Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Automatizace nasazování komplexní podnikové aplikace založené na architektuře mikroslužeb

Diplomová práce

Autor: Dominik Lohniský
Studijní obor: Aplikovaná Informatika

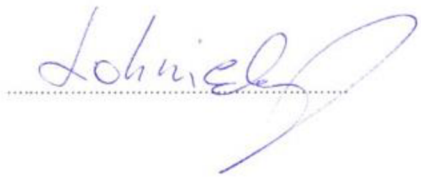
Vedoucí práce: Ing. Stanislav Mikulecký Ph.D.

Odborný konzultant: Jiří Dudek
Unicorn Systems a.s.

Hradec Králové duben 2023

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.



V Hradci Králové dne 25.4.2023 Dominik Lohniský

Poděkování:

Děkuji vedoucímu diplomové práce Ing. Stanislavu Mikuleckému, Ph.D za metodické vedení práce, ochotu a čas, který mi věnoval. Dále bych rád poděkoval Jiřímu Dudkovi z firmy Unicorn Systems za odborný dohled a užitečné rady.

Anotace

Cílem diplomové práce je identifikovat potřeby z oblasti nasazování komplexních podnikových aplikací založených na architektuře mikroslužeb, zmapovat možné způsoby řešení a nástroje dostupné na trhu a vyhodnotit jejich silné a slabé stránky. Na základě této rešerše poté navrhnout způsob nasazování konkrétní podnikové aplikace a implementovat funkční proof of concept.

Annotation

Title: Deployment automation of complex enterprise systems based on microservice architecture

The goal of this thesis is to identify the needs related to the deployment of complex business applications based on microservices architecture, pam possible solutions and tools available on the market, and evaluate their strengths and weaknesses. Based on this research, propose a way to deploy a specific business application and implement a functional proof of concept.

Obsah

Seznam obrázků

Seznam kódů

1 Úvod

Tato diplomová práce se zabývá automatizací komplexních podnikových aplikací založených na architektuře mikrslužeb s využitím moderních orchestračních technologií jako jsou Docker Swarm a Kubernetes. Jedná se o velmi využívané platformy dnešní doby, které přinášejí možnost izolace aplikací od konkrétní technologie či platformy.

V teoretické části práce jsou popsány předpoklady pro praktické využití těchto technologií. V první části je nastíněna zmíněná architektura mikroslužeb a jsou uvedeny její výhody oproti monolitické architektuře. Následuje přiblížení virtualizace mikroslužeb za pomoci kontainerizace a využitím již zmíněných orchestračních technologií. U obou technologií je podrobněji popsána jejich architektura a funkčnost.

Poslední kapitola teoretické části se zabývá pojmem Cloud Computing. Jedná se o jedno z nejvýznamnějších paradigmat současnosti v oblasti informačních technologií. Jsou zde popsány jeho hlavní charakteristiky, modely služeb a modely nasazování.

Cílem praktické části této diplomové práce je návrh a implementace automatizace pro kompletní nasazení celého informačního systému, který se skládá z více než třiceti mikroslužeb. Praktická část začíná popisem infrastruktury cílového prostředí, které je určené pro nasazení podnikové aplikace. Následuje detailní popis instalačních kroků, které je třeba vykonat v rámci nasazování. Každý dílčí krok je detailně popsán spolu s návrhem automatizace a následnou implementací:

- Inicializace infrastruktury aplikací
- Připravení nasazovacích konfigurací
- Nahrání knihoven do Nexus repositáře
- Nasazení aplikací
- Instalace aplikací (inicializace, nahrání dat)
- Nahrání frontendových knihoven do CDN

Praktická část je zakončena kapitolou s výsledky a testováním. Je zde popsán způsob testování jednotlivých dílčích kroků. Následně jsou doplněny ukázky výstupů automatizačních nástrojů a nově vzniklé prostředí s obchodním informačním systémem.

2 Cíl práce

Cílem diplomové práce je seznámení s problematikou v oblasti mikroslužeb a automatizace jejich nasazení s využitím jednoho z orchestračních nástrojů (Docker Swarm a Kubernetes). Cílem praktické části je pak navržení a implementace automatizace přípravy prostředí a vylepšení stávajícího nasazovacího procesu. Hlavním záměrem je navržení takového způsobu, který co nejlépe odstraní potřebu manuální práce. V rámci této diplomové práce vznikl zcela nový instalační proces, u kterého byly stanoveny hlavní cíle: vytvoření automatizace pro tvorbu infrastruktury a konfiguračních souborů aplikací, upravení způsobu nasazování aplikací pro testovací prostředí (vznik kroků pro aktualizaci verzí a stažení balíčků), vytvoření automatizačního nástroje pro nahrání nodejs knihoven a scriptů pro import frontendových knihoven do sítě CDN.

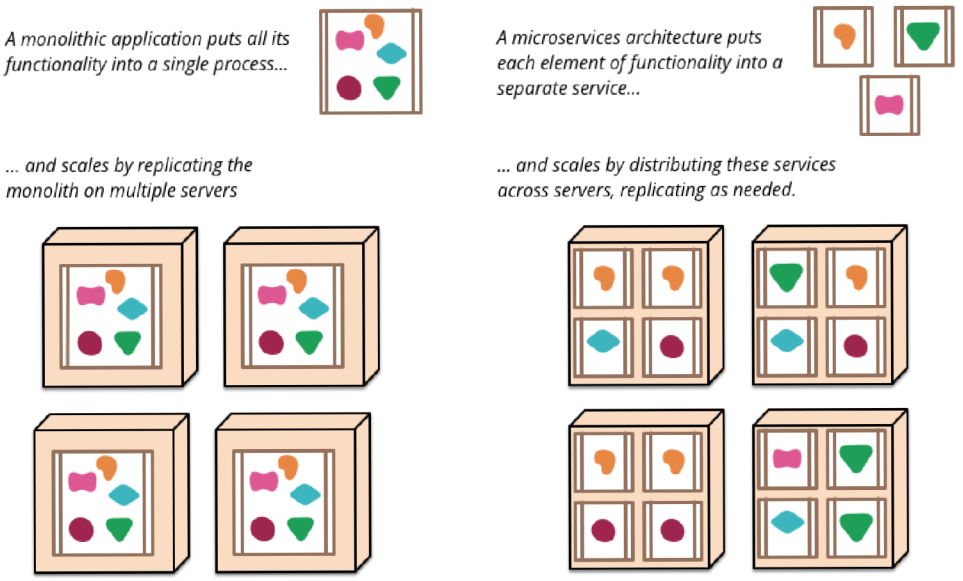
3 Architektura mikroslužeb

Pojmem mikroslužba je reprezentována jedna z variant architektury vývoje softwaru, která přistupuje k vývoji jedné aplikace jako k sadě menších služeb. Každá poté běží ve svém vlastním procesu a komunikuje s ostatními mikroslužbami přes síťové protokoly pomocí komunikačních mechanismů, jako jsou například REST API. Tyto služby jsou většinou postaveny okolo specifické obchodní funkcionality a lze je nasadit nezávisle na sobě. Existuje naprosté minimum centralizované správy těchto služeb, mohou být napsány v různých programovacích jazycích a používat například různé technologie pro persistenci dat [1, 2].

Mikroslužby se vyvinuly ze strategie vývoje aplikací orientované na služby. Jedna velká aplikace je tedy rozdělena do různých malých a nezávislých servisních jednotek, kde každá nabízí řešení pro odpovídající část celého řešení. V současné době mnoho společností (například Spotify, Amazon, Netflix, Unicorn, …) vyvinulo své aplikace právě s použitím této architektury. Mikroslužby mohou poskytovat významné výhody vývojem, navrhováním, realizací a testováním služeb s velkou agilitou [3]. Mezi nevýhody pak patří například složitější monitoring a správa, duplikace dat v různých servisech a větší složitost systémů.

3.1 Škálovatelnost

Dříve využívaná monolitická architektura aplikací se sebou nesla obrovský problém v oblasti škálování. Monolity nabízely veliké množství služeb, kde některé z nich byly populárnější než jiné. V případech, kde bylo nutné škálovat některou z často využívaných služeb, docházelo zároveň ke škálování celé sady, kterou monolit obsahoval. Tento přístup způsoboval zbytečnou alokaci velkého množství serverových zdrojů pro služby, které je nikdy nevyužily. Tento problém se stal jedním z hlavních důvodů, proč společnosti přecházejí k řešením založených na architektuře mikroslužeb. Výsledkem je zvýšení efektivity a umožnění snadného škálování pouze specifické části, kterou vyžaduje obchodní logika aplikace [4].



Obrázek 1 - Rozdíl škálování v architekturách [1]

4 Virtualizace pomocí kontainerů

Jedním velkým usnadněním, které přináší vývoj aplikací založených na mikroslužbách je virtualizace pomocí kontainerů (například pomocí Docker kontainerů) [5]. Pojmem kontainer je myšleno virtuální prostředí, které izoluje chod aplikace od její závislosti na okolí, včetně dalších aplikací a operačního systému hostitelského prostředí. Jsou vytvořeny pomocí virtualizace na úrovni operačního systému, kdy se aplikace a její závislosti spouštějí uvnitř jako samotné procesy. Kontainer může obsahovat vše, co potřebuje aplikace ke svému běhu, jako jsou knihovny, konfigurační soubory, binární soubory a další zdroje [6].

Kontainerizace je pak proces, kdy se mikroslužba a její závislosti zapouzdří do zmíněného kontaineru. Tento proces umožňuje rychle a snadno vytvářet, nasazovat a spravovat aplikace na různých prostředích, jako jsou vývojová, testovací a produkční prostředí. Kontainerizace také umožňuje rychle škálovat aplikace, když se potřebuje zvýšit nebo snížit výkon [6].

Kontainery jsou v poslední době velmi populární zejména díky orchestračním nástrojům, které mají za cíl zjednodušit správu aplikací založených na kontainerech v klastrovaných prostředích. Správa více ručně nasazených kontainerů by vyžadovala například monitorování jejich dostupnosti, případné restartování při závadě, použití pravidel škálování a řízení komunikace mezi jednotlivými prvky [5, 7]. Různé typy nástrojů pro orchestraci slouží k jiným účelům. Například Docker Compose je nástroj, který je používán pro definování a spouštění vícekontainerových aplikací s použitím YAML scriptů. Postrádá ale schopnost monitorování nebo clusteru a správu distribuovaných částí napříč sítěmi. Docker Swarm a Kubernetes jsou dvě hlavní technologie, které umí s těmito požadavky pracovat [8].

4.1 Docker swarm

První zmíněnou technologií pro orchestraci je engine, který byl vyvinut pod značkou Docker a používá se v takzvaném Swarm režimu. Tento orchestrátor byl vydán do produkčního prostředí v roce 2016. Jedná se o shlukovací a plánovací nástroj pro Docker kontainery. Vytváří kooperativní skupinu virtuálních strojů, které

poskytují redundanci a umožňují mechanismus převzetí služeb při selhání, pokud dojde k výpadku jednoho nebo více uzlů. Orchestrátor je založen na master/slave modelu. Master (manager) je uzel zodpovědný za plánování, zatímco slave (agent) je zodpovědný za spouštění přijatých kontainerů. Redundance i umístění jsou řízeny plánovací vrstvou. Pro vzájemné propojení kontainerů hostovaných na různých uzlech ve stejné síti vytváří Docker Swarm virtuální síň mezi hosty za pomoci VXLAN tunelu. Manager sleduje stav všech uzlů v clusteru s využitím takzvaného heartbeat mechanismu. Docker swarm také umožňuje škálovatelnost a replikaci jedné nebo více služeb [9]. Jednou z hlavních výhod je jeho přímá integrace s Docker API. Všechny funkce, které se vztahují na Docker kontainery lze také aplikovat ve Swarmu, což znamená snadné přenesení znalostí z Docker prostředí a výrazné zjednodušení správy infrastruktury [10].

4.1.1 Architektura

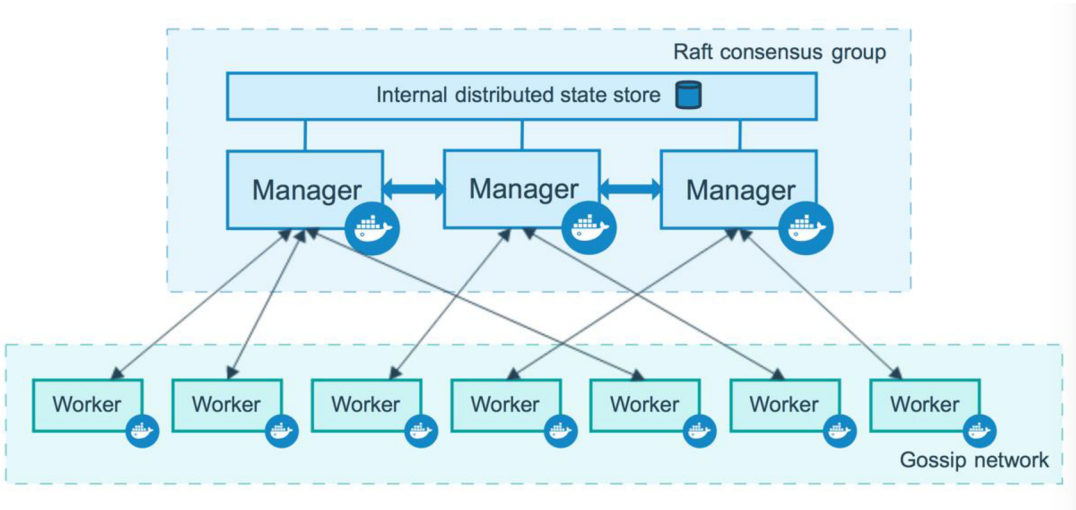
Za pomoci technologie Docker Swarm lze vytvářet a spravovat cluster kontainerů složených z komponent:

- **Task** – Kombinace jednoho Docker kontaineru s příkazy definujícími jeho spuštění a funkčnost.
- **Service** – Skládá se z jednoho nebo více tasků.
- **Raft Consensus Group** – Skládá se z interní databáze stavů a pracovních uzlů.
- **Internal Distributed State Store** – Zabudovaná databáze používaná ke správě stavů v clusteru (ve formátu klíč-hodnota).
- **Manager Nodes** – takzvaný manažerský uzel skládají se z částí:
    - **API** – Přijímá příkaz od uživatele a vytváří novou službu na základě vstupních parametrů.
    - **Orchestrator** – Vytváří task z definice služby.
    - **Allocator** – Přiděluji IP adresy.
    - **Scheduller** – Plánuje tasky a přiřadí je pracovním uzlům.
    - **Dispatcher** – Provádí kontrolu nad pracovními uzly.

