Architektura, vývoj a nasazení škálovatelných cloudových aplikací

Bc. Matej Makara

Diplomová práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Matej Makara**
Osobní číslo:	**A21440**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Architektura, vývoj a nasazení škálovatelných cloudových aplikací**
Téma práce anglicky:	**Architecture, Development and Deployment of Scalable Cloud Applications**

Zásady pro vypracování

1. Rozveďte problematiku a možnosti pokročilého vývoje moderních aplikací.
2. Popište přístup vývoje „Infrastructure as code".
3. Zvolte vhodné nástroje a prostředky pro vývoj aplikace s přístupem „Infrastructure as code".
4. Navrhněte aplikaci jako demonstraci dané problematiky.
5. Proveďte ukázkovou implementaci a zpracujte vlastní tutoriál.
6. Řešení vhodně popište a otestujte.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. RICHARDS, Mark a Neal FORD. Fundamentals of Software Architecture: An EngineeringApproach. Sebastopol: O'Reilly Media, 2020. ISBN 9781492043454.
2. RICHARDS, Mark a Neal FORD. Software Architecture: The Hard Parts: Modern Trade-OffAnalyses for Distributed Architectures. Sebastopol: O'Reilly Media, 2021. ISBN9781492086895.
3. NEWMAN, Sam. Building Microservices: Designing Fine-Grained Systems. 2nd edition.Sebastopol: O'Reilly Media, 2021. ISBN 9781492034025.
4. KIM, Gene, Jez HUMBLE, Patrick DEBROIS, John WILLIS a Nicole FORSGREN. The DevOpsHandbook: How to Create World-Class Agility, Reliability, & Security in TechnologyOrganizations. Second edition. Portland: IT Revolution Press, 2021. ISBN9781942788003.
5. MORRIS, Kief. Infrastructure as Code: Dynamic Systems for the Cloud Age. 2nd edition.Sebastopol: O'Reilly Media, 2021. ISBN 9781098114671.
6. LUKŠA, Marko. Kubernetes in Action. Shelter Island: Manning, 2018. ISBN 9781617293726.
7. HUMBLE, Jez a David FARLEY. Continuous Delivery: Reliable Software Releases throughBuild, Test, and Deployment Automation. Boston: Addison-Wesley Professional, 2010.ISBN 9780321601919.

Vedoucí diplomové práce: **Ing. Petr Žáček, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **2. prosince 2022**
Termín odevzdání diplomové práce: **26. května 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Bc. Matej Makara v. r.

podpis studenta

ABSTRAKT

Práca sa zaoberá popisom softvérových architektúr využitých pri tvorbe moderných aplikácií. Popisuje vlastnosti vybraných softvérových architektúr a venuje sa prístupu Infrastructure as Code. Taktiež popisuje dostupné nástroje využívané prístupom Infrastructure as Code. Práca obsahuje návrh ukážkovej aplikácie a tutoriál, ktorý čitateľa prevedie implememementáciou tejto aplikácie. Aplikácia je založená na microservice architektúre a je automaticky nasadená pomocou IaC nástroja Pulumi. Implementovaná aplikácia zahŕňa škalovanie pomocou Kubernetes, ktoré je otestované záťažovým testom.

Klíčová slova: architektúra aplikácie, microservice, Infrastructure as Code, Pulumi

ABSTRACT

This thesis deals with the description of software architectures used in the creation of modern applications. It describes the features of selected software architectures and discusses the Infrastructure as Code approach. It also describes the available tools used by the Infrastructure as Code approach. The thesis includes a design of a sample application and a tutorial that guides the reader through the implementation of this application. The application is based on a microservice architecture and is automatically deployed using the IaC tool Pulumi. The finalized application includes scaling using Kubernetes, which is then stress tested.

Keywords: application architecture, microservice, Infrastructure as Code, Pulumi

OBSAH

ÚVOD

Nároky a požiadavky na aplikácie v modernom svete neustále narastajú. V dôsledku zvyšujúceho sa počtu užívateľov je nutné, aby boli aplikácie stabilné, rýchle a dostupné takmer okamžite kdekoľvek na svete. Naplnenie všetkých týchto požiadaviek vyžaduje vytvorenie aplikácie pomocou architektúry a infraštruktúry, ktorá je schopná tieto nároky zvládnuť.

Táto diplomová práca sa zaoberá vývojom a architektúrou moderných aplikácií, ktoré sú následne nasadené v cloud prostredí pomocou prístupu Infrastructure as Code. Jej cieľom je uviesť čitateľa do problematiky voľby správnej architektúry pri tvorbe aplikácií a prostredníctvom tutoriálu predviesť ukážkovú implementáciu zvolenej architektúry za využitia prístupu Infrastructure as Code.

Teoretická časť práce sa venuje popisu rôznych architektonických vzorov, ktoré môžu byť použité pri tvorbe aplikácie. Vyzdvihuje prednosti a nevýhody daných prístupov a odporúča vhodné prípady na ich použitie. Práca ďalej popisuje prístup vývoja Infrastructure as Code, ktorý je používaný na automatizovanie nasadenia infraštruktúry vyvíjanej aplikácie a jej súčastí. Definuje vlastnosti a pravidlá, ktoré tento vývojový prístup dodržuje. V neposlednej rade sú v teoretickej časti spomenuté dostupné nástroje a prostriedky vhodné pre vývoj aplikácie pomocou prístupu Infrastructure as Code. Ide o nástroje, ktoré umožňujú vytvorenie požadovanej infraštruktúry na základe zdrojového kódu.

Praktická časť práce sa venuje návrhu aplikácie, ktorá vhodne demonštruje výber a implementáciu jednej z opísaných architektúr v teoretickej časti. Obsahuje popis funkcionality a architektúry jednotlivých častí projektovanej aplikácie. Následne prevedie čitateľa tutoriálom implementácie každej časti aplikácie pričom je braný dôraz na využitie prístupu a nástrojov, ktoré poskytujú funkcionalitu Infrastructure as Code. Tutoriál taktiež obsahuje návod ako s použitím spomenutých nástrojov vytvoriť automatizované nasadenie vyžadovaných zdrojov v cloud prostredí.

Na záver je vytvorená aplikácia a jej nasadená infraštruktúra otestovaná pomocou záťažového testu, ktorý overí nastavené parametre počas nasadzovania. Funkcionalita aplikácie je dodatočne prezentovaná pomocou implementovaného užívateľského rozhrania.

I. TEORETICKÁ ČÁST

1 ARCHITEKTONICKÉ VZORY

Architektonické štýly, nazývané taktiež aj architektonické vzory, popisujú pod spoločným združeným názvom vzťahy a charakteristiky medzi súčasťami navrhovaného systému. Tento názov reprezentuje typickú štruktúru systému, modely akým spôsobom bude architektúra nasadená, stratégie na prácu s dátami a množstvo ďalších informácií [1]. Každá z nižšie spomenutých architektúr popisuje často používanú známu topológiu spolu s jej charakteristikou a jej výhodami a nevýhodami.

1.1 Monolitické verzus distribuované architektúry

Architektonické vzory môžeme rozdeliť na dva základné typy - *monolitické*(kód celej aplikácie je nasadený ako funkčný celok) a *distribuované*(aplikácia je distribuovaná ako viacero funkčných jednotiek, ktoré sú spoločne navzájom prepojené pomocou protokolov určených na vzájomnú vzdialenú komunikáciu) [1]. Táto klasifikácia nie je dokonalá, ale keďže vzory používané pri distribuovaných architektúrach, zdieľajú spoločné výzvy a problémy, ide o vhodnú líniu separácie pre potreby tejto práce.

1.1.1 Problémy distribuovaných architektúr

Distribuované systémy majú oproti monolitom omnoho vyšší výkon, sú ľahko škálovatelné a vysoko dostupné. Tieto schopnosti však prinášajú niekoľko problémov a nevyhnutných kompromisov.

- **Spoľahlivosť siete** - Vývojári často zabúdajú, že internetová sieť je z vlastnej podstaty vysoko nespoľahlivá. Aj napriek pokrokom v sieťových architektúrach, zostávajú tieto siete naďalej nespoľahlivé, čo je pre distribuované architektúry podstatné, keďže časť komunikácie prebieha cez tieto siete [1, 2]. Príkladom môže byť *Služba A*, ktorá poslala požiadavku na spracovanie dát do *Služby B*. Avšak táto služba nikdy neobdrží tieto požiadavky v dôsledku chyby pri prenose v sieti. Z tohto dôvodu, je nutné v týchto službách spracovávať udalosti, ako napríklad *timeout*[1]. Čím je systém viac závislý na komunikácií cez sieť (napríklad *microservice* architektúra), tým je systém potenciálne menej spoľahlivý.
- **Latencia** - V prípade monolitickej architektúry je oneskorenie, spôsobené volaním metódy alebo funkcie iného komponentu, merateľné rádovo v nanosekundách. Ak však použijeme sieťový komunikačný protokol, ako napríklad *REST* alebo *RPC* dostávame hodnoty rádovo v inštanciách milisekúnd[1, 2]. Vzhľadom na to, ako fungujú distribuované systémy a množstvo komunikácie medzi službami, je sledovanie oneskorenia jeden z najdôležitejších faktorov. Príkladom, ak

vykonanie určitej funkcie vyžaduje 10 na seba nadväzujúcich požiadaviek a každá zaberá priemerne $100ms$, tak získanie odpovede na pôvodnú požiadavku zaberie priemerne $1000ms$ [1]. Preto je v prípade distribuovaných architektúr nutné sledovať a redukovať oneskorenie.

- **Množstvo dát** - Pri monolitickej architektúre nie je nutné tento problém zvyčajne riešiť pretože dáta sú prenášané v rámci celistvého systému. Akonáhle je však systém rozdelený na viacej služieb (*microservices*) dochádza k výraznému zvýšeniu nárokov na množstvo dát, ktoré si musia jednotlivé služby interne zasielať, čo samozrejme ovplyvňuje oneskorenie a stabilitu celého systému [3].
  Príklad: *Služba A* potrebuje na vykonanie svojej funkcionality údaje o *mene* užívateľa. Tieto údaje sú spravované *Službou B*, ktorá, však obsahuje len endpoint na získanie všetkých informácií o užívateľovi - čo je 50 atribútov. *Služba A* využije tento endpoint na získanie *mena*, pričom však prenesie veľké množstvo nepotrebných údajov. Pri častom opakovaní tejto funkcionality prenesie volanie tohto endpointu gigabajty nepotrebných dát[1].
  Takéto plytvanie je nutné redukovať a rieši sa viacero spôsobmi. Ich podstatou je zväčšenie granularity dostupných dát, ktoré si medzi sebou služby posielajú[1, 4].

- **Bezpečnosť siete** - Ide o ďalší z faktorov, ktorý zvyšuje náročnosť práce s distribuovanými systémami. Každý dostupný endpoint, či už ide o verejne dostupný koncový bod alebo o komunikáciu medzi službami, je vstupnou bránou a vektorom, ktorý môže útočník využiť[1]. Potreba kontrolovať zabezpečenie a práva v každom bode systému, je ďaľší z dôvodov, ktorý spomaľuje responzivitu distribuovaného systému[3].

- **Štruktúra siete** - Štruktúra siete opisuje všetky súčasti, ktoré tvoria danú sieť - ako napríklad routre, switche, firewall a podobne. Nemôžeme predpokladať, že táto štruktúra je nemenná, pretože zmeny v tejto štruktúre, môžu znamenať navýšenie latencie, čo môže v dôsledku vyvolať prekročenie limitov oneskorenia a vyústiť v nefunkčnosť produkčného prostredia. Preto je potrebné tieto zmeny monitorovať a prípadne upraviť nastavenia limitov[1, 2].

- **Cena** - Do úvahy je potrebné brať taktiež cenu, ktorá pri prechode z monolitickej na distribuovanú architektúru, výrazne narastá. Navýšenie je spôsobené zvýšením množstva potrebného hardware - primárne serverov, firewallov, proxy a podobne[3]. V ére, keď je tento hardware poskytovaný cloud providermi, je veľmi jednoduché, dosiahnuť nečakane vysoké ceny, v dôsledku zle nastavených limitov a upozornení na strane providera.

1.1.2 Ďalšie faktory

Okrem vyššie zmienených problémov, ktoré sa nachádzali aj pri monolitickej architektúre, nasledujúce faktory musíme brať do úvahy len v prípade distribuovaných systémov.

- **Distribuované logovanie** - keďže daný systém pozostáva z viacero služieb, z ktorých má každá vlastný nezávislý log a ktorý je umiestnený v inej lokalite, je nájdenie zdroja problému pomocou logov v distribuovaných systémov veľmi zložité[3]. Preto je zvyčajne odporúčané používať špeciálne vytvorené nástroje na prácu a konsolidáciu logov.
- **Distribuované transakcie** - V monolitickej aplikácií sa zvyčajne spoliehame na *ACID* princípy a ich *commit* a *rollback* funkcionality[1]. Distribuovaná architektúra spolieha na princíp tzv. *eventuálnej konzistencie*, ktorý zaručuje, že dáta spracované rozdielnymi službami, dosiahnu niekedy v nešpecifikovanom budúcom čase, konzistentný stav [1, 2]. Ide o kompromis, pri ktorom získavame vysokú škálovateľnosť, výkon a dostupnosť, ale strácame konzistenciu a integritu dát.
- **Spravovanie kontraktov a verzovanie** - Kontrakt popisuje funkcionalitu danej služby a formátovanie dát, na ktoré spoliehajú obe strany - klient aj server. Udržiavanie korektných kontraktov je v distribuovaných systémoch zložité, pretože jednotlivé služby sú zvyčajne spravované rôznymi tímami. Udržovanie správnych kontraktov a verzií vyžaduje dodržiavanie zavedených verzovacích politík v spoločnosti a precíznu komunikáciu medzi súčasťami organizácie [1]. Z implementačného hľadiska nám môžu pomôcť praktiky ako verzovanie *API* alebo tzv. *feature flags* [4, 5].

1.2 Service-based architektúra

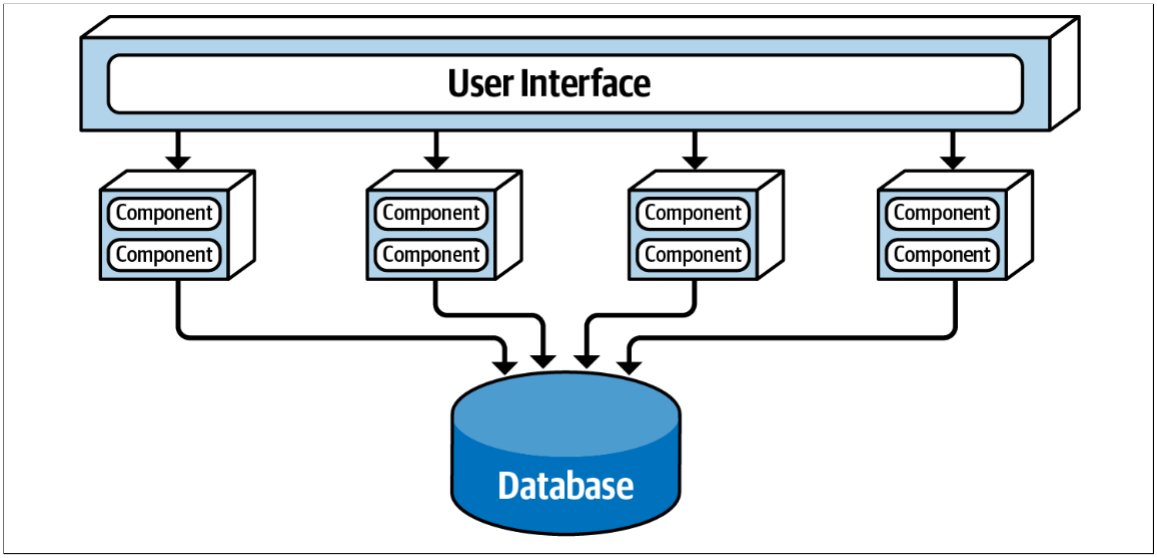
Service-based architektúra alebo architektúra založená na službách je čiastočným hybridom microservice architektúry a je považovaná za jednu z najviac obecných architektúr, hlavne kôli svojej flexibilite. Aj keď ide o distribuovanú architektúru, jej zložitosť a cena, je v porovnaní s inými architektúrami zanedbateľná[1, 7].

1.2.1 Topológia

Základná štruktúra pozostáva zo zvlášť nasadeného užívateľského rozhrania, viacero zvlášť nasadených služieb a monolitickej databáze. Služby nadsadené v tejto architektúre zvyčajne vystihujú určitú doménu potrieb danej aplikácie a nevyžadujú kontajnerizáciu(nasadenie tejto služby, pomocou kontajnera, ako napríklad *Docker* však nie je vylúčené). Pretože služby zdieľajú typicky jedinú monolitickú databázu, počet služieb

býva obmedzený do rozmedzia zvyčajne 4 až 12 služieb [1, 2]. Vo väčšine prípadov sú službe nasadené len v jednej inštancií. Je možné vytvoriť viacero inštancií, v prípade, že sú nároky na dostupnosť danej služby vysoké a za predpokladu, že do architektúry zakomponujeme *load-balancing*, ktorý nasmeruje užívateľské rozhranie na správnu a funkčnú inštanciu.

Užívateľské rozhranie pristupuje ku službám pomocou sieťových protokolov, zvyčajne *REST*, prípadne *RPC* alebo *Websocket*. Existuje možnosť pridať *API gateway* alebo *proxy*, ktoré smeruje požiadavky z jednotnej webovej adresy na korešpondujúce cieľové služby [1, 2]. Podstatnou súčasťou *service-based* architektúry je centralizovaná zdieľaná databáza, ktorá zjednodušuje prácu jednotlivým službám. K databáze je možné pristupovať, akoby išlo o tradičnú databázu v monolitickej aplikácií [1]. Základná topológia je ukázaná na obrázku 1.1.



Obrázek 1.1 Service-based topológia
[1]

1.2.2 Varianty topológie

Monolitické užívateľské prostredie môže byť rozdelené na viacero izolovaných častí, kde každá časť rozhrania komunikuje len s niektorými službami, prípadne každá služba môže mať svoje vlastné, izolované užívateľské rozhranie. Podobne to platí aj pri delení jednotnej monolitickej databáze, kde môžeme databázu rozdeliť na oddelené celky, podľa požadovanej granularity daného návrhu systému[1, 2].

V prípade delení databáze, je však potrebné dbať na to, aby jednotlivé dátové celky neboli potrebné v inej databáze, ktorú používa oddelená služba. Vyhneme sa tak komunikácií medzi službami a duplikácií dát, čo je pri *service-based* architektúre nežiadúce [1, 4]. Jednotlivé služby môžu byť implementované zvyčajne dvoma spôsobmi.

Prvým z nich je rozdelenie na *API vrstvu*, *vrstvu s business logikou* a *perzistentnú vrstvu*. *API vrstva* je zodpovedná za spracovanie prichádzajúcich požiadaviek, *vrstva s business logikou* spracuje a transformuje prijaté dáta a požiadavky a *perzistentná vrstva* sa postará o vykonanie týchto zmien v databáze [2].

Druhým prístupom je rozdelenie služieb na pod-služby, kde po prijatí požiadavky na *API vrstve* každá pod-služba vykoná svoju pridelenú prácu, ktorá pozostáva zo všetkých potrebných úkonov(spracovanie dát, transformácia, uloženie do databáze a iné) [1, 2].

1.2.3 Delenie databáze

Pri *service-based* architektúre využívame zvyčajne jednu monolitickú databázu. Toto pevné napojenie na databázu s viacerými službami, môže viesť k problémom pri zmenách schémy tabuliek databáze. V takomto prípade, môže zmena schémy vyžadovať zmeny vo všetkých službách a ich opätovné nasadenie, čo v nadväznosti vyžaduje zložitú koordináciu medzi rôznymi tímami a zvyšuje cenu spravovania týchto služieb. Tieto súbory, ktoré obsahujú triedy s deklaráciami schém tabuliek, sú väčšinou reprezentované ako zdieľané knižnice používané všetkými nasadenými službami(*.dll, .jar, .d.ts súbory*) [1, 2].

Tento spôsob zdieľania entít a udržaiavnia kontraktov je najmenej efektívny. Akákoľvek zmena bude vyžadovať zmeny vo všetkých službách. Verzovanie týchto zdieľaných knižníc dokáže tento problém mitigovať, ale neustále sledovanie, ktorá služba používa danú verziu je časovo a manuálne náročné [1, 4].

Vhodnejším prístupom je rozdelenie databáze a do logických celkov podľa domén, v ktorých sa dáta z databáze používajú. Následne je vhodné toto rozdelenie reflektovať vytvorením viacerých doménových knižníc, ktoré sú používané len službami, ktoré ich vyžadujú k svojmu fungovaniu [1, 2, 4]. Prípadné zmeny musíme vykonať len v službách, ktoré využívajú túto čiastkovú doménovú knižnicu.

1.2.4 Využitie architektúry

Service-based architektúra je jednou z najviac versatilných architektúr. Hoci neposkytuje takú výkonnosť a škálovateľnosť ako iné typy, jej jednoduchosť a cenová výhodnosť, si nájde uplatnenie vo veľa návrhoch systémov.

Táto architektúra je vhodná, ak dokážeme naše požiadavky na aplikáciu rozdeliť na jednotlivé oddelené služby a domény, ktoré medzi sebou nemusia komunikovať [1, 2]. Vykonávanie zmien a udržiavanie kontraktov je tak oveľa jednoduchšie. Práca s databázou je taktiež zjednodušená, keďže môžeme spoliehať na *ACID* princípy a nie na princíp *eventuálnej konzistencie* ako v prípade iných architektúr [4, 6].

V neposlednom rade dosahuje táto architektúra dobrú úroveň granularity a modularity bez toho aby sme sa museli obávať problémov ako sú *orchestrácia* a *choreografia* medzi službami [2].

Orchestrácia popisuje koordináciu medzi viacerými službami pomocou služby, ktorá monitoruje a koordinuje priebeh danej transakcie medzi službami. *Choreografia* je naopak komunikácia a koordinácia medzi viacerými službami, ktoré vykonávajú transakciu, bez toho aby boli riadené ďalšou službou [1]. Služby pri *service-based* architektúre zvyčajne nevyžadujú spomínanú úroveň vzájomnej koordinácie a organizácie, čo považujeme za výhodu.

