se spouští jednotkové a integrační testy,

které jsou považovány za kritické ke správnému běhu aplikace. Je nutné mít na paměti, že automatická CI pipeline by měla proběhnout co nejrychleji, proto zpravidla není vhodné pouštět kompletní sady testů. Sady testů pro automatické CI by měly být navrženy tak, aby byly schopné odhalit chyby v kódu, ale detailní testování je možné dělat až například před vydáním nové verze aplikace.
Klíčovou součástí CI je okamžitá oprava hlavní větve v případě, že dojde k selhání[37]. Celou podstatou CI je práce se stabilním základem, proto, pokud dojde k selhání, by měla mít náprava nejvyšší prioritu. Nejčastější a nejsnadnější opravou je návrat k poslednímu fungujícímu commitu. Následná oprava kódu, který chybu způsobil, by se už měla dít mimo hlavní větev.

Celý tento proces CI by měl být pro všechny transparentní a snadno dostupný[38]. Výsledek průběhu pipeline na hlavní větvi by měl být komunikovaný členům týmu, zejména v případě neúspěchu. Pro následné opravy je důležité mít přístupný log z průběhu.

2.3.2 Continuous Delivery

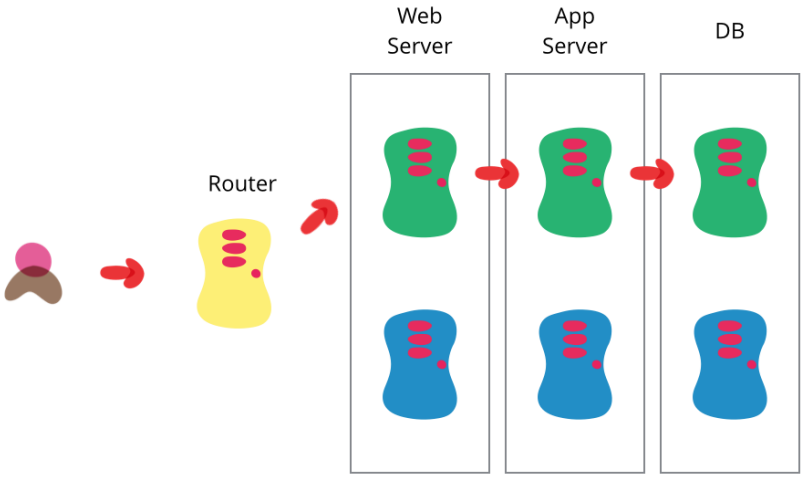
Jez Humble definoval CD jako schopnost bezpečně a rychle dostat změny všech typů, včetně nových funkcí, konfiguračních změn, oprav chyb (bug fix) a experimentů, do produkčního prostředí nebo do rukou uživatelů udržitelným způsobem[39]. Toho je možné dosáhnout tím, že zajistíme, aby byl kód pořád ve stavu, který lze nasadit, a to i v případě, že máme týmy tisíců vývojářů, kteří dělají změny na denní bázi.
Continuous delivery, neboli průběžné dodání, je proces následující po CI. Cílem CD je artefakt vyprodukovaný v CI procesu nasadit na všechna požadovaná prostředí[36]. Kritickým faktorem je, aby všechna před-produkční prostředí (tedy stage, test, development) byla co nejvíce podobná, nebo stejná jako produkční prostředí. Pro CD by produkční prostředí nemělo být ničím jiným, než dalším krokem pipeline.

Zpravidla můžeme mluvit o provádění CD pokud náš software splňuje následující podmínky: software je nasaditelný v průběhu celého životního cyklu, vývojový tým prioritizuje udržování nasaditelnosti oproti přidávání nových funkcí, po každé změně v systému může kdokoliv získat rychlou, automatickou zpětnou vazbu o produkční připravenosti systému a musí být možnost snadného nasazení libovolné verze do vybraného prostředí[40].

Jedním z principů CD je dělat časté nasazení do produkce s málo změnami[40].

Tím se snižuje riziko nasazení a zajistí, že případné chybné funkce jsou odstraněny, nebo opraveny, samostatně a správně fungující funkce systému zůstávají, na rozdíl od velkých releasů plných funkcí, kde se chyba odhaluje hůře a většinou je potřeba navrátit předchozí verzi celé aplikace a provést další případné úpravy potřebné k návratu ke stavu před zahájením releasu (tzv. rollback).
Dále je dobré udržovat vysokou úroveň automatizace. Zde se jedná o tzv. automatizace s dotekem člověka[41]. Automatizované by měly být zejména úkoly opakující se s každým vydáním, ve kterých by člověk, který je dělá manuálně stále dokola, mohl chybovat. Stále je ale často třeba lidských zásahů do procesu a zejména jeho vylepšení.

Jednou z technik, která se využívá v CD, je tzv. Blue Green deployment, obecný přehled je na obrázku 2.10. Jedná se o techniku, kde máme 2 identická produkční prostředí (modré a zelené), nebo alespoň tak identická, jak jen to jde. V jednom časovém okamžiku je produkčně dostupné jen jedno z nich (například modré), na druhém (například zelené) se připravuje nový produkční release. Ve chvíli, kdy je zelené prostředí připravené na produkci, přepne se router nebo loadbalancer před oběma těmito prostředími tak, aby nyní všechny požadavky šly na zelené prostředí a na modrém se začne připravovat nové vydání[42]. Tento přístup umožňuje velmi rychlý rollback v případě nalezené chyby.

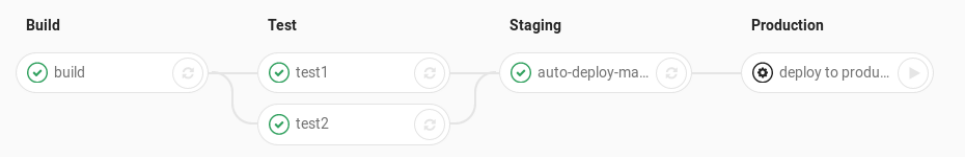


Obrázek 2.10: Příklad schématu pro Blue Green deployment, na obrázku je aktivní zelené prostředí a modré čeká na nasazení[42]

2.3.3 Nástroje na CI/CD

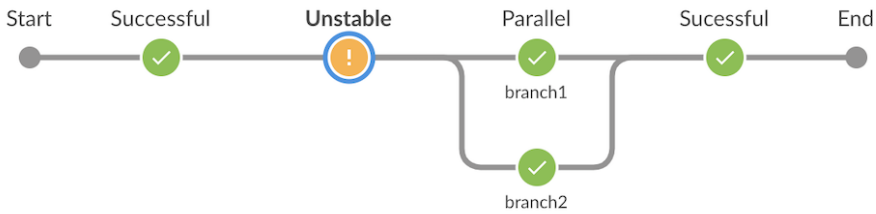
Ačkoliv lze CI/CD praktikovat i bez externích nástrojů, jedná se zpravidla o krok, který bude stát mnoho prostředků a bude trvat velmi dlouho, než se celý proces dostane na kvalitní úroveň. V současnosti existuje mnoho nástrojů, které umožňují a usnadňují využití CI/CD. V této kapitole zmíníme jen krátce několik z nejznámějších.

GitLab CI je součástí GitLabu. Využívá libovolného množství runnerů, které se periodicky dotazují na API GitLabu, zda je nová práce k dispozici[43]. Runnery mohou běžet na stejném nebo jiném serveru než GitLab. Celé CI/CD v GitLabu se skládá z pipeline. Jednotlivé pipeliny se skládají ze stage a jobs (příklad je na obrázku 2.11). Definice, co se má v každé pipeline stát, je definována v .gitlab-ci.yml souboru.



Obrázek 2.11: Příklad pipeline v GitLabu – jedná se o stage Build, Test (2 různé paralelní joby), Staging a Production[44]

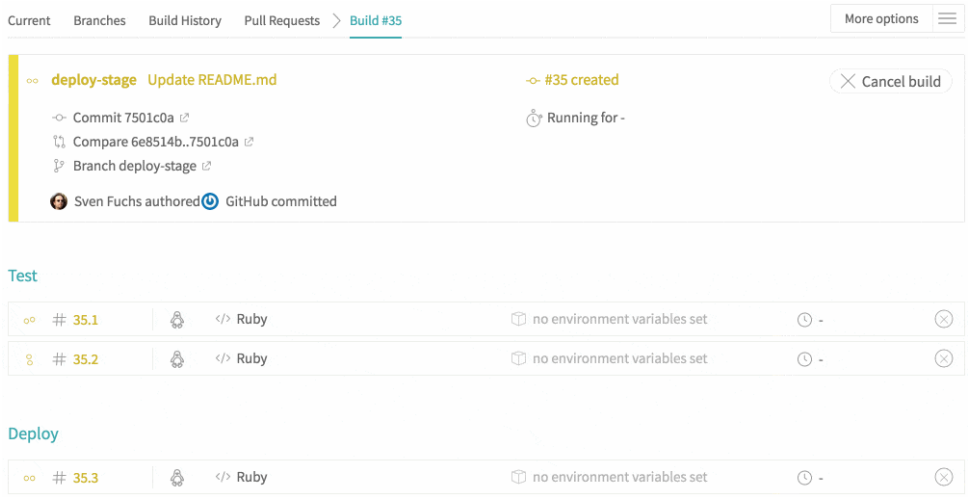
Jenkins je opensourcový server napsaný v jazyce Java. Je snadno rozšiřitelný pomocí pluginů. Architektura je master a agenti, kde master poskytuje API a webové rozhraní pro správu a koordinuje práci jednotlivých agentů[43]. Běh CI/CD se v Jenkins konfiguruje pomocí tzv. Jenkinsfile, nebo pomocí webového rozhraní, případně s využitím nějakého pluginu. Příklad pipeline je na následujícím obrázku 2.12.



Obrázek 2.12: Obecný příklad pipeline v Jenkins[45]

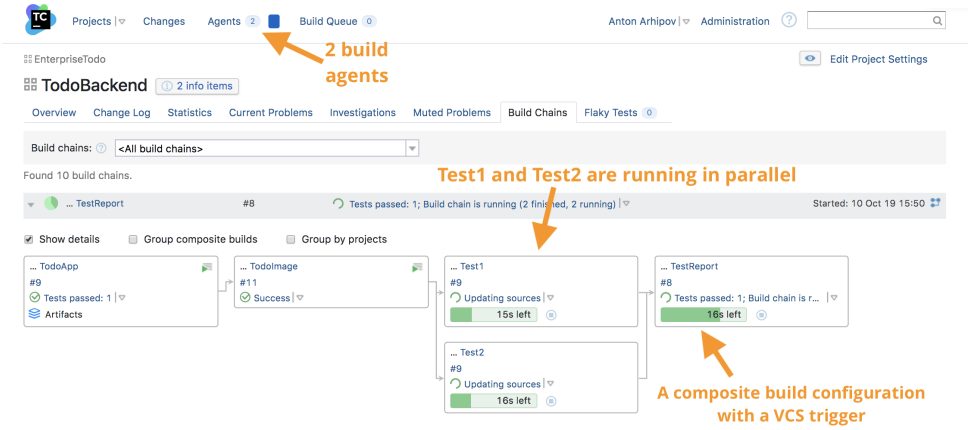
Travis CI je tzv. SaaS, tedy software jako služba. Pro každý job v CI/CD

spouští nový virtuální stroj. Pomocí Build Matrix umožňuje specifikovat jednotlivé verze závislostí a CI pak spustí job pro všechny kombinace[43] (příklad pipeline je na obrázku 2.13). Nejčastěji se využívá pro opensourcové projekty uchovávané na GitHubu. Definice jednotlivých pipeline se konfigurují v .travis.yml souboru.



Obrázek 2.13: Příklad build procesu v Travis CI[46]

TeamCity je nástroj od společnosti Jetbrains. Jako Jenkins využívá architekturu master a agentů[47]. Běh jednotlivých CI/CD se konfiguruje ve webovém rozhraní TeamCity v Build konfiguracích a jednotlivým krokům se říká Build step. Je možné ho rozšířit pomocí celé řady pluginů. Jednoduchý popis rozhraní a pipeline v TeamCity je na následujícím obrázku 2.14.



Obrázek 2.14: Příklad pipeline v TeamCity s popisem uživatelského rozhraní[48]

Artifactory je nástroj zaměřený na správu balíčků a dalších artefaktů

z build procesu. Z build procesu je získáno velké množství informací, které jsou k artefaktům uloženy. Kromě úložiště balíčků nabízí Artifactory i schopnosti CI/CD. Artifactory existuje v opensource variantě, dále nabízí licence pro servery a cloudové řešení.

2.4 Nástroje na analýzu logů

V této kapitole se budeme zabývat nástroji na agregaci a analýzu logů. Tyto nástroje umožňují rychle a snadno získat logy ze všech požadovaných služeb na jedno místo, kde s nimi můžeme dále pracovat ať už využitím možností těchto nástrojů nebo jiným způsobem.

2.4.1 Úvod do problematiky

Snad každý programátor někdy zažil ten hořký scénář, kdy musel hledat nějakou informaci v logu, který byl příliš dlouhý a příliš špatně čitelný pro lidské oko. V takovémto scénáři lze trávit hodiny a hodiny času hledáním potřebné informace – což je nejlepší cesta k bolení hlavy[49].
Možnou cestou je – a mělo by být – logování jen potřebných věcí. V produkčním prostředí by ladící zprávy měly být vypnuty, protože na produkci je to pouze šum, který skrývá potřebné informace a zabírá místo na disku. Dále by se měl zavést standard pro formát logované události. Tato cesta by měla zpřehlednit logy a umožnit snazší vyhledávání informací. Ani to ale bohužel nestačí. Zalogovaných informací může být stále příliš na vyhledání jedné konkrétní. Naopak pokud jsme logy moc "ořezali", můžou nám pak chybět životné důležité informace pro odhalení jiných chyb.
Samozřejmě lze používat nástroje zabudované v systému, například *"grep"* nebo *"awk"*, na parsování logů a dohledání dané informace. Dohledání informace tímto způsobem možné je, je to však často hodně práce. Tím se dostáváme k potřebě nástroje, který umožní snadno v logu vyhledávat.

V případě, že máme mikroslužbovou architekturu aplikace, tedy naše aplikace se skládá z více menších aplikací, nám vzniká navíc také velké množství různých log souborů. Dohledávání informací se tak stává ještě těžším, protože musíme hledat na více místech. Ať už se bude jednat o několik málo nebo naopak hodně aplikací, máme problém, který nám prodlužuje čas (a tím pádem i zvětšuje náklady) na nalezení dané informace[50]. Pro efektivnější práci s logy z více aplikací bychom potřebovali logy dostat na

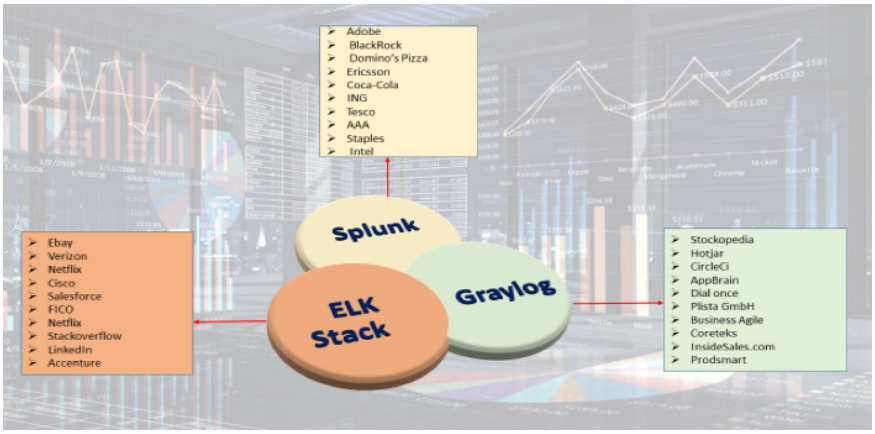
jedno místo, kde v nich můžeme snáze hledat.
Možností stále zůstává, že si manuálně stáhneme logy do vlastního počítače a pustíme nad nimi vyhledávací skripty. Toto řešení je však zdlouhavé a vyžaduje nové stažení logů při každém novém problému (kvůli aktualizaci – aktuálnosti logů).
Řešením není například nucené logování všech aplikací do jednoho souboru. To by jednak vedlo k obrovskému a nečitelnému logu, zároveň by to ale mohlo, v případě, že máme aplikace distribuované na více serverů, vést k zahlcení sítě logy, a tím pádem k omezené funkčnosti aplikace jako celku
Lepším řešením je najít způsob, jak efektivně dostat všechny logy na jedno organizované místo. Tím se dostáváme k termínu "agregace logů".

Agregace logů je proces sběru jednotlivých log souborů za účelem organizace dat pro snadné vyhledávání. Naštěstí tento proces nemusíme sami vymýšlet, jelikož už existuje řada fungujících řešení. Mezi nejznámější patří například Graylog, Splunk nebo ELK (zkratka pro Elasticsearch-Logstash-Kibana stack)[51]. Tato řešení se starají o celý proces – sběr dat od aplikací, parsování logů a poté i o vyhledávání. Vyhledávací dotazy je samozřejmě nutné napsat, ale tyto agregační nástroje pomáhají a velmi urychlují vyhledávání. Logy jsou jimi naparsovány na data, ve kterých lze vyhledávat nejen pomocí textového vyhledávání tzv. textsearch, ale také více sofistikovanými dotazy. Je tedy možné vyhledávat i podle povahy dotazovaných dat.

Agregátory logů umožňují monitorování logů v reálném čase (obdoba *"tail -f"*, ale nemusíme být připojení na daný server). To nám umožňuje sledovat konkrétní reakce aplikací na nějaké dotazy[52]. Navíc, oproti *tail*, agregátory i v reálném čase umožňují filtrování ve zprávách a dotazy.

2.4.2 Porovnání Elastic Stack, Graylog a Splunk

Pro porovnání různých možností byly vybrány následující 3 nástroje: Elastic Stack (Elastic-Logstash-Kibana stack (ELK) + Beats => Elastic stack[53]), Graylog a Splunk. Nejdříve uvedeme základní informace ohledně těchto nástrojů, poté je budeme vzájemně porovnávat. Na následujícím obrázku 2.15 jsou uvedeny příklady velkých společností využívající tyto nástroje.



Obrázek 2.15: Příklady společností využívající Graylog, Elastic Stack nebo Splunk[55]

Elastic Stack

ELK je akronym pro 3 opensourcové projekty – Elasticsearch, Logstash a Kibana. Později byla k této trojici dodána ještě aplikace Beats. Z toho vznikl nový název – The Elastic Stack. Elasticsearch je engine na vyhledávání a analytiku. Logstash pracuje na straně serveru, sbírá data z více zdrojů, zpracuje je (transformuje) a poté pošle do úložiště – např. Elasticsearch. Kibana slouží k vizualizaci dat z Elasticsearch. Beats se používá k odesílání dat ze serveru, hlavně v reálném čase[53].
Součástí Elastic Stacku také může být tzv. X-Pack. Jedná se o soubor funkcí, které rozšiřují funkčnosti celého stacku, jde například o funkce pro monitoring, alerting, reporting nebo bezpečnost[54]. Některé z těchto funkcí lze využívat zdarma, jiné jsou placené.

Elasticsearch. Elasticsearch je distribuovaný engine sloužící k vyhledávání a analytice nad různými typy dat, textových a numerických, geospaciálních, strukturovaných i nestrukturovaných. Poprvé byl vydán v roce 2010. Hlavními přednostmi jsou distribuovatelnost, rychlost a škálovatelnost, dále jednoduché REST API[56]. Elasticsearch je už povahově připravený k běhu ve více instancích díky tomu, že jsou dokumenty distribuovány podle povahy v různých kontejnerech (v Elasticsearch se jim říká “shards”), které se duplikují pro případ chyby hardware. To znamená, že Elasticsearch může běžet na mnoho (například stovkách) nodech a starat se o až petabyty dat. Obsahuje hodně funkcí, které zlepšují efektivitu v ukládání dat i při hledání. Elasticsearch využívá Apache Lucene vyhledávací jazyk.
Je centrální komponentou Elastic Stacku, jelikož ukládá data a vyhledává v nich.
Nejvíce se Elasticsearch využívá pro aplikační vyhledávání, webové vyhle-

dávání, zpracování a analýzu logů, dále různé analytiky (business nebo bezpečnostní)[56].

Elasticsearch funguje tak, že nejprve "raw" data z různých zdrojů vtečou do Elasticsearch, poté následuje proces, který se nazývá "data ingestion", tedy polknutí dat. Během tohoto procesu se data parsují, normalizují a obohacují, než jsou zaindexována v Elasticsearch. Jakmile jsou data zaindexována, může uživatel v datech vyhledávat.

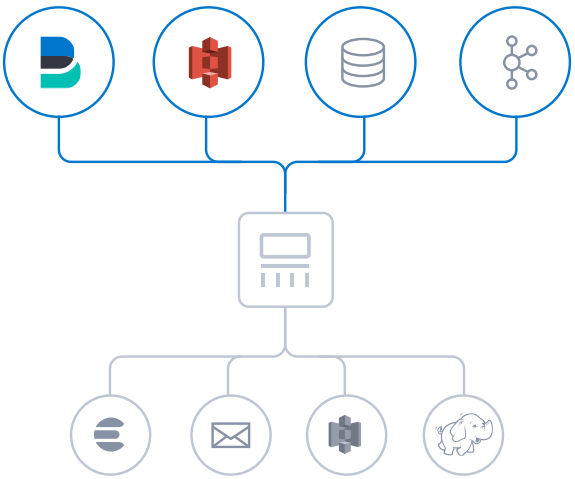
Index je v Elasticsearch kolekce dokumentů, které spolu nějakým způsobem souvisí. Elasticsearch ukládá data jako JSON dokumenty, každému takovému dokumentu pak odpovídá množina klíčů s jejich odpovídajícími hodnotami (ať už se jedná o stringy – texty, čísla, logické hodnoty, nebo například geolokační hodnoty a další).

Elasticsearch používá datovou strukturu "invertovaný index", která umožňuje velmi rychlé fulltextové vyhledávání. Invertovaný index obsahuje každé unikátní slovo, které se vyskytuje v jakémkoliv dokumentu a identifikuje dokumenty podle slov, která obsahují. Během indexace Elasticsearch ukládá dokumenty a vytváří invertované indexy, aby byla data prohledatelná v reálném čase[56].