- **Worker Nodes** – dostávají tasky přímo z vedoucího uzlu a následně je vykonávají. Tyto uzly také odesílají aktuální stav provedených úloh a informaci o svém vlastním stavu zpět do vedoucího uzlu. Tímto je zajištěna kontrola, zda je pracovní uzel stále aktivní a schopen přijímat nové úkoly.

Hlavní přidanou hodnotou technologie Docker Swarm je jeho odolnost vůči chybám. Díky přítomnosti více řídících uzlů v jednom kontainerovém clusteru, se dokáže zotavit z chyb, aniž by byl nucen přerušit svůj provoz. Pokud dojde k jakémukoliv výpadku vedoucího uzlu, je následně nahrazen jiným z uzlů řídicích, který provede zadané úlohy orchestrace [8, 10].
Manažerskému uzlu, který je zvolen jako vedoucí, je umožněno vykonávat následující úkoly:

- Příjem uživatelsky definovaných služeb
- Replikace úloh (probíhá na základě definice služby)
- Odeslání úloh do uzlu, který je bude vykonávat
- Provádění orchestrace a řízení kontainerového clusteru
- Vyvažování zátěže na vstupu (load balancing)
- Rozhodování ohledně plánování a distribuce úloh na základě monitorování aktuálního stavu clusteru



Obrázek 2 Architektura Docker Swarm [11]

4.2 Kubernetes

Druhou hlavní technologií na orchestraci kontainerů je Kubernetes. Tento orchestrační engine byl původně vyvíjen společností Google a jeho první verze jsou dostupné od ruku 2014 pro veřejné využití. O dva roky později byl darován společnosti Cloud Native Computing Foundation, která jej nadále rozvíjí.
Analogicky je Kubernetes také založen na architektonickém modelu master/slave, kde uživatel poskytne seznam aplikací hlavnímu uzlu a následně je platforma nasadí mezi slave a master uzly. Hlavní uzel představuje řídicí rovinu clusteru a lze jej replikovat pro zajištění vysoké dostupnosti a odolnosti vůči výpadku. Slave uzly spouštějí aplikační kontainery. Kubernetes poskytuje kontainerizovanou aplikaci jako sadu kontainerů, z nichž každý je specifický pro jednu mikroslužbu. Základní jednotka se nazývá Pod a představuje skupinu kontainerů, které jsou naplánovány společně. Všechny kontainery v podu jsou řízeny jako jedna aplikace, sdílejí stejné prostředí a jsou spouštěny ve sdíleném kontextu. Replikace podu je řízena komponentou s názvem Replication Controller, která zajišťuje, že určitý počet podů aktuálně poskytuje konkrétní službu. Pokud se aktuální stav odchyluje od očekávání (například kvůli výpadku uzlu), je automaticky spuštěno plánování nové instance na jiném slave uzlu [12].

4.2.1 Architektura

Architektonická stránka technologie Kubernetes se skládá z hlavních dvou částí, které jsou složeny z následujících komponent:

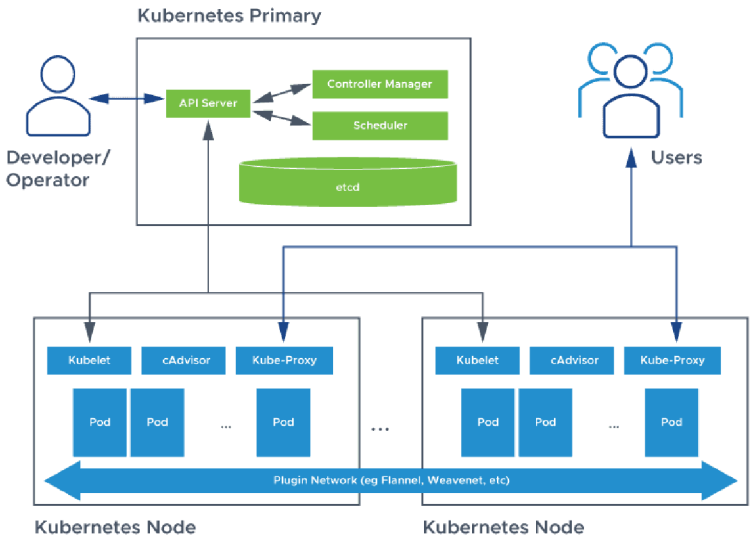
- **Master** – tuto komponentu lze chápat jako jádro Kubernetes, které představuje řídicí rovinu orchestrátoru. V tomto jádru jsou další klíčové komponenty, které se podílejí na orchestraci:
    - **API server** – centrální komponenta, která slouží jako API pro komunikaci s komponentami v clusteru. Pro navázání komunikace je nejčastěji používán nástroj kubectl, který s využitím příkazového řádku interaguje s API serverem a zasílá požadavky do Master uzlu.

- **Etcd** – komponenta uchovávající všechna konfigurační data ve formátu klíč-hodnota a zároveň slouží jako registr dat o tom, které kontainery aktuálně běží a kde jsou nasazeny.
- **Scheduler** – Dohlíží na dostupnost, kapacitu a výkon clusteru. Dále rozhoduje, na kterých uzlech jsou jednotlivé pody umístěny.
- **Controller Manager** – Monitoruje etcd přes API server a zároveň porovnává aktuální stav clusteru s požadovaným stavem. Skládá se z několika dalších kontrolerů, které běží jako samostatné procesy.
  - **Node Controller** – Je zodpovědný za správu, monitorování a stav všech uzlů v clusteru.
  - **Replication Controller** – Spravuje replikaci objektů v clusteru.
  - **Endpoint Controller** – Spravuje koncové body a zajišťuje, aby poskytované služby byly aktuální.
  - **Service Account and Token Controller** – Spravuje servisní účty a tokeny pro přístup k API.
  - **Cloud Controller –** Pokud je cluster částečně nebo zcela založen na cloudu, tento kontroler jej propojí s API rozhraním poskytovatele cloudové služby. Spustí se pouze ovládací prvky, které jsou pro cloud specifické. V clusterech, které jsou založeny na on-premis modelu tento kontroler neexistuje.

- **Node –** Pracovní uzly přijímají požadavky zaslané hlavním uzlem. Představují výpočetní výkon clusteru a jsou tedy místem, kde běží kontainery. Každý uzel obsahuje komponenty:
  - **Kubelet –** Hlavní služba na uzlu, která pravidelně přijímá nové nebo upravené specifikace pod (především prostřednictvím API serveru) a zajišťuje, že pody a jejich kontainery běží v požadovaném stavu. Tato komponenta také podává hlavnímu serveru zprávy o stavu hosta, na kterém běží.
  - **Kube-proxy –** Proxy služba, která běží na každém uzlu, aby se vypořádala s jednotlivými podsítěmi hosta a vystavila služby

vnějšímu světu. Provádí předávání požadavků do správných modulů či kontainerů napříč různými izolovanými sítěmi v clusteru.

- **Container Runtime** – Software zodpovědný za provoz kontainerových aplikací. Kubernetes podporuje jakoukoliv kontainer, který se řídí Kubernetes CRI (Container Runtime Interface).



Obrázek 3 Architektura Kubernetes [13]

Kubernetes, stejně jako Docker Swarm umožňuje přítomnost více Master uzlů v jednom clusteru (kvůli možnému selhání), ale může být pouze jeden aktivní v určitém období. Vždy tedy svoje úlohy vykonává pouze jeden kontroler a jeden plánovač. V případě selhání aktivního Master uzlu se aktivuje jeden z náhradních, který jej zastoupí.

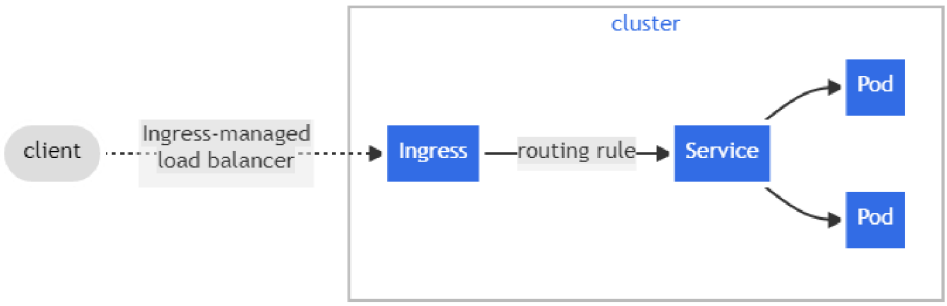
4.2.2 Koncept a infrastruktura

Kubernetes obsahuje specifický koncept abstrakce, která je využívána k reprezentaci stavu systému. Mezi základní infrastrukturní komponenty patří:

- **Pod** – Představuje nejnižší vrstvu v clusteru Kubernetes a obsahuje samotné kontainery. Jeden pod může obsahovat i několik kontainerů, které se používají jako skupina. Při vytváření clusteru kontainerů určuje, kolik zdrojů, CPU a paměti budou používat. Plánovač pak použije tato data

k rozhodnutí, na který uzel má pod umístit. Každému podu je při vytvoření přidělena vlastní IP adresa v clusteru, která je používaná pro jejich vzájemnou komunikaci. Nevýhodou však je, pokud dojde k výpadku, je pod nahrazen novou instancí s novou IP adresou.

- **Service** – Vzhledem k nestálosti podů Kubernetes nezaručuje, že daný fyzický pod zůstane zachován (Replication controller může z důvodu výpadku zabít a spustit novou sadu podů). Místo toho služba představuje logickou sadu podů a funguje jako brána, která umožňuje podům odesílat požadavky službě, aniž by bylo nutné sledovat, které fyzické pody ve skutečnosti službu tvoří. Služba má nastavenou permanentní IP adresu, která zůstává zachována i v případě náhlého výpadku. Zastává také funkci load balanceru, což znamená, že odchytí vstupní požadavek a přidělí ho podu (replice), který je nejméně vytížen.
- **Ingress** – Zpřístupňuje směrování HTTP a HTTPS z vnějšku clusteru službám v rámci clusteru. Ve zdrojových souborech této komponenty jsou pak nakonfigurovány routovací pravidla, které Ingress využívá pro správnou distribuci na jednotlivé služby.



Obrázek 4 – Routování v Kubernetes clusteru [14]

- **Config-map** – Externí konfigurace aplikací, kterou mohou pody využívat jako konfigurační soubor. Mohou zde být nadefinovány například proměnné prostředí jako je URL pro připojení do databáze. Přidanou hodnotu této komponenty je, že při změně konfigurace není nutné sestavení nového obrazu aplikace ani její opětovné nasazení do podu.

- **Secret –** Config-map komponenta není vhodná pro ukládání důvěrných dat, protože není žádným způsobem šifrována. Všechna data ukládá pouze v textovém formátu, tudíž jsou viditelná pro každého, kdo má k souboru přístup. Pro tato data existuje její obdoba s názvem Secret, která data zakóduje a uloží v base64 formátu.
- **Volume** – V případě restartování jakéhokoliv podu (dokonce i databázového) dochází ke ztrátě dat. Kubernetes cluster explicitně nespravuje žádnou persistenci dat. Pokud je požadováno dlouhodobé zachování dat, je nutné k podu připojit komponentu Volume, která mu přiřadí fyzický prostor na lokálním či vzdáleným disku. Komponenta se vztahuje na celý pod a je tedy připojena na všech jeho kontainerech.
- **Replica Set –** Zajišťuje, že v daném časovém okamžiku běží zadaný počet identických modulů.
- **Deployment** – Jedná se o metodu nasazení kontainerových aplikačních modulů. Požadovaný stav je popsaný v yaml souboru s názvem Deployment. Jakákoliv jeho změna způsobí, že kontrolery změní skutečný stav clusteru, aby dosáhl požadovaného (například vytvářením nebo odstraňováním replik). Dalo by se říci, že pod je abstrakce pro kontainer a Deployment je další úroveň abstrakce pro pod.
- **Stateful Set –** Replikaci databází nelze zajistit pomocí Deploymentu, protože databáze závisí na jejich stavu. Všechny repliky databázového podu přistupují ke sdílenému datovému uložišti. Pro eliminaci inkonzistence dat je nutné spravovat jaký pod může zapisovat data a jaký bude provádět pouze čtecí operace. Stateful Set je tedy určen pro aplikace, které jsou závislé na jejich stavu jako je například MongoDB či Elastic Search.
- **Namespace** – Jedná se o virtuální cluster (jeden fyzický cluster může provozovat více virtuálních) určený pro prostředí s mnoha uživateli rozmístěnými ve více týmech nebo projektech. Zdroje uvnitř jednoho jmenného prostoru musí být jedinečné a nemohou přistupovat ke zdrojům v jiném jmenném prostoru. Lze také alokovat určitý počet zdrojů,

aby se zajistilo omezení spotřeby na jeho maximální podíl na celkových prostředcích fyzického clusteru.

4.2.3 Helm

Technologie Kubernetes má nástroj pro správu balíčků, který nese pojmenování Helm. Umožňuje organizovat objekty v zabalené aplikaci, kterou si může kdokoli stáhnout a nainstalovat jedním kliknutím nebo nakonfigurovat dle požadovaných specifikací. Tyto balíčky jsou pojmenovány Helm Charts [15].

Helm je organizován podle několika klíčových konceptů:

- **Chart –** Balíček předem nakonfigurovaných zdrojů.
- **Release** – Konkrétní instance chartu, který byl nasazen do clusteru pomocí nástroje Helm.
- **Repository** – Skupina publikovaných chartů, které mohou být zpřístupněny ostatním uživatelům.

Tento nástroj využívá mnoho uživatelů po celém světe. Ulehčuje práci s vytvořením funkčního řešení, které je postaveno na Kubernetes technologii. Jeho přední přidané hodnoty jsou:

- **Zvýšení produktivity** – Namísto trávení času nasazováním testovacích prostředí pro vyzkoušení funkčnosti Kubernetes clusteru mohou vývojáři nasadit předem otestovanou aplikaci prostřednictvím Helm Charts a soustředit se na vývoj vlastních částí.
- **Poskytnutí existujících řešení** – Umožňují uživatelům získat například fungující databázi či platformu pro velká data nasazenou pro jejich aplikaci jedním kliknutím. Vývojáři pak mohou upravovat stávající charty nebo vytvářet své vlastní pro automatizaci vývojových, testovacích i produkčních procesů.
- **Snížená složitost** – Nasazení aplikací spravovaných tímto orchestračním nástrojem může být v některých případech extrémně složité. Použití nesprávných hodnot v konfiguračních souborech nebo nesprávné zavedení

aplikací ze šablon YAML může narušit samotné nasazení. Helm Charts umožňují komunitě předem nakonfigurovat aplikace, definovat hodnoty, které jsou pevné a jiné, které lze konfigurovat s rozumnými výchozími hodnotami a poskytují konzistentní rozhraní pro změnu konfigurace. To výrazně snižuje složitost a eliminuje chyby při nasazení tím, že zablokuje nesprávné konfigurace.