1.3 Event-driven architektúra

Event-driven architektúra je distribuovaná architektúra určená na tvorbu škálovateľných a výkonných aplikácií. Je použiteľná pre malé, aj vysoko komplexné a náročné aplikácie [1, 4].

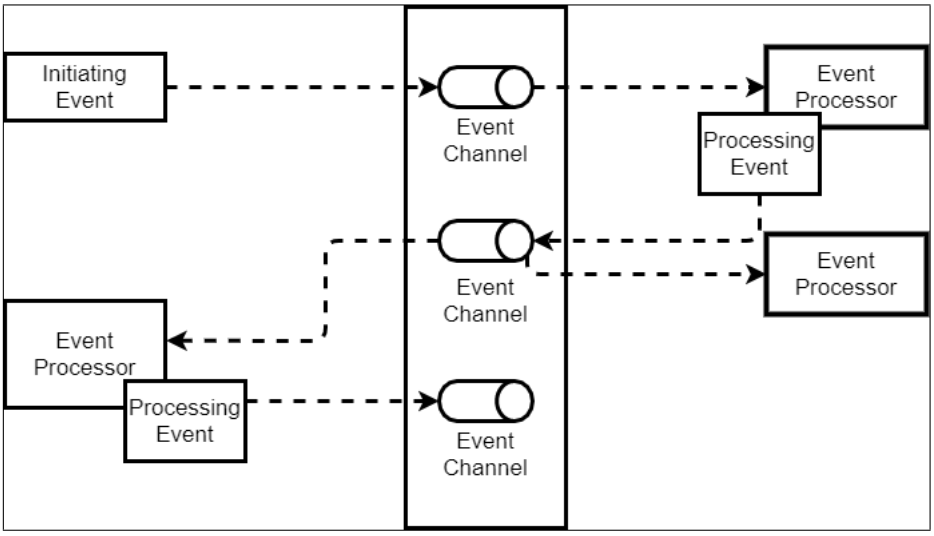
Je založená na princípe oddelených častí, ktoré nezávisle na sebe spracúvajú prichádzajúce udalosti. Model tejto architektúry môže byť použitý samostatne alebo ako súčasť iných architektúr. Väčšina aplikácií pracuje na princípe *request* modelu. V tomto prípade sú požiadavky na spracovanie zaslané do organizátora požiadaviek. Organizátor môže byť reprezentovaný užívateľským rozhraním, *API vrstvou* alebo zbernicou pre správy. Následne sú prijaté správy synchrónne zasielané jednotkám, ktoré sú schopné tieto požiadavky spracovať a obslúžiť [1].

Opakom *request* modelu je *event-based* systém, ktorý reaguje na prijatú správu alebo udalosť, vyvolaním príslušnej vhodnej akcie. *Event-driven* architektúra existuje v dvoch variantoch: *broker topológia* a *mediátor topológia*.

1.3.1 Broker topológia

Špecifikom broker topológie je, že neobsahuje centrálny koordinátor správ, ktoré systém prijíme. Namiesto toho sú správy distribuované cieľovým jednotkám, ktoré tieto správy spracujú, pomocou *message brokera*(napríklad *RabbitMQ* alebo *Apache Kafka*) [1, 8]. Tieto eventy sú smerované na základe kanálov na ktoré následne reagujú príslušné procesory. Základom *broker topológie* sú štyri typické súčasti.

Prvým z nich je iniciačný event, ktorý zaháji proces spracovania. Táto správa je zaslaná do správneho kanála sprostredkovateľa správ. Následne je táto správa prijatá jedným z dostupných procesorov, ktorý túto správu obslúži a spracuje. Po dokončení svojej práce, informuje o dokončení spracovania zaslaním novej správy do *message brokera*. Tento kolobeh zasielania nových správ a ich spracovanie sa následne opakuje. Topológia je zobrazená na obrázku 1.2.



Obrázek 1.2 Broker topológia
[8]

Výhodou tejto topológie je, že jednotlivé procesory udalostí sú na sebe vysoko nezávislé. Akonáhle je obsluha danej správy dokončená, procesor je uvoľnený a pripravený spracovať ďalšiu správu v poradí. V prípade potreby môžeme systém veľmi jednoducho škálovať, keďže nám v prípade spomalenia postačuje pridať viacej spracovacích jednotiek pre daný typ správy [1, 7].

Táto topológia má aj niekoľko nevýhod. Sledovanie priebehu spracovania biznis logiky danej transakcie je takmer nemožné, pretože systém neobsahuje centrálny koordinátor, ktorý by sledoval stav transakcie. V prípade, že dôjde k chybe, systém o tejto chybe nemusí byť informovaný a proces spracovania môže pokračovať napriek chybe. Reštartovanie celej biznis logiky danej transakcie taktiež nie je touto topológiu podporované [1, 7].

1.3.2 Mediátor topológia

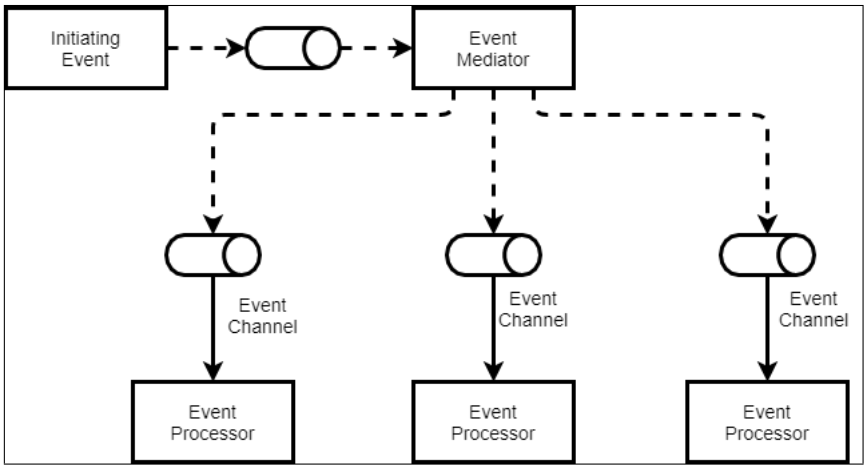
Mediátor topológia v rámci *event-driven* architektúry rieši niektoré nedostatky broker topológie. Hlavným prvkom tejto topológie je mediátor udalostí, ktorý riadi a kontroluje sériu operácií potrebných pre vzájomnú koordináciu spracovania medzi spracovateľmi udalostí [1, 8]. Základnými komponentami tejto topológie sú:

- udalosť, ktorú je potrebné obslúžiť
- frontu udalostí
- mediátor udalostí
- kanály jednotlivých typov udalostí

- procesory udalostí

Podobne ako pri *broker topológií* je celý proces spracovania udalostí naštartovaný prijatím prvej udalosti, ktorú je nutné obslúžiť. Na rozdiel od *broker topológie* je táto prvotná udalosť zaslaná do fronty, z ktorej sú udalosti spracované mediátorom. Mediátor udalostí pozná všetky kroky, ktoré sú potrebné na spracovanie prijatej udalosti [1].

Následne rozpošle čiastkové udalosti do jednotlivých typov kanálov. Tieto kanály sú sledované procesormi udalostí, ktoré na tieto udalosti zareagujú a začnú spracovanie týchto čiastočných udalostí. O dokončení spracovania je pomocou kanálov informovaný mediátor. Na rozdiel od *broker topológie* neinformujú spracovávatelia udalostí o dokončení práce iné časti systému [1, 2, 7]. Mediátor topológia je zobrazená na obrázku 1.3.



Obrázek 1.3 Mediátor topológia
[8]

Napriek tomu, že *mediátor topológia* rieši niektoré problémy spojené s *broker topológiou*, má aj táto architektúra svoje nevýhody. Jednou z nich je nemožnosť presne deklarovať a definovať komplexné procesy, ktoré nastávajú pri vytváraní udalostí [1, 2]. Z toho dôvodu využíva väčšina architektúr hybridný model, ktorý kombinuje vlastnosti oboch topológií [2, 8]. *Mediátor topológia* je teda prevažne využívaná na modelovanie všeobecných stabilných procesov.

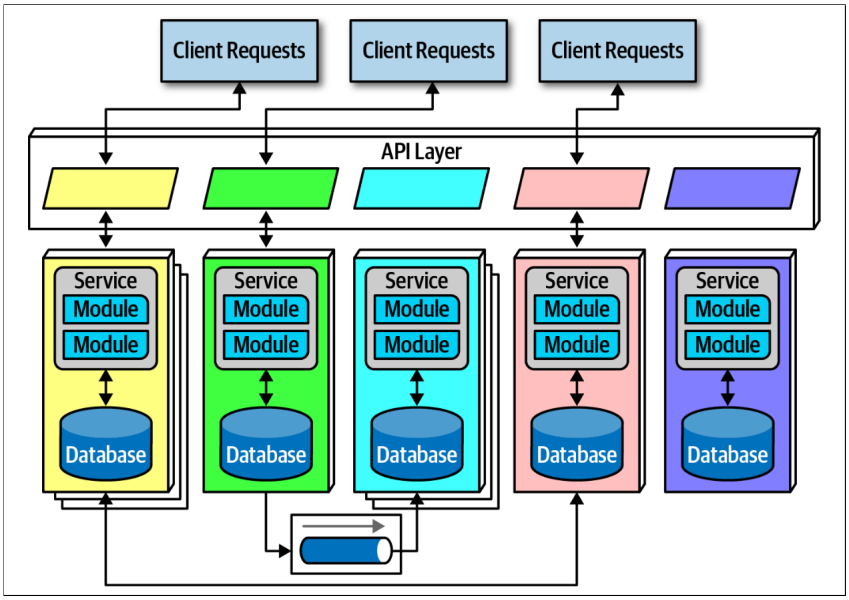
Okrem toho, je nutné brať do úvahy faktor, že aj keď môžeme škálovať počty procesorov udalostí takmer neobmedzene, samotný mediátor musí zvládnuť toto zvýšené zaťaženie. Mediátor sa tak niekedy môže stať úzkym miestom, ktorý spomaľuje celý navrhnutý systém [1, 2]. Poslednou nevýhodou je fakt, že samotné procesory udalostí nie sú vzájomne príliš oddelené ako v *broker topológií*. Výkon a účinnosť tejto topológie teda nie je tak vysoký pretože spracovanie všetkých udalostí je riadené *mediátorom* [1, 2].

Pri výbere medzi *broker topológiou* a *mediátor topológiou* musí architekt systému voliť kompromis medzi možnosťou kontroly nad stavom spracovania jednotlivých udalostí a vysokou priepustnosťou a škálovateľnosťou.

1.4 Microservice architektúra

Microservice architektúra je pravdepodobne najpopulárnejší architektonický štýl posledných rokov. Prvoradým cieľom mikroslužieb je vysoké oddelenie jednotlivých služieb a udržanie si kontextu v ktorom majú služby pracovať.

Každá služba je koncipovaná tak, že obsahuje všetky súčasti, ktoré potrebuje na svoje vlastné fungovanie. Služba zvyčajne obsahuje databázu a iné komponenty, ktoré s ňou následne komunikujú. Ukážka tejto architektúry je na obrázku 1.4. Rôzne zásady, ktoré dodržuje architektúra mikroslužieb sú popísané v nasledujúcich bodoch.



Obrázek 1.4 Topológia microservice architektúry
[1]

1.4.1 Distribuovanosť

Každá služba beží vo svojom vlastnom procese, ktorý je spustený buď vo virtualizovanom prostredí alebo v kontajneri. Oddelenie služieb do takejto miery umožňuje jednoduché riešenie bežného problému v architektúrach, ktorý je spojený s so zdieľanou infraštruktúrou medzi službami [1, 3]. Pred príchodom cloud infraštruktúry a technológie kontajnerizácie bolo nepraktické, aby mala každá služba vlastnú infraštruktúru.

Pokles výkonu je negatívnym dôsledkom distribuovanej povahy mikroslužieb. Získanie výsledku pomocou *REST API* trvá omnoho dlhšie ako zavolanie implementovanej funkcie [3].

Architektúru taktiež spomaľuje nutnosť dodržavať bezpečnostné zásady keďže služby navzájom komunikujú cez sieť a identitu jednotlivých požiadaviek je nutné overovať [1, 4]. Taktiež nie je odporúčané vykonávať transakcie, ktoré vyžadujú spoluprácu viacerých služieb. Lepším prístupom je voľba vhodnej granularity pri vytváraní dizajnu služby [3, 6].

1.4.2 Kontext

Základným princípom mikroslužieb je ohraničenie kontextu. To znamená, že služba by mala modelovať, implementovať a vykonávať len funkcie na ktoré je určená. Každá služba potom musí obsahovať všetko, čo potrebuje na svoje fungovanie v rámci systému, vrátane tried, iných komponentov a databázových schém [1, 3]. Udržuje sa tak princíp čo najväčšieho rozpojenia závislostí medzi službami. Z toho dôvodu preferuje táto architektúra duplikáciu namiesto zväčšenia závislostí [3].

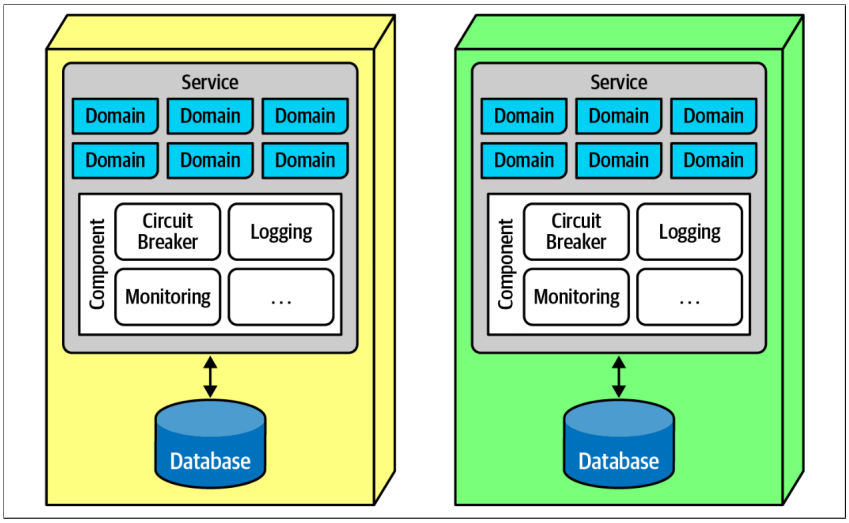
1.4.3 Izolácia dát

Ďalšou požiadavkou na mikroslužby, ktorá je podmienená požiadavkou na dodržiavanie kontextu, je izolácia dát medzi mikroslužbami. Mnohé iné architektonické štýly používajú jedinú databázu ako zdroj dát. Architektúra mikroslužieb vyžaduje separáciu dát aby bol dodržaný princíp čo najväčšieho rozpojenia závislostí [1, 3].

Napriek tomu, že dodržiavanie tejto požiadavky vytvára problémy spojené so zložitosťou práce s dátami a infraštruktúrou, môže prinášať aj výhody. Vývojári v rámci tímu si môžu v dôsledku vysokého rozpojenia dependencií vyberať nástroje a databázové služby, ktoré vyhovujú ich požiadavkám. Prípadná potreba zmien tak neovplyvní iné služby ani celú aplikáciu alebo systém.

1.4.4 Opätovné použitie

Ako bolo spomenuté v časti 1.4.3 architektúra mikroslužieb preferuje duplikáciu zdrojov, kôli udržaniu nízkeho vzájomného naviazania. Existujú však samozrejme prípady, kedy vývojári potrebujú prepoužiť už vytvorenú funkcionalitu. Spadajú sem funkcie ako monitorovanie, logovanie, alebo zastavenie propagácie zlyhaní. Preto existuje v prípade architektúry mikroslužieb snaha tieto funkcie separovať. Tento oddelený komponent, ktorý zoskupuje spomenuté funkcie, sa typicky nazýva *sidecar* [1, 3]. Jeho úlohou je spracovávať všetky typické požiadavky, ktoré má každá mikroslužba. V prípade potrebných zmien kódu *sidecar* obdrží každá služba, ktorá *sidecar* implementuje, novú zdieľanú verziu. Tento prístup je ukázaný na obrázku 1.5.



Obrázek 1.5 Ukážka prístupu sidecar
[1]

1.4.5 Komunikácia

Pri navrhovaní komunikácie medzi mikroslužbami je najdôležitejšie rozhodnutie medzi synchrónnou a asynchrónnou komunikáciou. Synchrónna komunikácia vyžaduje aby volajúca služba čakala na odpoveď inej služby [9]. Komunikácia medzi službami dodržiava tieto pravidlá:

- Každá mikroslužba by mala vedieť akým spôsobom môže kontaktovať iné služby. To je dosiahnuté pomocou protokolov, ktoré každá služba definuje [3, 9]. Patria sem protokoly ako *REST*, *gRPC*, fronta správ a pod.
- Keďže každá mikroslužba, môže byť implementovaná použitím iných technológií, musia mikroslužby podporovať komunikáciu pomocou rôznych technológií [1, 9].
- V rámci vzájomnej interoperability spolu komunikujú služby prostredníctvom siete za využitia už spomenutých protokolov [1].

Asynchrónna komunikácia využíva zasielanie udalostí a správ, pričom sa do architektúry mikroslužieb interne implementujú *broker* a *mediátor* topológie [1, 9], ktoré boli opísané v časti 1.3.

1.4.6 Orchestrácia a choreografia

Ide o funkcionality, ktoré sa snažia koordinovať komunikáciu a spoluprácu medzi oddelenýcmi službami, aby spoločne splnili požadované funkcie. V typických podmienkach kontaktuje každá služba ostatné služby podľa potreby. Nie je preto nutné použite globálneho mediátora na vzájomnú orchestráciu a choreografiu služieb [1, 3].

Ak je potrebné, aby dve služby svoju prácu vzájomne koordinovali, môže byť vytvorená tretia mikroslužba, ktorá zohráva úlohu lokálneho *mediátora*. Úlohou tohto *mediátora* je potom koordinované volanie funkcií, ktoré poskytujú napojené služby [1, 3].

2 INFRASTRUCTURE AS CODE

Infrastructure as Code popisuje novodobý prístup k automatizácií a manažmentu infraštruktúry, ktorá slúži ako základ pre všetky dnešné používané techolnógie. Základná definícia Infrastructure as Code popisuje postup, pri ktorom je správa, konfigurácia a získanie infraštruktúry potrebnej na nasdenie serverov a aplikácií vykonávaná pomocou predom definovaných skriptov, namiesto manuálnej konfigurácie hardvéru a manuálnych operácií [10].

Tento systém mení zaužívaný spôsob v rámci ktorého sa nastavenie a správa infraštruktúry - úloha, ktorá sa tradične vo veľkej miere spoliehala na fyzickú manipuláciu s hardvérom a potenciálne chybovú manuálnu konfiguráciu - teraz rieši spôsobom podobným vývoju softvéru. Požadovaný stav infraštruktúry je vyjadrený ako kód, ktorý je uložený v systéme na správu verzií, čím je uchovaná história zmien, umožnená kolabrácia alebo prípadný nutný návrat k staršej verzií [10].

Prechod od manuálnej správy infraštruktúry k automatizovanému prístupu založenému na kóde má niekoľko významných dôsledkov. Infraštruktúra je považovaná za softvér, ktorý píšu a spravujú softvéroví inžinieri alebo správcovia systémov s použitím rovnakých metodík ako typický aplikačný softvér, vrátane správy verzií, kontinuálnej integrácie, revízií kódu a automatizovaného testovania. *IaC* je teda neoddeliteľnou súčasťou moderného *DevOps* a agilného softvérového vývoja [10, 11].

Základný predpoklad IaC - infraštruktúra definovaná a spravovaná ako kód - sa môže prejavovať rôznymi spôsobmi, od jednoduchých skriptov, ktoré automatizujú konkrétne časti konfigurácie systému, až po komplexné nasadenia, ktoré definujú rozsiahle topológie infraštruktúry. V neustále sa vyvíjajúcom *IT* odvetví v ktorom sa stala agilita, spoľahlivosť a škálovateľnosť prvoradými poskytuje Infrastructure as Code jedinečný nadhľad na to ako spravovať jednoduchšie a efektívnejšie *IT* infraštruktúru.

2.1 Princípy infrastructure as Code

Prístup infrastructure as code je založený na niekoľkých základných princípoch, ktoré formujú spôsob implementácie a samotného prevádzkovania služieb. Tieto koncepty sú základom viacerých stratégií a techník využívaných v *IaC*. Ovplyvňujú všetky aspekty - od výberu vhodných nástrojov až po metódy, ktorými je infraštruktúra vytváraná, zabezpečovaná a spravovaná.

2.1.1 Idempotencia

Jedným z kľúčových princípov *IaC* je idempotencia, vlastnosť niektorých operácií v matematike a informatike. Jedná sa o jav kedy sa operácia môže použiť viackrát bez

toho, aby sa zmenil výsledok po prvom použití [10]. Idempotentná operácia v kontexte IaC vyjadruje operáciu, ktorá sa môže vykonávať opakovane, pričom sa dosiahne rovnaký stav infraštruktúry. To znamená, že bez ohľadu na to, koľkokrát sa skript na nasadenie infraštruktúry spustí, je výsledkom prostredie identické s predchádzajúcimi spusteniami [10].

Ak je napríklad skript na vytvorenie inštancie servera spustení viackrát, idempotentná operácia zabezpečí, že na konci bude existovať len jedna inštancia, namiesto toho, aby sa pri každom spustení vytvárali ďalšie inštancie. Tento princíp zabezpečuje stabilitu a konzistenciu systému. Zjednodušuje taktiež obnovu po chybe - po zlyhaní nasadenia môže byť skript spustený znova bez toho aby bolo nutné kontrolovať zdvojenie zdrojov alebo ponechanie systému v nekonzistentnom stave [10, 11].