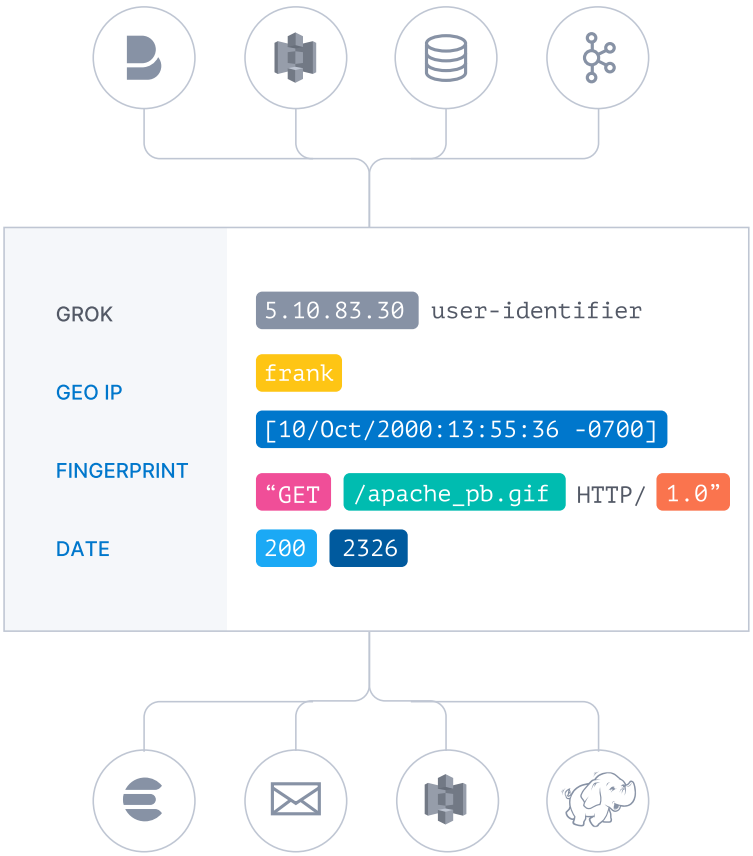
Elasticsearch je opensourcový projekt spravovaný společností Elastic, lze do něj komunitně přispívat na GitHubu a je zdarma. Společnost Elastic ale nabízí i placené varianty, ve kterých jsou zpravidla nějaké vylepšení, například možnost strojového učení nebo podpora přímo od Elastic. Oficiálně podporovanými jazyky jsou Java, JavaScript, C# (.NET), Go, PHP, Perl, Python a Ruby. Lze ho spustit na vlastním HW nebo v cloudu.

Logstash. Logstash je opensource pipeline, která pracuje na straně serveru (tzv. server-side) a zpracovává přicházející data z různých zdrojů a poté je předává do Elasticsearch (nebo do nějakého jiného úložiště). Dynamicky konzumuje, transformuje a předává data nezávisle na jejich obsahu, velikosti nebo formátu. Umí vytvářet strukturu v nestrukturovaných datech, případně z IP adres vytvořit geolokační data a také anonymizovat data[57]. Jedná se vlastně o přípravu dat pro Elasticsearch. Obecný diagram umístění Logstash v systémů je zobrazen na obrázku 2.16.

Data, která prochází přes Logstash, prochází filtry, které identifikují prvky, vytvoří strukturu a převedou data do běžného formátu, který je výhodnější pro analýzu. Logstash obsahuje takovýchto filtrů bohatou knihovnu. Příklad je zobrazen na následujícím obrázku 2.17.



Obrázek 2.16: Obecný diagram funkce Logstash (uprostřed) – sbírá data z Beats apod., agreguje je a následně tranferuje do dalších nástrojů[57]

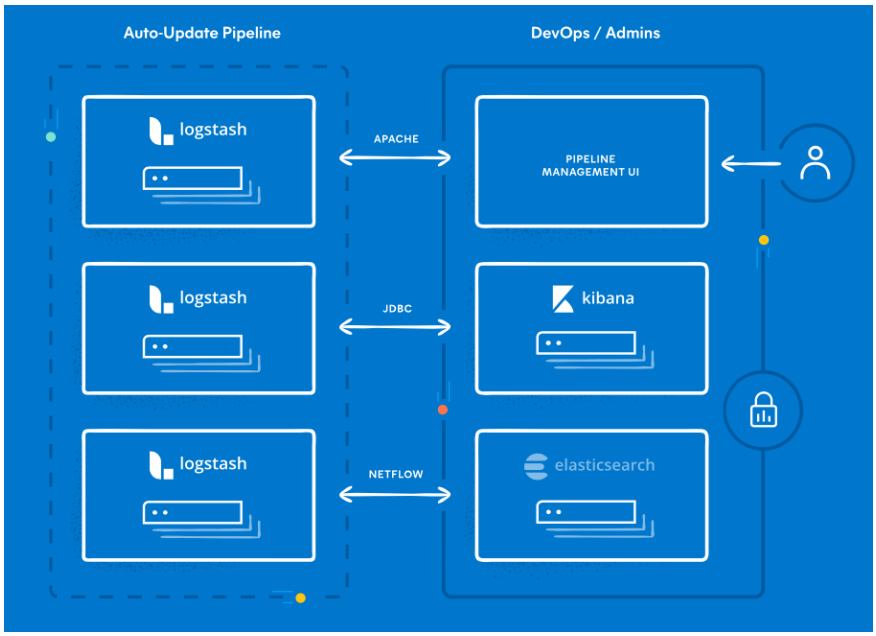


Obrázek 2.17: Příklad filtrování, agregace, obohacení a úpravy dat Logstashem[57]

Standardně, v rámci Elastic Stacku, jsou data předávána do Elasticsearch, ale je možné data předávat i jinam, podle požadavku uživatele, například do syslogu nebo Slacku.

Logstash využívá tzv. persistetní frontu (persistent queue) pro zaručení doručení eventů i v případě, že by některý z Logstash nodů selhal. Eventy, které nejsou úspěšně zpracovány, jsou hromaděny v "dead letter queue" v níž jsou dále zkoumány a případně poslány k znovuzpracování[57]. Logstash zároveň zvládá i nápory provozu (trafficu) bez potřeby použití externí "message queuing" vrstvy.

S využitím Kibany a rozšíření pro monitorování pipeline lze snadno monitorovat traffic protékající přes jednotlivé Logstash nody, lze získávat informace o úzkých hrdlech a podobně. Následující diagram 2.18 popisuje propojení s ostatními aplikacemi Elastic Stacku.



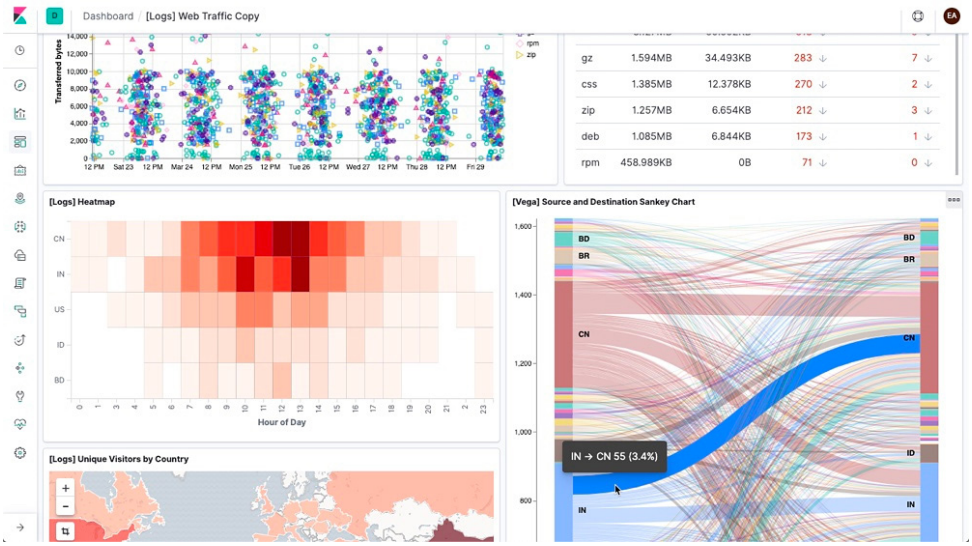
Obrázek 2.18: Obecné schéma umístění Logstash v log systému[57]

Logstash je také opensourcový projekt spravovaný společností Elastic, lze do něj přispívat na GitHubu a je také zdarma. Aplikaci lze spustit na vlastním HW nebo v cloudu.

Kibana. Kibana je opensource frontend, který jako backend používá Elasticsearch. Používá se k hledání, prohlížení a vizualizaci dat z Elasticsearch

a další analýze těchto dat pomocí různých grafů, map, histogramů a tabulek. Je možné si vytvořit vlastní upravené dashboardy (dashboard je kolekce vizualizací dat), které mohou nabízet pohled na více skupin dat. To se využívá zejména k prohlížení logů, monitoringu výkonu aplikací, monitoringu infrastruktury, analýze geospaciálních dat a business a bezpečnostním analytikám. Dále slouží jako uživatelské rozhraní pro monitoring a správu celého Elastic Stacku[58].
Kibana nabízí velkou škálu možných vlastních úprav a je možné si přizpůsobit uživatelské rozhraní přesně na konkrétní use case.
Důležitou funkcí pro správu uživatelů je podpora SSO (Single Sign On), podpora security realms (například LDAP) a podpora RBAC (role based access control). Další důležitou funkcí je Kibana Lens, která slouží pro snadnou manipulaci a vizualizaci dat (příklad zobrazen na obrázku 2.19), podporuje mimo jiné drag-and-drop. Celý seznam podporovaných funkcí lze nalézt na oficiálních stránkách společnosti Elastic (`www.elastic.co`).

Kibana je také opensourcový projekt spravovaný společností Elastic, lze do něj přispívat na GitHubu a je také zdarma (pod Apache 2 licencí). Za některé funkce je ale nutné si připlatit. Aplikaci lze spustit na vlastním HW nebo v cloudu.



Obrázek 2.19: Příklad dashboardu s vizualizací dat v Kibaně[58]

Beats. Beats je platforma jednoúčelových data shipperů, které posílají data z serverů (aplikací, systémů) do Logstashe (nebo případně přímo do Elasticsearch). Nacházejí se přímo na aplikačních serverech nebo v kontejnerech. Zajišťují data z jednotlivých prostředí a obohacují je o důležitá metadata, která se pak používají při zpracování dat[59].
Beaty se dělí do skupin podle toho, jaká data posílají. Základními skupinami

Beatů jsou Filetbeat, který posílá logy a další data, Metricbeat, který posílá metrická data, Packetbeat sloužící pro síťová data, Winlogbeat, který posílá data z Windows eventů, Heartbeat, který zajišťuje data pro monitoring uptime, Fuctionbeat zajišťuje data z cloudu a Auditbeat, který se stará o data pro audit.

Beats jako opensourcový projekt opět spoléhají na komunitu při tvorbě nových funkcí, případně zcela nových Beatů[60]. Výše vyjmenované Beaty jsou zaštítěny společností Elastic, oficiální seznam komunitních Beatů ale obsahuje přes 100 položek.

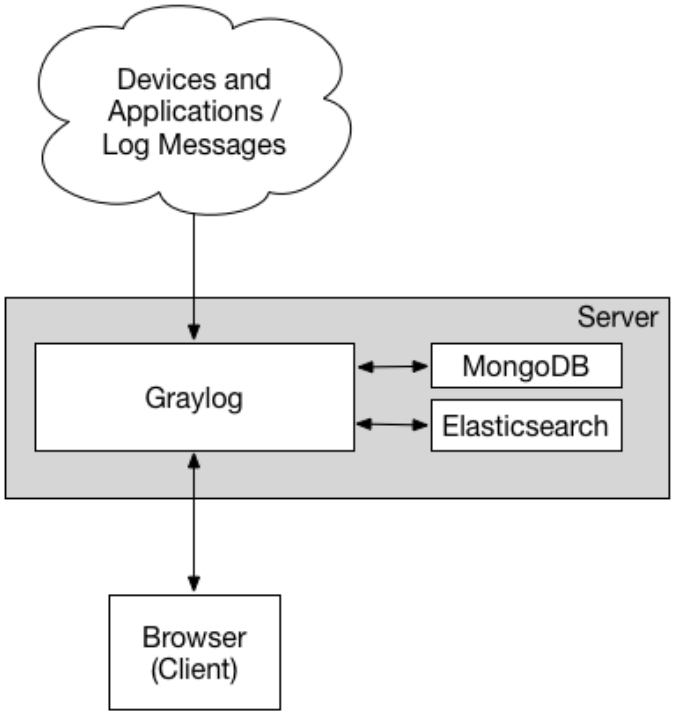
Graylog

Graylog je systém na centralizovaný log management. Přichází ve 2 variantách – opensource a enterprise. Systém se skládá ze tří vrstev a to konkrétně z Elasticsearch, který plní stejnou funkci jako v Elastic Stacku, dále z MongoDB a Graylog serveru, který slouží jako uživatelské rozhraní pro správu a vizualizaci dat[61]. Obecná architektura serveru je popsaná na následujícím obrázku 2.20. Má schopnost pojmout a zpracovat ohromné množství dat, ale Enterprise varianta se od hranice 5 GB za den se z něho stává placený software. Vyhledávání v logách přes uživatelské rozhraní je pomocí Lucene query.

Graylog umí zpracovávat velké množství různých typů dat, například syslog data, GELF data, JSON, Netflow, ale i data z Logstash nebo přímo z Beats. Je tedy možné ho integrovat do jakéhosi pseudo-Elastic Stacku, kde hraje roli Kibany.

Pro případ, že je potřeba sbírat logy z více zdrojů, má Graylog tzv. Graylog Sidecar[62], který se stará o centralizovanou konfiguraci a management těchto zdrojů. Sidecar kontroluje agenty a zároveň udržuje konzistentní konfiguraci na všech prostředích na různých hostech pomocí štítkovacího (tagovacího) systému. Tagy se vytvářejí ve webové konzoli pro jednotlivé typy logů, například DNS logy nebo Apache logy. Jakmile jsou tagy použité na endpoint, Sidecar začne automaticky stahovat data a přenášet je do Graylogu. Standardním typem agentu je například Filebeat, Winlogbeat nebo NXlog.

Data musí být parsována a často i obohacena, aby byla užitečná pro uživatele. K tomu u Graylogu slouží extraktory, které instruují Graylog, do jaké formy má daná data zpracovávat. Vytváření těchto extraktorů se děje pomocí



Obrázek 2.20: Obecné schéma architektury Graylogu[61]

REST API[61]. Dále se využívá streamů pro směrování zpráv do správných, zvolených kategorií v reálném čase, případně pro třídění nebo access control. Pipeliny a pipelinová pravidla se v Graylogu využívají k vyčištění dat a jejich uspořádání (případně i upravení) do žádané struktury. Lookup tabulky slouží k překládání zpráv, nebo jejich částí, na nové hodnoty – zde jako příklad můžeme uvést překlad IP adresy na geolokaci.

Vyhledávání dat v Graylogu je velmi rychlé. To je především díky integraci Elasticsearch, který je rychlý sám o sobě. Graylog je dále velmi dobrý v případech, kdy potřebujeme zjistit, co se děje v prostředí, tedy ve vykonávání velmi složitých dotazů. Možnost nastavení dashboardů v Graylogu umožňuje větší přehled. Navíc je možné si ukládat a sdílet složité dotazy.

Export dat z Graylogu je možné provádět několika způsoby. Jedním z nich je REST API, které je v Graylogu velmi jednoduché a přehledné a snadno integrovatelné s dalšími aplikacemi. Je možné si nastavit zasílání pravidelných reportů na email – to lze provést velmi snadno přímo ve webovém rozhraní. Report je vlastně takový snapshot dashboardu v konkrétním nastaveném čase. Data forwarder je součást Graylogu, díky které je snadné si nastavit více instancí Graylogu a zajistit komunikaci mezi nimi, což dělá Graylog snadno škálovatelným řešením. Dále umožňuje například geografickou separaci dat,

a tedy zamezuje zbytečnému přelévání dat na dlouhé vzdálenosti.

Graylog nabízí škálu dalších funkcí[63]. Je možné sledovat, co který uživatel v Graylogu dělal díky Audit logu. Dále je možné archivovat logy pro ušetření nákladů. Důležitou bezpečnostní funkcí je RBAC (Role based access control), která umožňuje přístup do různých pohledů nebo dashboardů podle role. Další funkcí je možnost alertingu v požadovaných případech.

MognoDB je v Graylogu používána ke skladování konfiguračních informací a metadat. Ukládají se do ní uživatelské informace a konfigurace streamů, proto je to výkonnostně nenáročná součást aplikace a lze ji nainstalovat přímo na Graylog server[64].

GELF je zkratka pro Graylog Extended Log Format. Jedná se o formát logu, který není limitován standardními logovými nedostatky, jako je omezená délka zprávy nebo absence datových typů. Díky různým knihovnám je snadno implementovatelný přímo do aplikací. Na rozdíl od standardních logů lze logy v GELF formátu komprimovat, a ušetřit tak šířku pásma sítě.

Graylog je opensourcová aplikace o kterou se stará společnost Graylog, lze do ní komunitně přispívat na GitHubu a je z velké části zdarma.

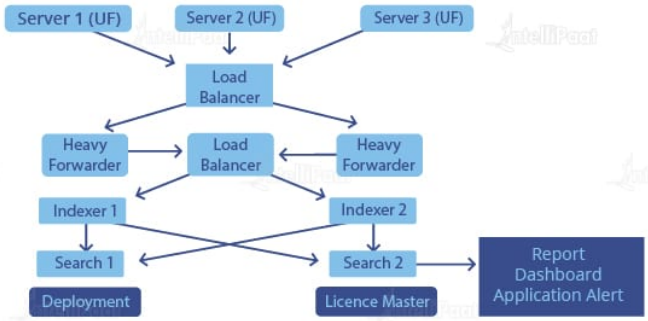
Splunk

Splunk je platforma vzniklá s cílem zorganizovaní a polidštění dat z logů. Aktuálně platforma umožňuje prohledávat, analyzovat a vizualizovat data (Big Data) z různých aplikací, je to tedy celkově spíše analytická platforma, která se nezaměřuje pouze na logy[65]. Nejedná se o opensourcový projekt, tím pádem nevidíme, jak Splunk uvnitř funguje a je to takový black-box, kterému musíme věřit. Lze ho využít k managementu aplikací, bezpečnosti, ale také k různým business nebo webovým analytikám[66].

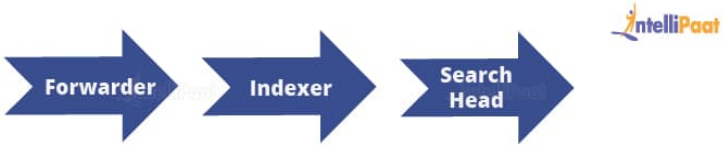
Splunk sám o sobě provádí "odchyt" dat, jejich indexaci a korelaci v reálném čase do prohledávatelného kontejneru, ze kterého poté může generovat grafy, reporty, alerty, dashboardy a další vizualizace[67]. Umožňuje tedy monitorování logů v reálném čase. Zaměřuje se hlavně na překlad dat generovaných stroji tak, aby byly informace dostupné v potřebné formě (zde je myšleno forma čitelná pro lidi). Umí rozeznávat vzory v těchto strojově generovaných datech a také z nich produkovat metriky, případně diagnostikovat problémy, dále se také umí učit nové vzory díky schopnosti strojového učení (machine

learning). Díky své podstatě podporuje širokou řadu input formátů jak logů, tak ostatních dat, například databázových.

Primární komponenty jsou Forwardery, Indexery a Search Heads[65]. Forwardery se starají o odchyt dat a jejich transport do ostatních instancí Splunku. Existují 2 typy Forwarderů – Heavy a Universal. Universal Forwardery se instalují na servery, odkud chceme data transportovat do Splunku, Heavy Forwardery jsou na straně Splunk serveru (případně mezi ním a aplikačními servery) a umožňují filtrovat přicházející data. Základní architektura je zobrazena na obrázku 2.21. Indexery se starají o uložení a indexaci dat, indexace umožňuje rychlé vyhledávání podobně jako v Elasticsearch. Indexace je rozdělení dat do logických datastorů podle nějakých indicií. Search Heads se starají o analýzu a vizualizaci těchto dat, případně o reporting). Popis procesu dat je zobrazen na obrázku 2.22.



Obrázek 2.21: Obecné schéma architektury Splunku[65]



Obrázek 2.22: Sekvence základních kroků v Splunku[65]

Je možné si v aplikaci nakonfigurovat tzv. knowledge objects, jedná se o objekty, kterými jsou obohacována přitékající data[68]. Může se jednat o různé typy objektů, například uložené dotazy, typ eventu ale i reporty nebo alerty. Je možné nakonfigurovat alerting podle vlastních pravidel. Velkou výhodou Splunku je existence mobilní aplikace. V mobilní aplikaci je možné nastavit si dashboardy, ale lze z ní také dostávat upozornění.
Je velmi jednoduché nakonfigurovat a spustit Splunk na vlastních serverech nebo v cloudu[69]. Nejedná se o opensource aplikaci, ale přesto lze Splunk využívat zdarma. Bohužel tato možnost je limitována na 500 MB dat denně,

což z ní už pro menší až středně velké firmy dělá placenou funkci. Aktuálně Splunk vychází v Cloud a Enterprise verzi.

Shrnutí

V případě Graylogu a Elastic Stacku se jedná o vhodné řešení pro firmy. Oba sety aplikací mají velkou možnost vlastních úprav a otevřený zdrojový kód. Splunk bohužel takovouto možnost nenabízí, což z něj dělá v tomto případě nejméně vhodnou volbu, navíc placená verze začíná už na půl GB dat denně. Díky tomu, že jak Graylog tak Elastic Stack využívají stejný základ (Elasticsearch), není obtížné provést migraci z jedné aplikace na druhou (jedná se vlastně jen o změnu vizualizační aplikace)[55].

Přehled vlastností Graylogu, Elastic Stacku a Splunku je zobrazen v tabulce v příloze B.

2.5 Možnosti sestavení testovacího prostředí

Vývojář často potřebuje při vývoji softwaru testovat funkčnost kódu, případně jeho integraci s jinými aplikacemi (v případě webových aplikací například backend a frontend). Dále je třeba si zajistit takové prostředí, které je snadno obnovitelné, restartovatelné do původního nastavení a není ovlivněné vlastním systémem vývojáře, abychom se vyhnuli typickému "ale na mém stroji to funguje"[70].
Dalším use-casem je například nábor nového pracovníka. S využitím kontejnerů na OS lze snadno a rychle spustit nové prostředí na novém stroji.