- **Opětovná použitelnost** – Jakmile uživatel vytvoří Helm Chart, jednou jej otestuje a stabilizuje, lze jej znovu použít ve více skupinách v organizaci i mimo ni. Před příchodem tohoto nástroje bylo mnohem obtížnější sdílet aplikace Kubernetes a replikovat je mezi prostředími.

Helm se architektonicky skládá z dvou hlavních komponent – Helm Client, který umožňuje uživatelům vytvářet nové charty a spravovat jejich repositáře. Dále je používaný pro komunikaci s Tiller Serverem, který běží uvnitř clusteru a převádí definice chartů a konfigurace do Kubernetes API. Tiller je také zodpovědný za aktualizaci chartů, jejich odinstalování a odstranění z clusteru.

4.3 Porovnání Kubernetes a Docker Swarm

Jak Kubernetes, tak Docker Swarm jsou populární orchestrátory, které pomáhají se správou a řízením kontainerových aplikací. Klíčovými rozdíly mezi těmito dvěma technologiemi může být například [16–18]:

- Jak je patrné z předchozích kapitol, mezi technologiemi je zásadní rozdíl v jejich architektuře. Kubernetes je skládá z několika hlavních komponent jako jsou master/worker uzly, API server, scheduler a další. Docker Swarm má jednodušší architekturu znatelně jednodušší. Všechny jeho uzly jsou stejné a řízení clusteru zajišťuje orchestrátor, který běží jako kontainer.
- Další rozdíl spočívá ve škálování. Kubernetes podporuje jak horizontální, tak vertikální škálování a umožňuje škálování na úrovni služby, což umožňuje škálovat pouze určité části aplikace. Docker Swarm podporuje škálování pouze horizontální. Tímto pojmem je myšleno přidávání dalších replik služeb pro zvýšení kapacity a odolnosti aplikace při velkém provozu. Swarm pak vytváří více kontainerů (instancí) dané služby a rovnoměrně je distribuuje

na dostupné uzly. Vertikální škálování je naopak proces zvyšování výkonu jednoho kotaineru nebo repliky tím, že se změní jejich konfigurace na vyšší výkonovou úroveň (v Kubernetes je nazváno „vertical pod autoscaling“). Vertikální škálování je užitečné pro aplikace, které vyžadují vysokou výpočetní kapacitu nebo paměť v rámci jednoho kontaineru.

- Kubernetes je navržen tak, aby lépe spravoval stavové aplikace než Docker Swarm. To znamená, že lze lépe spravovat aplikace, které vyžadují ukládání stavu, jako jsou databáze.
- Kubernetes nabízí oproti Docker Swarmu více možností pro správu a monitorovaní aplikací jako jsou nástroje pro sledování stavu a logování.
- Docker Swarm je mnohem jednodušší na použití.

Nelze jednoznačně určit, který ze zmíněných orchestrátorů je lepší pro použití. Kubernetes je využívanější pro větší, složitější aplikace, které vyžadují větší flexibilitu, škálovatelnost a správu stavových aplikací jako jsou databáze. Docker Swarm je naopak díky své snadné použitelnosti vhodný pro menší aplikace. Výběr správného orchestračního nástroje může být náročný a je důležité zhodnotit potřeby cílové aplikace.

5 Cloud Computing

Cloud computing je v posledních letech považován za jedno z nejvýznamnějších paradigmat v oblasti informačních technologií. Tato technologická strategie umožňuje svým spotřebitelům zavést bezproblémové komunikační připojení k systému výpočetních zdrojů, kde uživatelé mohou snadno škálovat své požadavky s minimálním zapojením třetích stran [17, 18]. Zjednodušeně by se dalo říci, že se jedná o ukládání a přístup k datům a programům přes internet namísto pevného disku lokálního počítače [21]. V širším slova smyslu je to model umožňující všudypřítomný a pohodlný síťový přístup ke sdíleným konfigurovatelným výpočetním zdrojům na vyžádání, které lze rychle poskytnout a uvolnit s minimálním úsilím správy nebo interakce s poskytovatelem služeb. Tento cloudový model se skládá z pěti základních charakteristik, tří modelů služeb a čtyř nasazovacích modelů [22].

Poskytovatelé cloudových služeb (CSPs) jsou prodejci, kteří svým zákazníkům poskytují prostředky a služby Cloud Computingu, které jsou dynamicky využívány na základě poptávky zákazníka podle určitého obchodního modelu. Služby v různých oblastech, jako je obchod a vzdělávání, jsou zákazníkům poskytovány online a jsou přístupné přes internet pomocí webového prohlížeče, zatímco data a programy jsou uloženy na cloudových serverech umístěných v datových centrech [21].

5.1 Charakteristiky

Mezi dříve zmíněné základní charakteristiky Cloud Computingu patří [22]:

- **Samoobsluha na vyžádání** – Zákazník může využít výpočetní schopnosti, jako je čas serveru a síťové uložiště, podle potřeby automaticky, aniž by vyžadoval interakci s poskytovatelem služeb.
- **Síťový přístup** – Výpočetní prostředí jsou dostupné přes síť a jsou přístupné prostřednictvím standardních mechanismů, které podporují použití tenkými nebo tlustými klientskými platformami (mobilní telefony, tablety, pracovní stanice).

- **Resource pooling** – Výpočetní zdroje jsou sdíleny, aby obsluhovaly více spotřebitelů pomocí multi-tenant modelu, kde jsou fyzické a virtuální zdroje dynamicky přeřazovány podle poptávky uživatelů. Na pozadí je softwarově zajištěna izolace dat, aby uživatelé měli přístup pouze k těm, na které mají oprávnění.
- **Elasticita** – Zdroje lze rychle zajišťovat a uvolňovat na základě poptávky spotřebitele.
- **Měřená služba** – Cloudové systémy automaticky řídí a optimalizují využití zdrojů na základě schopnosti měření na určité úrovni abstrakce vhodně pro typ služby (stav uložiště, šířka pásma, počet aktivních účtu, …). Využití zdrojů lze monitorovat, řídit, hlásit, což poskytuje transparentnost jak pro poskytovatele, tak pro spotřebitele využívané služby.

5.2 Modely služeb

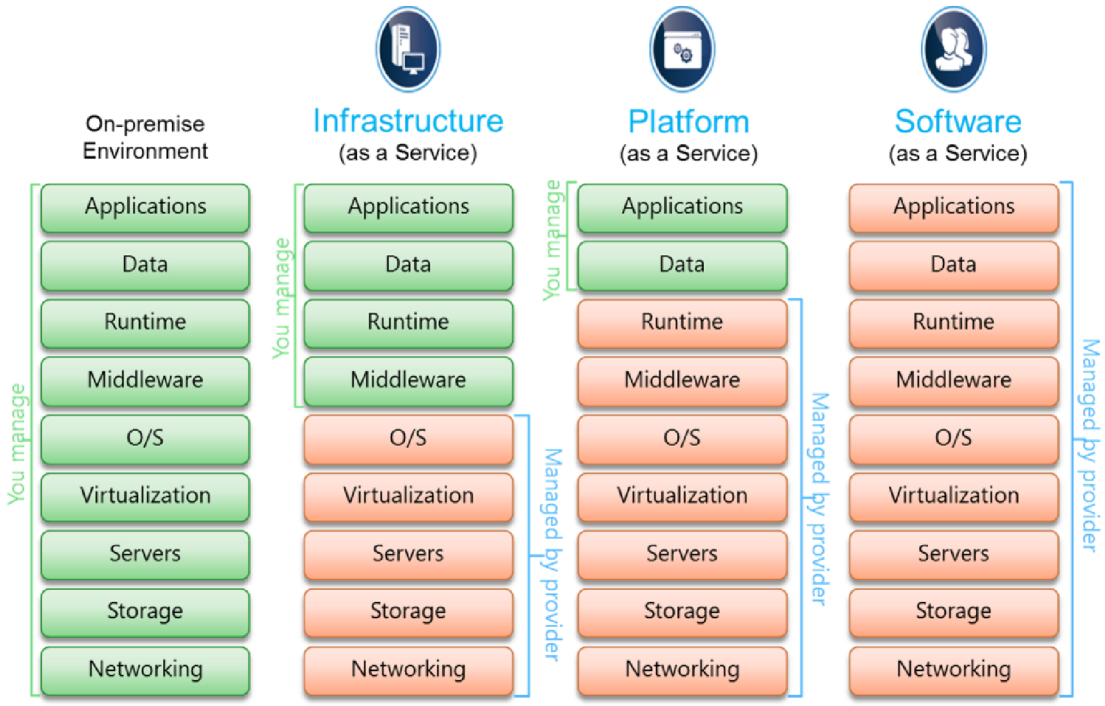
Mezi základní modely Cloud Computingu patří:

- **Infrastructure as a Service (IaaS)** – Poskytovatel cloudových služeb poskytuje sadu virtualizovaných výpočetních zdrojů, jako je například CPU, paměť, OS nebo aplikační software v cloudu. IaaS využívá virtualizační technologii k převodu fyzických prostředků na logické, které mohou být dynamicky poskytovány a uvolňovány zákazníky podle potřeby. Mezi hlavní společnosti, které nabízejí tento model Cloud Computingu patří například Google, Amazon, IBM. Mezi výhody IaaS řešení patří:
    - Snižuje náklady na kapitálové výdaje
    - Uživatelé platí za službu, kterou chtějí
    - Přístup k podnikovým IT zdrojům a infrastruktuře
    - Uživatelé mohou kdykoliv libovolně škálovat svoje zdroje na základě jejich požadavků
- **Platform as a Service (PaaS)** – Jedná se o pokročilejší typ služby Cloud Computingu. Poskytovatel cloudových služeb nabízí, provozuje a udržuje jak systémový software, tak další výpočetní zdroje. Služby PaaS zahrnují návrh, vývoj a hosting aplikací. Mezi další služby patří také spolupráce, integrace

databáze, zabezpečení, integrace webových služeb a škálování. Spotřebitelé se tedy nemusí starat jak o hardwarové či softwarové zdroje, ani o odborníky na jejich správu. Toto schéma poskytuje flexibilitu při instalaci softwaru do systému. Mezi hlavní výhodu tohoto řešení patří absence aktualizací infrastruktury na straně spotřebitele (všechna údržba je v roli poskytovatele). Nevýhodou je však nedostatečná spolupráce a přenositelnost mezi různými poskytovateli.

- **Software as a Service (SaaS)** – V tomto modelu je poskytovatel nejen za provoz operačního systému a zdrojů, ale i za údržbu aplikačního softwaru. SaaS se spotřebiteli jeví jako webové aplikační rozhraní, kde se používá internet k poskytování služeb, ke kterým se přistupuje pomocí webového prohlížeče. Příkladem je například služba Gmail, ke které lze přistupovat prostřednictvím různých zařízení jako jsou chytré telefony a notebooky. Na rozdíl od tradičního softwaru má SaaS výhodu, že zákazník nemusí kupovat licence, instalovat, upgradovat ani udržovat software na svém vlastním zařízení. Mezi další výhody patří zejména:
    - Možnost konfigurace a rychlá škálovatelnost
    - Odstraňuje problémy s infrastrukturou

Z obrázku níže by se dalo usoudit, že pokročilejší služba Cloud Computingu vždy rozšiřuje tu jednodušší a jsou tak postaveny jedna na druhé.



Obrázek 5 Servisní modely Clound Computing [23]

5.3 Nasazovací modely

V rámci Cloud Computingu lze na výše zmíněných modelech služeb nasadit více aplikačních modelů. Specifické modely nasazení lze rozdělit na základě jejich povahy poskytování, které závisí na umístění cloudové služby a specifikují tak, jak jsou zpřístupněny uživatelům [24, 25]:

- **Privátní cloud** – V souladu se svým názvem je privátní cloud obvykle infrastruktura používaná jednou organizací. Takovou infrastrukturu může spravovat samotná organizace pro podporu různých skupin uživatelů, nebo ji může spravovat poskytovatel služeb. Soukromé cloudy jsou dražší než veřejné cloudy kvůli kapitálovým výdajům spojeným s jejich pořízením a údržbou. V dnešní době jsou však schopny lépe řešit obavy o bezpečnost a soukromí.
- **Veřejný cloud** – Jak název napovídá, tento typ cloudového modelu nasazení podporuje všechny uživatele, kteří chtějí využívat výpočetní prostředky, jako

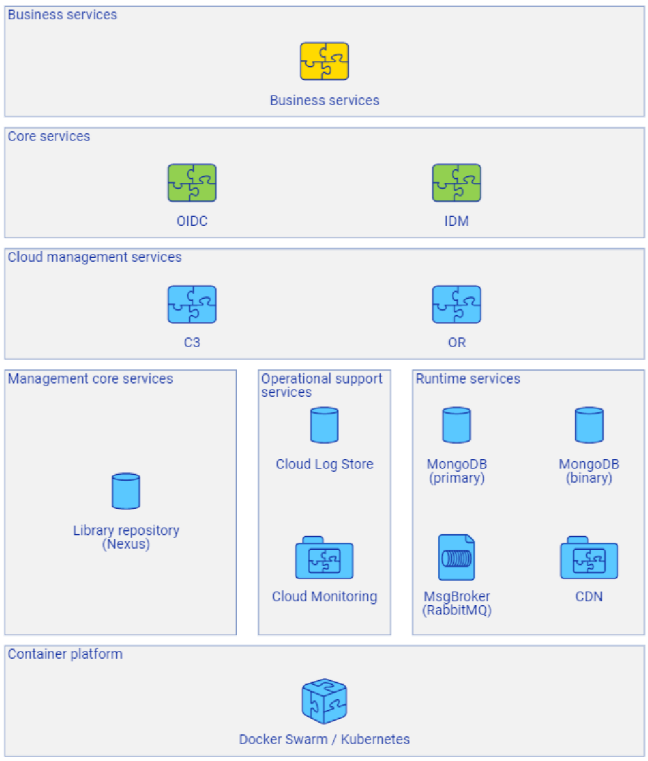
je hardware (CPU, paměť, uložiště) nebo software (aplikační server, databáze) na základě nějakého platebního plánu. Veřejné cloudy se nejčastěji používají pro vývoj a testování aplikací, nekritické úlohy, jako je sdílení souborů a e-mailové služby.

