Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Bezpečnostní hrozby aplikací postavených na principu Microservices

Diplomová práce

Autor: Bc. Edward Zářecký
Studijní obor: Aplikovaná informatika

Vedoucí práce: Mgr. Josef Horálek, Ph.D.

Hradec Králové duben 2022

Prohlášení:

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

V Hradci Králové dne 29. 4. 2022 Edward Zářecký

Poděkování:

Rád bych touto cestou vyjádřil poděkování vedoucímu diplomové práce Mgr. Josefu Horálkovi, Ph.D. za jeho odbornou pomoc, ochotu, čas a trpělivost při vedení mé diplomové práce. Rovněž bych chtěl poděkovat své partnerce a své rodině.

Anotace

Diplomová práce se zabývá softwarovou architekturou mikroslužeb a bezpečnostními hrozbami v rámci této architektury. Diplomová práce je rozdělena na teoretickou a praktickou část. Teoretická část se věnuje popisu softwarové architektury mikroslužeb. Součástí popisu mikroslužeb jsou i používané technologie pro vývoj. V teoretické části jsou pro porovnání popsány i softwarové architektury monolit a Service Oriented Architecture. Poslední kapitola teoretické části se pak zabývá bezpečnostními hrozbami v rámci mikroslužeb. Praktická část se věnuje popisu aplikace využívající softwarovou architekturu mikroslužeb. Popis je zaměřen na obsluhu a fungování aplikace. Součástí praktické části je řešení nalezených bezpečnostních hrozeb. Jejich řešení je implementováno v aplikaci nebo teoreticky popsáno v rámci praktické části.

Klíčová slova

Softwarová architektura, mikroslužby, monolit, Service Oriented Architecture, bezpečnostní hrozby, technologie, aplikace

Annotation

Title: Security threats of applications based on Microservices

The diploma thesis deals with the software architecture of microservices and security threats within this architecture. The diploma thesis is divided into theoretical and practical parts. The theoretical part is devoted to the description of microservices architecture. The description of microservices includes technologies used for development. In the theoretical part, monolithic architecture and Service Oriented Architecture are described for comparison with microservices architecture. The last part of theory focuses on security threats within microservices. The practical part is focused on description of an application using microservices architecture. The description focuses on the control and operation of the application. The practical part includes solutions of the security threats identified. The solutions to the security threats are implemented in the application or theoretically described in the practical part.

Keywords

Software architecture, microservices, monolithic architecture, Service Oriented Architecture, security threats, technologis, application

Obsah

Seznam obrázků

1 Úvod

Informatika představuje odvětví vyvíjející se rychlostí jako žádné jiné. Každý rok přináší celou řadu technických novinek. Z pohledu softwaru vznikají nové programovací jazyky, technologie či nástroje, nemluvě o rozvoji již existujících nástrojů a technologií. Vývoj aplikací se s postupem času také mění. V minulosti probíhal vývoj převážně pouze pro desktopové aplikace. S rozvojem internetu a využívání chytrých telefonů se vývoj přesunul i tímto směrem. V současné době se od desktopových aplikací ustupuje. Naopak vývoj mobilních a webových aplikací roste na popularitě.

Zároveň i volba softwarové architektury pro aplikace se mění v průběhu času. Tyto architektury představují soubor návrhových rozhodnutí v průběhu návrhu i v následném vývoji. Od populární architektury monolitu se v současnosti upouští a vývojáři zkouší hledat nové cesty. V současné době se vývoj aplikací ubírá směrem rozdělení aplikací na menší celky. Architekturou reprezentující tuto myšlenku je například Service Oriented Architecture. Tato architektura s sebou přináší spoustu zajímavých myšlenek a využití. Přestože jsou již v této architektuře využity prvky rozdělení aplikace, jsou zde stále určité závislosti mezi těmito službami. Service Oriented Architecture pro řadu případů nedostačuje, nebo není vhodná. Z tohoto důvodu se začala využívat architektura nazývající se mikroslužby.

Mikroslužby stejně jako zmíněné softwarové architektury nejsou novým pojmem. Diplomová práce se zabývá analýzou mikroslužeb a bezpečnostních hrozeb této architektury. V teoretické části jsou popsány principy mikroslužeb a porovnání s dalšími architekturami. Teoretická část zkoumá výhody i nevýhody této architektury. Následně jsou hledány bezpečnostní hrozby aplikací využívajících tuto architekturu. Praktická část se věnuje aplikaci postavené na mikroslužbách a řešení nalezených bezpečnostních hrozeb.

2 Cíl práce

Cílem diplomové práce je nalezení bezpečnostních hrozeb aplikací postavených na mikroslužbách. Cíle diplomové práce jsou rozděleny na cíle pro teoretickou a praktickou část.

Cílem teoretické části diplomové práce je analýza architektury mikroslužeb a nalezení bezpečnostních hrozeb této architektury. Teoretická část je rozdělena na tři části. Úkolem první části je popis základních pojmů a zkoumání architektur monolit a Service Oriented Architecture. Cílem druhé části je podrobná analýza a popis mikroslužeb včetně porovnání s monolitem a Service Oriented Architecture. Součástí popisu je i nalezení vhodných technologií pro vývoj mikroslužeb. Cílem poslední části teoretické části je najít bezpečnostní hrozby v rámci mikroslužeb.

Cílem praktické části je implementace aplikace využívající architekturu mikroslužeb. Dále je cílem implementovat adekvátní řešení pro nalezené bezpečnostní hrozby, případně alespoň teoreticky navrhnout řešení pro nalezené bezpečnostní hrozby.

3 Základní popis architektur monolitu a SOA

Softwarová architektura představuje soubor návrhových rozhodnutí během vývoje i v následném vývoji. Softwarová architektura se zabývá tím, jak je softwarový systém navržen a vytvořen. Správně vytvořený softwarový systém vychází právě ze softwarové architektury. Dalším důležitým pojmem jsou architektonické styly. Architektonické styly určují, jak má být kód organizován a jak mezi sebou jednotlivé části komunikují. Zároveň představují nejabstraktnější úroveň návrhu systému. Architektonické vzory jsou pak obecná, znovupoužitelná řešení pro opakující se problémy. Architektonické vzory poskytují řešení problémů v architektonickém stylu. Lze je považovat za implementaci architektonických stylů. Návrhové vzory se zaměřují na řešení problému nějaké části kódu. Naproti tomu architektonické vzory se zaměřují na řešení problémů na vyšší úrovni. [1][2]

Softwarová architektura může využívat více architektonických stylů. Každý z těchto architektonických stylů může dále využívat více architektonických vzorů. Použití architektonických stylů a vzorů se uplatňuje pro aplikace desktopové, mobilní i webové. Pro vývoj aplikací existuje celá řada softwarových architektonických stylů a vzorů. Při využívání těchto technik dochází k implementaci rozdílnými způsoby nesoucími s sebou řadu výhod i nevýhod. Každý z těchto architektonický stylů a vzorů přináší jiný pohled na problematiku a má své využití. Nelze proto prohlásit žádný architektonický styl ani vzor za nejlepší, a to právě s odkazem na rozdílná použití a řešení specifické problematiky. [1][2]

Pro srovnání a popis jsou vybrány tři architektury:

- monolit,
- SOA,
- mikroslužby.

V této kapitole jsou popsány a srovnávány dvě architektury monolit a SOA. Popisu mikroslužeb se věnuje následující samostatná kapitola.

3.1 Monolit

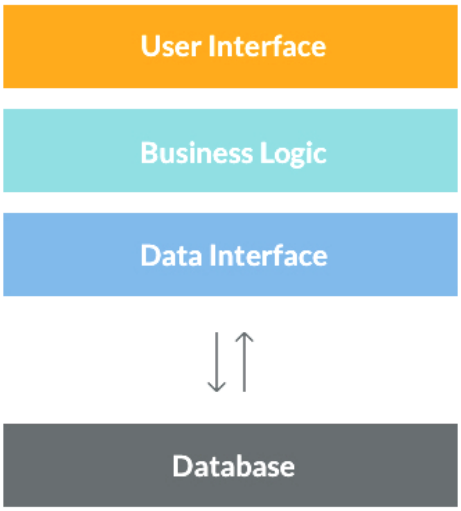
Monolitická architektura patří mezi klasické softwarové architektury. Monolit představuje softwarovou architekturu, která je rozdělena dále na moduly obsahující funkcionalitu pro řešení dané problematiky. Vnitřní rozdělení aplikace pak obsahuje závislosti mezi jednotlivými moduly. Takové závislosti tvoří například knihovny využívané napříč jednotlivými moduly aplikace. Aplikace s touto architekturou jsou zpravidla napsány pouze v jednom programovacím jazyce. Škálování aplikací s monolitickou architekturou s sebou nese problém s velkou granularitou aplikace postavené na monolitu. Velká granularita monolitu je způsobena jedním samostatným dále nedělitelným celkem aplikace. Monolitické aplikace obvykle používají pouze jednu databázi. V této architektuře je náročnější měnit technologie a povyšovat verze. Monolit je vhodnou architekturou pro nově vznikající aplikaci. S rozvíjejícím se systémem lze později změnit architekturu. Monolit lze v případě potřeby například rozdělit na služby podle modulů a použít tak architekturu mikroslužeb. [3]

3.1.1 Vrstvy monolitu

Monolit je rozdělen na tři základní vrstvy:

- prezentační vrstva,
- business logika,
- datová vrstva. [3]

Každá z těchto tří vrstev má svou specifickou roli a zodpovědnost. Komponenty uvnitř každé z těchto vrstev se zabývají pouze logikou uvnitř své vrstvy a žádné jiné. Požadavek od uživatele prochází z prezentační vrstvy přes vrstvu s business logikou až po datovou vrstvu. Vrstvy jsou mezi sebou navzájem izolovány. Tato izolace zamezuje ovlivnění komponent z jednotlivých vrstev. Prezentační vrstvy by neměly mít přímý přístup do datové vrstvy. Tato vlastnost znamená, že jsou jednotlivé vrstvy na sobě nezávislé. Ačkoliv je aplikace složena z těchto tří vrstev, působí jako celek. [3][2]



Obr. 1 – Architektura monolitu [4]

3.1.1.1 Prezentační vrstva

Úkolem prezentační vrstvy je obsluha uživatelského rozhraní. Pomocí prezentační vrstvy uživatel pracuje s aplikací. Vrstva zasílá uživatelské požadavky do vrstvy s business logikou. Prezentační vrstva by neměla mít přímý přístup do datové vrstvy z důvodu izolace vrstev mezi sebou. Mezi nejvíce používané frameworky v této vrstvě řadíme Angular, React, Vue.js a knihovnu jQuery. [3][2]

3.1.1.2 Business logika

Vrstva s business logikou bývá sloučena s perzistentní logikou. Perzistentní logika slouží ke komunikaci mezi business logikou a datovou vrstvou. Pro komunikaci mezi těmito vrstvami lze využít ORM sloužící pro mapování dat mezi relačními databázemi a objektově orientovanými programovacími jazyky. Mezi technologie využívající ORM řadíme například Hibernate nebo iBatis. V business logice je implementována rozhodovací logika a transformace přenášených dat. Další úlohou této vrstvy je pak komunikace mezi prezenční a datovou vrstvou. Uvnitř této vrstvy pro webové aplikace se můžeme setkat s frameworky jako jsou .NET nebo Spring. [3][2]

3.1.1.3 Datová vrstva

Tato vrstva se zabývá požadavky do databáze z vrstvy s business logikou. Datová vrstva disponuje databází sloužící pro plnění SQL požadavků. Tyto požadavky jsou prováděny nativními dotazy SQL nebo jinými jazyky, jako je například HQL z výše uvedené technologie Hibernate. Datová vrstva může mít databázi například relační, objektovou, objektově relační, síťovou, hierarchickou, grafovou. Každý z těchto typů má své využití a práce s daty v nich je rozdílná. Dříve se používaly databáze převážně relační, ale v dnešní době roste zájem o různé NoSQL databáze. [3][2]

3.1.2 Výhody a nevýhody monolitu

3.1.2.1 Výhody

Monolit má řadu výhod, a proto je stále využívanou architekturou. Monolit je doporučován pro začínající programátory nebo jako architektura pro použití v novém projektu. [5][6]

S postupem času lze monolit převést například do SOA nebo mikroslužeb. Díky rozdělení aplikace do modulů jsou již známy řešené problematiky v rámci projektu. Moduly si lze představit jako jednotlivé celky při následném rozdělení aplikace například pro využití architektury mikroslužeb. [5][6]

Výhody využití monolitické architektury:

- jednoduché nasazení aplikace,
- vhodné pro nový projekt,
- jednoduché end-to-end testování,
- jednoduchý počáteční vývoj,
- jednoduchá správa,
- jednoduché pro ladění aplikace,
- lehké horizontální škálování. [3][7]

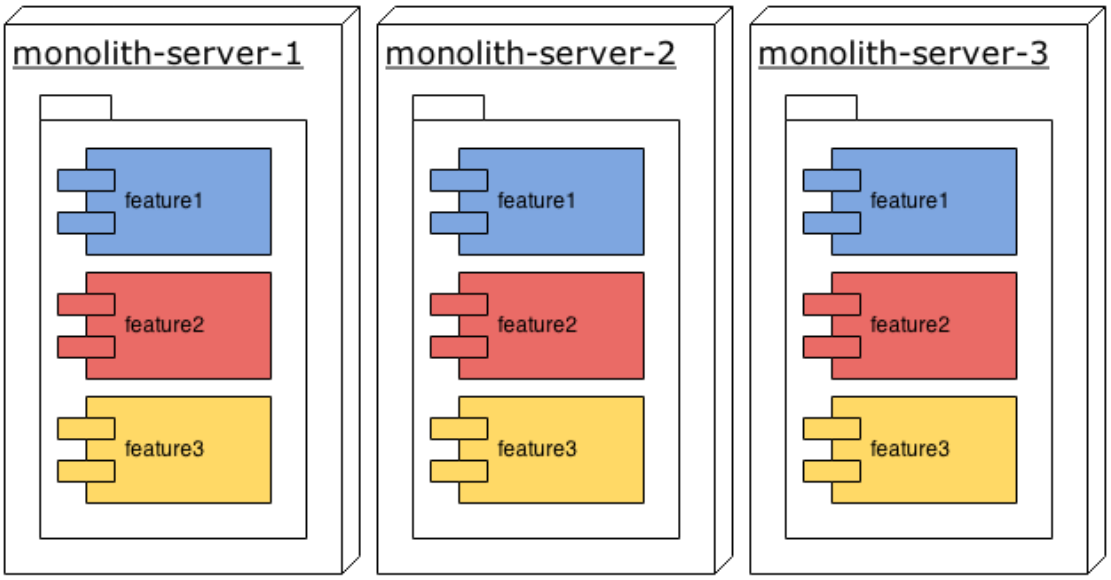
3.1.2.2 Nevýhody

Stejně jako jiné architektury i monolit s sebou přináší i jisté nevýhody. Ty se zvýrazňují s rozsahem projektu. Monolitická architektura tedy začíná být problematická pro rozsáhlé projekty. Provázanost modulů v aplikaci mezi sebou v průběhu vývoje přináší řadu problémů. Začlenění nových programátorů je náročné z důvodu pochopení celé aplikace. S postupem vývoje je aplikace složitá pro pochopení a úpravu. Z tohoto důvodu postupně klesá rychlost vývoje. Změna vybraných technologií nebo povýšení verzí jsou problematické. [7]

Nevýhody využití monolitické architektury:

- využití pouze jednoho programovacího jazyka,
- dlouhá doba sestavení aplikace,
- dlouhá doba otestování celé aplikace,
- při pádu aplikace je celá aplikace nedostupná,
- složité na pochopení celé aplikace,
- nutnost znovu nasazení celé aplikace při aktualizaci,
- úprava kódu v jednom modulu může mít vliv na jiné moduly,
- náročnost adaptace nových technologií,
- pro škálování lze využít pouze replikaci celé aplikace na server. [7]

Jak je již uvedeno v části věnující se nevýhodám této architektury, tak při škálování aplikace je nutné naškálovat vždy celý monolit na server. Na níže uvedeném obrázku je znázorněn příklad škálování aplikace využívající tuto architekturu.