2.1.2 Nemennosť(Imutabilita)

S idempotenciou úzko súvisí aj imutabilita, ktorá je ďalším princípom IaC. V imutabilnej infraštruktúre je nasadený komponent nemenný, čim sa zvyšuje stabilita a predvídateľnosť systému. V prípade, že je vyžadovaná zmena, je vytvorená nová verzia komponentu a stará verzia je úplne nahradená [10]. Tento koncept výrazne znižuje riziko spojené s tzv. driftom konfigurácie (odchýlenie očakávaného stavu infraštruktúry v dôsledku manuálnych zmien). Namiesto aktualizácie nasadeného servera novým kódom alebo konfiguráciou je vytvorený nový server s požadovanou konfiguráciou a pôvodné nasadenie servera je vyradené z prevádzky [10]. Odstráni sa tak možnosť výskytu nezrovnalostí medzi vývojovým, testovacím a produkčným prostredím. V prípade výskytu problémov, je návrat k pôvodnému nasadeniu veľmi jednoduchý.

2.1.3 Deklaratívny vs. imperatívny prístup

Infrastructure as Code môže byť implementované buď deklaratívnym, alebo imperatívnym prístupom. Deklaratívny prístup zahŕňa definovanie požadovaného stavu infraštruktúry bez špecifikácie podrobných krokov potrebných na jeho dosiahnutie. *IaC* nástroj je zodpovedný za zistenie, ako tento požadovaný stav dosiahnuť [10]. Vývojár je tak odbremenený od zložitosti určovania vhodných krokov na dosiahnutie tohto požadovaného stavu infraštruktúry.

Naopak imperatívny prístup vyžaduje špecifikáciu konkrétnych inštrukcií, ktoré sa musia vykonať, aby bol dosiahnutý požadovaný stav infraštruktúry. Tento prístup umožňuje detailnú kontrolu nad nastavením infraštruktúry, čo môže byť vyžadované v zložitých alebo neobvyklých prostrediach a nasadeniach [10]. Z pohľadu vývojára sú však vyžadované podrobné znalosti relevantných systémov a s tým súvisiaca narastajúca komplexnosť systému.

Výber medzi imperatívnym a deklaratívnym prístupom je spojený s konkrétnymi požiadavkami a obmedzeniami projektu, ako aj od možností, ktoré ponúka zvolený nástroj *IaC*.

2.1.4 Účel systému na správu verzií

V Infrastructure as Code sú podobne ako pri vývoji softvéru využívané systémy na správu verzií, ako napríklad *Git*. Uchováva sa tak história zmien v kóde, uľahčuje sa vzájomná kolaborácia a zmeny v kóde musia byť pred použitím skontrolované ostatnými členmi tímu [10, 11].

Na zmeny v kóde sú aplikované automatizované testy a integrácie, čím sú zachytené chyby, ktoré by inak ovplyvnili nasadenú infraštruktúru [10, 11].

2.1.5 IaC a kontinuálna integrácia/kontinuálne nasadenie (CI/CD)

IaC je podstatnou časťou CI/CD postupov, ktoré automaticky vytvárajú, testujú a nasadzujú kód vždy, keď dochádza k zmenám. Definovanie potrebnej infraštruktúry kódom umožní automatické nastavenie a vytvorenie prostredí, ktoré potrebujú tieto procesy na svoje fungovanie [10, 11]. To taktiež znamená, že testovanie a nasadzovanie je vykonávané v prostrediach, ktoré kopírujú produkčné prostredie, čím sa znižuje pravdepodobnosť chýb pri nasadzovaní. Vytvorenú infraštruktúru je možné dynamicky upravovať a škálovať na základe potrieb rôznych fáz *CI/CD* pipeline.

Okrem toho dokáže *IaC* obohatiť existujúce *CI/CD* procesy, tým, že samotné nastavenie infraštruktúry je integrované priamo do *CI/CD* pipeline. Celá aplikácia, vrátane svojej infraštruktúry, je tak integrovaná, testovaná a nasadzovaná automaticky pomocou *CI/CD* procesov a pipelines [11].

2.2 Výhody a nevýhody IaC

Prechod na *IaC* bol spôsobený veľkým počtom výhod, ktoré tento prístup ponúka. Patrí sem hlavne:

- Rýchlosť nasadenia - *IaC* umožňuje rýchle nasadenie potrebnej infraštruktúry, čím je skrátený čas potrebný na vývoj a nasadenie novej funkcionality do produkcie. Keďže je infraštruktúra definovaná ako kód, je možné ju vytvárať a upravovať oveľa rýchlejšie ako v prípade manuálnej konfigurácie [10, 11].
- Konzistencia a redukcia chybovosti - Dodržiavaním princípov idempotencie a imutability je zabezpečená konzistentnosť infraštruktúry, ktorá neobsahuje rozdiely medzi jednotlivými prostrediami [10, 11].

- Dokumentácia - Tým, že je infraštruktúra spravovaná ako kód, slúži samotný kód ako forma dokumentácie. Aktuálny stav infraštruktúry je odzrkadlený aktuálnym zdrojovým kódom, ktorého história a vykonané zmeny, sú spravované systémom na správu verzií [10, 4].

- Škálovateľnosť - *IaC* umožňuje rýchle škálovanie potrebných zdrojov v prípade ak narastá celkové zaťaženie systému. Škálovanie je dosiahnuté buď automatizovanými systémom, ktorý monitoruje zaťaženie, alebo manuálne spusteným potrebných skriptov [10, 11].

- Cenová efektivita - Správne využitie *IaC* vedie zvyčajne k úsporám na nákladoch. Infraštruktúru možno dynamicky škálovať na základe zaťaženia a znížiť tak plytvanie peniazmi, v dôsledku poplatkov za nevyužité zdroje [11].

Napriek mnohým benefitom, ktoré poskytuje *IaC* nie je využívanie tohto prístupu bez problémov a rizík.

- Komplexnosť nástrojov - Funkcionalita *IaC* je poskytovaná veľkým množstvom nástrojov, z ktorých každý, má svoje silné a slabé stránky. Orientácia v týchto možnostiach výberu nástroja, ktorý sa hodí k existujúcim technológiám použitých v projekte, môže byť náročná [10, 12]. Integrácia nových nástrojov do už existujúceho systému dokáže byť taktiež časovo náročná a zložitá.

- Prílišná automatizácia - Napriek tomu, že automatizácia je jednou z hlavných výhod *IaC* dokáže byť aj rizikom. V prípade výskytu nezachytených chýb, môžu byť tieto chyby veľmi rýchlo rozšírené do celého systému, v dôsledku automatizovaných procesov [10, 12].

- Bezpečnosť - Bezpečnosť celého systému *IaC* musí byť prvoradá. keďže infraštruktúra, je spravovaná ako kód, podlieha rovnakým rizikám ako bežný kód aplikácie, ako napríklad bezpečnostné zraniteľnosti, nezabezpečené konfigurácie alebo odhalené prístupové tokeny a heslá. Okrem toho je potrebné monitorovať, prístup k nástrojom *IaC*, keďže prístup k týmto nástrojom, poskytuje v zásade kontrolu nad celou infraštruktúrou [4, 10, 12].

2.3 Budúcnosť IaC

Prístup Infrastructure as Code sa neustále vyvíja. Dochádza k neustálemu zvyšovaniu automatizácie, užšej integrácií s už existujúcimi službami a zvyšovaniu bezpečnosti tejto metodiky. Trend používania *IaC* neustále rastie čo značí, že *IaC* sa stáva neoddeliteľnou súčasťou vývoja a nasadzovania moderného softvéru.

Neustály pokrok v oblasti umelej inteligencie a technológií strojového učenia naďalej automatizuje úlohy spojené so správou infraštruktúry vrátane odhaľovania typických chýb a optimalizácie nasadení. Zároveň sú nástroje *IaC* čoraz viac integrované do systémov používaných pri vývoji softvéru. Medzi tieto integrácie patria rôzne *CI/CD* platformy, nástroje na orchestráciu a správu kontajnerov až po poskytovateľov couldových služieb, ktorý poskytujú fyzický hardvér na ktorom funguje daná infraštruktúra.

3 NÁSTROJE INFRASTRUCTURE AS CODE

Infrastructure as Code využíva počas svojho vývoja rôzne technológie a nástroje, ktoré umožňujú kódovanie, správu a automatizáciu infraštruktúry. V tejto časti sú uvedené niektoré z hlavných nástrojov používaných v *IaC* a v praktickej časti tejto práce.

3.1 Nástroje na správu konfigurácie

Tieto nástroje umožňujú definovať požadovaný stav systému a potom s ním pracovať a udržiavať tento bežiaci systém. To je ideálne pri správe rozsiahlych a zložitých systémov [10]. Medzi tieto nástroje patria:

- Ansible - ide o *IaC* nástroj využívajúci imperatívny prístup, ktorý zabezpečuje nielen vytvorenie a nasadenie infraštruktúry, ale aj jej následnú konfiguráciu. Ansible vo veľkej miere využíva jazyk *YAML* na definovanie tzv. *YAML Playbooks* a čiastočne jazyk Python. Sú tak definované úlohy nasadenia až po manažment existujúceho systému. Nevýhodou je, že si Ansible nedrží interný prehľad o stave infraštruktúry a nesleduje vzájomné dependencie. Riešenie eventuálnych problémov môže byť zložitejšie z dôvodu menšej komunity [12].
- Puppet a Chef - ide o nástroje, ktoré problematiku *IaC* riešili medzi prvými. Oboje využívajú vlastné doménové jazyky, ktoré sa programátor musí naučiť, predtým ako začne s infraštruktúrou pracovať. Tieto nástroje pred svojim spustením taktiež vyžadujú nasadenie vlastných agentov, na každý z počítačov alebo serverov, ktoré spravujú [12].

3.2 Nástroje na vytvorenie a orchestráciu infraštruktúry

Tieto nástroje poskytujú funkcionalitu, ktorá je určená na automatizáciu celého procesu nasadzovania a následnej správy aplikácií a služieb vo viacerých systémoch alebo na viacerých platformách.

3.2.1 Terraform

Ide o open-source nástroj, ktorý používa jazyk HCL (Hashicorp Configuration Language), ktorý je založený na jazyku Golang, ktorý mnohí považujú za jeden z najľahšie naučiteľných jazykov *IaC*. Terraform obsahuje množstvo výhod, ktoré z neho robia typickú voľbu vo väčšine spoločností. Nástroj podporuje väčšinu cloud providerov a je najviac využívaný v prostrediach kde je infraštruktúra poskytovaná viacerými rôznymi cloud providermi [10, 13].

Terraform využíva na svoju funkcionalitu stavový súbor v ktorom je uložený aktuálny stav nasadenej infraštruktúry. Pred samotným nasadením je vytvorený plán, ktorý presne informuje vývojára o tom, aké zmeny budú v infraštruktúre vykonané. Nástroj funguje najlepšie pre organizácie, ktoré potrebujú neustále nasadzovať a kontinuálne meniť svoju infraštruktúru [13].

3.2.2 AWS CloudFormation

Ide o *IaC* nástroj od spoločnosti *AWS*, priamo určený na prácu s infraštruktúrou, ktorá je poskytovaná *AWS*. CloudFormation umožňuje modelovať celú infraštruktúru pomocou špecifických šablón vo formáte *JSON* alebo *YAML* [12, 14]. Tento nástroj je najvýhodnejšie využívať ak projekt implementujúci *IaC* využíva najmä infraštruktúru a iné služby, ktoré sú poskytované spoločnosťou *AWS*.

3.2.3 Pulumi

Pulumi je open-source nástroj Infrastructure as Code (IaC), ktorý umožňuje vývojárom definovať a spravovať infraštruktúru pomocou zaužívaných programovacích jazykov ako je Python, Go, C# alebo Javascript/Typescript [15]. To je hlavný dôvod, ktorý odlišuje *Pulumi* od ostatných spomenutých nástrojov, ktoré vyžadujú znalosť vlastného špecifického programovacieho jazyka. Ďalšou výhodou je, že vývojári môžu použiť všetky súčasti daného jazyka(podmienky, slučky, znovu-použiteľné funkcie) na definovanie skriptov, ktoré vytvárajú infraštruktúru [15].

Skripty napísané pomocou *Pulumi* opisujú, ako by mala byť cloudová infraštruktúra vytvorená. Pulumi využíva nasledovné koncepty, pri vytváraní infraštruktúry:

- **Stack** - Každý *Pulumi* program je nasadený do *Stacku*. Je to izolovaná, nezávisle konfigurovateľná inštancia programu Pulumi. Zvyčajne sa využívajú na označenie rôznych fáz vývoja alebo prostredí (napríklad *development*, *staging* alebo *production*). Projekt môže obsahovať neobmedzený počet *Stackov* [15].
- **Resource** - Predstavujú základné jednotky, ktoré tvoria cloudovú infraštruktúru. Môže nimi byť výpočtová inštancia, databázová inštancia alebo aj *Kubernetes* cluster. Do tejto kategórie spadajú všetky inštancie a služby, ktoré sú ponúkané rôznymi cloud providermi [15]. Každý *Resource* vyžaduje názov, ktorý je unikátny pre daný *Stack* do ktorého je *Resource* pridelený. Zároveň prijíma každý *Resource* objekt nastavenia(*Vstupy*), ktoré určujú, ako budú nastavené počiatočné hodnoty a ako sa má inštancia chovať po svojom nasadení [15].
- **Vstupy a Výstupy** - Všetky vlastnosti, ktoré sú nastavované pre jednotlivé *Resources* sú považované za *Vstupy*. Výsledky, ktoré sú naopak obdržané z nasa-

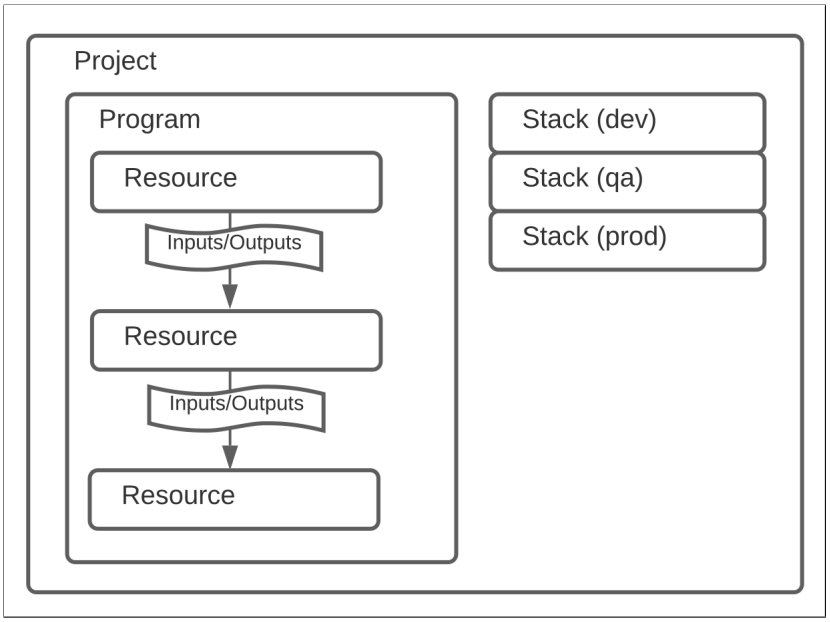
dených inštancií, sú považované za *Výstupy*. S obidvoma dátovými typmi pracuje *Pulumi* špecifickým spôsobom [15].

Vstupy dovoľujú použitie základných dátových typov daného jazyka(textový reťazec, číslo alebo boolean hodnota apod.), asynchrónne vypočítanú hodnotu alebo použitie *Výstupu* z iného *Resource*.

Výstupy sa správajú veľmi podobným spôsobom ako *promises* v jazyku *Javascript*. To znamená, že výsledná hodnota je vypočítaná asynchrónne a o svojom dokončení informuje volajúcu funkciu. Toto správanie je potrebné, keďže výsledky nasadzovania infraštruktúry sú dostupné až po kompletnom dokončení nasadenia daného zdroja [15].

Pulumi rovnakým spôsobom sleduje závislosti medzi nasadenými zdrojmi. Štruktúra popísaných komponentov a vzťahov medzi nimi je na obrázku 3.1. *Výstupy* teda môžu v *Pulumi* reprezentovať dva stavy:

- Eventuálnu výslednú hodnotu *Výstupu*
- Vzájomné dependencie medzi jednotlivými *Resources*



Obrázek 3.1 Štruktúra komponentov Pulumi
[15]

3.3 Kubernetes

Ide o open-source systém určený na nasadzovanie, škálovanie a správu kontajnerových aplikácií. Kubernetes automatizuje úlohy spojené so správou kontajnerov a obsahuje vstavané príkazy a nástroje na nasadzovanie aplikácií, zavádzanie zmien v aplikáciách,

škálovanie aplikácií podľa meniaceho sa zaťaženia, monitorovanie aplikácií a ďalšie funkcie, ktoré uľahčujú správu aplikácií [16, 17]. Základnými konceptami, ktorými sa riadi Kubernetes sú:

- Deklaratívna konfigurácia - pomocou konfiguračných súborov je deklarovaný požadovaný stav systému. Kubernetes následne vytvára a udržuje tento požadovaný stav [16].
- Automatizácia - nástroj je navrhnutý tak aby automatizoval čo najviac manuálnych procesov spojených s nasadzovaním a škálovaním aplikácií [16]. Vykonáva tak samostatne úlohy, ako nasadzovanie priebežných aktualizácií, zisťovanie existencie služieb, vyrovnávanie záťaže medzi inštanciami a ďalšie.

Architektúra *Kubernetes* sa skladá z niekoľkých kľúčových komponentov, z ktorých každý zohráva určitú úlohu pri správe aplikácií.

- **Pod** - základným stavebným prvkom systému Kubernetes je *pod*. *Pod* predstavuje jednu inštanciu bežiaceho procesu v clusteri a môže obsahovať jeden alebo viac kontajnerov. Kontajnery v rámci podov zdieľajú rovnaký sieťový adaptér, čo znamená, že je medzi nimi umožnená jednoduchá lokálna komunikácia [16, 17].
- **Node(Uzol)** - predstavuje jednotku, ktorá spúšťa a spravuj jednotlivé *pody*. V závislosti od typu *clustera* môže byť uzol vytvorený pomocou virtuálneho alebo fyzického hardvéru. Každý uzol je spravovaný hlavným(master) uzlom a obsahuje všetky potrebné funkcie a služby na spúšťanie *podov* [17].
- **Service(Služba)** - ide o logickú sadu *podov*, ktoré sú združené dokopy [16]. Služba definuje prístupové pravidlá, ktoré určujú, akým spôsobom je možné pristúpiť k jednotlivým *podom* z externej sieťe alebo iných služieb. Služby taktiež definujú tzv. *selector*, ktorým je ich možné v rámci Kubernetes clustera adresovať [16, 17].
- **Deployment(Nasadenie)** - pomocou tohto komponentu je definovaný požadvaný stav podov a replík v rámci služieb. Programátor pomocou konfiguračného súboru nastaví požadovaný stav systému a ovládač nasadení postupne mení stav systému do požadovaného stavu [16, 17].
- **ReplicaSet** - nastavuje požadovanú stabilnú sadu replík(inštancie rovnakého kontajnera), ktoré sú naraz spustené. Využívajú sa na nastavenie minimálneho a maximálneho počtu inštancií, ktoré poskytujú danú službu. Zaručí sa tak dostupnosť danej služby v prípade zlyhania jedného alebo viacerých bežiacich inštancií [16, 17].

- **Ingress** - objekt poskytovaný pomocou Kubernetes API, ktorý spravuje prichádzajúce spojenia z externej siete. Tieto spojenia následne smeruje k interným službám, ktoré poskytuje *cluster*. Poskytuje taktiež funkcionalitu ako je vyrovnávanie zaťaženia alebo SSL zabezpečenie [16, 17].
- **ConfigMaps** - určujú objekty, ktorými je možné oddeliť konfiguráciu aplikácie od obrazu kontajnera. Vytvára sa tak prenosná konfigurácia, ktorá nie je priamo vložená do vytváraného obrazu kontajnera, čím sa redukuje vzájomná previazanosť systému [17].

II. PRAKTICKÁ ČÁST

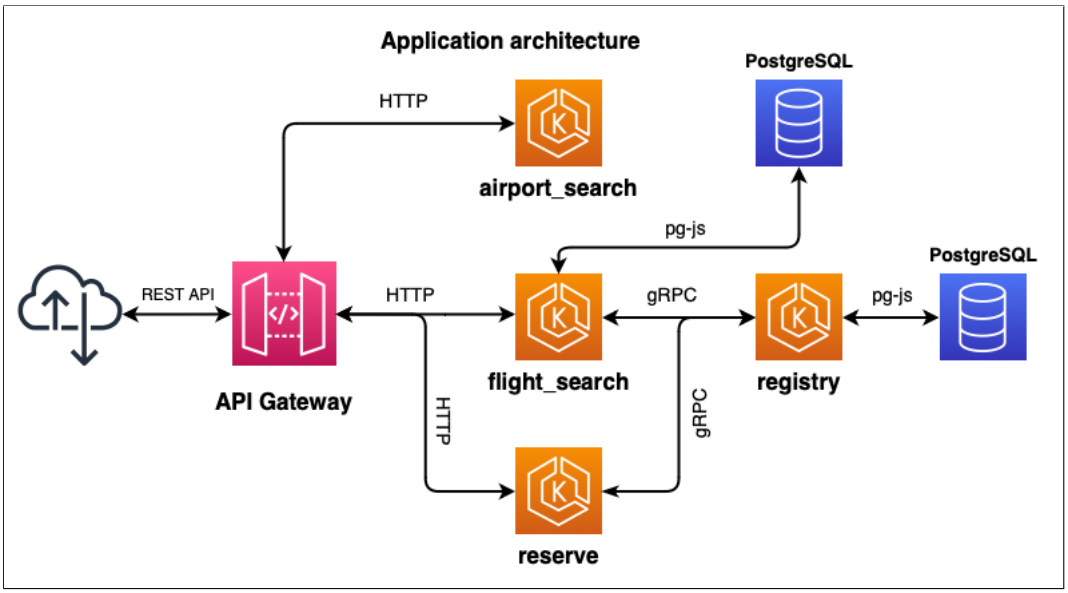
4 NÁVRH APLIKÁCIE

Účelom praktickej časti je vytvorenie aplikácie, ktorá vhodne demonštruje moderný vývoj aplikácií a využíva prístup *Infrastructure as Code* na nasadenie potrebných zdrojov. Ako architektonický vzor pre vytváranú aplikáciu bola zvolená architektúra mikroslužieb, keďže sa jedná o najpopulárnejší spôsob vývoja aplikácií v súčasnosti. Zároveň aplikácia používa služby a zdroje ktoré sú poskytované cloud providermi a priamo s nimi pracuje počas vývoja a procesu nasadenia. Inšpiráciou pre túto demonštračnú aplikáciu boli rôzne vyhľadávače leteniek, ktoré poskytujú možnosti vyhľadať, rezervovať a zakúpiť letenky na základe kritérií, ktoré zadá užívateľ.