- **Hybridní cloud** – V hybridním cloudu organizace využívá propojenou privátní a veřejnou cloudovou infrastrukturu. Mnoho organizací využívá tento model, když potřebují rychle rozšířit svou IT infrastrukturu, například když využívají veřejné cloudy k doplnění kapacity dostupné v rámci privátního cloudu. Pokud například online prodejce potřebuje více výpočetních zdrojů pro provoz svých webových aplikací během sezóny vysokého provozu, může tyto zdroje získat prostřednictvím veřejných cloudů.
- **Komunitní cloud** – Tento model nasazení podporuje více organizací sdílejících výpočetní zdroje, které jsou součástí komunity. Příklady zahrnují univerzity spolupracující v určitých oblastech výzkumu nebo policejní oddělení v rámci okresu nebo státu sdílející výpočetní zdroje. Přístup do prostředí komunitního cloudu je obvykle omezen pouze na členy komunity.

U veřejných cloudů mohou být náklady pro koncové uživatele o něco vyšší, ale nevyžadují tolik kapitálových a personálních nákladů za provozování vlastních hardwarových a softwarových prostředků. Použití privátních cloudů zahrnuje kapitálové výdaje, které jsou ale stále nižší než náklady na vlastnictví a provoz infrastruktury díky vyšší úrovni resource poolingu. Soukromé cloudy také nabízejí větší podporu zabezpečení než veřejné cloudy. Některé organizace se proto mohou rozhodnout používat privátní cloud pro kriticky důležité aplikace a veřejné cloudy pro základní úkoly, jako je vývoj aplikace, testování prostředí a e-mailové služby.

6 Popis cílové infrastruktury

Hlavním cílem této diplomové práce je automatizace nasazení komplexní podnikové aplikace. Cílový informační systém se aktuálně skládá z více než třiceti dílčích aplikací. Každá z nich běží jako samostatná mikroslužba na on-premise cloudu spravovaná orchestračním nástrojem Docker Swarm. Minimalistická architektura cílového systému je vyobrazena na obrázku číslo 6 a mezi její hlavní části patří především:



Obrázek 6 - Minimalistická architektura cílového systému [autor].

- **Business services** – Dílčí podnikové aplikace cílového systému.
- **OIDC** a **IDM** – Služby zodpovědné za autentizaci a autorizaci uživatelů.
- **OR** – Aplikace spravující provozní informace o cloudové infrastruktuře a zdrojích.
- **C3** – Aplikace používaná k řízení nasazování a provozu ostatních aplikací v cloudovém prostředí. Po provedení nasazení zaznamená alokované zdroje do OR. Umožňuje také změny škálování prostřednictvím konfigurace prostředí jednotlivé služby.
- **Nexus** – Jedná se o open-source uložiště, které podporuje uchovávání Java, Docker a npm balíčků. Díky integraci této služby je možné publikovat a získávat verze aplikací se všemi jejich závislostmi [26].
- **Operational support services** – Služby používané aplikacemi i cloudem například pro ukládání logů či monitorování stavu cloudu.

- **Runtime Services** – Služby vyžadované aplikacemi při provozu.
    - **MongoDB** – Multiplatformní dokumentová databáze, která je využívána aplikacemi pro ukládání konfiguračních, businessových i binárních dat.
    - **RabbitMQ** – Open-source služba pro zprostředkování zpráv.
    - **CDN** – Síť pro doručování obsahu, která je využita k uchování a distribuci frontendových knihoven.

Cílem však není pouze automatizace samotného procesu nasazení, ale také uvedení nového prostředí či nové verze do funkčního stavu. Praktická část této diplomové práce se tedy zabývá návrhem a automatizací celého instalačního procesu, jehož výsledkem je plně funkční prostředí. Obchodní aplikace, které jsou předmětem nasazování mají velmi specifickou strukturu a definují několik základních pojmů:

- **asid** – Jedinečný identifikátor instance aplikace.
- **asidOwner** – Uživatel, který má administrátorská práva v rámci jedné instance a je použit k její inicializaci.
- **workspace** – Pracovní prostředí, které je uvnitř jedné instance. Každá instance může mít více na sobě nezávislých workspaců (jejich data jsou od sebe systémově odděleny a nijak se neovlivňují).
- **awid** – Jedinečný identifikátor jednoho workspacu v rámci instance.
- **awidOwner** – Uživatel, který má administrátorská práva v rámci jednoho workspacu, je použit k jeho inicializaci a nahrání obchodních dat.
- **systémová identita** – Systémová identita dané instance či workspacu. Aplikace ji používá k provedení systémových volání (její token nelze získat z venku a požadavek musí být vždy podepsán aplikací)

V prvním kroku instalačního procesu je nezbytné provést přípravu infrastruktury pro cílové aplikace. Jak již bylo zmíněno, pro řízení samotného nasazení je využita služba C3, která zaznamenává provozní informace do služby OR, kde je nejprve zapotřebí vytvořit registrace asidů a awidů cílových aplikací (tenant). Navržený automatizační script musí také zajistit vytvoření instalačních uživatelů, databází, databázových uživatelů a případnou inicializaci služby RabbitMQ dle potřeb

aplikací. Pro zvýšení bezpečnosti je možné uchovávat vytvořený řetězec pro připojení k databázi ve službě OR a v aplikaci na něj pouze odkazovat jedinečným identifikátorem (OSID pro objektovou a BSID pro binární databázi).

Každá obchodní aplikace má ve svém git repositáři umístěné konfigurační soubory pro každé prostředí. Jejich manuální udržování při vysokém počtu aplikací a prostředí má vysokou časovou náročnost. Konfigurace různých prostředí mají ve většině případů téměř stejnou strukturu a liší se pouze v několika základních parametrech (typicky awidy, asidy, přístupové údaje a uživatelé). Díky této vlastnosti mohl být pro zefektivnění přípravy navržen nástroj, který umožňuje globální generování těchto souborů prostřednictvím konfigurovatelných šablon.

Třetí část nasazovacího procesu je zaměřena na správnou distribuci všech použitých nodejs knihoven do cílového Nexus repositáře. Každá nodejs aplikace v průběhu jejího sestavení vytváří takzvaný package-lock soubor, ve kterém jsou dostupné informace o všech použitých knihovnách spolu se zdrojovým odkazem. Nástroj automatizující tento krok byl navržen tak, aby dokázal zpracovat všechny tyto soubory, zjistil rozdíly mezi dvěma repositáři a nahrál všechny chybějící knihovny do cílového Nexusu.

Dalšími kroky je nasazení a instalace aplikací. Jak již bylo zmíněno v předchozí části této kapitoly, nasazení je převážně řešeno službou C3. Nasazovací script pak pouze sesbírá aplikační konfigurace spolu s odkazy na aplikační balíčky a předá tyto informace do C3 prostřednictvím http API. Obchodní aplikace nejsou okamžitě po nasazení připraveny k použití. Každá má svůj životní cyklus a je potřeba provést několik inicializačních kroků – zejména inicializace instance, založení a inicializace workspacu (spolu s nějakou aplikační konfigurací) a případné nahrání obchodních či testovacích dat. Pro splnění tohoto požadavku byla vytvořena datová sada obsahující všechny potřebné příkazy, které jsou následně sekvenčně spouštěny automatizačním nástrojem pro instalaci.

Posledním krokem pro dokončení instalace je nahrání všech frontendových knihoven do CDN. Navržené scripty exportují produkční knihovny, které jsou součástí firemní CDN prostřednictvím http API a je doplňují o balíčky projektu. Výsledek exportu je poté druhým scriptem importován přímo na cílovou CDN.

7 Inicializace infrastruktury

Jak již bylo zmíněno, prvním krokem automatizace je vytvoření infrastruktury pro cílové služby. Prerekvizity pro tento krok jsou mít k dispozici Kubernetes či Docker Swarm cluster s nainstalovanými službami pro jeho správu (C3, OR) a práci s uživateli (OIDC a IDM). Přípravu infrastruktury lze dále rozčlenit do několika dílčích bodů, pro které byl navržen následující konfigurační soubor, který je využíván scriptem pro inicializaci:

```
[{
 "code": "demo_application",
 "asid": "10100000000000000000000000000000",
 "awid": "10100000000000000000000000000010",
 "asidOwner":{
    "code": "asidOwner@email.cz",
    "accessCode1": "...",
    "accessCode2": "...",
    "identityAccountCode": "1-1-0-100"
 },
 "awidList": [],
 "mongodb": [{
  // Typ databáze - primární / binární
       "db": "primary",
       "name": "demo_development",
       "user": "...",
       "password": "...",
  // Možnost přegenerovat náhodná hesla k databázím
       "forcePasswordGeneration": false,
       "resourceCode": "10101010000000000000000000000000"
 }],
 "rabbit":[{
       "vhost": "demo_vhost",
       "password": "..."
 }]
}]
```

Kód 1 - Konfigurace infrastruktury [Autor]

- **code** – Unikátní označení nasazované aplikace.
- **asid, awid –** Značí unikátní identifikátor aplikace. Asid je použit pro instanci a awid pro workspace. Na základě této konfigurace bude vytvořen tenant ve službě Operation Registry (OR).
- **asidOwner, awidOwner** – Uživatelé, kteří mají oprávnění k inicializaci instance či workspacu. Automatizační script dle zadaného unikátního kódu a přihlašovacích údajů vytvoří uživatele ve službě OIDC. Atribut

identityAccouuntCode dále předepisuje označení pro systémovou identitu aplikace, kterou cílová služba používá pro komunikaci se zbytkem systému.

- **awidList** – Možnost specifikace více workspaců v rámci jedné aplikační instance.
- **mongodb** – Konfigurace objektové databáze, kterou bude aplikace využívat. Script na základě této konfigurace vytvoří databázovou kolekci, které následně přiřadí nového uživatele. Pokud je atribut „password“ vyplněn, použije se předdefinované heslo a v opačném případě se vygeneruje náhodné. V případě specifikace atributu „resourceCode“ je pod zadaným kódem uložen odpovídající připojovací řetězec do služby OR. Koncové aplikace pak nemusí definovat přihlašovací údaje k databázi ve vlastní konfiguraci prostředí, ale stačí použít pouze tento odkaz ve formátu „osid:<resourceCode>“.
- **rabbit** – Konfigurace pro inicializaci služby RabbitMQ. Automatizační script na základě této konfigurace vytvoří vhosty a uživatele s právy.

8 Příprava deployment deskriptorů

Druhým bodem přípravy před nasazením je vytvoření nasazovacích konfigurací pro všechny dílčí aplikace. Každá z aplikací má svoje deployment deskriptory umístěné v samostatném git repositáři spolu se zdrojovými kódy, což má za následek velice neefektivní provádění změn stávajících nebo vytváření nových prostředí. Pro zefektivnění byl navržen koncept nástroje pro správu takovýchto konfigurací, který bude generovat deployment deskriptory do cílových repositářů.

Prvním krokem je poskytnutí jednoduché JSON konfigurace, ve které jsou specifikovány repositáře cílových aplikací, zdrojová a cílová větev, umístění na disku a další specifické atributy aplikace.

```
{
  // Nastavení gitu
  "commitChanges": true,
  "pushChanges": true,
  "git": {
    "baseUri": "ssh://git@demogit.net:9422",
    "baseCheckoutDir": "C:/work",
    "sourceBranch": "sprint",
    "targetBranch": "sprint"
  },
  // Nastavení cesty ke složce s projekty a konfiguračními soubory
  "uuProjectsPath": "../",
  "initDataPath": "../../initdata",
  // Konfigurace jednotlivých projektů - obsahuje specifické
atributy aplikací, které šablony používají pro generování
descriptorů
  "projects": {
    "demo-application": {
      "code": "DEMOAPPLICATION",
      "type": "nodejs",
      "path": "demo_application",
      "uuPath": "demo_application",
      "user": "demo_application",
      "uuSubAppRootPath": "demo_application",
      "clientPath": "demo_application-client",
      "descriptorPath": "demo_application-server/env",
      "name": "demoApplication",
      "commit": "HEAD"
    }...
}}
```

Kód 2 - Ukázka konfigurace nástroje pro generování deskriptorů [autor].

Dále je nutné vytvořit šablony konfiguračních souborů, které budou výstupem generování. Myšlenka celého konceptu spočívám v tom, že se od sebe jednotlivá prostředí liší pouze v malém množství konfigurací, které jsou specifické pro dané

prostředí, a proto je lze genericky skládat na základě malého množství hodnot. Šablony tedy reflektují kompletní strukturu cílových deployment deskriptorů a umožňují je modifikovat na jednom místě. Všechna prostředí mají například rozdílné kódy instance (asid) a workspacu (awid). V šabloně lze tedy stanovit, že kód prostředí bude zastupovat první znak těchto kódů. Ve všech ostatních prostředích bude poté stačit nakonfigurovat pouze odpovídající prefix.

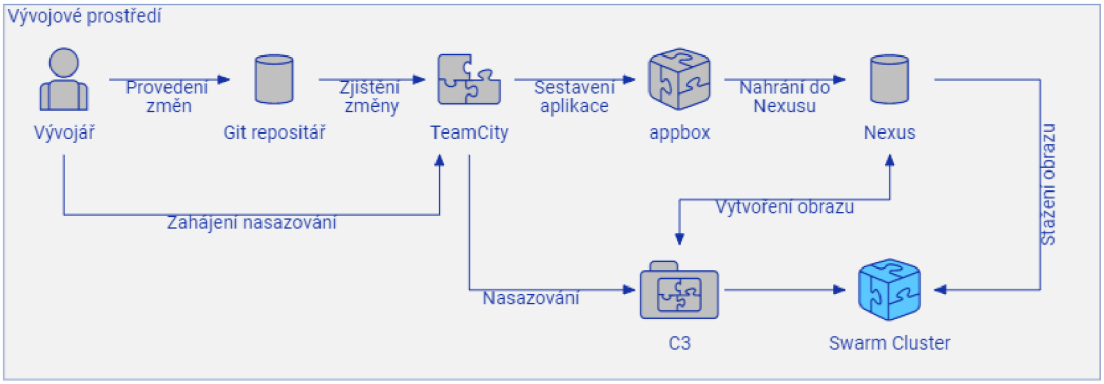
```
// Nastavení názvu a prefixu pro dané prostředí
const ENVIRONMENT = "development";
const PREFIX = "1"

function main({ inventories, config }) {
  // Získání descriptoru prostředí ze šablony, které jsou předány
atributy specifické pro prostředí.
  let envConfig = {
    ...core.main({inventories, config}, ENVIRONMENT, PREFIX),
    environment: ENVIRONMENT,
    envLabel: `demo${ENVIRONMENT}`,
    resourcePoolUri: `ues:UNI-BT:${ENVIRONMENT.toUpperCase()}`,
    cdnBaseUri: `http://${ENVIRONMENT}.cdn.demo/`,
    environmentIdentifierColor: "#FFFF00"
  }

  // Možnost přidat / overridnout custom attributy v configMapě
  let subApps = core.subApps(inventories, config, envConfig, {
    [SubApp.DEMO_APPLICATION]: {
      development_specific_config_attribute: true
    },
  });

  return {
    ...envConfig,
    subApps
  };
}
```

Kód 3 - ukázka konfigurace development prostředí [autor].