Obr. 2 - Příklad škálování monolitické aplikace [8]

3.2 Service Oriented Architecture

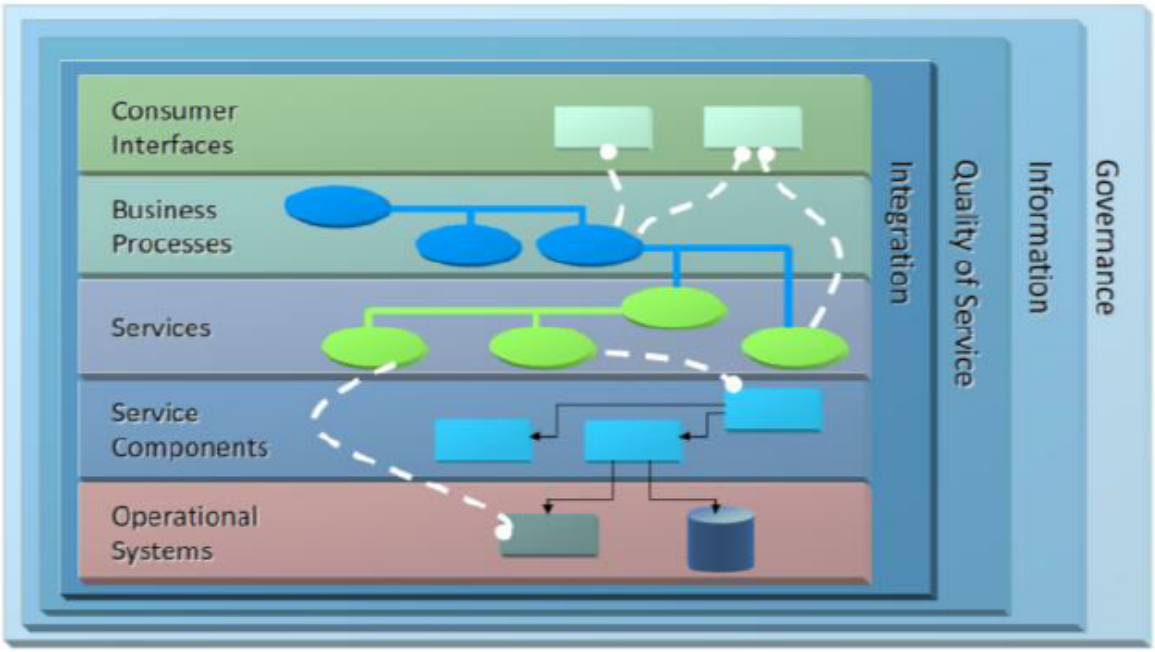
SOA představuje soubor principů určených k sestavení aplikací z nezávislých komponent poskytujících služby. První zmínka o SOA byla v roce 1998. Od tohoto roku roste SOA popularita i vývoj SOA samotné. V roce 2007 vydala The Open Group popis SOA. Tato architektura má více než jednu oficiální definici. V SOA jsou služby poskytovány aplikacím prostřednictvím síťového volání přes internet. Komponenty jsou mezi sebou navzájem nezávislé. SOA umožňuje opakované použití komponent prostřednictvím rozhraní služeb. Připojení komponenty je rychlé a jednoduché. Komponenty dodržují svůj kontrakt mezi producentem a konzumentem. [9][10][11][12][13]

Implementace a logika služby je skryta pod rozhraním služby. Rozhraní služeb je často definováno ve WSDL. Tuto architekturu není vhodné využívat pro malé a jednoduché systémy. Naopak SOA má smysl použít pro komplexní systémy. Komponenta poskytuje řadu funkcí. Funkce vždy odpovídají business logice služby, v niž jsou implementovány. Služby mají na starost například správu uživatelů nebo validaci platby. Databázové úložiště je sdíleno mezi službami. Služby nevyužívají svou databázi, jako je tomu například u mikroslužeb. SOA přímo nesouvisí s žádným nástrojem ani technologií. SOA a mikroslužby jsou odlišné architektury. [9][10][11][12][13]

3.2.1 Vrstvy SOA

Architektura SOA se skládá z pěti horizontálních vrstev:

- rozhraní pro spotřebitele,
- business vrstva,
- vrstva služeb,
- vrstva komponent služeb,
- vrstva provozních systémů. [12]



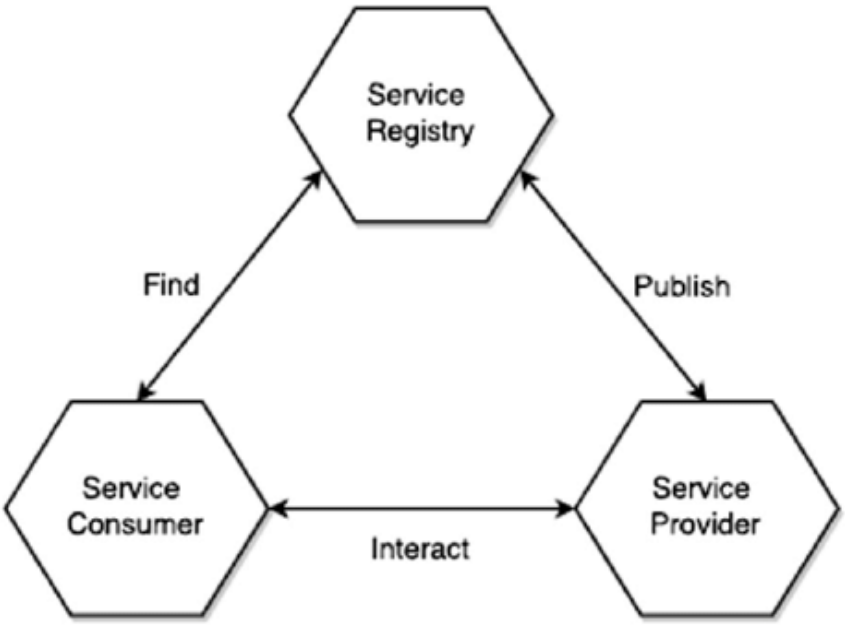
Obr. 3 – SOA rozdělení na vrstvy [12]

Na výše zobrazeném obrázku je znázorněno jednotlivé rozdělení vrstev SOA. Rozhraní pro spotřebitele je reprezentováno GUI, skrze které uživatelé komunikují se službami. Úkoly business vrstvy jsou business případy užití a business procesy. Vrstva služeb se stará o popis a propojení služeb, které budou použity v době běhu. Vrstva komponent služeb se skládá z rozhraní a knihoven. Vrstva provozních systémů zahrnuje datové modely a úložiště dat. [12]

Dále dělíme SOA na čtyři druhy služeb:

- business,
- enterprise,
- aplikační,
- služby infrastruktury. [14]

Každá z těchto služeb obsahuje svou vnitřní logiku. Architekturu SOA lze rozdělit na poskytovatele služeb, konzumenta služeb a registr služeb. Na obrázku níže je znázorněno schéma tohoto rozdělení. Implementace každé ze služeb vychází ze společného standardu pro všechny služby. [9][10][11][14]



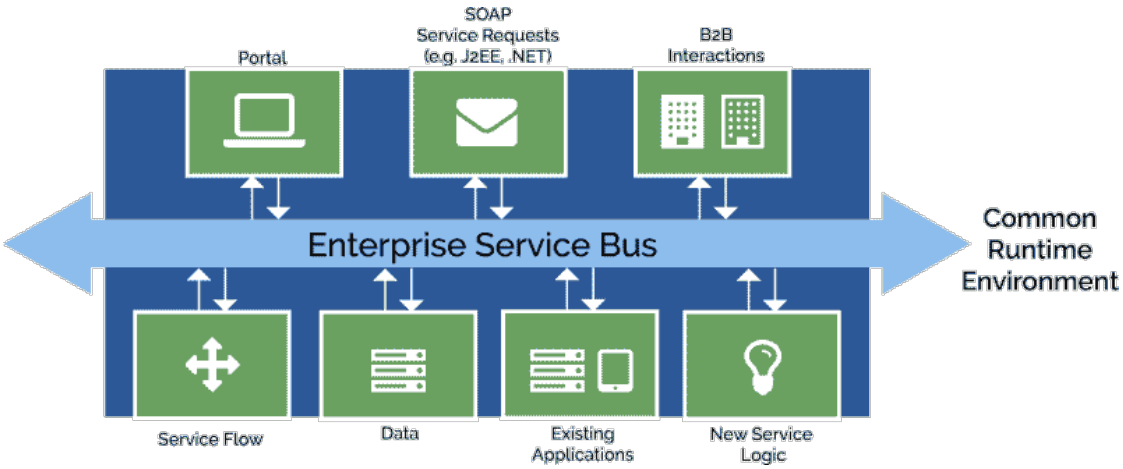
Obr. 4 – Architektura SOA [15]

Poskytovatele služeb si lze představit jako správce. Tento správce zpřístupňuje jednu či více služeb pro ostatní. Poskytovatel služeb tyto služby zveřejňuje do registru služeb s patřičnými požadavky. Mezi tyto požadavky patří kontrakt služby, bezpečnost, použití samotné služby a další. S konzumentem služeb komunikuje skrze odpovědi na jeho požadavky. Konzument služeb hledá metadata v registru služeb. Po nalezení se konzument služeb spojí s poskytovatelem služeb. Konzument služeb může komunikovat s více službami. Konzument služeb zasílá poskytovateli služeb požadavky, na které poskytovatel služeb odpovídá. [9][10][11]

Registr užeb představuje zdroj neustále se aktualizujících metadat o jednotlivých službách. Spojení mezi poskytovatelem služeb a spotřebitelem služeb vytváří registr služeb. Díky tomuto spojení vzniká komunikace mezi spotřebitelem služeb a poskytovatelem služeb. Na tomto registru závisí, jak úspěšná bude komunikace mezi jednotlivými službami. [9][10][11]

3.2.2 Propojení komponent

Služby využívají ke svému vystavení standardní síťové protokoly. SOAP komunikuje přes síť se službami a využívá k tomu formát XML. Pro komunikaci SOAP se nejvíce využívá protokol HTTP. Aplikace se skládá z řady komponent a jejich služeb. Služby mohou být nové či pouze zveřejňovat rozhraní převzatých systémů. Pro komunikaci a propojení nejen s převzatými systémy se využívá ESB. ESB transformuje datové modely, zajišťuje konektivitu, provádí směrování zpráv, převádí komunikační protokoly a případně řídí složení více požadavků. ESB patří mezi architektonické vzory. ESB lze považovat za páteř celé SOA. Níže na obrázku je znázorněn příklad ESB. SOA lze využívat i bez ESB, ačkoliv to přináší celou řadu nevýhod, a to například problémy údržbou. ESB provede nezbytnou transformaci a směrování pro připojení ke službě staršího systému. Připojení provádí skrze konektor nebo adaptér. ESB odstíní aplikaci od rozhraní starších systémů.



Obr. 5 – ESB [16]

Vlastnosti ESB:

- směrování,
- bezpečnost,
- logování,
- monitorování,
- management služeb,
- transformace dat. [17]

3.2.3 Výhody a nevýhody SOA

3.2.3.1 Výhody

Služba může být znovu využita v jiné aplikaci. To má velkou výhodu pro vývoj nové aplikace. Z důvodu znovupoužitelnosti klesá jak časová, tak i finanční náročnost pro nové projekty. Složitost jednotlivých služeb je nízká. Tato vlastnost je důležitá pro vývoj a správu služeb. [9][10][11]

Výhody použití SOA:

- znovupoužitelnost,
- snižuje náklady pro vývoj další aplikace,
- dostupnost,
- spolehlivost,
- jednoduchá správa jednotlivých služeb,
- škálovatelnost,
- nezávislost na technologiích mezi službami. [9][10][11]

3.2.3.2 Nevýhody

Při prvním využití této architektury je nutná rozsáhlá investice. Nutnost implementovat a pochopit nové metody zabere více času než u výše zmíněného monolitu. Služby si mezi sebou mohou vyměňovat až miliony zpráv. Velký počet zpráv může vést ke složitějšímu zpracování požadavků. [9][11]

Nevýhody použití SOA:

- vysoké investiční náklady,
- vysoká režie,
- vyšší složitost správy. [9][10][11]

4 Mikroslužby

4.1 Historie

První zmínky o této architektuře se datují k roku 2005. V tomto roce byl na konferenci představen termín mikro webové služby Dr. Peterem Rodgersem. Ve své prezentaci popisoval základní model a koncept mikro webových služeb. V roce 2011 se konala konference v Itálii, na které byl představen nový pojem mikroslužba. O rok později byl tento termín upraven, a to na nový termín mikroslužby. Tento termín se zachoval až do současnosti. Mikroslužby jsou v současnosti běžně využívanou softwarovou architekturou, které neustále roste popularita. [18][19]

Velké společnosti využívají architekturu mikroslužeb již řadu let. Mezi tyto společnosti lze zařadit například Amazon, eBay, Facebook, Google, Netflix nebo Twitter. Například Google pro své vyhledávání využívá desítky mikroslužeb, než je dosaženo výsledku. Popularita této architektury roste i díky podpoře DevOps, nezávislosti na technologiích nebo možnosti kontejnerizace. V budoucnu se očekává růst využití mikroslužeb v nových i stávajících aplikacích. [18][19]

4.2 Popis

Základním principem mikroslužeb je aplikace skládající se z mnoha malých autonomních služeb. Tyto autonomní služby mezi sebou komunikují a pracují jako jeden celek. Služby jsou mezi sebou nezávislé, a to platí i o jejich nasazování. Jednotlivé služby využívají vlastní technologie i databáze. Vývoj jednotlivých služeb je na sobě nezávislý. Mezi další charakteristiku mikroslužeb patří decentralizace. Komunikace mezi službami probíhá za pomocí REST API, streamování událostí nebo zprostředkovatele zpráv. Každá služba se zabývá vlastní logikou a díky tomu jsou na sobě nezávislé. Logika uvnitř těchto služeb by neměla být komplikovaná. Jednoduchost služby je velkou výhodou při její správě. Správou služby je míněna údržba, vylepšení či přidání nové funkcionality. [5][20]

Při výpadku jedné ze služeb stačí znovu nasadit pouze jednu službu, nikoliv celou aplikaci. Popis mikroslužeb je podobný s výše zmíněnou architekturou SOA. Některé zdroje považují mikroslužby za jednu z variant SOA. [5][20]

Detailnější popis mikroslužeb a porovnání těchto dvou architektur v podkapitole bude demonstrovat rozdíly mezi nimi. Doporučení při použití mikroslužeb jsou pokročilá znalost a práce s DevOps a agilním řízením projektu. Mikroslužby by měly být vyvíjeny v iterativním procesu. Nedoporučuje se sdílet knihovny a SDK mezi službami. Každá ze služeb může implementovat nezávislý zabezpečovací mechanismus. Jednotlivé služby by měly být monitorovány a poskytovat možnosti řešení problémů při potížích. [5][20]

4.2.1 Charakteristické vlastnosti služeb v mikroslužbách

V této kapitole bude uvedeno a popsáno několik typických charakteristik pro služby v mikroslužbách. Díky uvedeným charakteristikám si lze představit fungování a způsoby vývoje služeb.

Mezi tyto charakteristiky patří:

- decentralizace,
- autonomní vývoj,
- nezávislé nasazení,
- ohraničení kontextu,
- nasazení a sestavení pomocí automatizovaných procesů,
- malá velikost,
- vhodně navržené API. [6]

4.2.1.1 Decentralizace

Mikroslužby jsou rozmanité počtem využívaných technologií. Z tohoto důvodu centrální správa není vhodná při použití této architektury. Mikroslužby naopak využívají decentralizaci. Decentralizace mikroslužeb se dělí na dvě části.

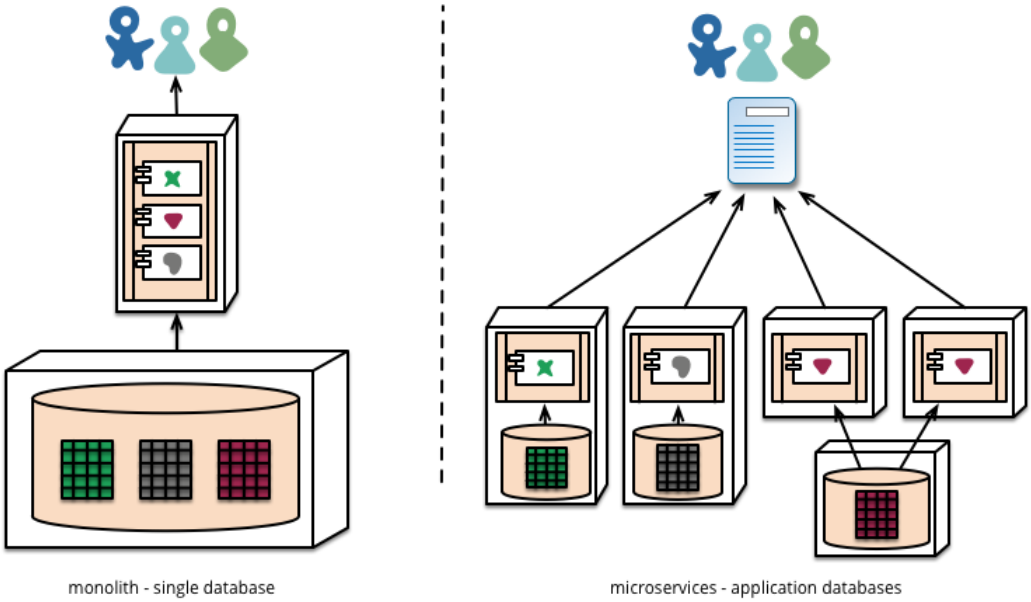
Decentralizovaná správa

Decentralizovaná správa je důležitou vlastností pro vývojáře využívající softwarovou architekturu mikroslužby. Tato vlastnost dává vývojářům svobodu při vývoji softwarových komponent. Vývojářské týmy využívající mikroslužby dávají

přednost různým přístupům i standardům. V důsledku toho vývojáři upřednostňují vývoj užitečných nástrojů řešících problematiku, kterou mohou využít další vývojáři pro řešení stejné nebo podobné problematiky. Největší výhodou decentralizované správy je vývoj, správa a případný rozvoj jedním vývojářským týmem. Tým stojící za vývojem služby se obvykle i nadále stará o její správu. [21][22]

Decentralizovaná správa dat

Obdobně jako u správy dochází k decentralizaci i ve správě dat. Decentralizovaná správa dat znamená, že každá služba obvykle pracuje pouze se svou databází. Zde nastává velký rozdíl oproti monolitu. V monolitu je obvyklé využívat pouze jednu databázi. Vznikají menší databáze pro konkrétní službu místo jedné rozsáhlé pro celou aplikaci. Velkou výhodou je rychlost a zároveň jednodušší správa menší databáze než jedné velké. Problém ovšem nastává pro dodržení konzistence napříč těmito databázemi. Na níže uvedeném obrázku je znázorněn rozdíl mezi databázemi využitých v mikroslužbách a monolitu. [21][22]



Obr. 6 – Porovnání práce s databází v monolitu a mikroslužbách [22]

4.2.1.2 Autonomní vývoj

Služby se vyvíjí jako samostatné celky a jsou navzájem nezávislé. Velkou výhodou této charakteristiky je možnost využití vhodných technologií pro každou službu. Pro každou z deseti služeb mohou být využity zcela jiné technologie. Použití konkrétních technologií záleží na požadovaných vlastnostech služby a v neposlední řadě na vývojářském týmu. Každý vývojářský tým má své preference pro zvolení technologií. Mezi požadované vlastnosti může například patřit rychlost, bezpečnost, dlouhodobá podpora. Preference pro technologie mají na základě svých zkušeností a znalostí z předchozího vývoje.

Jako jednoduchý příklad lze uvést dva týmy, které vyvíjí v jazyce Java anebo C#. Vývojářský tým používající jazyk Java bude pravděpodobně chtít vyvíjet v tomto programovacím jazyce na backendu a použít framework Spring. Naopak druhý tým bude preferovat programovací jazyk C# a framework .NET. Stejný příklad lze uvést pro databáze, frontend, technologie pro komunikaci a další. Jsou týmy využívající relační databáze Oracle, MySQL a naopak týmy, které upřednostňují NoSQL databáze, jako je třeba MongoDB nebo Neo4j. V současnosti existuje velké množství technologií. Technologie může zaniknout a tím se může ukončit i její podpora. Využívat technologie, které ztratily svou podporu je nevhodné. Nevhodnost použití technologie s ukončenou podporou má hned několika důvodů, avšak hlavním důvodem je bezpečnost. Každá technologie s sebou přináší řadu výhod, ale i nevýhod. Zvolení vhodných technologií je nelehký úkol.

4.2.1.3 Nezávislé nasazení

Nezávislé nasazení je velice důležitou vlastností této architektury. Nasazením je míněno zpřístupnění aplikace do produkce. Typickou vlastností monolitu je znovu nasazení celé aplikace při výpadku nebo změně aplikace. Znovu nasazení celé aplikace je velice náročné jak z pohledu režie a správy, tak i z pohledu znepřístupnění celé aplikace v tomto čase. Naopak nezávislé nasazení přináší velkou výhodu oproti znovu nasazení celé aplikace. Hlavní výhodou nezávislého nasazení je znovu nasazení pouze nějaké části aplikace myšleno služby či několika málo služeb.