Ide o vhodný systém, ktorý je ľahko rozdeliteľný na jednotlivé mikroslužby, ktoré vykonávajú unikátne činnosti vo svojej doméne. Zároveň je proces vyhľadávania leteniek výpočetne náročný proces, ktorý musí spracovať veľké množstvo dát a zvládnuť vysokú záťaž celého systému. Systém je navrhnutý ako skupina mikroslužieb, ktoré sú umiestnené v spoločnom *Kubernetes* clusteri v rámci ktorého komunikujú medzi sebou a spracúvajú prichádzajúce požiadavky. Taktiež komunikujú s externými službami v *AWS* cloude. Celý systém je prístupný ako verejné *REST API* , ktoré je vytvorené cez *API Gateway*, ktorá smeruje požiadavky na príslušné mikroslužby. Systém obsahuje nasledujúce mikroslužby:

- Služba na vyhľadávanie letísk - `airport_search`.
- Služba na vyhľadávanie leteniek - `flight_search`.
- Služba registra leteniek - `registry`.
- Služba na rezerváciu a zakúpenie leteniek - `reserve`.

Popis, účel, architektúra a implementácia jednotlivých mikroslužieb je popísaná v nasledujúcich častiach. Architektúra celej aplikácie je zobrazená na obrázku 4.1. Implementovaná aplikácia je nasadená pomocou *Pulumi SDK* a *Kubernetes*. Na záver je funkcionalita jednotlivých súčastí a nastavení otestovaná záťažovým testom.



Obrázek 4.1 Architektúra aplikácie

4.1 Služba na vyhľadávanie letísk

Z užívateľského hľadiska chceme užívateľom umožniť vyhľadávať letiská podľa názvu mesta z ktorého chcú hľadať dostupné letenky. V danom vyhľadávanom meste sa však môže nachádzať viacero letísk. Preto sú letiská v systémoch zvyčajne reprezentované ako *ICAO* alebo *IATA* kódy. Vytváraná mikro služba poskytuje API, ktoré na základe užívateľského vstupu do vyhľadávacieho poľa, vráti dostupné letiská, spolu s kódom daných letísk, ktoré môže následne použiť frontend aplikácie na vyhľadanie dostupných letov.

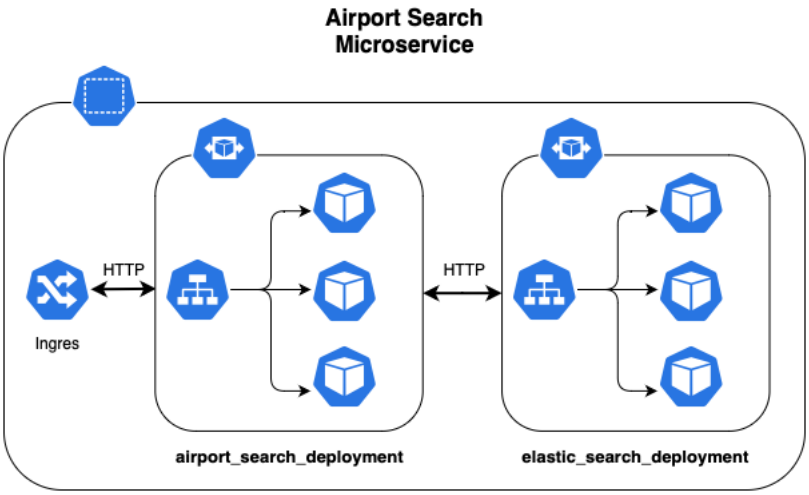
4.1.1 Architektúra mikroslužby na vyhľadávanie letísk

Mikroslužba je vytvorená ako *REST API*, vytvorené pomocou jazyka Typescript a prostredia Node.js. Samotný HTTP server je implementovaný pomocou knižnice *Express* Požiadavka na vyhľadávanie letiska v danom meste je očakávaná ako textový query parameter `text` vo forme `GET` požiadavky. Voliteľnými query parametrami sú pararametre `limit`, ktorý slúži na obmedzenie počtu výsledkov a `page`, ktorý umožňuje stránkovanie výsledkov. Príkladom požiadavky je:

```
http://localhost:3000/airports?text=Bratisl&limit=10
```

Užívatelia však nezadávajú do vyhľadávacieho poľa zvyčajne presné názvy, prípadne nepoznajú presný názov hľadaného mesta. Preto bolo nutné pridať funkcionalitu, ktorá bude poskytovať full-text search v dostupných letiskách. Toto bolo dosiahnuté pridaním služby Elasticsearch, ktorá bola naplnená dostupnými dátami o letiskách. Server pri obdržaní požiadavky kontaktuje službu elastic search, ktorá vyhľadá letiská a zo-

radí ich podľa presnosti zhody. Vrátené dáta server spracuje a vráti odpoveď na `GET` požiadavku.



Obrázek 4.2 Architektúra služby na vyhľadávanie letísk

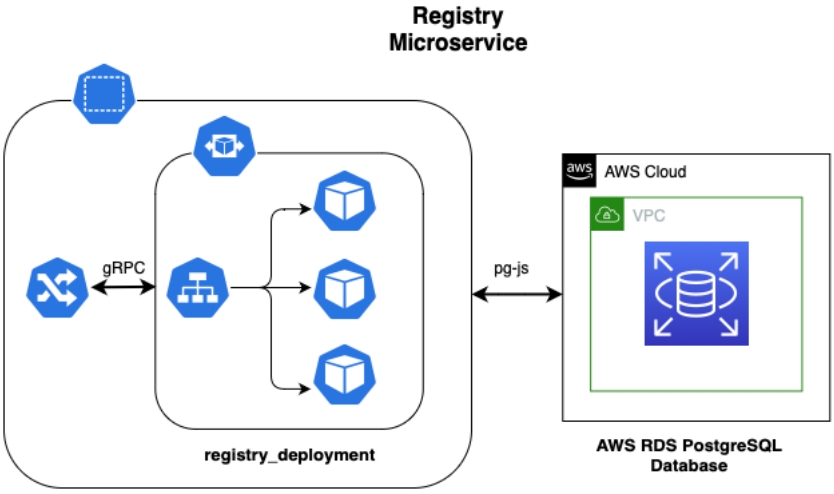
4.2 Služba registra leteniek

Z hľadiska vytvorenej architektúry aplikácie chceme aby boli informácie o letenkách uložené v centrálnom registri. Toto modeluje reálnu situáciu, kde letenky poskytované rôznymi vyhľadávačmi, sú v skutočnosti poskytované, vyhľadávané a kupované cez *API*, ktoré poskytujú niekoľko celosvetových firiem(napr. firma Amadeus alebo firma Sabre). Keďže v prípade našej demonštračnej aplikácie by množstvo zasielaných požiadaviek vysoko presiahlo dostupné limity od týchto spoločností, bolo nutné vytvoriť službu, ktorá emuluje tento centrálny register leteniek. Služba je koncipovaná ako interná aplikácia, ktorá poskytuje API, na ktoré sa môžu ostatné mikroslužby napojiť a získať informácie o dostupných letenkách, letenku zarezervovať (rezerváciou rozumieme úkon, ktorý nastaví letenku ako nedostupnú, pokiaľ nedôjde k vyplneniu potrebných údajov a zaplateniu, aby letenka nebola dostupná pre iných zákazníkov, počas toho ako zákazník platí a vypĺňa potrebné údaje.)

4.2.1 Architektúra mikroslužby registra leteniek

Mikroslužba je vytvorená ako gRPC API, vytvorené pomocou jazyka Typescript, prostredia Node.js. Samotný gRPC server je implementovaný pomocou knižnice *grpc-js*. Ako databáza na ukladanie dát o dostupných letenkách bola zvolená SQL databáza *PostgreSQL* a server komunikuje spolu s dostupnou databázou s pomocou knižnice *pg*. Databáza taktiež ukladá informácie o cestujúcich, ktorý si letenku kúpili. Server poskytuje funkcie(RPC volania), ktoré vykonajú zmeny v databáze a vrátia odpoveď na

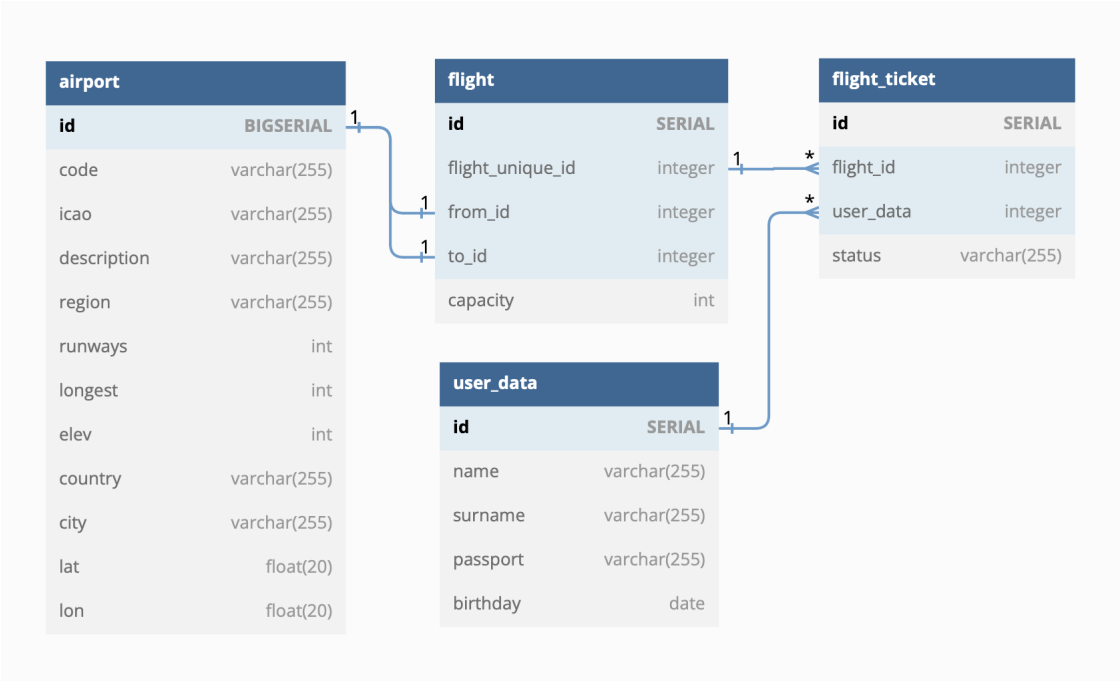
požiadavky. Samotná databáza je nasadená pomocou *Pulumi* v cloude *AWS*. Architektúra je ukázaná na obrázku 4.3.



Obrázek 4.3 Architektúra služby registra leteniek

4.2.2 Databáza mikroslužby registra leteniek

Ako už bolo zmienené, databáza je vytvorená pomocou *PostgreSQL* a sú do nej vložené niektoré z dát, ktoré budú vytvorené v časti 5.4. Návrh štruktúry databáze a vzťahov medzi tabuľkami je zobrazený na obrázku 4.4. Databáza obsahuje nasledujúce tabuľky a vzťahy:



Obrázek 4.4 Architektúra databázy služby registra leteniek

- Tabuľka `airport` obsahuje informácie o detailoch letísk, a nadväzuje *One-to-One* vzťahy s tabuľkou `flight`.
- Tabuľka `flight` obsahuje informácie o čísle letu, príletových a odletových destináciách a maximálny počte pasažierov, ktorý môže tento let previesť. Táto hodnota je náhodne generovaná počas vkladania letov do tabuľky. Nadväzuje *One-to-One* vzťahy s tabuľkou `airport` a *One-to-Many* vzťahy s tabuľkou `flight_ticket`.
- Tabuľka `flight_ticket` slúži ako tabuľka na vytvorenie *Many-to-Many* vzťahu medzi tabuľkami `flight` a `user_data`. Vytvára tak informáciu o tom, ktorý užívateľ si zakúpil akú letenku. Obsahuje taktiež informáciu o stave danej letenky - teda či je daná letenka už zakúpená alebo ešte len rezervovaná.
- Tabuľka `user_data` obsahuje informácie o pasažierovi, ktorý si zakúpil letenku a ukladá jeho osobné informácie. Nadväzuje *One-to-Many* vzťah s tabuľkou `flight_ticket`.

4.3 Služba na vyhľadávanie leteniek

Služba umožní vyhľadávanie dostupných leteniek, podľa kritérií, ktoré zadá užívateľ. Podporovanými kritériami je letisko odletu a príletu, dátum odletu, počet pasažierov a voľba, či daná trasa môže obsahovať prestupy.

4.3.1 Architektúra služby na vyhľadávanie leteniek

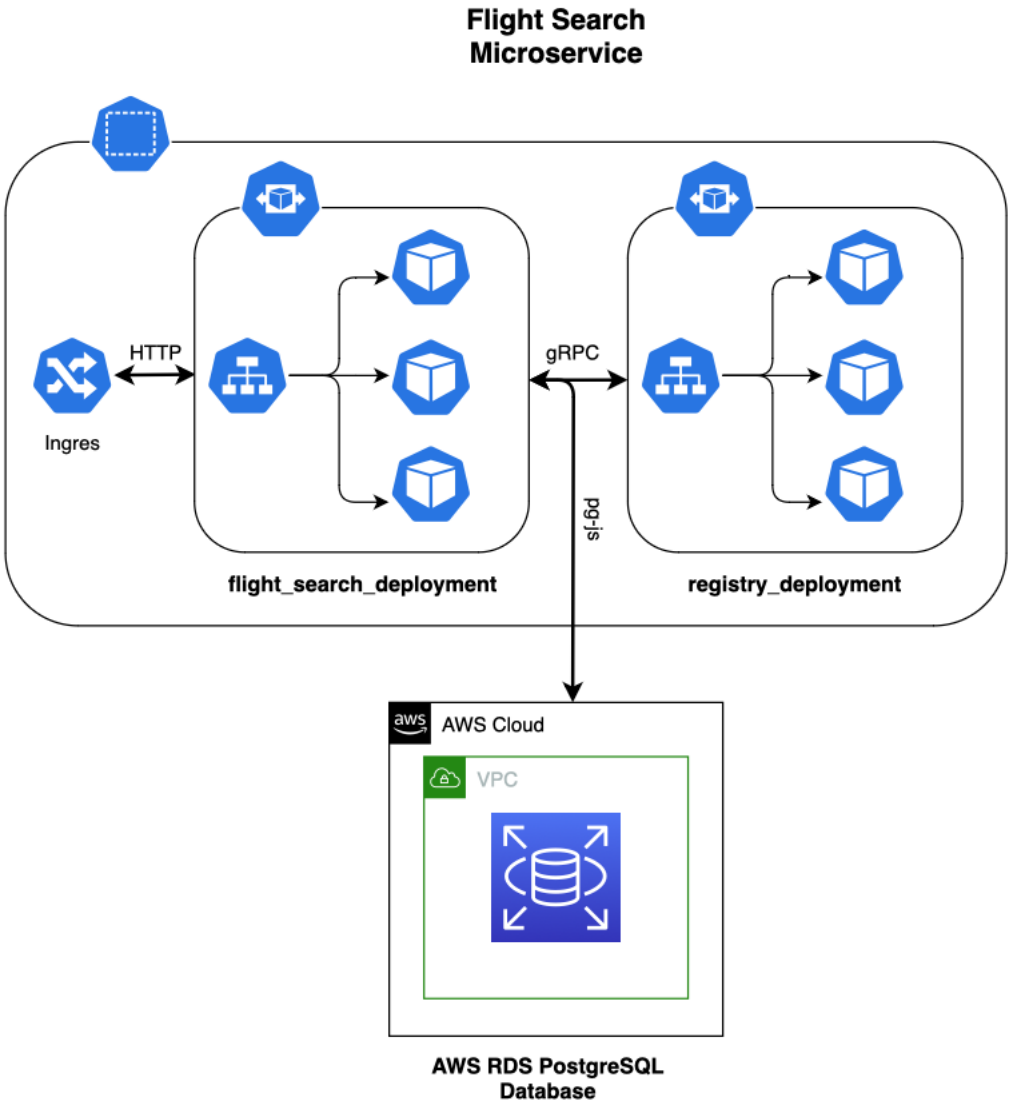
Mikroslužba bola vytvorená ako *REST API* pomocou jazyka Typescript a prostredia Node.js. HTTP server je implementovaný prostredníctvom knižnice *Express* a služba poskytuje `GET` *endpoint* `searchFlight`. Požiadavka na vyhľadanie letenky prijíma nasledovné *query* parametre:

- `from` - očakáva *IATA* kód odletového letiska.
- `to` - očakáva *IATA* kód destinácie.
- `departure_date` - dátum odletu vo formáte (*YYYY-MM-DD*).
- `passanger_count` - počet pasažierov.
- `hasStops` - voľba, či má služba vyhľadávať len priame lety.

Príkladom požiadavky je:

```
http://localhost:3000/searchFlight?from=LHR&
to=JFK&
passenger_count=10&
hasStops=true&departure_date=2023-05-04
```

Služba obsahuje vlastnú *PostgreSQL* databázu v ktorej sa nachádzajú detaily dostupných leteniek. Po spustení mikroslužby nadviaže služba spojenie s vlastnou *PostgreSQL* databázou pomocou knižnice *pg*. Server taktiež nadviaže priame *gRPC* spojenie s mikroslužbou registra leteniek. Po obdržaní požiadavky vyhľadá služba vo vlastnej databáze dostupné letenky. Zároveň kontaktuje mikroslužbu na registra leteniek a zistí počet dostupných leteniek pre možnosti, ktoré boli nájdené v databáze. Následne sú výsledky nájdené v databáze vyfiltrované na základe informáciách o dostupnosti, ktoré boli získané ako odpoveď z mikroslužby registra leteniek. Počas vývoja sú použité rovnaké dáta aké boli vygenerované v časti 5.4. Vyfiltrované výsledky sú vrátené ako odpoveď na HTTP požiadavku. Táto architektúra je zobrazená na obrázku 4.5.



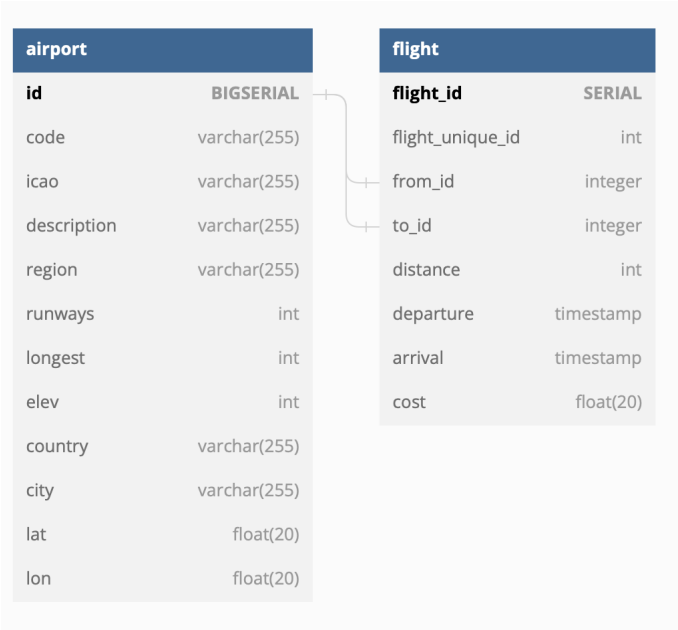
Obrázek 4.5 Architektúra služby na vyhľadáanie leteniek

4.3.2 Databáza mikroslužby na vyhľadávanie leteniek

Databáza je pri štarte naplnená údajmi, ktoré boli vytvorené v časti 5.4. Dôvodom vytvorenia tejto databáze je emulovanie reálnej situácie, kde mikroslužba udržuje vlastnú kópiu, prípadne rozšírený dataset dát, aby bolo urýchlené vyhľadávanie. Nové dáta by boli v produkčnom prostredí pravidelne synchronizované z centrálneho registra. Databáza obsahuje tabuľky:

- Tabuľka `airport` - Obsahuje rovnaké dáta o letiskách ako ostatné mikroslužby.
- Tabuľka `flight` - Obsahuje detailné informácie o letoch. Vrátane časov odletov, príletov a cien leteniek.

Štruktúra databáze je zobrazená na obrázku 4.6.



Obrázek 4.6 Štruktúra databáze mikroslužby na vyhľadávanie leteniek

4.4 Služba na rezerváciu a kúpu leteniek

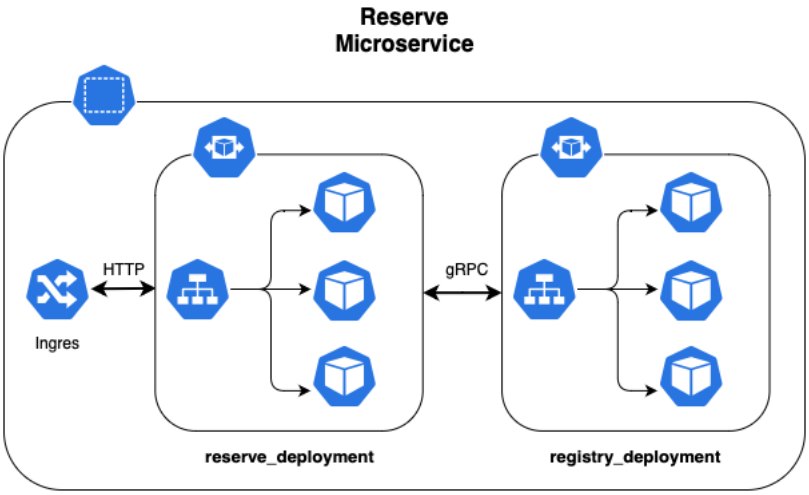
Služba umožní užívateľom zakúpiť letenky, ktoré boli vyhľadané na základe zvolených parametrov. Služba taktiež umožňuje zaslať informáciu z užívateľského prostredia, že užívateľ aktuálne vypĺňuje svoje osobné informácie a jeho letenka by mala byť v systéme označená ako rezervovaná. Server slúži ako *REST API*, cez ktoré užívateľské rozhranie nepriamo kontaktuje mikroslužbu registra leteniek a jej *gRPC* metódy.