Pro testování a vývoj na vlastním stroji je nejčastěji využívána služba Docker, která se stará o virtualizované prostředí pomocí kontejnerů.

2.5.1 Kontejnery vs Virtuální stroje

Virtuálních strojů (zkráceně VM) jsou 2 typy. Typ A je virtualizován přímo na na hardware – tedy přímo na HW je nainstalován hypervizor, který se

stará o jednotlivé VM, typickým příkladem je třeba ESXi od VMware. Typ B je virtualizace nad OS – tedy hypervizor je nainstalován jako aplikace v operačním systému, zde je typickým příkladem například VirtualBox.
Kontejnery jsou lehkou formou druhého typu VM. Instalují se na již běžící HW s OS – příkladem Docker. Kontejnery jsou abstrakcí aplikační vrstvy, která dohromady balíčkuje kód a dependence[70]. Jeden OS kernel může sdílet více kontejnerů, přestože každý běží v izolovaných procesech. Porovnání jednotlivých virtualizačních metod je na obrázku 2.23.

Možnosti izolace jsou podobné jak v případě VM tak v případě kontejnerů. Rozdílná je velikost, VM virtualizuje hardware a potřebuje tedy plnou kopii operačního systému (zabírá většinou desítky GB místa na disku) a trvá jim delší dobu nabootovat, kontejnery na druhou stranu virtualizují OS, jsou "lehčí" (zabírají většinou řádově desítky MB), efektivnější a snáze přenositelné[71]. V kontejnerech na jednom stroji může běžet více aplikací, které si vyžádají méně OS a VM.
Kombinace VM s kontejnery nabízí širokou škálu využití při nasazování a správě aplikací.

2.5.2 Docker

Docker je virtualizační služba běžící na operačním systému. Obecné schéma běhu kontejnerizované aplikace je zobrazeno na obrázku 2.24. Jedná se o lightweight virtualizační platformu, jedním z důvodů vzniku je velikost tradičních VM, kde je třeba plná kopie operačního systému jako nadstavba nad hypervizorem. Docker vznikl jako opensource projekt v roce 2013. Je napsán v jazyce Go. Přináší zjednodušené využívání funkcí, které už operační systém Linux (konkrétně jeho kernel) nabízí, navíc přidává unifikované API pro kontrolu těchto funkcí. Základními funkcemi, které jsou využívány jsou LinuX Containers (LXC), cgroups a copy-on-write filesystem. LXC je user-space control package, který používá kernelové jmenné prostory (namespaces) na izolaci kontejneru od hosta[74]. Namespace procesu se pak stará pouze o procesy, které běží uvnitř kontejneru. Síťový namespace poskytuje kontejneru síťové zařízení a virtuální IP adresu.

Control Groupy se starají o limitování a zajišťování zdrojů. Díky tomu si Docker může upravovat zdroje (například paměť, CPU, prostor na disku nebo I/O operace), které jsou dostupné danému kontejneru. Cgroups dále poskytují metriky o použití zdrojů a dovolují tak Dockeru monitorovat využití zdrojů v kontejnerech a procesech a rozvrhnout volné zdroje pro větší stabilitu.

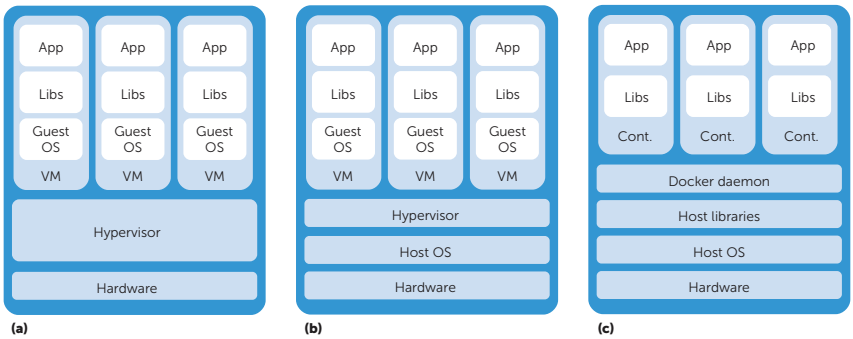
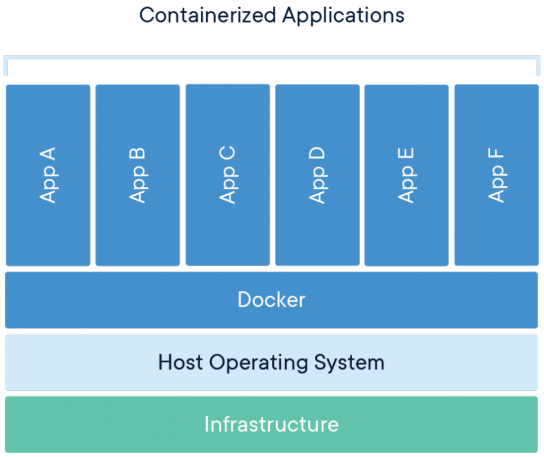


FIGURE 1. Comparing various application runtime models: (a) a type 1 hypervisor, (b) a type 2 hypervisor, and (c) a container.

Obrázek 2.23: Porovnání jednotlivých VM[72]



Obrázek 2.24: Obecné schéma infrastruktury Dockeru – podobné Virtual Machine typu B[73]

Docker využívá AuFS (Advanced Multi-Layered Unification Filesystem) jako filesystem pro kontejnery. Jedná se o vrstevnatý filesystem, který může transparentně překrývat jeden nebo více dalších filesystemů. Pokud tedy proces potřebuje změnit nějaký soubor, AuFS vytvoří kopii tohoto souboru a poté je schopný sjednotit (merge) více vrstev do jediné reprezentace filesystemu. Tento proces je nazývaný copy-on-write[71]. Využití AuFS umožňuje Dockeru používání některých obrazů (images) jako základ pro více kontejnerů (například jeden CentOS obraz jako základ pro mnoho kontejnerů). Díky tomu se ušetří paměť i místo na disku a vede to také k mnohem rychlejšímu vytvoření kontejnerů. Další výhodou je možnost verzování obrazů – každá nová verze obsahuje pouze diff (soubor změn) oproti předchozí verzi, což

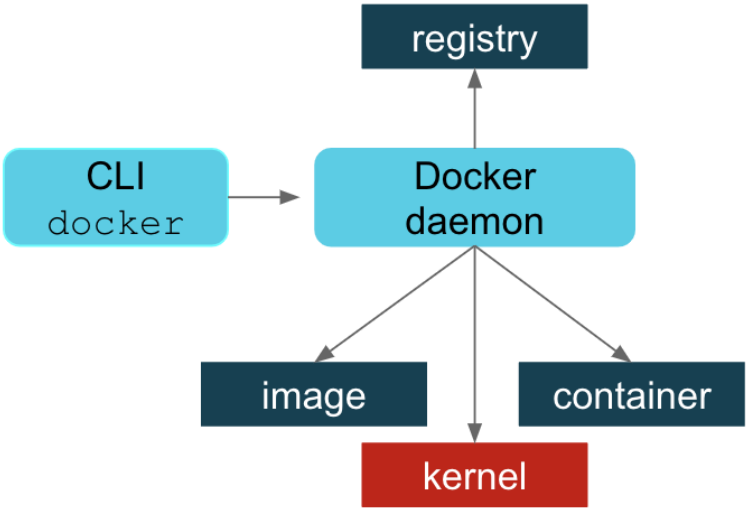
vede k minimalizaci velikosti obrazů a šetření místa na disku. Dále to také znamená jednoduše reprodukovatelné prostředí a možnost zobrazení si tzv. complete audit trail změn mezi jednotlivými verzemi kontejnerů.

Vytvoření Docker kontejneru je velmi rychlé, trvá řádově setinu doby vytvoření klasické VM. Další výhodou je velká možnost škálovatelnosti. Kontejnery jsou snadno přenositelné, lze je zkomprimovat pomocí jednoho příkazu "docker export <container> > container.tar" a poté tento kontejner spustit na jiném stroji. Kontejnerovaný software (ať už Windows/BSD nebo Linux based) poběží vždy stejně bez ohledu na infrastrukturu. Jak už bylo zmíněno, kontejnery izolují aplikaci od prostředí, což zajišťuje jednolitost přes všechny prostředí (stage, produkce, test apod.)[75].

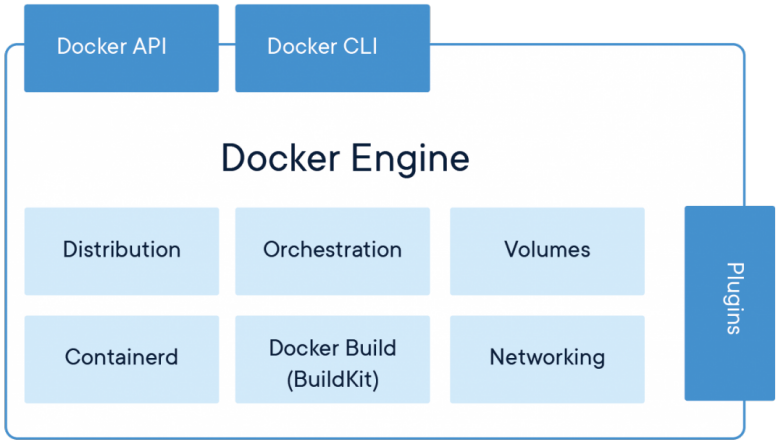
Kontejnery na Docker engine mají výhodu standardu, který se používá napříč celým odvětvím. Díky tomu, že každý kontejner nepotřebuje vlastní OS, nabízejí efektivnější využití zdrojů serveru. Přidává bezpečností prvek díky další vrstvě izolace.

Docker funguje tak, že jakýkoli CLI příkaz, který uživatel zadá, je odeslán do Docker Daemona [13], který ho následně provede – tj. stáhne/odešle Docker obraz (image) z registry (vzdálený repozitář), spravuje kontejnery, komunikuje s kernelem – dělá tedy vlastně všechnu práci. Daemon je tak single-point-of-failure, jak lze vidět na následujícím diagramu 2.25, selhání daemonu znamená selhání celého prostředí. Docker nabízí komunitní hub pro sdílení obrazů (image) zvaný DockerHub[77]. Jakýkoliv registrovaný uživatel si tam může vystavit své Docker image a DockerHub mu je sestaví a připraví závislosti ke stažení. Tyto uživatelské image mohou být privátní i veřejné.
Nevýhodou Docker kontejnerů je fakt, že k jejich spuštění a běhu je potřeba uživatel s root (superuser) právy. To z nich dělá možný attack vektor na systém[72]. Tento problém je adresován například v CharlieCloud, který umožňuje spustit Docker kontejnery bez root práv, avšak k vytvoření kontejneru jsou stále superuser práva potřeba[78].

Docker containers (kontejnery). Kontejnery jsou standardizované jednotky softwaru, které umožňují vývojářům izolovat aplikaci od prostředí, a řeší tak problém "funguje to na mém stroji", je proto snadné si rozběhnout stejné prostředí na více strojích[70]. Docker container image je lightweight, standalone a spustitelný soubor softwaru, který obsahuje vše, co je potřeba ke spuštění aplikace, tj. kód, runtime, systémové nástroje, knihovny a nastavení[73]. Obrazy (image) se stávají kontejnery za běhu (runtime), v případě Dockeru za běhu využívající Docker Engine. Funkce Docker Engine lze vidět na následujícím obrázku 2.26.



Obrázek 2.25: Schéma architektury procesů Dockeru – centrální prvek je Docker daemon[76]



Obrázek 2.26: Obecné schéma Docker Engine[79]

Dockerfile poskytuje jednoduchý skript (podobný Makefile), který definuje přesně jak se má sestavit daný Docker image. Syntaxe Dockerfile je velmi jednoduchá, jednodušší než například Ansible nebo Chef. Uživatel potřebuje jen o málo více než základ psaní v shellu. Dockerfile poskytuje řadu výhod – zatímco Docker image může zabírat desítky MB, Dockerfile je plaintext soubor, jehož velikost se pohybuje v řádech tisíckrát nižších, snadno se tedy může verzovat například v gitu. Popisuje celý obsah obrazu v snadno čitelné formě – závislosti programů, enviromentální proměnné – slouží tedy i jako dokumentace. Protože jsou Dockerfily přeměněné na obrazy pomocí docker build, je velmi nepravděpodobné, že by na různých strojích vznikly různé obrazy.
Dockerfile soubory se snadno udržují a upravují. Lze v nich také používat dědění – tedy vytvořit si základní Dockerfile, a ten následně rozšiřovat pomocí jiných Dockerfile souborů. Příklad Dockerfile je v následujícím v následující ukázce 2.1[80]:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Listing 2.1: Příklad jednoduchého Dockerfile, který rozvíjí Docker image ubuntu:18:04

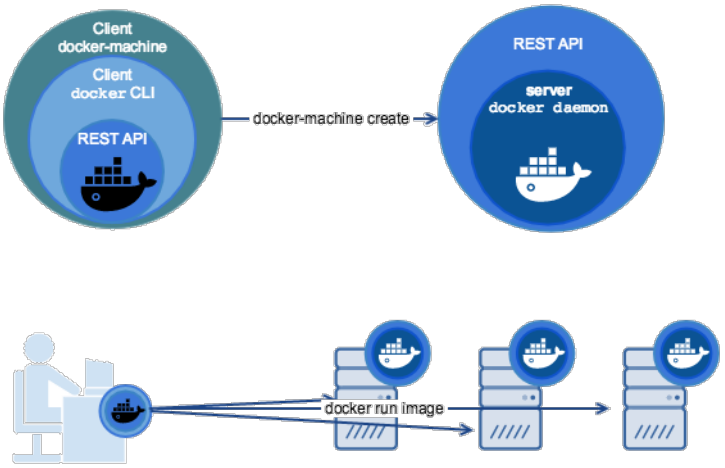
Docker Compose. Docker compose je nástroj pro definování a běh (správu) multikontejnerových Docker aplikací[81]. Ke konfiguraci prostředí se používá YAML soubor s názvem "docker-compose.yml". Poté lze celé prostředí spustit pomocí jediného příkazu.
Použití Compose vyžaduje standardně 3 kroky. Prvním je nakonfigurovaní Dockerfile. Druhým krokem je vytvoření docker-compose.yml souboru a nakonfigurovaní prostředí, příklad je v následující ukázce kódu 2.2[81].

```
version: '2.0'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Listing 2.2: Příklad jednoduchého konfiguračního souboru Compose, který vytváří 2 kontejnery (web je vytvořen z Dockerfile a Redis, který je vytvořen z dostupného Docker image) a jeden volume, který je mapován do aplikace web na cestu /var/log

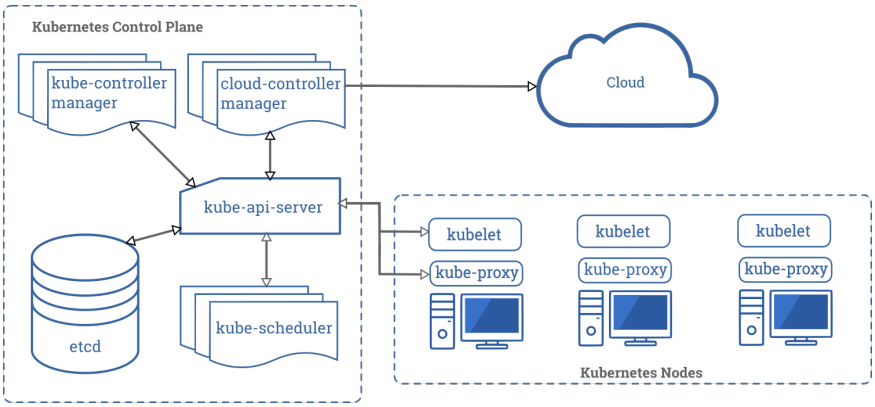
Třetím krokem je spuštění prostředí pomocí příkazu “docker-compose up”. Compose obsahuje příkazy pro správu kompletního životního cyklu aplikace. Umožňuje startovat, zastavovat a rebuildovat služby. Dále je možné si zobrazit status vybraných služeb, vytvořit proud (stream) logů ze služeb nebo spustit nějaký příkaz ve vybrané službě.
Díky Compose lze snadno spustit i více izolovaných prostředí na jednom hostu. Například v průběhu CI Compose spouštět s přepínačem -p, který umožňuje nastavení vlastního jména projektu (případně nastavení enviromentální proměnné COMPOSE_PROJECT_NAME)[82].
Compose se nejčastěji využívá pro development (vývojová) prostředí, ale také pro prostředí na automatické testování. Ačkoliv to není standardem, lze Compose využívat i v předprodukčním a produkčním prostředí[83] – v oficiální dokumentaci se k tomu nabízí možnost ”restart: always” a dále možnost použití kromě základního docker-compose.yml souboru i další – díky ”-f” přepínači – který by obsahoval konfigurace pro jednotlivá prostředí.
Compose se nemusí pouštět pouze lokálně, lze ho spustit i proti vzdálenému stroji za použití enviromentálních proměnných DOCKER_HOST, DOCKER_TLS_VERIFY, DOCKER_CERT_PATH.

Docker na vzdálených strojích. Docker Machine umožňuje správu Docker kontejnerů na vzdálených strojích[84], dále možnost běhu Docker kontejnerů na strojích s OS X a Windows[85] a také správu Swarm clusteru. Schéma Docker Machine je zobrazeno na obrázku 2.27.
Díky Swarm clusteru je možné použití Compose vůči více hostům, a tím pádem i běh aplikací na více serverech.



Obrázek 2.27: Obecné schéma funkce Docker Machine a ovládání vzdálených Docker hostů[84]

Kubernetes (K8s) je opensourcový systém pro správu a orchestraci kontejnerů. Jedná se o snadno přenosnou a rozšiřitelnou platformu[86]. Shlukuje kontejnery do logických jednotek pro snadnou správu. Jedná se o pokročilejší technologii než je Docker Compose. Kubernetes má mnoho různých funkcí, stará se o load balancing, service discovery, obnovu služeb při pádu (nebo při zaseknutí), topologie služeb, správu konfigurací (ukládání konfigurací, přihlašovacích klíčů), orchestraci storage a mnoho dalšího. Schéma Kubernetes a jeho funkcí je na následujícím obrázku 2.28.



Obrázek 2.28: Obecné schéma Kubernetes[86]

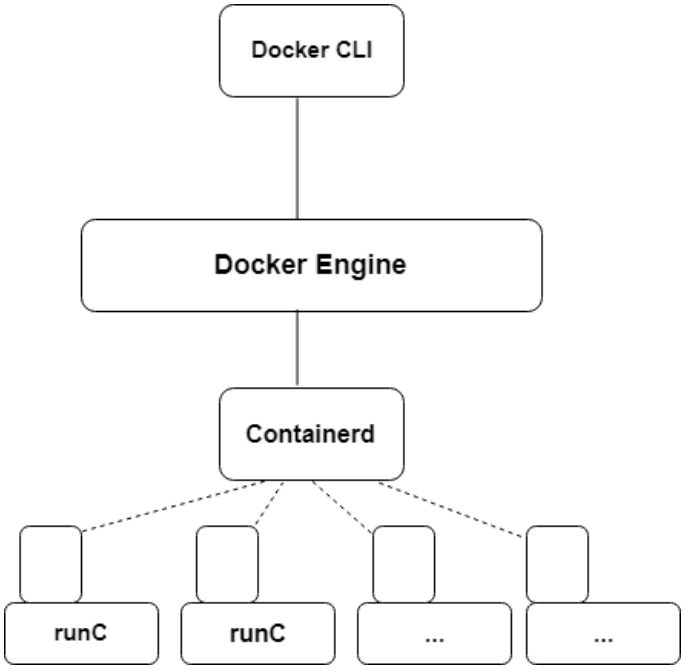
Docker a bezpečnost. Zabezpečení Dockeru závisí na 3 hlavních faktorech[72]: izolaci procesů na userspace úrovni (o to se stará Docker daemon), vymáhání izolace pomoc kernelu a zabezpečení síťových operací.
Jak již bylo řečeno, kontejnery Dockeru závisí zcela na Linux kernelu a jeho funkcích – namespace, cgroups, hardening a capabilities. Izolace přes namespace a zahazování capabilities je v základu povoleno, ale limitace pomocí cgroups není, ta se musí spouštět pro každý kontejner zvlášť. Host hardening pomocí Linuxového kernelu uplatňuje bezpečností limitace na kontejnery, aby bylo zabráněno možnému úniku z kontejneru do hosta – zde jako příklad můžeme uvést například SELinux nbeo Seccomp. Defaultně je izolační konfigurace poměrně striktní, možnou slabinou ale zůstává sdílený síťový most (bridge), což umožňuje útoky na ARP (address resolution protocol) cache. Defaultní izolační konfigurace ale mohou být uvolňovány při startu kontejnerů.
Docker využívá síťové zdroje pro distribuci obrazů a vzdálenou kontrolu Docker daemonů (na remote hostech). V prvním případě platí, že si Docker obrazy ověřuje pomocí hashe a spojení s registry navazuje pomocí TLS. V druhém případě, tedy v případě vzdálené kontroly, se navazuje spojení pomocí socketu (buď Unix nebo TCP). Přístup k socketu by útočníkům dal přístup k pull a spuštění jakéhokoliv kontejneru v privilegovaném módu a tedy by i snadno získali root přístup do hosta.
Útoky na Docker prostředí mohou být přímé a nepřímé[72]. Mezi přímé se

řadí útoky na síťovou nebo systémovou komunikaci, případně přímé útoky na produkční kontejnery vystavené do internetu. V případě získání root práv pro kontejner může útočník například zahájit DoS útok proti ostatním kontejnerům běžícím na hostu, případně z něho uniknout, a ovládnout tak celý systém hosta. Nepřímé útoky mohou způsobit stejné škody jako útoky přímé, k provedení ale místo přímého přístupu využívají ekosystém Dockeru (jako například repozitáře obrazů).
Hlavními cestami do systému jsou zejména: zranitelnosti v deployment toolchainu Dockeru, nezabezpečená lokální konfigurace (zpravidla se jedná o úpravy defaultní konfigurace, které oslabují zabezpečení), slabý lokální access control a případné zranitelnosti v distribuci Docker obrazů.

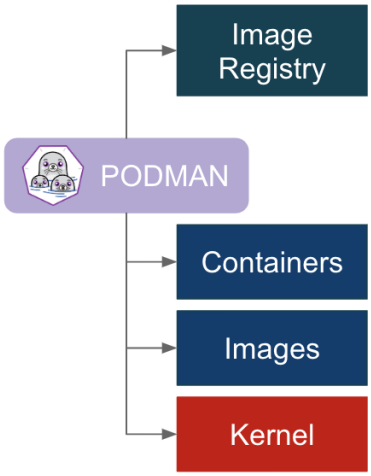
2.5.3 Podman

Podman je opensource projekt dostupný pro většinu Linuxových platforem vytvořený organizací Containers. Jednoduše řečeno, Podman je upravený Docker – "alias docker = podman". Jedná se o daemonless kontejnerový engine pro vývoj, správu a běh OCI (Open Container Initiative) kontejnerů a kontejnerových obrazů v systému Linux[87]. Podman poskytuje CLI kompatibilní s CLI Dockeru, proto je využití Podmanu snadné – není třeba se nic nového učit.
V Podmanu, díky absenci daemonu, dochází k veškeré komunikaci napřímo pomocí runC runtime procesu[76, 88]. Kde runC běží je zobrazeno na obrázku 2.29 Jedná se o lightweight přenosný kontejnerový runtime, který je využíván i Docker daemonem, v Podmanu se využívá přímo, a díky tomu, že odpadá daemon proces, je vyřešena jedna ze slabin Dockeru, kterou je "single point of failure" daemon proces, jak lze vidět na následujícím obrázku 2.30.

Výhodou oproti Dockeru je možnost spuštění a běhu kontejnerů bez superuser práv. Kontejnery mohou být spuštěny uživatelem s normálními právy. V takovém případě je využit uživatelův namespace pro nastavení root v kontejneru pro uživatele, který spouští Podman kontejner[87]. Rootless Podman běží v tzv. lock-down módu s omezeními, která vyplývají z práv uživatele, který Podman spouští. Některá omezení lze odstranit, ale nikdy nelze dosáhnout kontejneru s právy vyššími, než má uživatel. Rootless Podman kontejnery mají k dispozici téměř všechny funkce jako kontejnery s root právy, ale stále je možné narazit na občasné zádrhely, většinou způsobené nesprávnou konfigurací operačního systému.



Obrázek 2.29: Schéma interních procesů Docker[88]



Obrázek 2.30: Obecné schéma architektury procesů Podmanu[76]

Podman spravuje celý ekosystém od podů, kontejnerů až po obrazy a container volumes pomocí knihovny libpod[89]. Obecné schéma container engine je na obrázku 2.31.

Podman podporuje OCI i Docker formáty obrazů. Dále je k dispozici více

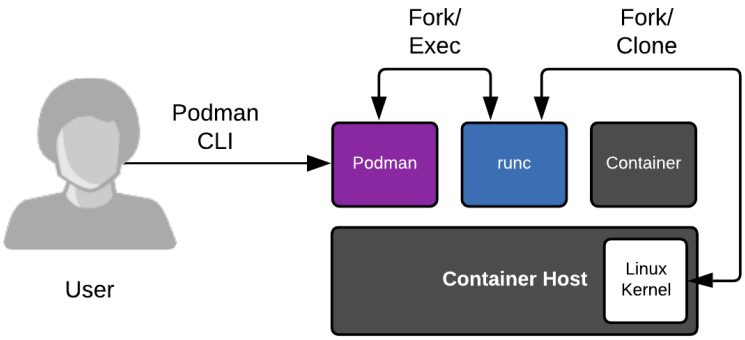
možností bezpečného stahování obrazů a jejich ověření. Umožňuje správu kontejnerových obrazů – podobně jako Docker umožňuje překrývání FS a vrstvetní obrazů. Poskytuje kompletní možnost správy kontejnerů za celou dobu jejich běhu. Podporuje pody pro možnost správy skupiny kontejnerů dohromady. Dále umožňuje sdílení obrazů a kódu pomocí CRI-O[90], což je obdoba DockerHubu.

K definici obrazů lze použít stejně jako u Dockeru Dockerfile, další možností je využití jiného opensource projektu Buildah, který se specializuje na tvorbu OCI kontejnerových obrazů[91]. Příkazy Buildahu replikují příkazy v Dockerfile, to umožňuje tvorbu obrazů s použitím, ale i bez použití Dockerfile, a zároveň odstraňuje nutnost tvorby obrazů s root právy. Možnost absence Dockerfile umožňuje integraci jiných skriptovacích jazyků. Buildah používá jednoduchý fork-exec model, neběží jako daemon a je založen na golang API. Příkaz "buildah run" emuluje příkaz "RUN" v Dockerfile[76]. Příklad Buildah skriptu[91] je v následující ukázce kódu 2.3:

```
#!/usr/bin/env bash -x

ctr1=$(buildah from "${1:-fedora}")

## Get all updates and install our minimal httpd server
buildah run "$ctr1" -- dnf update -y
buildah run "$ctr1" -- dnf install -y lighttpd

## Include some buildtime annotations
buildah config --annotation "com.example.build.host=$(uname -n)"
    "$ctr1"

## Run our server and expose the port
buildah config --cmd "/usr/sbin/lighttpd -D -f
    /etc/lighttpd/lighttpd.conf" "$ctr1"
buildah config --port 80 "$ctr1"

## Commit this container to an image name
buildah commit "$ctr1" "${2:-$USER/lighttpd}"
```

Listing 2.3: Příklad jednoduchého Buildah skriptu

Podman také, jako Docker, má rozšíření o Compose. Bohužel projekt je stále ještě nedokončený. Hlavním cílem je možnost spouštění kontejnerových prostředí pomocí stejmných docker-compose.yml souborů, jaké jsou používány v Docker Compose, pouze s Podman kontejnerovým managementem[93]. Je doporučováno používat ho pouze pro vývojová, případně testovací prostředí. Pro produkční prostředí na single-hostu se doporučuje MiniKube nebo MiniShift, pro více-hostová prostředí pak Kubernetes nebo OpenShift od společnosti Red Hat.
Základními podporovanými Linuxovými distribucemi jsou Ubuntu a Fedora



Obrázek 2.31: Popis práce jednotlivých částí Podmanu – není centrální prvek[92]

od společnosti Red Hat. Společnost Red Hat, která je v mnoha ohledech v IT odvětví trendsetter, je předním průkopníkem (investorem) Podmanu a CRI-O[92]. Jako důvody udávají jednak snadný přechod z Dockeru na Podman, díky stejné CLI syntaxi, dále vidí velkou hodnotu v daemonless a rootless běhu kontejnerů.

Kapitola 3

Návrh

V této části práce navrhnu, jak je vhodné postupovat při verzování a nasazování aplikací používajících mikroslužbovou architekturu. K tomu budu využívat stávajících a ověřených technologií, které již byly zmíněny v předchozí kapitole. Přestože je cílem práce vytvořit nástroj, který umožní lepší správu mikroslužeb, je nejprve třeba se zaměřit i na samotný vývoj a nasazování jednotlivých služeb. Proto se v této kapitole budu věnovat nejen návrhu nástroje, ale také návrhu procesu vývoje a nasazení.

Navrhovaný nástroj jako celek se bude skládat z několika kroků.

Prvním krokem celého procesu je vytvoření jednotlivých služeb a jejich verzování. Toho dosáhneme pomocí CI procesu v některém z CI/CD nástrojů, které byly zmíněné v předchozí kapitole. V průběhu CI se otestuje služba, vytvoří se Docker obraz a aplikace se sestaví, poté se celý kontejner exportuje, komprimuje, otestuje a nahraje do repozitáře. Detailněji je tento krok popsán v kapitole 3.1.

Druhým krokem je sestavení celé aplikace a následné nasazení. Toho je dosaženo také pomocí CI procesu, jako v prvním kroku. V průběhu CI se získají dostupné verze služeb, sestaví se celé prostředí a pustí se integrační testy a následuje nasazení na jednotlivá prostředí. Celkově se oba výše zmíněné kroky zaměřují zejména na kvalitně nastavené CI/CD, které automatizují velkou část životního cyklu kódu. Detailněji je tento krok popsán v kapitole 3.2.

Dalším důležitým krokem je využití nástrojů pro analýzu logů, které se postarají o náročnou část detekce chyb a jejich reportování. Tento krok je popsán v kapitole 3.3.

Posledním krokem je vytvoření nástroje, který umožní přehled a správu jednotlivých prostředí a zobrazí případné problémy v procesu. Jedná se o jednoduchý nástroj, který bude využívat API jednotlivých nástrojů použitých pro předchozí kroky a data získaná dotazy na API zobrazí v přehledné formě. Tento krok je detailněji popsán v kapitole 3.4.

Předpokladem je využívání verzovacího nástroje (například Git) a nějakého nástroje s podporou CI/CD. Pro verzování služeb a jejich běh budou využívány kontejnery, které zajistí rychlejší a snazší automatizaci, lepší zabezpečení i snazší přesunovatelnost[13].

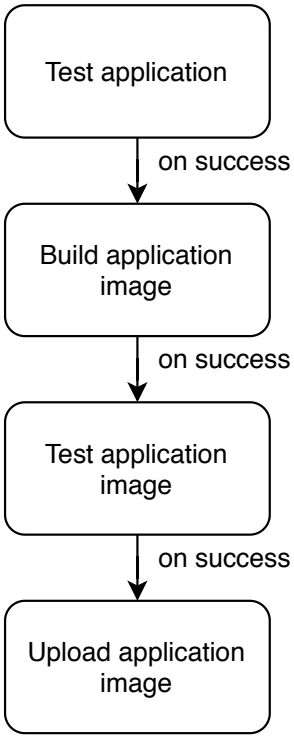
3.1 Příprava mikroslužeb

Jak již bylo zmíněno v kapitole 2.5, v mikroslužbové architektuře je trendem dnešní doby využívání kontejnerového prostředí. Pro jednotlivé služby navrhuji v základní podobě stejný postup při CI, který by se sestával minimálně ze 4 kroků (další přidané kroky už záleží na konkrétní službě), celý proces je zobrazen na následujícím obrázku 3.1.

3.1.1 Test aplikace

Prvním krokem je test aplikace. Tento krok by se měl provádět na všech větvích při každém commitu nebo pushi do upstreamu. Měla by se zde pouštět minimální sada automatických testů, které zkontrolují, že základní funkčnost služby nebyla narušena. Délka trvání těchto testů a celého tohoto kroku by měla být co možná nejkratší a rozhodně by neměla přesáhnout 10 minut. Přesáhnutím této doby už by byl vývojář příliš blokován v práci.

Alternativní možností je provádět testy při vytvoření “Pull Requestu” (v případě GitHubu) nebo “Merge Requestu” (v případě GitLabu). V takovém případě by se mohla pouštět komplexnější sada testů. Tato možnost by ale nedávala vývojáři okamžitou zpětnou vazbu, což je v agilním vývoji nevýhoda.



Obrázek 3.1: Schéma CI pro samostatnou mikroslužbu

Ideálním řešením by byla kombinace obou předchozích možností. To by sice znamenalo přidání dalšího kroku do přípavy mikroslužby, ale také by to zaručilo důkladněji otestovanou službu před případnou (a velmi doporučenou) manuální kontrolou formou Code Review.

3.1.2 Sestavení obrazu aplikace

Druhým krokem je sestavení samotné aplikace a přiřazení verze. Zde se nejedná pouze o verzování zdrojového kódu služby, vzhledem k využívání Gitu nebo například SVN je o to už postaráno. Důležité je mít verzovanou aplikaci jako celek a to včetně závislostí, které se zpravidla stahují ze vzdálených repozitářů třetích stran, nad kterými nemáme z naší strany kontrolu.

Nemožnost kontroly využívaných závislostí je často problémem při sestavování aplikací. Stejný kód naší aplikace se může, například kvůli nedostatečnému specifikování verzí těchto závislostí, nebo kvůli problémům v síti

a dočasné nedostupnosti závislostí, chovat při dvou různých sestaveních jinak. Proto je uložení a zaverzování stažených závislostí důležitou součástí verzování jak dané služby, tak celé mikroslužbové aplikace.

Sestavení služby (build) včetně závislostí je vhodné provést v kontejneru specifikovaném a nastaveném pro potřeby běhu aplikace. Úspěšné sestavení je dalším testem, kterým si musí každá služba projít. Následně se celý kontejner exportuje a uloží.

Tento krok nemusí být součástí každé větve, ale například pouze těch větví, které slouží pro release služby. Samozřejmě zde záleží na nastavení vývojového procesu, dát vývojáři možnost získat Docker obraz hotové služby může být vhodné.

3.1.3 Test obrazu aplikace

V tomto kroku je třeba ověřit, že se předchozí krok provedl správně. Je potřeba obraz kontejneru rozbalit, kontejner spustit a ověřit, že výsledné chování je podle očekávání. Tento krok by měl vždy následovat po kroku sestavení obrazu aplikace a měl by tedy být zakomponován v CI všech větví, které předchozí krok využívají.

3.1.4 Nahrání obrazu do repozitáře

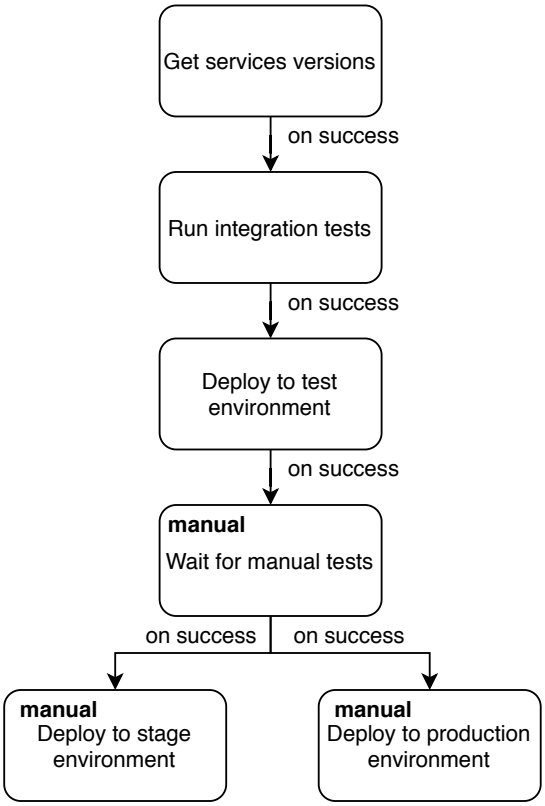
Posledním krokem přípravy mikroslužby je nahrání obrazu sestaveného kontejneru do repozitáře, odkud bude dostupný ke stažení pro vývojáře a servery, které poté aplikaci spouští (ať se jedná o testování v průběhu CI nebo při nasazení do některého z prostředí). Tento krok by měl vždy být pouze součástí těch větví, které slouží pro release služby.

3.2 Integrace mikroslužeb

Pro integrování mikroslužeb by měl vzniknout ve verzovacím systému (například Gitu) vlastní repozitář, který bude obsahovat informace o tom, jak

na sobě mají být služby závislé a také jaké služby daná aplikace využívá. Dále by repozitář měl obsahovat podpůrné skripty, které slouží například k doplnění či získání verzí služeb dostupných ve vzdáleném repozitáři.

Celé CI/CD integrace mikroslužeb by se mělo skládat minimálně z pěti kroků, které na sebe vzájemně navazují. Celkový počet kroků může být vyšší s ohledem na například počet testů nebo počet prostředí, kam se aplikace nasazuje. V této práci se zaměřuji zejména na 3 standardní prostředí, kterými jsou prostředí testovací, před-produkční (stage), kde zpravidla probíhá prezentace zákazníkovi a závěrečné testování, a produkční prostředí. Schéma CI/CD pro celou aplikaci je zobrazeno na následujícím obrázku 3.2.



Obrázek 3.2: Schéma CI/CD pro celou aplikaci (projekt)

K integraci služeb bude využit compose nástroj, který usnadňuje manipulaci a nasazení aplikace jako celku, třebaže je rozdělena na větší množství kontejnerů. Repozitář pro integraci by měl obsahovat konfigurační soubory (nebo jejich šablony) pro compose nástroj, které budou zároveň sloužit jako model systému.

3.2.1 Získání dostupných verzí služeb

Prvním krokem integračního procesu je vytvoření konfiguračního souboru pro nasazení aplikace. Jedná se hlavně o doplnění připravené šablony compose konfigurace o požadované verze jednotlivých služeb. V případě, že se jedná o nové nasazení, případně nespecifikovanou verzi, bude použita poslední dostupná verze aplikace.

3.2.2 Integrační testování

Dalším krokem je sestavení celé mikroslužbové aplikace s minimální konfigurací a daty (v případě, že je využívána databáze). Tento krok slouží ke spuštění automatických integračních testů, které ověří, že aplikaci v daném složení má smysl nahrávat na testovací prostředí.

3.2.3 Nasazení na testovací prostředí

Následující krok je nasazení do testovacího prostředí. Tento krok může být automatický, ale to s sebou nese riziko přenasazení prostředí v průběhu provádění testů, proto se jako vhodnější varianta jeví nasazování manuální – zde se nejedná o manuální psaní příkazů, pouze manuální spuštění kroku v CI/CD nástroji. Ve své podstatě by kroky nasazení na prostředí měly být všechny stejné, pouze spuštěny s požadovanou konfigurací.

K nasazení na vzdálené prostředí lze využít compose nebo jiný nástroj pro správu kontejnerových prostředí.

3.2.4 Výsledky manuálních testů

Následující navrhovaný krok nazývám “blokem”. Tento krok by se měl pouštět manuálně, a to pouze v případě, že testování na testovacím prostředí neodhalí žádné problémy. Důvodem zavedení tohoto kroku je, aby nemohlo dojít k nechtěnému nasazení neotestované aplikace na předprodukční nebo dokonce produkční prostředí. Následující kroky by měly být závislé na úspěšném splnění tohoto kroku a bez něho nemožné spustit.

3.2.5 Nasazení na další prostředí

Následující 2 kroky, které zde popíši, by měly více či méně kopírovat třetí krok. Jedná se o nasazení aplikace na stage (předprodukční) a produkční prostředí. Mezi tyto 2 kroky lze přidat blokující krok, který ale v tomto případě nepovažuji za tak podstatný, jelikož by z testovacího prostředí měla aplikace vycházet bez zásadních chyb.

Ačkoliv agilní přístup k vývoji říká, že lze automatizovat i nasazování do produkčního prostředí, nemělo by se tak dít bez dohledu člověka, který v případě, že proces selže podnikne potřebné kroky. Proto tyto 2 kroky navrhuji pouštět jako manuální.

3.3 Detekce chyb

Pro detekování chyb z aplikací se bude využívat aplikačních a serverových logů. Díky nástrojům pro agregaci logů, které byly zmíněny v předchozí kapitole lze snadno nastavit vzory, které má nástroj detekovat a reagovat na ně.

V případě správy více mikroslužbových aplikací je příhodné tyto aplikace od sebe oddělit například na úrovni indexu v Elasticsearch, případně na úrovni streamu v Graylogu.

Jednotlivá prostředí budou oddělena jmennou konvencí, která se bude udržovat na úrovni “hostname” jednotlivých serverů. Pro každé prostředí bude unikátní sufix hostname, který bude popisovat, o které prostředí se jedná, v případě produkčního prostředí lze sufix vynechat a označit prostředí bez sufixu za produkční.

3.4 Návrh nástroje

Samotný nástroj se bude starat hlavně o agregaci a zobrazení dat získaných z REST API nástrojů, které jsou využity pro předchozí části práce. Dále

bude nástroj umožňovat nasazení aplikace na jednotlivá prostředí a volbu konkrétní konfigurace verzí aplikace.

3.4.1 Přihlášení

Po spuštění nástroje je nejdříve nutné se přihlásit. Z přihlašovacích údajů se poté získají tokeny, které se budou využívat pro komunikaci s API využívaných nástrojů. Využití uživatelových přihlašovacích údajů do systému společnosti umožní omezení přístupu uživatele pouze k projektům, ke kterým má nastavený přístup a zároveň nebude muset nástroj obsahovat žádné globální přihlašovací údaje či tokeny, to zajistí lepší bezpečnost. Uživatel nástroje bude limitovaný omezeními vyplývajícími z přístupových práv v jednotlivých nástrojích.

3.4.2 Úvodní stránka

Po přihlášení, které se ověří úspěšným získáním tokenů do nástrojů, uživatel uvidí stránku s výběrem projektů a popisem funkčnosti a účelu nástroje. Po výběru projektu bude uživatel automaticky přesměrován na stránku, kde budou zobrazené možnosti pro aplikaci a přehled informací o jednotlivých prostředích.

3.4.3 Aplikační stránka

Prvním prvkem na stránce aplikace bude stále výběr aplikací, který umožní přechod na stránku jiné aplikace, tato funkcionalita zůstává od první stránky po přihlášení stejná.

Dále bude možnost spustit nové nasazení prostředí, a to včetně spuštění automatických integračních testů s vybranými verzemi aplikací. Tato funkcionalita je vhodná zejména pro reprodukci chyb, které se mohly vyskytnout v produkčním prostředí. Není nutné tuto funkci využít přímo k nasazení na testovací prostředí. Vývojáři budou mít možnost stáhnout si konfiguraci compose vygenerovanou v prvním kroku integrace aplikací, a díky tomu si prostředí spustit a opravit na vlastním stroji.

Následně bude zobrazena tabulka, která bude přehledně zobrazovat verze jednotlivých služeb na každém prostředí. Dále odkaz do nástroje pro agregaci logů spolu s případnými detekovanými problémy, které budou nastaveny v nástroji.

Posledním zobrazeným prvkem pro každou aplikaci bude tabulka posledních 10 integrací, ve které budou zobrazené verze s jakými integrace běžela, odkaz na integraci do CI/CD nástroje, celkový status, se kterým integrace proběhla a všechny kroky, kterými integrace prošla. V případě manuálních kroků bude umožněno z nástroje daný krok spustit.

Kapitola 4

Implementace

Tato kapitola je zaměřena na konkrétní implementaci všech funkcí této práce. Implementace probíhala na interním projektu FIMS3.
Pro práci byly využity nástroje GitLab, Artifactory, Docker, Docker-Compose a Graylog. Dále byl využit programovací jazyk Python spolu s YAML konfiguračními soubory.

4.1 FIMS3

FIMS je interní webová aplikace v Quanti s.r.o., která dovoluje projektovým manažerům a učetním monitorovat a řídit finance firmy: finanční stav projektů, jednotlivé faktury, platy zaměstnanců apod. V současnosti se používá verze 2.0 aplikace a ve vývoje se nachází verze 3.0.

Skládá se z 3 základních a 2 pomocných služeb. Základní služby jsou backend v Javě, MySQL databáze MariaDB a webový frontend. Pomocnými službami jsou Adminer, který slouží jako nástroj pro ruční náhled a správu databáze a Consul, který se používá ke konfiguraci aplikace. Vnitřně vyvíjenými aplikacemi jsou frontend a backend, ostatní služby jsou pouze využívané.

Backend, je služba starající se o logiku aplikace. Jedná se o aplikaci napsanou v programovacím jazyce Java, která využívá Java 11, Maven a framework Spring. Projektové jméno je fims_backend.

FIMS 3 Backend je serverová část aplikace FIMS, která se zabývá perzistencí finančních dat a poskytováním webového API podle JSON:API standardu[94]. Dané API je konzumované aplikaci FIMS 3 Frontend pro zobrazení uživatelského rozhraní. Jako základní technologie se používá Java 11 se Spring Bootem. Na pozadí běží relační databáze MariaDB pro persistenci dat a Consul pro jednoduchou konfiguraci.

Frontend je aplikace poskytující grafické uživatelské rozhraní. Jedná se o single-page aplikaci, která má základ v NodeJS a využívá wrapper Yarn a Angular 9. Pro state management je použita technologie NgRx. Po stažení všech závislostí se pomocí Angularu z aplikace vytvoří minifikované javascriptové soubory doplněné o konfiguraci pro konkrétní prostředí.
V průběhu této práce byla služba fronendu rozšířena a doplněna o službu Nginx v jednoduché konfiguraci, která se stará o publikování dat. Vzniklé 2 služby nesou projektová jména fims_web_client a fims_web_client_nginx.

4.2 Implementace CI/CD

Pro CI/CD byl využit nástroj GitLab. Hlavním důvodem užití GitLabu byly předchozí zkušenosti s definováním .gitlab-ci.yml souborů. Dalšími důvody pro využití GitLabu byly existence python knihovny pro práci s GitLabem v kombinaci s velmi obsáhlým GitLab REST API, dále využívání GitLabu jako Git nástroje a také možnost využívání šablon pro konfiguraci pipeline.

V GitLabu je využíván Docker exekutor pro zajištění specifičtějšího a neměnného prostředí pro jednotlivé kroky. Tento exekutor funguje tak, že se spustí Docker kontejner se zadaným obrazem (image), v případě, že obraz systému není specifikovaný se použije defaultní Docker obraz nastavený při registraci runneru.
V tomto případě je využíván upravený Docker obraz, který jsem pojmenoval docker_build:v4, jeho zdrojový Dockerfile je popsán v ukázce v příloze A.1. Jedná se o obraz vycházející z aktuální stabilní verze Docker dostupné na DockerHubu se specifickým tagem dind[95] (tedy Docker in Docker), který umožňuje spouštět vnořené kontejnery Dockeru. Dále vychází z Dockerfile uživatele arielkv na GitHubu, který připravuje podporu pro spojení s Artifactory. Následné úpravy v Dockerfile jsou instalace potřebných nástrojů pro jednotlivé kroky CI/CD. Tyto nástroje jsou curl, bash, jq, git a docker-compose.

4.2.1 Pipeline mikroslužby

Prvním krokem implementace bylo určení úkonů, které se budou opakovat pro téměř každou mikroslužbu, případně které se budou v jediné pipeline opakovat. Tím vznikl prostor pro vytvoření šablony (template), kterou lze aplikovat na každou přidanou službu i na služby na dalších případných zpracovávaných projektech.
Kroky, které se opakují v rámci jedné pipeline, jsou: vytvoření připojení k Artifactory, naplnění proměnných služby pro build, nahrání a import komprimovaného Docker obraz a na konci zastavení a odstranění běžících Docker kontejnerů.
Kroky, které se opakují pro každou službu, jsou, kromě již dříve zmíněných, následující: vytvoření Docker obraz pro službu, spuštění Docker obraz jako kontejner a nahrání obrazu do repozitáře.

Pro vytvoření připojení k Artifactory bylo třeba v Artifactory vytvořit uživatele, který bude mít dostatečná práva na repozitářích. Dále bylo třeba v GitLabu naplnit u daných repozitářů, v tomto případě nejsou všechny repozitáře projektu FIMS v jedné oddělené skupině, proto bylo třeba naplnit proměnné CI/CD u každého repozitáře samostatně.
Plněny byly **ARTIFACTORY_URL, ARTIFACTORY_USER, ARTIFACTORY_PASS a ARTIFACTORY_DOCKER_REPOSITORY**. Pro snazší přístup do Docker repozitáře v Artifactory byla také zavedena proměnná **DOCKER_REPOSITORY_KEY**, která nese jméno Docker repozitáře v Artifactory. Tento stage nese název “setup_artifactory_connection” a je implementován v následující ukázce 4.1.