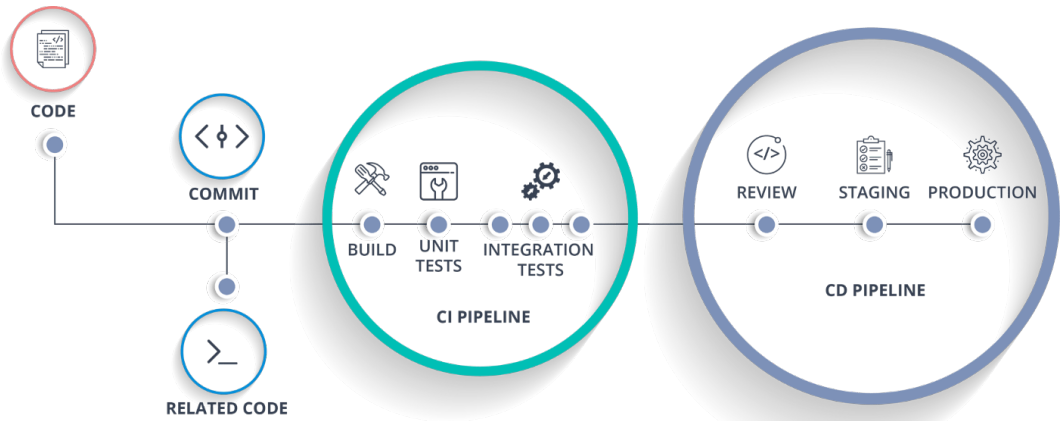
Při výpadku nebo potřebě znovu nasadit službu je nedostupná pouze část aplikace. Přidání funkcionality do služby nebo v případě její nedostupnosti stačí znovu nasadit službu. Není tedy potřeba znovu nasadit celou aplikaci, jako je tomu v případě aplikace využívající architekturu monolitu.

4.2.1.4 Ohraničení kontextu

Eric Evans představil v roce 2003 Domain-driven design. Hlavním záměrem Domain-driven designu je vývoj doménového modelu. Doménový model disponuje znalostí celé domény a především jeho procesů a pravidel. Tento termín byl představen dříve než mikroslužby samotné. Ohraničenost kontextem je základním stavebním kamenem v Domain-driven designu. Mikroslužby jsou vysoce kompatibilní s Domain-driven designem. Domain-driven design se zaobírá rozsáhlými modely a jejich rozdělením do ohraničených kontextů. Ohraničenost kontextem si lze představit jako jednu uzavřenou specifickou zodpovědnost. Každý ohraničený kontext má vlastní explicitní rozhraní. Skrze explicitní rozhraní umožňuje ostatním kontextům získat poskytnuté informace v tomto explicitním rozhraní. Určení správného ohraničení kontextu je vhodná cesta k určení hranic služby. [6]

4.2.1.5 Nasazené a sestavení pomocí automatizovaných procesů

Automatizace procesů je odvětví, které v posledních letech velmi rychle roste. Vhodnou platformou pro mikroslužby je cloudová infrastruktura. Cloud a především Amazon Web Service výrazně urychluje celý proces provozování, sestavení a nasazení mikroslužeb. Jak již bylo zmíněno, pro vývoj je vhodné disponovat znalostí DevOps. Především je užitečné znát a využívat CI/CD pipeline. CI se zabývá nalezením problémů v sestavení a odhalením problémů během jednotkových a integračních testů. CD se zabývá automatizovaným nasazením otestovaného kódu z CI do produkce. Díky CI/CD pipeline a jejím procesům sestavení, testování a nasazení je celý proces vývoje rychlejší a spolehlivější. Celý tento proces je automatizován a není potřeba žádných dalších manuálních kroků. Na uvedeném obrázku pod odstavcem je znázorněn CI/CD pipeline s celým jejím procesem zpracování. [22]



Obr. 7 – CI/CD pipeline [23]

4.2.1.6 Malá velikost

Zvolení vhodné velikosti jak služby samotné, tak i vývojářského týmu ji spravujícího, není přesně definováno. V Amazonu je pro velikost týmu využito pravidlo dvou pizz. Pravidlo dvou pizz říká, že celý tým si u oběda vystačí se dvěma pizzami. Velikost týmu by neměla být větší než osm lidí. Při větším počtu lidí v týmu klesá efektivita. Tým o velikosti osmi lidí je tedy maximální doporučený. Pravidlo dvou pizz je obecné pravidlo, nikoliv specifické přímo pro službu. Nicméně lze na něm dobře demonstrovat vhodnou velikost vývojářského týmu pro službu. Malá velikost služby přináší řadu výhod, přičemž největší výhodou je rychlost vývoje a menší počet problémů při vývoji. Služba by se neměla zabývat rozsáhlou problematikou. Při zjevném jednoduchém oddělení logiky je vhodné vytvořit novou službu. Z toho plyne, že služba by neměla být rozsáhlý projekt vyvíjený řadu měsíců pro nasazení. Přesnou velikost služby ani vývojářského týmu nelze říct exaktně. Služba musí být jednoduchá pro správu a vývojářský tým musí být efektivní. Vývojářský tým má na starost jednu či více služeb. [22]

4.2.1.7 Vhodně navržené API

Služba v mikroslužbách komunikuje s ostatními. V opačném případě vytvoření takové služby nedává smysl. Služba má své API nebo rozhraní využívající pro komunikaci. Rozhraní nebo API služeb musí být nezávislé na ostatních. Závislost na ostatních s sebou nese problémy například při nasazení služeb. [24]

Závislost služeb mezi sebou je v rozporu s vhodným návrhem služeb v této architektuře. Pro návrh vhodně vytvořených API odpovídajícím nezávislosti je rozdělujeme na dva druhy. [24]

Rozdělení API v mikroslužbách:

- Message-orinted,
- Hypermedia-driven. [24]

Message-oriented

Message-oriented je nejúčinnějším způsobem, jak upravovat služby a neovlivnit tím spolupráci s ostatními částmi systému. Zároveň umožňuje bezpečné refaktorování. Zasílání zpráv umožňuje sdílení dat mezi službami. Za efektivní způsob se považuje nahlížet na komplexní systém jako na soubor služeb vyměňující si data prostřednictvím zpráv. Tento návrh umožňuje vytvořit vstupní bod pro službu a zároveň přijímat zprávy specifické pro danou úlohu. Netflix využívá pro svou interní komunikaci zprávy ve formátech Avro, Protobuf a Thirft. Všechny tyto formáty využívají pro komunikaci TCP/IP. Pro externí komunikaci využívá JSON přes HTTP. [24]

Hypermedia-driven

Hypermedia-driven rozšiřuje myšlenku message-oriented. Zprávy nesou více informací než pouhá data. Zprávy obsahují popis akcí nebo stavů. Amazon API gateway využívá pro odpovědi HAL. Hypermedia API má dle svého návrhu nezávislé vazby na ostatních. Hypermedia API je také označováno jako Hypermedia as the Engine of Application state API. HTTP zprávy jsou zaslány na port a IP adresu. Data a akce ve zprávě jsou ve formátu HTML. [24]

4.3 Technologie pro vývoj

Možnost použití velkého počtu frameworků a technologií pro mikroslužby vede i k celé řadě otázek, zejména jaké technologie zvolit. Tato kapitola se bude zabývat popisem několika aktuálně používaných frameworků a technologií v praxi.

Pro ukázku tu bude uvedeno několik technologií nebo frameworků pro každou kategorii. Blíže popsány budou pouze technologie a frameworky použité v praktické části diplomové práce. Pokud zde nebude nějaká technologie uvedena, neznamená to, že není vhodná pro použití. Technologie pro vývoj služeb budou pro všechny služby stejné. Nicméně lze využívat rozdílné technologie v každé službě.

Databáze:

- MongoDB,
- Coudant,
- MySQL,
- PostgreSQL,
- Neo4J,
- Redis.

4.3.1 MySQL

Vychází ze standardu SQL a využívá relační databázový model. Mimo to pracuje i s kolekcemi JSON. MySQL využívá CRUD operace pro práci se záznamy v tabulkách. Verze vychází ve dvou licencích GNU a komerční. Původním autorem je švédská společnost MySQL AB. MySQL vzniklo v 90. letech minulého století. Firma MySQL AB byla v roce 2010 koupena firmou Sun Microsystems. Po prodeji Sun Microsystems patří i MySQL AB v současnosti firmě Oracle Corporation. [25][26]

Mezi vlastnosti MySQL řadíme rychlost, jednoduchost a práce s rozsáhlými datovými soubory. Právě díky těmto vlastnostem se používá pro malé i rozsáhlé projekty. MySQL funguje na řadě operačních systémech jako jsou macOS, Linux, Microsoft a UNIX. MySQL splňuje atomicitu, konzistenci, trvalost i izolovanost pro své transakce. Mezi podporované programovací jazyky náleží například PHP, C, C++, C#, Ruby, Python, Perl. MySQL se řadí mezi nejpopulárnější databáze současnosti. Pro návrh a modelování databáze v MySQL existuje nástroj MySQL Workbench. Nicméně s postupem času ztrácí na své popularitě. MySQL je obvykle využíváno pro menší projekty. Aktuální stabilní verze 8.0.27 vyšla 19. října 2021. [25][26]

Řešení objektově relačního mapování:

- Hibernate,
- iBatis,
- TopLink,
- Django ORM,
- Doctrine.

4.3.2 Hibernate

Hibernate je objektově relační framework určený pro programovací jazyk Java. Hibernate má na starost perzistenci dat, mapování objektů Java do databázových tabulek a převod datových typů mezi programovacím jazykem Java a SQL. Hibernate se vyznačuje spolehlivostí, škálovatelností, vysokým výkonem a rozšiřitelností. Tento framework je jednou z implementací JPA. První verze frameworku Hibernate byla vydána 23. května 2001 autorem Gavinem Kingem a řadou dalších vývojářů. O další vývoj frameworku se starala společnost JBoss, která jej prodala v roce 2006 společnosti Rad Hat. Rad Hat dále rozvíjí tento framework. Hibernate je licencován pod LGPL. Hibernate využívá HQL vycházející ze standardního SQL. HQL je na rozdíl od SQL objektově orientovaný dotazovací jazyk. HQL pracuje s perzistentními objekty a jejich atributy. Dotazy z HQL jsou přeloženy do SQL a zpracovány databází. Poslední stabilní verze 5.6.3 vyšla 16. prosince 2021. [27][28]

Backend programovací jazyky:

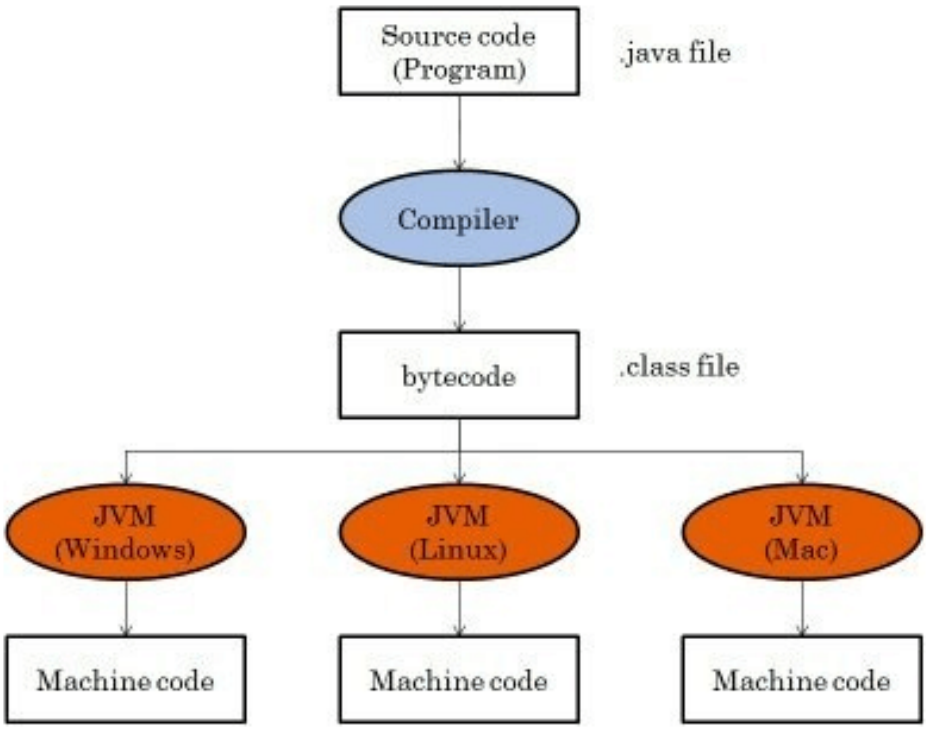
- Java,
- Kotlin,
- C#,
- Python,
- Go,
- Ruby,
- PHP.

4.3.3 Java

Java je objektově orientovaný, multiplatformní a silně typový programovací jazyk. Tento programovací jazyk vznikl v roce 1995. Java byla vyvinuta společností Sun Microsystems. [29]

Společnost Sun Microsystems koupila společnost Oracle Corporation v roce 2009. Java se řadí mezi deset nejpopulárnějších programovacích jazyků současnosti. Mimo popularitu jazyka je i zároveň vyhledáván mezi zaměstnavateli. Java funguje na operačních systémech jako je macOS, Windows a řadě distribucí Linuxu. Mezi jeho přednosti řadíme práci s většími objemy dat, poměrně jednoduché naučení se jazyka a v neposlední řadě bezpečnost. Java bytecode představuje sadu instrukcí pro JVM. [29][30][31]

Java bytecode vzniká při kompilaci a vznikají soubory s příponou *.class. Právě díky Java bytecodu je Java multiplatformní. JVM je běhové prostředí pro spuštění bytecodu. Na níže zobrazeném obrázku je znázorněna ukázka bytecodu a JVM. [30][31]



Obr. 8 – Java bytecode a JVM [30]

Platformy jazyka Java:

- Java Standard Edition,
- Java Enterprise Edition,
- Java Card,
- Java Micro Edition,
- JavaFX.

Nové verze jazyka vychází každých šest měsíců. Verze vychází v březnu a následně v září. Mezi používanými verzemi jsou verze 7, 8 a 11. Tři zmíněné verze jazyka jsou hojně používané především díky LTS. Aktuální verze jazyka Java je verze 17. Verze jazyka Java 17 vyšla 14. září 2021. Jedná se o verzi s dlouhotrvající podporou neboli LTS. Právě díky LTS bude i Java verze 17 patřit mezi značně využívané verze i v budoucnosti. [31]

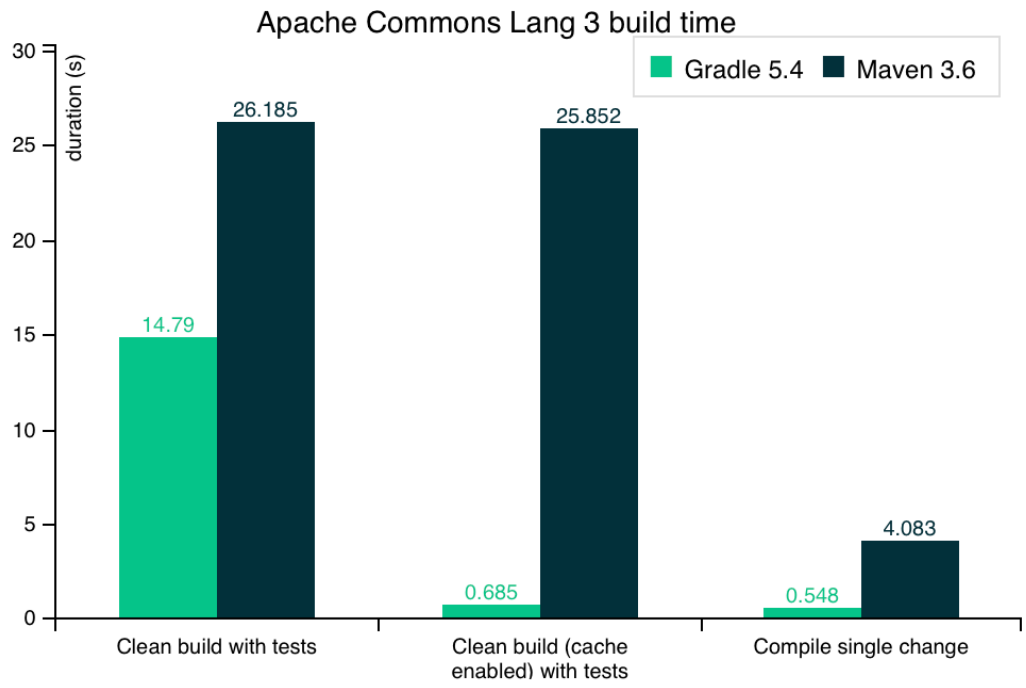
Sestavovací nástroje:

- Apache Maven,
- Gradle,
- Apache Ant,
- Bazel,
- Cake,
- Meson.

4.3.4 Gradle

Gradle je sestavovací nástroj vyvíjený jako open-source. Je vhodný pro sestavování jak mobilních aplikací, tak i mikroslužeb. Gradle je nástroj určený pro automatizované sestavování aplikací. Tento nástroj běží na JVM. K použití Gradlu je nutné mít nainstalované JDK. První verze Gradlu vznikla v roce 2008. Za jeho vývojem stojí společnost Gradle Inc. Gradle se skládá z úloh sestavujících skripty. Sestavující skripty mohou být naprogramovány buď v Groovy nebo Kotlinu. Gradle spouští pouze nezbytně nutné úlohy. Právě tato vlastnost stojí za rychlostí Gradlu. Gradle je rychlejším nástrojem než Apache Maven. Oproti Apache Maven se konfigurace Gradlu provádí v DSL místo XML. Mezi vývojová prostředí podporující

Gradle patří IntelliJ IDEA, NetBeans, Eclipse a Android Studio. Gradle je oficiálním sestavovacím nástrojem pro Android aplikace. Poslední stabilní verze 7.3.3. vyšla 22. prosince 2021. [32][33]



Obr. 9 – Porovnání rychlosti sestavení Gradle a Apache Maven [33]

Backend frameworky:

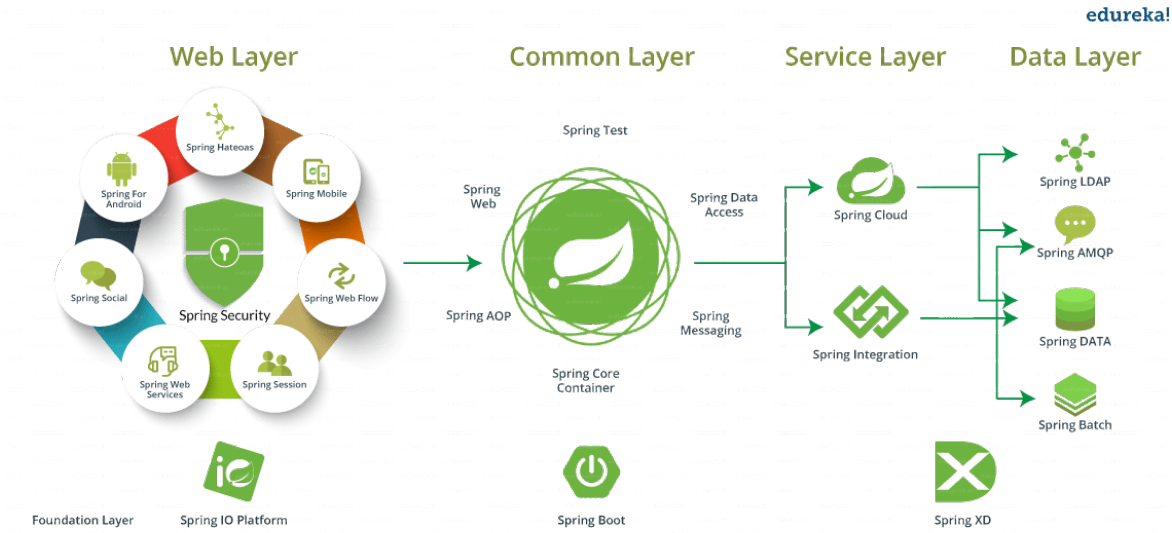
- Spring,
- .Net,
- Vaadin,
- Django,
- Laravel,
- Flask.
- Meteor,
- Ruby on Rails.