4.4.1 Architektúra služby

Mikroslužba bola vytvorená ako *REST API* pomocou jazyka Typescript a prostredia Node.js. *HTTP server* poskytuje dva `POST` *endpointy*:

- `reserve` - Slúži na rezerváciu letenky. Ako telo požiadavky je zasielané `id` letu, na ktorom sa má zarezervovať letenka.
- `buy` - Slúži na zakúpenie letenky. Ako telo požiadavky sú zasielané osobné údaje cestujúceho a `id` rezervovanej letenky, ktoré bolo získané pri rezervácií letenky.

Služba po spustení nadviaže *gRPC* spojenie s mikroslužbou registra leteniek a začne prijímať požiadavky na popísaných *endpointoch*. Po prijatí požiadavky, server skontroluje, či požiadavka obsahuje všetky potrebné dáta a cez klienta *grpc-js* kontaktuje mikroslužbu registra leteniek v ktorej zavolá požadovanú funkciu. Server po obdržaní odpovede preposiela získané dáta ako odpoveď na *HTTP* požiadavku. Popisovaná architektúra je zobrazená na obrázku 4.7.



Obrázek 4.7 Architektúra služby na rezerváciu a zákúpenie leteniek

5 TUTORIÁL IMPLEMENTÁCIE APLIKÁCIE

Nasledujúca časť popisuje tutoriál, ktorý prevedie čitateľa návodom ako implementovať navrhnutú aplikáciu v predchádzajúcej časti pomocou prístupu *Infrastructure as Code* a nástrojov *Pulumi*, *Docker*, *Kubernetes* a *LocalStack*.

5.1 Vytvorenie projektu

Prvým krokom je vytvorenie projektu a spôsobu akým budú jednotlivé mikroslužby organizované. Tento proces pozostáva z nasledujúcich častí:

5.1.1 Organizácia projektu

Keďže návrh aplikácie pozostáva z viacerých mikro služieb, ktoré sú na sebe závislé a zdieľajú medzi sebou typové definície a niektoré súbory, bolo potrebné zvoliť vhodný prístup, ktorým bude repozitár, v ktorom sa nachádza zdrojový kód. Preto bola zvolená stratégia monolitického repozitára (*monorepo*). Ide o stratégiu, kde je zdrojový kód viacerých projektov alebo mikro služieb uložený v jednom spoločnom repozitári. To je v protiklade od typického projektu, kde je každá časť, knižnica alebo aplikácia uložená vo svojom vlastnom repozitári. Výhodami tohto prístupu sú:

- Zjednodušená správa závislostí - všetky závislosti a knižnice sú uložené na jednom mieste, čo uľahčuje ich správu a aktualizáciu. Taktiež tým zabránime konfliktom medzi závislosťami.
- Prehľadnosť kódu - je uľahčené rozpoznanie závislostí a vytvorenie znovu-použiteľných komponentov a riešení.
- Zjednodušená refaktorizácia - vykonávanie zmien v kóde aplikácie, je taktiež jednoduchšie, keďže akákoľvek zmena, ktorá ovplyvňuje iné aplikácie alebo súčasti, je automaticky zachytená prekladačom, v prípade správneho nastavenia.
- Konzistentné prostredie - testovanie a následné zostavenie projektu je zdieľané medzi všetkými časťami čo redukuje chyby, ktoré vznikajú v dôsledku rôznych konfigurácií.

Tento prístup má samozrejme aj svoje nevýhody ako je napríklad zložitosť pri práci s rozsiahlymi repozitármi alebo problémy pri škálovaní. Pri vytváraní tohto projektu však ide o vhodné riešenie, ktoré slúži na demonštráciu potrebných princípov. Projekt bol vytvorený pomocou správcu balíkov `yarn` a príkazu `yarn init`. Po inicializácií bol upravený vytvorený `package.json` súbor aby obsahoval nasledujúce nastavenia: Dosiahneme tak, že repozitár sa bude správať ako monorepo. Ďalšou výhodou je, že

```
{
 "private": true,
 "workspaces":["packages/*"]
}
```

Obrázek 5.1 Upravený package.json pre monorepo nastavenie

akékoľvek použité stiahnuté knižnice sú uložené v globálnej *node_modules* zložke pre celý repozitár, takže nasledujúce inštalácie a pridanie závislosti, je pre ďalšie projekty veľmi rýchle, keďže inštalácia vyžaduje len vytvorenie symbolického odkazu. Následne v každej založenej službe inicializujeme repozitár pomocou `yarn init` a spustíme príkaz `tsc init`, ktorý inicializuje Typescript konfiguráciu pre jednotlivé služby. Bola vytvorená štruktúra zložiek, ktorá určuje nasledujúcu hierarchiu zložiek pre vytvorený repozitár.

```
.
└── packages/
    ├── node_modules
    ├── airport_search/
    │   ├── pulumi/
    │   │   ├── Pulumi.yaml
    │   │   ├── index.ts
    │   │   └── ...
    │   ├── src/
    │   │   ├── Dockerfile
    │   │   ├── index.ts
    │   │   └── ...
    │   ├── package.json
    │   └── yarn.lock
    ├── flight_search/
    │   ├── pulumi/
    │   ├── src/
    │   └── ...
    ├── package.json
    └── yarn.lock
```

Obrázek 5.2 Ukážka štruktúry zložiek v repozitári

5.1.2 Založenie mikroslužby

Každá z opísaných mikroslužieb v nasledujúcich kapitolách dodržuje jednotnú štruktúru zložiek a bola inicializovaná rovnakým procesom. Mikroslužby boli vytvorené spustením už spomenutých príkazov `yarn init` a `tsc init`, ktoré vytvorili súbory `package.json` a `tsconfig.json`.

package.json Súbor `package.json` bol upravený spôsobom na obrázku 5.3. Podstatnými časťami tohto súboru sú:

```
{
  "name": "registry",
  ...
  "dependencies": {
    "@grpc/grpc-js": "^1.8.14",
    "pg": "^8.10.0",
        ...
  },
  "scripts": {
    "dev": "ts-node src/index.ts",
    "build": "tsc"
  },
  "devDependencies": {
        "typescript": "^5.0.4"
        "ts-node": "^10.9.1",
        ...
  }
}
```

Obrázek 5.3 Jednotný package.json súbor

- Časť `"dependencies"` - sú do nej vkladané závislosti, ktoré každá mikroslužba potrebuje na svoje fungovanie. Nové dependencie sú pridávané príkazom `yarn add`.
- Časť `"scripts"`, do ktorej boli pridané nasledujúce časti:
  - `"dev"` - ide o príkaz, ktorý sa spúšťa počas vývoja aplikácie. Využíva závislosť *ts-node*, ktorá automaticky kompiluje *Typescript* súbory do *Javascriptu*.
  - `"build"` - príkaz, ktorý spúšťa vybudovanie a skompilovanie *Typescript* súborov do *Javascriptu*.
- Časť `"devDependencies"` - sú do nej vkladané závislosti, ktoré potrebujeme len počas vývoja alebo budovania aplikácie. Za behu aplikácie nie sú tieto dependencie potrebné. Môžeme tak redukovať výslednú veľkosť súborov a aplikácie.

tsconfig.json Do automaticky vygenerovaného súboru `tsconfig.json` boli pridané nastavenia ukázané na obrázku 5.4. Pridané časti sú:

- `"outDir"` - Určuje zložku do ktorej budú ukladané súbory po úspešnom zostavení mikroslužby.
- `"rootDir"` - Nastavuje koreňový zdrojový adresár v ktorom má *Typescript* vyhľadávať zdrojové súbory.

```
{
  "compilerOptions": {
    "outDir": "./build",
    "rootDir": "./src",
    "strict": true,
  },
  "include": ["src/**/*.ts"]
}
```

Obrázek 5.4 Jednotný tsconfig.json súbor

- `"strict"` - Zabezpečuje, že aplikácia nebude úspešne zostavená bez vyriešenia všetkých typových chýb, ktoré objaví *Typescript* kompilátor.
- `"include"` - Určuje všetky adresáre a podadresáre v ktorých majú byť hľadané súbory.

Pulumi V zdrojovom kóde každej mikroslužby je vytvorená zložka `pulumi` v ktorej je spustený príkaz `pulumi new`. Príkaz vytvorí v adresári potrebný súbor `Pulumi.yaml`, ktorý obsahuje základné definície danej mikroslužby a súbor `Pulumi.dev.yaml`, ktorý obsahuje nastavenia daného `Stacku` danej služby. V tomto adresári sú následne ukladané všetky súbory týkajúce sa nasadenia konkrétnej služby pomocou *Pulumi*.

5.2 Inštalácia potrebných knižníc

Pre úspešnú implementáciu aplikácie musíme nainštalovať niekoľko potrebných knižníc a aplikácií, ktoré sú používané pre všetky implementované služby.

Docker Docker klient umožní vytvorenie vybudovanie kontajnerov na základe preddefinovanej konfigurácie, ktoré môžu byť nasadené do cloud prostredia alebo do lokálneho prostredia počas vývoja [20].

Kubernetes Kubernetes slúži ako orchestrátor kontajnerov, ktorý automatizuje nasadenie, škálovanie a manažment kontajnerov, ktoré sú automaticky vytvárané počas vykonávania zmien v zdrojovom kóde. Kubernetes cluster môže byť nasadený lokálne alebo v cloud prostredí od rôznych poskytovateľov, ako napríklad *AWS* vo forme *Elastic Container Service for Kubernetes*. Počas vývoja aplikácie bola využitá funkcionalita poskytovaná od Docker Desktop, ktorá umožnila vytvorenie lokálneho Kubernetes clustera.

Pulumi Pulumi je platforma, ktorou bola počas vývoja definovaná infraštruktúra projektu a vytvorené nasadenie a manažment potrebných zdrojov pre každú z mikro služieb.

Emulátor cloud prostredia Navrhnutá aplikácia vyžaduje prostriedky poskytované rôznymi cloud providermi. Keďže nasadenie veľkého množstva týchto prostriedkov(API Gateway, RDS databáze apod.) by bolo finančne náročné počas vývoja, môžeme využiť aplikáciu *LocalStack*, ktorá dokáže emulovať AWS cloud prostredie na lokálnom počítači. Aplikácia využíva bežiaci Docker kontajner, ktorý zachytáva API volania AWS služieb a emuluje väčšinu služieb, ktoré AWS ponúka a navrhnutá aplikácia používa [21]. Nakoniec môžeme nastaviť *Pulumi* pomocou konfiguračného súboru, aby zasielalo volania určené pre *AWS API* do LocalStack kontajneru.

5.3 Implementácia služby na vyhľadávanie letísk

Mikroslužba využíva na svoje fungovanie knižnice *Express* a *elasticsearch* javascript klient knižnicu na komunikáciu s *Elasticsearch* serverom. Pri štarte služby je nadviazané spojenie so elasticsearch serverom a server je naplnený údajmi o existujúcich letiskách [18]. Dáta sú vo formáte JSON a obsahujú nasledovné dáta:

```
"LKLT": {
"icao": "LKLT",
"iata": "",
"name": "Letnany Airport",
"city": "Letnany",
"state": "Praha",
"country": "CZ",
"elevation": 909,
"lat": 50.131401062,
"lon": 14.5256004333,
"tz": "Europe\/Prague"
},
```

Obrázek 5.5 Vzor dostupných dát letísk
[18]

Dostupné dáta boli vložené na server pomocou nasledujúcej funkcie:

```
export const createIndex = async () => {
  try {
    await esclient.indices.create({
      index,
      mappings: {
        properties: {
          icao: { type: "text" },
          author: { type: "text" },
          iata: { type: "text" },
          name: { type: "text" },
          city: { type: "text" },
          state: { type: "text" },
          country: { type: "text" },
          elevation: { type: "integer" },
          lat: { type: "float" },
          lon: { type: "float" },
          tz: { type: "text" },
        },
      },
    });
    const operations = getAirportArray().flatMap((doc) => [
      { index: { _index: index } },
      doc,
    ]);
    const bulkResponse = await esclient.bulk({ refresh: true, operations });
    console.log(`Created index ${index}`);
  } catch (err) {
    console.error(`An error occurred while creating the index ${index}:`);
  }
};
```

Obrázek 5.6 Funkcia na vloženie dát do Elasticsearch serveru

Ako je vidieť v kóde vyššie funkcia najprv vytvorí mapovanie typov vlastností objektu na typy, ktoré podporuje Elasticsearch. Následne je vytvorené pole operácií a tieto operácie sú hromadne pridané do Elasticsearch aplikácie pomocou funkcie `esclient.bulk({ refresh: true, operations });` Express knižnica inicializuje poskytované *REST API* a čaká na prijatie *HTTP* požiadavky so správnymi parametrami na adrese `/airports`. Po prijatí požiadavky HTTP server kontaktuje pomocou klientskej knižnice bežiacu inštanciu elasticsearch servera, ktorý sa pokúsi vyhľadať zhody s vloženým textom a vráti *HTTP* serveru odpoveď s dostupnými dátami, ktoré server pretransformuje a vráti odpoveď na prijatú požiadavku, spolu s nájdenými zhodami. Funkcia na vyhľadanie zhôd s vloženým textom má tvar zobrazený nižšie na obrázku 5.7. Najpodstatnejšou časťou funkcie je objekt `query` a jej časť `match`, ktorá

nastavuje parametre, podľa ktorých prebieha vyhľadávanie zhôd v dostupných dátach, ktoré boli indexované. Nakoniec sú dáta pretransformované, do správneho formátu, ktorý očakáva *HTTP* server.

```
export async function getAirports({
  page,
  limit,
  text,
}: GetAirportsRequestQuery) {
  const result = await esclient.search<Airport>({
    index: index,
    from: page || 0,
    size: limit || 10,
    query: {
      match: {
        city: {
          query: text,
          operator: "AND",
          fuzziness: "auto",
        },
      },
    },
  });

  const results = result.hits.total;
  const values = result.hits.hits.map((hit) => {
    return {
      id: hit._id,
      name: hit._source?.name,
      icao: hit._source?.icao,
      city: hit._source?.city,
      country: hit._source?.country,
      score: hit._score,
    };
  });
}
```

Obrázek 5.7 Funkcia na vyhľadanie letísk v Elasticsearch

5.3.1 Implementácia Infrastructure as Code

Aby sme mohli použiť *Pulumi* na nasadenie infraštruktúry potrebujeme najprv vytvoriť nový *Pulumi* projekt pre mikroslužbu na hľadanie letísk. Nový projekt vytvoríme pomocou príkazu `pulumi new typescript` a postupujeme podľa pokynov v príkazovom

riadku.

Dockerfile - Následne vytvoríme `Dockerfile` konfiguráciu podľa ktorej sa bude vytvárať kontajner mikroslužby, ktorý je následne nasadený do Kubernetes clusteru. Obsah `Dockerfile` je zobrazený na 5.8.

```
FROM node:20-alpine

WORKDIR usr/src/app
COPY package.json ./
COPY tsconfig.json ./
RUN yarn
CMD "yarn build"
COPY ../build/. ./build

EXPOSE 3000
EXPOSE 9200

CMD ["node","build/index.js"]
```

Obrázek 5.8 Dockerfile definícia kontajneru mikroslužby

Obsah `Dockefile` vykonáva nasledujúce funkcie:

- `FROM` príkaz - nastavuje obraz, ktorý používame ako základ vybudovaného kontajneru. V tomto prípade bol zvolený obraz `node:20-alpine`, ktorý je optimalizovaný pre produkčné prostredie a má menšiu veľkosť ako iné dostupné obrazy.
- `WORKDIR` príkaz - nastaví zložku do ktorej budú ukladané súbory.
- `COPY` príkaz - skopíruje zvolené súbory do zvolenej destinácie. V tomto prípade kopírujeme konfiguračné súbory `package.json` a `tsconfig.json`, ktoré potrebuje aby sme mohli nainštalovať potrebné dependencie.
- `yarn` a `CMD "yarn build"` príkaz - nainštaluje všetky potrebné súčasti definované v `package.json` a skompiluje *Typescript* súbory na *Javascript* súbory na základe konfigurácie v `tsconfig.json`.
- `EXPOSE` príkaz - zadefinuje porty, cez ktoré kontajner komunikuje a dovoľuje pripojenie.
- `CMD ["node","build/index.js"]` príkaz - zadefinuje príkaz, ktorý po vybudovaní kontajnera spustí node prostredie a spustí mikroslužbu.

Vybudovanie kontajneru pomocou IaC - Kontajner vybudujeme pomocou knižnice *Pulumi*, ktorá ponúka modul na prácu a interagovanie s Docker kontajnermi. Funkcia na vybudovanie kontajneru je zobrazená na obrázku 5.9. Funkcia prijíma ako argu-

```
export const buildAirportSearch = (ecrRegistry: aws.ecr.Repository) => {
  const imageNameInternal = "airport-search-image";
  const authToken = aws.ecr.getAuthorizationTokenOutput({
    registryId: ecrRegistry.registryId,
  });
  const airportSearchImage = new docker.Image(imageNameInternal, {
    build: {
      args: {
        BUILDKIT_INLINE_CACHE: "1",
      },
      platform: "linux/arm64",
      cacheFrom: {
        images: [pulumi.interpolate`${ecrRegistry.repositoryUrl}:latest`],
      },
      context: "../airport_search",
      dockerfile: "../airport_search/src/Dockerfile",
    },
    imageName: pulumi.interpolate`${ecrRegistry.repositoryUrl}:latest`,
    registry: {
      username: pulumi.secret(authToken.userName),
      password: pulumi.secret(authToken.password),
      server: ecrRegistry.repositoryUrl,
    },
  });
  return airportSearchImage;
};
```

Obrázek 5.9 Funkcia na automatické vybudovanie kontajneru

ment nasadený `ECR Register` v cloud službe spoločnosti *AWS*. Táto služba poskytuje automatizované ukladanie, manažment a nasadzovanie kontajnerov v AWS cloud prostredí. Ostatné AWS služby tak môžu jednoducho čítať dostupné kontajnery a získavať najnovšie zmeny. Funkcia ďalej nastavuje potrebné parametre ako `platform` a `cacheFrom`, ktorý určuje, že budovanie kontajneru nie je nutné počas ďalších nasadení mikroslužby, pokiaľ nedošlo ku zmenám v kóde. Následne sú nastavené správne cesty zdrojvému kódu služby a konfiguračnému súboru `Dockerfile`. Nakoniec sú nastavené prístupové údaje k registru kontajnerov. Skript niekoľkokrát používa funkciu `pulumi.interpolate`<Pulumi Output>``, ktorá vkladá do textových reťazcov výstupové hodnoty hodnoty iných *Pulumi* objektov. keďže objekty, ktoré poskytuje *Pulumi*

sú zdroje a infraštruktúra poskytované cloud službami, ich stav a výstupné hodnoty nie sú dopredu známe. Tieto hodnoty budú dostupné až v čase keď je skript spustený a cloud zdroje boli úspešne rezervované a spustené. Preto je podstatné použiť funkciu `pulumi.interpolate`, ktorá signalizuje *Pulumi*, že daný reťazec nemôže byť prečítaný a použitý až pokiaľ relevantné služby nebudú pripravené. Ostatné objekty vytvorené priamo pomocou knižnice *Pulumi* sú automaticky spúšťané v správnom poradí na základe interných závislostí.

Nasadenie mikroslužby pomocou IaC Nasadenie mikroslužby prebieha taktiež použitím knižnice *Pulumi*. Prvým krokom úspešného nasadenia tejto mikroslužby je vybudovanie kontajnera použitím funkcie, ktorú sme vytvorili vyššie 5.9 a vytvorením `ECR` registra, ktorý tejto funkcií predáme. Následne vytvoríme nasadenia pre server na vyhľadávanie letísk a Elasticsearch server. Nasadenie servera na vyhľadanie letísk je dosiahnuté kódom na obrázku 5.10. Podstatnými časťami tejto konfigurácie sú:

- `selector` - Nastavenie poľa `selector` na hodnotu

  ```
  { matchLabels: { app: "airport-search-app" } }
  ```

  je podstatné pretože na základe tejto hodnoty prebieha interné vyhľadávanie v sieti vo vnútri Kubernetes clustera.

- `replicas` - Nastavuje minimálny počet replík, ktoré budú pri štarte Kubernetes clustera spustené.

- `Pole containers` - V tomto poli nastavujeme hodnoty pre jednotlivé kontajnery:

  - `image` - Určuje, ktorý vybudovaný obraz kontajnera bude použitý pre spustenie. V tomto prípade je použitý obraz vybudovaného kontajnera pre vyhľadávanie letísk.
  - `ports` - Definuje a otvára porty, na ktoré kontajner spolieha.
  - `env` - Nastavuje premenné v prostredí kontajnera. V tomto prípade je nastavená interná adresa na ktorej má služba vyhľadávania letísk komunikovať splou s Elasticsearch serverom.
  - `resources` - Nastavuje hodnoty, ktoré môže nasadenie minimálne vyžadovať alebo maximálne využívať. V tomto prípade je nasadenie umelo limitované na nízku hodnotu, aby sme v ďalšej časti úspešne otestovať automatické škálovanie.