```
.setup_artifactory_connection: & setup_artifactory_connection
    # Install JFrog CLI
    - curl -fL https:// getcli . jfrog . io | sh
    # Configure Artifactory instance with JFrog CLI
    - ./ jfrog rt config --url=$ARTIFACTORY_URL
        --user=$ARTIFACTORY_USER --password=$ARTIFACTORY_PASS
    - ./ jfrog rt c show
    # Docker client info
    - docker info
    # Login to Artifactory docker registry
    - docker login -u $ARTIFACTORY_USER -p $ARTIFACTORY_PASS
        https://$ARTIFACTORY_DOCKER_REPOSITORY
```

Listing 4.1: Implementace stage nastavení připojení k Artifactory podle oficiální dokumentace

Plnění proměnných služby pro build pod sebou skrývá vytvoření unikátní verze a dočasné jméno kontejneru, který je používán v průběhu build procesu. Verze je generována jako ”poslední git tag + ‘_‘ + číslo aktuálního běhu pipeline”. Tím se získá unikátní verze pro každý build. V případě, že se jedná o nové nasazení s využitím release větve ve tvaru ”release/verze”, je použita

verze z názvu větve místo posledního tagu. Implementace je popsána v ukázce 4.2.
Dále je vytvořen dočasný název kontejneru používaný pro build. Ten se tvoří spojením názvu projektu a _app_1, což vyplynulo ze specifikace v docker-compose.yml souboru, který je využíván. Tento krok nese název “get_version”.

```
.get_version: &get_version
    - touch app.version
    - export COMMIT_HASH=${CI_COMMIT_SHORT_SHA}
    - export APP_VERSION=$( [[ ${CI_COMMIT_REF_NAME} =~
      ^release\/ ]] && echo "$CI_COMMIT_REF_NAME" |sed -e
      's|release\/||g' |||[[ -z ${CI_COMMIT_TAG} ]] && echo $(git describe
      --tags |cut -d'-' -f1) ||echo ${CI_COMMIT_TAG})
    - export APP_NAME=${CI_PROJECT_NAME}
    - export APP_TAG=${APP_VERSION}_${CI_PIPELINE_ID}
    - echo ${APP_NAME}:${APP_TAG} >app.version
    - cat app.version
    - export CONTAINER_NAME=${CI_PROJECT_NAME}_app_1
```

Listing 4.2: Implementace parsování verze z proměnných dostupných v CI

Nahrání a import komprimovaného Docker obraz je krok, který se opakuje několikrát v každé pipeline. Jedná se o dekompresi artefaktu přesouvaného mezi jednotlivými stage a následný import do Dockeru jako image. Ke kompresi a dekompresi se využívá populární UNIX nástroj Gzip. Tento krok se nazývá “load_docker_image_from_gzip” a je demonstrován v následující ukázce kódu 4.3.

```
.load_docker_image_from_gzip: &load_docker_image_from_gzip
    - gunzip -c images/${CONTAINER_NAME}.tar.gz |docker import - app
```

Listing 4.3: Ukázka implementace šablony kroku nahrání Docker obrazu

Protože se téměř pro každý krok využívá nástroj Compose pro Docker, je tento nástroj využit i v závěrečném, úklidovém kroku. Jedná se pouze o jediný příkaz, který ukončí a smaže všechny Docker kontejnery a image spuštěné pomocí nástroje Compose (viz následující ukázka 4.4). Tento krok se jmenuje “clean_up”.

```
.clean_up: &clean_up
  after_script:
    - docker-compose down --rmi all
```

Listing 4.4: Ukázka implementace šablony kroku úklid

Vytvoření obrazu pro službu je krok, který je nejčastěji upravován u jednotlivých služeb, jelikož každá technologie využívaná jednotlivými službami má částečně jiné nároky. Tento krok také využívá 3 z výše zmíněných kroků, konkrétně *get_version*, *setup_artifactory_connection* a *clean_up*. Šablona tohoto kroku tedy slouží spíše jako návod, jak by měl stage vypadat pro každou

jednotlivou službu. Výstupem tohoto stage je komprimovaný (gzip) artefakt obsahující exportovaný docker kontejner, který obsahuje kompletně všechny závislosti potřebné pro běh služby. Tento stage nese název "build_docker_image" a je popsán v následující ukázce 4.5.

```
.build_docker_image:
  before_script:
    - *get_version
    - *setup_artifactory_connection
  script:
    - docker-compose up -d
    - mkdir images
    - docker export ${CONTAINER_NAME} |gzip >
        images/${CONTAINER_NAME}.tar.gz
  <<: *clean_up
  artifacts:
    paths:
      - images/*.tar.gz
    expire_in: 1 day
```

Listing 4.5: Ukázka implementace šablony kroku vytvoření image

Nahrání obrazu do Artifactory je stage, který je pro všechny služby stejný a neměnný a jeho implementace je demonstrována v následující ukázce 4.6. Využívá 3 z výše zmíněných kroků, konkrétně *get_version*, *load_docker_image_from_gzip* a *setup_artifactory_connection*. V tomto stage se otaguje obraz a následně se otagovaný obraz se odešle do repozitáře v Artifactory.

```
.upload_docker_image:
  before_script:
    - *get_version
    - *setup_artifactory_connection
    - *load_docker_image_from_gzip
  script:
    # Push Docker image to Artifactory
    - docker tag app
        ${ARTIFACTORY_DOCKER_REPOSITORY}/docker/${APP_NAME}:${APP_TAG}
    - ./jfrog rt dp
        $ARTIFACTORY_DOCKER_REPOSITORY/docker/${APP_NAME}:${APP_TAG}
        $DOCKER_REPOSITORY_KEY --build-name=${APP_NAME}
        --build-number=${APP_TAG}
    # Collect build environment variables and build tools information using JFrog
        CLI
    - ./jfrog rt bce ${APP_NAME} ${APP_TAG}
    # Publish build information to Artifactory
    - ./jfrog rt bp ${APP_NAME} ${APP_TAG}
```

Listing 4.6: Implementace nahrání Docker obrazu do Artifactory implementována podle oficiální dokumentace

Pipeline fims_backend

Služba backendu potřebuje k běhu Javu 11, dále využívá Maven a framework Spring. Pro potřeby aplikace byl opět vytvořen vlastní Dockerfile, který vychází z openjdk:11-jdk oficiálního Docker obrazu a je doplněn o Maven. V posledním kroku buildu je ještě navíc nahrání aplikace do kontejneru. Konfigurace této služby se děje v runtime pomocí služby Consul. Lze tedy vygenerovat jeden **.jar** soubor, který se využije pro všechny prostředí.

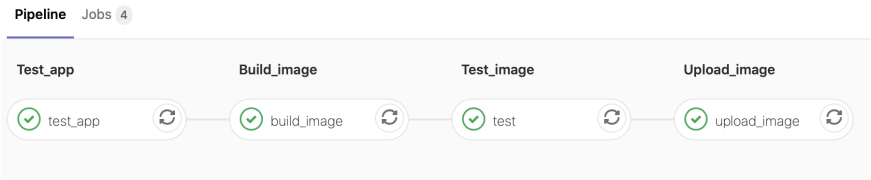
Prvním krokem je spuštění automatických testů. Pro správný běh potřebuje aplikace služby databáze (MariaDB) a consul. Proto byl i pro testovací účely vytvořen konfigurační soubor Compose s názvem testing-compose.yml, který sestaví image z výše zmíněného Dockerfile, a následně zkusí spustit na krátkou dobu aplikaci příkazem "spring-boot:run". Tento stage lze spouštět pro každou větev, doba běhu se v prvních bězích pohybovala okolo 8 minut. Stage je pojmenován test_app.

Druhým krokem je build obrazu. Pro build je použit stejný Dockerfile jako v předchozím kroku. Po vytvoření obrazu a spuštění kontejneru se opět stáhnou závislosti, vytvoří **.jar** souboru a následně je kontejner exportován a pomocí gzip komprimován. Na konci tohoto stage je kontejner zničen. V tomto případě je tento krok spuštěn pouze pro master a release/verze větve. Tento stage je nazván build_docker_image.

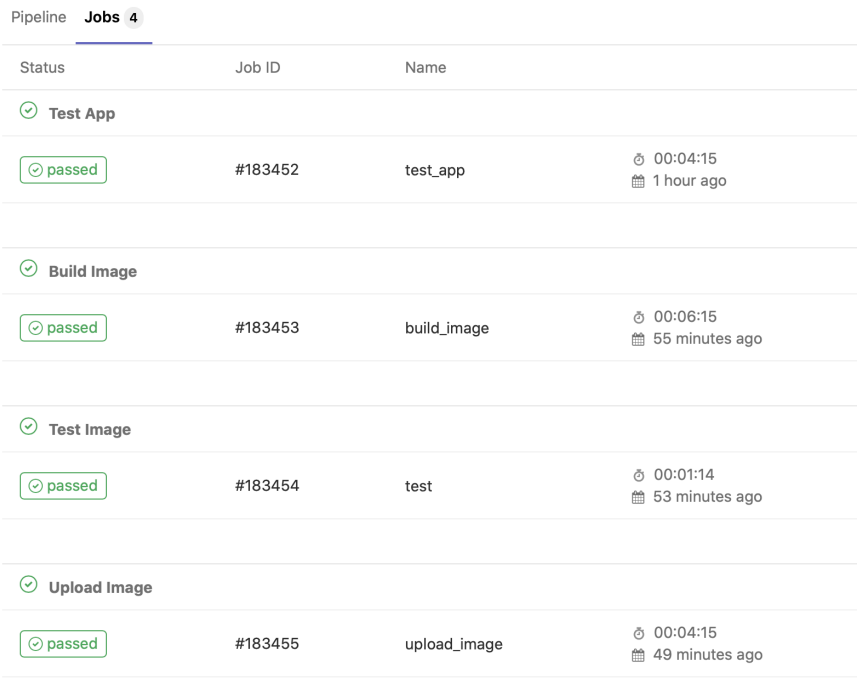
Následuje stage testování obrazu. V tomto kroku se po dekompresi a nahrání obrazu z předchozího kroku za použití dalšího konfiguračního souboru test-compose.yml sestaví minimalistické prostředí nutné pro běh aplikace. Poté, co prostředí naběhne, se pomocí nástroje cURL otestuje, že se aplikace chová očekávaně. Tento stage navazuje na build_docker_image.

Posledním krokem je nahrání obrazu do Artifactory. Tento stage vychází z obecné šablony a není nijak upraven. Spouští se pouze v případě, že všechny předchozí kroky skončily úspěšně. Tento krok se bude zpravidla pouštět pouze na master a release/verze větvích, které jsou určeny pro vydání.

Výslednou pipeline služby fims_backend lze vidět na následujícím obrázku 4.1. Detailnější zobrazení včetně doby běhů jednotlivých jobů lze vidět na obrázku 4.2.



Obrázek 4.1: Výsledná pipeline služby fims_backend skládající se ze 4 jobů



Obrázek 4.2: Detail pipeline služby fims_backend

Pipeline fims_web_client

Pipeline frontendu se drží postupu navrženého v předchozí kapitole. Jednotlivé stage pipeline jsou tedy test aplikace, vytvoření Docker obrazu, otestování obrazu a poslední je nahrání obrazu do repository.
Tato webová aplikace potřebuje ke svému běhu přítomnost aplikací Node, Yarn a Angular. Pro aplikaci byl vytvořen vlastní upravený Dockerfile vycházející z oficiálního image node:13-buster, který je doplněný o Yarn a Angular. Posledním krokem buildu je přidání souborů frontendu do kontejneru.
Pro webové aplikace využívající NodeJS (a případně Yarn nebo Angular) obecně platí, že využívají konfigurační soubory[96], kde jsou specifikovány adresy backendů, případně dalších služeb. To je dáno tím, že JavaScript je

spouštěn a web následně renderován v prohlížeči uživatele a nikoliv na straně serveru. Proto je v tomto případě verzován obraz po stažení závislostí, ale ještě před generováním jednotlivých JavaScriptových souborů, protože ty musí být generovány s ohledem na konfiguraci pro prostředí. Jelikož pro generování JavaScriptových souborů již není třeba stahovat žádné závislosti, proto lze tento proces označit za bezpečný a lze ho aplikovat pro každé prostředí zvlášť.

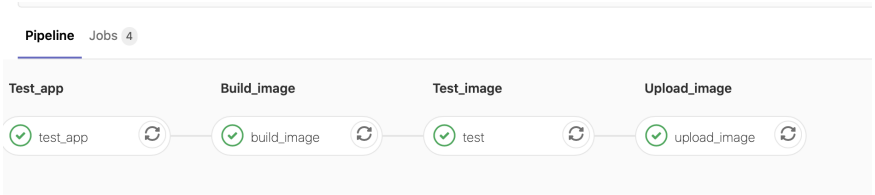
Prvním krokem je spuštění automatických testů aplikace. Nejdříve je ale třeba vytvořit obraz a kontejner pro běh aplikace. Jakmile je obraz vytvořený a kontejner spuštěný, spustí se stažení závislostí pomocí Yarnu a následně pomocí Angular-CLI implementované testy. Tento krok je možné provádět pro jakoukoliv větev, a proto by testy neměly zabrat moc času.

Druhým krokem je build obrazu. Pro build je použit stejný Dockerfile jako v předchozím kroku. Po vytvoření obrazu a spuštění kontejneru se opět stáhnou závislosti a následně je kontejner exportován a pomocí gzip komprimován. Na konci tohoto stage je kontejner zničen. V tomto případě je tento krok spuštěn pouze pro master a release/verze větve. Tento stage je nazván build_docker_image.

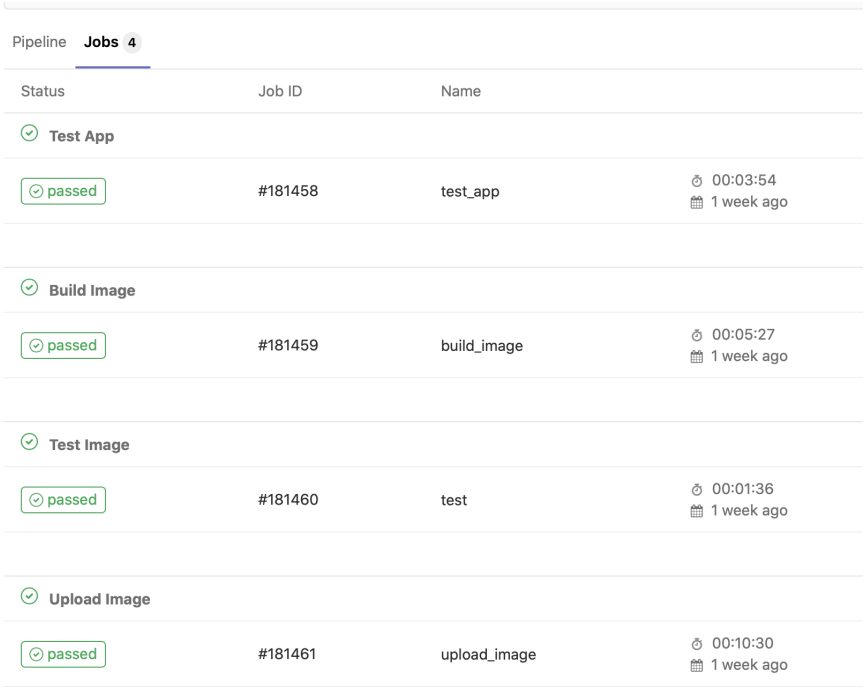
Třetím krokem je test obrazu. Docker obraz je po dekompresi nahrán do Dockeru a spuštěn jako kontejner, následně je spuštěn příkaz *ng serve*, který spustí aplikaci na portu 4200, a pomocí nástroje cURL otestován. Tento stage navazuje na build_docker_image.

Posledním krokem je nahrání obrazu do Artifactory. Tento stage vychází z obecné šablony a není nijak upraven. Spouští se pouze v případě, že všechny předchozí kroky skončily úspěšně. Tento krok se bude zpravidla pouštět pouze na master a release/verze větvích, které jsou určeny na vydání.

Výslednou pipeline služby fims_web_client lze vidět na následujícím obrázku 4.3. Detailnější zobrazení včetně doby běhů jednotlivých kroků lze vidět na obrázku 4.4.



Obrázek 4.3: Výsledná pipeline služby fims_web_client složená ze 4 jobů.



Obrázek 4.4: Detail pipeline služby fims_web_client

4.2.2 Pipeline projektu

Vzhledem k různorodosti jednotlivých mikroslužbových aplikací je šablona pro mikroslužbové projekty velmi omezená.
Kroky, které se opakují napříč projekty, jsou zejména příprava spojení s Artifactory, spuštění prostředí a následný úklid.

Příprava spojení s Artifactory již byla popsána v předchozí kapitole 4.2.1, proto se jí nyní věnovat nebudu.

Spuštění prostředí slouží ke spuštění prostředí kontejnerů v daném stage z defaultního konfiguračního souboru docker-compose.yml. V případě, že je třeba využít jiného konfiguračního souboru je tento krok přepsán v konkrétním projektovém .gitlab-ci.yml souboru. Tento krok je nazván "prepare_docker_env" a konkrétní implementace je v následující ukázce 4.7.

```
.prepare_docker_env: &prepare_docker_env
    - docker-compose up -d
```

Listing 4.7: Implementace šablony spuštění docker-compose projektu

Úklid prostředí byl také popsán v předchozí kapitole 4.2.1, a proto se mu nyní věnovat nebudu.

Dalšími prvky šablony agregují a rozšiřují předchozí 3 zmíněné kroky. Stage nazvaný "run_project" ve svém před-skriptu (before_script) spustí *setup_artifactory_connection* a následně ve script části spustí krok prepare_docker_env (viz následující ukázka 4.8).
Stage nazvaný "run_test_project" rozšiřuje stage zmíněný v předchozím odstavci o post-skript, ve kterém se vyčistí spuštěné prostředí (viz následující ukázka 4.9). Jak již název napovídá, tento stage se využívá zejména pro integrační testy, kde je po dokončení, ať již úspěšném nebo neúspěšném, nutno ukončit spuštěné prostředí.