4.3.5 Spring

Spring Framework je DI kontejner s několika dalšími vrstvami běžící na platformě Jakarta EE. Díky tomu je bezpečnost, vývoj, rychlost a produktivita výrazně vyšší při samotném programovaní v programovacím jazyce Java. Spring je vůbec nejpopulárnějším frameworkem pro programovací jazyk Java. Spring Framework je využíván nejen pro programovací jazyk Java. Spring Framework vznikl 1. října 2002. Jeho autorem je Roderick Johnson. Později Roderich Johnson spolu s dalšími založili společnost SpringSource. [34][35]

SpringSource stojící na Spring Frameworku byla v roce 2009 prodána společnosti VWware. V roce 2013 VMware a EMC Corporation vytvořili společnost Pivotal Software. Ta vznikla pro řadu softwarových produktů od VMware zahrnující například Spring. Ten nabízí 21 projektů tvořících jeden velký ekosystém. Je modulární a lze jednotlivě kombinovat. Pro ukázku je níže přiložen obrázek s přehledem několika Spring projektů. Spring umožňuje reaktivní programování, vývoj mikroslužeb, vývoj webových aplikací, cloudová řešení, řízení událostmi a další. [34][35]

Nejznámějšími projekty jsou Spring Framework, Spring Boot, Spring Data, Spring Cloud, Spring Security. Poslední stabilní verze Spring Frameworku 5.3.15 vyšla 13. ledna 2022. V diplomové práci bude využit Spring Boot, Spring Cloud, Spring Security a Spring Data. Spring Boot umožňuje vytvořit jednoduchou samostatnou Spring aplikaci. První verze Spring Bootu 1.0 vznikla 1. dubna 2014. Aktuální stabilní verze Spring Bootu 2.6.2 vyšla 5. ledna 2022. Spring Cloud je framework určený pro sestavení robustních cloudových aplikací. Spring Security je framework starající se o autentizaci a autorizaci pro Java aplikace. Spring Data je projekt od Spring určený pro přístup k datům v SQL i NoSQL databázích. V praktické části diplomové práce se bude využívat Spring Boot, Spring Cloud, Spring Cloud Gateway, Spring Security, Spring Data JPA. [34][35]



Obr. 10 – Spring projekty [36]

Frontend programovací jazyky:

- JavaScript,
- TypeScript,
- jQuery.

4.3.6 TypeScript

TypeScript je typový objektově orientovaný programovací jazyk. Tento jazyk vyvinula a nadále spravuje společnost Microsoft. Samotný TypeScript vznikl v první veřejné verzi v roce 2012 a je vyvíjen pod licencí Apache License 2.0. Jedná se o nástavbu nad programovacím jazykem JavaScript. Hlavní výhodou TypeScriptu oproti JavaScriptu je typovost jazyka. Mezi další vlastnosti TypeScritpu patří anotace, genericita a používání asynchronního programování. TypeScript umožňuje plnou spolupráci s knihovnami JavaScriptu. Veškerá syntaxe z JavaScriptu je validní i v TypeScriptu. TypeScript je při kompilaci převeden do JavaScritpu. Právě díky tomu funguje TypeScript všude, kde funguje i JavaScript. Kompilace TypeScriptu nalezne i případné syntaktické chyby. Kód v TypeScriptu je spolehlivější skrze využití typů a také je jednoduší pro údržbu. V roce 2020 byl TypeScript zvolen jako druhý nejoblíbenější programovací jazyk mezi programátory v anketě od StackOverflow. Poslední stabilní verze 4.5.5 vyšla 20. ledna 2022. [37]

Frontend frameworky:

- Angular,
- React,
- Vue.JS,
- Mithril,
- Riot.

4.3.7 Angular

Angular je TypeScriptový framework pro frontend. Primárně je Angular určen pro SPA. Angular vznikl v roce 2012. Framework Angular byl vyvinut a je spravován společností Google. Angular je jiný framework než AngularJS. Původně byl Angular nazván jako Angular 2.0 z důvodu odlišení od AngularJS. AngularJS lze označit jako předchůdce Angular. Hlavním rozdílem je podpora pouze JavaScriptu v AngularuJS. Z AngularJS se s postupem času stal velice komplikovaný a nepřehledný framework. Od 1. ledna 2022 Google oficiálně nepodporuje AngularJS. Oficiálním doporučením je přechod na nový Angular. [38]

Angular pro své správné fungování využívá DOM. DOM je strom skládající se z jednotlivých uzlů reprezentujících jednotlivé části dokumentu. Při změně v projektu Angular aktualizuje všechny HTML značky. Angular se skládá z komponent skládaných do sebe. Mimo komponenty se Angular skládá z direktiv, služeb, pipe, šablon dekorátorů. Komponenta je obvykle složena z HTML šablony a TypeScriptové třídy označené dekorátorem @Component(). Součástí komponenty může být i soubor s CSS. V dekorátoru @Component() se specifikuje selektor, šablona HTML a případné CSS. HTML šablona určuje zobrazení komponenty. HTML šablona se použije dvěma způsoby. V případě méně obsáhlého kódu v HTML šabloně lze zapsat přímo do TypeScriptové třídy v dekorátoru @Component(). Pro obsáhlejší zápis v šabloně se přidá cesta k samostatnému souboru v dekorátoru @Component(). Angular používá dva typy formulářů. Formuláře se dělí na reaktivní a řízené šablonou. Oba z těchto dvou typů mají svá specifická využití. [38]

Pro reaktivní programování se v Angularu využívá knihovna RxJS. RxJS usnadňuje práci s asynchronním kódem a kódem založeným na zpětném volání. Aktuální stabilní vydaná verze 13.1.3 je z 19. ledna 2022. [38]

Technologie pro vytvoření API:

- REST,
- GraphQL,
- gRPC,
- SOAP.

4.3.8 REST

REST představuje softwarový architektonický styl pro distribuované hypermediální systémy. REST byl představen v roce 2000 autorem Royem Fieldingem v jeho disertační práci. REST se skládá z vrstev. Komunikace probíhá na bázi klient – server. REST umožňuje cachování. REST API je webové rozhraní API odpovídající architektonickému stylu REST. Klient může obdržet odpověď v několika formátech. Využívanými formáty jsou XML a JSON. Klient pošle požadavek, na který server následně reaguje. Nejvíce využívaným protokolem pro komunikaci je HTTP. [39]

Požadavek se skládá ze čtyř částí:

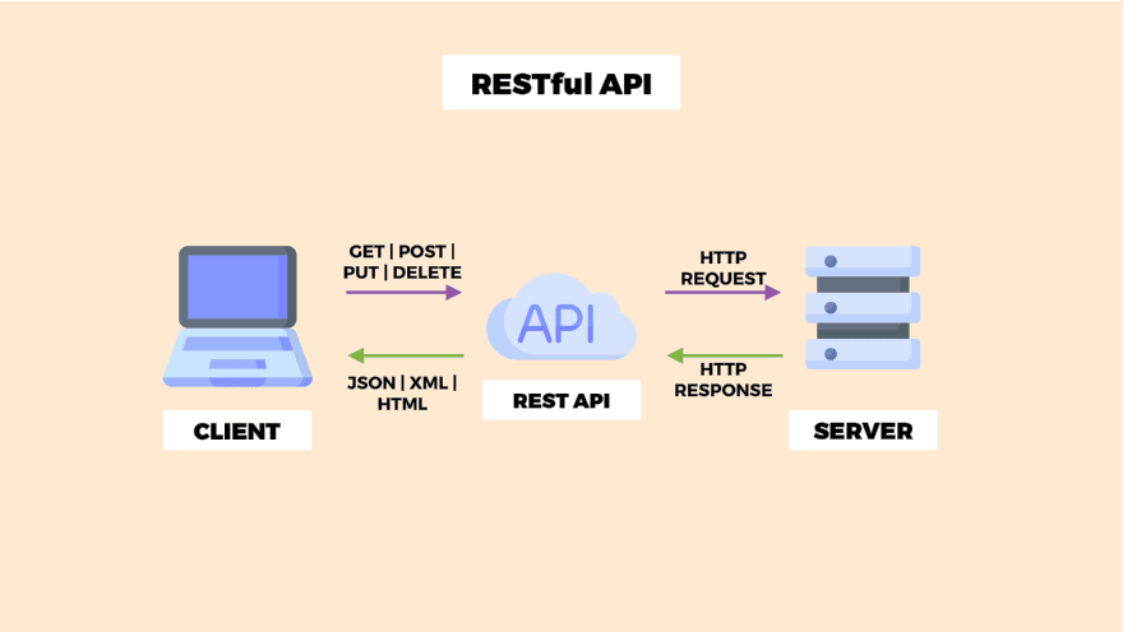
- URI,
- hlavička,
- tělo,
- HTTP požadavek. [39]

URI je určen k jednoznačnému volání příslušného zdroje v API. Hlavička obsahuje informace o požadavku. Tělo požadavku je volitelné a obsahuje data. Obvykle se využívají požadavky bez těla. HTTP požadavky typu GET neobsahují tělo. Naopak HTTP požadavky typu POST a PUT tělo obsahují.

Součástí odpovědi jsou i HTTP stavové kódy. HTTP stavové kódy jsou vždy reprezentovány trojciferným čísly. HTTP stavové kódy se dělí na pět kategorií.

Informační stavové kódy v rozsahu 100–199. Stavové kódy v intervalu 200–299 známé jako úspěšné. Stavové kódy obsahující zprávy o přesměrování v rozsahu 300–399. Chybu u klienta reprezentují stavové kódy v intervalu 400–499. Chybě na serveru odpovídají stavové kódy v rozsahu 500–599. Mezi nejznámější HTTP stavové kódy patří 200 OK a 400 špatný požadavek. [39]

HTTP požadavky rozdělujeme na DELETE, GET, POST a PUT. HTTP požadavek GET slouží pro získání dat na základě požadavku. Požadavek POST slouží pro zaslání a následné uložení nového záznamu. PUT se používá pro editaci již existujícího záznamu a jeho následné uložení. PUT nelze použít pro uložení neexistujícího záznamu. HTTP požadavek DELETE je určen pro smazání záznamu typicky na základě jeho identifikátoru. [39]



Obr. 11 – Ukázka komunikace skrze REST API [40]

Nástroje pro komunikaci:

- Apache Kafka,
- RabbitMQ,
- ActiveMQ,
- Apache Storm,
- Redis.

4.3.9 RabbitMQ

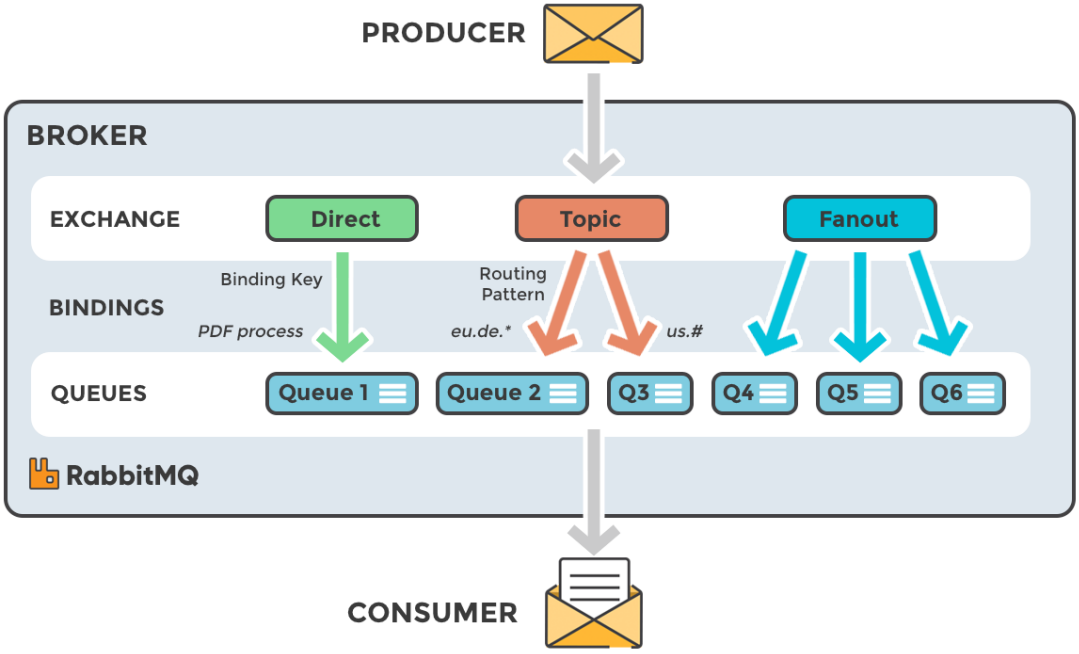
RabbitMQ je message broker vyvíjený jako open source a původně vycházející z AMQP. RabbitMQ vznikl v roce 2007. Za vývojem RabbitMQ stojí společnost Rabbit Technologies. V roce 2013 se tato společnost stala součástí Pivotal Software. RabbitMQ byl vyvinut v jazyce Erling a využívá OTP. Zároveň se řadí mezi nejpoužívanější message brokery současnosti. RabbitMQ podporuje jak řadu operačních systémů, tak i celou řadu programovacích jazyků. V podporovaných programovacích jazycích nalezneme například programovací jazyk Java, C#, Python, Ruby, Go nebo Swift. Základním principem fungování RabbitMQ je producent a konzument. Producent vytváří zprávu a konzument ji zpracovává. Poslední verze 3.9.13 vyšla 19. ledna 2022. Mezi producentem a konzumentem RabbitMQ nabízí čtyři režimy směrování zpráv. Zprávy jsou směrovány do front zpráv. [41]

Režimy směrování zpráv:

- fanout,
- topic,
- direct,
- headers. [41]

Fanout funguje na principu rozeslání zpráv do všech navazujících front. Směrování typu direct je nejjednodušším typem směrování. Zpráva obsahuje klíč, dle kterého je vybrána konkrétní fronta.

Složitějším typem je směrování typu topic. Podobně jako směrování direct pracuje s klíčem. S klíčem se v tomto případě pracuje odlišným způsobem. Klíč se zpracovává na základě regulárních výrazů a může být přesměrován do více front. Posledním typem směrování je směrování na základě hlaviček. Ve směrování typu headers se využívají atributy hlavičky zpráv pro směrování. Na uvedeném obrázku na další straně jsou pro představu zobrazeny tři z těchto typů směrování zpráv. [41]



Obr. 12 – RabbitMQ se třemi strategiemi směrování zpráv [42]

Technologie pro kontejnerizaci:

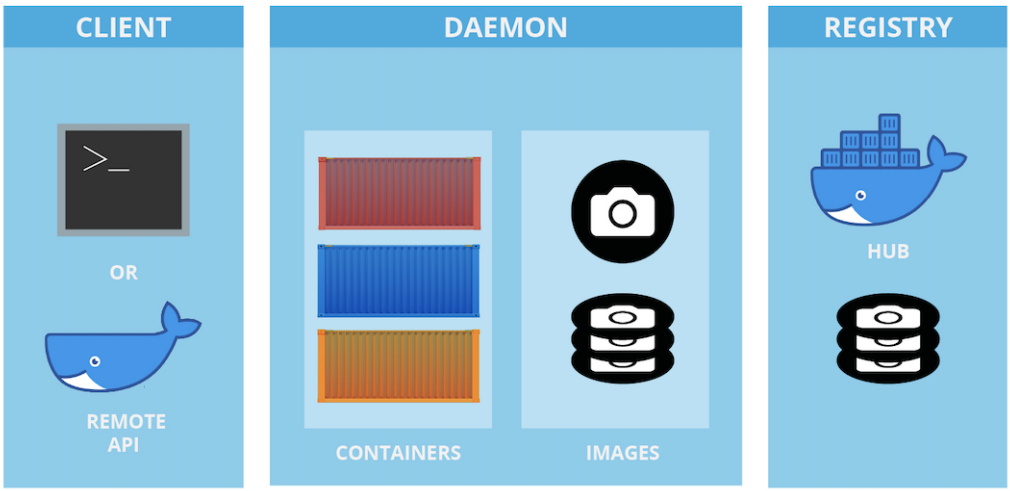
- Docker,
- Containerd,
- Podman
- OpenVZ,
- ZeroVM,
- RunC.