Nástroj po spuštění stáhne všechny repositáře projektů specifikovaných v první konfiguraci (viz kód číslo 2), vygeneruje odpovídající konfigurace pro nasazení a pushne je do cílových větví. Při starém přístupu bylo nutné při tvorbě nového prostředí vytvořit více než 30 konfiguračních souborů, kde každý obsahoval kolem 100 řádků. S použitím generování pomocí šablon bylo docíleno vytvoření všech deskriptorů s konfigurací o velikosti 80 řádků.

9 Nasazování prostředí

Dalším bodem instalačního procesu se stává nahrání knihoven do cílového Nexus repositáře a samotné nasazení aplikací. Postup pro tyto dva požadavky se navíc liší dle cílového prostředí. Pro zefektivnění vývoje je aktuálně na pracovní prostředí integrována technologie CI/CD pipeliny. Jedná se o sadu kroků, které musí být vykonané v určitém pořadí, aby bylo možné dodat novou verzi systému. CI/CD pipelina je postup zaměřený na zlepšení poskytování softwaru během vývoje prostřednictvím automatizace, kde je každý krok definován pomocí powershell scriptů [27]. Každý vývojář přispívající k tvorbě systému má možnost pushnout vlastní změny do cílového repositáře. Jakákoliv změna je automaticky zaregistrována nástrojem TeamCity, který spustí sestavení aplikace spolu s testy a kontrolou syntaxe. Pokud je aplikace úspěšně sestavena, zabalí ji do příslušného archivu (takzvaný appbox ve formátu .war nebo .zip), který nahraje do Nexus repositáře. Vývojář poté může ze sestaveného balíčku spustit nasazení, kde TeamCity předá balíček do C3, která z něj vytvoří obraz a spustí nasazení do kontainerového clusteru, který je spravován orchestračním nástrojem Docker Swarm.



Obrázek 7 - Automatizace nasazení na vývojové prostředí [autor].

Využití této CI/CD pipeliny nesplňuje všechny požadavky stanovené zákazníkem. Z tohoto důvodu je využita pouze pro interní prostředí, kde je vyžadováno časté dodávání nových funkcionalit a oprav, které jsou díky jejímu využití efektivně nasazovány. Pro externí prostředí (UAT) může mít tento postup několik nedostatků. Pojmem UAT se rozumí poslední stupeň testování aplikace, který je provozován

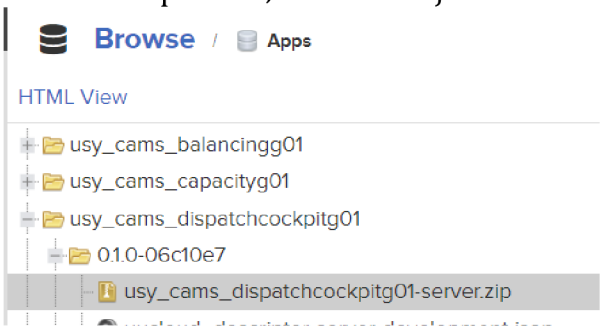
na cílovém zařízení zákazníka a kde probíhá uživatelské akceptační testování [28]. Jeden z nedostatků dříve zmíněného způsobu nasazování může být ten, že nástroj TeamCity nemusí mít přístup k cílovému prostředí, které se nachází ve virtuální soukromé síti zákazníka (VPN). Dále je požadováno vytvoření instalačního balíčku s cílovou verzí, do které již nebudou přibývat žádné změny. Kvůli těmto požadavkům byla vytvořena sekvence nodejs scriptů, které jsou používány pro nasazení cílové verze informačního systému na externí prostředí.

Nexus repositář může obsahovat více různých archivů k jedné verzi aplikace a při výběru správného sestavení může nastat chyba. O tento krok se tedy stará první automatizační script.

```
async updateLastVersion(){
  // Nastavení cesty k aplikaci v Nexus repositáři
  let nexusLocation = `${nexusConfig.urlInternal ||
nexusConfig.url}/repository/Apps/${appConfig.product}`
  // Vylistování souborů z Nexus repositáře
  let nexusFiles = await this._listFileFromNexus(nexusLocation,
appEnvConfig.uuSubApp.version.replace(/-[a-f0-9]{7}$/, ""));
  // Získání poslední verze z nexusu (dle data vytvoření)
  const { lastAppBox } = this._processNexusFiles(nexusFiles);
  // Získání řetězce s verzí z vrácené cesty souboru
  let grpSplit = lastAppBox.group.split("/")
  let lastVersion = grpSplit[grpSplit.length - 1];

  // Aktualizace deployment deskriptorů
  appConfig.version = lastVersion;
  appEnvConfig.uuSubApp.version = lastVersion
  // Uložení na disk
  this._updateUuAppJson(appConfig);
  this._updateUuCloudConfig(appEnvConfig);
}
```

Kód 4 - Aktualizace požadované verze [autor].

Nejprve je složena cesta k cílovému adresáři s umístěním appboxů zvolené aplikace v Nexus repositáři, ze které jsou následně vylistovány všechny soubory dle požadované verze. Jak lze vidět na obrázku číslo 8, názvy verzí mají formát {verze}-{číslo sestavení}, kde číslem sestavení je náhodný sedmimístný hash. Cílem je získat poslední appbox k dané verzi, a proto jsou soubory listovány bez čísla



Obrázek 8 - Struktura Nexus repositáře [autor].

sestavení. Výsledkem listu jsou informace o všech appboxech k dané verzi, ze kterých lze jednoduše určit, které sestavení je nejaktuálnější (viz kód číslo 5). Získané číslo verze se poté aktualizuje a uloží zpět do konfiguračního souboru pro nasazení.

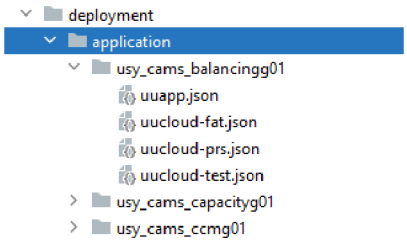
```
_processNexusFiles(nexusFiles){
  // Získání knihovny moment pro porovnávání datumů
  const moment = require("moment");
  let lastAppBox;
  // Získání koncovky souboru (.war / .zip)
  const ext = this.getArchiveExtension();
  // Získání názvu souboru dle vzoru
     <názevAplikace><postfix(-server)>.<koncovka>
  let appBoxFileName =
`${appConfig.product}${this.getArchivePostfix()}.${ext}`;
  // Najdi nejnovější soubor, který odpovídá názvu archívu aplikace
v nexusu a ulož ho do lastAppBox proměnné
  nexusFiles.items.forEach(file => {
    if(file.name.includes(appBoxFileName)){
      if(!lastAppBox){
        lastAppBox = file
      } else {
if(moment(file.assets[0].lastModified).isAfter(moment(lastAppBox.as
sets[0].lastModified))){
          lastAppBox = file
        }
      }
    }
  })
  return lastAppBox;
}
```

Kód 5 - Získání posledního appboxu [autor].

Po úspěšné aktualizaci verzí v deployment deskriptorech je možné zahájit proces stažení. Postup je obdobný jako v předchozím scriptu, s tím rozdílem, že je nyní známo číslo sestavení verze. Hledání posledního appboxu v konkrétním sestavení je zde z důvodu existence více archivů v jednom adresáři. Z nalezeného souboru lze poté prostřednictvím streamu stáhnout všechny jeho náležitosti a uložit je na disk.

```
async download() {
  // Cesta k nexus složce s cílovou aplikací
  let nexusLocation = `${nexusConfig.urlInternal ||
nexusConfig.url}/repository/Apps/${appConfig.product}`
  // Získání všech souborů, které jsou ve složce s danou verzí
  let nexusFiles = await this._listFileFromNexus(nexusLocation,
appEnvConfig.uuSubApp.version);
  // Získání poslední verze (stejné jako v ukázce 5)
  let lastAppBox = this._processNexusFiles(nexusFiles);
  // Stažení souboru z Nexus repositáře a jeho uložení na disk
  await this._downloadFileFromNexus(lastAppBox);
  return {requestedVersion: appEnvConfig.uuSubApp.version, appBox:
lastAppBox};
}

async _downloadFileFromNexus(file){
  // Pro všechny přílohy daného souboru - stáhni a ulož na disk
  for (let asset of file.assets) {
    let paths = asset.path.split("/");
    let filename = paths[paths.length - 1];
    let version = paths[paths.length - 2];
    await this._downloadFromNexus(asset.downloadUrl, filename);
    console.log(`Attachment ${filename} in version ${version} (last
modified: ${asset.lastModified}) was downloaded successfully.`);
  }
}
```

Kód 6 - Stažení appboxu [autor].



Obrázek 9 - Struktura umístění deskriptorů [autor].

Nad těmito dvěma scripty existuje řídicí script, který má v parametrech cestu k umístění deskriptorů všech aplikací (adresář application viz obrázek 9). Script poté načte všechny jeho podadresáře, v cyklu aktualizuje verze, stáhne odpovídající appboxy všech nalezených aplikací a vytvoří jednoduchý logovací soubor s informacemi o stažení.

Po stažení balíčků je nutné před jejich nasazením na UAT prostředí vykonat poslední krok, kterým je nahrání serverových knihoven do Nexus repositáře. Pro tento účel byl vytvořen CLI nástroj, který kontroluje, zda jsou všechny knihovny využívané aplikacemi dostupné v cílovém prostředí. Nástroj má dva způsoby použití – jeden pro kontrolu pouze jedné aplikace (spíše pro debugovací účely) a druhý, který je schopen rekurzivně prohledat poskytnutý adresář a zkontrolovat všechny

nalezené mikroslužby. Lze jej libovolně spustit prostřednictvím příkazového řádku a parametrizovat pěti hlavními atributy:

Obrázek 10 Nástroj pro kontrolu knihoven v Nexus repositáři [autor].

- **applicationDir –** Cesta ke složce s aplikacemi (viz obrázek 9). Pokud by byl namísto příkazu „check-all“ použit příkaz „check-one“, lze v této cestě specifikovat pouze jednu cílovou aplikaci.
- **downloadDir –** Cesta k lokálnímu adresáři pro stažení knihoven (pokud není specifikován, nástroj vypíše chybějící knihovny pouze do konzole).
- **targetNexusRegistry –** Cílový Nexus repositář pro porovnání chybějících knihoven (pokud není specifikován, stáhnou všechny, které aplikace používají).
- **uploadConfig –** Konfigurační soubor s přístupovými údaji k cílovému Nexus repositáři. Při specifikaci tohoto parametru se stažené knihovny automaticky nahrají dle zadané konfigurace.
- **regexFilters** – Možnost specifikace regulárních výrazů, které určují, jaké knihovny mají podléhat kontrole.

Z důvodu, že zabalená Java aplikace obsahuje kompletní strukturu včetně knihoven, je tuto kontrolu nutné provádět pouze pro nodejs aplikace. Nástroj tedy pracuje se soubory package-lock, které mají informace o stažených knihovnách:

```
{
  "name": "DemoProject",
  "version": "1.0.0",
  "dependencies": {
      "demolibrary": {
        "version": "1.5.2",
        "resolved": "http://nexus.demo/repository/npm-
group/.../demolibrary-1.5.2.tgz"
      }, ... //více závislostí
  }, ... nějaké další parametry
}
```

Kód 7 Ukázka struktury package-lock.json [autor].

```
async function processDependencies(dependencies, checkRegistry,
     regexFilters = []) {

  // Inicializace listu, kam se budou ukládat chybějící závislosti
  let missingDependencies = [];
  // Pro každou závislost z package-lock.json souboru
  for (let [name, dependency] of Object.entries(dependencies)) {
  // Pokud závislost má svoje závislosti, zpracuj je rekurzivně
    if (dependency.dependencies) {
      await processDependencies(dependency.dependencies,
          checkRegistry, regexFilters);
    }

  // Zkontroluj, jestli závislost splňuje definované filtry ze
vstupu
    if (regexFilters.some((regex) => name.match(regex))) {
  // Pokud je vyžadováno porovnání s cílovým registrem -> ověř
jestli v něm už existuje a případně ho přidej do listu chybějících
závislostí
      if (checkRegistry) {
        await checkDependency(dependency, checkRegistry,
            missingDependencies);
  // Pokud není definován cílový registr, přidej všechny knihovny
      } else {missingDependencies.push(dependency);}
    }
  }

  return missingDependencies;
}
```

Kód 8 Kontrola chybějících závislostí [autor].

Nástroj na základě atributu „dependencies“ dokáže vyhodnotit chybějící závislosti. Kontrola je prováděna rekurzivně, protože každá závislost může mít nějaké další, které implicitně využívá (viz kód 8).

Pokud nalezená závislost odpovídá jednomu ze zadaných filtrů, je zkontrolováno, jestli se již nenachází v cílovém repositáři. Pro tento požadavek je nejprve upravena část odkazu na knihovnu tak, aby odpovídala správnému umístění v cílovém repositáři. Dále je proveden jednoduchý http požadavek, který v případě neúspěchu značí, že se daná knihovna nenachází na cílové cestě (viz kód 9).

```
async function checkDependency(dependency, checkRegistry,
missingDependencies) {
  // Pokud má závislost parametr „resolved" (uri ke stažení
závislosti)
  if (dependency.resolved) {
    return new Promise((resolve, reject) => {
  // Nahraď zdrojový repositář cílovým
      let url = dependency.resolved;
      url = url.replace(/.*repository\/npm-group/, checkRegistry);

  // Ověř, zda závislost existuje v cílovém adresáři -> pokud vrátí
http status jiný než úspěch, znamená to, že neexistuje
      const req = http.request(url, (res) => {
        if (res.statusCode !== 200) {
          missingDependencies.push(dependency);
        }
      });
      req.on("close", () => {resolve();});
      req.end();
    });
  }
}
```

Kód 9 Ověření existence v cílovém repositáři [autor].