```
.run_project:
  extends: .  setup_artifactory_connection
  script:
    - *prepare_docker_env
```

Listing 4.8: Implementace přípravného kroku spuštění prostředí Docker Compose

```
.run_test_project:
  extends: .run_project
  after_script:
    - *clean_up
```

Listing 4.9: Implementace rozšíření přípraveného kroku o úklid prostředí

Pipeline fims_microservices

Integrační pipeline projektu FIMS se skládá z 6 kroků. Ve třech z těchto kroků se jedná o nasazení na vzdálené prostředí – testovací, předprodukční a produkční. Dále obsahuje krok doplnění konfiguračního souboru Compose, stage na spuštění integračních testů a blokovací stage, který byl zmíněný v předchozí kapitole.

Prvním krokem je doplnění konfiguračního souboru pro Docker Compose. Jedná se o spuštění krátkého skriptu, který, pomocí nástroje cURL a jq, získá data z API Artifactory ohledně posledních verzí v případě, že nejsou přímo specifikované při spuštění pipeline. Tyto verze jsou následně doplněny do souborů docker-compose.yml a testing-compose.yml. Soubor testing-compose.yml posléze slouží ke spuštění integračních testů. Hlavní soubor docker-compose.yml se využívá k nasazování na vzdálená prostředí. Výsledkem tohoto stage jsou 3 artefakty – jedná se o dříve zmíněné 2 soubory compose a soubor **services.versions**, který obsahuje rozdělené názvy proměnných a verze jednotlivých služeb. Tento krok může být šablonizován

a jmenuje se “create_compose”.

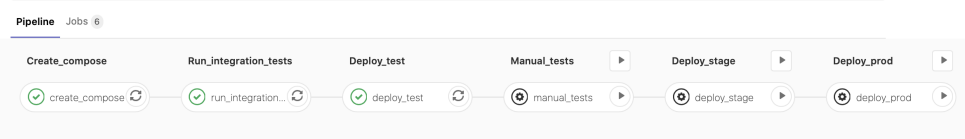
Druhým krokem je stage spuštění integračních testů. V tomto kroku se poprvé spustí celá aplikace složená ze služeb pohromadě. K tomu se využije soubor testing-compose.yml vytvořený v předchozím kroku. V případě selhání nebude žádný z následujících kroků dostupný a tím se také zajistí, že aplikace na vzdálených strojích bude spustitelná. Následně se spustí připravené automatické integrační testy. Mělo by se jednat pouze o testy, které ověří správnou funkčnost aplikace, nikoli reakci na vytížení (stress testy). Vzhledem k možné velikosti aplikace budou nároky na výkon runner stroje už tak velmi vysoké.

Třetím krokem, nazvaným “deploy_test”, je nasazení aplikace na vzdálený testovací server. Toho se dosáhne pomocí vytvořeného docker-compose.yml souboru v kombinaci s proměnnou **DOCKER_HOST**, která stanovuje, kam se budou kontejnery Dockeru spouštět. V tomto případě se využije přístup přes SSH na testovací server fims3-docker-testing. Ještě před přístupem je třeba přidat tento server společně do SSH known_hosts aktuálního runner kontejneru společně s privátním klíčem pro přístup získaným z proměnných CI/CD.

Dalším krokem je blocker, který jsem pojmenoval “manual_tests”. Jeho funkce byla již popsána v předchozí kapitole 3.2.4. V tomto případě se jedná pouze o jednoduchý manuální krok, který nedělá nic kromě vypsání “TESTS PASSED!” do outputu. Po dokončení manuálního testování v testovacím prostředí lze tento krok spustit, a odemknout tak stage pro nasazení na stage a produkci.

Následující 2 kroky – nasazení na předprodukční (stage) prostředí a produkční prostředí jsou také manuální kroky. Ve své podstatě kopírují třetí krok (nasazení na testovací prostředí), změnou je pouze jiný obsah proměnné **DOCKER_HOST** a také záměna přepínačů z test na stage, případně prod, pro použití správného konfiguračního souboru. Změna těchto přepínačů je navržena tak, že stačí použít jednoduchý příkaz “sed”, který v prvním případě nahradí “testing” za “stage” pro předprodukční prostředí a následně “stage” za “prod” v případě produkčního.
Nemožnosti spustit tyto 2 kroky bez překonání blocker kroku je dosaženo pomocí konfigurace .gitlab-ci.yml souboru, kde je použita závislost ”needs: ‘manual_tests’”. Nevýhodou této závislosti je, že pipeline zůstává ve stavu “running” do doby, než je buď zrušena, nebo je krok “manual_tests” spuštěn a dokončen.

Výsledná pipeline projektu fims_microservices lze vidět na následujícím obrázku 4.5. Tato pipeline čeká na dokončení a spuštění manuálních testů.



Obrázek 4.5: Výsledná pipeline projektu fims_microservices složená z 6 jobů

4.3 Detekce chyb

Pro detekování chyb je využit nástroj Graylog. V Graylogu je vytvořen stream, který získává data pouze ze serverů, na kterých běží prostředí FIMS, to je zařízeno filtrováním podle zdrojových hostname. Tento stream je pojmenován fims_microservices, podle projektu, který je využit pro integraci.
Na tomto vybraném streamu je nadefinován event, který detekuje přítomnost klíčového slova "error" v logu a v tom případě je vytvořen event typu Alert a odeslána emailová notifikace. Tento event slouží pro demonstrační účely a měl by být doplněn o další klíčové vzory.

4.4 Nástroj pro správu

Nástroj pro správu, jak již bylo zmíněno v kapitole návrhu lze rozdělit do 3 stránek. První stránka je přihlášení, zde musí uživatel zadat přihlašovací údaje, než je možné pokračovat na další stránku. Úvodní stránka obsahuje možnost zvolit projekt a následně základní informace o nástroji. Aplikační stránky jsou pro každý projekt a obsahují informace získané z API využitých nástrojů.
Nástroj je napsaný v jazyce Python. Hlavním důvodem volby Pythonu je udržitelnost a možnost dalšího rozvoje. Python je mezi administrátory široce používaný pro své skriptovací vlastnosti a jednoduchou syntaxi. Dalším důvodem je existence obalu (wrapperu) pro API GitLabu.

Nástroj je logicky rozdělen do 4 Python skriptů podle poskytovaných funkcí. Jedná se o skripty poskytující přístup k API použitých nástrojů a o agregační skript poskytující logiku a uživatelské rozhraní.

4.4.1 GitLab API

GitLab API je wrapper odesílající požadavky službě GitLab. Díky rozsáhlému REST API GitLabu je vzdálená práce s GitLabem jednoduchá. Využívána je pouze část API související s pipelinami a projekty. Pro komunikaci s API je využit uživatelský token pro ochranu uživatelských údajů.
API skript využívá Python modulů ast, asq, json. Dále je využit gitlab modul a komunitní skript parse_token, který jsem upravil pro Python3 a volání z jiných Python skriptů.
Tento skript obsahuje 15 veřejných (public) metod a dvě pomocné metody privátní.

- **___exists(items, item_name)** – privátní pomocná metoda, která ověří, zda v seznamu Gitlab objektů existuje objekt s názvem item_name. Vstup je seznam objektů a název hledaného objektu. Výstup je true v případě, že se objekt v listu nachází, false v případě, že se objekt v listu nenachází.

- **___get_older_artifacts(project, branch, job_name, artifact_name, pipeline_number)** – privátní pomocná metoda pro hledání artefaktu v předcházejících pipelinách. Jedná se o rekurzivní metodu. Vstupem jsou objekt projektu (instance třídy gitlab.base.project), název větve, název jobu, název hledaného artefaktu a pořadí prohledávané pipeline. Výstup je získaný artefakt nebo prázdný textový řetězec zakódovaný UTF-8 v případě, že artefakt nebyl nalezen.

- **get_api_token(host, login, password)** – public metoda využívající parse_token skript, pomocí kterého získá uživatelský token pro GitLab. Vstup je URL GitLabu, uživatelské jméno a heslo. Výstup je získaný uživatelský token.

- **prepare_gitlab(host, token)** – public metoda pro připravení objektu GitLabu, který je následně využíván pro další dotazy. Vstup je URL GitLabu a uživatelský API token. Výstup je objekt třídy gitlab.Gitlab.

- **get_projects(gitlab_api)** – public metoda pro získání seznamu přístupných projektů pro uživatele. Vstup je instance třídy gitlab.Gitlab vytvořená v prepare_gitlab(). Výstup je seznam (List) objektů třídy gitlab.base.project.

- **get_project_id(projects, project_name)** – public metoda pro získání id zadaného projektu. Využívá privátní metody ___exists(). Vstup je list projektů (instance třídy gitlab.base.project) a název zadaného projektu. Výstup je id projektu v případě, že ___exists() vrátí true, jinak je výstup None.

- **get_project(gitlab_api, project_id)** – public metoda pro získání objektu projektu. Vstup je objekt třídy gitlab.Gitlab vytvořená v prepare_gitlab() a id projektu. Výstup je None v případě, že id projektu je None, jinak je vrácena instance třídy gitlab.base.project s žádoucím id.
- **get_project_pipelines(project, results)** – public metoda pro pro získání pipeline pro daný projekt. Vstup je objekt projektu (instance třídy gitlab.base.project) a počet výsledků, který, pokud není specifikovaný, je 10. Výstup je seznam (List) objektů třídy gitlab.base.pipeline.
- **get_pipeline(project, pipeline_id)** – public metoda pro získání objektu konkrétní pipeline. Vstup je objekt projektu (instance třídy gitlab.base.project) a id pipeline. Výstup je objekt třídy gitlab.base.pipeline se zadaným id.
- **get_pipeline_status(project, pipeline_id)** – public metoda pro získání aktuálního stavu dané pipeline. Vstup je objekt projektu (instance třídy gitlab.base.project) a id pipeline. Výstup je atribut status získaného objektu gitlab.base.pipeline.
- **get_pipeline_jobs(pipeline)** – public metoda pro získání seznamu jobů v dané pipeline. Vstupem je objekt pipeline (instance třídy gitlab.base.pipeline). Výstup je seznam (List) objektů třídy gitlab.base.job.
- **get_job_id(pipeline, job_name)** – public metoda pro získání id daného jobu (stage). Využívá se metody ___exists(). Vstup je objekt pipeline (instance třídy gitlab.base.pipeline) a název hledaného jobu. Výstup je, v případě že daný job existuje v zadané pipeline, id hledaného jobu, jinak None.
- **get_job(project, job_id)** – public metoda pro získání objektu konkrétního jobu. Vstup je objekt projektu (instance třídy gitlab.base.project) a id hledaného jobu. Výstup je None v případě, že id není, jinak je vrácen objekt třídy gitlab.base.job.
- **start_pipeline_job(project, pipeline_id, job_name)** – public metoda pro spuštění daného jobu v konkrétní pipeline. Vstup je objekt projektu (instance třídy gitlab.base.project), id žádané pipeline a název daného jobu. Výstup je status spuštěného jobu v případě, že job lze spustit, jinak None.
- **create_new_pipeline(project, branch, variables)** – public metoda pro vytvoření a spuštění nové pipeline s vybranými proměnnými. Vstup je objekt projektu (instance třídy gitlab.base.project), název žádané větve a seznam proměnných s hodnotami. Výstup je objekt nově spuštěné pipeline třídy gitlab.base.pipeline.
- **get_pipeline_versions(project, pipeline_id)** – public metoda pro získání seznamu verzí služeb, se kterými byla daná pipeline spuštěna. Vstup je objekt projektu (instance třídy gitlab.base.project) a id dané

pipeline. Výstup je seznam proměnných a verzí, v případě že existuje, jinak textový řetězec ”no versions available”.

- **download_job_artifact(project, branch, job_name)** – public metoda pro stažení artefaktu s verzemi na prostředí pro zadaný projekt a název jobu. Tato metoda je výhradně používána pro joby s názvem “deploy_test/stage/prod” pro získání souboru s verzemi nasazenými na dané prostředí. Vstupem je objekt projektu (instance třídy gitlab.base.project), název žádané větve a název jobu. Výstup je seznam služeb a verzí na daném prostředí (podle názvu jobu).

4.4.2 Artifactory API

Artifactory API je wrapper odesílající požadavky proti službě Artifactory. Artifactory poskytuje rozsáhlé REST API, pro potřeby této práce jsou využity ale pouze části týkající se uživatelských tokenů a získání dat z Docker registry. Pro komunikaci s API je využit uživatelský token pro ochranu uživatelských údajů.
API script využívá Python modulů requests, json a ast.
Je zde využívána globální proměnná **ARTIFACTORY_URL**.
Dále tento script poskytuje 5 veřejných a 1 privátní metodu.

- **___get_authorization_header(token)** – jedná se o privátní pomocnou funkci. Vstup parametr token je uživatelský API token pro Artifactory. Výstup metody je serializovaný JSON autorizační header.
- **get_api_token(username, password)** – public metoda pro získání uživatelského tokenu pro API. Vstup jsou uživatelské jméno a heslo. Výstup jsou v případě úspěchu token pro komunikaci na API a refresh_token pro prodloužení platnosti tokenu. V případě neúspěchu je vrácena prázdná hodnota.
- **refresh_api_token(token, refresh_token)** – public metoda pro prodloužení platnosti tokenu pro API. Vstup jsou API token a refresh_token získané předchozí metodou. Výstup jsou v případě úspěchu token pro komunikaci na API a refresh_token pro prodloužení platnosti tokenu. V případě neúspěchu je vrácena prázdná hodnota.
- **get_api_tokens(token)** – public metoda pro získání aktivních tokenů na API. Využívá privátní metodu ___get_authorization_header(). Jedná se pouze o ověřovací metodu a v následné implementaci není použita. Vstup je API token. Výstup je neparsovaná odpověď HTTP GET requestu.

- **get_docker_images(token, repository)** – public metoda pro získání seznamu Docker obrazů v Artifactory. Využívá privátní metodu ___get_authorization_header(). Vstup je API token a název repozitáře (standardně docker pro všechny). Výstup je seznam (List) docker obrazů.
- **get_docker_tags(token, repository, image_name)** – public metoda pro získání seznamu tagů pro konkrétní Docker obraz. Využívá privátní metodu ___get_authorization_header(). Vstup je API token, název repozitáře a název obrazu. Výstup je seznam (List) tagů pro daný obraz, pokud image neexistuje pak je Výstup seznam obsahující pouze jeden prvek "external'.

4.4.3 Graylog API

Graylog API je wrapper odesílající požadavky proti službě Graylog, která se stará o agregaci logů. Graylog také poskytuje široké REST API. Pro komunikaci s API je využit uživatelský token pro ochranu uživatelských údajů.
API script využívá Python modulů datetime, requests, ast a json. Dále je využíván opensource modul pro usnadnění komunikace s Graylog API zvaný Grapi.
Je využívána globální proměnná **GRAYLOG_URL**.
Dále tento script poskytuje 5 veřejných a jednu privátní metodu.

- **___initialize_graylog_connection(token, endpoint)** – jedná se o privátní pomocnou funkci, která inicializuje spojení s Graylogem. Vstup je uživatelský token a endpoint na API. Metoda vrací objekt Grapi inicializovaný na zadaný endpoint.
- **get_api_token(login, password)** – public metoda pro získání uživatelského tokenu. Vstup je uživatelské jméno a heslo. Výstup v případě úspěchu API token, v případě neúspěchu None.
- **get_streams(token)** – public metoda pro získání seznamu streamů existujících v Graylogu. Využívá privátní metodu ___initialize_graylog_connection(). Vstup je uživatelský API token. Výstup je seznam (List) párů stream title a stream id.
- **get_single_stream(token, stream_id)** – public metoda pro získání informací o streamu. Využívá privátní metodu ___initialize_graylog_connection(). Vstup je uživatelský API token a id požadovaného streamu. Návratová hodnota je JSON odpovědi na HTTP GET požadavek.
- **get_errors(token)** – public metoda pro získání všech alertů vygenerovaných v pro všechny streamy v Graylogu. Využívá privátní metodu

___initialize_graylog_connection(). Vstup je uživatelský API token. Výstup je JSON odpovědi na HTTP GET request.

- **get_error(token, stream_id, app)** – public metoda pro získání alertů vygenerovaných v Graylogu pro daný stream a pro vybranou aplikaci. Vstup je uživatelský API token, id streamu a jméno aplikace. Výstup je JSON odpovědi na HTTP GET request.

4.4.4 Nástroj a uživatelské rozhraní

Pro uživatelské rozhraní aplikace je využit framework Flask[97]. Jedná se o modul pro Python, který umožňuje vytvoření jednoduché webové aplikace s pomocí Jinja šablon a Werkzeug, které slouží jako WSGI. Flask původně vznikl jako wrapper okolo Jinja a Werkzeug. Flask jsem zvolil pro jeho jednoduchost, která je pro prototyp aplikace klíčová. Dále jsou využity Python moduly datetime a functools. Požadavky pro spuštění nástroje jsou dostupné v příloze D, ukázky z UI nástroje jsou dostupné v příloze E.

Struktura aplikace je následující: v adresáři templates/ jsou uloženy používané HTML šablony, adresář static obsahuje CSS soubory, hlavní část aplikace je v kořenovém adresáři pod názvem app.py.

Globální proměnné využívané ve skriptu app.py jsou **GITLAB_TOKEN, ARTIFACTORY_TOKEN, GRAYLOG_TOKEN, GITLAB_URL, ARTIFACTORY_URL, GRAYLOG_URL** a **GITLAB_API**. Proměnné se suffixem TOKEN se při přihlášení plní uživatelskými tokeny pro přístup na API. Proměnné se suffixem URL musí být před spuštěním vyplněné správnými adresami nástrojů. Proměnná GITLAB_API se plní hned po přihlášení a obsahuje inicializovaný objekt třídy gitlab.Gitlab.

Aplikace je logicky členěna do 3 základních druhů stránek. První je stránka login. Na té uživatel přistane při prvním spuštění aplikace. Dále je stránka úvod, která je umístěna na kořeni cesty. Na cestě /název-projektu jsou umístěny jednotlivé stránky projektů. Stránky úvodu a projektů jsou dostupné až po přihlášení. Skript tedy obsahuje 3 základní metody ovládané pomocí Flasku. Tyto metody nesou název home, project_home a login.

Dále jsou součástí skriptu pomocné metody. Těchto metod je aktuálně 7. Jedná se o metody, které agregují nebo jinak upravují data získaná z API nástrojů.

- **get_project_names_list(projects)** – jedná se o metodu, která z objektů třídy gitlab.base.project získá seznam názvů projektů. Vstup je seznam projektů (tedy objektů třídy gitlab.base.project). Výstup je seznam textových řetězců – názvů projektů.
- **get_pipelines_object(project, pipelines)** – jedná se o metodu, která ze seznamu objektů gitlab.base.pipeline agreguje data a vytvoří modifikovaný seznam slovníků s upravenými daty. Vstup je objekt project třídy gitlab.base.project a seznam objektů třídy gitlab.base.pipeline. Výstup je seznam slovníků s agregovanými daty.
- **contains(collection, object)** – jedná se o pomocnou metodu, která zkontroluje, zda objekt stejného jména jako object je v kolekci. Pokud se nachází v kolekci, je nahrazen. Vstup je kolekce a hledaný objekt. Výstup je boolean True v případě, že objekt byl nalezen, jinak False.
- **get_app_names(applications)** – jedná se o metodu, která ze seznamu služeb a verzí získá seznam služeb. Vstup je seznam služeb a verzí ve formátu služba:verze. Výstup je seznam služeb.
- **get_app_versions(applications)** – jedná se o metodu, která ze seznamu služeb a verzí vytvoří slovník. Vstup je seznam služeb a verzí ve formátu služba:verze. Výstup je slovník, kde klíč je služba a hodnota je verze.
- **get_apps_object(project, applications)** – jedná se o metodu, která ze seznamu názvů služeb vytvoří obohacený seznam služeb. Vstup je objekt třídy gitlab.base.project a seznam názvů služeb. Výstup je seznam slovníků, které reprezentují jednotlivé služby. Slovníky obsahují informace ohledně dostupných a nasazených verzí.
- **get_app_versions_available(application)** – jedná se o pomocnou metodu, která získá z Artifactory seznam dostupných verzí pro danou službu. Vstup je název služby. Výstup je seznam (List) verzí.

Přihlášení (funkce login)

Stránka přihlášení je renderována pomocí Flasku z šablony login.html, která je napsána v Jinja šablonovacím jazyku.
Stránka přihlášení je pouze jednoduchý formulář, kde je třeba vyplnit jméno a heslo. Poté jsou odeslány dotazy na API využívaných nástrojů a získány tokeny. V případě, že některý token získán není přihlášení končí neúspěšně a uživatel je přesměrován zpět na stránku přihlášení obohacenou o chybovou hlášku.
Po získání tokenů jsou tokeny uloženy do globálních proměnných. Poté se

nastaví session s trváním jedné hodiny – tento čas je zvolen i z důvodu, že jedna hodina je maximální doba trvání tokenu pro Artifactory, aniž by bylo nutné ho aktualizovat. Dále se nastaví na session hodnota ”logged_in” = True. Tím se umožní uživateli přístup na ostatní stránky nástroje, které jsou do té doby chráněné pomocí wrapperu login_required, který kontroluje přítomnost hodnoty ”logged_in” v session.
Po těchto operacích je uživatel přesměrován na úvodní stránku.
Tato funkce nemá vstup. Výstupem je renderovaná šablona login.html s parametrem error nebo přesměrování na jinou stránku.

Úvodní stránka (funkce home)

Úvodní stránka je renderována pomocí Flasku z šablony introduction.html, která je napsána v šablonovacím jazyku Jinja. Tato šablona dědí z index.html, která je zároveň rodičem i stránky aplikace. Pro renderování jsou do šablony předány parametry project_name a projects.
Nachází se na kořeni cesty (tedy /). Na úvodní stránce se nejdříve vytvoří spojení s GitLab API a získá se seznam dostupných projektů. Poté je uživatel vyzván k volbě projektu, který chce spravovat. Tato volba je hlídána jednoduchým JavaScriptovým skriptem, který po vybrání projektu přesměruje uživatele na stránku projektu.
Dále úvodní stránka nabízí krátký popis funkcí nástroje.
Tato funkce nemá vstup. Výstupem je renderovaná šablona introduction.html s parametry project_name a projects.

Stránka aplikace (funkce project_home(project_name))

Stránka aplikace je renderována pomocí Flasku z Jinja šablony project.html. Jak již bylo zmíněno u Úvodní stránky (kapitola 4.4.4), tato šablona dědí z index.html.
Stránky aplikace jsou hlavními stránkami celého nástroje. Po získání dat z API je stránka vyrenderována a zobrazí uživateli data, která jsou získána. Implementace na straně Pythonu je šablona pro libovolný projekt. Cesta k jednotlivým projektům je /název-projektu. Tato stránka reaguje na POST requesty od uživatele voláním metod spuštění pipeline nebo jobu, podle obsahu formuláře v requestu.
V případě post requestu obsahující “manual” se jedná o žádost o spuštění jobu, jehož id je získáno z formuláře a následně je spuštěna metoda *start_pipeline_job()* z GitLab API. V opačném případě se jedná o žádost spuštění nové pipeline, a po získání verzí z formuláře je zavolána metoda

create_new_pipeline() na GitLab API.
Pro získání a úpravu dat jsou v této metodě volány pomocné metody a metody z GitLab API zmíněné výše, které připraví data pro použití v šabloně.
V případě této práce je využívána pouze větev master.
Vstupem této funkce je název projektu. Výstupem je renderovaná šablona project.html s parametry project_name, projects, apps a pipelines.

Kapitola 5

Diskuze

Tato kapitola se zabývá diskuzí nad slabými stránkami této práce a možnostmi vylepšení celé práce. Tato kapitola má své opodstatnění, jelikož je nástroj a celý systém pouze ve formě prototypu, jsou přítomny nedostatky, které by bylo vhodné do budoucna vyřešit. Implementovaný systém bude vhodné rozšířit zejména o další možnosti testování jednotlivých služeb a celé aplikace.

Aplikace FIMS3, na které je v současné době projekt implementován se může jevit spíše jako monolitická aplikace využívající pouze backend a frontend. V tomto případě se ale jednalo o testovací projekt, na kterém se řešení navržené v této práci testovalo. V současné době je navíc ve vývoji třetí služba s názvem Jira Connector, která se do tohoto projektu připojí, což z něj udělá "více" mikroslužbovou aplikaci. Toto řešení lze také využít na monolitické aplikace.

V předchozích kapitolách bylo zmíněno, že testování celé aplikace se bude provádět hned na několika místech. V první řadě se bude testovat každá služba samostatně. Dále se provede sada automatických integračních testů a následovat bude manuální a případně i automatické testování na testovacím projektu.

5.1 Testování jednotlivých služeb

Aktuálně je v této práci navrženo a implementováno CI pro služby, které počítá s jednoduchou a krátkou sadou automatických testů, které se spustí pro každou větev a tím vývojáři během několika málo minut potvrdí, že jeho kód žádné testy nerozbil. Dalším testem je už ale pouze kontrola, zda se vytvořený Docker image chová očekávaně po znovuspuštění.

Pro větve, které jsou označeny jako nasazovací (v našem případě se jedná o master a release/verze), je vhodné přidat krok, který spustí kompletní sady testů připravené pro danou službu, a v případě, že je to vyžadováno, i krok na potvrzení manuálních testů. Implementace těchto kroků do CI by neměla být náročná, jelikož je možné využít kroky, které jsou již naimplementované, a mírně je upravit. Kompletní sady automatických testů by měly zahrnovat unit testy a následně další případné sady komplikovanějších testů, které by měly ověřit kompletní funkčnost služby jako celku, a ne jen jednotlivých metod, jak je to v případě jednotkových testů. Specifikace těchto testů by měla vycházet z dokumentace a jejich komplexnost a počet by měl odpovídat rizikům a případným škodám, které mohou vzniknout.

5.2 Testování celé aplikace

Testování integrace služeb vzájemně je stejně důležité nebo důležitější než testování služeb jednotlivě. V této práci je navržen krok automatických integračních testů, který předchází jakémukoli nasazení na prostředí. V tomto kroku by měly být odhaleny možné integrační problémy a jelikož se jedná o velmi důležitý krok, neměl by čas hrát velkou roli. Samozřejmě není vhodné, aby zde integrační testy běžely dlouhou dobu, a zatěžovaly tím zdroje build serveru (runneru). Jelikož následuje nasazení na testovací prostředí, měly by se v tomto kroku odhalit zejména problémy, které by mohly zabránit úspěšnému nasazení na testovací prostředí.

Po nasazení na testovací prostředí by měly proběhnout kompletní a důkladné automatické i manuální testy. Jak již bylo zmíněno u testování jednotlivých služeb, specifikace integračních testů by měla také vycházet z dokumentace. Testování by mělo odpovídat rizikům a škodám, které mohou vzniknout. Obecně známou pravdou o testování je, že je nemožné odhalit všechny problémy v aplikaci. Navržené testy by tedy měly odhalit pokud možno co nejvíce chyb a zejména kritických chyb pro běh aplikace.

5.3 Budoucí úpravy a vylepšení

Možností vylepšení a rozšíření této práce je celá řada, jelikož je aktuálně nástroj ve formě prototypu. Aktuální řešení přináší problém například se zabraným místem na disku archivem Docker obrazů, které jsou v této chvíli zbytečně velké. Cílem dalšího rozvoje tedy je zmenšit velikost základních obrazů Dockeru a případně si Dockerfile definovat od počátku (tedy bez využití dědění z jiných Docker image), pro větší kontrolu nad instalovanými verzemi a nástroji.

V tuto chvíli využívá tato práce velké množství dotazů na API. Pro zrychlení nástroje pro správu a ušetření prostředků serverů by bylo vhodné snížit množství dotazů – to lze udělat například využitím cache pro ukládání výsledků. Další možností je využití databáze, která by spravovala výsledky a byla nástrojem pravidelně plněna. To by s sebou ale přineslo potřebu řešení zabezpečení přístupů k jednotlivým položkám v databázi.

Dalšími, méně důležitými vylepšeními, jsou podpora 2-faktorového ověření vůči externím nástrojům (zejména GitLab) a vylepšení integrace nástroje s Graylogem.

Kapitola 6

Závěr

Cílem práce bylo pro mikroslužbovou architekturu navrhnout a implementovat proces a nástroje, které by usnadnily verzování a přehled o prostředích, které využívají mikroslužeb. Bylo také potřeba prozkoumat možnosti centralizace logů a detekce chyb v nich. Dále bylo cílem poskytnout vývojářům co možná nejlepší podporu při vývoji jednotlivých služeb a při opravách chyb, které mohou vzniknout integrací. Cíle vytyčené v úvodu práce byly splněny.

V úvodu práce je postupně zkoumána problematika mikroslužbové architektury a agilního vývoje. Následně je práce zaměřena na CI/CD, možnosti analýzy logů a existující nástroje k tomu využívané, a nakonec jsou popsány možnosti prostředí kontejnerů.

Druhá část práce se zaměřuje na konkrétní návrh verzování a nasazování v mikroslužbové architektuře, kde byly jako řešení vybrány obrazy a kontejnery Dockeru, které umožňují verzování celých kontejnerů pomocí tagů, a nástroj Docker Compose pro správu celé aplikace. Je navrženo CI a CD jednotlivých služeb i aplikace jako celku, tento návrh je zaměřený nejen na rychlou zpětnou vazbu a podporu vývojáře, ale také na důkladné testování. Pro centralizaci logů a detekci chyb v prostředí a aplikaci je využit nástroj Graylog v opensource verzi s emailovými notifikacemi. Následně je navržen jednoduchý nástroj, který umožňuje lepší správu a poskytne centrální přehled o jednotlivých prostředích. Implementace je provedena z části formou šablon, které lze snadno aplikovat na další projekty, a konkretizována na projektu společnosti Quanti s.r.o.

Výsledkem této práce jsou šablony GitLab CI pro projekt i službu, které lze snadno aplikovat na vybraný projekt. Dále byl vytvořen nástroj, který

lze využít pro projekt využívající CI/CD navržené v této práci. Součástí zmíněného nástroje byla vytvořena rozhraní pro získání dat z Artifactory, GitLabu a Graylogu.
Bylo počítáno s možností přidání další služby do vybraného projektu a proto je v tomto ohledu práce snadno rozšiřitelná.
Dalšími konkrétními výsledky jsou upravené CI pro projekt FIMS3 a archiv verzí jednotlivých služeb v Artifactory od doby zavedení nového CI.

Navržený a implementovaný systém je využíván pro vývoj ve společnosti Quanti s.r.o. na testovacím prostředí na projektu FIMS3. Projekt potvrdil přínos pro jednotlivé vývojáře v integraci a spuštění celého projektu v lokálním prostředí. Vzhledem k tomu, že celý projekt je aktuálně ve vývoji, nelze potvrdit ani vyvrátit všechny přínosy této práce. Projeveným nedostatkem je velikost Docker obrazů jednotlivých aplikací, kterou bych chtěl do budoucna zredukovat, a dlouhý čas načítání stránky aplikace v nástroji pro správu v případě, že projekt obsahuje velké množství úloh CI v GitLabu, z důvodu mnoha dotazů na API.

Práce na projektu pro mě osobně byla velmi přínosná. Umožnila mi hlouběji nahlédnout do zajímavých technologií (Docker, Podman, agregátory logů) a dozvědět se mnoho nových věcí. Dále mi umožnila se lépe seznámit s jazykem Python a vyzkoušet modul Flank, který se jeví jako velmi užitečný nástroj pro UI.

Literatura

[1] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley, Pearson, 2017.

[2] James A. Highsmith. *Agile software development ecosystems*. Addison-Wesley, 2006.

[3] What are microservices? `https://www.redhat.com/en/topics/microservices/what-are-microservices`. [Online, cit. 15.05.2020].

[4] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, AIMC '15, page 19–24, New York, NY, USA, 2015. Association for Computing Machinery.

[5] Martin Fowler. Microservices guide. `https://martinfowler.com/microservices/`. [Online, cit. 12.03.2020].

[6] Martin Fowler and James Lewis. Microservices. `https://martinfowler.com/articles/microservices.html`. [Online, cit. 15.04.2020].

[7] Chris Richardson. What are microservices? `https://microservices.io/`. [Online, cit. 15.04.2020].

[8] Dave Farinelli. Api vs. microservices: A microservice is more than just an api. `https://www.scalyr.com/blog/api-vs-microservices/`, Mar 2019. [Online, cit. 20.04.2020].

[9] Chris Richardson. Microservices pattern: Microservice architecture pattern. `https://microservices.io/patterns/microservices.html`. [Online, cit. 15.04.2020].

[10] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[11] J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[12] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[13] D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5, 2016.

[14] T. Ueda, T. Nakaike, and M. Ohara. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[15] Gregor Hohpe. Starbucks does not use two-phase commit. `https://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html`, Nov 2004. [Online, cit. 25.04.2020].

[16] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[17] Martin Fowler. Monolith first. `https://martinfowler.com/bliki/MonolithFirst.html`, Jun 2015. [Online, cit. 18.03.2020].

[18] Chris Richardson. Microservices pattern: Monolithic architecture pattern. `https://microservices.io/patterns/monolithic.html`. [Online, cit. 15.04.2020].

[19] Martin Fowler and Jim Highsmith. The agile manifesto. `http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf`, Aug 2001. [Online, cit. 25.04.2020].

[20] A. Cockburn and J. Highsmith. Agile software development, the people factor. *Computer*, 34(11):131–133, 2001.

[21] Manifest agilního vývoje software. `https://agilemanifesto.org/iso/cs/manifesto.html`. [Online, cit. 25.04.2020].

[22] Martin Fowler. Agile software guide. `https://martinfowler.com/agile.html`, Aug 2019. [Online, cit. 25.04.2020].

[23] Mike Cohn. *Agile estimating and planning*. Prentice Hall PTR, 2012.

[24] Geoffrey Wiseman. Do agile methods require documentation? `https://www.infoq.com/news/2007/07/agile-methods-documentation/`, Jul 2007. [Online, cit. 18.05.2020].

[25] The agile fluency model. https://martinfowler.com/articles/agileFluency.html. [Online, cit. 05.05.2020].

[26] Ondřej Macek and Martin Komárek. Životní cyklus sw projektu, přehled metodik. https://moodle.fel.cvut.cz, Oct 2016.

[27] What is scrum? https://www.scrum.org/resources/what-is-scrum. [Online, cit. 25.04.2020].

[28] Ken Schwaber and Jeff Sutherland. Průvodce scrumem. https://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-CS.pdf. [Online, cit. 25.04.2020].

[29] Jiří Knesl. Agilní vývoj: Scrum. https://www.zdrojak.cz/clanky/agilni-vyvoj-scrum, Dec 2009. [Online, cit. 25.04.2020].

[30] Scrum framework. https://s3.amazonaws.com/scrumorg-website-prod/drupal/2016-06/ScrumFramework_17x11.pdf. [Online, cit. 20.05.2020].

[31] The agile fluency model. http://blog.scrumstudy.com/the-daily-scrum-update-using-scrum-board/. [Online, cit. 20.05.2020].

[32] What is extreme programming (xp)? https://www.agilealliance.org/glossary/xp, Sep 2019. [Online, cit. 25.04.2020].

[33] Don Wells. Extreme programming: A gentle introduction. http://www.extremeprogramming.org/, Oct 2013. [Online, cit. 25.04.2020].

[34] Rachaelle Lynn. What is fdd in agile? https://www.planview.com/resources/articles/fdd-agile, Mar 2020. [Online, cit. 25.04.2020].

[35] Scott W. Ambler. Feature driven development (fdd) and agile modeling. http://agilemodeling.com/essays/fdd.htm. [Online, cit. 25.04.2020].

[36] What is ci/cd? https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Online, cit. 15.04.2020].

[37] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous integration improving software quality and reducing risk*. Addison-Wesley, 2013.

[38] Martin Fowler. Continuous integration. https://martinfowler.com/articles/continuousIntegration.html. [Online, cit. 20.03.2020].

[39] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley, 2010.

[40] Martin Fowler. Continuous delivery. https://martinfowler.com/bliki/ContinuousDelivery.html, May 2013. [Online, cit. 11.05.2020].

[41] Christian Meléndez. What is cicd? what's important and how to get it right. https://stackify.com/what-is-cicd-whats-important-and-how-to-get-it-right, Apr 2020. [Online, cit. 12.05.2020].

[42] Martin Fowler. Blue green deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html, Mar 2010. [Online, cit. 11.05.2020].

[43] Mikuláš Bc. Dítě. *Hloubkové porovnání CI/CD systémů*. Magisterská práce, Katedra softwarového inženýrství, ČVUT FIT, 2019. Vedoucí práce Ing. Tomáš Vondra, Ph.D.

[44] Ci/cd pipelines. https://docs.gitlab.com/ee/ci/pipelines/. [Online, cit. 20.05.2020].

[45] Devin Nusbaum. Jenkins pipeline stage result visualization improvements. https://www.jenkins.io/blog/2019/07/05/jenkins-pipeline-stage-result-visualization-improvements/. [Online, cit. 20.05.2020].

[46] Build stages. https://docs.travis-ci.com/user/build-stages/. [Online, cit. 20.05.2020].

[47] Teamcity: the hassle-free ci and cd server by jetbrains. https://www.jetbrains.com/teamcity/. [Online, cit. 20.05.2020].

[48] Anton Arhipov. Build chains: Teamcity's blend of pipelines. part 2 – running builds in parallel. https://blog.jetbrains.com/, Oct 2019. [Online, cit. 20.05.2020].

[49] Log aggregation: What is it and how does it help you? https://www.scalyr.com/blog/log-aggregation-help/, Mar 2019. [Online, cit. 01.05.2020].

[50] Christian Meléndez. Learn microservices logging best practices. https://www.scalyr.com/blog/microservices-logging-best-practices/, Apr 2019. [Online, cit. 20.05.2020].

[51] Josep Brugués i Pujolràs. Design and simulation of an elk-based logging infrastructure. https://ddd.uab.cat/record/196880, Oct 2018. [Online, cit. 01.05.2020].

[52] Aleksandr Alekseev, Tatiana Korchuganova, and Siarhei Padolski. The BigPanDA self-monitoring alarm system for ATLAS. Technical Report ATL-SOFT-PROC-2018-054, CERN, Geneva, Dec 2018.

[53] What is the elk stack? https://www.elastic.co/what-is/elk-stack. [Online, cit. 01.05.2020].

[54] We opened x-pack. https://www.elastic.co/what-is/open-x-pack. [Online, cit. 01.05.2020].

[55] Ritu Bhargava. Best of 2018: Log monitoring and analysis: Comparing elk, splunk and graylog. https://devops.com/, Mar 2020. [Online, cit. 20.04.2020].

[56] What is elasticsearch? https://www.elastic.co/what-is/elasticsearch. [Online, cit. 01.05.2020].

[57] Logstash: Collect, parse, transform logs. https://www.elastic.co/logstash.

[58] What is kibana? https://www.elastic.co/what-is/kibana. [Online, cit. 01.05.2020].

[59] Beats: Data shippers for elasticsearch. https://www.elastic.co/beats. [Online, cit. 01.05.2020].

[60] Community beats. https://www.elastic.co/guide/en/beats/libbeat/current/community-beats.html. [Online, cit. 01.05.2020].

[61] What's inside: Graylog features. https://www.graylog.org/features. [Online, cit. 02.05.2020].

[62] Introduction to graylog sidecar. https://www.graylog.org/features/sidecar. [Online, cit. 20.05.2020].

[63] Enterprise log management for any scale. https://www.graylog.org/products/enterprise. [Online, cit. 02.05.2020].

[64] Frequently asked questions¶. https://docs.graylog.org/en/3.2/pages/faq.html. [Online, cit. 02.05.2020].

[65] What is splunk - splunk meaning and splunk architecture. https://intellipaat.com/blog/what-is-splunk, Feb 2020. [Online, cit. 02.05.2020].

[66] Why splunk. https://www.splunk.com/en_us/about-us/why-splunk.html. [Online, cit. 02.05.2020].

[67] What is splunk? a beginners guide to understanding splunk. https://www.edureka.co/blog/what-is-splunk, May 2019. [Online, cit. 02.05.2020].

[68] Machine data management analytics: Splunk enterprise. https://www.splunk.com/en_us/software/splunk-enterprise.html. [Online, cit. 02.05.2020].

[69] Cloud-based service and log management for collecting, analyzing and visualizing machine data: Splunk cloud. https://www.splunk.com/en_us/software/splunk-cloud.html. [Online, cit. 02.05.2020].

[70] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. https://www.linuxjournal.com/. [Online, cit. 10.05.2020].

[71] C. Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[72] T. Combe, A. Martin, and R. Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.

[73] What is a container? https://www.docker.com/resources/what-container. [Online, cit. 10.05.2020].

[74] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[75] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[76] William Henry. Podman and buildah for docker users. https://developers.redhat.com/blog/2019/02/21/podman-and-buildah-for-docker-users/, Feb 2019. [Online, cit. 12.05.2020].

[77] Why docker? https://www.docker.com/why-docker. [Online, cit. 10.05.2020].

[78] A. Sheka, A. Bersenev, and V. Samun. The problem of reprodicible results on the hpc cluster. In *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, pages 0833–0837, 2019.

[79] Docker engine overview. https://docs.docker.com/engine, May 2020. [Online, cit. 10.05.2020].

[80] Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. [Online, cit. 20.05.2020].

[81] Overview of docker compose. https://docs.docker.com/compose/, May 2020. [Online, cit. 10.05.2020].

[82] Frequently asked questions. https://docs.docker.com/compose/faq/, May 2020. [Online, cit. 10.05.2020].

[83] Use compose in production. https://docs.docker.com/compose/production/, May 2020. [Online, cit. 10.05.2020].

[84] Docker machine overview. https://docs.docker.com/machine/overview, May 2020. [Online, cit. 10.05.2020].

[85] Docker desktop overview. https://docs.docker.com/desktop, May 2020. [Online, cit. 10.05.2020].

[86] What is kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes. [Online, cit. 10.05.2020].

[87] What is podman? simply put: 'alias docker=podman'. https://podman.io/whatis.html, Nov 2019. [Online, cit. 12.05.2020].

[88] Replacing docker with podman - power of podman. https://cloudnweb.dev/2019/06/replacing-docker-with-podman-power-of-podman. [Online, cit. 12.05.2020].

[89] Containers. containers/libpod. https://github.com/containers/libpod, May 2020. [Online, cit. 12.05.2020].

[90] Cri-o - lightweight container runtime for kubernetes. https://cri-o.io. [Online, cit. 12.05.2020].

[91] Containers. containers/buildah. https://github.com/containers/buildah, May 2020. [Online, cit. 12.05.2020].

[92] Why red hat is investing in cri-o and podman. https://www.redhat.com/en/blog/why-red-hat-investing-cri-o-and-podman. [Online, cit. 12.05.2020].

[93] Containers. containers/podman-compose. https://github.com/containers/podman-compose, May 2020. [Online, cit. 12.05.2020].

[94] A specification for building apis in json. https://jsonapi.org. [Online, cit. 21.05.2020].

[95] docker - dockerhub. https://hub.docker.com/_/docker. [Online, cit. 21.05.2020].

[96] Configure your node.js applications. https://www.npmjs.com/package/config, Mar 2020. [Online, cit. 20.05.2020].

[97] Welcome to flask¶. https://flask.palletsprojects.com/en/1.1.x/. [Online, cit. 20.05.2020].

[98] Fdd process diagram. https://upload.wikimedia.org/wikipedia/commons/9/99/Fdd_process_diagram.png. [Online, cit. 22.05.2020].

Příloha A

Dockerfile obrazu docker_build