4.3.10 Docker

Docker je platforma určená pro správu, nasazení a vytváření kontejnerových aplikací vyvíjená jako open source. První verze vyšla v březnu 2013. O vývoj a správu se stará Docker Inc. Docker byl vyvinut v programovacím jazyce Go. [43][44]

Docker se skládá z klienta, deamona a registrů. Docker vychází z linuxových kontejnerů. Linuxové kontejnery rozšířil o jednodušší správu nebo přenositelné image. Kontejnerizace a virtualizace jsou dvě odlišné věci. Virtualizace funguje jako samostatný server s vlastním operačním systémem. Oproti tomu kontejnerizace

využívá virtualizaci jádra operačního systému. Díky tomu se zdroje jako například operační paměť a další využívají mnohem efektivněji. Kontejner neobsahuje samotný operační systém. Dockerfile je soubor instrukcí pro sestavení Docker image. Docker image je šablona pouze pro čtení skládající se z instrukcí pro sestavení Docker kontejneru. Docker image jsou ukládány v registrech Dockeru. Docker image lze stahovat z Docker Hubu. Docker Hub je služba pro vytváření, správu a sdílení jednotlivých Docker image. Docker Compose je nástroj v příkazovém řádku sloužící pro správu služeb. Docker má na starost celý životní cyklus kontejnerů. Docker podporuje Windows, macOS i Linux. [43][44]



Obr. 13 – Architektura Dockeru [44]

Orchestrační nástroje:

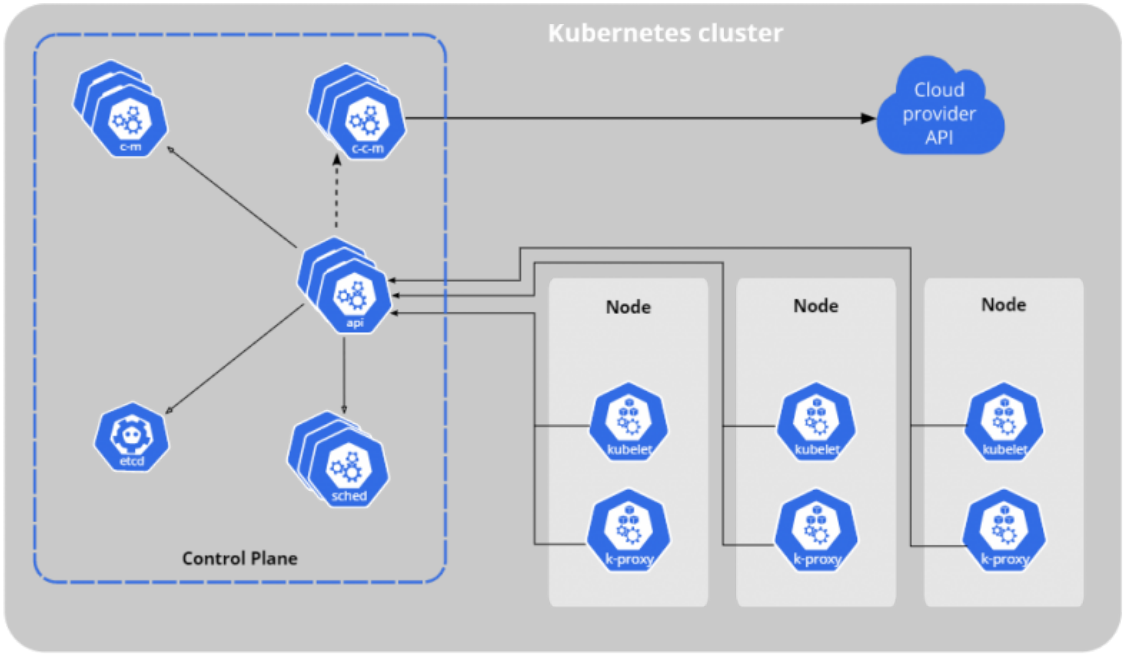
- Kubernetes,
- RedHat OpenShift,
- Nomad,
- Amazon ECS,
- Amazon Fargate.

4.3.11 Kubernetes

Kubernetes je systém pro škálování, automatizované nasazování a správu kontejnerizačních aplikací. Kubernetes se označuje jako K8s. Je vyvíjen pod licencí Apache a vznikl v roce 2014. Za vývojem Kubernetes stojí společnost Google.

O správu a vývoj Kubernetes se stará Cloud Native Computing Foundation. Kubernetes je uspořádán do clusteru. Součástí Kubernetes clusteru jsou uzly. Ten může být buď fyzický nebo virtuální stroj. [45][46]

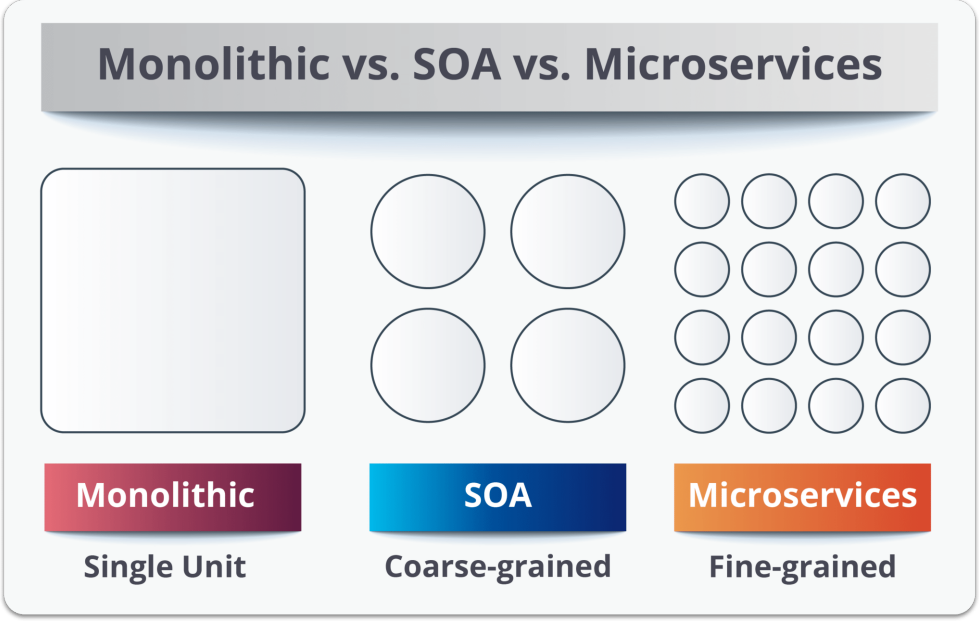
Každý uzel obsahuje pody. Pod představuje nejmenší jednotku pro nasazení a správu v K8s. Pody pracují ve dvou režimech. V případě režimu s více kontejnery se musí spouštět na stejném logickém hostiteli. Pody poskytují síťové prostředky a sdílené úložiště. Každý pod má svou IP adresu. Pod spouští jeden kontejner nebo více kontejnerů závislých na sobě. O dohled nad pody a uzly se stará řídící rovina. K8s podporuje IPv4 i IPv6. Kubernetes je schopen restartovat, znovu nasadit, odpojit kontejnery nebo nepropagovat plně funkční kontejnery. Kubernetes v případě problému je schopen se navrátit do původního stavu. Komunikace v K8s mezi clusterem a koncovým uživatelem probíhá skrze Kubernetes API. Kubernetes je vyvíjen v programovacím jazyce Go. Aktuální verze Kubernetes 1.23.3 vyšla 25. ledna 2022. [45][46]



Obr. 14 – Kubernetes cluster [46]

4.4 Porovnání architektur

V této kapitole budou porovnávány následující architektury – monolit, SOA a mikroslužby. Podrobnější popis všech těchto tří architektur je uveden v předchozích kapitolách. Na obrázku pod odstavcem je zobrazen rozdíl mezi těmito architekturami z pohledu samostatných celků. Monolit funguje jako jeden samostatný celek, který se dále nerozděluje na menší celky. Naproti tomu SOA disponuje menšími celky nazývajícími se služby. Služby mezi sebou komunikují dvěma způsoby. První způsob je skrze jednoduché posílání dat. Druhý způsob je řízení aktivity pomocí dvou či více služeb. Služby v SOA jsou navíc vždy propojeny. Mikroslužby jsou ještě menší samostatné nezávislé služby než služby v SOA. Tyto samostatné nezávislé služby jsou vždy ohraničeny svým kontextem. V následujícím textu budou porovnávány architektury monolit a SOA s mikroslužbami.



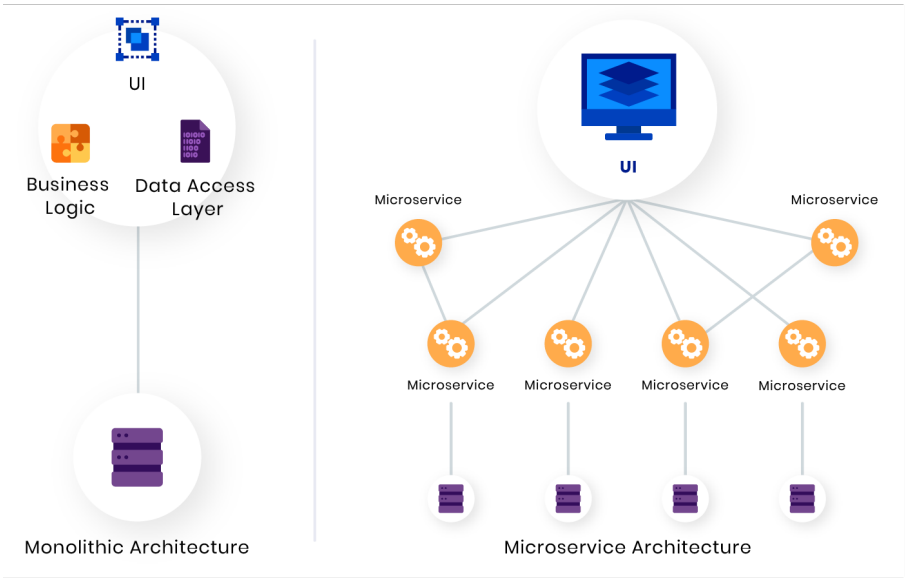
Obr. 15 – Porovnání monolitu, SOA a mikroslužeb [47]

4.4.1 Mikroslužby a monolit

Monolit je vhodnou architekturou pro většinu nově vznikajících aplikací. Novou aplikaci je jednodušší nejdříve vytvořit jako jeden celek a až potom případně rozdělit na menší celky. Naopak není vhodné postavit nově vznikající projekt a využít architekturu mikroslužeb. [4]

Monolitické aplikace pracují s jednou databází. Oproti tomu mikroslužby obvykle využívají vlastní databázi pro každou službu. Správa, provoz, testování a logování monolitu je jednodušší, protože se vždy jedná pouze o jeden celek. Složitost stejných operací u mikroslužeb je vyšší z důvodu většího počtu služeb. Pochopení a úprava aplikace postavené na architektuře monolit je náročnější, než je tomu u aplikace postavené na mikroslužbách. Služby z mikroslužeb jsou naopak velice jednoduché pro pochopení a správu. Škálování monolitu je neefektivní a musí se vždy naškálovat celá aplikace. U mikroslužeb lze naškálovat jednotlivé služby. Při výpadku aplikace postavené na monolitu je mimo provoz celá aplikace. Při výpadku mikroslužeb dojde k výpadku pouze konkrétní služby a zbytek aplikace funguje. Znovupoužitelnost monolitické aplikace nebo pouze nějaké části je složité. Jednotlivé části aplikace postavené na architektuře monolitu jsou značně provázány mezi sebou. Provázanost je příčinou složitosti znovupoužití pouze části aplikace. [4]

Naproti tomu služby z mikroslužeb lze využít i v dalších aplikacích jednoduše. Změny nebo vývoj nových částí monolitické aplikace jsou náročnější oproti aplikaci postavené na mikroslužbách. Službu v mikroslužbách lze upravit velice jednoduše z důvodu jednoduchosti celé služby. Práce s transakcemi je v monolitu běžnou praxí. V mikroslužbách je práce s transakcemi obtížnější a využívají se například ságy. Na obrázku pod kapitolou je demonstrován rozdíl mezi aplikací využívající architekturu monolit a mikroslužby. [4]



Obr. 16 – Porovnání mikroslužeb a monolitu [48]

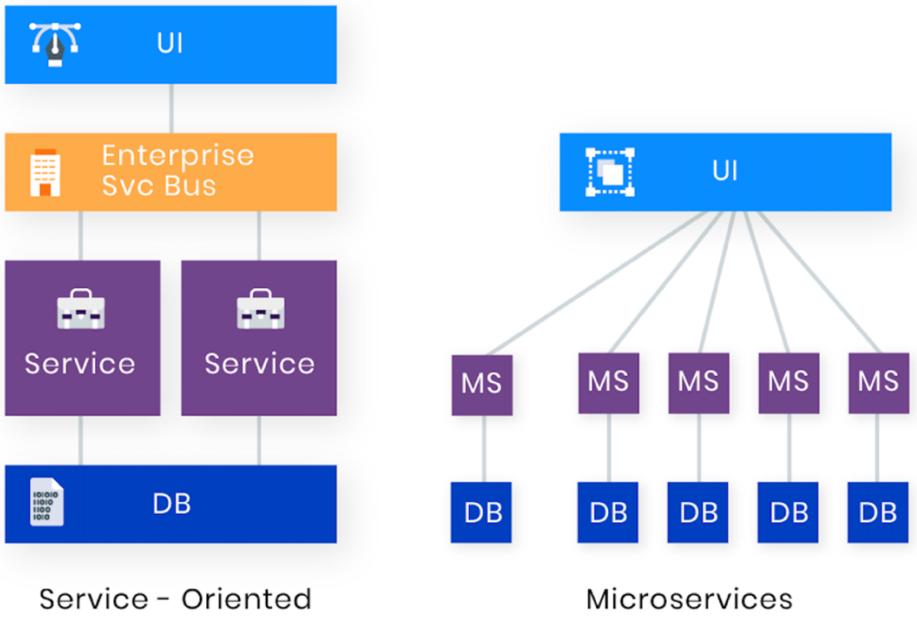
Hlavní rozdíly těchto dvou architektur jsou:

- znovupoužitelnost,
- škálovatelnost,
- správa,
- testování,
- logování,
- práce s transakcemi,
- počet použitých databází.

4.4.2 Mikroslužby a SOA

SOA je vhodné pro větší a složitější aplikace. SOA není vhodná pro malé aplikace. Mikroslužby je naopak vhodné použít pro menší a webové aplikace. Služby SOA aplikace jsou určené pro řadu použití. Mikroslužby se zabývají vždy pouze jedním úkolem. Škálovatelnost SOA je horší oproti mikroslužbám. Služby v SOA mají mezi sebou závislosti. Služby v mikroslužbách závislosti mezi sebou nemají. Z tohoto důvodu lze škálovat služby z mikroslužeb nezávisle na ostatních. [14]

Používání DevOps a CI/CD je běžnou praxí u mikroslužeb. V SOA lze využívat DevOps i CI/CD, ale zatím to není běžnou praxí. SOA sdílí databáze mezi službami. Služby v mikroslužbách mohou mít každá svou databázi. Velikost služeb se výrazně liší u mikroslužeb a SOA. V SOA mohou být služby od malých až po ty velké. U mikroslužeb jsou služby vždy malé celky. Nasazení mikroslužeb je rychlé a jednoduché oproti SOA. Při nasazení služby v SOA musí být nasazena celá aplikace z důvodů propojení služeb. Komunikace u mikroslužeb probíhá skrze API. V SOA je komunikace zajištěna skrze ESB. Mikroslužby pro vzdálený přístup využívají JMS a REST. Namísto toho SOA používá AMQP a SOAP. Mikroslužby mezi sebou sdílí co nejméně zdrojů. Díky propojení služeb v SOA jsou zdroje v této architektuře hojně sdíleny. Dále uvedený obrázek demonstruje rozdíl mezi těmito dvěma architekturami. [14]



Obr. 17 – Porovnání mikroslužeb a SOA [48]

Mezi odlišnosti těchto dvou architektur patří:

- granularita služeb,
- sdílení služeb,
- sdílení datových úložišť,
- sdílení zdrojů,
- vzdálený přístup k službám,
- škálovatelnost,
- vhodnost použití podle velikosti.

4.5 Výhody mikroslužeb

Výhody mikroslužeb jsou nezpochybnitelné a je to i důvodem jejich neustále rostoucí popularity mezi vývojáři. Z tohoto důvodu bude růst i počet technologií vhodných pro mikroslužby. Důležitou výhodou je rychlost vývoje. S tím souvisí i jednoduchá správa služby a možnost použití rozdílných frameworků a technologií vhodných pro každou službu.

Mezi výhody mikroslužeb patří:

- rychlý vývoj služeb,
- snadná správa služeb,
- jednoduché testování služeb,
- autonomní vývoj služeb,
- menší vývojářský tým pro vývoj,
- nezávislost jednotlivých služeb mezi sebou,
- možnost využití kontejnerizace,
- škálovatelnost služeb,
- použití vhodných technologií pro každou službu,
- využití moderních technologií,
- znovupoužitelnost,
- při potřebě lze znovu nasadit pouze konkrétní službu. [6][21][22]

4.6 Nevýhody mikroslužeb

Mikroslužby mají řadu výhod, avšak jsou spojeny i s celou řadu nevýhod. S rostoucím projektem roste i počet potenciálních problémů. V případě, že začne vznikat celá řada služeb, tak s velkým počtem služeb roste automaticky i komplexnost jejich správy. Vývoj začne být složitější s rostoucím počtem využitých frameworků a technologií.

Pokud vývojáři v průběhu vývoje přechází mezi týmy, tak může vzniknout problém s neznalostí technologií použitých v jiném týmu.

Za nevýhody mikroslužeb považujeme:

- složitá práce s transakcemi,
- náročnost správy roste s počtem služeb a využitých technologií,
- složitost testování díky distribuovanému rozvoji,
- komplikovanější přechod programátorů mezi týmy při použití rozdílných technologií pro vývoj,
- časová náročnost rozdělení na jednotlivé služby,
- vyšší nároky na síť,

- zvýšená šance útoků na externí API,
- zapracování komunikace mezi službami,
- komplexnost distribuovaného systému. [6][21][22]

5 Bezpečnostní hrozby mikroslužeb

5.1 Definice pojmů

Aktivum je jedním ze základních pojmů bezpečnosti a je definováno následující formulací. Jedná se o *„cokoliv, co má hodnotu pro jednotlivce, organizace nebo veřejnou správu.“* [49]

Hrozba dle kybernetického výkladového slovníku odpovídá následující definici: *„potencionální příčina nechtěného incidentu, jehož výsledkem může být poškození systému nebo organizace.“* [49]

Bezpečnostní hrozba z pohledu kybernetické bezpečnosti je pak formulována následovně: *„potencionální příčina nežádoucích událostí, která může mít za následek poškození systému a jeho aktiv, např. zničení, nežádoucí zpřístupnění (kompromitaci), modifikaci dat nebo nedostupnost služeb.“* [49]

5.2 Dělení hrozeb

Hrozby dělíme na několika kategorií. V následujícím textu budou uvedena rozdělení s následným popisem těchto kategorií a typů.

Dle zdroje působení:

- vnější,
- vnitřní. [50]

Vnější hrozby se nachází mimo organizaci a přichází zvenčí. Naopak vnitřní hrozby se nachází uvnitř organizace a přichází zevnitř. Pro předcházení vnějších i vnitřních hrozeb je nutné využívat bezpečnostní standardy. Bezpečnostní standardy lze vytvořit buď vlastní, nebo vhodnějším případem je využití již existující normy ISO/IEC 27001.

Dle úmyslu:

- úmyslné,
- náhodné. [50]

Úmyslné hrozby jsou cíleně plánované hrozby. Mezi úmyslné hrozby řadíme například úmyslné poškození informačního systému, sabotáže, napadení počítačovými viry. Náhodné hrozby jsou naopak způsobeny neúmyslně, náhodou. Jako příklady náhodných hrozeb lze uvést požáry, povodně, poruchu hardwaru, neúmyslné selhání lidského faktoru.

Dle působení na aktiva:

- uživatele,
- informace,
- operační systém,
- aplikace,
- hardware,
- síť. [50]

Dle původu:

- způsobené člověkem,
- přírodní. [50]

Hrozby způsobené člověkem jsou úmyslné i náhodné. Jako příklad takové hrozby lze uvést úmyslnou i neúmyslnou chybu uživatele, odposlech, krádež dat. Hrozby přírodního původu jsou hrozbami neméně důležitými. Za přírodní hrozby považujeme blesk, zemětřesení, povodeň, tornádo, tsunami, sesuvy půdy, sopečné erupce a další přírodní katastrofy.