Ze získaného listu chybějících závislostí jsou v řídicím scriptu odstraněny všechny duplicity. Výsledek je poté vypsán do konzole a dle konfigurace jsou knihovny uloženy na disk a nahrány do zvoleného repositáře. Prerekvizitou pro tento nástroj je existence package-lock souboru, který nemusí být vždy součástí zabalené aplikace. Některé firemní mikroservisy například, které jsou využívané napříč firmou, jsou baleny bez tohoto souboru, což může také vést k nestabilitě verze. Po vydání nové verze projektu je nezbytné zajistit, aby každá její instalace probíhala stejně. Soubor package-lock také zajišťuje, aby se pokaždé stahovaly stejné verze knihoven, a proto je jej nutné vygenerovat ručně. Pro automatizaci tohoto kroku byl vytvořen bashový script, který má na vstupu tři argumenty:

- $1 – Cesta k adresáři, kde se nachází soubor se zabalenou aplikací
- $2 – Cesta s k souboru npmrc. Jedná se o konfigurační soubor pro správce JavaScript balíčků npm, což je výchozí správce balíčků pro prostředí nodejs. Tento konfigurační soubor je potřeba především kvůli nasměrování npm na privátní registr knihoven [29].
- $3 – Číslo cílové verze aplikace.

Vytvořený script nejprve vybere soubor se zabalenou aplikací ve zvoleném adresáři, který následně rozbalí. Dále vyhledá pomocí příkazu „find“ soubor package.json, ke kterému zkopíruje konfigurační soubor npmrc a spustí instalaci balíčků, která vygeneruje package-lock. Vytvoření soubor následně přibalí zpět do archivu a pro přehlednost upraví původní verzi o postfix s verzí informačního systému, ve které došlo k vytvoření zámku knihoven.

```
#!/bin/bash
TARGET_DIR=$1
SERVER_ZIP_FILE=`echo $TARGET_DIR*-server.zip`

# Rozzipování appboxu
unzip -d $TARGET_DIR $SERVER_ZIP_FILE

# Získání package.json souboru z appboxu
PACKAGE_JSON_LOCATION=`find $TARGET_DIR/*/package.json | sed
's/\/package.json/ /g'`

# Nakopírování .npmrc souboru vedle package.json souboru (pro
použití správného repositáře pro stahování knihoven)
cp $2 $PACKAGE_JSON_LOCATION
# Spuštění příkazu npm install (sestaví aplikaci a vygeneruje
package-lock)
npm install --prefix $PACKAGE_JSON_LOCATION

# Přibalení package-lock souboru do původního archívu
cd $TARGET_DIR
TARGET_ZIP=`echo *-server.zip`
PACKAGE_LOCK_LOCATION=`find ./*/package-lock.json`
zip -ur $TARGET_ZIP $PACKAGE_LOCK_LOCATION

# Aktualizace verze aplikace - přidání postfixu s informací, kdy
byl package-lock vytvořen
ORIG_VERSION=`cat uuApp.json | jq  -r .version |sed 's/-lock.*//'`
cat uuApp.json | jq --arg lockversion $3 --arg orig $ORIG_VERSION
'.version= $orig + "-lock" + $lockversion' | tee uuApp.json
```

Kód 10 Vytvoření package-lock souboru pro zabalenou aplikaci [autor].

Po úspěšném provedení předchozích kroků je možné spustit samotné nasazování aplikací. Jak již bylo zmíněno v sekci s popisem cílové infrastruktury, nasazování probíhá za pomoci služby C3. Byl vytvořen script, který obstarává integraci této služby prostřednictvím rest API. V prvním kroku poskytne potřebné informace o nasazované aplikaci jako je umístění vytvořeného appboxu v Nexus repositáři a její nasazovací konfigurace. Služba C3 poté zahájí nasazování do swarmového clusteru, které probíhá asynchronně na pozadí. Kontrolu stavu nasazení je možné provádět pomocí API příkazu "AppDeployment/deploy/execAsyncStatus", který je scriptem volán v cyklu každou vteřinu, dokud nasazení neskončí (vrácený stav může být buď „RUNNING“ nebo „FINISHED“, případně může skončit chybou). Nad nasazovacím scriptem je opět jeden řídicí script, který tvoří frontu pro nasazení všech požadovaných aplikací.

```
async deploy() {
  console.log("Starting deploy");
// Sestavení vstupu do služby C3
  const data = {
        // Odkaz na alokované zdroje (resource pool)
        uuUri: appEnvConfig.uuSubApp.resourcePoolUri,
        // Cesta k appboxu aplikace (umístěné v Nexus repositáři)
        appBoxUri:
`${nexusConfig.url}/repository/Apps/${appConfig.product}/${appConfig
.version}/uucloud_descriptor${this.getArchivePostfix()}-
${env}.json`,
        // Nasazovací konfigurace aplikace
        config: {...this._getDefaultConfig(),
...appEnvConfig.uuSubApp.uuConfigMap}
  };

  // Spuštění nasazovacího příkazu pomocí služby C3
  let {resultData} = await
this.httpPostC3("AppDeployment/deploy/exec", data);

  // Kontrola statusu nasazení
  const result = JSON.parse(resultData);
  await
this.waitForTaskFinish("AppDeployment/deploy/execAsyncStatus",
result.taskUri);
}
```

Kód 11 Nasazení aplikace [autor].

10 Instalace aplikací

Nově nasazené aplikace nejsou v prvotním stavu připraveny k reálnému použití uživatelem. Je pro ně nezbytné vykonat sadu inicializačních kroků, které uvedou aplikaci do provozu. Mezi hlavní kroky instalace patří například:

- Vytvoření a inicializace instance aplikace
- Vytvoření a inicializace jednoho či více workspaců
- Inicializace aplikační konfigurace
- Napojení aplikace na ostatní mikroservisy
- Nastavení profilů a práv
- Další obchodní logika (vytvoření kódovacích tabulek, nastavení alarmů, naplánování opakujících se úloh …)

Všechny tyto kroky musí proběhnout po sobě v určitém pořadí a nelze mezi nimi přeskakovat. Dalšími požadovanými vlastnostmi jsou schopnost zotavení se z chyb a možnost opakovaného spouštění se stejným výsledkem. Pro splnění těchto požadavků je postupně rozvíjen interní CLI nástroj nesoucí název initdata_loader. Pro zahájení instalace je potřeba nástroji předat dva hlavní konfigurační parametry. Prvním z nich je takzvaný inventář, který uchovává proměnné specifické pro dané prostředí, které jsou následně předávány instalačním scriptům. Inventář musí mimo jiné specifikovat atributy (viz kód číslo 12):

- **identityMap** – Mapa uživatelů, pod kterými se spouští scripty. Obsahuje přihlašovací údaje k získání přístupového tokenu a unikátní identifikátor uživatele (uid).
- **subApps** – Mapa dílčích aplikací obsahující konfigurace jejich instancí a workspaců. Označení daného oddílu je specifikováno kódem, který musí být unikátní v rámci jednoho prostředí. Parametrem uid je pak nastaven výchozí uživatel pro spuštění scriptů odpovídajícího workspacu.
- Dodatečně lze specifikovat libovolné konfigurační parametry dle potřeb instalačních scriptů.

```
const inventory = {
  // Host cílových aplikací
  host: `http://demo.host/`,
  // Mapa uživatelů s přístupovými údaji
  identityMap: {
    authorities: {
      _comment: ["Authorities user"],
      username: "...",
      password: `...`,
      uid: "authorities",
    },
  },
  // Mapa aplikací a jejich konfigurací
  subApps: {
    "demo_application": {
      workspaces: {
        asid: {
          code: `10100000000000000000000000000000`,
          systemIdentityUid: `1-1-0-100`,
          uid: "authorities",
        },
        awid: {
          code: `10100000000000000000000000000010`,
          systemIdentityUid: `1-1-10-100`,
          uid: "authorities",
          config: {
            param1: "..."
          },
        },
      },
    }
  },
};
```

Kód 12 Ukázka konfigurace inventáře [autor].

Druhým parametrem je cesta k samotným instalačním scriptům. Pokud je cílem adresář, nástroj vylistuje všechny soubory na uvedené cestě a spustí je v alfabetickém pořadí. V případě nasměrování na jednotlivý soubor, je vykonána pouze jeho vlastní logika. Spuštění specifické kombinace scriptů lze docílit nadefinováním odkazu na soubor, který vrací pole relativních cest k ostatním scriptům v atributu dataSets. Nástroj poté vykoná scripty dle jejich pořadí ve vytvořeném poli.

Instalační script lze definovat jako funkci napsanou v jazyce JavaScript, do které vstupují dvě proměnné – dříve zmíněný inventář s konfigurací cílového prostředí a kontext, do kterého lze předávat výsledky z předchozích scriptů. Struktura instalačního scriptu je znázorněna na následujícím kódu a jeho výstupní atributy musí obsahovat:

- **subApp** – Cílová aplikace, na které budou vykonány API příkazy. Musí se shodovat s aplikací definovanou v inventáři.
- **workspace** – Požadovaný workspace (instance) aplikace.
- **commands** – Seznam konfigurací pro jednotlivé API volání.
    - **method** – http metoda.
    - **command –** Název koncového bodu, na kterém bude vykonán API příkaz.
    - **authorization –** Uživatel použitý k provolání příkazu.
    - **dtoIn –** Parametry, které budou vstupovat do volání.
    - **allowedErrorCodes** – Seznam očekávaných chyb, pro které není chtěné ukončit celý proces instalace. Tento parametr zajišťuje možnost opakovaného spouštění instalačních scriptů se stejným výsledkem. Typickými příklady jsou chyby způsobené opakovanou inicializací či zakládáním identických produktů.
    - **contextKey –** Specifikace tohoto atributu zajistí uložení výsledku volání na kontext pod zadaným klíčem. Výsledek je však vhodné ukládat pouze v potřebě použití jedním z dalších scriptů v instalačním řetězci.

```
const metaCommand = require("../metaCommands/metaCommand");

module.exports = (inventory, context) => {
  // logika
  return [{
  // Cílová aplikace a workspace
    subApp: "demo_application",
    workspace: "awid",
    commands: [
      {
        method: "GET",//http metoda příkazu
        command: "sys/uuSubAppInstance/init", //volaný příkaz
        authorization: { uid: "authorities" },//použitý uživatel
        // metaCommand: metaCommand,
        dtoIn: {  //vysup do příkazu
          envConfig: inventory.subApps.demo_application.config
        },
        allowedErrorCodes: ["asidAlreadyInitialized"],
        contextKey: "demoInitResult",//kontext pro uložení výsledku
      }, //... nextCommand
    ],
  }];
}
```

Kód 13 Ukázka struktury instalačního scriptu [autor].

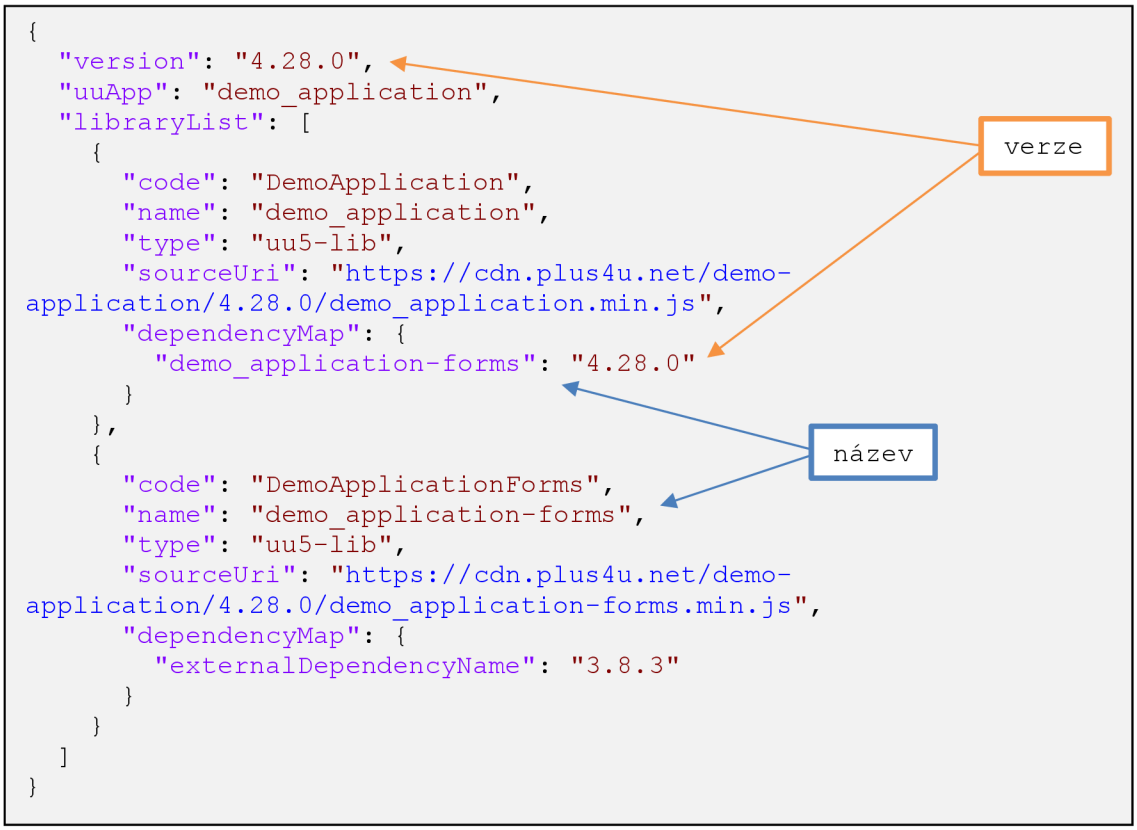
Složitějších operací, ve kterých je požadováno provolání více na sobě závislých příkazů, lze docílit rozšířením předchozí definice parametrem “metaCommand“. Jedná se o zcela novou sadu příkazů, která se vykoná namísto provolání příkazu specifikovaného atributem „command“. V meta příkazech lze navíc definovat libovolnou logiku či transformace nad vstupy a výstupy pro cílové volání obchodních aplikací.

```
async function metaCommand(dtoIn, executeCommand, context) {
  // Provolání příkazu
  let loadedProduct = await executeCommand({
    method: "GET",
    command: "product/get",
    dtoIn: {code: dtoIn.code},
  });
  // další logika, transformace, ... a provolání dalšího příkazu
  await executeCommand({
    method: "POST",
    command: "product/update",
    dtoIn: {
      code: dtoIn.code,
      value: loadedProduct.data.value * 1,21
    },
  });
}
module.exports = metaCommand;
```

Kód 14 Ukázka meta příkazu [autor].