```
FROM docker:dind

ENV LANG=C.UTF-8

RUN
     ALPINE_GLIBC_BASE_URL="https://github.com/sgerrand/alpine-pkg-glibc/releases/download"
     && \
    ALPINE_GLIBC_PACKAGE_VERSION="2.27-r0" && \
    ALPINE_GLIBC_BASE_PACKAGE_FILENAME="glibc-$ALPINE_GLIBC_PACKAGE_VERSION.apk"
        && \
    ALPINE_GLIBC_BIN_PACKAGE_FILENAME="glibc-bin-$ALPINE_GLIBC_PACKAGE_VERSION.apk"
        && \
    ALPINE_GLIBC_I18N_PACKAGE_FILENAME="glibc-i18n-$ALPINE_GLIBC_PACKAGE_VERSION.apk"
        && \
    apk add --no-cache --virtual=.build-dependencies wget ca-certificates && \
    wget \
        "https://alpine-pkgs.sgerrand.com/sgerrand.rsa.pub" \
        -O "/etc/apk/keys/sgerrand.rsa.pub" && \
    wget \
        "$ALPINE_GLIBC_BASE_URL/$ALPINE_GLIBC_PACKAGE_VERSION/$ALPINE_GLIBC_BASE_PACKAGE_FILENAME"
            \
        "$ALPINE_GLIBC_BASE_URL/$ALPINE_GLIBC_PACKAGE_VERSION/$ALPINE_GLIBC_BIN_PACKAGE_FILENAME"
            \
        "$ALPINE_GLIBC_BASE_URL/$ALPINE_GLIBC_PACKAGE_VERSION/$ALPINE_GLIBC_I18N_PACKAGE_FILENAME"
            && \
    apk add --no-cache \
        "$ALPINE_GLIBC_BASE_PACKAGE_FILENAME" \
        "$ALPINE_GLIBC_BIN_PACKAGE_FILENAME" \
        "$ALPINE_GLIBC_I18N_PACKAGE_FILENAME" && \
    \
    rm "/etc/apk/keys/sgerrand.rsa.pub" && \
    /usr/glibc-compat/bin/localedef --force --inputfile POSIX --charmap UTF-8
         "$LANG" || true && \
    echo "export LANG=$LANG" > /etc/profile.d/locale.sh && \
    \
    apk del glibc-i18n && \
    \
    rm "/root/.wget-hsts" && \
    apk del .build-dependencies && \
    rm \
        "$ALPINE_GLIBC_BASE_PACKAGE_FILENAME" \
```

```
        "$ALPINE_GLIBC_BIN_PACKAGE_FILENAME" \
        "$ALPINE_GLIBC_I18N_PACKAGE_FILENAME"

## Customization of https://github.com/arielkv/dind-glibc/blob/master/Dockerfile:
### install curl
RUN apk update && apk add curl curl-dev bash jq git
### install docker compose
RUN curl -L
     "https://github.com/docker/compose/releases/download/1.25.5/docker-compose-$(uname
     -s)-$(uname -m)" -o /usr/local/bin/docker-compose && \
    chmod +x /usr/local/bin/docker-compose && \
    ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose && \
    docker-compose --version

ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["sh"]
```

Listing A.1: Specifikace Docker image pro build

Příloha B

Přehled porovnání Elastic Stack, Graylog a Splunk

Tabulka B.1: Přehled porovnání Elastic Stack, Graylog a Splunk

	Elastic Stack	**Graylog**	**Splunk**
Licence	Opensource, Commercial, Cloud ,Enterprise (cena podle instance)	Opensource, Enterprise (zdarma pod 5GB/den)	Zdarma 500MB/den, Cloud, Enterprise
Implementace	Java	Java	C++, Python
Log format	JSON	GELF	JSON, CSV, Textfile
Support	Community, Enterprise	Community, Enterprise	Community, Enterprise
Podporované platformy	OS-X, Windows, RedHat Linux, CentOS, Ubuntu, Debian	OS-X, Windows, RedHat Linux, CentOS, Ubuntu, Debian	OS-X, Windows, RedHat Linux, CentOS, Ubuntu, Debian
Konfigurace	středně obtížná	středně obtížná	jednoduchá
Základní komponenty	Elasticsearch, Logstash, Kibana, Beats	Elasticsearch, Graylog, MongoDB	Splunk server, Splunk forwarder
Kolekce dat	Beats, Logstash	GELF TCP/UDP, Graylog sidecar	Splunk forwarder

Tabulka B.1: Přehled porovnání Elastic Stack, Graylog a Splunk

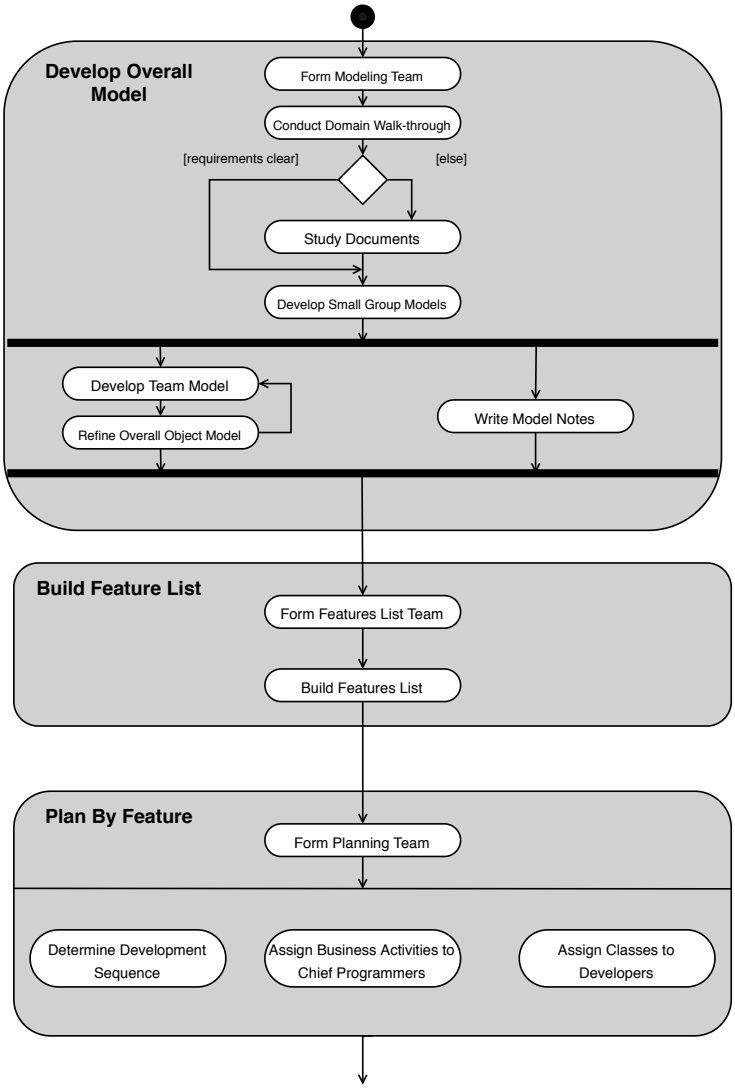
	Elastic Stack	**Graylog**	**Splunk**
Formát log inputů	běžné log formáty	běžné log formáty, syslog, rsyslog, GELF	CSV, JSON, běžné log formáty, XML
Další inputy	HTTP, TCP, syslog input	GELF Kafka, GELF HTTP, Beats, syslog input	HTTP, TCP, logstash plugin
Databáze	Elasticsearch	Elasticsearch, MongoDB	built-in
Centralizované logy	ANO – s log shippers	ANO – s log collector Graylog sidecar, rsyslog	ANO – s Splunk Enterprise forwarder, Universal forwarder
import/export dat	z různých zdrojů	GELF output plugin, REST API	pomocí Splung DB connect
transport dat	Kafka, RabbitMQ, Redis	Kafka, RabbitMQ	persistentní fronta, vnitřní komponenty
Interval kolekce dat	real-time, v dávkách	real-time, v dávkách	real-time, v dávkách
vyhledávání	full-text, analysis engine	full-text, intuitive search	dynamic data exploration
jazyk query	Query DSL (Lucene)	prakticky Lucene	SPL
Protokol na R/W operace	REST API, HTTP	REST API, HTTP	REST API, HTTP
Reporting	X-Pack rozšíření	built-in	built-in
Alerting	X-Pack rozšíření	built-in feature	built-in feature
monitoring – server, zařízení	metricbeat, filebeat	graylog beat, nxlog	Splunk insight for infrastructure
monitoring – network	Packetbeat	SNMP, netflow plugin	Splunk MINT
monitoring – cloud logs	logstash, filebeat, moduly	plugin	add-ons
monitoring – container logs	logstash, filebeat	filebeat, nativní graylog integrace	driver na docker
monitoring – kubernetes	fluentd	filebeat collector sidecar	Kubernetes collector

Tabulka B.1: Přehled porovnání Elastic Stack, Graylog a Splunk

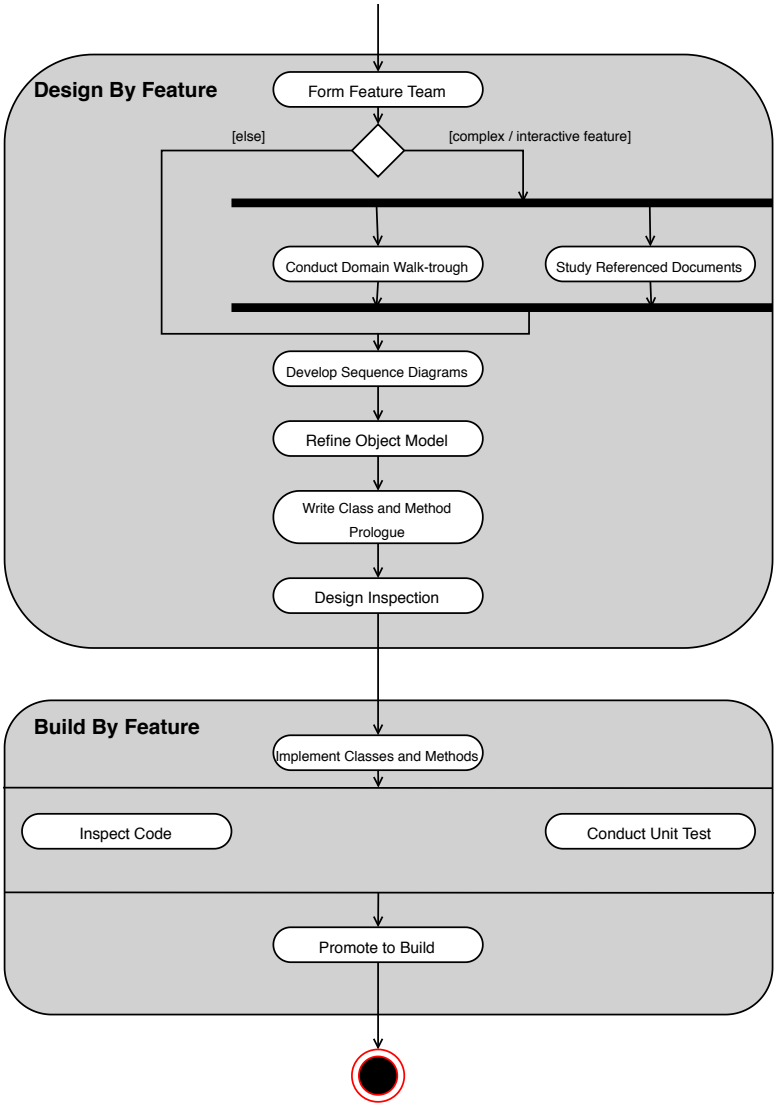
	Elastic Stack	**Graylog**	**Splunk**
monitoring – db	logstash, filebeat moduly	plugin, add-on	add-on, Splunk DB connect
monitoring – performance	APM, logstash, X-Pack rozšíření	add-on (marketplace)	Splunk app, Splunk MINT SDK
analytika	Machine learning X-Pack rozšíření	není	Splunk IT service intelligence
škálovatelnost	možnost clusterů	více nodů ES, mongodb, graylog server + kafka/rabbitmq	distribuovanost v Enterprise verzi
zálohování	snapshot indexů	archivační plugin v Enterprise verzi	zálohování konfigurace, indexů, db
integrace s jinými službami	webhooky, REST API, plugin	plugin, REST API,	webhooky, REST API, plugin
ostatní		user management integrace – např LDAP	out-of-the-box user management

Příloha C

Procesní diagram FDD



Obrázek C.1: Procesní diagram FDD[98] – část 1



Obrázek C.2: Procesní diagram FDD[98] – část 2

Příloha D

Požadavky pro spuštění

V této kapitole jsou popsány základní požadavky pro spuštění jednotlivých částí této práce.

D.1 CD/CI

Pro CI/CD je v této práci využíváno nástrojů GitLab a Artifactory. V GitLabu je používán Docker executor, který pro jednotlivé joby pipeline používá upravený Docker image "build_docker:v4". Artifactory je využíváno v Pro verzi – v opensource verzi ho nelze využít jako Docker registry.

D.2 Analýza logů

Pro analýzu logů je využit nástroj Graylog v opensource verzi. Na aplikačním serveru je nastaveno odesílání logů do Graylogu a v tomto nástroji je připraven stream s názvem projektu, který ukazuje pouze data z aplikačních serverů daného projektu.

D.3 Aplikace s uživatelským rozhraním

Pro spuštění nástroje pro správu mikroslužeb je třeba mít nainstalovaný Python3. Dále jsou vyžadovány následující moduly pro Python:

- artifactory
- ast
- asq
- json
- gitlab
- requests
- urllib
- bs4
- datetime
- grapi
- functools
- flask

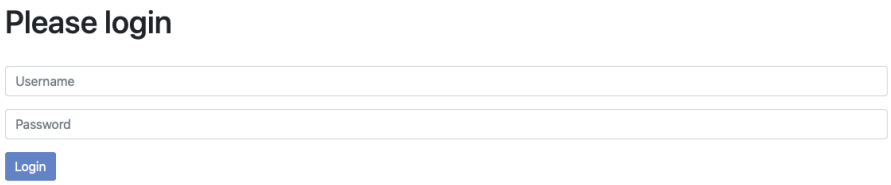
Všechny tyto moduly lze snadno nainstalovat pomocí "pip install <module>" nebo ve vývojovém prostředí.

Před spuštěním aplikace je třeba nastavit následující proměnné **GRAYLOG_URL, ARTIFACTORY_URL** a **GITLAB_URL**.

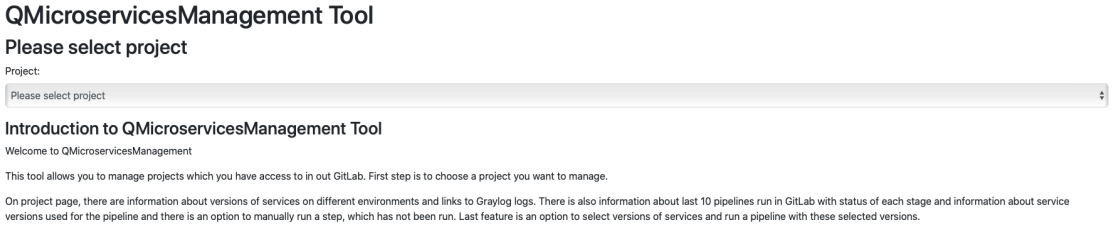
Spuštění aplikace se provádí příkazem "python3 app.py"a následně je nástroj dostupný na adrese "localhost:5000".

Příloha E

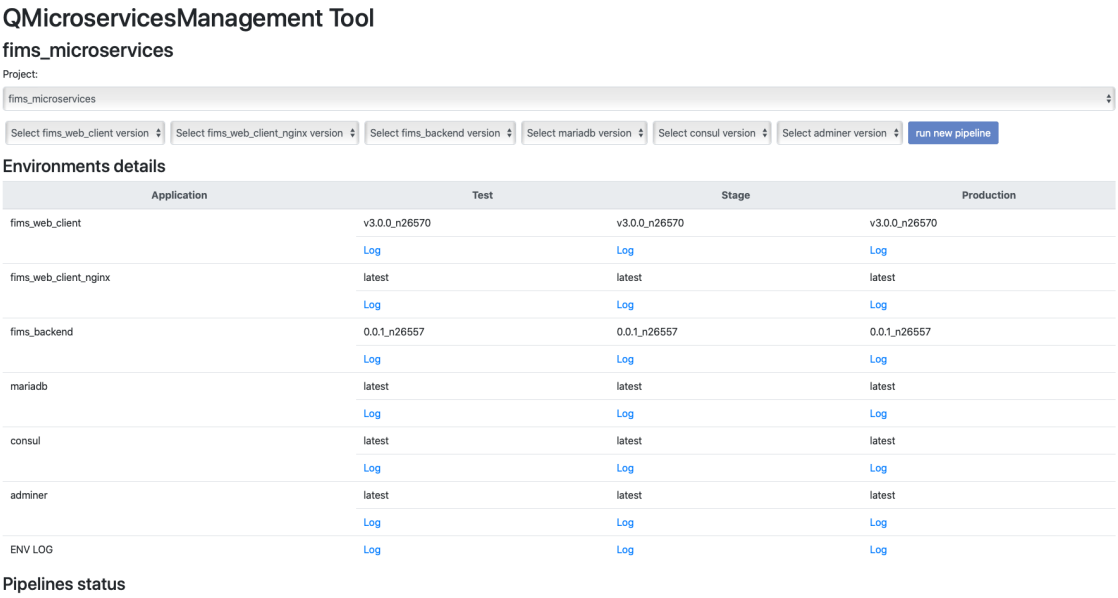
Ukázky z nástroje



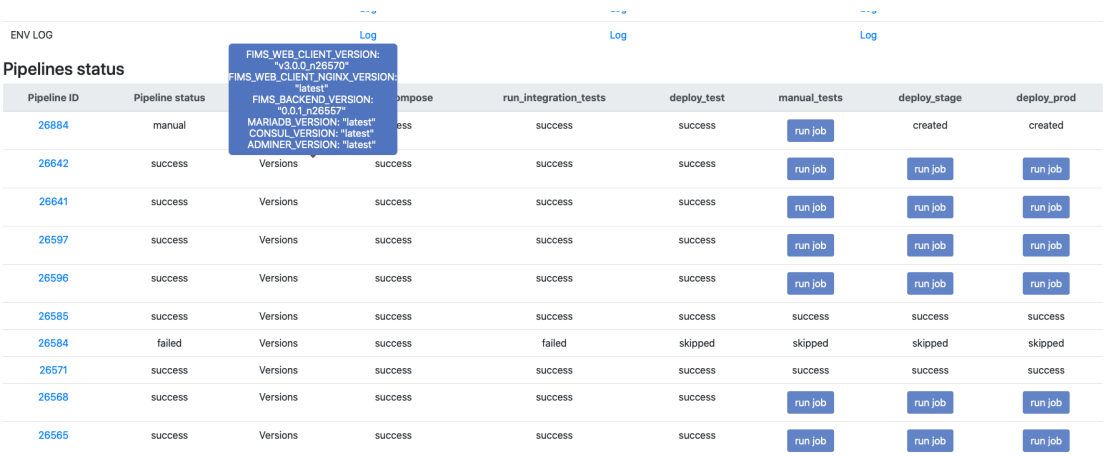
Obrázek E.1: Ukázka přihlašovací stránky do nástroje



Obrázek E.2: Ukázka úvodní stránky nástroje



Obrázek E.3: Ukázka stránky aplikace nástroje – část 1



Obrázek E.4: Ukázka stránky aplikace nástroje – část 2

Příloha F

Seznam použitých zkratek

API Application Programming Interface.

ARP Address Resolution Protocol.

AuFS Advanced Multi-Layered Unification Filesystem.

CI/CD Continuous Integration, Continuous Delivery.

CLI Command Line Interface.

CPU Central Processing Unit.

CRI-O Container Runtime Interface OCI.

DoS Denial of Service.

DSDM Dynamic Software Development Method.

DSL Domain-Specific Language.

ELK Elasticsearch, Logstash & Kibana.

ESB Enterprise Service Bus.

FDD Feature Driven Development.

FS Filesystem.

GELF Graylog Extended Log Format.

HTML HyperText Markup Language.

HTTP(S) Hyper Text Transfer Protocol (Secure).

HW Hardware.

IDE Integrated Development Environment.

JSON Java Script Object Notation.

LDAP Lightweight Directory Access Protocol.

LXC Linux Containers.

OCI Open Container Initiative.

OS Operating System.

RBAC Role Based Access Control.

REST REpresentational State Transfer.

SaaS Software as a Service.

SDK Software Development Kit.

SOA Service-Oriented Architecture.

SPL Search Processing Language.

SSO Single Sign-On.

SVN Subversion.

SW Software.

VM Virtual Machine.

WSGI Web Server Gateway Interface.

XML Extensible Markup Language.

XP Exreme Programming.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Svoboda** Jméno: **Vojtěch** Osobní číslo: **434745**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Verzování v mikroslužbové architektuře

Název diplomové práce anglicky:

Versioning in microservice architecture

Pokyny pro vypracování:

V poslední době se prosazuje mikroslužbová architektura pro backend procesy v rozsáhlých informačních systémech. Společně s tím je snaha o rychlý vývoj s krátkými testovacími cykly s co nejdřívějším dodání funkcionalit do produkčního prostředí (napomáhá tomu i agilní metodika SCRUM). Je potřeba mít kvalitní způsob nasazování mikroslužeb a vyřešený způsob verzování, aby bylo jednoduše zjistitelné, co je na jednotlivých prostředích nasazené, jaké zdrojové kódy jsou pro mikroslužby použity a jakou novou funkcionalitu přináší.
Je potřeba vymyslet vhodný způsob verzování jednotlivých mikroslužeb s přihlédnutím k mikroslužbové architektuře a tento proces automatizovat.
Cílem diplomové práce bude vytvoření nástroje pro modelování systému s mikroslužbami a jejich verzování.
Pro mikroslužby navrhněte vhodný systém
- detekce chyb a jejich centralizovaného zpracování
- určení služby a jejího prostředí, konfigurace aj. pro reprodukci chyby
- pro způsob testování opravené verze před nasazením do běžícího systému
- použití modelu systému, vytvoření kopie systému pro testování
Nástroj by mohl podporovat tvorbu statického modelu již běžícího systému.
Součástí práce by měla být i diskuze možností testování systému.

Seznam doporučené literatury:

[1] DUVALL, Paul M, Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk. Upper Saddle River, NJ: Addison-Wesley, [2007]. ISBN 978-0321336385.
[2] HUMBLE, Jez a David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. Upper Saddle River, NJ: Addison-Wesley, [2010]. ISBN 978-0321601919.
[3] Docker Documentation [online]. San Francisco: Docker [cit. 2020-01-20]. Dostupné z: https://docs.docker.com/
[4] WOLFF, Eberhard. Microservices: flexible software architecture. Boston: Addison-Wesley, [2017]. ISBN 0134602412.
[5] JARAMILLO, David, Duy V NGUYEN a Robert SMART. Leveraging microservices architecture by using Docker technology. In: SoutheastCon 2016 [online]. IEEE, 2016, 2016, s. 1-5 [cit. 2020-01-21]. DOI: 10.1109/SECON.2016.7506647. ISBN 978-1-5090-2246-5. Dostupné z: http://ieeexplore.ieee.org/document/7506647/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Martin Chloupek, Ph.D., Quanti s.r.o.

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.02.2020** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2021**

Ing. Martin Chloupek, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ _____

Datum převzetí zadání Podpis studenta