Dle působení na bezpečnostní atributy:

- dostupnosti,
- integrity,
- důvěrnosti. [50]

Hrozba způsobující narušení dostupnosti představuje výpadek informačního systému. Mezi hrozby omezující dostupnost řadíme například přírodní katastrofy nebo DDoS útok. Narušení integrity může způsobit nekonzistenci dat v databázi

způsobené například chybou v databázové transakci. Narušení důvěrnosti způsobuje poškození nebo ohrožení aktiv jednotlivce, organizace nebo státní správy. Za příklady hrozeb narušení důvěrnosti uvádíme odcizení notebooku, odcizení telefonu, odcizení počítače nebo krádež dat z databáze.

Dle motivace útočníka:

- finanční zisk,
- získání konkurenční výhody,
- pomsta,
- testování svých dovedností. [50]

5.3 Zvolené bezpečnostní hrozby

V této podkapitole budou popsány bezpečnostní hrozby dále řešené v praktické části. Tato kapitola se bude zabývat popisem a zdůvodněním, proč jsou tyto konkrétní podkapitoly považovány za bezpečnostní hrozby pro aplikace využívající mikroslužby. Samotnému řešení nebo návrhu řešení jednotlivých bezpečnostních hrozeb uvedených níže bude věnována poslední kapitola.

5.3.1 Autentizace

Autentizace entity dle výkladového slovníku kybernetické bezpečnosti je formulována následující definicí: *„provedení testů, umožňujících systému zpracování dat rozpoznání a potvrzení entity.“* [49] Autentizace entity z pohledu informatiky je ověření, zda entita má či nemá přístup do systému. Autentizace může být jednofaktorová nebo vícefaktorová. Jednofaktorová autentizace se provádí například pomocí klíče, hesla, vstupní karty, osobního dokladu. Každý z těchto typů se označuje jako faktor ověřující identitu. Naopak vícefaktorová autentizace využívá kombinaci více faktorů dohromady. Běžně používanou vícefaktorovou autentizací představuje kombinace hesla a PIN kódu nebo hesla a otisku prstu. Vícefaktorová autentizace představuje bezpečnější řešení než jednofaktorová. V systémech bez autentizace uživatelů jsou v systému neověření uživatelé. V systémech pracujících s osobními či důvěrnými informacemi je autentizace nutností. Příkladem takového systému je například bankovnictví. Naopak ve volně dostupných systémech, pro

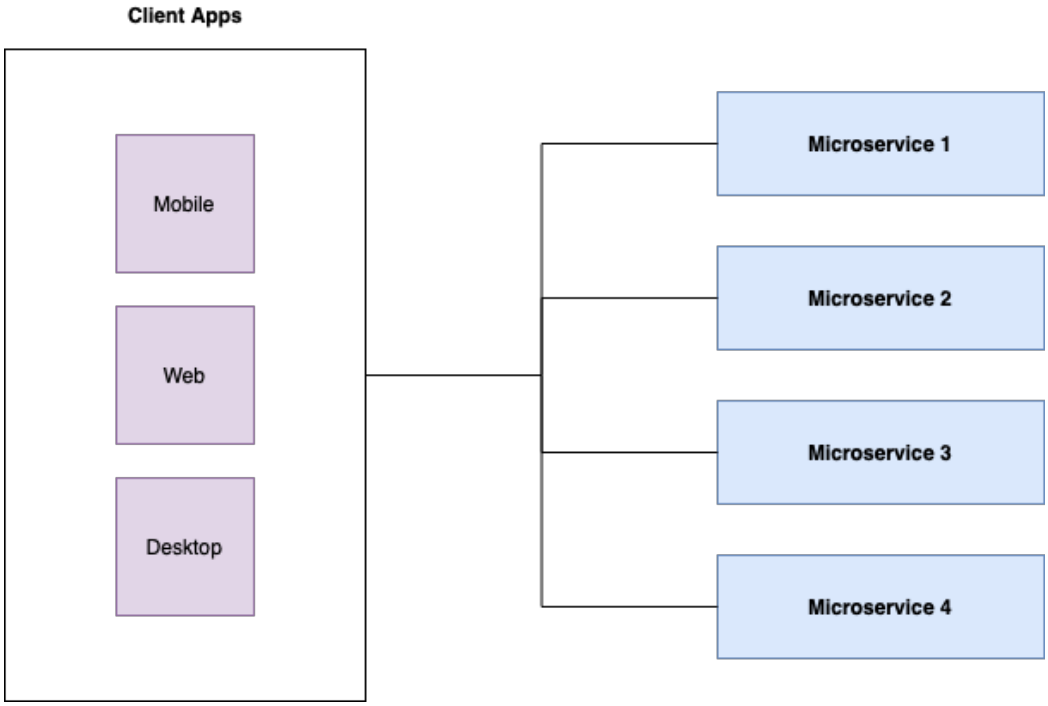
příklad lze uvést prohlížení zpravodajských webů, autentizace není nutností. Systém využívající autentizaci je vždy bezpečnější a je vhodným nástrojem pro zabezpečení.

5.3.2 Autorizace

Autorizaci definuje následující formulace. *„Udělení práv, které zahrnuje udělení přístupu na základě přístupových práv. Proces udělení práv subjektu pro vykonávání určených aktiv v informačním systému.“* [49] Autorizace z pohledu informatiky je procesem obvykle následujícím po autentizaci a procesem rozhodujícím o oprávněních uživatele uvnitř systému. Po autentizaci jsou uživateli přidělena oprávnění. Na základě těchto oprávnění jsou uživateli zpřístupněny operace a umožněn přístup do částí systému dle jeho oprávnění. Administrátor systému má nejvyšší oprávnění v celém systému. Administrátor může v systému například prohlížet, vytvářet, upravovat a mazat záznamy. Naopak ostatní uživatelé mají oprávnění omezená a mají oprávnění například pouze pro prohlížení záznamů v systému.

5.3.3 Komunikace s mikroslužbami

Komunikaci se službami mezi klientem a mikroslužbou dělíme na dvě kategorie. První kategorii představuje komunikace klienta s mikroslužbou na přímo. V tomto řešení má každá mikroslužba svůj veřejný koncový bod. Případně má mikroslužba veřejný koncový bod a zvolený rozdílný TCP port pro každou mikroslužbu. Komunikace na přímo je realizovaná skrze požadavek od klienta na veřejný koncový bod mikroslužby. S rostoucí aplikací vzniká řada problémů při využívání komunikace s mikroslužbami na přímo. Každá mikroslužba musí řešit vlastní implementaci autentizace a autorizace. Jedna část klientské aplikace může přinášet požadavky do více mikroslužeb najednou. Důsledkem toho výrazně klesá rychlost celé aplikace. Dále skrze veřejné přístupové body vzniká složitost při změně z důvodu přímého napojení klientské aplikace na mikroslužbu. Mikroslužby nejsou odstíněny a jsou veřejně přístupné. Z tohoto důvodu je mikroslužba snazším cílem pro útok. Na níže uvedeném obrázku je znázorněno schéma s využitím komunikace na přímo.

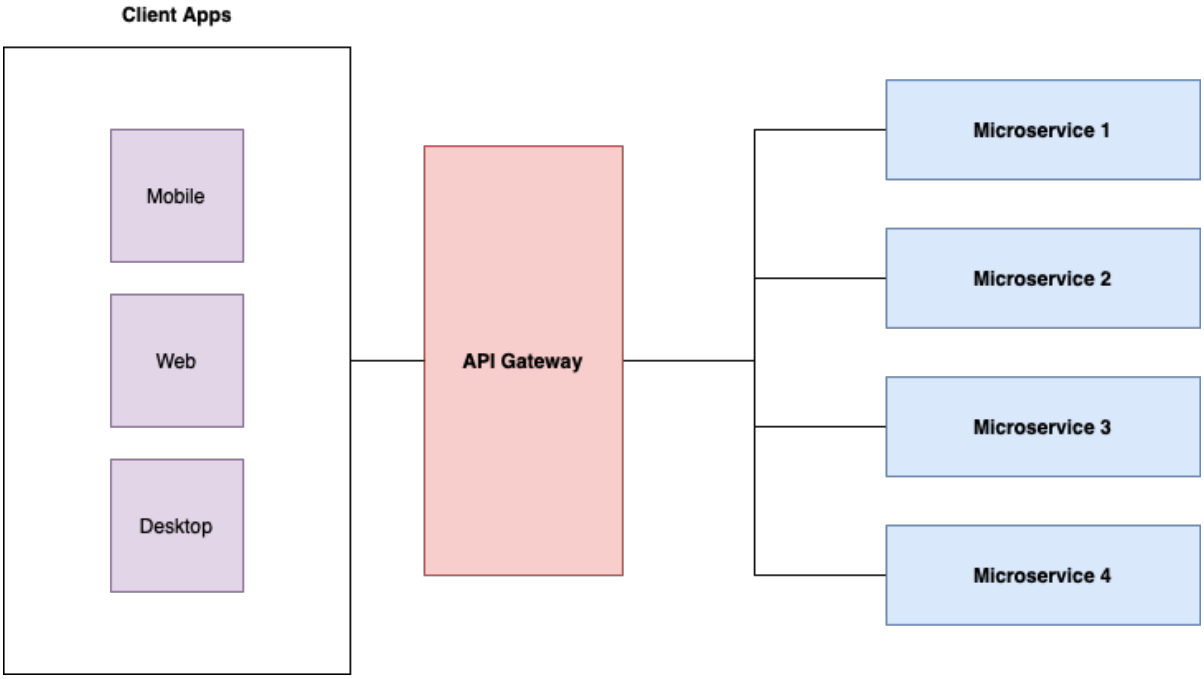


Obr. 18 – Komunikace na přímo [autor]

Druhou kategorií je komunikace s mikroslužbou skrz API Gateway. API Gateway představuje prostředníka mezi klientem a mikroslužbami. API Gateway má veřejný koncový bod pro komunikaci s klientem. Přístupové body mikroslužeb jsou v privátní síti a klient s nimi komunikovat z tohoto důvodu nemůže. S využitím API Gateway roste komplexita a je přidán další bod v komunikaci mezi klientem a mikroslužbou. Nicméně ve většině případů je zpomalení zanedbatelné. Pro zabezpečení komunikace skrze API Gateway se využívá ověření identity již na úrovni API Gateway, nikoliv až na konkrétní mikroslužbě. Neověření uživatelé nezískají přístup do mikroslužeb. Bezpečnostní hrozba při využití API Gateway vzniká na základě jednoho bodu pro komunikaci. Bez API Gateway není možné komunikovat s mikroslužbami. V praxi je tento problém řešen rozdělením API Gateway pro jednotlivé klientské aplikace - webové, desktopové a mobilní. Díky tomuto rozdělení se rozdělí zatížení, a i případná nedostupnost jedné API Gateway nemá vliv na všechny klientské aplikace. Na obrázku na další stránce je znázorněno schéma s využitím API Gateway.

API Gateway umožňuje:

- vyrovnání zatížení,
- směrování,
- monitorování,
- validaci vstupů,
- využití návrhového vzoru jistič,
- sběr metrik,
- bezpečnostní řešení.



Obr. 19 – API Gateway [autor]

5.3.4 Dostupnost

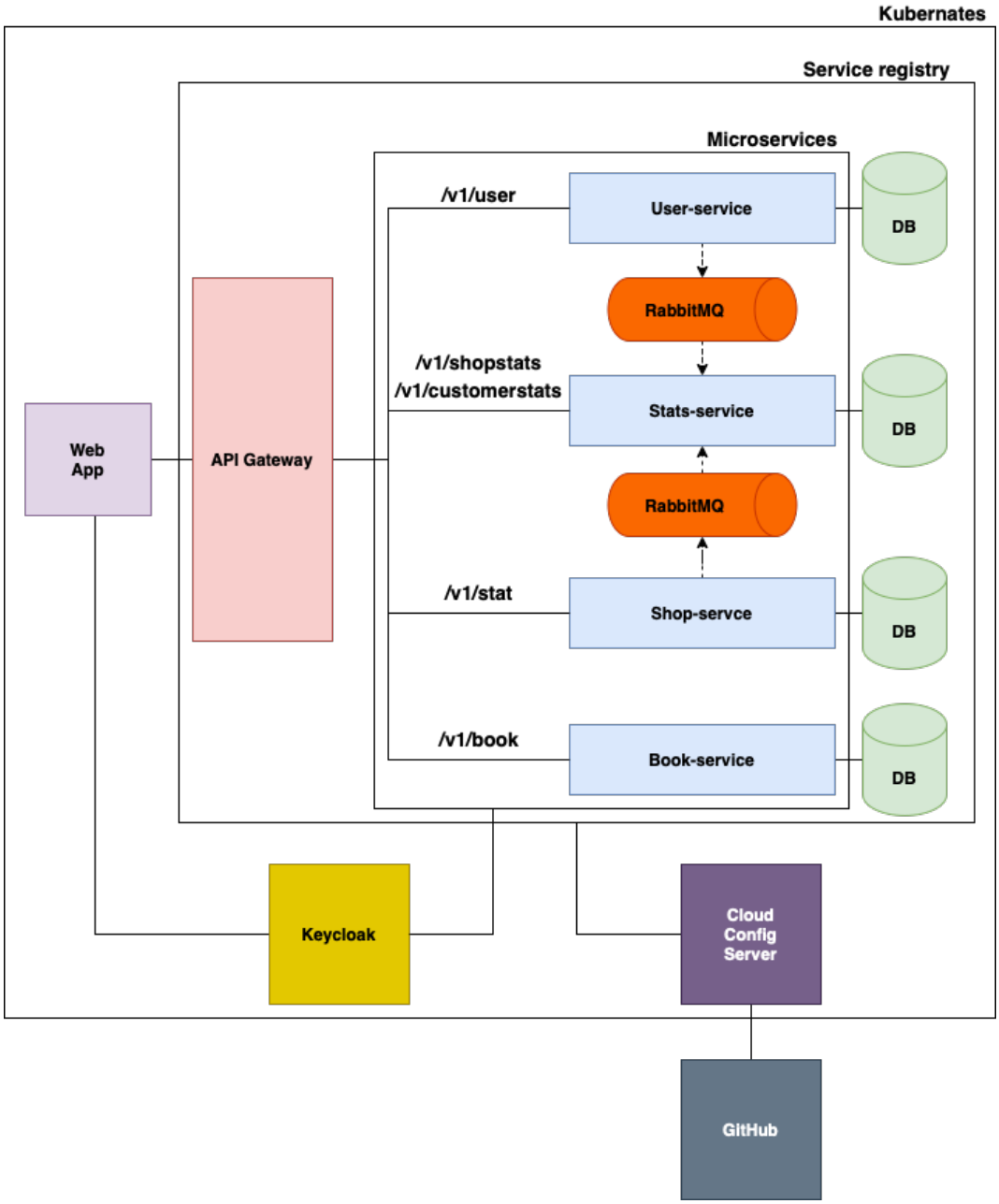
Dostupnost je definována následující formulací. Jedná se o *„vlastnost přístupnosti a použitelnosti na žádost oprávněné entity.“* [49] Dostupnost služby z pohledu informatiky představuje možnost poskytovat a možnost využívat tuto službu. V případě přetížení, například v důsledku útoku DDoS nebo jiného typu útoku, může být služba mimo provoz. V takovém případě je tato služba nedostupná.

5.3.5 Zabezpečení mikroslužby

Pro zabezpečení služby lze využít dva způsoby řešení problematiky. Buď si každá služba implementuje autentizaci a autorizaci individuálně, nebo využívá externí řešení pro jednotné přihlášení napříč službami. Na základě autentizace a autorizace jsou ve službách uživateli zpřístupněny části aplikace dle jeho oprávnění. Zabezpečení služby zahrnuje i zabezpečení její dostupnosti. Nezabezpečená služba je snazším potenciálním cílem útočníka.

6 Implementace

Pro řešení zvolených bezpečnostních hrozeb byla implementována aplikace využívající architekturu mikroslužeb. Aplikace se skládá ze tří hlavních částí. První částí je webová aplikace pro komunikaci s klientem. Druhou částí je API Gateway a poslední částí jsou čtyři mikroslužby s databázemi. Na obrázku níže je znázorněna architektura celé aplikace.



Obr. 20 – Architektura aplikace [autor]

6.1 Popis

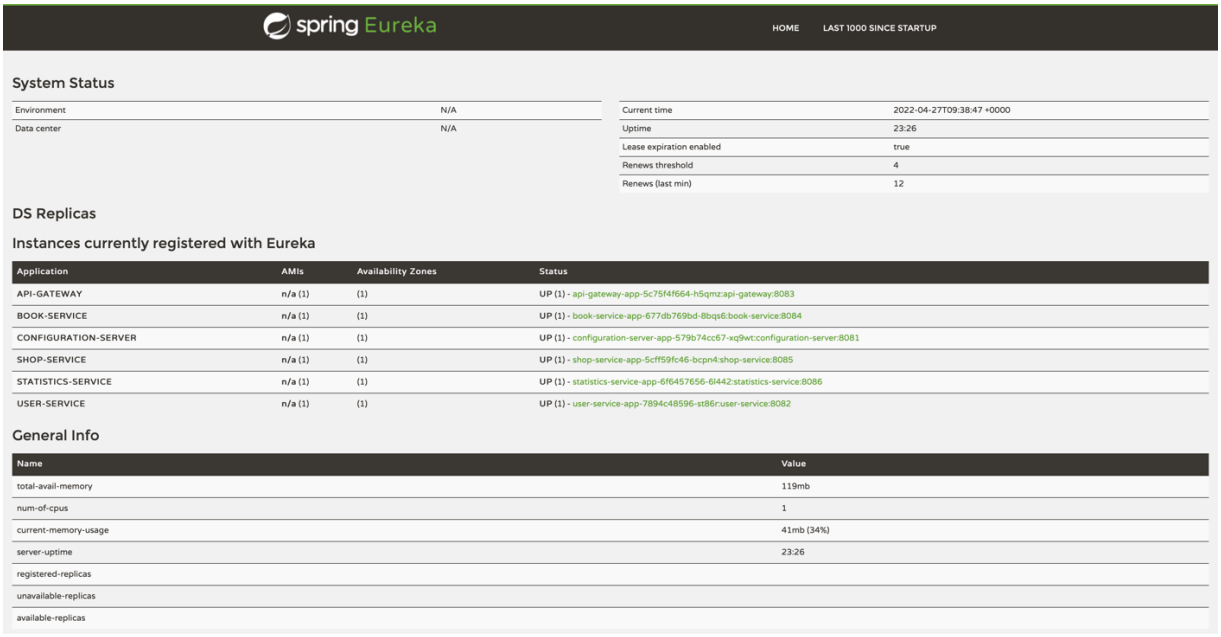
6.1.1 Nástroje a technologie

Aplikace je postavená na technologiích popsaných ve třetí kapitole. Kromě těchto technologií a nástrojů aplikace využívá minikube, Keycloak, Spring Cloud Netflix, Resilience4j. Minikube představuje odlehčenou implementaci Kubernetes. Minikube vytvoří virtuální stroj na lokálním počítači a nasadí jednoduchý cluster obsahující pouze jeden uzel. Minikube je kompatibilní s Windows, Linux i macOS. Keycloak vznikl v roce 2014 jako řešení pro správu identit a přístupů. Jedná se o open source produkt vyvíjen společností WildFly.