- `dependsOn` - Pole obsahujúce *Pulumi* objekty, ktoré určuje závislosti na ktoré musí nasadenie počkať pred svojim vlastným spustením.

```
const airport_search_deployment = new k8s.apps.v1.Deployment(
    "airport-search-deployment",
    {
      metadata: {
        name: "airport-search-app",
      },
      spec: {
        selector: { matchLabels: { app: "airport-search-app" } },
        replicas: 1,
        template: {
          metadata: { labels: { app: "airport-search-app" } },
          spec: {
            containers: [
              {
                name: "airport-search-app",
                image: airport_search_image.imageName,
                ports: [{ containerPort: 3000 }, { containerPort: 9200 }],
                imagePullPolicy: "Always",
                env: [
                  {
                    name: "ELASTIC_HOST",
                    value: "https://elastic-search-app:1234",
                  },
                ],
                resources: {
                  limits: {
                    cpu: "40m",
                  },
                },
              },
            ],
          },
        },
      },
    },
    { dependsOn: [airport_search_image] }
  );
```

Obrázek 5.10 Definícia nasadenia mikroslužby na vyhľadávanie letísk

Nasadenie Elasticsearch servera vyžaduje podobnú konfiguráciu ako na obrázku 5.9, pričom bol zmenený selector na hodnotu `"elastic-search-app"` a nastavenia kontajneru boli zmenené do podoby na obrázku 5.11. Obraz kontajnera je automatický stiahnutý z oficiálneho zdroja, ktorý poskytuje firma Docker. Taktiež boli nastavené hod-

noty, ktoré prostredia tohto kontajnera vyžaduje na základe dostupnej oficiálnej dokumentácie. V neposlednom rade bola zredukovaná pamäť nastavením `"ES_JAVA_OPTS"`, ktorá je využívaná java prostredím, ktoré kontajner interne využíva, aby bola znížená záťaž na lokálny systém počas testovania.

```
containers: [
        {
        name: "elastic-search-app",
        image: "elasticsearch:8.7.1",
        ports: [{ containerPort: 9200 }],
        env: [
                {
                name: "discovery.type",
                value: "single-node",
                },
                { name: "ES_JAVA_OPTS", value: "-Xms1024m -Xmx1024m" },
        ],
        },
]
```

Obrázek 5.11 Časť definície nasadenia Elasticsearch

Vytvorenie služieb Aby mohli nasadené služby komunikovať medzi sebou a externými sieťami potrebujeme pomocou *Pulumi* vytvoriť Kubernetes služby. Funkcia použité na vytvorenie služby je zobrazená na obrázku 5.12. V tomto objekte nastavujeme:

- `selector` - Určuje, ktoré nasadenie je pripojené k danej službe.
- `ports` - Objekt obsahuje tieto vlastnosti:
  - `port` - Hodnota externého portu, na ktorom bude služba prijímať požiadavky.
  - `targetPort` - Hodnota interného portu, na ktorý budú externé požiadavky presmerované.
- `type` - Určuje ako sa bude služba správať pri vytváraní siete.
  - `"LoadBalancer"` - Služba bude dostupná v externej sieti, tj. mimo Kubernetes clustera pod jednotnou adresu a prichádzajúce požiadavky budú rozdelené medzi existujúce *Kubernetes pody*.

– `"ClusterIP"` - Služba bude dostupná pod jednotnou adresou len v internej sieti Kubernetes clustera a prichádzajúce budú rozdelené medzi existujúce *Kubernetes pody*.

```
new k8s.core.v1.Service(
    "airport-search-service",
    {
      metadata: { name: "airport-search-app" },
      spec: {
        selector: airport_search_deployment.spec.template.metadata.labels,
        ports: [{ port: 80, targetPort: 3000 }],
        type: "LoadBalancer",
      },
    },
    { dependsOn: [airport_search_deployment] }
  );
```

Obrázek 5.12 Definícia služby hľadania letísk

Nastavenie automatického škálovania Automatické škálovanie umožňuje prispôsobenie počtu dostupných inštancií na základe zvolenej metriky, ktorú budeme sledovať. Toto zaťaženie systému je následne rovnomerne rozdelené medzi existujúce a novo vytvorené pody v Kubernetes clusteri. Túto funkcionalitu poskytuje Kubernetes prostredie a nastavenie tohto škálovania sme dosiahli pomocou knižnice *Pulumi* a nasledovného skriptu, ktorý je zobrazený na obrázku 5.13. Pri konfigurácií horizontálneho škálovania sme nastavovali objekt `spec` na nasledujúce hodnoty:

- `scaleTargetRef` - V objekte sme nastavili napojenie na škálovania na existujúce nasadenie servera pomocou mena nasadenia a zvoli správnu verziu Kubernetes API.
- `minReplicas` - Nastavuje minimálny počet replík(*Kuberenetes podov*), ktoré majú byť spustené.
- `maxReplicas` - Nastavuje maximálny počet replík(*Kuberenetes podov*), ktoré môžu byť naraz spustené.
- `targetCPUUtilizationPercentage` - Nastavuje optimálne vyťaženie zdrojov CPU podov, pri ktorého prekročení dochádza k vytvoreniu a spustení ďalších inštancií v bežiacom clusteri, aby došlo k zníženiu celkového zaťaženia.

```
new k8s.autoscaling.v1.HorizontalPodAutoscaler(
        "airport-search-deployment-scale",
        {
        spec: {
                scaleTargetRef: {
                kind: "Deployment",
                name: airport_search_deployment.metadata.name,
                apiVersion: "apps/v1",
                },
                minReplicas: 1,
                maxReplicas: 5,
                targetCPUUtilizationPercentage: 40,
        },
        }
);
```

Obrázek 5.13 Definícia automatického horizontálneho škálovania

5.4 Získanie, úprava a generovanie dát letov

Na vytvorenie služieb, ktoré poskytuje a pracujú s dostupnými letenkami, potrebujeme získať relevantné dáta o reálnych letoch, ich destináciách, dĺžkach letov, prípadne ich dátumy odletov a príletov alebo ich ceny. Takýto zdroj dát však nie je verejne dostupný, prípadne vyžaduje špecifické a drahé licencie určené na komerčné použitie, čo v tomto prípade nebolo vhodné. Preto bolo rozhodnuté tieto dáta vytvoriť a vygenerovať z čiastočných verejných dát. Z verejnej databázy [19] bol stiahnutý súbor *.csv* súbor, ktorý obsahuje zoznam všetkých reálnych letových trás, ktoré existovali v roku 2022. Tento dataset obsahuje kódy odletového a príletového letiska a dĺžku danej trasy. Časť obsahu tohto súboru je zobrazený v tabuľke 5.1. Celkovo obsahuje dataset cez 50 tisíc letových trás. Dataset taktiež zahŕňa aj zoznam letísk, ktorých *ID* je použité v zozname trás v poliach *Departure Airport ID* a *Arrival airport ID*. Dáta o letiskách majú takmer rovnaký obsah ako dáta zobrazené na obrázku 5.5.

ID	Departure Airport ID	Arrival airport ID	Route length
3749	1	3	809
3750	1	4	214
...	...	...	...
54384	3503	3312	1394

Tabulka 5.1 Vzor dát z datasetu letov

[19]

Ostatné parametre potrebné na nahradenie reálneho zoznamu leteniek, sú generované funkciami na obrázku 5.14. Funkcia generuje parametre `departure`, `arrival` a

```
const getRandomValues = (distance: number): [Date, Date, number] => {
  const departure = getRandomDate(new Date(), <Input max Date here>);
  const durationNum = distance / speed;
  let [durationHour, durationMinute] = convertNumToTime(durationNum);
  if (durationHour === 0 && durationMinute <= 10) {
    durationMinute = 10;
  }
  const copy = new Date(departure);
  const arrival = addHours(addMinutes(copy, durationMinute), durationHour);
  const price = durationNum * pricePerHour;
  return [departure, arrival, price];
};
```

Obrázek 5.14 Funkcia na generovanie náhodných dát jednej letenky

`price`.

- Hodnota `departure` je generovaná ako náhodná hodnota medzi aktuálnym dátumom a náhodným dátumom v budúcnosti, ktorý zvolíme pred spustením funkcie. Vyjadruje čas odletu daného letu.
- Hodnota `arrival` je vypočítaná ako súčet dátumu príletu a doby trvania letu(vypočítané na základe vzdialenosti, ktorej musí lietadlo preletieť a priemernej rýchlosti moderných dopravných lietadiel). Vyjadruje čas príletu daného letu. Minimálna dĺžka letu je nastavená na 10 minút.
- Hodnota `price` je vypočítaná ako cena daného letu pre jedného pasažiera ako násobok dĺžky letu a ceny jednej letovej hodiny, ktorá bola zvolená ako konštanta. Vyjadruje cenu jednej letenky.

Obsah verejného datasetu je čítaný riadok po riadku a nasledujúca funkcia generuje nový dataset obohatený o nové dáta. Obsah funkcie je zobrazený na obrázku 5.15. Funkcia vytvorí rozhrania na čítanie datasetu a zapisovanie nového datasetu. Následne je na každý riadok zavolaná funkcia `processLine`, ktorá dáta v riadku rozdelí na základe čiarok v *.csv* súbore a pomocou vyššie popísanej funkcie obohatí pôvodné dáta o nové náhodne vygenerované data, ktoré sú zapísané do nového datasetu. Vzorka dát nového datasetu je zobrazená v tabuľke 5.2. Tieto vygenerované dáta sú následne využívané ako zdroj dát v databázach ďalších mikroslužieb.

5.5 Implementácia služby registra leteniek

Táto mikroslužba vytvára *gRPC* server na ktorý sa môžu napojiť iba ostatné mikroslužby v clusteri a nie je prístupná ako *REST API*. Na úspešné spustenie tohto servera

```
const processLineByLine = () => {
        const readStream = fs.createReadStream("data/edges.csv");
        const writeStream = fs.createWriteStream("data/flights.csv");

        const rl = readline.createInterface({
        input: readStream,
        crlfDelay: Infinity,
        output: writeStream,
        });
        writeStream.write(
        "~id,~from,~to,~label,dist:int,departure:date,arrival:date,price:float\n"
        );
        rl.on("line", (data) => {
        if (data.split(",")[3] === "route") {
                writeStream.write(processLine(data));
        }
        });
};
```

Obrázek 5.15 Funkcia na generovanie nového datasetu

ID	...	...	...	Departure Date	Arrival Date	Price
3749	...	...	...	2023-05-02T06:08:10.838Z	2023-05-02T08:01:10.838Z	188.13
3750	...	...	...	2023-05-04T13:48:51.326Z	2023-05-04T14:18:51.326Z	49.77
...	...	...	...	...	...	...
54384	...	...	...	2023-05-01T23:21:29.567Z	2023-05-02T02:36:29.567Z	324.19

Tabulka 5.2 Vzor dát nového datasetu

musíme najprv vytvoriť definície *Protocol buffers* pomocou *.proto* súborov.

5.5.1 Definícia Porotocol Buffers

Ide o jazykovo agnostické definície požiadaviek, odpovedí a RPC metód, ktoré server poskytuje a podporuje. Následne sú tieto definície skompilované do zvoleného jazyka jedným z dostupných kompilátorov. V našom prípade bolo rozhodnuté použiť kompilátor *protoc*, ktorý skompiluje vytvorené definície do Javascriptu. Taktiež bol použitý plugin *protoc-gen-ts_proto* pre kompilátor, ktorý pri generovaní vytvorí dodatočné *Typescript* definície, čím sa zachová typová bezpečnosť vytváraného kódu a napájajúcich sa služieb. Takto vytvorené *Javascript* funkcie a *Typescript* typové definície sú následne používané vo všetkých ostatných mikroslužbách, ktoré vyžadujú dáta poskytované touto službou. Obsah celého súboru je zobrazený na obrázku 5.16.

```
syntax = "proto3";
package services.registry.v1;

service RegistryService {
  rpc ReserveTicket(ReserveTicketRequest) returns (ReserveTicketResponse);
  rpc AvailableTicketCount(AvailableTicketCountRequest)
                                    returns (AvailableTicketCountResponse);
  rpc BuyTicket(BuyTicketRequest) returns  (BuyTicketResponse);
}

message ReserveTicketRequest {
  int32 flight_id = 1;
}

message ReserveTicketResponse {
  int32 reserved_ticket_id = 1;
}

message AvailableTicketCountRequest {
    int32 flight_id = 1;
}

message AvailableTicketCountResponse {
    int32 flight_id=1;
    string from=2;
    string to=3;
    string available_capacity=4;
}
message BuyTicketRequest{
    int32 reserved_ticket_id=1;
    string name=2;
    string surname=3;
    string passport=4;
    string birthday=5;
}
message BuyTicketResponse{
        string flight_ticket=1;
}
```

Obrázek 5.16 Definícia protocol buffers

Jednotlivé príkazy majú nasledovný význam:

- `syntax` - Určuje použitú verziu definície *Protocol Buffers*, keďže historicky bolo vytvorených viacero špecifikácií *Protocol Buffers*. V našom prípade využívame najnovšiu verziu `"proto3"`.

- `message` - Definuje dátový typ, ktorý funkcie prijímajú ako argument alebo vracajú ako návratovú hodnotu. Objekt špecifikuje dátový typ, názov vlastnosti a poradové číslo v objekte. Všetky podporované dátové typy sa nachádzajú v oficiálnej dokumentácií *Protocol Buffers*.
- `service` - Vytvára službu, ktorú bude náš *gRPC* server poskytovať. Každá metóda definuje svoj typ(teda či sa jedná o *request/reply* funkciu alebo o funkcie, ktoré podporujú postupné streamovanie dát po častiach) a `message`, ktoré slúžia ako typy argumentov a typy návratových hodnôt z funkcií. Služba obsahuje nasledujúce metódy:
  - `rpc ReserveTicket` - Ide o základný typ, kde klient zasiela požiadavku na server a čaká na odpoveď od servera. Teda chová sa ako typická funkcia. Funkcia slúži na rezervovanie letenky v systéme.
  - `rpc AvailableTicketCount` - Taktiež ide o základný typ funkcie. Ako argument prijíma číslo letu a vracia počet dostupných leteniek.
  - `rpc BuyTicket` - Ide o základný typ funkcie, ktorý ako argument prijíma osobné informácie cestujúceho a v prípade úspechu vracia reťazec s číslom zakúpenej letenky.

Bol vytvorený *bash* skript, ktorý spustí kompilátor *protoc*, ktorý skompiluje už zmienené definície, ktoré automaticky fungujú s používanou knižnicou *grpc-js*. Obsah skriptu je zobrazený na obrázku 5.17. Skript volá lokálne nainštalovaný kompilátor `protoc`,

```
protoc --plugin=protoc-gen-ts_proto=registry/node_modules/.bin/protoc-gen-ts_proto
--ts_proto_out=registry/src proto/services/**/**/*.proto
--ts_proto_opt=outputServices=grpc-js,env=node,esModuleInterop=true
```

Obrázek 5.17 Skript na kompiláciu Protocol Buffers do Javascriptu a Typescriptu

ktorému sú predané nasledovné parametre:

- `--plugin` - nastaví umiestenie pluginu `protoc-gen-ts_proto`, ktorý vytvára *Typescript definície*.
- `--ts_proto_out` - nastaví zložku do ktorej majú byť umiestnené vygenerované súbory.
- `--ts_proto_opt` - nastaví dodatočné nastavenia pre kompilátor:
  - Možnosť `outputServices=grpc-js` - nastavuje výstup, s ktorým je jednoduché pracovať v knižnici `grpc-js`.

- Možnosť `env=node` - nastaví prostredie v ktorom budeme spúšťať aplikáciu na `node`.
- Možnosť `esModuleInterop=true` - zapne príznak `esModuleInterop` čo umožní importovať vygenerované súbory moderným *Javascript* syntaxom *ES6*.

Nakoniec musíme upraviť `package.json` súbor danej mikroslužby aby spúšťala vytvorený skript, ktorý sme nazvali `build_proto.sh` pri každom vybudovaní aplikácie. Časť upraveného `package.json` súboru je zobrazená na obrázku 5.18.

```
"scripts": {
"dev": "ts-node src/index.ts",
"build": "./build_proto.sh && tsc"
},
```

Obrázek 5.18 Upravený package.json súbor

5.5.2 Inicializácia serveru

Prvým krokom pri spúšťaní servera je úspešné nadviazanie spojenia s *PostgreSQL* databázou, ktorej inštancia beží v cloud prostredí firmy *AWS*. Potrebné prístupové údaje na napojenie(*URL adresa, port, názov databáze, užívateľské meno a heslo*) sú získané z premenných prostredia(*enviroment variables*) pomocou *Node* premennej `process.env`. Hodnoty sú nastavené na správne hodnoty počas nasadenia služby pomocou *Pulumi*. V prípade lokálneho vývoja sú použité predvolené hodnoty. Server následne skontroluje, či databáza obsahuje už nejaké dáta. V prípade, že nie, je databáza naplnená dátami, ktoré boli získané a vygenerované v časti 5.4. Príklad funkcie, ktorá naplní databázu dostupnými letmi je na obrázku 5.19.

Funkcia prečíta každý riadok zo súboru, prečíta relevantné časti súboru a vytvorí textový reťazec obsahujúci *SQL* príkaz `INSERT INTO`. Následne je tento príkaz zaslaný cez knižnicu *pg* do databáze. Každý príkaz vracia *Javascript promise*, ktorý je pridaný do poľa `promises`, v ktorom sú uchovávané výsledky bežiacich alebo vyriešených operácií. Po skončení čítania súboru vyvolá *Node* prostredie udalosť `"on"` počas ktorej čaká funkcia pomocou príkazu `Promise.all(promises)` na dokončenie všetkých bežiacich operácií.

```
async function processFlights(client: any) {
  const promises: any = [];
  return new Promise<void>((resolve, reject) => {
    fs.createReadStream("data/flights.csv")
      .pipe(parse({ delimiter: ",", from_line: 2 }))
      .on("data", function (row) {
        const query = `INSERT INTO flight (flight_unique_id,
                        from_id,to_id,capacity) VALUES
                        (
                                ${row[0]},${row[1]},${row[2]},
                                ${randomIntFromInterval(180, 330)}
                        )`;
        promises.push(client.query(query));
      })
      .on("end", () => {
        Promise.all(promises)
          .then(() => {
            console.log("Finished loading flights");
            resolve();
          })
          .catch((e) => {
            reject(e);
          });
      })
      .on("error", function (error) {
        console.error(error.message);
        reject();
      });
  });
}
```

Obrázek 5.19 Funkcia na naplnenie databáze letmi

5.5.3 Implementácia metód serveru

Server implementuje dva typy funkcií. Na svoju funkcionalitu potrebuje funkcie na komunikáciu s databázou a funkcie, ktoré obsluhujú požiadavky, ktoré prijíma *gRPC* server. Príkladom funkcie, ktorá komunikuje s databázou je funkcia na získanie počtu dostupných leteniek z databáze na obrázku 5.20. Funkcia nadviaže spojenie s databázou a vykoná *SQL* príkaz v tele funkcie.

```
export const getAvailableTicketCountForFlight = async (
  flight_id: number,
  client: Client
) => {
  return client.query<{
    flight_unique_id: number;
    from: string;
    to: string;
    available_capacity: string;
  }>(
    `SELECT
  f.flight_unique_id,
  a1.code as "from",
  a2.code as "to",
  f.capacity - COALESCE((SELECT COUNT(*)
  FROM flight_ticket WHERE flight_id = $1), 0) AS available_capacity
FROM flight AS f
JOIN airport AS a1 ON f.from_id = a1.id
JOIN airport AS a2 ON f.to_id = a2.id
WHERE f.flight_unique_id = $1;`,
    [flight_id]
  );
};
```

Obrázek 5.20 Funkcia na získanie počtu dostupných leteniek z databáze

Druhým typom funkcií, ktoré definuje server, sú odpovede na požiadavky, ktoré prijíma *gRPC* server. Príklad funkcie, ktorá obsluhuje tieto požiadavky je ukázaná na obrázku 5.21. V priebehu obsluhy získa najprv funkcia dáta, ktoré zaslala prichádza-

```
const availableTicketCountHanlder = async (
  call: ServerUnaryCall<
    AvailableTicketCountRequest,
    AvailableTicketCountResponse
  >,
  callback: sendUnaryData<AvailableTicketCountResponse>,
  dbClient: Client
) => {
        const { flightId } = call.request;
        try {
    const result = await getAvailableTicketCountForFlight(flightId, dbClient);
      callback(null, {
        flightId: result.rows[0].flight_unique_id,
        from: result.rows[0].from,
        to: result.rows[0].to,
        availableCapacity: result.rows[0].available_capacity,
      });

    }
        ...
}
```

Obrázek 5.21 Príklad funkcie na obslúženie prijatej RPC požiadavky

júca požiadavka. Následne sú tieto dáta vložené do internej funkcie, ktorá kontaktuje databázu a vyhľadá relevantné údaje. Po obdržaní výsledkov *RPC* funkcia pretransformuje dáta a zašle ich ako *RPC* odpoveď.

5.5.4 Implementácia Infrastructure as Code

Vytvorenie databáze Na nasadenie tejto mikroslužby potrebujeme zostaviť obraz pre Docker, ktorý môžeme nasadiť. Tento *image* je vybudovaný podobným *Dockerfile* ako na obrázku 5.8 a skriptom na obrázku 5.9. V oboch prípadoch boli zmenené len cesty k súborom a zložkám. Ako vidíme v návrhu architektúry danej mikroslužby v časti 4.2.1 potrebujeme nasadiť inštanciu *PostgreSQL* databáze v cloud prostredí. To dosiahneme pomocou nasledovných *Pulumi* skriptov. Najprv bolo potrebné nasadiť virtuálny pri-

```
const vpc = new aws.ec2.Vpc("registry-vpc", {
  cidrBlock: "10.0.0.0/16",
});

const subnet1 = new aws.ec2.Subnet("registry-subnet-1", {
  cidrBlock: "10.0.1.0/24",
  vpcId: vpc.id,
});

const subnet2 = new aws.ec2.Subnet("registry-subnet-2", {
  cidrBlock: "10.0.2.0/24",
  vpcId: vpc.id,
});
const dbSubnetGroup = new aws.rds.SubnetGroup("registry-db-subnet-group", {
  subnetIds: [subnet1.id, subnet2.id],
});
```

Obrázek 5.22 Nasadenie VPC

vátny cloud (*VPC*) a v rámci tohto *VPC* vytvoriť subnety, čo je vyžadované priamo dokumentáciou nasadzovaných AWS služieb. V rámci tohto skriptu boli nastavené požadované *CIDR* bloky. Skript je definovaný na obrázku 5.22.

Následne bola nastavená vyžadovaná bezpečnostná politika, ktorá bola napojená na vytvorenú *VPC* a boli nastavené povolené komunikačné porty a protokoly pomocou skriptu na obrázku 5.23.