11 Nahrání frontendových knihoven do CDN

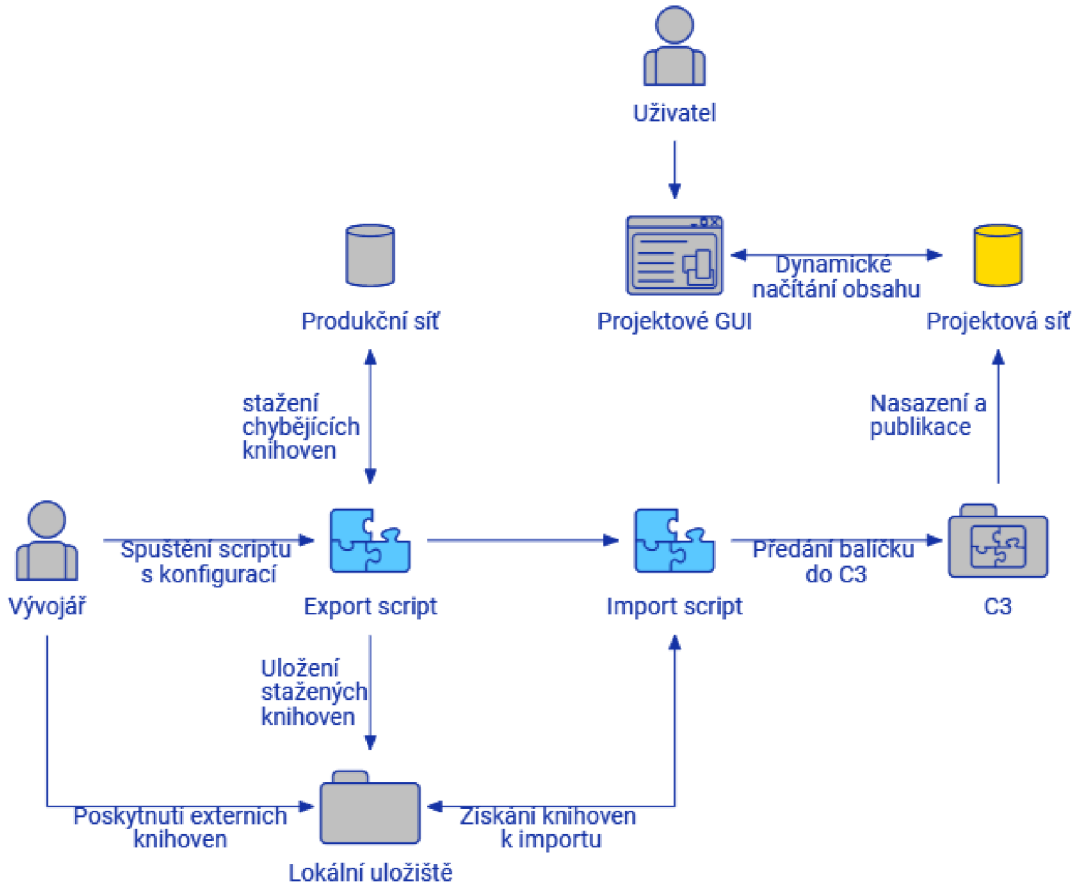
Frontendové knihovny jsou klientskou částí webové aplikace dynamicky načítány v momentu, kdy je aplikace potřebuje použít k vykreslení požadovaného obsahu. Pro vysokou rychlost a dostupnost jsou tyto knihovny umístěny v projektové CDN. Během procesu sestavení frontendové knihovny vzniká její minifikovaná verze a JSON soubor s hlavními informacemi včetně použitých závislostí. Minifikace je proces, při kterém se redukuje velikost kódu tím, že se odstraní zbytečné znaky a mezery, což zlepšuje výkon a rychlost načítání webové aplikace. Při minifikaci může také docházet k přejmenování proměnných a funkcí na kratší názvy.



Kód 15 Ukázka deskriptoru frontendové knihovny [autor].

Pro splnění tohoto kroku jsou vytvořeny dva scripty napsané s pomocí softwarového systému nodejs. První z nich se stará o export požadovaných knihoven a jejich uložení na disk. Pro jeho spuštění je vyžadována mapa knihoven k exportu, která má stejný formát jako parametr dependencyMap z předchozí ukázky. Script nejprve nahlédne do lokálního adresáře s externími knihovnami,

kde je možné ručně poskytnout požadovaný balíček a přetížit tak logiku pro stažení (typicky se používá pro projektové knihovny, které nejsou publikované na veřejné síti). Pokud není knihovna nalezena v lokálním adresáři, je zahájeno její stažení z produkční sítě prostřednictvím http API. Po úspěšném zpracování knihovny je následně zkontrolován její deskriptor a pro každou její závislost (která již nebyla zpracována) je rekurzivně vykonán stejný postup exportu. Druhý script v pořadí slouží k importu stažených knihoven do projektové sítě, která je využívána nasazenými aplikacemi. V deskriptorech je nejprve upraven parametr sourceUri, aby odpovídal projektové síti. Následně jsou balíčky postupně předávány službě C3, která je zodpovědná za jejich správné nasazení a publikaci.

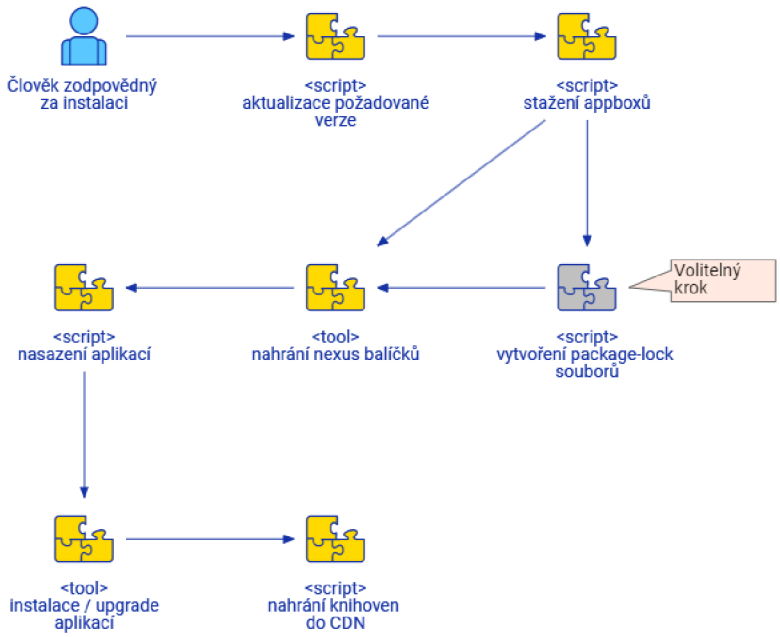


Obrázek 11 Proces nasazení frontendových knihoven [autor].

12 Výsledky a testování

Testování skriptů a nástrojů probíhalo převážně ve fázi vývoje. Nejprve byla provedena analýza daného požadavku, ze které byl vytvořen návrh řešení. Navržená funkcionalita byla postupně vyvíjena a testována na experimentálním prostředí. Po úspěšném dokončení požadavku, byl výsledný nástroj předán do akceptace a do vývojové fáze vstoupil následující krok v pořadí. V opačném případě byly opraveny zjištěné nedostatky a opět testovány.

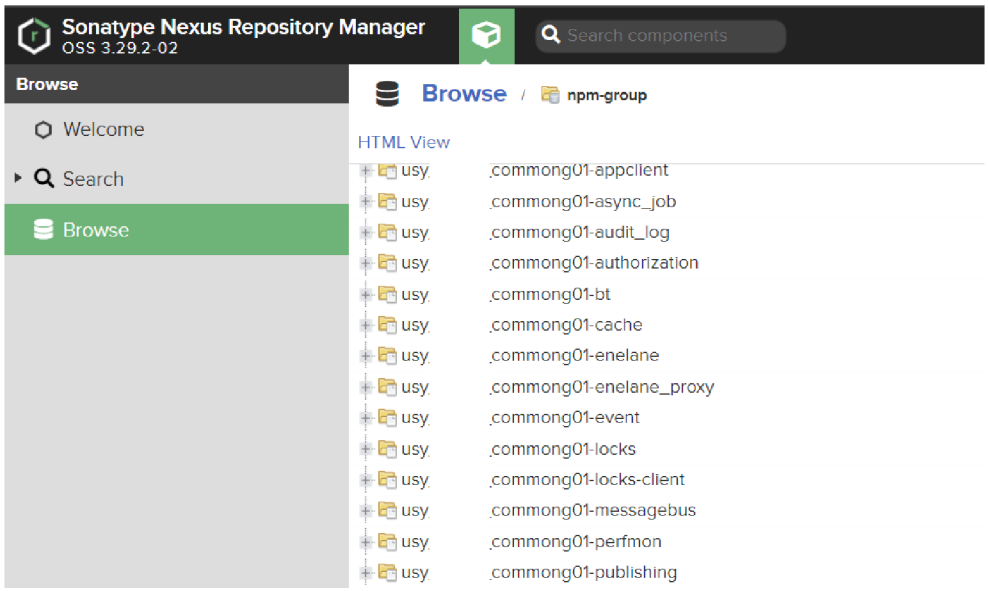
Po akceptaci nástrojů byly upraveny instalační postupy pro cílenou obchodní aplikaci a nadcházející verze byla s jejich pomocí nasazena na UAT prostředí. V současné době je nově vytvořený nasazovací proces používán denně a podléhá průběžným úpravám a vylepšením.



Obrázek 12 Instalační proces [autor].

První dva kroky zajišťují stažení appboxů s požadovanými verzemi na lokální disk, do kterých následující volitelný script přibalí package-lock soubory. Tyto kroky byly řídicím scriptem prováděny pro celý informační systém (36 dílčích aplikací). Následujícím požadavkem je nahrání Nexus balíčků. Tento krok byl spouštěn s filtry ve formě regulárních výrazů, které omezují jeho funkci pouze na projektové

knihovny („usy_*“, „uu_*“, …). Script následně vyhodnotil chybějící knihovny v cílovém Nexus repositáři a postaral se o jejich distribuci.



Obrázek 13 Výsledek nahraných knihoven v cílovém Nexus repositáři [autor].

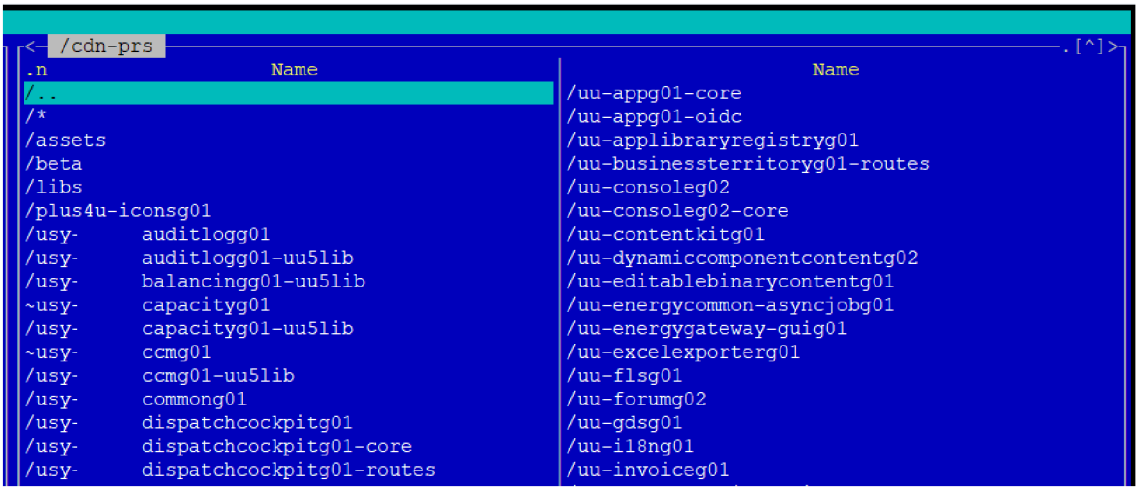
Do navazujícího kroku byl následně předán seznam aplikací k nasazení, pro které bylo za pomoci služby C3 postupně zahájeno nasazování do cílového prostředí, které je spravováno orchestračním nástrojem Docker Swarm. Připojením na odpovídající linuxový server byla následně ověřená správnost nasazení požadovaných služeb.

```
[root@appprsman ~]# docker service ls
ID                  NAME                                                                MODE                REPLICAS            IMAGE
                                                                                                                            PORTS
490vxooqgvcw        apps-usy_    _balancingg01-1669141872557                            replicated          1/1                 mongoa.c
ams:5000/uni_bt-prs-usy_    _balancingg01_cmd-apps:4.30.0-fbd9ef6-202304131146                                             *:30112->8080/tcp
ot89z4lgvdan        apps-usy_    _balancingg01_public-1669141918983                     replicated          1/1                 mongoa.c
ams:5000/uni_bt-prs-usy_    _balancingg01_cmd_public-apps:4.30.0-fbd9ef6-202304131146                                       *:30114->8080/tcp
roi5gena5vit        apps-usy_    _capacityg01-1669142573993                             replicated          1/1                 mongoa.c
ams:5000/uni_bt-prs-usy_    _capacityg01_cmd-apps:4.30.0-e5ff2e4-202304131146                                               *:30115->8080/tcp
ajwnw24fynqz        apps-usy_    _capacityg01_public-1669142620841                      replicated          1/1                 mongoa.c
ams:5000/uni_bt-prs-usy_    _capacityg01_cmd_public-apps:4.30.0-e5ff2e4-202304131146                                        *:30117->8080/tcp
6kc69z6wfx40        apps-usy_    _dispatchcockpitg01-1669143914108                      replicated          1/1                 mongoa.c
ams:5000/uni_bt-prs-usy_    _dispatchcockpitg01_cmd-apps:4.30.0-1daaafb-202304131146                                        *:30120->8080/tcp
t4ax1uiay1az        apps-usy_    _etrmg01-1669144402932                                 replicated          1/1                 mongoa.c
```

Obrázek 14 Seznam nasazených služeb [autor].

Následně byla spuštěna instalace nasazených aplikací pomocí nástroje initdata_loader. Po úspěšném vykonání všech instalačních scriptů byly také importovány frontendové knihovny, které jsou vyžadovány aplikacemi. Po ukončení scriptu byla správnost jejich importu nejprve ověřena připojením na linuxový

server, který obsahuje adresář cdn, ze kterého jsou v provozu dynamicky stahovány klientskou částí systému.



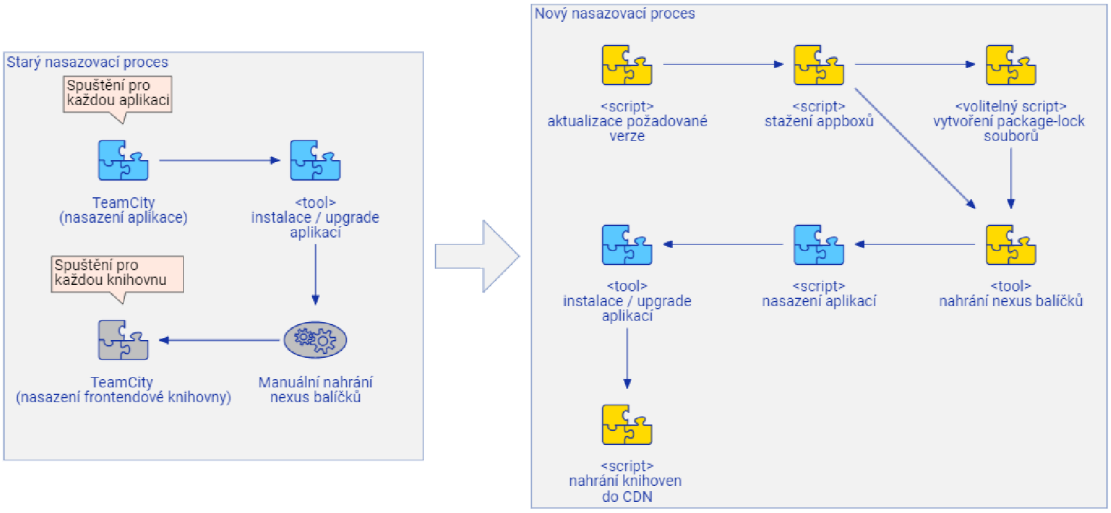
Obrázek 15 Importované knihovny v CDN [autor].

Splněním všech předchozích kroků bylo dosaženo uvedení cílového prostředí do provozu. Aplikace a všechny její části byly dostupné pomocí webového prohlížeče, kde následně proběhlo rychlé otestování funkcionalit a v případě úspěchu bylo prostředí předáno testovacímu týmu.

13 Závěr

Tato práce naplnila všechny své zadané cíle. V první časti seznámila čtenáře s problematikou mikroslužeb doplněnou možnostmi jejich kontainerizace s využitím orchestračních nástrojů Docker Swarm a Kubernetes. Následně byla detailně vysvětlena technologie Cloud Computing spolu s jejími hlavními náležitostmi.

Cílem praktické části byla automatizace přípravy prostředí a vylepšení nasazovacího procesu pro celý informační systém. Hlavním záměrem bylo navrhnout způsob, který co nejvíce odstraní potřebu manuální práce a co nejlépe zefektivní instalaci verzí a tvorbu nových prostředí. Sada kroků, která byla používána pro instalaci před tvorbou této diplomové práce je znázorněna na levé části následujícího obrázku. Žlutě jsou zvýrazněné nové scripty a nástroje, které vznikly v rámci této diplomové práce. Modře jsou označeny kroky, které byly převzaty a použity z předchozího řešení.



Obrázek 17 Změny nasazovacího procesu [autor].

Praktická část práce začíná přiblížením cílové infrastruktury, která je použita pro nasazení informačního systému. Jsou zde objasněny její hlavní aspekty a funkce. Následuje popis nově vytvořených způsobů pro přípravu infrastruktury pro cílové aplikace, které jsou předmětem nasazování a jejich nasazovacích konfigurací.

Použitím těchto nástrojů výrazně zefektivnilo přípravu před začátkem nasazování a minimalizuje nutnost manuálních zásahů.

Další část se zabývá samotným nasazovacím procesem. Scripty pro získání posledních verzí a jejich následné stažení byly úspěšně dokončeny dle návrhu a nyní jsou využívány při každém nasazení. Tvorba package-lock souborů má v aktuální verzi jeden zásadní nedostatek. Jeho spuštění je chtěné vykonat prostřednictvím stejné verze npm jako používá cílová aplikace. Struktura package-lock souborů se může v různých verzích zásadně lišit a aplikace jej nemusí umět použít. V budoucnu je plánováno spouštět tento krok prostřednictvím Dockeru, ve kterém bude nainstalována kompatibilní verze npm.

Nástroj pro nahrání serverových knihoven do cílového Nexus repositáře se úspěšně podařilo dopracovat dle požadavků. Jeho použitím bylo dosaženo úplné eliminace manuálního vyhledávání chybějících knihoven a jejich nahrávání. Pro nasazení obchodních aplikací na prostředí a jejich následnou instalaci byly pouze s menšími úpravami převzaty nástroje z již existujícího řešení. Posledním krokem instalačního procesu je nahrání frontendových knihoven do CDN. Tento krok se podařilo vyřešit exportem všech použitých knihoven a následným importem do cílové CDN. Tento proces se sice osvědčil svou efektivitou pro kompletní nasazení nového prostředí či nové verze, ale vzhledem k jeho časové náročností není vhodný k použití na vývojové prostředí. Rozšířením pro tento script by bylo vhodné umožnit přímý import pouze jedné knihovny například prostřednictvím TeamCity.

Do budoucna by bylo také vhodné řídit celý instalační proces pouze jedním scriptem, který by běžel uvnitř Dockeru. Důvodem tohoto postupu je izolace automatizačních nástrojů od různých infrastruktur a operačních systémů používaných vývojáři. Bude tak zajištěno multiplatformní použití a scripty se budou chovat konzistentně. Spuštěním všech kroků v jednom řídicím scriptu bude navíc docíleno nasazení kompletního informačního systému pouze na jedno stisknutí tlačítka.

14 Seznam použité literatury

[1] FOWLER, M. a LEWIS, J. Microservices a definition of this new architectural term. *martinfowler.com* [online]. [vid. 2023-02-08]. Dostupné z: https://martinfowler.com/articles/microservices.html

[2] NADAREISHVILI, Irakli, Ronnie MITRA, Matt MCLARTY a Mike AMUNDSEN. *Microservice Architecture: Aligning Principles, Practices, and Culture*. B.m.: O'Reilly Media, Inc., 2016. ISBN 978-1-4919-5634-2.

[3] SRIRAMA, Satish Narayana, Mainak ADHIKARI a Souvik PAUL. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications* [online]. 2020, **160**, 102629. ISSN 10848045. Dostupné z: doi:10.1016/j.jnca.2020.102629

[4] VILLAMIZAR, Mario, Oscar GARCÉS, Harold CASTRO, Mauricio VERANO MERINO, Lorena SALAMANCA, Rubby CASALLAS a Santiago GIL. *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud* [online]. 2015. Dostupné z: doi:10.1109/ColumbianCC.2015.7333476

[5] YUSSUPOV, Vladimir, Uwe BREITENBUCHER, Christoph KRIEGER, Frank LEYMANN, Jacopo SOLDANI a Michael WURSTER. Pattern-based Modelling, Integration, and Deployment of Microservice Architectures. In: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)* [online]. Eindhoven, Netherlands: IEEE, 2020, s. 40–50 [vid. 2023-01-08]. ISBN 978-1-72816-473-1. Dostupné z: doi:10.1109/EDOC49727.2020.00015

[6] PAHL, Claus. Containerisation and the PaaS Cloud. nedatováno.

[7] ČERNÝ, Tom, Michael DONAHOO a Michal TRNKA. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review* [online]. 2018, **17**, 29–45. Dostupné z: doi:10.1145/3183628.3183631

[8] PAN, Yao, Ian CHEN, Francisco BRASILEIRO, Glenn JAYAPUTERA a Richard SINNOTT. *A Performance Comparison of Cloud-Based Container Orchestration Tools* [online]. 2019. Dostupné z: doi:10.1109/ICBK.2019.00033

[9] JAWARNEH, Isam Mashhour Al, Paolo BELLAVISTA, Filippo BOSI, Luca FOSCHINI, Giuseppe MARTUSCELLI, Rebecca MONTANARI a Amedeo

PALOPOLI. Container Orchestration Engines: A Thorough Functional and Performance Comparison. In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)* [online]. 2019, s. 1–6. ISSN 1938-1883. Dostupné z: doi:10.1109/ICC.2019.8762053

[10] MORAVCIK, Marek a Martin KONTSEK. *Overview of Docker container orchestration tools* [online]. 2020. Dostupné z: doi:10.1109/ICETA51985.2020.9379236

[11] How nodes work. *Docker Documentation* [online]. 23. únor 2023 [vid. 2023-02-23]. Dostupné z: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

[12] BURNS, Brendan, Joe BEDA a Kelsey HIGHTOWER. Kubernetes: Up and Running. nedatováno.

[13] What is Kubernetes Architecture? *Avi Networks* [online]. [vid. 2023-02-24]. Dostupné z: https://avinetworks.com/glossary/kubernetes-architecture/

[14] Ingress. *Kubernetes* [online]. [vid. 2023-03-05]. Dostupné z: https://kubernetes.io/docs/concepts/services-networking/ingress/

[15] SPILLNER, Josef. *Quality Assessment and Improvement of Helm Charts for Kubernetes-Based Cloud Applications* [online]. B.m.: arXiv. 3. leden 2019 [vid. 2023-03-05]. Dostupné z: http://arxiv.org/abs/1901.00644. arXiv:1901.00644 [cs]

[16] *Docker Swarm vs Kubernetes: Top Differences* [online]. [vid. 2023-04-18]. Dostupné z: https://www.knowledgehut.com/blog/devops/docker-swarm-vs-kubernetes

[17] MARATHE, Nikhil, Ankita GANDHI a Jaimeel M SHAH. Docker Swarm and Kubernetes in Cloud Computing Environment. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)* [online]. 2019, s. 179–184. Dostupné z: doi:10.1109/ICOEI.2019.8862654

[18] Docker Swarm vs Kubernetes: how to choose a container orchestration tool. *CircleCI* [online]. 12. říjen 2021 [vid. 2023-04-18]. Dostupné z: https://circleci.com/blog/docker-swarm-vs-kubernetes/

[19] ALAM, Tanweer. Cloud Computing and its role in the Information Technology. *IAIC Transactions on Sustainable Digital Innovation (ITSDI)*. 2020, **1**(2), 108–115. ISSN 2686-6285 (print), 2715-0461 (online).

[20] BELLO, Sururah A., Lukumon O. OYEDELE, Olugbenga O. AKINADE, Muhammad BILAL, Juan Manuel DAVILA DELGADO, Lukman A. AKANBI,

Anuoluwapo O. AJAYI a Hakeem A. OWOLABI. Cloud computing in construction industry: Use cases, benefits and challenges. *Automation in Construction* [online]. 2021, **122**, 103441. ISSN 0926-5805. Dostupné z: doi:10.1016/j.autcon.2020.103441

[21] RASHID, Aaqib a Amit CHATURVEDI. Cloud Computing Characteristics and Services: A Brief Review. *INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING* [online]. 2019, **7**, 421–426. Dostupné z: doi:10.26438/ijcse/v7i2.421426

[22] MELL, Peter a Timothy GRANCE. The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*. 2011, Special Publication 800-145.

[23] *Cloud Service Model – Understand The Types, Characteristics, & Advantages* [online]. [vid. 2023-02-25]. Dostupné z: https://www.linkedin.com/pulse/cloud-service-model-understand-types-characteristics-peerzade

[24] LASZEWSKI, Tom a Prakash NAUDURI. Chapter 1 - Migrating to the Cloud: Client/Server Migrations to the Oracle Cloud. In: Tom LASZEWSKI a Prakash NAUDURI, ed. *Migrating to the Cloud* [online]. Boston: Syngress, 2012 [vid. 2023-02-25], s. 1–19. ISBN 978-1-59749-647-6. Dostupné z: doi:10.1016/B978-1-59749-647-6.00001-6

[25] SCHOOL OF TECHNOLOGY, ASIA PACIFIC UNIVERSITY OF TECHNOLOGY AND INNOVATION (APU), KUALA LUMPUR, MALAYSIA, Tinankoria DIABY a Babak Bashari RAD. Cloud Computing: A review of the Concepts and Deployment Models. *International Journal of Information Technology and Computer Science* [online]. 2017, **9**(6), 50–58. ISSN 20749007, 20749015. Dostupné z: doi:10.5815/ijitcs.2017.06.07

[26] *Nexus: An open source repository for build artifacts - IBM Garage Practices* [online]. [vid. 2023-03-19]. Dostupné z: https://www.ibm.com/garage/method/practices/deliver/tool_nexus/

[27] *What is a CI/CD pipeline?* [online]. [vid. 2023-03-23]. Dostupné z: https://www.redhat.com/en/topics/devops/what-cicd-pipeline

[28] OTADUY, I. a O. DIAZ. User acceptance testing for Agile-developed web-based applications: Empowering customers through wikis and mind maps. *Journal of Systems and Software* [online]. 2017, **133**, 212–229. ISSN 0164-1212. Dostupné z: doi:10.1016/j.jss.2017.01.002

[29] *npmrc | npm Docs* [online]. [vid. 2023-04-01]. Dostupné z: https://docs.npmjs.com/cli/v9/configuring-npm/npmrc

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Dominik Lohniský**

Studium: I2100070

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Automatizace nasazování komplexní podnikové aplikace založené na architektuře mikroslužeb**

Název diplomové práce AJ: Deployment automation of complex enterprise systems based on microservice architecture

Cíl, metody, literatura, předpoklady:

Cílem diplomové práce je identifikovat potřeby z oblasti nasazování komplexních podnikových aplikací založených na architektuře mikroslužeb, zmapovat možné způsoby řešení a nástroje dostupné na trhu a vyhodnotit jejich silné a slabé stránky. Na základě této rešerše poté navrhnout způsob nasazování konkrétní podnikové aplikace a implementovat funkční proof of concept.

1. Prozkoumat stávající metody a nástroje dostupné na trhu
2. Vyhodnotit existující metody pro potřeby cílové infrastruktury
3. Navrhnout vlastní řešení dle potřeb cílové infrastruktury a aplikací
4. Implementovat návrh řešení
5. Otestovat řešení na vlastním použití
6. Zhodnotit dosažené výsledky

YUSSUPOV, Vladimir et al. *Pattern-based Modelling, Integration, and Deployment of Microservice Architectures*. Eindhoven, Netherlands: IEEE, 2020. ISBN 978-1-72816-473-1. DOI: 10.1109/EDOC49727.2020.00015

GABBRIELLI, Maurizio et al. Self-Reconfiguring Microservices. In: ÁBRAHÁM, Erika, Marcello BONSANGUE a Einar Broch JOHNSEN, eds. *Theory and Practice of Formal Methods*. 9660. Cham: Springer International Publishing, 2016, s. 194–210. Lecture Notes in Computer Science. ISBN 978-3-319-30733-6. DOI: 10.1007/978-3-319-30734-3_14

BRAVETTI, Mario, et al. Optimal and automated deployment for microservices. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 2019. p. 351-368.

SRIRAMA, Satish Narayana, Mainak ADHIKARI a Souvik PAUL. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications*. 2020, roč. 160, s. 102629. ISSN 10848045. DOI: 10.1016/j.jnca.2020.102629

Zadávající pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: Ing. Stanislav Mikulecký

Datum zadání závěrečné práce: 26.1.2021