Funkce a vlastnosti Keycloaku:

- SSO a Single-Sign Out pro prohlížeče,
- podpora vícefaktorové autentizace,
- přihlášení skrze sociální sítě,
- podpora LDAP,
- podpora OAuth 2.0,
- podpora CORS,
- zprostředkování identity,
- přidělování rolí,
- podpora OpenID Connect,
- podpora celého přihlašovacího procesu (registrace, obnovení hesla a ověření emailu). [51]

Spring Cloud Netflix Eureka představuje service registry a umožňuje klientům komunikovat mezi sebou bez přímého zadání adresy hosta a portu. Jediný statický bod představuje service registry. Každá mikroslužba se registruje právě do service registry. Klient může fungovat zároveň jako server. Klienti posílají signál označovaný jako heartbeat. V případě neodeslání signálu Eureka server vyřadí klienta ze service registry. Mezi klienty v rámci aplikace řadíme i API-Gateway a configuration-server. Na obrázku níže je znázorněný Eureka dashboard s instancemi klientů včetně jejich jména, portu a statusu.



Obr. 21 – Eureka dashboard [autor]

Resilience4j je knihovna výhodná pro svou odolnost proti chybám a s podporou funkcionálního programování. Resilince4j představuje alternativu k Netflix Hystrixu. Resilince4j je inspirován právě Netflix Hystrixem. Netflix Hystrix je komplexnější nástroj s řadou závislostí na dalších zdrojích. Resilince4j poskytuje dekorátory rozšiřující funkční rozhraní, lambda výraz nebo odkaz na metodu. Dekorátory lze kombinovat a používat jich více zároveň. Nicméně stačí využít pouze potřebné dekorátory. [52]

Resilince4j nabízí řešení:

- omezování rychlosti,
- opakování,
- přepážky,
- jističe. [52]

6.1.2 Popis aplikace

Celá aplikace byla vyvíjena, spouštěna a testována v minikube. Image služeb jsou stahovány z Docker Hubu. Jednotlivé konfigurace jsou popsány

v konfiguračních souborech s příponou yml. Konfigurace aplikace se skládá ze14 konfiguračních souborů.

Pro fungování aplikace v minikube je spuštěna:

- user-service,
- mysql-user,
- book-service,
- mysql-book,
- statistics-service,
- mysql-stats,
- shop-service,
- mysql-shop,
- api-gateway,
- rabbitmq,
- keycloak,
- configuration-server,
- frontend,
- service-registry.

Pro jednotlivé konfigurace jsou nastaveny ConfigMapy, deploymenty, pody, služby, kódované údaje (secrets), ReplicaSety, PersistantVolumeClaim pro databáze a StatefulSet pro Eureku. V konfiguračním souboru pro Kubernetes se nastavuje apiVersion, typ, název, porty, počet replik, cesta k image v Docker Hubu a další podrobnější nastavení. Nastavení se liší podle typu, pro který se konfigurace nastavuje. Konfigurace dále obsahují například proměnné pro propojení mikroslužby s databází. Na prvním obrázku na následující straně je zobrazeno nastavení proměnných v konfiguračním souboru pro Kubernetes.

Obr. 22 – Nastavení proměnných v konfiguraci pro Kubernets [autor]

Na dalším obrázku je znázorněno použití těchto proměnných v mikroslužbě. Právě díky nastavení těchto proměnných se databáze propojí s mikroslužbou.

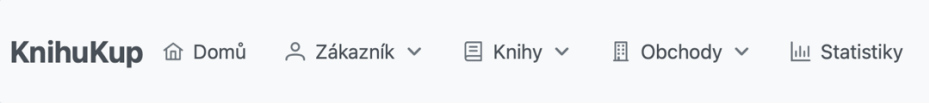
```
spring:
  application:
    name: user-service
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://${DB_HOST}/${DB_NAME}?useSSL=false
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
```

Obr. 23 – Použití proměnných v mikroslužbě [autor]

6.1.3 Popis webové aplikace

Pro komunikaci aplikace s klientem je vyvinuta webová aplikace v Angularu. Tato aplikace se jmenuje KnihuKup. Aplikace slouží pro správu knih, obchodů a zákazníků v rámci KnihuKup. Navigace v aplikaci je řízena navigačním menu obsahujícím pět záložek. Záložka „Domů“ je úvodní stránkou webové aplikace. Další záložka „Zákazník“ obsahuje zákazníky s jejich detaily a správou. Třetí záložku představuje položka „Knihy“ nabízející správu knih pro KnihuKup. Následuje záložka „Obchody“ nabízející detaily obchodů a jejich správu. Poslední záložka „Statistiky“ zobrazuje oblíbenost knih u zákazníků a obchody nabízející konkrétní knihu. Po kliknutí na knihu v druhém přehledu se zobrazí obchody, kde jsou

jednotlivé knihy prodávány. Na obrázku pod odstavcem je znázorněno navigační menu se záložkami.

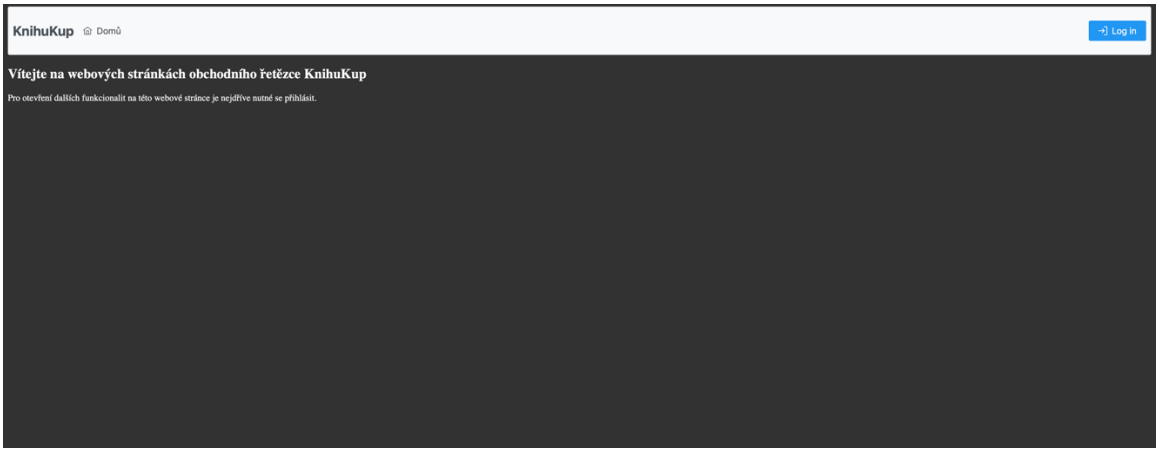


Obr. 24 – Navigační menu [autor]

Pro vstup do aplikace je nutná autentizace. O přihlášení se stará Keycloak. Aplikace využívá notifikace pro informování uživatele. V případě chybného požadavku, jako je například neoprávněný přístup, je uživateli zobrazena notifikace. Naopak jsou zobrazovány i informativní notifikace v rámci úspěšného uložení. Webová aplikace je určena pouze pro prohlížení. Obsah aplikace je obsluhován skrze API Gateway. Všechny HTTP požadavky z aplikace kromě přihlášení vedou do API Gateway. Následně API Gateway zajišťuje směrování požadavků do konkrétní mikroslužby. Bez přihlášení lze navštívit pouze úvodní stránku. Po úspěšném přihlášení jsou uživateli umožněny další akce v aplikaci dle jeho oprávnění. Oprávnění v aplikaci se dělí na dva typy. První oprávnění je pro přihlášené uživatele a druhé pro administrátora.

Přihlášenému uživateli je umožněno spravovat si svůj profil, prohlížet knihy a obchody. V rámci profilu si přihlášený uživatel upravuje své údaje a přidává nebo odebírá své oblíbené knihy nabízené v aplikaci. Přihlášenému uživateli je zpřístupněn obsah s knihami a obchody. V rámci knih i obchodů si přihlášený uživatel může prohlížet i detaily knih a obchodů. Nicméně přihlášený uživatel nemá oprávnění pro vytváření, upravování knih a obchodů ani v jejich detailech. Dále má přihlášený uživatel přístup ke statistikám aplikace.

Administrátorovi je v rámci webové aplikace umožněn přístup do všech částí aplikace a k veškeré správě. V rámci správy knih lze knihu přidat nebo upravit. U zákazníků lze přidat a odebrat knihu. Dále může přidávat, mazat a upravovat zákazníky. Stejné operace lze provádět i v rámci obchodů. Na obrázku na další straně je obrázek z webové aplikace.



Obr. 25 – Ukázka z webové aplikace [autor]

6.1.4 Popis API Gateway

API Gateway je implementováno v aplikaci a jmenuje se api-gateway. Api-gateway je jediným zveřejněným bodem uvnitř service registry. Skrze tento bod prochází veškerá komunikace s mikroslužbami. Api-gateway zajišťuje směrování požadavků do konkrétních služeb a zasílání chyb v případě nedostupnosti mikroslužby. Pro kontrolu dostupnosti mikroslužeb je využit Resilience4j. Právě Resilience4j a návrhový vzor jistič v případě nedostupnosti vracejí upravenou definovanou hlášku jako odpověď na požadavek. Lze využít jednu obecnou hlášku nebo specifickou pro konkrétní mikroslužby. V aplikaci jsou implementovány specifické chybové hlášky v případě nedostupnosti některé z mikroslužeb. V api-gateway musí být uvedena všechna směrování do mikroslužeb. Pokud není směrování do mikroslužby v api-gateway uvedené, tak odpověď na HTTP požadavek je se stavovým kódem HTTP 404 Not Found. Pokud má být mikroslužba přístupná zvenčí, musí být vždy uvedena mezi směrováními v api-gateway. Na obrázku na další straně je zobrazena implementace směrování s filtry v případě nedostupnosti.

Obr. 26 – Směrování s filtry [autor]

6.1.5 Popis mikroslužeb a dalších aplikací

Aplikace obsahuje celkem čtyři mikroslužby. Každá z těchto mikroslužeb má svou MySQL databázi. První mikroslužba se jmenuje user-service. Tato mikroslužba má na starost práci se zákazníky. Zákazníci jsou uloženi v databázi náležící k mikroslužbě. Mikroslužba s názvem shop-service zajišťuje práci s jednotlivými obchody. Obchody jsou ukládány do databáze pro tuto mikroslužbu. Mikroslužba book-service obstarává agendu týkající se knih a jejich ukládání do databáze. Poslední mikroslužba statistics-service má na starost zpracování statistik v aplikaci. Statistics-service využívá pro zpracování zpráv RabbitMQ. Tato služba je konzumentem služeb shop-service a user-service. Shop-service a user-service jsou naopak v roli producenta. Pokud proběhne nějaká změna u záznamu v jedné z těchto dvou služeb, je poslána zpráva. Tuto zprávu přijme statistics-service a přepočítá statistiky a ty následně uloží do databáze. Každá z těchto mikroslužeb má svůj přiřazený port a nastavené připojení do service registry. Všechny mikroslužby přijímají HTTP požadavky dle svého API. Každý HTTP požadavek je následně validován Keycloakem. Pokud proběhne validace v pořádku a má patřičná oprávnění, je požadavek zpracován. V opačném případě vznikne chyba se stavovým kódem HTTP 401 Unauthorized. Mikroslužby nemají veřejnou IP adresu

a komunikace zvenčí přímo s nimi je tedy nemožná. Pro komunikaci s mikroslužbami je nutné využít api-gateway.

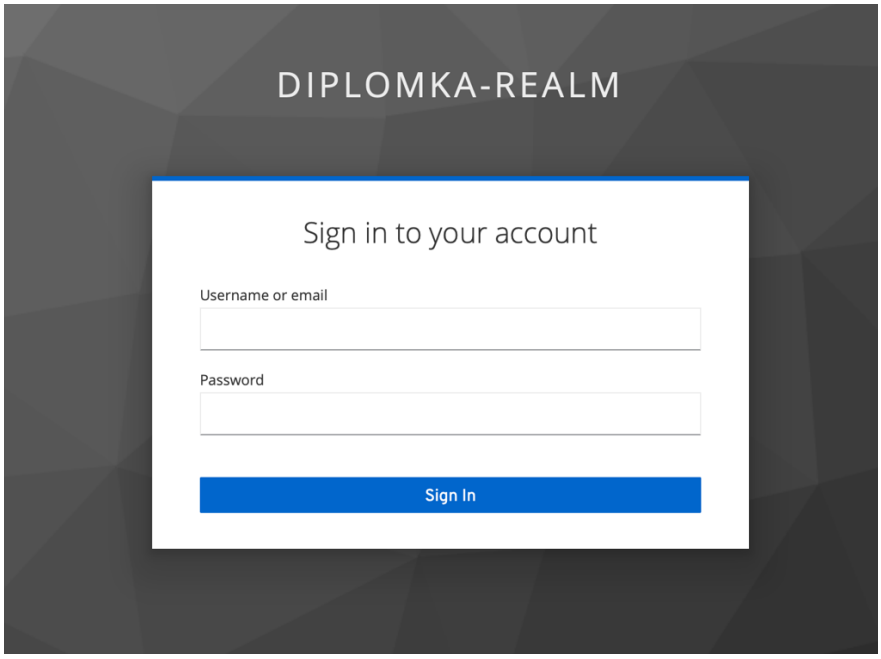
Mezi další nezmíněné aplikace patří service-registry a configuration-server. Service-registry je jednoduchá aplikace představující Spring Cloud Netflix Eureka server. Do této aplikace se všechny ostatní mikroslužby a api-gateway připojují. Configuration-server je Spring Cloud Config Server pro všechny mikroslužby v aplikaci i api-gateway. Tato aplikace stahuje konfiguraci pro připojení do service registry z GitHubu a následně ji poskytuje všem připojeným mikroslužbám i api-gateway. Mikroslužby i api-gateway mají konfigurační soubor bootstrap.yml, ve kterém mají nastavené připojení právě ke configuration-serveru. Díky tomuto propojení využijí připojené aplikace konfiguraci configuration-serveru a nemusí ji implementovat samostatně.

6.2 Ukázky řešení vybraných bezpečnostních hrozeb

Jak již bylo uvedeno v páté kapitole, budou v této kapitole představena řešení jednotlivých bezpečnostních hrozeb. V rámci aplikace byla nalezena řešení pro zvolené bezpečnostní hrozby. Nicméně tato implementace je pouze jedním z množiny možných řešení. Řešení se můžou lišit převážně z důvodů použití jiných nástrojů, technologií nebo programovacích jazyků.

6.2.1 Řešení autentizace

Autentizace je zajišťována v aplikaci Keycloakem. Pro autentizaci ve webové službě je uživatel přesměrován na přihlášení se do Keycloaku. V Keycloaku je využito OpenID Connect pro ověření identity. Po úspěšném přihlášení je uživateli vygenerován JWT s použitím algoritmu RS256. Tento algoritmus se řadí mezi asymetrické algoritmy. Uživatel má po úspěšné autentizaci prověřenou identitu. Ověření jeho identity je vyžadováno při jeho komunikaci s mikroslužbami. Po obdržení HTTP požadavku mikroslužba validuje JWT s Keycloakem. V případě neověření identity uživatele je jeho komunikace s mikroslužbou zamítnuta. Na obrázku pod odstavcem je zobrazena přihlašovací stránka Keycloaku.



Obr. 27 – Přihlašovací stránka Keycloaku [autor]

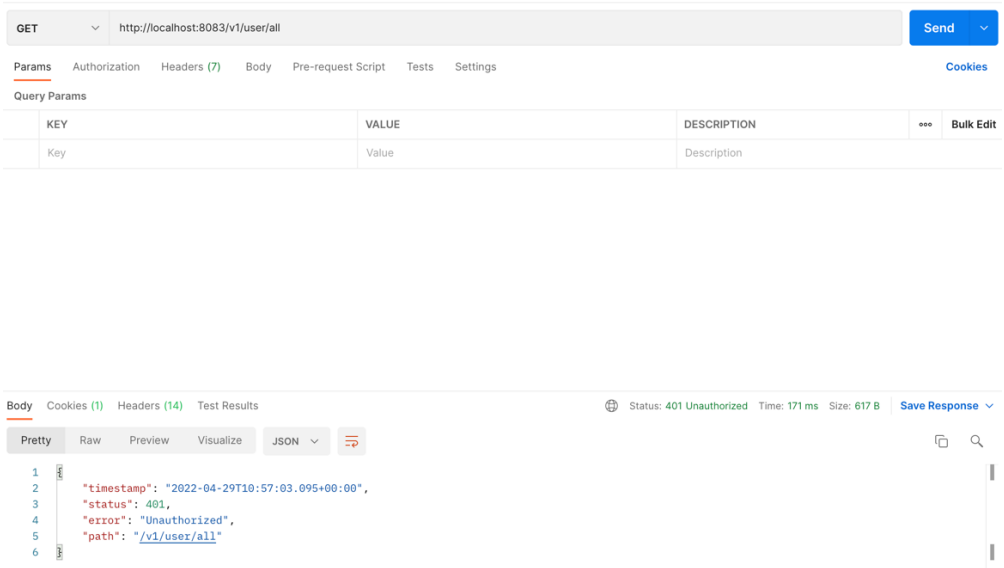
6.2.2 Řešení autorizace

Vhodné řešení autorizace je spojeno s autentizací uživatele přes Keycloak. Uživateli je v Keycloaku přidělena jedna či více rolí. Každá z těchto rolí dále určuje jeho oprávnění pro přístup do částí aplikace nebo možnost zasílat HTTP požadavky do mikroslužeb. V mikroslužbě jsou jednotlivé koncové body zabezpečené dle role. JWT je v mikroslužbě dekódován a mikroslužba umožňuje uživateli interakce dle jeho role v Keycloaku. Při zaslání požadavku, kterému neodpovídá oprávnění uživatele, je zaslán stavový kód HTTP 401 Unauthorized. Validace JWT tokenu je implementována v mikroslužbě user-service. Nicméně pro použití aplikace v produkčním prostředí je nutné provést obdobnou implementaci a zabezpečit i ostatní mikroslužby. Na obrázku níže je zobrazena ukázka implementace zabezpečení metody dle oprávnění uživatele.

```
@GetMapping("/all")
@RolesAllowed({"role_admin", "role-user"})
public List<User> findAllUsers() { return userService.findAllUser(); }
```

Obr. 28 – Ukázka zabezpečení metody podle role uživatele [autor]

Na obrázku na další straně je znázorněna ukázka HTTP GET požadavku bez tokenu. V takovém případě je vrácen stavový kód HTTP 401 Unauthorized.