Pri každom nasadení novej inštancie databázy sa vygeneruje nové užívateľské meno a heslo, ktoré je následne pridané ako *secret* do služby *AWS Secret Manager*, ktorá sa stará o automatické spravovanie a rotovanie vytvorených prístupových údajov. Kód je zobrazený na obrázku 5.24.

Nakoniec bola definovaná konfigurácia ktorou je spúšťaná inštancia *PostgreSQL* databáze. V konfigurácií sú nastavené veľkosti a typy použitých úložísk, ktoré poskytuje *AWS*, vytvorená inštancia bezpečnostnej politiky a vygenerované prihlasovacie údaje. Konfigurácia je ukázaná na obrázku 5.25.

```
const dbSecurityGroup = new aws.ec2.SecurityGroup(
  "registry-db-security-group",
  {
    vpcId: vpc.id,
    ingress: [
      {
                ...
        fromPort: 5432,
        toPort: 5432,
        protocol: "tcp",
      },
    ],
  }
);
```

Obrázek 5.23 Nasadenie bezpečnostnej politiky

Poslednou podstatnou časťou je definovanie nových *Pulumi* <Output> hodnôt, ktoré sú potrebné pri finálnom nasadení celej aplikácie a ktoré používajú iné mikroslužby. Exportované sú prihlasovacie údaje, názov databáze a port na ktorom komunikuje databáza. Skript je zobrazený na obrázku 5.26.

Vytvorenie služieb a nastavenie automatického škálovania Služby a nastavenie automatického škálovania bolo dosiahnuté rovnakým spôsobom ako v predchádzajúcej časti, podobnými skriptami ako na obrázkoch 5.10, 5.12 a 5.13. Zmenené boli pritom, len názvy jednotlivých služieb a do kontajnerov boli predané `env` premenné, ktoré sme exportovali na obrázku 5.26.

```
const username = new random.RandomUuid("username");
const password = new random.RandomPassword("password", {
  length: 16,
  special: true,
});
const secret = new aws.secretsmanager.Secret("registry-db-credentials");
const secretVersion = new aws.secretsmanager.SecretVersion(
  "dbCredentialsVersion",
  {
    secretId: secret.id,
    secretString: pulumi.interpolate`{
      "username": ${username.result},
      "password": ${password.result}
  }`,
  }
);
```

Obrázek 5.24 Generovanie a uloženie prihlasovacích údajov do služby AWS Secret Manager

```
const postgresDatabase = new aws.rds.Instance("registry-postgres-database", {
  engine: "postgres",
  engineVersion: "13.3",
  instanceClass: "db.t2.2xlarge",
  allocatedStorage: 20,
  storageType: "gp2",
  dbName: "registry-database",
  username: username.result,
  password: password.result,
  vpcSecurityGroupIds: [dbSecurityGroup.id],
  dbSubnetGroupName: dbSubnetGroup.name,
  publiclyAccessible: false,
  skipFinalSnapshot: true,
});
```

Obrázek 5.25 Vytvorenie databáze pomocou Pulumi

```
export const dbName = postgresDatabase.dbName;
export const dbPort = postgresDatabase.port;
export const secretArn = secret.arn;
export const dbUsername = pulumi.secret(username.result);
export const dbPassword = pulumi.secret(password.result);
```

Obrázek 5.26 Exportované hodnoty pri vytváraní databáze

5.6 Implementácia služby na vyhľadávanie leteniek

Mikroslužba je implementovaná rovnakým prístupom ako v časti 5.3. Mikroslužba po svojom spustení nadviaže spojenie s databázou a s mikroslužbou registra leteniek. Po obdržaní *HTTP* požiadavky cez verejné *REST API* vyhľadá server v databáze dostupné lety. Príklad funkcie, ktorá vyhľadá dostupné lety s povolenými prestupmi je ukázaná na obrázku 5.27. Funkcia najprv čaká na odpoveď z databázového servera aby získala lety, ktoré vyhovujú parametrom pomocou funkcie `getFlightsWithOneStop()`. Následne je pre každý nájdený let zaslaná požiadavka do mikroslužby registra leteniek - `availableTicketCount()`, ktorá vráti počet voľných miest danej letenky. Výsledné letenky sú vyfiltrované a vrátené ako odpoveď.

```
const flights = (
        await getFlightsWithOneStop(from, to, departure_date, dbClient)
).rows;
const filteredFlights = await Promise.all(
        flights.map(async ({ flight1, flight2, total_cost, total_duration }) => {
        const ticketArray = await Promise.all([
                availableTicketCount(
                { flightId: flight1.flight_unique_id },
                registryClient
                ),
                availableTicketCount(
                { flightId: flight2.flight_unique_id },
                registryClient
                ),
        ]);
        return ticketArray.some(
                ({ availableCapacity }) => Number(availableCapacity)
                > passenger_count
        )
                ? { flight1, flight2, total_cost, total_duration }
                : null;
        })
);
```

Obrázek 5.27 Funkcia na vyhľadanie letu

5.6.1 Implementácia Infrastructure as Code

Zostavenie servera Zostavenie obrazu pre Docker prebehlo podobným spôsobom ako na obrázku 5.8 a skriptom na obrázku 5.9. V oboch prípadoch boli zmenené len cesty k súborom a zložkám. V prípade tejto mikroslužby sú kroky potrebné k jej nasadeniu

zhodné ako v prípade mikroslužby registra leteniek, keďže potrebujeme taktiež nasadiť *VPC*, vygenerovať prístupové údaje a nasadiť inštanciu *PostgreSQL* databáze v clude *AWS*. Postup pre nasadenie je preto zhodný s obrázkami 5.22, 5.24 a 5.25.

Vytvorenie služieb a nastavenie automatického škálovania Vytvorenie služieb v *Kubernetes* prostredí a nastavenie škálovanie bolo dosiahnuté podobným spôsobom ako v časti 5.10, 5.12 a 5.13. `Env` premenné, ktoré potrebuje k svojej činnosti *image* mikroslužby na vyhľadávanie letísk boli získané kódom na obrázku 5.28. Najprv získame referenciu na existujúce nasadenie vytvorenej databáze pomocou funkcie *pulumi.StackReference()*, ktorá ako argument prijíma názov nasadenej služby. Následne funkciou `getOutput()`, získame hodnotu, ktorú môžeme vložiť a predať do prostredia nasadzovaného *Docker* kontajneru.

```
const flight_search_stackRef = new pulumi.StackReference(
    `org_name/flight_search/${stack}`
  );
const dbName = flight_search_stackRef.getOutput("dbName");
const dbUsername = flight_search_stackRef.getOutput("dbUsername");
const dbPassword = flight_search_stackRef.getOutput("dbPassword");
const dbPort = flight_search_stackRef.getOutput("dbPort");
```

Obrázek 5.28 Získanie výstupov existujúceho nasadenia

5.7 Implementácia a nasadenie služby na rezerváciu a kúpu leteniek

Keďže ide o jednoduchú mikroslužbu, ktorá principiálne slúži ako *REST HTTP* proxy pre niektoré funkcie mikroslužby registra leteniek, je implementácia a nasadenie tejto mikroslužby pomerne jednoduché. Implementácia zahŕňa už popísané prístupy v predchádzajúcich kapitolách (5.3, 5.5). Príklad funkcie, ktorá slúži ako *proxy* k rezervácií letenky je na obrázku 5.29. Obrázok popisuje časť funkcie, ktorá overí správnosť prijatých parametrov a následne počká na odpoveď *gRPC* servera. Výsledok je potom zaslaný ako odpoveď v *JSON* formáte.

Nasadenie *Kubernetes* služby zahŕňa zostavenie *Docker image*, rovnakým spôsobom ako v predchádzajúcich častiach a vytvorenie samotného nasadenia a služby v *Kubernetes* clusteri pomocou rovnakého postupu, aký bol použitý na obrázku 5.10.

```
export async function reserveTicket(
clients: ReserveClients,
req: PostReserveTicketApiRequest,
res: Response,
next: NextFunction
) {
...
const parsedBody = parseNumberSafely(req.body.flight_id);
if (parsedBody === null) {
        throw new ApiError(422, "Unable to parse body parameters");
}
const result = await reserveTicketModel({ flightId: parsedBody }, clients);
res.json({ success: true, data: result });
...
}
```

Obrázek 5.29 Funkcia na rezerváciu letenky

5.8 API Gateway

API Gateway je služba, ktorá slúži ako jediný a jednotný vstupný bod, cez ktorý môžu klienti pristupovať k vytvoreným backendovým mikroslužbám. Poskytuje dodatočnú vrstvu abstrakcie a správu dostupných *endpointov API* a ponúka možnosti ako zvýšiť bezpečnosť a škálovateľnosť dostupných služieb. Brána ponúka nasledovné funkcie:

- Zjednotenie API - Brána umožňuje spojiť viacero backendových mikroslužieb do jednotného *API*. Klienti a požiadavky môžu byť tak smerované na jeden *endpoint*. Zjednoduší sa tak komunikácia medzi serverom a klientom.
- Smerovanie požiadaviek a *Load Balancing* - Gateway smeruje prichádzajúce požiadavky na relevantnú backendovú službu podľa vopred definovaných pravidel. Dokáže tiež rozdeliť zvýšené zaťaženie aplikácie pomocou *load balancingu* na viacero inštancií tej istej služby.
- Zabezpečenie - Brána poskytuje služby ako je autentifikácia, autorizácia a šifrovanie. Pracuje s autentifikačnými metódami ako *API* kľúče, *OAuth* a *JWT tokeny*, čím zabezpečí, že k backendovým službám budú mať prístup len autorizovaní klienti. Pre vývojára taktiež uľahčuje implementáciu mikroslužieb, keďže zabezpečenie nemusí byť riešené v rámci každej mikroslužby, ale centrálne pre celú aplikáciu.
- Caching - Gateway dokáže ukladať odpovede, ktoré zasiela backend, do vyrovnávacej pamäte a redukovať tak zaťaženie a znižovať latenciu služieb.

- Monitorovanie a analytika - Gateway dovoľuje zhromažďovať a sledovať metriky súvisiace s používaním *API* a monitorovať tak výkon, stav a chybovosť služieb.

5.8.1 Nasadenie API Gateway

API Gateway bola nasadená použitím *AWS* služby *API Gateway* prostredníctvom *Pulumi*. Keďže však používame počas vývoja *LocalStack* aplikáciu na emulovanie *AWS* prostredia musíme vykonať špecifické úpravy. Konfigurácia brány vyžaduje existujúcu doménu na ktorej je nasadený vytvorený *Kubernetes* cluster a backendové mikroslužby. Vytvorené služby sú však nasadené cez emulátor *LocalStack* v *Kubernetes* clusteri na lokálnom počítači na adrese `localhost`. Na riešenie tejto situácie bola využitá aplikácia na vytvorenie tunela, ktorý vystaví `localhost` prostredie pod verejnou IP adresou, ktorú už dokáže *API Gateway* kontaktovať. Na vytvorenie tohto tunela bola použitá open-source aplikácia *bore*, ktorá ponúka serverové aj klientske varianty. Následne bola prenajatá inštancia serveru s verejnou IP adresou. Na tomto serveri bol spustený *bore* server pomocou príkazu `bore server`, ktorý bude presmerovávať prichádzajúce požiadavky pripojeným klientom. Na lokálnom počítači bol vytvorený tunel pomocou príkazu:

```
bore local <LOCAL PORT> --to <SERVER IP> --port <SERVER PORT>
```

Následne bola nasledovnými krokmi nasadená *API Gateway* pomocou *Pulumi*:

- Vytvorenie inštancie *gateway*

```
const apiGateway = new aws.apigateway.RestApi("apiGateway", {
name: "api-gateway",
});
```

Obrázek 5.30 Vytvorenie inštancie API Gateway

- Vytvorenie *endpointu*, ktorý bude brána podporovať. V tomto prípade ide o vytvorenie endpointu pre mikroslužbu na hľadanie letísk na adrese `airports`.

```
const apiGatewayResource = new aws.apigateway.Resource(
        "airportSearchResource", {
    parentId: apiGateway.rootResourceId,
    pathPart: "airports",
    restApi: apiGateway,
  });
```

Obrázek 5.31 Vytvorenie endpointu API Gateway

- Vytvorenie inštancie metódy, ktorá bude na úrovni *API Gateway* spracovávať prichádzajúce požiadavky.
  - `authorization` - nastavuje protokoly, ktoré sú potrebné na zabezpečenie *endpointu*. V tomto prípade je hodnota nastavená na `"NONE"`, keďže tento *endpoint* má byť verejne dostupný.
  - `httpMethod` - nastavuje podporovanú *HTTP* metódu na tomto *endpointe*.
  - `requestParameters` - nastavujeme parametre, ktoré by mala obsahovať prijatá požiadavka. Nastavením *boolean* hodnoty určujeme, či je daný parameter vyžadovaný.

```
const apiGatewayMethod = new aws.apigateway.Method(
        "airportSearchMethod", {
        authorization: "NONE",
        httpMethod: "GET",
        resourceId: apiGatewayResource.id,
        restApi: apiGateway,
        requestParameters: {
                "method.request.querystring.text": true,
                ...
        },
});
```

Obrázek 5.32 Vytvorenie inštancie metódy API Gateway

- Vytvorenie inštancie integrácie, ktorá presmeruje požiadavku na správnu mikroslužbu.
  - `type` - určuje typ integrácie na ktorý bude požiadavka presmerovaná,
  - `uri` - určuje adresu mikroslužby na ktorú ma byť požiadavka presmerovaná.
  - `integrationHttpMethod` - určuje *HTTP* metódu, ktorú očakáva služba na ktorú ma byť požiadavka presmerovaná.

```
const apiGatewayIntegration = new aws.apigateway.Integration(
"airportSearchIntegration",
{
        httpMethod: apiGatewayMethod.httpMethod,
        resourceId: apiGatewayResource.id,
        restApi: apiGateway,
        type: "HTTP",
        uri: pulumi.interpolate
        `${targetUrl}/${apiGatewayResource.pathPart}`,
        integrationHttpMethod: "GET",
        requestParameters: {
        "integration.request.querystring.text":
                "method.request.querystring.text",
        },
}
);
```

Obrázek 5.33 Vytvorenie inštancie integrácie pre API Gateway

  - `requestParameters` - určuje spôsob, akým spôsobom majú byť namapované parametre požiadavky na parametre, ktoré očakáva mikroslužba.

- Vytvorenie `stage`, ktorá určuje aktuálnu verziu *API* a vykonanie samotného nasadenia.

```
const apiGatewayStage = new aws.apigateway.Stage("apiGatewayStage", {
        deployment: apiGatewayDeployment,
        restApi: apiGateway,
        stageName: "v1",
});
const apiGatewayDeployment = new aws.apigateway.Deployment(
    "apiGatewayDeployment",
    {
      restApi: apiGateway,
    })
```

Obrázek 5.34 Vytvorenie stage pre API Gateway

V prípade mikroslužby na rezerváciu a zakúpenie leteniek chceme zabezpečiť aby bol tento *endpoint* prístupný len autentifikovaným užívateľom, ktorý v hlavičke požiadavky zašlú validný token. To je zabezpečené použitím a vytvorením inštancie služby *AWS Cognito Pool*, ktorá sa stará o ukladanie užívateľských údajov. Užívateľské rozhranie potom môže vyzvať užívateľa na prihlásenie(napríklad pomocou knižnice *AWS Amplify*,

ktorá dokáže prihlásiť užívateľa použitím služby *AWS Cognito Pool*). Následne bola *API Gateway* nastavená, aby na určitých *endpointoch* overovala zaslané autorizačné tokeny. Toto bolo dosiahnuté nasledujúcimi krokmi:

- Vytvorenie inštancie *AWS Cognito Pool*

```
const apiUserPool = new aws.cognito.UserPool(
        "apiUserPool", {
  name: "userPool",
});
```

Obrázek 5.35 Vytvorenie inštancie AWS Cognito Pool

- Vytvorenie inštancie *Gateway Authorizer*

```
const apiAuthorizer = new aws.apigateway.Authorizer(
        "apiAuthorizer", {
        restApi: apiGateway.id,
        type: "COGNITO_USER_POOLS",
        identitySource: "method.request.header.Authorization",
        providerArns: [apiUserPool.arn],
        name: "apiAuthorizer",
});
```

Obrázek 5.36 Vytvorenie inštancie AWS Gateway Authorizer

  - `restApi` - nastaví *API bránu*, ku ktorej bude *Authorizer* priradený.
  - `type` - nastaví *AWS* službu, ktorá bude overovať identitu užívateľa.
  - `identitySource` - nastavuje cestu, kde v hlavičke požiadavky má byť hľadaný autorizačný token.
  - `providerArns` - nastaví poskytovateľa na vytvorený *AWS Cognito Pool*.

- Nastavenie *endpointu*, aby používal vytvorený *Authorizer*. To je dosiahnuté pridaním nastavení do inštancie metódy *AWS Gateway*.

```
authorizerId: apiAuthorizer.id,
authorization: "COGNITO_USER_POOLS",
```

Obrázek 5.37 Nastavenie endpointu, aby používal AWS Authorizer

6 NASADENIE A TESTOVANIE APLIKÁCIE

6.1 Nasadenie aplikácie

Pred spustením aplikácie je odporúčané nainštalovať nástroj `k9s`. Ide o terminálovú aplikáciu, ktorá poskytuje prehľadnejší spôsob ako interagovať a manipulovať s Kubernetes clustermi. V prípade vytvorenej aplikácie umožňuje vizualizáciu postupného nasadzovania jednotlivých služieb pomocou *Pulumi*. Počas testovania aplikácie bolo podobným spôsobom vizualizované automatické škálovanie jednotlivých podov v ktorých bežali inštancie mikroslužieb.

Aplikácia je ako celok nasadená nasledovnými krokmi. V prípade využitia lokálneho vývoja je spustená aplikácia *LocalStack* pomocou príkazu `localstack start`, ktorý spustí emulátor služieb *AWS* v *Docker* kontajneri. Taktiež bol spustený už spomenutý klient, ktorý vytvára tunel, ktorý presmerováva požiadavky určené *AWS Gateway* na adresu `localhost`. Nasadenie jednotlivých mikroslužieb bolo dosiahnuté spustením príkazu `pulumi up` v adresároch jednotlivých mikroslužieb. V konzole môžeme následne sledovať výstup z programu *Pulumi*. *Pulumi* najprv overí správnosť nadefinovaných skriptov a čaká na potvrdenie nasadenia užívateľom. Výstup z konzole je zobrazený na obrázku 6.1. Po potvrdení nasadenia sú spustené jednotlivé časti a skripty nasadenia.

```
    Type                                                    Name                                    Status
    pulumi:pulumi:Stack                                     skynet-dev
+   ├─ kubernetes:yaml:ConfigFile                           metrics-server                          created (0.
+   │  ├─ kubernetes:rbac.authorization.k8s.io/v1:ClusterRoleBinding  system:metrics-server         created (2s
+   │  ├─ kubernetes:core/v1:Service                        kube-system/metrics-server              created (12
+   │  ├─ kubernetes:rbac.authorization.k8s.io/v1:RoleBinding  kube-system/metrics-server-auth-reader  created (2s
+   │  ├─ kubernetes:rbac.authorization.k8s.io/v1:ClusterRole  system:metrics-server               created (3s
+   │  ├─ kubernetes:rbac.authorization.k8s.io/v1:ClusterRoleBinding  metrics-server:system:auth-delegator  created (3s
+   │  ├─ kubernetes:rbac.authorization.k8s.io/v1:ClusterRole  system:aggregated-metrics-reader     created (3s
+   │  ├─ kubernetes:apps/v1:Deployment                     kube-system/metrics-server              created (25
+   │  ├─ kubernetes:core/v1:ServiceAccount                 kube-system/metrics-server              created (5s
+   │  └─ kubernetes:apiregistration.k8s.io/v1:APIService   v1beta1.metrics.k8s.io                  created (5s
+   ├─ aws:ecr:Repository                                   reserve-repo                            created (0.
+   ├─ aws:ecr:Repository                                   start_registry-repo                     created (1s
+   ├─ aws:apigateway:RestApi                               apiGateway                              created (1s
+   ├─ aws:cognito:UserPool                                 apiUserPool                             created (3s
+   ├─ aws:ecr:Repository                                   flight-search-repo                      created (3s
+   ├─ kubernetes:apps/v1:Deployment                        elastic-search-deployment               created (2s
+   ├─ aws:apigateway:Resource                              buyResource                             created (2s
+   ├─ aws:apigateway:Resource                              apiGatewayResource                      created (2s
+   ├─ aws:apigateway:Resource                              flightSearchResource                    created (2s
+   ├─ aws:apigateway:Resource                              reserveResource                         created (3s
+   ├─ docker:index:Image                                   airport-search-image                    created (6s
+   ├─ aws:cognito:UserPoolClient                           apiUserPoolClient                       created (3s
+   ├─ aws:apigateway:Authorizer                            apiAuthorizer                           created (4s
```

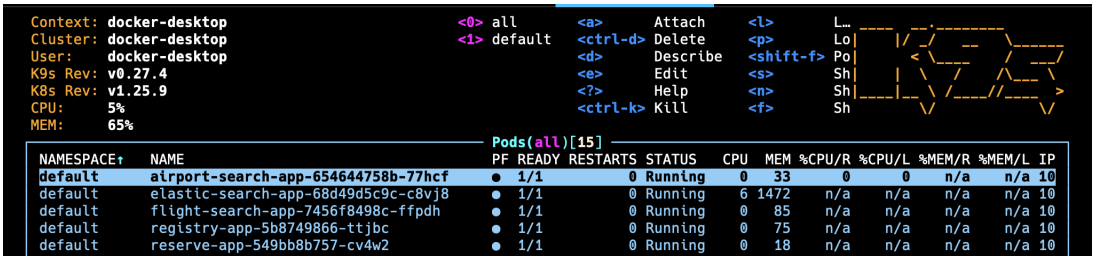
Obrázek 6.1 Ukážka priebehu nasadenia pomocou Pulumi

Výstupom úspešného nasadenia je *HTTP* adresa na ktorej je dostupná *API Gateway*, ktorá je pripravená spracovávať požiadavky. Výsledok je zobrazený na obrázku 6.2.

```
Outputs:
  ~ gateway_url     : "https://xqwcp02ghj.execute-api.eu-central-1.amazonaws.com/v1" => "https://1gj2ziodby.execute-api.eu-
central-1.amazonaws.com/v1"
```

Obrázek 6.2 Výstup nasadenia aplikácie

Priebeh nasadenia jednotlivých mikroslužieb a ich podov môžeme taktiež sledovať cez aplikáciu `k9s`, ktorej výstup je na obrázku 6.3.



Obrázek 6.3 Ukážka nasadeného Kubernetes clustera v aplikácií K9S

6.2 Testovanie aplikácie

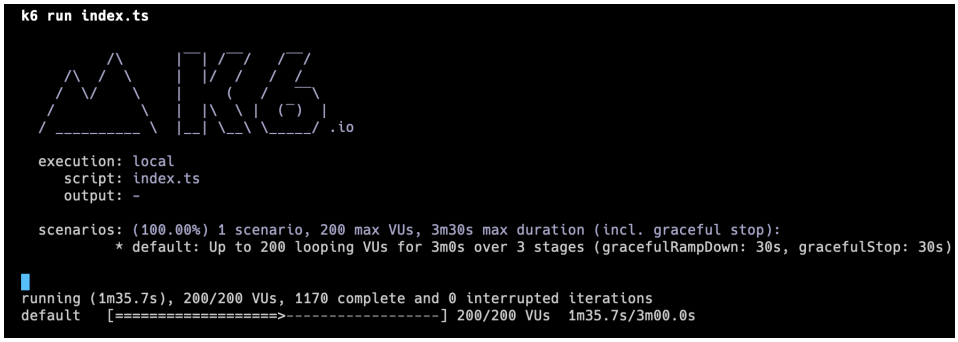
Testovanie aplikácie bolo realizované testovaním nastaveného automatického škálovania, ktoré bolo nastavené pri nasadzovaní jednotlivých mikroslužieb. Na simulovanie zvýšenej záťaže a vyššieho počtu požiadaviek bola použitá *Javascript* knižnica `K6`. Umožňuje vytvorenie skriptov, ktoré vytvárajú virtuálnych používateľov, ktorý navštevujú zvolené *endpointy* zvoleného *API*. V každom skripte je nadefinovaný scenár, ktorý určuje ako rýchlo pribúda, alebo odchádza počet užívateľov a ktoré adresy majú byť navštívené. Príklad tohto skriptu, ktorý zašle požiadavku na vyhľadanie letenky a vyhľadanie letov je na obrázku 6.4. Scenár v priebehu prvej minúty postupne zvyšuje počet súbežných užívateľov, následne je držaná hodnota užívateľov po určenú dobu a následne nápor postupne upadá.

```
export const options = {
  stages: [
    { duration: "1m", target: 200 },
    { duration: "1m", target: 200 },
    { duration: "1m", target: 0 },
  ],
};
const airports_url = BASE_URL + "<QUERY_PARAMS>";
const flight_url = BASE_URL +"<QUERY_PARAMS>";

export default () => {
  const airport_search_res = http.get(airports_url.toString());
  sleep(5);
  const flight_search = http.get(flight_url.toString());
  sleep(3);
};
```

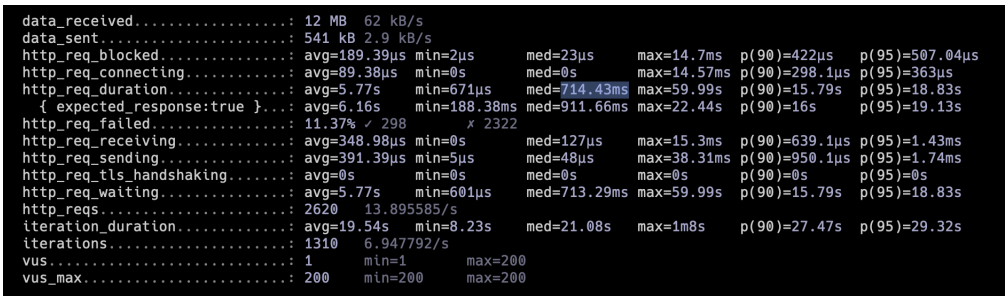
Obrázek 6.4 Skript na generovanie záťaže pomocou knižnice K6

Priebeh *load testingu* je zobrazený na obrázku 6.5. Po skončení testovania je získaný výstup, ktorý informuje o priebehu testovania, ktorý ukazuje napríklad počet zaslaných požiadaviek, priemerné oneskorenie, počet zlyhaných požiadaviek a iné. Výsledok

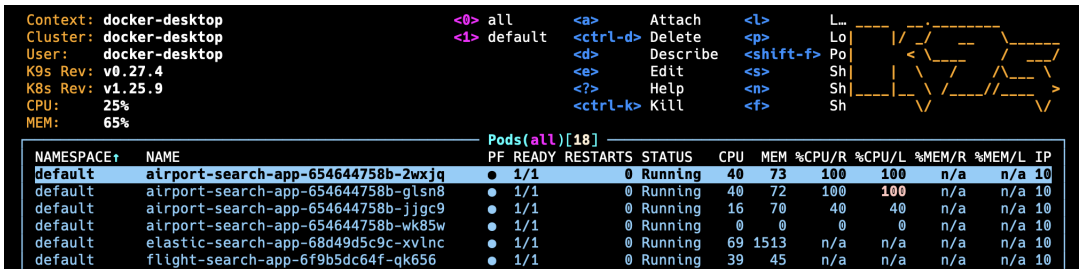


Obrázek 6.5 Ukážka priebehu load testingu

z konzole je zobrazený na obrázku 6.6. V priebehu testovania bol taktiež pomocou aplikácie `k9s` sledovaný počet bežiacich inštancií resp. počet bežiacich podov, ktoré boli v clusteri spustené. Tento proces je ukázaný na obrázku 6.7.



Obrázek 6.6 Konzolový výstup load testingu



Obrázek 6.7 Ukážka automatického škálovania

6.3 Zhodnotenie výsledkov

Počas zvýšenej záťaže jednej z mikroslužieb bolo zaznamenané automatické škálovanie a zvýšenie počtu bežiacich inštancií aby sa aplikácia dokázala vysporiadať so zvýšenou záťažou. Obrázok 6.7 ukazuje zvýšený počet podov bežiacej služby na vyhľadávanie letísk. Bola tak dokázaná funkčnosť automatického škálovania a správneho nastavenia nasadenia mikroslužby tak, aby reagovala na zvýšené zaťaženie.

Dôležité časti výsledku load testingu boli zobrazené v tabuľkách 6.1 a 6.2, pričom boli testované dva scenáre, s rozličným počtom súčasných užívateľov. Pri počte 10

Minimálna doba čakania	145 ms
Medián doby čakania	471 ms
Maximálna doba čakania	5.2s
Percento zlyhaných požiadaviek	1.45%
Počet požiadaviek za sekundu	1.66/s

Tabulka 6.1 Výsledky load testingu - 10 súčasných užívateľov

Minimálna doba čakania	188 ms
Medián doby čakania	911 ms
Maximálna doba čakania	22.4s
Percento zlyhaných požiadaviek	11.37%
Počet požiadaviek za sekundu	13.9/s

Tabulka 6.2 Výsledky load testingu - 200 súčasných užívateľov

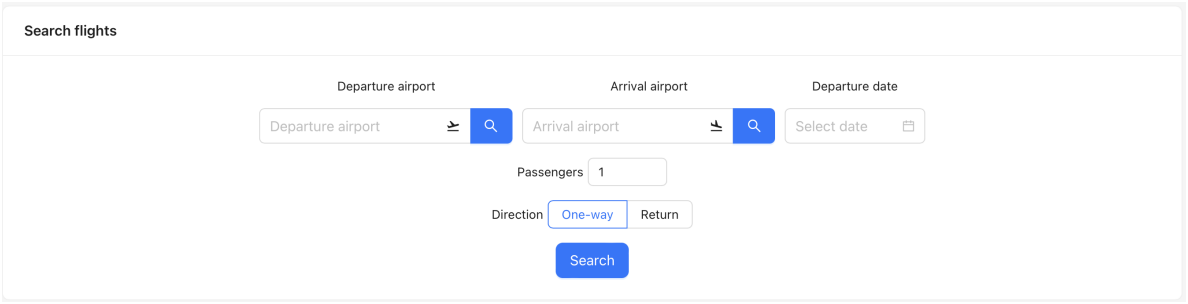
súčasných užívateľov systém zvládal záťaž a dosahoval výsledky v rámci očakávaní, ako je vidieť v tabuľke 6.1.

Ako však vyplýva z tabuľky 6.2 došlo v navrhnutej aplikácií, pri počte 200 súčasných užívateľov, aj napriek škálovaniu, k zlyhaniam niektorých požiadaviek a dlhému čakaniu na ich obslúženie. Najpravdepodobnejším odôvodnením toho výsledku, je, že cloud infraštruktúra beží pomocou emulátora *LocalStack* na lokálnom počítači. Taktiež požiadavky, ktoré sú generované pomocou nástroja *K6* pochádzajú z lokálneho prostredia. Následne musia byť požiadavky zasielané cez tunel na server, ktorý kontaktuje *API Gateway*. Celý tento proces výrazne spomaľuje obsluhu požiadaviek a emulátor infraštruktúry nezvláda túto záťaž.

6.4 Užívateľské rozhranie

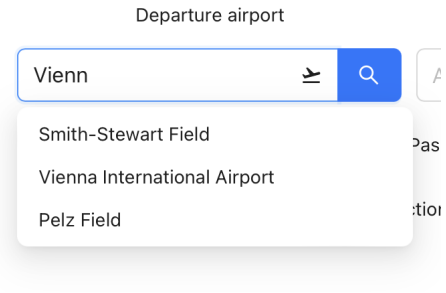
V rámci demonštrácie funkčnosti aplikácie, bola vytvorená webová aplikácia pomocou knižnice React. Aplikácia volá všetky implementované endpointy pomcou *API Gateway*.

Po príchode na stránku je zobrazená domovská stránka, na ktorej môže užívateľ vyhľadať letenky poľa zvolených kriterií(Obrázok 6.8).



Obrázek 6.8 Domovská stránka aplikácie

Ak užívateľ vloží text do textového poľa *Arrival* alebo *Departure*, dôjde k zobrazeniu rozbaľovacej ponuky, ktorá ponúka dostupné letiská na základe čiastočnej zhody pomocou služby na vyhľadávanie letísk(Obrázok 6.9). Po vyplnení potrebných kritérií

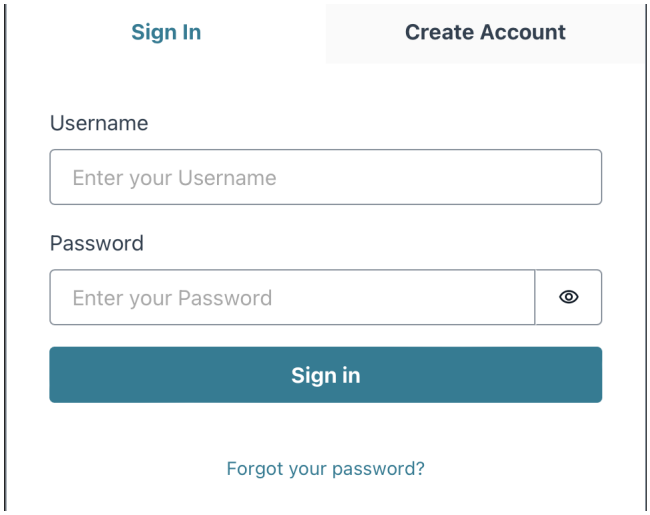


Obrázek 6.9 Užívateľské rozhranie pre vyhľadávanie letísk

sú vyhľadané dostupné letenky a v užívateľskom rozhraní je zobrazený zoznam letov, ktoré si môže užívateľ rezervovať(Obrázok 6.10). Po kliknutí na tlačidlo *Reserve* je

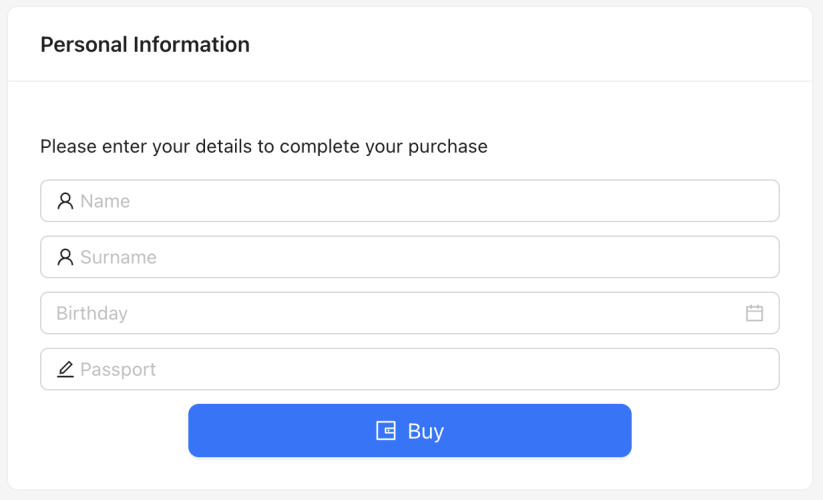
Obrázek 6.10 Dostupná letenka

overné, či je užívateľ prihlásený. Ak nie, tak je presmerovaný na prihlasovaciu stránku, ktorá je implementovaná pomocou *AWS Amplify*(Obrázok 6.11). Po prihlásení ob-



Obrázek 6.11 Formulár určený na prihlásenie

drží užívateľ *JWT Token*, ktorý je užívateľským rozhraním využívaný pri komunikácií so zabezpečenou časťou *API Gateway*. Následne je užívateľovi umožnené zakúpenie letenky po vyplnení osobných informácií(Obrázok 6.12).



Obrázek 6.12 Formulár pre vyplnenie osobných údajov

ZÁVĚR

Cieľom tejto diplomovej práce bolo uviesť čitateľa do problematiky architektúry moderných aplikácií, ktoré sú následne nasadené v cloud prostredí pomocou prístupu Infrastructure as Code a vytvorenie tutoriálu, ktorý prevedie čitateľa implementáciou a nasadením navrhnutej aplikácie.

V teoretickej časti boli popísané rozdiely medzi monolitickými a distribuovanými architektúrami. Z oblasti distribuovaných architektúr boli popísané prístupy service-based architektúry, event-driven architektúry a microservice architektúry. Následne bol definovaný prístup vývoja Infrastructure as Code, princípy tohto prístupu a jeho výhody a nevýhody. V poslednej časti boli popísané dostupné nástroje a prostriedky, z ktorých boli v praktickej časti vybrané nástroje použité pri implementácií.

V praktickej časti bola navrhnutá aplikácia, ktorá slúži na demonštráciu princípov popísaných v teoretickej časti. Ide o vyhľadávač leteniek, ktorý umožňuje užívateľom vyhľadať dostupné letenky na základe zvolených parametrov. Vyhľadané letenky je možné následne zakúpiť. Aplikácia využíva architektúru mikroslužieb. Každá služba je popísaná z hľadiska účelu, jej funkcionality v rámci celej aplikácie a architektúry samotnej služby. Jednotlivé služby sú automaticky zostavené pomocou aplikácie *Docker*. Z dostupných nástrojov popísaných v teoretickej časti bol zvolený *IaC* nástroj *Pulumi* pomocou ktorého boli implementované nasadenia jednotlivých služieb aplikácie a ich automatické škálovanie založené na aktuálnej záťaži. Nástrojom *Pulumi* bolo taktiež implementované nasadenie clustera Kubernetes, ktorý orchestruje nasadené služby v kontajneroch. Celá definovaná infraštruktúra bola automaticky nasadená do cloud prostredia spoločnosti *AWS*. Toto prostredie bolo emulované využitím aplikácie *LocalStack*.

Jednotlivé spomenuté kroky sú popísané v rámci tutoriálu, ktorý prevedie čitateľa všetkými krokmi, ktoré potrebuje k úspešnej implementácií navrhnutej aplikácie s využitím architektúry mikroslužieb a prístupu Infrastructure as Code. Tutoriál zahŕňa návod ako správne nastaviť repozitár a nainštalovať potrebné súčasti a knižnice. Následne tutoriál popisuje implementáciu osobitných mikroslužieb, ktoré na svoju funkcionalitu využívajú technológie ako Javascript/Typescript, Node.js, Elasticsearch, gRPC, PostgreSQL a Express. V rámci implementácie každej služby bol popísaný taktiež návod ako nasadiť infraštruktúru potrebnú na jej fungovanie za použitia nástroja *Pulumi*.

Na záver bola nasadená aplikácia otestovaná pomocou záťažového testu, ktorého účelom bolo vyskúšanie automatického škálovania mikroslužieb. Priebeh záťažového testu bol monitorovaný pomocou nástroja *k9s*, ktorý kontroloval aktuálny stav Kubernetes clustera. Výsledkom testu bolo úspešné pozorovanie automatického škálovania jednotlivých mikroslužieb založené na aktuálnom zaťažení systému. Pre doplnenie testovania funkcionality navrhnutej aplikácie bola implementovaná webová aplikácia s využitím

knižnice *React*, ktorá demonštruje proces vyhľadania letenky až po jej zakúpenie.

SEZNAM POUŽITÉ LITERATURY

[1] RICHARDS, Mark a Neal FORD. Fundamentals of software architecture: an engineering approach. Beijing: O'Reilly, 2020. ISBN 9781492043454.

[2] RICHARDS, Mark a Neal FORD. Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. Sebastopol: O'Reilly Media, 2021. ISBN9781492086895.

[3] NEWMAN, Sam. Building Microservices:Designing Fine-Grained Systems. 2nd edition. Sebastopol:O'Reilly Media, 2021. ISBN 9781492034025.

[4] MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. 1. London, England: Prentice Hall, [2018]. Robert C. Martin series. ISBN 0134494164.

[5] BUCHANNAN, Ian. Feature flags: How to progressively expose your features with feature flags. Atlassian [online]. Sydney: Atlassian [cit. 2023-05-20]. Dostupné z: https://www.atlassian.com/continuous-delivery/principles/feature-flags

[6] Eventual Consistency. Scylla [online]. Palo Alto, 2021 [cit. 2023-05-15]. Dostupné z: https://www.scylladb.com/glossary/eventual-consistency

[7] Architecture styles [online]. Redmond: Microsoft Corporation, 2022 [cit. 2023-05-10]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/

[8] KAWUREK, Dennis. The two topologies of event driven architectures. Dkwr.de [online]. Giessen, 2022, 21.03.2022 [cit. 2023-05-12]. Dostupné z: https://blog.dkwr.de/architecture/event-driven-architecture-topologies/

[9] Communication in a microservice architecture. Microsoft [online]. Redmond: Microsoft Corporation, 2022, 04/13/2022 [cit. 2023-05-19]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

[10] MORRIS, Kief. Infrastructure as Code: Dynamic Systems for the Cloud Age. 2nd edition. Sebastopol:O'Reilly Media, 2021. ISBN9781098114671.

[11] KIM, Gene, Jez HUMBLE, Patrick DEBROIS, John WILLIS a Nicole FORSGREN. The Dev Ops Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations. Second edition. Portland: IT Revolution Press, 2021. ISBN9781942788003.

[12] EDWARDS, April. Infrastructure as Code (IaC): Comparing the Tools. Microsoft Corporation [online]. Redmond, 2022, 22.04.2022 [cit. 2023-05-10]. Dostupné z: https://techcommunity.microsoft.com/t5/itops-talk-blog/infrastructure-as-code-iac-comparing-the-tools/ba-p/3205045

[13] Terraform Documentation [online]. San Francisco: Hashicorp, 2023 [cit. 2023-05-20]. Dostupné z: https://developer.hashicorp.com/terraform/docs

[14] AWS Documentation [online]. Seattle: Amazon Web Services [cit. 2023-05-15]. Dostupné z: https://docs.aws.amazon.com

[15] Pulumi concepts. Pulumi Docs [online]. Seattle: Pulumi Corporation, 2023 [cit. 2023-05-13]. Dostupné z: https://www.pulumi.com/docs/

[16] LUKŠA, Marko. Kubernetes in Action. Shelter Island: Manning, 2018. ISBN 9781617293726.

[17] Kubernetes Documentation - Concepts [online]. Kubernetes Authors, 2020 [cit. 2023-05-18]. Dostupné z: https://kubernetes.io/docs/concepts/

[18] Airports. Github [online]. Github, 2023, 25.11.2014 [cit. 2023-05-09]. Dostupné z: https://github.com/mwgg/Airports

[19] KELVIN, Lawrence. Practical Gremlin - An Apache TinkerPop Tutorial. Github [online]. San Francisco: Github, 2017 [cit. 2023-05-10]. Dostupné z: https://github.com/krlawrence/graph

[20] Docker Documentation: How to build, share, and run applications [online]. Palo Alto: Docker [cit. 2023-05-20]. Dostupné z: https://docs.docker.com

[21] LocalStack Documentation [online]. Zürich: LocalStack, 2022 [cit. 2023-05-01]. Dostupné z: https://docs.localstack.cloud/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
CI	Continuous Integration
CIDR	Classless Inter-Domain Routing
CD	Continuous Delivery/Deployment
CPU	Central Processing Unit
gRPC	Google Remote Procedure Call
ECR	Elastic Container Registry
HCL	Hashicorp Configuration Language
HTTP	HyperText Transfer Protocol
IaC	Infrastructure as Code
IATA	International Air Transport Association
ICAO	International Civil Aviation Organisation
IP	Internet Protocol
IT	Information Technology
JSON	Javascript Object Notation
JWT	JSON Web Token
RDS	Relational Database Service
REST	Representational State Transfer
RPC	Remote Procedure Call
SDK	Software Development Kit
SSL	Secure Socket Layer
SQL	Structured Query Language
URL	Universal Resource Locator
VPC	Virtual Private Cloud
YAML	Yet Another Markup Language

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I. ZDROJOVÝ KÓD DEMONŠTRAČNEJ APLIKÁCIE NA VYHĽADÁVANIE LETENIEK

Elektronická príloha obsahuje všetky zdrojové kódy jednotlivých služieb, ktoré sú potrebné na zostavenie aplikácie na vyhľadávanie letniek opísanej v tejto práci.