Obr. 29 – Ukázka HTTP požadavku GET bez tokenu [autor]

6.2.3 Řešení komunikace s mikroslužbami

Komunikace s jednotlivými mikroslužbami v aplikaci probíhá skrze api-gateway. Komunikace na přímo v aplikaci není možná. Jediným přístupným bodem zvenčí je právě api-gateway. Api-gateway zajišťuje řízení komunikace pro všechny mikroslužby. Aplikace využívá pro komunikaci s klientem pouze webovou aplikaci, a proto je implementována pouze jedna API Gateway. V případě více klientských aplikací je vhodné implementovat API Gateway pro každou z nich. Implementace pro každou klientskou aplikaci přináší rozložení zatížení v jednom místě.

6.2.4 Řešení dostupnosti

Kubernetes v případě výpadku podu s mikroslužbou automaticky znovu nasadí pod. Pod a aplikace v něm je po tuto dobu nedostupná. Nedostupnost v tuto dobu obsluhuje návrhový vzor jistič s Resilience4j. Ten v případě nedostupnosti upozorní na výpadek při komunikaci s mikroslužbou v době nedostupnosti. Dále je možné nastavit vyšší počet replik. V takovém případě je zátěž rozložena mezi více podů.

6.2.5 Řešení zabezpečení mikroslužby

Výše zmíněná řešení bezpečnostních hrozeb souvisí se zabezpečením mikroslužby. Všechna tato řešení přispívají k zabezpečení mikroslužby samotné.

Díky správné implementaci autentizace a autorizace se komunikuje pouze s ověřenými uživateli s patřičnými oprávněními.

Aplikace využívá API Gateway, což přináší koordinovanou komunikaci. Mikroslužby v důsledku použití API Gateway nejsou přímo přístupné skrze veřejnou IP adresu. Skrytí mikroslužby za API Gateway je vhodným bezpečnostním řešením. Dále lze implementovat omezení počtu požadavků na mikroslužbu. Pro nastavení omezení počtu požadavků se dá například využít Redis. Následně je na API Gateway nutno nastavit tři parametry pro každé API.

Parametry pro nastavení omezení:

- replenishRate,
- burstCapacity,
- requestedTokens.

ReplenishRate nastavuje počet požadavku pro klienta v rámci jedné minuty. Naproti tomu burstCapacity reprezentuje hodnotu možných požadavků klienta za sekundu. Třetím parametr requestedTokens představuje cenu jednoho dotazu. RequestedTokens je obvykle nastaven na hodnotu jedna. Při překročení počtu HTTP požadavků budou nadbytečné HTTP požadavky zahozeny a klient dostane odpověď s chybou se stavovým kódem HTTP 429 Too Many Requests.

7 Shrnutí výsledků

V diplomové práci se v rámci teoretické části podařilo prozkoumat architektury monolitu a Service Oriented Architecture. Po jejich prozkoumání byly architektury detailně popsány včetně svých výhod i nevýhod. Byl proveden podrobný popis architektury mikroslužeb a jeho jednotlivých charakteristik. Nezávislost, malá velikost nebo autonomní vývoj jsou velice relevantní důvody, proč využívat tuto architekturu. Porovnání monolitu, mikroslužeb a Service Oriented Architecture přineslo zajímavé závěry. Například rozdílnost mezi mikroslužbami a Service Oriented Architecture je výrazná, ačkoliv mikroslužby jsou někdy označovány jako typ Service Oriented Architecture. Byly nalezeny vhodné technologie pro vývoj mikroslužeb. V kaptitole pět teoretické části byly vyjmenovány a popsány bezpečnostní hrozby pro aplikace využívající mikroslužby.

V rámci praktické části pak byla vyvinuta aplikace dle požadavků využívající architekturu mikroslužeb. V rámci aplikace byla úspěšně implementována řešení nalezených bezpečnostních hrozeb. Jednotlivá řešení bezpečnostních hrozeb byla popsána v rámci kapitoly šest diplomové práce.

8 Závěry a doporučení

Softwarové architektury mají v informatice nezpochybnitelně své místo. Pochopení a správná implementace těchto architektur přináší řadu benefitů. Žádnou z architektur nelze považovat za nejlepší. Každá architektura je vhodná pro specifické použití a nelze ji efektivně využívat všude.

Mikroslužby jsou velice zajímavou architekturou. Stejně jako je tomu u ostatních architektur, i mikroslužby s sebou přináší řadu výhod i nevýhod. Decentralizace, malá velikost, nezávislost či autonomní vývoj jsou významnými vlastnostmi této architektury. Naopak potřebná znalost nástrojů a technologií pro použití této architektury je poměrně vysoká. V případě vývoje webové aplikace s důrazem na znovupoužitelnost, snadné nasazení a dobrou škálovatelnost je použití mikroslužeb vhodným řešením. Díky nástrojům a technologiím jako je Docker, Kubernetes, AWS, Azures získávají mikroslužby řadu výhod v porovnání s ostatními architekturami.

V rámci diplomové práce byla vyvinuta aplikace využívající architekturu mikroslužeb. Byly nalezeny bezpečnostní hrozby pro aplikace využívající mikroslužby. Tyto bezpečnostní hrozby byly popsány v teoretické části a v praktické části byla implementována vhodná řešení těchto bezpečnostních hrozeb.

Nalezené bezpečnostní hrozby však nejsou všechny možné. Implementace pro zvolené bezpečnostní hrozby se odlišují dle vybraných nástrojů a technologií. Popularita mikroslužeb v budoucnu bude nejspíše i nadále růst. Z těchto důvodů lze v budoucnu tuto práci dále rozvíjet.

9 Seznam použité literatury

[1] MEDVIDOVIC, Nenad a Richard N. TAYLOR. Software architecture: foundations, theory, and practice. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2* [online]. New York, NY, USA: Association for Computing Machinery, 2010, s. 471–472 [vid. 2021-03-15]. ICSE '10. ISBN 978-1-60558-719-6. Dostupné z: doi:10.1145/1810295.1810435

[2] RICHARDS, Mark. *Software architecture patterns: understanding common architecture patterns and when to use them*. 2015. ISBN 978-1-4919-2424-2.

[3] TIŠNOVSKÝ, Pavel. *Mikroslužby: moderní aplikace využívající známých konceptů - Root.cz* [online]. 16. květen 2019 [vid. 2021-03-15]. Dostupné z: https://www.root.cz/clanky/mikrosluzby-moderni-aplikace-vyuzivajici-znamych-konceptu/#k02

[4] *Microservices vs Monolith: which architecture is the best choice for your business?* [online]. [vid. 2022-02-02]. Dostupné z: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/

[5] NEWMAN, Sam. *Building microservices: designing fine-grained systems*. First Edition. Beijing Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1-4919-5035-7.

[6] NADAREISHVILI, Irakli. *Microservice architecture: aligning principles, practices, and culture*. First edition. Sebastopol, CA: O'Reilly Media, Inc, 2016. ISBN 978-1-4919-5625-0.

[7] HAQ, Siraj ul. Introduction to Monolithic Architecture and MicroServices Architecture. *Medium* [online]. 11. červenec 2018 [vid. 2021-03-15]. Dostupné z: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63

[8] HORDĚJČUK, Vojtěch. *Mikroslužby - Vojtěch Hordějčuk* [online]. [vid. 2021-03-20]. Dostupné z: http://voho.eu/wiki/mikrosluzba/

[9] GARG, Sarthak. *Service-Oriented Architecture - GeeksforGeeks* [online]. [vid. 2021-09-17]. Dostupné z: https://www.geeksforgeeks.org/service-oriented-architecture/

[10] COMMUNITY, Software Development. What Is Service-Oriented Architecture? *Medium* [online]. 13. únor 2019 [vid. 2021-09-17]. Dostupné z: https://medium.com/@SoftwareDevelopmentCommunity/what-is-service-oriented-architecture-fa894d11a7ec

[11] ERL, Thomas, Paulo MERSON a Roger STOFFERS. *Service-oriented architecture: analysis and design for services and microservices*. Second edition. Boston:

Prentice Hall, 2017. The Prentice Hall service technology series from Thomas Erl. ISBN 978-0-13-385858-7.

[12] *SOA Reference Architecture – Overview of the SOA RA Layers* [online]. [vid. 2022-03-21]. Dostupné z: https://www.opengroup.org/soa/source-book/soa_refarch/p5.htm

[13] Service Oriented Architecture: A Dead Simple Explanation - DZone Microservices. *dzone.com* [online]. [vid. 2022-03-21]. Dostupné z: https://dzone.com/articles/service-oriented-architecture-a-dead-simple-explan

[14] *Microservices vs SOA | What's the Difference | Edureka* [online]. [vid. 2021-09-26]. Dostupné z: https://www.edureka.co/blog/microservices-vs-soa/

[15] Fig. 6. Interaction among service provider, service consumer and... *ResearchGate* [online]. [vid. 2022-03-28]. Dostupné z: https://www.researchgate.net/figure/Interaction-among-service-provider-service-consumer-and-service-registry_fig4_338363841

[16] AINS Professional IT Services | Enterprise Service Bus. *AINS | Low-Code Platform & Case Management Solutions* [online]. [vid. 2021-09-26]. Dostupné z: https://www.ains.com/enterprise-service-bus-esb/

[17] *What is an ESB (Enterprise Service Bus)? | IBM* [online]. [vid. 2021-09-26]. Dostupné z: https://www.ibm.com/cloud/learn/esb

[18] FOOTE, Keith D. A Brief History of Microservices. *DATAVERSITY* [online]. 22. duben 2021 [vid. 2021-10-16]. Dostupné z: https://www.dataversity.net/a-brief-history-of-microservices/

[19] A Quick Primer on Microservices - DZone Integration. *dzone.com* [online]. [vid. 2021-10-16]. Dostupné z: https://dzone.com/articles/a-quick-primer-on-microservices

[20] *microservices* [online]. 23. červen 2021 [vid. 2021-10-16]. Dostupné z: https://www.ibm.com/cloud/learn/microservices

[21] *What are Microservices? | API Basics | SmartBear* [online]. [vid. 2021-10-16]. Dostupné z: https://smartbear.com/solutions/microservices/

[22] Microservices. *martinfowler.com* [online]. [vid. 2021-10-16]. Dostupné z: https://martinfowler.com/articles/microservices.html

[23] BALAJEE, Nanduri. What is CI/CD Pipeline? *Medium* [online]. 7. leden 2020 [vid. 2021-10-24]. Dostupné z: https://nanduribalajee.medium.com/what-is-ci-cd-pipeline-e2f25db99bbe

[24] NADAREISHVILI, Irakli. *Microservice architecture: aligning principles, practices, and culture*. First edition. Sebastopol, CA: O'Reilly Media, Inc, 2016. ISBN 978-1-4919-5625-0.

[25] *MySQL :: MySQL Document Store* [online]. [vid. 2022-01-14]. Dostupné z: https://www.mysql.com/products/enterprise/document_store.html

[26] *MySQL - Introduction* [online]. [vid. 2022-01-14]. Dostupné z: https://www.tutorialspoint.com/mysql/mysql-introduction.htm

[27] Your relational data. Objectively. - Hibernate ORM. *Hibernate* [online]. [vid. 2022-01-14]. Dostupné z: https://hibernate.org/orm/

[28] *Learn Hibernate Tutorial - javatpoint* [online]. [vid. 2022-01-14]. Dostupné z: https://www.javatpoint.com/hibernate-tutorial

[29] 11 Most In-Demand Programming Languages in 2021. *Berkeley Boot Camps* [online]. 16. prosinec 2020 [vid. 2021-11-13]. Dostupné z: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

[30] *Java Bytecode - Javatpoint* [online]. [vid. 2022-01-09]. Dostupné z: https://www.javatpoint.com/java-bytecode

[31] What is Java? A Beginner's Guide to Java and its Evolution. *Edureka* [online]. 19. duben 2017 [vid. 2022-01-14]. Dostupné z: https://www.edureka.co/blog/what-is-java/

[32] *What is Gradle?* [online]. [vid. 2022-02-02]. Dostupné z: https://docs.gradle.org/current/userguide/what_is_gradle.html

[33] *Gradle | Gradle vs Maven Comparison* [online]. [vid. 2022-02-02]. Dostupné z: https://gradle.org/maven-vs-gradle/

[34] *History of Spring Framework and Spring Boot* [online]. [vid. 2022-01-14]. Dostupné z: https://www.quickprogrammingtips.com/spring-boot/history-of-spring-framework-and-spring-boot.html

[35] *Why Spring?* [online]. [vid. 2022-01-14]. Dostupné z: https://spring.io/why-spring

[36] *Spring Tutorial | Getting Started With Spring Framework | Edureka* [online]. [vid. 2022-01-14]. Dostupné z: https://www.edureka.co/blog/spring-tutorial/comment-page-2/#comments

[37] *Documentation - The Basics* [online]. [vid. 2022-03-27]. Dostupné z: https://www.typescriptlang.org/docs/handbook/2/basic-types.html

[38] *Angular - What is Angular?* [online]. [vid. 2022-01-22]. Dostupné z: https://angular.io/guide/what-is-angular

[39] *REST Resource Naming Guide* [online]. [vid. 2022-01-22]. Dostupné z: https://restfulapi.net/resource-naming/

[40] REST API vs. GraphQL [comparison]. *DEV Community* [online]. [vid. 2022-01-30]. Dostupné z: https://dev.to/duomly/rest-api-vs-graphql-comparison-3j6g

[41] *Messaging that just works — RabbitMQ* [online]. [vid. 2022-01-29]. Dostupné z: https://www.rabbitmq.com/#getstarted

[42] *exchanges-topic-fanout-direct.png (PNG obrázek, 1206 × 736 bodů)* [online]. [vid. 2022-01-23]. Dostupné z: https://www.cloudamqp.com/img/blog/exchanges-topic-fanout-direct.png

[43] *Docker overview | Docker Documentation* [online]. [vid. 2022-01-29]. Dostupné z: https://docs.docker.com/get-started/overview/

[44] *What Is Docker? A Beginner's Guide | JFrog* [online]. [vid. 2022-01-29]. Dostupné z: https://jfrog.com/knowledge-base/the-basics-a-beginners-guide-to-docker/

[45] *Kubernetes* [online]. [vid. 2022-01-30]. Dostupné z: https://kubernetes.io/

[46] What is Kubernetes (K8s)? *ARMO* [online]. [vid. 2022-01-29]. Dostupné z: https://www.armosec.io/glossary/kubernetes/

[47] *Microservices vs SOA: What's the Difference? - DZone Microservices* [online]. [vid. 2021-10-16]. Dostupné z: https://dzone.com/articles/microservices-vs-soa-whats-the-difference

[48] *Monolith vs SOA vs Microservices vs Serverless Architecture* [online]. [vid. 2021-10-16]. Dostupné z: https://rubygarage.org/blog/monolith-soa-microservices-serverless

[49] JIRÁSEK, Petr, Luděk NOVÁK a Josef POŽÁR. Výkladový slovník kybernetické bezpečnosti. 2015, 242.

[50] *Hrozby* [online]. [vid. 2022-04-25]. Dostupné z: https://www.kybez.cz/hrozby/

[51] *Server Administration Guide* [online]. [vid. 2022-04-27]. Dostupné z: https://www.keycloak.org/docs/latest/server_admin/#features

[52] Introduction. *resilience4j* [online]. [vid. 2022-04-27]. Dostupné z: https://resilience4j.readme.io/docs

Seznam zkratek

API	Application Programming Interface
AWS	Amazon Web Services
AMQP	Advanced Message Queuing Protocol
CD	Continous delivery
CI	Continous integration
CLI	Command Line Interface
CORS	Cross-origin resource sharing
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheet
DI	Dependecy Injection
DSL	Domain-specific language
DDoS	Distributed denial-of-service
EE	Enterprise Edition
ESB	Enterprise Service Bus
GPL	General Public Licence
GUI	Graphic User Interface
GDPR	General Data Protect Regulation
HAL	Hypertext Application Language
HQL	Hibernate Query Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JPA	Java Persistence API
JVM	Java Virtual Machine
JWT	JSON Web Token
LTS	Long-Term-Support
LDAP	Lightweight Directory Access Protocol
LGPL	Lesser General Public License
ORM	Object-relation mapping
OTP	Open Telecom Platform

REST	Representational state transfer
RxJS	Reactive Extension for JavaScript
SDK	Software Development Kit
SOA	Service Oriented Architecture
SPA	Single-page application
SQL	Structured Query Language
SSO	Single sign-on
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
URI	Uniform Resource Idetifier
WSDL	Web Service Definition Language
XML	Extensible Markup Language

10 Přílohy

1) Zdrojový kód
2) Zadání diplomové prác

Zdrojový kód

Zdrojové kódy jsou přiloženy:

1. Součástí diplomové práce v souboru microservices.zip součástí tohoto projektu je i soubor navod.pdf s popisem zprovoznění aplikace
2. Zveřejněny na GitHubu na odkazu https://github.com/edyzar/Diplomka

Oskenované zadání práce

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2019/2020

Studijní program: Aplikovaná informatika
Forma studia: Kombinovaná
Obor/kombinace: Aplikovaná informatika (ai2-k)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Edward Zářecký**
Osobní číslo:	**I1900556**
Adresa:	Orlice 272, Letohrad – Orlice, 56151 Letohrad, Česká republika
Téma práce:	Bezpečnostní hrozby aplikací postavených na principu Microservices
Téma práce anglicky:	Security threats of applications based on Microservices
Vedoucí práce:	Mgr. Josef Horálek, Ph.D. Katedra informačních technologií

Zásady pro vypracování:

Cílem práce je analyzovat relevantní bezpečností hrozby aplikací postavených na Microservices Oriented Architecture a navrhnout metody pro jejich eliminaci. V teoretické části práci se autor zaměří na popis hlavních principů Microservices Oriented Architecture v komparaci vůči Service Oriented Architecture. Dále se autor zaměří na způsoby uložení dat v aplikacích založených na mikroslužbách, nástroje a služby využívané při nasazování mikroslužeb a posílání zpráv v aplikacích založených na mikroslužbách. V praktické části autor navrhne a realizuje aplikaci využívající principu Microservices Oriented Architecture. Aplikaci analyzuje s důrazem na relevantní bezpečnostní hrozby ovlivněné využitím mirkroslužeb a navrhne technická opatření pro jejich eliminaci.

Seznam doporučené literatury:

ERL, Thomas, Paulo MERSON a Roger STOFFERS. *Service-oriented architecture: analysis and design for services and microservices*. Second edition. Boston: Prentice Hall, [2017]. Prentice Hall service technology series from Thomas Erl. ISBN 9780133858587.
NADAREISHVILI, Irakli. *Microservice architecture: aligning principles, practices, and culture*. Sebastopol, CA: O'Reilly Media, 2016. ISBN 9781491956250.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum: