Návrh aplikace pro sdílení výletů a ubytování s využitím mikroslužeb

Bc. Ondrej Mičina

Diplomová práca
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. Ondrej Mičina**
Osobní číslo:	**A21161**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Návrh aplikace pro sdílení výletů a ubytování s využitím mikroslužeb**
Téma práce anglicky:	**Design of an Application for Tour and Accommodation Sharing with the Use of Microservices**

Zásady pro vypracování

1. Popište současný stav technologií mikroslužeb.
2. Popište rozdíly mezi monolitickým řešením aplikace a řešením s využitím mikroslužeb.
3. Vypracujte návrh aplikace pro sdílení výletů a ubytování s využitím mikroslužeb.
4. Vytvořte klíčové části řešení a tato řešení v práci popište.
5. Zhodnoťte dosažené výsledky a možnosti dalšího rozvoje aplikace.

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

Seznam doporučené literatury:

1. NEWMAN, Sam, 2019. Building Microservices: Designing Fine-Grained Systems. 2. O´Reilly. ISBN 9781492034025.
2. C# 10 and .NET 6: Modern Cross-Platform Development, 2021. Sixth edition. Packt Publishing. ISBN ISBN 978-1--80107-736-1.
3. PUJARINI MOHAPATRA, Biswa, Baishakhi BANERJEE a Gaurav ARORAA, 2019. Microservices by Example Using .Net Core. India: BPB Publications. ISBN 978-93-87284-58-6.
4. WHITESELL, Sean, Rob RICHARDSON a Matthew D. GROVES, 2022. Pro Microservices in .NET 6:: With Examples Using ASP.NET Core 6, MassTransit, and Kubernetes. Apress. ISBN 978-1484278321.
5. J. PRICE, Mark, 2022. C# 11 and .NET 7: Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 7, Blazor, and EF Core 7, 7th Edition. 7th ed. Edition. Packt Publishing. ISBN 978--1803237800.
6. MARTIN, Robert C., 2008. Clean code: a handbook of agile software craftmanship. Pearson: Prentice Hall. Robert C. Martin series. ISBN 978-013-2350-884.

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

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 diplomová 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 UTB 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

Ondrej Mičina,v.r.
podpis studenta

ABSTRAKT

Táto práca prispieva k lepšiemu porozumeniu softvérovej architektúry založenej na mikroslužbách a poskytuje konkrétny príklad implementácie aplikácie. Teoretická časť popisuje súčasný stav technológií mikroslužieb a popisuje rozdiely medzi monolitickými aplikačnými riešeniami a riešeniami s využitím mikroslužieb. Praktickú časť tvorí návrh, implementácia aplikačného riešenia a popis kľúčových častí vývoja.

Kľúčové slová: mikroslužby, architektúra, software, návrh, architektúra mikroslužieb, monolit, API, REST API, .NET, C#, Docker, Kubernetes, Postman, kontajnerizácia, orchestrácia, autentifikácia, Jwt, MongoDB

ABSTRACT

This work contributes to a better understanding of microservices-based software architecture and provides a specific example of application implementation. The theoretical part describes the current state of microservices technologies and outlines the differences between monolithic application solutions and solutions using microservices. The practical part consists of the design, implementation of the application solution, and a description of key development components.

Keywords: microservices, architecture, software, design, microservices architecture, monolith, API, REST API, .NET, C#, Docker, Kubernetes, Postman, containerization, orchestration, authentication, Jwt, MongoDB

Moje poďakovanie patrí Ing. et Ing. Erik Král, Ph.D. za to že ma prijal ako diplomanta, za odborné rady a bezproblémové vedenie mojej práce. Ďakujem všetkým mojim najbližším, ktorí ma aj napriek ťažkým životným situáciám vedeli motivovať a podporovať nielen počas posledných mesiacov ale počas celého môjho štúdia.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

V súčasnom technologickom prostredí zohráva architektúra mikroslužieb kľúčovú úlohu pri vývoji a nasadení softvérových aplikácií. Tento prístup k návrhu a implementácii softvéru prináša mnoho výhod a umožňuje organizáciám efektívne reagovať na dynamické potreby trhu a používateľov.

Teoretická časť práce sa venuje vysvetleniu rôznych typov architektúr, ako je monolitická architektúra, objektovo orientovaná architektúra a architektúra orientovaná na služby. V ďalších častiach sa práca zameriava výhradne na mikroslužby, kde vysvetľuje ich definíciu a základné koncepty. Ďalej porovnáva rozdiely medzi monolitickým riešením a riešením s využitím mikroslužieb v oblasti škálovateľnosti, vývoja a nasadzovania, nasadenia, technologickej flexibility, izolácie porúch, odolnosti a bezpečnosti.

Praktická časť sa zaoberá návrhom a implementáciou aplikácie pre zdieľanie výletov a ubytovania s využitím mikroslužieb ktorá bude slúžiť na demonštráciu všetkých úkonov ktoré patria k vývoju takéhoto typu aplikácie. V tejto časti je predstavený koncept a požiadavky aplikácie, návrh riešenia a použitých technológií. Ďalej sa popisuje postup implementácie návrhu, vytvorenie projektu a implementácia jednotlivých služieb vrátane autentifikácie a komunikácie medzi službami. Okrem toho sa táto časť zaoberá aj kontajnerizáciou a orchestráciou služieb pomocou platforiem Docker a Kubernetes.

Na záver práce je zhodnotenie a možnosti ďalšieho rozvoja, kde sa diskutuje o voľbe medzi mikroslužbami a monolitom, nových požiadavkách aplikácie, použití API Gateway a možnosti využitia asynchrónnej komunikácie.

I. TEORETICKÁ ČASŤ

1 SOFTVÉROVÁ ARCHITEKTÚRA

1.1 Úvod

Pojem softvérová architektúra sa dostal do popredia v 80. rokoch, kedy došlo k obrovskému nárastu záujmu výskumnej komunity o návrh softvéru a jeho vplyv na proces vývoja.

S narastajúcou náročnosťou softvérových riešení, narastá aj nespočetné množstvo problémov, nedorozumení a chýb ktoré môžu viesť ku kolapsu, a hrozí že takýto projekt nebude mať dlhú životnosť alebo sa vôbec nedokončí.

Tak ako každá iná zložitá štruktúra, aj softvér musí byť postavený na pevných základoch.

Softvérová architektúra definuje schému alebo plán ktorého účelom je uľahčiť vývoj, nasadenie, prevádzku a údržbu daného softvérového systému. Forma tohto plánu zobrazuje systém rozdelený na komponenty, usporiadanie týchto komponentov a spôsoby, akými tieto komponenty navzájom komunikujú.

Softvérová architektúra sa snaží vybudovať most medzi obchodnými požiadavkami a technickými požiadavkami tým, že detailne analyzuje prípady použitia a potom nájde spôsoby, ako tieto prípady použitia implementovať do softvéru.

Dobrá architektúra znižuje podnikateľské riziká spojené s budovaním technického riešenia. Dobrý dizajn je dostatočne flexibilný na to, aby dokázal zvládnuť prirodzený posun, ktorý sa časom vyskytne v hardvérovej a softvérovej technológii, ako aj v používateľských scenároch a požiadavkách. [1]

Medzi riziká, ktorým sa vystavuje zlá architektúra, patrí softvér, ktorý je nestabilný, nedokáže podporovať existujúce alebo budúce obchodné požiadavky alebo sa ťažko nasadzuje či spravuje v produkčnom prostredí.

Štýly architektúry možno rozdeliť na dva hlavné typy: monolitickú (jediná jednotka nasadenia celého kódu) a distribuovanú (viacero jednotiek nasadenia prepojených prostredníctvom protokolov vzdialeného prístupu). Hoci žiadna klasifikačná schéma nie je dokonalá, všetky distribuované architektúry majú spoločný súbor výziev a problémov, ktoré sa nenachádzajú v monolitických štýloch architektúry, vďaka čomu táto klasifikačná schéma dobre oddeľuje rôzne štýly architektúry. [2]

1.2 Monolitická architektúra

O monolitickej architektúre aplikácie hovoríme vtedy ak je aplikácia samostatná a nezávislá od iných počítačových aplikácií. Takáto aplikácia zahŕňa všetky časti kódu úzko prepojené v jednej samostatnej kódovej základni. V praxi to znamená, že spoločne súvisiace metódy, sú implementované v jednej alebo viacerých vrstvách, ktoré sú zapuzdrené do jedného bloku.[3]

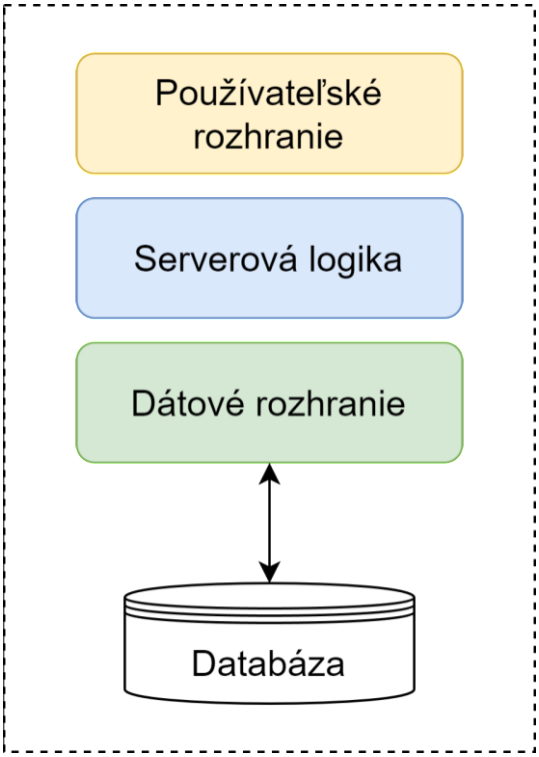
Monolitický prístup sa považuje za štandardný spôsob, ako začať vyvíjať aplikácie a nesie so sebou niekoľko výhod. [4]

Monolitické aplikácie sa vyvíjajú jednoducho a rýchlo, pretože vývojové prostredia IDE a iné vývojové nástroje sú orientované na vývoj práve jednej aplikácie a teda nemusíme riešiť komplexné interakcie medzi rôznymi časťami/funkciami aplikácie. Taktiež sa ľahko sa testujú, keďže počas procesu hľadania chýb máme spustenú iba jednu aplikáciu. Monolitické aplikácie je možné jednoducho nasadiť. Stačí nahrať jednotku nasadenia - súbor alebo adresár - do počítača, na ktorom beží príslušný druh servera.[5]

Jednou z najväčších výhod monolitickej architektúry je rýchlosť a výkonnosť aplikácie. Tento typ architektúry umožňuje jednoduché a rýchle načítanie aplikácie, pretože všetky jej časti sú uložené na jednom mieste. Komunikácia prebieha na " krátke vzdialenosti " v rámci jednej spustenej inštancie a z tohto dôvodu môže aplikácia veľmi rýchlo reagovať na požiadavky používateľov a poskytnúť im takmer okamžitú spätnú väzbu. Tieto vlastnosti sa využívajú v bankových systémoch kedy celý proces kvôli bezpečnosti a jednoduchosti zastrešuje práve jedna služba.

Monolitická aplikácia používa jednu jedinú databázu na spracovanie všetkých údajov. Databázu síce možno škálovať na rôzne oddiely pomocou shardingu, ale aj tak tieto oddiely používajú rovnakú schému. S jednou databázou je práca s transakciami ľahko zvládnuteľná pretože väčšina databázových systémov poskytuje transakcie typu ACID (Atomicity, Consistency, Isolation, Durability). Vývojári sú schopný tieto transakcie ľahko definovať a môžu sa viac sústrediť na poskytovanie nových funkcií koncovým používateľom. Na druhej strane má jedna databáza aj svoje obmedzenia. Monolitická aplikácia môže mať viacero rôznych druhov údajov. Niektoré z údajov by mohli byť vhodnejšie na uloženie v databáze NoSQL a niektoré z údajov v relačnej databáze. Pri monolitickom prístupe si však vývojári zvyčajne musia vybrať len jeden databázový systém a ten používať pre všetky druhy údajov. [6]

Väčšina monolitických aplikácií môže byť na začiatku pomerne jednoduchá, ale s rastom aplikácie rastie aj jej zložitosť. Typickým spôsobom, ako sa vysporiadať so zložitosťou aplikácie, ktorá má monolitickú architektúru, je rozdeliť aplikáciu na rôzne vrstvy. Takto rozdelená aplikácia sa skladá z vrstvy používateľského rozhrania, vrstvy serverovej logiky alebo služieb a vrstvy dátového rozhrania. Vrstva prístupu k údajom potom zvyčajne pristupuje k práve k jednej databáze, ktorá spracúva všetky údaje, ktoré sa týkajú tejto aplikácie.



Obrázok 1. Vrstvy monolitickej architektúry

S rastúcou veľkosťou aplikácie však rastú aj jednotlivé vrstvy. Ak sa architektúre a kvalite kódovej základne nevenuje veľká pozornosť, je veľmi pravdepodobné, že kvalita jednotlivých vrstiev sa zhorší. K zhoršeniu dochádza predovšetkým v dôsledku zmeny obchodných požiadaviek, ktoré nútia vývojárov vytvárať aj riešenia, ktoré nie sú optimálne. Tieto neoptimálne riešenia by sa mali refaktorovať, ale čas potrebný na refaktorovanie sa dá získať obtiažne, čo vedie k tomu, že z krátkodobých riešení sa stávajú dlhodobé riešenia. S rastúcou veľkosťou kódovej základne a zhoršujúcou sa kvalitou kódu je náročné pridávať nové funkcie a upravovať tie súčasné.[7] Vývojári musia vynaložiť veľa síl pri aplikovaní týchto zmien a musia rátať s vysokou pravdepodobnosťou znefunkčnenia iných častí aplikácie.

Veľká monolitická aplikácia môže byť pre vývojárov náročná nielen na pochopenie ale aj na údržbu. Ďalšou prekážkou môže byť aj časté nasadzovanie aplikácie. Ak vykonáme zmeny jednej funkcie aplikácie (napr. funkciu platieb), musíme zostaviť a nasadiť celý monolit. Vykonanie tohto úkonu môže byť zložité, riskantné, časovo náročné, vyžaduje si koordináciu mnohých vývojárov a vedie k dlhým testovacím cyklom.

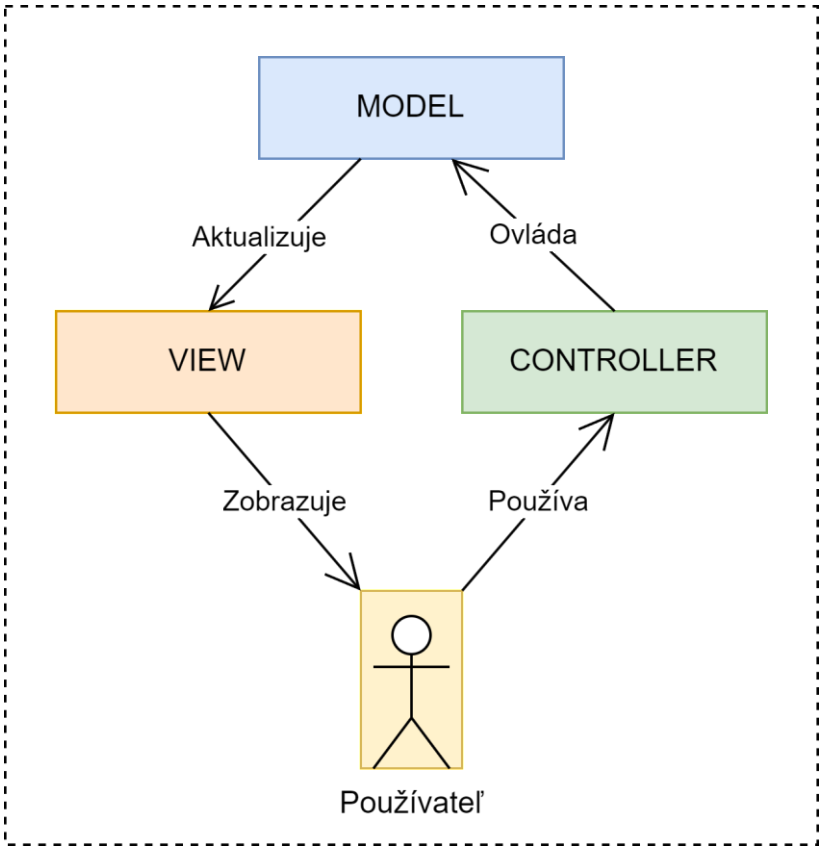
Monolitická architektúra tiež sťažuje skúšanie a prijímanie nových technológií. Je napríklad ťažké vyskúšať nový infraštruktúrny rámec bez prepísania celej aplikácie, čo je riskantné a nepraktické. V dôsledku toho často zostávame pri technologických rozhodnutiach, ktoré ste urobili na začiatku projektu. Inými slovami, monolitická architektúra sa neškáluje tak, aby podporovala veľké aplikácie s dlhou životnosťou. [5]

1.3 Objektovo orientovaná architektúra

Na rozdiel od monolitickej architektúry, objektovo orientovaná architektúra sa tak ako to z názvu vyplýva orientuje na objekty. Tento prístup k návrhu softvéru využíva objekty ako základné stavebné kamene pre všetky druhy softvérových aplikácií s touto architektúrou. Objektovo orientovaná architektúra (OOA) hovorí že štruktúru a správanie každej softvérovej aplikácie možno reprezentovať pomocou viacerých a vzájomne spolupracujúcich objektov. [8]

Objekt možno definovať ako pole dátových prvkov a príslušných operácií, ktoré môžu byť vykonávané na týchto dátach. [10] Objekty reprezentujú skutočné entity ako napríklad človek ktorý je opísaný vlastnosťami(meno, adresa, vek) a procesy alebo malé programy, ako sú widgety v aplikácii. Objekty sa spájajú, komunikujú a spolupracujú prostredníctvom vhodne implementovaných rozhraní. Z tohto dôvodu sa OOA stala dominantným štýlom pri tvorbe objektovo orientovaných softvérových aplikácií. Softvérový systém sa v konečnom dôsledku vníma ako dynamická kolekcia spolupracujúcich objektov namiesto súboru rutín alebo procedurálnych inštrukcií. [8]

OOA využíva princípy objektovo orientovaného programovania (OOP) a ponúka rôzne techniky a prístupy, ktoré programátorom zlepšujú organizáciu a údržbu zložitých systémov. Medzi najznámejšie prístupy patria architektonické návrhové vzory. Model-View-Controller (MVC) je hlavným príkladom takýchto vzorov, ktorý programátori často implementujú aj pre rozsiahle aplikácie. [9]



Obrázok 2. Princíp MVC

1.3.1 Kľúčové vlastnosti objektovo orientovaného programovania

- **Zapuzdrenie**. Tento princíp hovorí, že všetky dôležité informácie sú obsiahnuté vo vnútri objektu a vystavené sú len vybrané informácie. Implementácia a stav každého objektu sú súkromne uchovávané vo vnútri definovanej triedy. Iné objekty nemajú prístup k tejto triede ani oprávnenie vykonávať zmeny. Môžu len volať zoznam verejných funkcií alebo metód. Táto vlastnosť skrývania údajov poskytuje väčšiu bezpečnosť programu a zabraňuje neúmyselnému poškodeniu údajov.
- **Abstrakcia**. Objekty odhaľujú iba vnútorné mechanizmy, ktoré sú dôležité pre použitie iných objektov, čím sa skrýva akýkoľvek nepotrebný implementačný kód. Odvodená trieda môže mať rozšírenú svoju funkcionalitu. Tento koncept môže vývojárom pomôcť ľahšie vykonávať ďalšie zmeny alebo doplnenia v priebehu času.
- **Dedičnosť**. Triedy môžu opätovne používať kód z iných tried. Medzi objektmi možno priradiť vzťahy a podtriedy, čo umožňuje vývojárom opätovne použiť spoločnú logiku a zároveň zachovať jedinečnú hierarchiu. Táto vlastnosť OOP si

vynucuje dôkladnejšiu analýzu údajov, skracuje čas vývoja a zabezpečuje vyššiu úroveň presnosti.

- **Polymorfizmus**. Objekty sú navrhnuté tak, aby zdieľali správanie a mohli nadobúdať viac ako jednu formu. Program určí, ktorý význam alebo použitie je potrebné pre každé vykonanie daného objektu z nadradenej triedy, čím sa znižuje potreba duplikovať kód. Potom sa vytvorí podriadená trieda, ktorá rozširuje funkčnosť rodičovskej triedy. Polymorfizmus umožňuje rôznym typom objektov prechádzať cez rovnaké rozhranie [10]

1.4 Architektúra orientovaná na služby

Akonáhle monolitické aplikácie narastú do obrovských rozmerov, úkony ako údržba alebo pridávanie nových funkcií sa stávajú čoraz viac náročné. Aj napriek rozdeleniu monolitu na vrstvy, inovácii hardvéru a podobným zmenám v štruktúre nemusí takýto systém zvládať obrovský nápor požiadaviek. Vývojový tým prestáva byť schopný udržiavať všetky časti a vzniká čoraz viac chýb. Tieto problémy rieši architektúra orientovaná na služby.

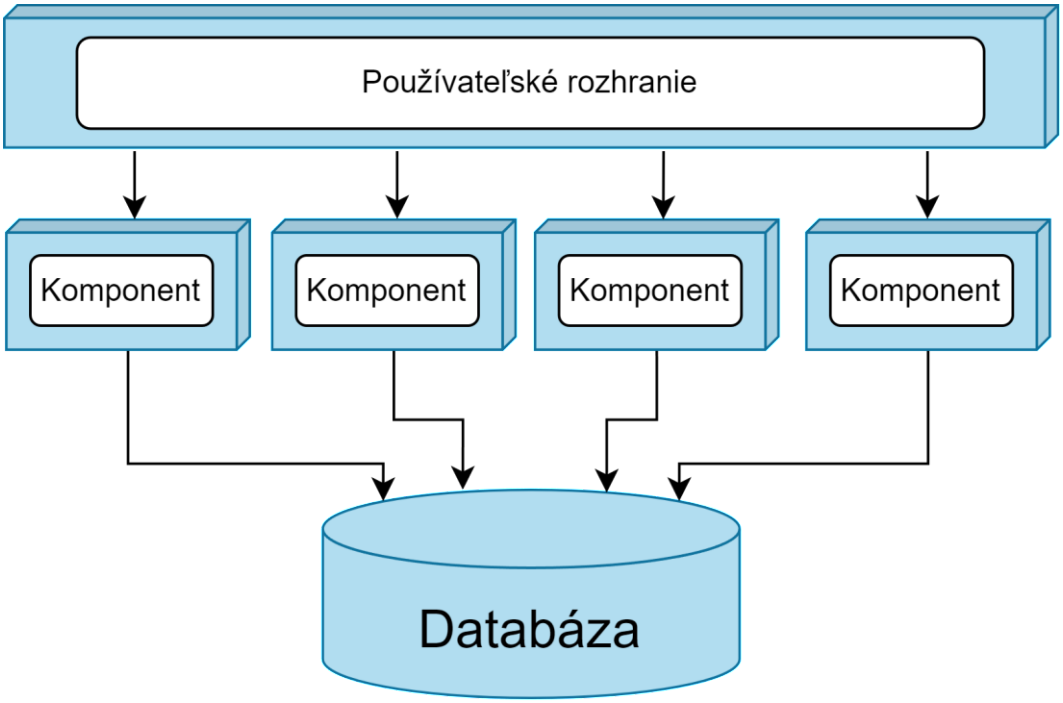
Monolitická architektúra oddeľuje spoločne súvisiace metódy do vrstiev v rámci jedného bloku. Architektúra orientovaná na služby (SOA) súvisiace metódy alebo komponenty aplikácie rozdeľuje na samostatné služby. Tieto služby reprezentujú väčšie časti aplikácie (zvyčajne nazývané doménové služby) a odrážajú reálne činnosti ako napríklad vykonanie platby, registrácia a podobne. [11] Služby fungujú nezávisle od iných služieb a môžu sa nasadzovať rovnakým spôsobom ako akékoľvek monolitické aplikácie a preto nevyžadujú kontajnerizáciu. [2]

Tento prístup dovoľuje vývojárom rozdeliť sa do špecifických tímov a orientovať sa na prácu na špecifických funkciách aplikácie. V prípade potreby vykonania zmeny alebo aktualizácie nie je nutné zastavovať celú aplikáciu ale len jednotlivú službu ktorej sa zmena dotýka.

Služby sú schopné fungovať nezávisle od základnej technológie. Rovnako sa služby dajú implementovať pomocou ľubovoľných programovacích a skriptovacích jazykov.

Jedným z dôležitých aspektov architektúry založenej na službách je, že obvykle používa centrálne zdieľanú databázu. To umožňuje službám využívať výrazy SQL a spojenia rovnakým spôsobom ako tradičná monolitická vrstvená architektúra. Vzhľadom na malý počet služieb (väčšinou 4 až 12) nie sú databázové pripojenia v architektúre založenej na

službách vo väčšine prípadoch problémom. Čo však môže spôsobovať problémy sú zmeny štruktúry alebo schémy databázy. Ak sa zmena schémy tabuľky nevykoná správne, môže potenciálne ovplyvniť každú službu, čím sa zmeny databázy stávajú veľmi nákladnou úlohou z hľadiska úsilia a koordinácie. [2]



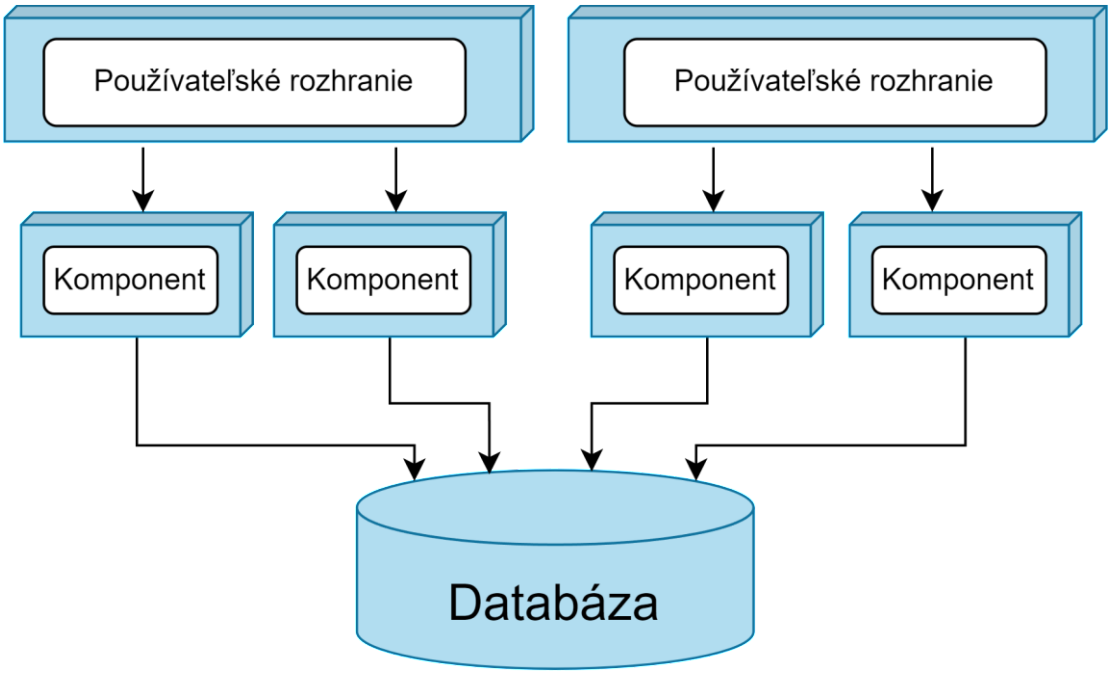
Obrázok 3. Topológia architektúry SOA s jednou centrálnou databázou

K službám sa pristupuje vzdialene z používateľského rozhrania pomocou protokolu vzdialeného prístupu. Hoci sa na prístup k službám z používateľského rozhrania zvyčajne používa REST, môže sa použiť aj posielanie správ, vzdialené volanie procedúr (RPC) alebo dokonca SOAP. [2,11]

Vo väčšine prípadov je v rámci SOA iba jedna inštancia každej doménovej služby. Na základe škálovateľnosti, odolnosti voči chybám a potrieb priepustnosti však určite môže existovať viac inštancií doménovej služby. Viacnásobné inštancie služby si zvyčajne vyžadujú určitý druh schopnosti vyrovnávania záťaže medzi používateľským rozhraním a doménovou službou, aby sa používateľské rozhranie mohlo presmerovať na zdravú a dostupnú inštanciu služby. [2]

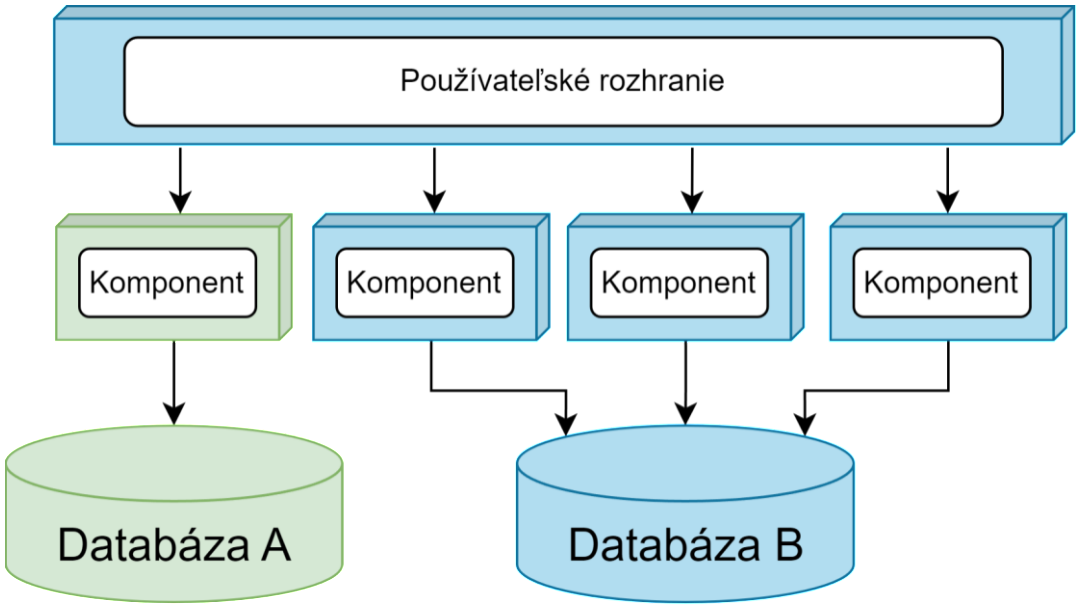
V rámci SOA existuje mnoho variantov topológie, čo z nej robí pravdepodobne jeden z najflexibilnejších štýlov architektúry. Napríklad jediné monolitické používateľské rozhranie, ako je znázornené na obrázku č.3, možno rozdeliť na domény používateľského

rozhrania, a to dokonca na úroveň zodpovedajúcu jednotlivým doménovým službám. Tieto varianty používateľského rozhrania sú znázornené na obrázku č.4. [2]



Obrázok 4. Používateľské rozhranie architektúry SOA rozdelené na domény

Podobne môžu existovať možnosti rozdeliť jedinú monolitickú databázu na samostatné databázy, dokonca až na úroveň databáz zodpovedajúcich každej doménovej službe (podobne ako pri mikroslužbách). V týchto prípadoch je dôležité zabezpečiť, aby údaje v každej samostatnej databáze nepotrebovala iná doménová služba. Tým sa zabráni komunikácii medzi doménovými službami (čomu sa pri architektúre založenej na službách rozhodne treba vyhnúť) a tiež duplicite údajov medzi databázami. Tieto varianty databáz sú znázornené na obrázku č.5. [2]



Obrázok 5. Architektúra SOA s dvoma databázami

2 MIKROSLUŽBY

2.1 Úvod k mikroslužbám

S neustálym pridávaním funkcií do našej aplikácie sa rozrastá aj samotná kódová štruktúra programu. Po určitej dobe môžeme dospieť k tomu že sa nebudeme vedieť v tejto štruktúre zorientovať. Stratíme priveľa času tým že nebudeme vedieť zistiť kde sa v takomto kóde nachádza chyba alebo kde presne je potrebné vykonať požadovanú zmenu.

Aj keď sa v priebehu vývoja snažíme o prehľadnú monolitickú štruktúru, takmer vždy sme nútený pristúpiť k rozdeleniu určitých častí a túto štruktúru narušíme. Kód týkajúci sa podobných funkcií je často duplikovaný s minimálnymi úpravami. Kódová štruktúra začne byť preplnená, čoho dôsledkom je že oprava chýb alebo implementácia novej funkcionality začne byť veľmi náročná.

V rámci monolitického systému riešime tieto problémy tým, že sa snažíme zabezpečiť väčšiu súdržnosť nášho kódu vytváraním abstrakcií, modulov alebo vrstiev. Súdržnosť - snaha o zoskupenie súvisiaceho kódu - je dôležitým konceptom, keď uvažujeme o mikroslužbách. [11]

Mikroslužby sú v súčasnosti novým trendom v softvérovej architektúre, ktorý kladie dôraz na návrh a vývoj vysoko udržiavateľného a škálovateľného softvéru. Mikroslužby zvládajú rastúcu zložitosť pomocou rozloženia veľkých systémov na súbor nezávislých služieb. Vďaka kompletnej nezávislosti služieb pri vývoji a nasadzovaní aplikácie, mikroslužby kladú dôraz na voľné prepojenie a vysokú súdržnosť tým, že posúvajú modulárnosť na ďalšiu úroveň. Tento prístup prináša najrôznejšie výhody z hľadiska udržiavania, škálovateľnosti a podobne. Prináša však aj kopec problémov, ktoré sú zdedené od distribuovaných systémov a priameho predchodcu - SOA. [12]

2.2 Čo sú to mikroslužby

Mikroslužby sú architektonický vzor ktorý ponúka lepší spôsob budovania na oddelenie komponentov v rámci hranice aplikácie. Architektúra mikroslužieb je vzor na vývoj aplikácie postavenej viacerých menších služieb. Mikroslužby sú navzájom nezávislé a bežia vo vlastných procesoch. Každá z mikroslužieb má svoj vlastný tím, ktorý na nej pracuje, takže sú od seba úplne oddelené. To umožňuje, aby každá služba spustila svoj vlastný

jedinečný proces a komunikovala autonómne bez toho, aby sa musela spoliehať na iné služby alebo na aplikáciu ako celok. [13,14]

2.2.1 Mikroslužby sú modulárne a nezávislé

Mikroslužby sú voľne previazané. To znamená, že každá služba je malá a navrhnutá na riešenie konkrétnej aplikačnej biznis funkcie. Mikroslužby sú podľa návrhu rozdelené na viacero komponentov služieb, ktoré môže vyvíjať malý vývojový tím, takže každú zo služieb možno vyvíjať a nasadzovať nezávisle bez toho, aby bola ohrozená integrita aplikácie. V prístupe založenom na architektúre mikroslužieb každá mikroslužba vlastní svoj proces a dáta, čo ju robí nezávislou z hľadiska vývoja a nasadenia. [13]

2.2.2 Mikroslužby sú decentralizované a multifunkčné

Ideálna organizácia pre mikroslužby má malý, angažovaný tím, kde každý tím má na starosti biznis funkciu zloženú z rôznych mikroslužieb, ktoré môžu byť nasadené nezávisle. Tímy majú na starosti všetky časti vývoja svojej mikroslužby, od vývoja až po nasadenie, a teda všetky úrovne členov tímu potrebných na poskytovanie mikroslužieb od vývojárov, inžinierov kvality, DevOps tímu a produktových architektov. [13]

2.2.3 Mikroslužby sú odolné

Aplikácie vyvinuté ako mikroslužby sú lepšie z hľadiska izolácie porúch, ak jedna služba zlyhá, ostatné budú fungovať ďalej. To je jedna z výhod budovania distribuovaných systémov ako mikroslužieb, teda schopnosť systému odolávať chybám a neočakávaným zlyhaniam prvkov, sietí, počítačových zdrojov atď. Tieto systémy sú odolné aj v rámci chýb. Koncept tejto odolnosti vyzerá jednoducho: ak naša monolitická aplikácia zlyhá, zlyhá po jej boku všetko, čo za ňu zodpovedá; rozdeľme preto veci na menšie moduly, aby sme odolali zlyhaniu jednotlivých častí našej aplikácie bez toho, aby to ovplyvnilo celý systém. Riešenie izolácie porúch v distribučnom systéme nie je veľmi jednoduché čo je dôvodom prečo mikroslužby potrebujú orchestrátor na vysokú dostupnosť. Na orchestráciu mikroslužieb je nevyhnutné vybrať lepšiu infraštruktúru cloud computingu pre mikroslužby, pretože jednotlivé časti aplikácie sú oddelené. Problém sa môže vyskytnúť v jednej časti bez toho, aby ovplyvnil ostatné oblasti aplikácie. [13]

2.2.4 Mikroslužby sú vysoko škálovateľné

Škálovateľnosť je najdôležitejšia pre každý distribuovaný systém a mikroslužby sú navrhnuté na škálovanie. Škálovanie umožňuje aplikáciám reagovať na premenlivé zaťaženie zvyšovaním a znižovaním počtu inštancií rolí, frontov a alternatívnych služieb, ktoré používajú. Je ľahké kvantifikovať efektívnosť monolitickej aplikácie, ale hodnotenie a kvantifikácia schopnosti širokého ekosystému mikroslužieb sú pomerne zložité, pretože mikroslužby sú rozdelené na tisíce malých služieb. [13]

2.3 Komunikácia v mikroslužbách

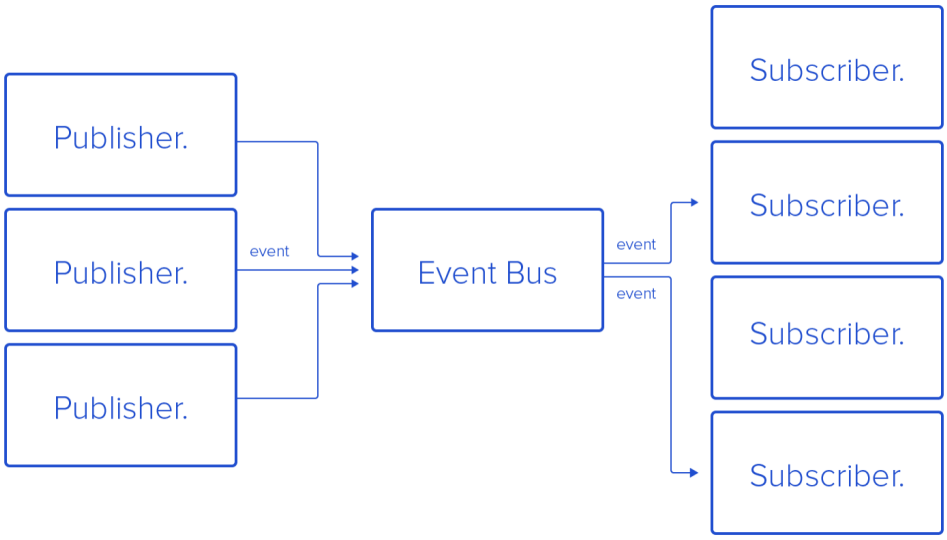
Jedným z najvýznamnejších aspektov používania mikroslužieb namiesto monolitickej aplikácie je komunikácia medzi službami. V prípade monolitickej aplikácie, ktorá beží na jednom procese, sú požiadavky medzi komponentami získané prostredníctvom volaní metód na úrovni jazyka. Aplikácia založená na mikroslužbách je distribuovaný systém, ktorý funguje na viacerých procesoch alebo službách, často dokonca na viacerých serveroch alebo hostiteľoch. Každá inštancia služby je samostatný proces. Z tohto dôvodu musia služby komunikovať pomocou protokolov interprocesovej komunikácie, ako je napríklad HTTP, AMQP alebo binárny protokol ako TCP, v závislosti na charakteristikách jednotlivých služieb. Najčastejšie používané protokoly sú synchrónne HTTP požiadavky/odpovede s využitím rozhraní na zdroje a asynchrónna správa pri komunikácii medzi viacerými mikroslužbami. [13]

Klient a služby môžu komunikovať prostredníctvom niekoľkých typov komunikácie, z ktorých každý je zameraný na samostatný scenár a ciele. Tieto typy komunikácie možno kategorizovať do dvoch typov:

- **Synchrónny protokol**: Klient zvyčajne odošle požiadavku službe a čaká na odpoveď – jedná sa o cyklus požiadavka/odpoveď. Primárne ide o to, že protokol (HTTP/HTTPS) je synchrónny a úloha na strane klienta môže pokračovať až vtedy, keď dostane odpoveď servera HTTP. Na komunikáciu webových aplikácií je použitý štandardný protokol HTTP a rovnako to platí aj v prípade mikroslužieb. Pri synchrónnej komunikácii klient vysiela požiadavku a čaká na odpoveď. Zaujímavé je, že pomocou toho istého protokolu môže klient komunikovať so serverom aj asynchrónne, čo znamená, že vlákno nie je blokované a odpoveď nakoniec dostane spätné volanie.[13] Nevýhodou tohto spôsobu komunikácie je, že je potrebné

jednotlivé mikroslužby previazať čo v prípade veľkého projektu môže viesť k zvýšeniu doby čakania.

- **Asynchrónny protokol:** Iné protokoly, ako napríklad AMQP (odporúčaný mnohými OS a cloudovými prostrediami), používajú asynchrónne správy. Pri asynchrónnej komunikácii volajúci nečaká na dokončenie operácie pred jej návratom a môže mu byť dokonca jedno, či sa operácia dokončí. V tomto spôsobe nefiguruje cyklus požiadavka/odpoveď. Odosielateľ odosiela správy do frontu správ alebo iného sprostredkovateľa správ. Tento typ komunikácie je využitý aj v mechanizme publish/subscribe používaným vo vzoroch, ako je architektúra riadená udalosťami. Jednotlivé služby sa prihlásia na odber správ z fronty a reagujú len na správy ktoré ich zaujímajú. [11,13]



Obrázok 6. Využitie Event Bus [15]

3 ZÁKLADNÉ KONCEPTY MIKROSLUŽIEB

3.1 Zameranie na jeden účel

Veľká monolitická aplikácia môže mať desiatky miliónov riadkov kódu a vykonávať stovky jednotlivých obchodných funkcií. Jedna aplikácia môže napríklad obsahovať kód na spracovanie produktov, zásob, cien, propagačných akcií, nákupných košíkov, objednávok, profilov atď.

Na druhej strane mikroslužby vykonávajú práve jednu obchodnú funkciu čo umožňuje tímom zostať sústredenými a zložitosť zostáva minimálna. Je úplne prirodzené, že aplikácie časom naberajú čoraz viac funkcií. To, čo začína ako mikroslužba s 2 000 riadkami, sa môže vyvinúť na mikroslužbu s viac ako 10 000 riadkami, pretože tím si buduje kompetencie a podnik sa vyvíja. Veľkosť kódovej základne nie je taká dôležitá ako veľkosť tímu zodpovedného za danú mikroslužbu, pretože mnohé výhody mikroslužieb vyplývajú zo spolupráce v malom úzkom tíme (2 až 15 ľudí). Ak počet ľudí pracujúcich na mikroslužbe presiahne 15, táto mikroslužba sa pravdepodobne snaží robiť príliš veľa vecí a mala by sa rozdeliť.

Okrem veľkosti tímu je ďalším rýchlym testom pozrieť sa na názov mikroslužby. Názov mikroslužby by mal presne opisovať, čomu sa mikroslužba venuje. Napríklad mikroslužba by sa mala volať „Cenotvorba“ alebo „Inventár“ - nie „Cenotvorba_a_inventár“. [16]

3.2 Zapuzdrenie

Každá mikroslužba by mala vlastniť výlučne svoje dáta. Každá mikroslužba by mala mať svoje vlastné dátové úložisko, úložisko vyrovnávacej pamäte a podobne. Žiadna iná mikroslužba ani systém by nikdy nemali obchádzať API mikroslužby a zapisovať priamo do dátového úložiska, vrstvy vyrovnávacej pamäte, súborového systému alebo čohokoľvek iného. Jediná interakcia mikroslužby so svetom by mala prebiehať výlučne prostredníctvom jasne definovaného API.

Cieľom mikroslužieb je znížiť vzájomnú závislosť alebo previazanosť. Ak mikroslužba vlastní svoje vlastné údaje, neexistuje žiadna previazanosť. Ak dve alebo viac mikroslužieb čítajú tie isté údaje, vzniká úzka previazanosť tam, kde predtým nebola.

Hoci je výhodné, aby každá mikroslužba mala svoje vlastné dátové úložisko, úložisko vyrovnávacej pamäte, úložný zväzok alebo iný mechanizmus ukladania údajov, nie je

nevyhnutné, aby každá mikroslužba poskytovala vlastnú inštanciu týchto vecí pre jedného používateľa. Napríklad môže byť rozumnejšie vytvoriť jednu veľkú databázu alebo iný systém, do ktorého môžu zapisovať všetky mikroslužby. Avšak pri tomto prístupe je dôležité zabezpečiť pevné rozdelenie medzi údajmi každej mikroslužby a taktiež aby každá mikroslužba vlastnila výlučne svoje údaje. Vhodným príkladom je spravovať jednu veľkú databázu so 100 schémami, a nie 100 databáz s jednou schémou. [16]

Nevýhodou zdieľania je previazanosť jednotlivých mikroslužieb ktoré vzniká pri zdieľaní jednej databázy. Dostupnosť takejto mikroslužby bude potom závisieť od dostupnosti databázy, ktorú spravuje niekto iný. Tím, ktorý spravuje zdieľanú databázu, ju možno bude musieť odstaviť kvôli plánovanej údržbe.

3.3 Vlastníctvo

Typická komerčná aplikácia na podnikovej úrovni má stovky zamestnancov, ktorí na nej pracujú. Nie je napríklad ničím výnimočným, ak veľkú monolitickú aplikáciu buduje 100 vývojárov backendu. Problémom tohto modelu je, že zamestnanci nemajú pocit, že niečo vlastnia. Jediný vývojár sa bude podieľať na tvorbe kódovej základne len jedným percentom. To spôsobuje, že každý jednotlivý vývojár nadobúda pocit že nič nevytvoril keďže jeho zmeny sa prejavia minimálne.

Tu nastáva problém kde vo veľkej monolitickej aplikácii - stovky zamestnancov konajúcich vo vlastnom záujme nakoniec monolitickú aplikáciu komplikujú a zvyšujú jej technický dlh. So zložitosťou a technickým dlhom sa musí vysporiadať každý, nielen jednotlivec, ktorý ho vytvoril.

Mikroslužby fungujú z veľkej časti práve vďaka vlastníctvu. Malý tím 2 až 15 ľudí vyvíja, nasadzuje a spravuje jednu mikroslužbu počas celého jej životného cyklu. Tento tím je skutočným vlastníkom mikroslužby. Vlastníctvo prináša úplne inú mentalitu. Vlastníci sa viac starajú, pretože majú dlhodobý záujem na tom, aby ich mikroslužba bola úspešná. Na rozdiel od veľkej monolitickej aplikácie vývojári vnímajú mikroslužbu ako jeden veľký celok. Predpokladajme, že tím má päť členov - troch vývojárov, jedného operátora a jedného obchodného analytika. V tomto prípade sa každý vývojár podieľa na tvorbe kódu 33 %. Každá osoba v tomto tíme prispieva podstatnou mierou a tento príspevok sa dá ľahko rozpoznať. Ak je mikroslužba v prevádzke 100 % času a funguje perfektne, tento tím si môže pripísať zásluhy. Podobne, ak mikroslužba nie je úspešná, je ľahké priradiť zodpovednosť.

Ako uvádza Kelly Goetsch v knihe Microservices for Modern Commerce [17], za jednu mikroslužbu by mal byť zodpovedný tím o veľkosti dva až pätnásť ľudí pričom ideálna veľkosť tímu je sedem +/- dvaja ľudia. Pri menšom obsadení (2-4) môžu nastať rozkoly medzi vývojármi ktorý si svojimi riešeniami môžu oponovať a zase pri príliš veľkom obsadení (10-15) sa môže stať že nezostane dostatok priestoru pre vyjadrenie všetkých a menšie skupiny sa môžu názorovo oddeľovať.

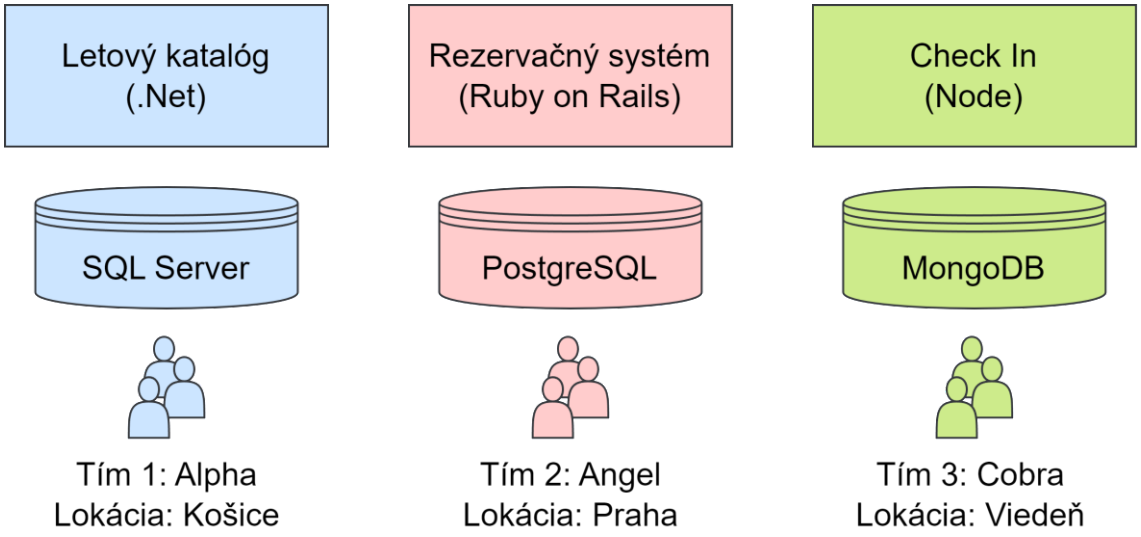
3.4 Autonómia

Vlastníctvo nemôže existovať bez autonómie. Autonómia môže v kontexte mikroslužieb znamenať veľa vecí.

Každý tím by mal mať možnosť vybrať si technológiu, ktorá je najvhodnejšia na riešenie konkrétneho obchodného problému - programovací jazyk, dátové úložisko a ďalšie podobné aspekty. Napríklad technológie použité v mikroslužbe, ktorá rieši zmenu veľkosti obrázkov produktov, budú úplne iné ako technológie pre mikroslužbu nákupného košíka. Každý tím by mal mať možnosť vybrať si technológie, ktoré najlepšie vyhovujú konkrétnym potrebám, a mal by byť za toto rozhodnutie zodpovedný. Keďže mikroslužba by mala sprístupniť len API, na implementačných detailoch by nemalo až tak záležať. Vzhľadom k tomu má pre podniky zmysel vyžadovať, aby si každý tím vybral z ponuky možností. Napríklad programovací jazyk môže byť Java, Scala alebo Node.js. Dátové úložisko môže byť MongoDB, Cassandra alebo MariaDB. Dôležité je, aby si každý tím mohol vybrať typ (relačný, dokumentový, key/value atď.) produktu, ktorý funguje najlepšie, ale nie nevyhnutne samotný produkt (MongoDB, Cassandra, MariaDB). Od tímov by sa malo vyžadovať, aby štandardizovali vonkajšie implementačné detaily, ako je protokol/formát API, zasielanie správ, logovanie, upozorňovanie atď. Interne používaná technológia by mala byť do veľkej miery na každom tíme.

Okrem výberu technológie by mal mať každý tím možnosť prijímať rozhodnutia o architektúre a implementácii, pokiaľ tieto rozhodnutia nie sú viditeľné mimo mikroslužby. Nikdy by nemala existovať celopodniková rada pre posudzovanie architektúry, ktorá schvaľuje architektúru každej mikroslužby. Preskúmanie kódu by mali vykonávať vývojári v rámci tímu - nie niekto zvonka. V lepšom či horšom prípade rozhodnutia o implementácii patria tímu, ktorý mikroslužbu vlastní a je za ňu zodpovedný. Ak mikroslužba nefunguje podľa očakávaní, je vhodné zapojiť do jej budovania nový tím.

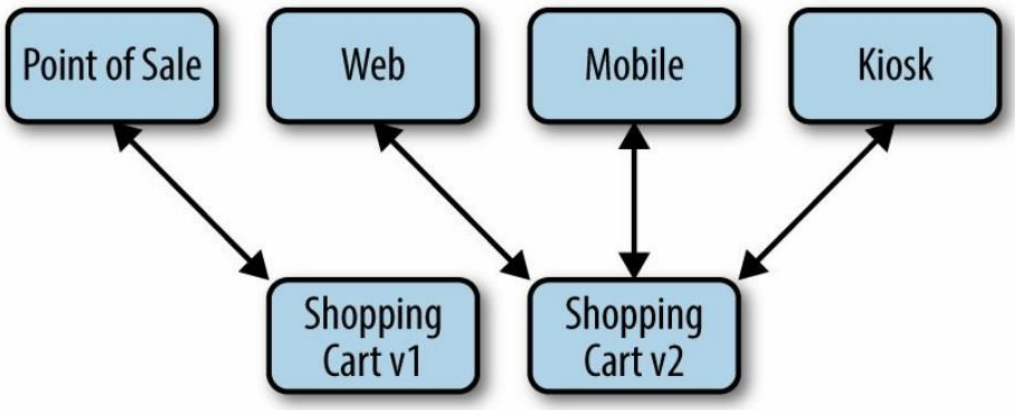
Každý tím by mal byť schopný vybudovať a spustiť svoju vlastnú mikroslužbu v úplnej izolácii, bez závislosti od iného tímu. Jeden tím by nemal budovať knižnicu, ktorú spotrebúva iný tím. Jedna mikroslužba by nemala potrebovať volať inú mikroslužbu, keď sa spúšťa alebo beží. Každá mikroslužba je nezávislá aplikácia, ktorá je vytvorená, nasadená a spravovaná izolovane. Každý tím by mal mať tiež možnosť kedykoľvek a z akéhokoľvek dôvodu zverejniť novú verziu mikroslužby naživo bez obmedzenia iných mikroslužieb.[17]



Obrázok 7. Rozdelenie tímov s rôznymi technológiami

3.5 Viacero verzií

Ďalšou charakteristickou vlastnosťou mikroslužieb je možnosť (ale nie povinnosť) nasadiť viac ako jednu verziu mikroslužby v tom istom prostredí súčasne. Napríklad verzie 2.2, 2.3, 2.4 a 3.0 katalógovej mikroslužby môžu byť v nasadené do produkcie súčasne. Všetky tieto verzie môžu obsluhovať prevádzku. Klienti a iné mikroslužby takto môžu pri požiadavke HTTP požadovať konkrétnu verziu mikroslužby. Často sa to vykonáva prostredníctvom adresy URL (napr. /inventory/2/ alebo /inventory?version=2). Možnosť nasadenia viac verzií sa využíva napríklad aj na vydávanie minimálnych životaschopných produktov (MVP). Zo začiatku môže verzia 1 poslúžiť ako prvotný produkt pre používateľov a presvedčiť ich aby ho začali používať a neskôr sa vydáme verziu 2 ako kompletnejší produkt. Ďalším využitím môže byť prípad kedy potrebujeme z nejakého dôvodu oddeliť vývoj jednotlivých verzií z dôvodu ako vidieť na nasledujúcom obrázku. [16]



Obrázok 8. Rozdielne vezie mikroslužby umožňuje nezávislý vývoj verzií [16]

V monolitickej architektúre je určitom čase v prostredí živá len jedna verzia monolitickej aplikácie. Ak by sme chceli v prípade monolitu uviesť novú verziu, musíme najskôr zastaviť predchádzajúcu, nasadiť novú a znova spustiť prevádzku. Pri mikroslužbách vieme docieliť plynulejší prechod medzi verziami tak, že k starej verzii spustíme naraz aj novú a po jej uvedení do prevádzky starú verziu vypneme.

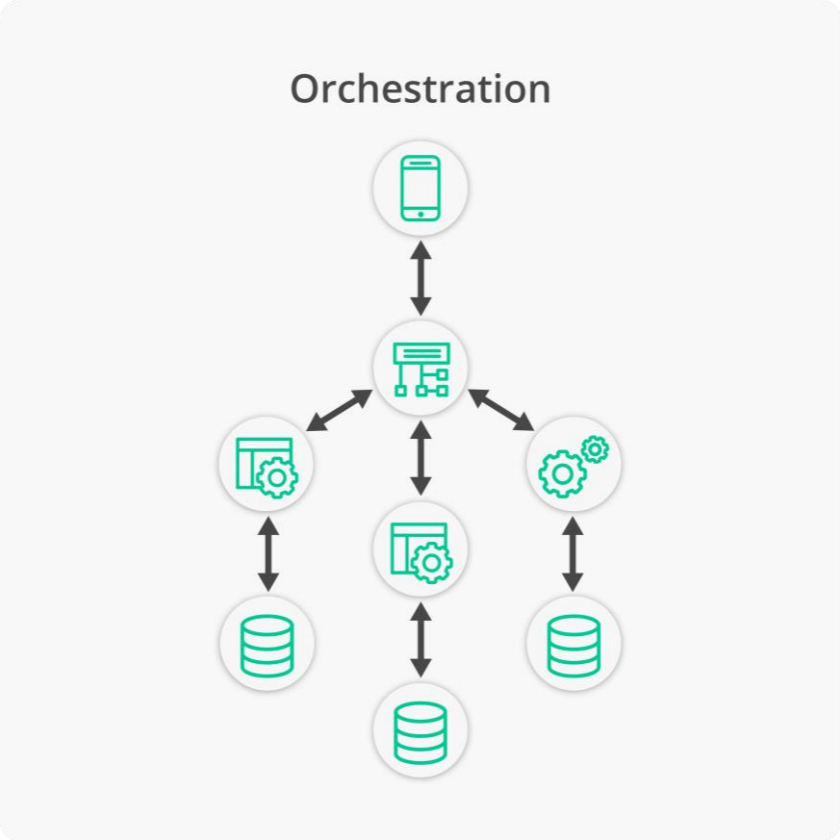
Monolitické aplikácie trpia problémom, že nútia všetkých klientov používať rovnakú verziu aplikácie. Pri každej zmene API, ktorá naruší kompatibilitu, sa musia aktualizovať všetci klienti. Ak by jediným klientom bola webová stránka, bolo to v poriadku. Ak by to boli mobilné zariadenia a web, bolo to zložitejšie. V dnešnom multiplatformovom svete však môžu existovať desiatky klientov, z ktorých každý má svoj vlastný cyklus vydávania. Nie je možné, aby desiatky klientov vydávali nové verzie v rovnakom čase a preto musí mať každý klient svoj vlastný cyklus vydávania nových verzií s vhodne implementovanou migráciou. Tento prístup umožňuje jednotlivej mikroslužbe rýchlo inovovať bez ohľadu na klientov. [16]

3.6 Orchestrácia a choreografia

Akonáhle v rámci nášho projektu spustíme niekoľko mikroslužieb, je samozrejme potrebné, aby zdieľali údaje a vytvárali obchodnú logiku.

Orchestrácia mikroslužieb znamená že nadradená služba riadi požiadavky prichádzajúce z vonka. Vhodne implementovaný orchestrátor presmeruje danú požiadavku na príslušnú mikroslužbu. Získané dáta poputujú späť do orchestrátora a vygeneruje sa konečná odpoveď, ktorá je poskytnutá klientskej aplikácii.

Úlohou orchestrátora je taktiež komunikovať s každou službou, ktorá je potrebná na splnenie akejkoľvek požiadavky API. [18]



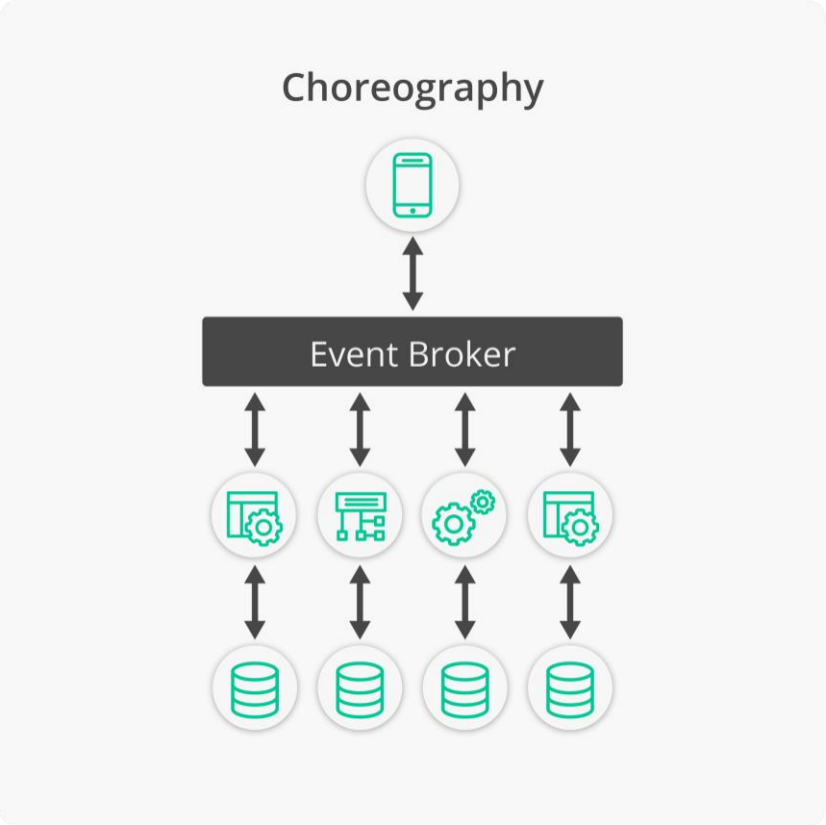
Obrázok 9. Orchestrácia v mikroslužbách [19]

Pracovný postup orchestrácie musí volať rozhrania API rôznych mikroslužieb, pričom výstup z jednej z nich sa prenáša do ďalšej čím vzniká úzka väzba medzi nimi. Zmena API jednej z týchto mikroslužieb si následne vyžaduje aktualizáciu a opätovné testovanie celého pracovného postupu. Dokonca aj jednoduché zmeny kódu API môžu vyžadovať opätovné pretestovanie celého backendu spoločnosti, čo vedie k neskorším nasadeniam do produkcie. Taktiež treba poznamenať že ak znefunkčníme orchestrátor, znefunkčníme vlastne celú aplikáciu. [17]

Mikroslužby uprednostňujú skôr choreografiu ako orchestráciu. Namiesto toho, aby každej mikroslužbe centrálny orchestrátor hovoril, čo má robiť, má každá aplikácia dostatok inteligencie na to, aby sama vedela, čo má robiť. [17]

Túto problematiku rieši choreografia mikroslužieb.

Pri choreografii mikroslužby pracujú paralelne, na rozdiel od orchestrácie. Celý systém pracuje na architektúre založenej na udalostiach, kde služba zbiera údaje zo zbernice udalostí a vykonáva obchodnú logiku a na oplátku odosiela údaje späť.[18] Na choreografiu mikroslužieb je potrebné navrhnúť vhodný spôsob výmeny správ, sprostredkovateľa alebo zbernicu udalostí. [19]

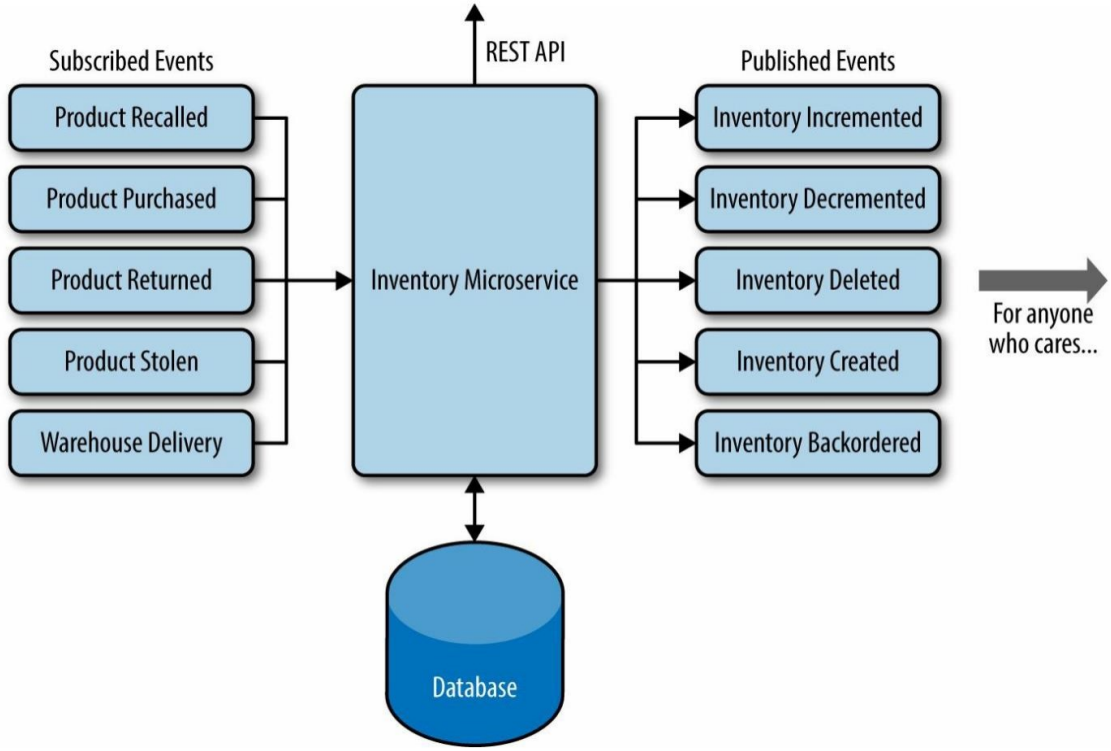


Obrázok 10. Choreografia v mikroslužbách [19]

Požiadavka klienta je posielaná do zbernice udalostí. Správy alebo udalosti, ktoré prichádzajú, sa posielajú účastníkom alebo službám, ktoré majú o danú správu záujem. Jednotlivé mikroslužby sa môžu prihlásiť na príjem tejto udalosti a vždy keď nastane, mikroslužba vykoná svoju operáciu podľa požiadavku. V prípade úspechu môže služba poslať správu späť do rovnakej alebo do zbernice, aby iná služba mohla v prípade potreby pokračovať v pracovnom postupe. Ak operácia zlyhá, zbernica správ môže túto operáciu opakovať.[19][20]

Každá mikroslužba má možnosť prihlásiť sa k udalostiam avšak jednotlivé mikroslužby nevedia ktoré aplikácie udalosti vyvolávajú a ktoré odchytávajú. Je to oddelená architektúra určená pre distribuované systémy. [19]

Príkladom tohto prístupu môže byť situácia kedy po objednaní položky z obchodu je potrebné znížiť počet položky v sklade. V tomto prípade by mikroslužba košíka vyvolala udalosť nákupu a poslala ju do zbernice udalostí. Z tejto zbernice si túto udalosť odchytí mikroslužba skladu a vykoná funkciu zníženia počtu položky v sklade.



Obrázok 11. Udalosťami riadená mikroslužba [16]

3.7 Konzistencia založená na udalostiach

Mikroslužby sú distribuované už zo svojej podstaty. Namiesto jednej veľkej aplikácie môžu existovať desiatky, stovky alebo tisíce jednotlivých mikroslužieb. Každá z týchto mikroslužieb má vlastné dátové úložisko a komunikácia medzi nimi prebieha vo väčšine prípadov asynchrónne. To znamená, že keď služba aktualizuje svoje vlastné dátové úložisko, nie je zaručené, že ostatné služby túto aktualizáciu okamžite uvidia. [16]

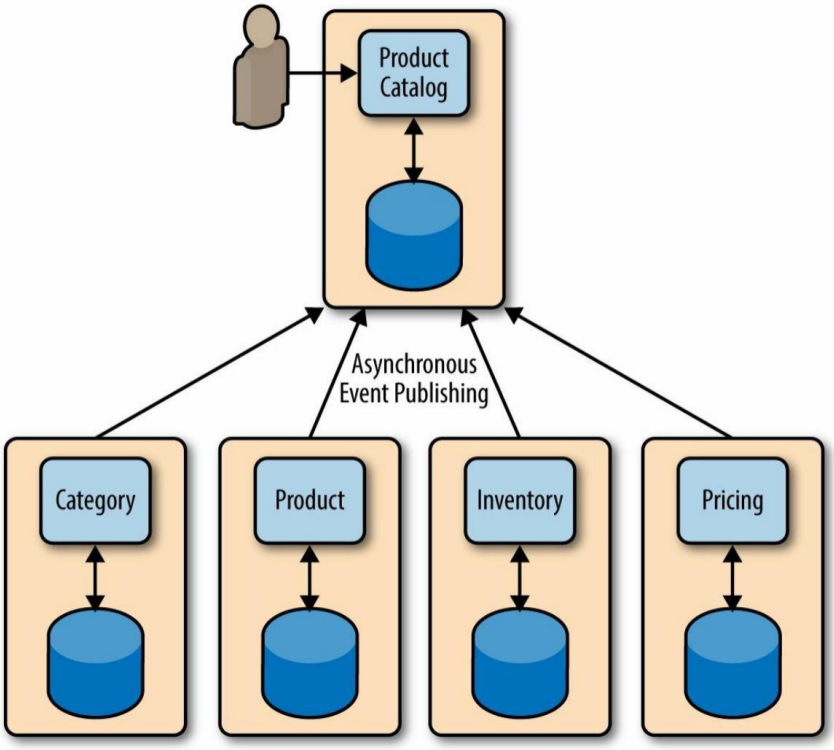
Konzistencia založená na udalostiach alebo angl. pojem Eventual consistency je charakteristickou vlastnosťou architektúry mikroslužieb, ktorá sa zaoberá zabezpečením konzistentnosti údajov v distribuovanom systéme, kde na seba vzájomne pôsobí viacero služieb. Je to prístup k replikácii údajov, ktorý potvrdzuje, že môžu existovať krátke časové úseky, počas ktorých môžu byť údaje v jednotlivých službách nesynchronizované a tým pádom zobrazovať odlišné údaje.[21] Pojem taktiež hovorí že systém môže zostať

konzistentným aj napriek rozdeleniu ak je vhodne navrhnuté vzájomné pôsobenie jednotlivých služieb.

Konzistencia založená na udalostiach hovorí že nesúlad medzi údajmi jednotlivých mikroslužieb môže byť vyriešený vykonaním replikácie, tvorbou verzií a riešením konfliktov. Keď nadradená mikroslužba aktualizuje svoje dáta, táto aktualizácia sa replikuje do ostatných mikroslužieb v systéme pomocou udalostí. Každá z mikroslužieb funguje ako systém záznamov pre svoje vlastné jednotlivé údaje, ale tieto údaje poskytuje iba nadradenej mikroslužbe vo forme udalostí. [16]

Obrázok č. 9. znázorňuje rozdelenie jednotlivých mikroslužieb kde každá z nich má vlastné dátové úložisko. Po vykonaní zmeny v jednej mikroslužbe sa vyvolá udalosť zmeny. Týmto spôsobom sa predáva informácia o zmene zainteresovaným mikroslužbám ktoré následne aplikujú zmeny vo svojich dátach.

Každá mikroslužba môže mať svoju vlastnú verziu údajov a tieto verzie sa môžu navzájom mierne líšiť. Konflikty môžu vzniknúť, keď viacero mikroslužieb aktualizuje tie isté údaje v rovnakom čase. Na riešenie konfliktov sa používajú rôzne techniky ktoré zabezpečia že sa všetky mikroslužby v priebehu času zhodnú na konzistentnom zobrazení údajov. [21, 22]



Obrázok 12. Replikácia dát na základe odlišnej funkcionality [16]

Jednou z hlavných výhod tejto vlastnosti je opäť škálovateľnosť. Tým, že umožníme mikroslužbám pracovať nezávisle a asynchrónne, systém dokáže zvládnuť veľké objemy prevádzky a údajov bez toho, aby došlo k ich zahlteniu. Okrem toho konzistencia založená na udalostiach umožňuje odolnosť voči poruchám, keďže zlyhanie jednotlivých služieb nemusí nevyhnutne viesť k zlyhaniu celého systému. [16,23]

Konzistencia založená na udalostiach má však aj niektoré nevýhody. Jednou z hlavných výziev je zabezpečiť, aby sa systém nakoniec stal konzistentným. Čas potrebný na šírenie aktualizácií a riešenie konfliktov sa môže značne líšiť a môže byť ťažké predpovedať, ako dlho bude trvať, kým systém dosiahne konzistentný pohľad na údaje. Okrem toho môže konzistencia založená na udalostiach sťažiť zdôvodnenie stavu systému v danom čase, čo môže skomplikovať ladenie a riešenie problémov. [21,23z]

Celkovo je konzistencia založená na udalostiach dôležitou vlastnosťou architektúry mikroslužieb, ktorá umožňuje škálovateľnosť a odolnosť voči chybám, ale vyžaduje si aj dôkladné zváženie a správu, aby sa zabezpečila konzistencia údajov v celom systéme.

4 ROZDIELY MEDZI MONOLITICKÝM RIEŠENÍM A RIEŠENÍM S VYUŽITÍM MIKROSLUŽIEB

Pri vytváraní softvérovej aplikácie je jedným z kľúčových rozhodnutí, či použiť monolitickú architektúru alebo architektúru mikroslužieb. Monolitická architektúra sa skladá z jedného veľkého celku, zatiaľ čo architektúra mikroslužieb predstavuje súbor menších, nezávisle nasaditeľných služieb. Oba prístupy majú svoje výhody a nevýhody a výber medzi nimi závisí od mnohých faktorov, ako sú škálovateľnosť, rýchlosť vývoja, variabilnosť technológií, spoľahlivosť, údržba, štruktúra tímu, náklady, zložitosť, bezpečnosť a ďalšie. Táto časť práce je určená na porovnanie monolitickej architektúry a architektúry mikroslužieb práve na základe týchto problematík.

4.1 Škálovateľnosť

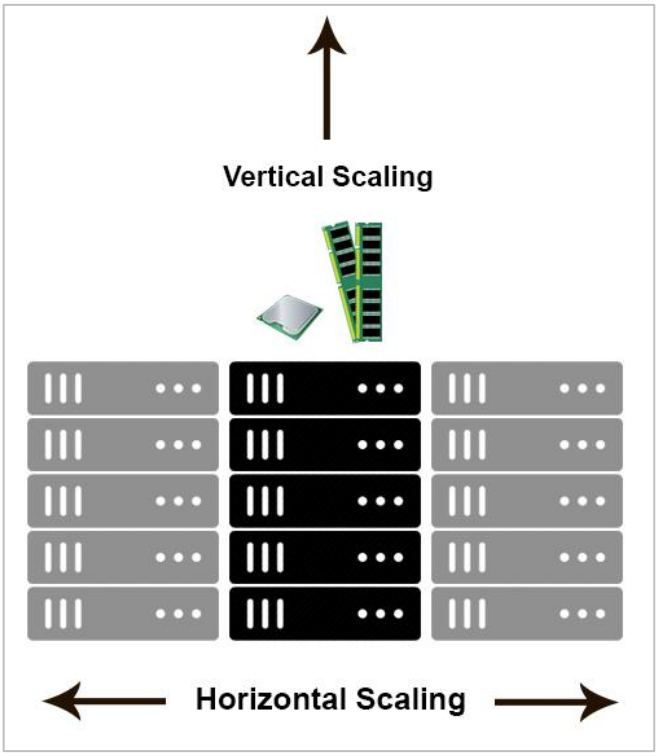
Ak je aplikácia úspešná, je možné očakávať prudký nárast používateľov čím vzniká stále väčšia potreba prispôsobiť sa rastúcej používateľskej základni. Škálovanie je jednou z najväčších výziev, ktorým čelí každý podnik, keď sa snaží uspokojiť túto rastúcu základňu.

Pojem škálovateľnosť v tomto kontexte znamená schopnosť systému/programu lepšie zvládať nárast práce. Inými slovami, škálovateľnosť je schopnosť systému/programu škálovať.[24]

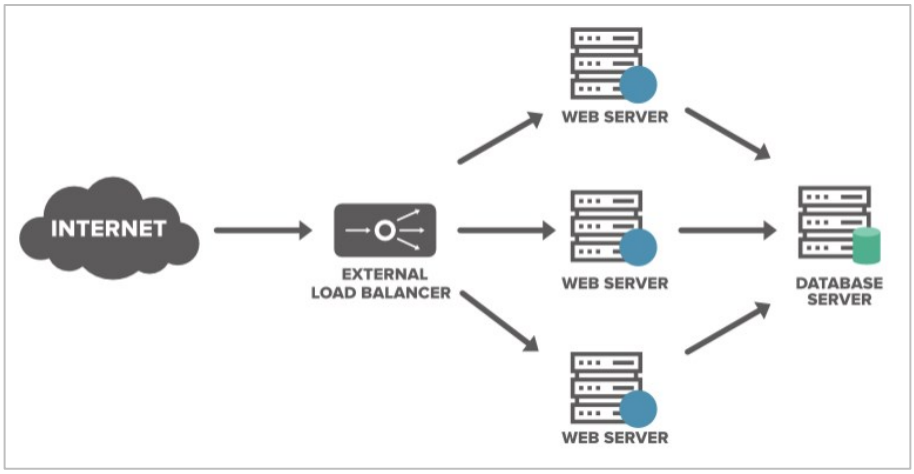
V prípade monolitickej aplikácie možno škálovanie vykonať nasledovnými spôsobmi:

- **Vertikálne škálovanie –** Vertikálne škálovanie spravidla znamená zvýšenie zdrojov ako napríklad procesora, pamäte, siete, úložiska atď. čo nám vďaka väčšiemu výkonu umožní vykonávať viac práce a zvládnuť väčšiu záťaž. [24]
- **Horizontálne škálovanie -** Horizontálne škálovanie znamená rozloženie záťaže na mnoho zdrojov. Na škálovanie je povinné celú aplikáciu replikovať. Rozdelenie aplikácie na súbor dynamických aplikačných služieb však uľahčuje výber a replikáciu jednej alebo viacerých aplikačných komponentov/služieb na účely škálovania. V praxi sa tento problém rieši tak že sa rovnaká aplikácia takpovediac naklonuje na viac serverov čím sa zabezpečí lepšia dostupnosť. Tento spôsob si osvojili viaceré vzory ako napríklad SOA. [24] [8]
- **Použitie vyrovnávačov záťaže –** Pri tomto spôsobe všetky prichádzajúce požiadavky spracúva implementovaný vyrovnávač záťaže (angl. load balancer). Jeho

úlohou je optimalizovať riadenie požiadaviek tým že ich rovnomerne distribuuje medzi jednotlivé funkcie tak aby nedochádzalo k zbytočným čakaniam a výpočtom. Existujú rôzne spôsoby a algoritmy ako implementovať vyrovnávač záťaže. Jedným z nich je Round Robin Scheduler [24]

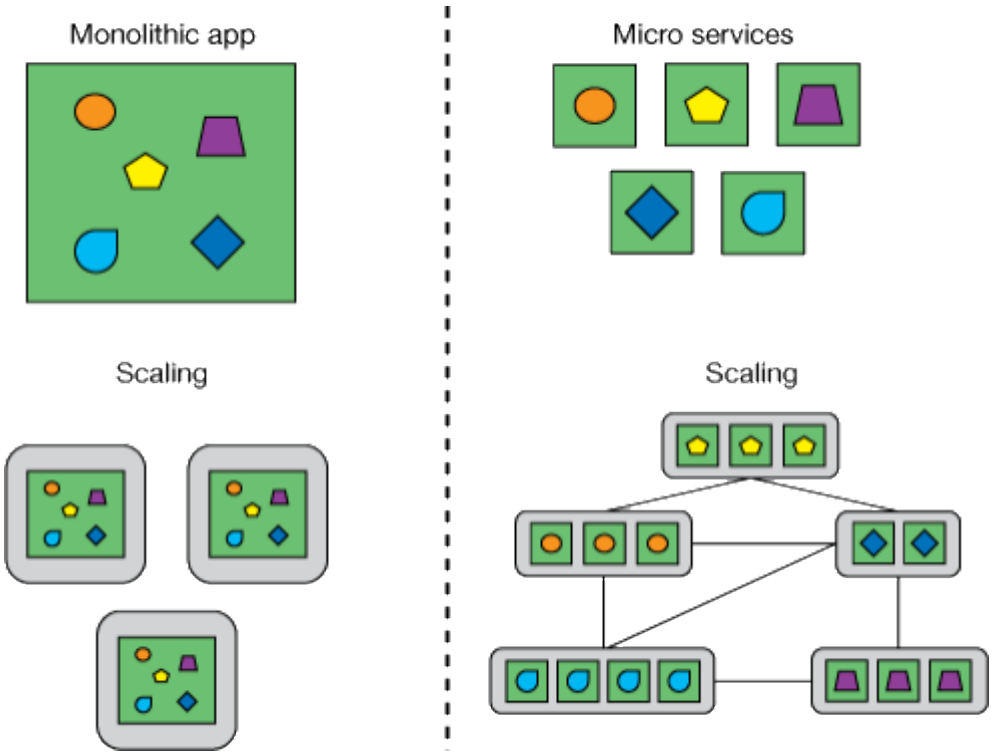


Obrázok 13. Príklad vertikálneho a horizontálneho škálovania monolitickej aplikácie [24]



Obrázok 14. Príklad škálovania s využitím vyrovnávača záťaže [24]

Architektúra mikroslužieb rozdeľuje komplexnú štruktúru na viac interaktívnych a menších komponentov pričom každý z nich odpovedá určitej biznis funkcionalite. Ak nastane zvýšenie dopytu po týchto funkciách, architektúra mikroslužieb umožňuje vďaka spomínanému rozdeleniu škálovať jednotlivé časti samostatne na rozdiel od celej aplikácie. Týmto sa zameriame priamo na miesto problému a vieme docieliť efektívnejšie využitie zdrojov. Mikroslužby možno nasadzovať nezávisle od seba na servery s rôznou výkonnosťou vďaka čomu môžeme ušetriť na nákladoch.



Obrázok 15. Porovnanie škálovania monolitickej aplikácie a aplikácie využívajúcej architektúru mikroslužieb [25]

4.2 Obtiažnosť vývoja a nasadzovania

Aj keď v poslednej dobe pozorujeme rozmach aplikácií postavených na architektúre mikroslužieb nie vždy sa tento spôsob vývoja oplatí. Samozrejme je vhodné ak myslíme o krok dopredu ale niekedy to môže priniesť viac škody ako úžitku. [4]

Ako odporúča Martin Fowler vo svojom článku[4]:

> *O mikroslužbách ani neuvažujte pokiaľ nemáte systém, ktorý je príliš zložitý na to, aby ste ho spravovali ako monolit.*

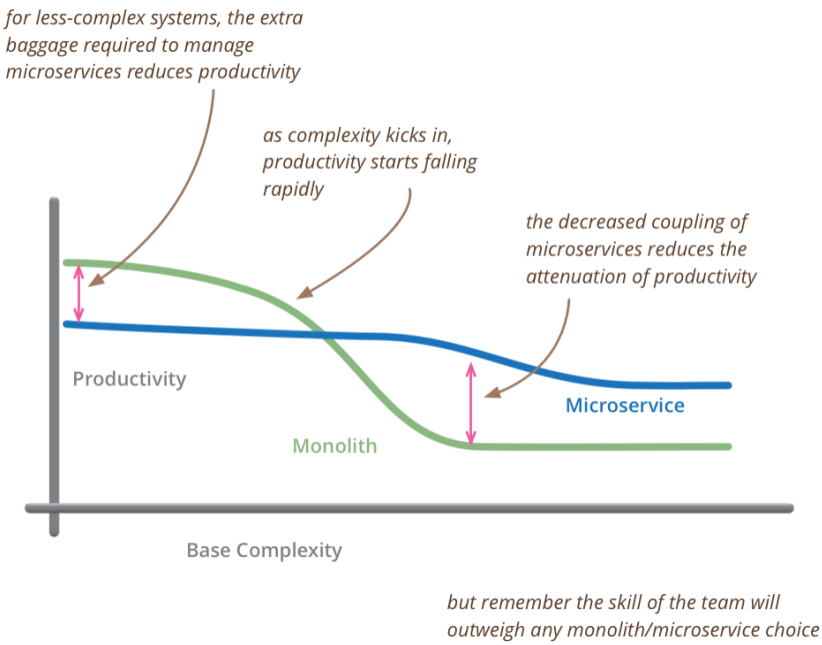
V prípade monolitických aplikácií je celý softvérový systém vytvorený ako jedna samostatná jednotka. Všetky komponenty, moduly a funkcie sú úzko prepojené, čo uľahčuje ich vývoj ako celku. Monolitické aplikácie majú vo všeobecnosti jednoduchšiu štruktúru s jedinou kódovou základňou a jednotnou databázou. Táto jednoduchosť môže viesť k rýchlejšiemu počiatočnému vývoju, pretože vývojári sa môžu sústrediť na budovanie funkcií bez potreby zložitých komunikačných mechanizmov medzi komponentmi. [26,27]

Vývoj sa môže spomaliť s narastajúcou komplexnosťou systému pretože sa stáva neprehľadným, vzniká mnoho duplicitného kódu ktorý musí byť prepojený s ostatnými časťami a šanca na vytvorenie chyby narastá.

Pri aplikáciách ktoré využívajú architektúru mikroslužieb sa narastajúca komplexnosť rieši rozdelením jednotnej štruktúry na menšie komponenty – služby. Vývoj jednotlivých mikroslužieb prebieha nezávisle. Založia sa nové tímy ktoré nemusia udržiavať obrovský kód ale iba jednu špecifickú časť pričom rýchlosť vývoja sa zmení nepatrne.

Na druhú stranu tímom pribudne práca naviac – správa a implementácia komunikácie, keďže výmena informácií medzi jednotlivými službami je v prípade mikroslužieb vykonávaná vzdialene.

Nasledujúci obrázok znázorňuje ako narastajúca komplexnosť problému vplýva na produktivitu.



Obrázok 16. Porovnanie produktivity medzi monolitickou architektúrou a architektúrou mikroslužieb [4]

4.3 Nasadenie

Na vydanie zmeny v monolitickej aplikácii je nutné nasadiť celú aplikáciu. Nasadenie môže byť relatívne jednoduché, pretože je potrebné nasadiť a nakonfigurovať vždy iba jeden balík aplikácie. Ak sa však naša monolitická aplikácia rozrastie, s nasadením môžu nastať problémy. V praxi sa vysoko rizikové nasadenia uskutočňujú zriedkavo kvôli pochopiteľným obavám. Bohužiaľ to znamená, že naše zmeny sa hromadia a narastajú medzi jednotlivými vydaniami, až kým sa ich nenazbiera dostatočne veľa na uvoľnenie novej verzie do produkcie. Čím väčšie je rozpätie medzi vydaniami, tým vyššie riziko, že sa niečo pokazí. V prípade chyby sa vyžaduje ďalšie testovanie a je nutné nasadzovať celý systém znova a znova, čo vedie k pomalším cyklom vydávania. [17]

Mikroslužby na rozdiel od monolitickej architektúry poskytujú flexibilnejšie nasadzovanie. Pri mikroslužbách môžeme vykonať zmenu v jednej službe a nasadiť ju nezávisle od zvyšku systému čo nám umožňuje náš kód nasadiť rýchlejšie. Taktiež môže byť zavedený systém kontinuálneho nasadzovania (continuous deployment), kde každá zmena v kóde je automaticky nasadená po úspešnom testovaní. [4,3,17]

Napriek tomu, že nasadzovanie mikroslužieb ponúka viac flexibilnosti, vyžaduje to aj lepšiu infraštruktúru a nástroje na riadenie. Je potrebné mať mechanizmy pre správu rôznych verzii služieb, riešiť otázky synchronizácie dát medzi službami a zabezpečiť efektívne monitorovanie a ladenie distribuovaného systému. Nasadenie mikroslužieb môže byť tiež náročnejšie vzhľadom na potrebu riadenia a správy množstva služieb v rámci aplikácie.[28]

4.4 Technologická flexibilita

Ak chceme aktualizovať ktorúkoľvek zložku monolitickej aplikácie, je potrebné opätovne nasadiť celú aplikáciu. Vývojári monolitických aplikácií sú obmedzení na pôvodnej technológii, ktorú použili na začiatku vývoja. Ak by chceli aktualizovať svoju aplikáciu na novú technológiu alebo programovací jazyk s cieľom zlepšiť fungovanie konkrétnej funkcie v rámci aplikácie, museli by začať od začiatku a celú aplikáciu preprogramovať na novú technológiu. [29]

Pri aplikáciách mikroslužieb je každá služba vyvíjaná vlastným programovacím jazykom a používa technológie, ktoré najlepšie vyhovujú jej požiadavkám. To znamená, že v rámci jednej aplikácie môžu byť použité rôzne jazyky a technológie pre jednotlivé mikroslužby.

Tento prístup poskytuje vývojárom väčšiu slobodu výberu technológií a možnosť využitia najnovších nástrojov a postupov v každej mikroslužbe.

4.5 Izolácia porúch a odolnosť

Z dôvodu vzájomnej závislosti všetkých častí monolitickej architektúry sa v prípade zlyhania komponentu alebo modulu môže stať že bude ovplyvnená celá aplikácia. Keďže všetky komponenty sú úzko prepojené, zlyhanie jednej časti sa môže rozšíriť a spôsobiť zlyhanie celého systému. Izolácia a obnova po poruche môže byť náročná vzhľadom na vzájomnú závislosť medzi modulmi. Aby sme znížili pravdepodobnosť zlyhania, môžeme nasadiť aplikáciu na viac serverov naraz.[17]

Pri mikroslužbách môžeme vytvoriť systémy, ktoré zvládnu úplné zlyhanie služieb a vedia sa prispôsobiť ak nastane degradácia funkčnosti. [17] V architektúre mikroslužieb sú služby navzájom izolované. Ak zlyhá konkrétna služba, zvyšok systému môže naďalej fungovať nezávisle. Táto izolácia zvyšuje odolnosť voči chybám a odolnosť celého systému. Obdobne ako pri monolitickej architektúre môžeme zvýšiť odolnosť tým že navrhneme jednotlivé služby so zabudovanou redundanciou a mechanizmami prechodu na náhradnú službu, aby sa poruchy zvládli elegantne.

Ďalším problémom pri mikroslužbách je sieť. Keďže jednotlivé mikroslužby medzi sebou komunikujú prostredníctvom siete je nevyhnutné zabezpečiť redundanciu dôležitých služieb pre prípad možného výpadku.

4.6 Bezpečnosť

Pri monolitickej architektúre je v prípade útoku ohrozená celá aplikácia. Tradičný firewall poskytne primeranú ochranu malej monolitickej aplikácii, ale veľké, komplexné aplikácie budú zraniteľnejšie.

Pri architektúre mikroslužieb je však hrozba rozložená na všetky jednotlivé služby. Vo väčšine prípadov bude celá aplikácia naďalej fungovať, aj keď sa jeden modul stane terčom útoku. Potenciálnych miest útoku je však viac, preto by mali tímy vyvíjajúce softvér dôkladne zvážiť bezpečnosť každej služby. Keďže integrácie tretích strán a rozhrania API sú kritickými súčasťami mikroslužieb, autentifikačné protokoly a šifrovanie sú obzvlášť dôležité.[30]

4.7 Sumarizácia

Obe architektúry prinášajú výhody aj nevýhody. Záleží na vývojovom tíme aby starostlivo zvážili všetky aspekty projektu a požiadavky klienta.

Zatiaľ čo zložitejšie aplikácie môžu využívať výhody mikroslužieb, monolitické aplikácie zostávajú obľúbené pre mnohé jednoduché aplikácie, pretože sa ľahko vytvárajú a nasadzujú. Ak vyvíjame jednoduchú aplikáciu, napríklad webové fórum alebo elektronický obchod, alebo chceme vytvoriť koncept aplikácie pred začatím ambicióznejšieho projektu, monolitická architektúra môže byť vhodným spôsobom budovania nášho softvéru.

Ďalším prvkom na zváženie je počet vývojárov ktorý sa budú podieľať na vývoji aplikácie. Ich zručnosti by mali byť jedným z hlavných rozhodujúcich faktorov pri výbere typu architektúry. Ak tím nemá skúsenosti s mikroslužbami, vytvorenie aplikácie založenej na mikroslužbách môže byť náročné. Monolitické aplikácie sú vhodnejšie aj pre jednotlivých vývojárov alebo malé tímy. Na druhej strane, ak je tím skúsený v nasadzovaní mikroslužieb a plánuje sa časom rozšíriť, začať s mikroslužbami v rannej fáze môže v budúcnosti ušetriť množstvo času.

V prípade nárastu obľúbenosti a dopyte po nových funkciách sa monolitické aplikácie môžu stať zložitejšími a ťažšie upravovateľnými a taktiež môžu nastať problémy so škálovaním. Ak očakávame navýšenie počtu používateľov našej aplikácie, je nevyhnutné rozšíriť jej funkcionalitu a zväčšiť tím, ktorý aplikáciu spravuje. Mikroslužby nám svojim rozdelením umožnia jednoduchšie škálovanie. Naopak pre aplikácie ktoré sú vytvorené na špecifické prípady použitia bude vhodnejšie použiť jednotnú a kompaktnú monolitickú architektúru.

Pri vývoji aplikácie by sa mali brať do úvahy aj náklady na jej vytvorenie a časový harmonogram nasadenia. Hoci monolitické aplikácie môžu byť pri raste drahšie, ich vytvorenie môže byť nákladovo efektívnejšie a rýchlejšie. Počiatočné zdroje potrebné na vývoj mikroslužieb sú často vysoké, ale v budúcnosti môžu priniesť úsporu nákladov pri škálovaní aplikácie.[31]

II. PRAKTICKÁ ČASŤ

5 NÁVRH APLIKÁCIE PRE ZDIEĽANIE VÝLETOV A UBYTOVANIA S VYUŽITÍM MIKROSLUŽIEB

Pre priblíženie postupu vytvárania aplikácie s architektúrou mikroslužieb v praxi je potrebné vymyslieť aplikáciu ktorá je vhodná pre tento typ architektonického vývoja. Aplikáciu so zameraním na zdieľanie výletov a ubytovaní bude vhodným príkladom na demonštráciu využitia tejto architektúry.

V úvodných častiach je predstavený účel aplikácie a definícia základných požiadaviek aplikácie. Následne je predstavený návrh riešenia s popisom použitých technológií a návrh architektúry jednotlivých mikroslužieb.

5.1 Koncept aplikácie

Účelom tejto aplikácie, ako vyplýva z názvu práce, je v prvom rade zdieľanie výletov a ubytovaní. Čo si však pod tým predstaviť?

Poznáme mnoho aplikácií ktoré ponúkajú podobné služby avšak za poplatok. Hlavnou úlohou bude teda sprostredkovanie výmeny alebo zdieľania týchto služieb medzi používateľmi s minimálnymi poplatkami, ideálne bez poplatku.

Princíp je jednoduchý. Používatelia vytvoria ponuky s ubytovaním alebo s a následne si môžu prezerať dostupné ponuky iných používateľov. V prípade záujmu môžu ponúknuť výmenný pobyt alebo výmenný výlet. Po prijatí ponuky používateľ absolvuje výlet v krajine druhého používateľa, a naopak, niekedy v budúcnosti druhý používateľ absolvuje ponúkaný výlet v krajine prvého používateľa. Používatelia môžu tieto ponuky prijať alebo zamietnuť.

5.2 Požiadavky aplikácie

V tejto kapitole sú popísané požiadavky aplikácie ktoré slúžia ako základ pre návrh a implementáciu softvéru. Taktiež definujú požadované vlastnosti systému, ktoré nie sú priamo spojené s konkrétnymi funkciami, ale ovplyvňujú jeho výkon, spoľahlivosť, použiteľnosť a iné charakteristiky.

5.2.1 Funkčné požiadavky

Registrácia používateľov: Aplikácia by mala umožňovať používateľom vytvárať účty a registrovať sa pomocou e-mailovej adresy.

Vytvorenie ponuky: Používatelia by mali mať možnosť vytvárať ponuky s ubytovaním alebo výletom. To zahŕňa poskytovanie informácií o mieste, dátume a podobne.

Zobrazenie zoznamu dostupných ponúk: Používatelia by mali mať prístup k zoznamu dostupných ponúk ubytovania a výletov iných používateľov. Môžu tieto ponuky filtrovať a triediť podľa rôznych kritérií.

Ponuka rezervácie a výmeny: Používatelia by mali mať možnosť reagovať na ponuky iných používateľov a navrhnúť výmenu ubytovania alebo výletu. V prípade dohody medzi používateľmi vznikne rezervácia na oboch stranách.

5.2.2 Nefunkčné požiadavky

Dostupnosť: Aplikácia by mala byť dostupná pre používateľov 24 hodín denne, 7 dní v týždni, s minimálnym výpadkom služby.

Výkonové požiadavky: Aplikácia by mala mať dostatočný výkon a kapacitu na rýchle spracovanie dopytov a zobrazenie výsledkov používateľom bez oneskorenia.

Nezávislosť služieb: Jednotlivé služby (vytváranie ubytovania alebo výletov) by mali fungovať nezávisle od seba. To znamená, že ak jedna služba nedostupná alebo nefunkčná, ostatné služby by mali byť stále dostupné a použiteľné.

Redundancia a škálovateľnosť: Aplikácia by mala byť navrhnutá s redundanciou a škálovateľnosťou v jednotlivých službách. To znamená, že ak narastie počet používateľov alebo zaťaženie systému, služby by mali byť schopné rýchlo reagovať a prispôsobiť sa zvýšenej záťaži bez straty dostupnosti alebo výkonu.

Bezpečnosť: Aplikácia by mala zabezpečovať ochranu používateľských účtov a súkromia. Heslá a iné citlivé údaje by mali byť správne šifrované a chránené pred neoprávneným prístupom.

5.3 Návrh riešenia

Navrhované riešenie zahŕňa implementáciu aplikácie v architektúre mikroslužieb, kde je každá funkcia obsluhovaná samostatnou mikroslužbou. Každá mikroslužba bude pracovať v rámci svojej oblasti prislúchajúcej špecifickej problematike (napr. mikroslužba ubytovania sa bude venovať výlučne ubytovaniu.) a nebude úzko prepojená s inými mikroslužbami.

Každá mikroslužba bude predstavovať samostatnú aplikáciu s vlastnou databázou a vlastným rozhraním API ktoré bude riadiť všetku komunikáciu. Na komunikáciu medzi službami bude využitá synchrónna komunikácia pomocou http. Mikroslužby budú komunikovať medzi sebou na základe definovaných rozhraní a štandardov, čo umožní flexibilitu pri vývoji a rozšírení jednotlivých služieb.

Mikroslužby budú zabalené do vlastného kontajnera čo umožní škálovanie v prípade zvýšeného dopytu čím sa zefektívni využitie zdrojov.

5.4 Technológie

V návrhu tejto aplikácie bude použitá technológia .NET ktorá poskytuje výkonný a flexibilný framework pre vývoj webových aplikácií s využitím jazyka C#. Výhodou použitia .NET je široká podpora knižníc a nástrojov, ktoré môžu zjednodušiť implementáciu jednotlivých mikroslužieb.

Pre vývoj webových API je navrhnuté využitie ASP.NET Core framework. Tento framework poskytuje robustné nástroje pre tvorbu a správu webových API. Okrem toho zahŕňa funkcie ako spracovanie požiadaviek, autentifikáciu, autorizáciu a podporu pre vytváranie REST API.

Pre prácu s objektami a databázou je navrhnuté využitie Entity Framework (EF), ktorý je populárnym ORM (Object-Relational Mapping) nástrojom v .NET technológii. Využitie EF v aplikácii zjednodušuje prístup k dátam a zabezpečuje správne mapovanie medzi objektovým modelom a štruktúrou databázy aj vďaka anotáciám. Ďalej je navrhnuté využitie databázy MongoDB. Jedná sa o NoSQL databázu, ktorá ponúka dokumentový model, kde dáta sú uložené vo formáte JSON podobných dokumentov. Vďaka svojej schopnosti ukladať a spravovať rôznorodé a neštruktúrované dáta je MongoDB ideálna pre aplikácie s rastúcimi a premenlivými požiadavkami dátových schém.

```
public class Customer
    {
        [Key]
        [Column("CustomerID")]
        [Required]
        public string Id { get; set; }
        [Required]
        [StringLength(40)]
        public string CompanyName { get; set; }
        [StringLength(30)]
        public string ContactName { get; set; }
        [StringLength(60)]
        public string ContactTitle { get; set; }
        [Timestamp]
        public Byte[] TimeStamp { get; set; }
    }
```

Ukážka kódu 1 Entity framework umožňuje pridať anotácie k property hodnotám

Pre kontajnerizáciu a orchestráciu mikroslužieb je navrhnuté využitie platforiem Docker a Kubernetes. Docker umožní balenie aplikácií a jej závislostí do samostatných kontajnerov, ktoré sú izolované a môžu byť nasadené na rôznych platformách bez ohľadu na konkrétnu konfiguráciu hostiteľského systému. Ďalšou výhodou Dockeru je škálovateľnosť. Kontajnery môžu byť jednoducho duplikované a škálované podľa potreby. Ak je potrebné zvýšiť kapacitu služby, jednoducho pridáme ďalšie kontajnery.

Kontajnery budú potom nasadené a spravované pomocou platformy Kubernetes, ktorá umožňuje centralizovanú správu a riadenie kontajnerových aplikácií vrátane automatického škálovania, obnovy zlyhaní a riadenia premávky. Kubernetes zabezpečuje, že mikroslužby budú dostupné a nasadené s minimálnym úsilím.

Pre zabezpečenie vhodnej autentifikácie používateľov, bude implementovaný autentifikačný systém s využitím technológie JSON Web Tokens (JWT). Tento server bude overovať používateľov v databáze a pridelí im bearer token s ktorým budú vykonávať všetky požiadavky.

Pre testovanie jednotlivých výstupov aplikácie bude použitá aplikácia Postman, ktorá umožňuje jednoduché a efektívne testovanie jednotlivých mikroslužieb pomocou rôznych HTTP metód ako GET, POST, PUT, DELETE. Pomocou Postman je možné vytvárať a odosielať požiadavky na rôzne end pointy mikroslužieb a overiť ich správne fungovanie a zobraziť odpovede.

5.5 Návrh architektúry mikroslužieb

Služba pre správu používateľov: Táto mikroslužba bude zodpovedná za registráciu a prípadnú správu používateľských profilov. Bude zahŕňať end pointy pre registráciu nového používateľa a získanie informácií o používateľovi. Najvhodnejšou databázou pre túto mikroslužbu by mohla byť relačná databáza ako MySQL alebo PostgreSQL.

Služba pre Autentifikáciu: Táto služba bude overovať používateľov v databáze a poskytovať im bearer token ktorý využijú pri požiadavkách v ostatných mikroslužbách. Zabezpečuje bezpečný prístup a kontrolu oprávnení pre jednotlivé používateľské akcie.

Služba pre správu ubytovaní: Úlohou tejto mikroslužby bude vytváranie, zobrazovanie, a spravovanie ponúk ubytovaní. Bude zahŕňať end pointy pre vytvorenie novej ponuky, zobrazenie dostupných ponúk a úpravu existujúcich ponúk. Pre túto mikroslužbu by mohla byť vhodnou databázou NoSQL databáza ako MongoDB alebo Cassandra, pre ukladanie a správu informácií o ponukách ubytovania.

Služba pre správu ponúk výletov: Táto mikroslužba bude zodpovedná za vytváranie, zobrazovanie a správu výletov. Bude zahŕňať end pointy pre vytvorenie nového výletu, zobrazenie dostupných výletov a správu existujúcich výletov. Podobne ako mikroslužba pre správu ubytovania, aj pre túto mikroslužbu by mohla byť vhodnou NoSQL databáza pre ukladanie a správu informácií o výletoch.

Služba pre správu rezervácií: Táto mikroslužba bude zodpovedná za správu rezervácií ubytovania a výletov medzi používateľmi. Taktiež bude komunikovať s ostatnými službami aby vytvorila rezerváciu. Bude zahŕňať end pointy pre vytvorenie rezervácie, zobrazenie existujúcich rezervácií a správu rezervácií (napr. zrušenie, aktualizácia).

6 POSTUP IMPLEMENTÁCIE NÁVRHU

V tejto kapitole je demonštrovaný priebeh vývoja aplikácie s architektúrou mikroslužieb. Je vytvorený postup v ktorom sú popísané jednotlivé kroky vývoja. Cieľom tejto demonštrácie nie realizovanie kompletného produktu ale priblíženie všetkých kľúčových častí vývoja a praktík, ktoré neodlúčiteľne patria k architektúre mikroslužieb.

Postup bude obsahovať demonštráciu implementácie mikroslužieb, použitie vlastných databáz pre jednotlivé mikroslužby, využitie autentifikácie, demonštráciu komunikácie medzi službami, demonštráciu kontajnerizácie pomocou Docker a nasadenie s využitím Kubernetes.

Tento postup je vysvetlený v čo možno najjednoduchšom spôsobe a figurujú v ňom najpoužívanejšie metódy ktoré sa uvádzajú v súčasnej literatúre, rôznej dokumentácii a zdrojoch na internete.

V úvode je predstavený popis, ako takýto projekt založiť. Taktiež je popísané vývojové prostredie, správa verzií (GitHub) a popis súborov vo vygenerovaných službách.

Následne je implementovaná logika mikroslužieb ktoré sa budú venovať najdôležitejším funkciám aplikácie ako mikroslužba spravovania účtov, mikroslužba autentifikácie, mikroslužba ubytovania, mikroslužba výletu a mikroslužba rezervácií. V každej z týchto služieb budú základné modely, obslužné funkcie aAPI kontroler. V jednej službe bude implementovaná komunikácia medzi službami. Jednotlivé požiadavky na služby budú vo väčšine prípadov odosielané pomocou aplikácie Postman.

Ďalej bude demonštrovaný proces kontajnerizácie kde sú pre jednotlivé služby definované nastavenia pre vytvorenie kontajnerov.

V závere bude pridaná podpora orchestrátora v podobe Kubernetes kde prebehnú nastavenia pre nasadenie a replikáciu jednotlivých služieb v podobe kontajnerov.

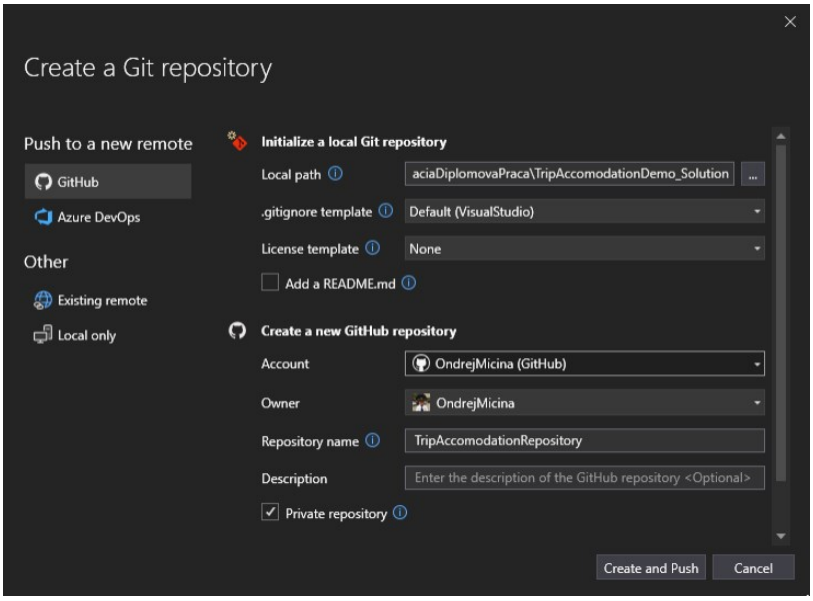
6.1 Vytvorenie projektu

Počas vývoja budú použité dve vývojové prostredia – Visual Studio (VS) a Visual Studio Code (VS Code). VS bude využité na tvorbu aplikácií keďže pomáha pri generovaní a napovedaní kódu, a taktiež prehľadne organizuje štruktúru celého systému. VS Code bude využité prevažne na vytvorenie Docker kontajnerov, ich nastavenie, a uvedenie do prevádzky spoločne s Kubernetes.

Na začiatku je vytvorený prázdny solution pomocou Visual Studio.

Pre správu verzií a správu zdrojových kódov projektov je založený GitHub repozitár ktorý bude slúžiť na priebežné ukladanie práce, na vytvorenie rôznych vetiev (branch) a taktiež ho možno využiť v prípade výskytu chyby, pre jednoduché vrátenie úprav.

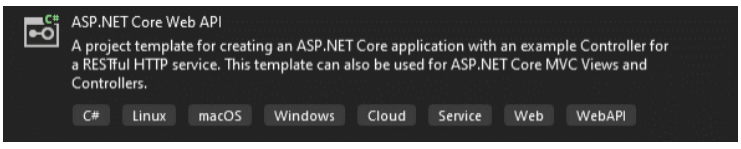
Vďaka VS je možné vytvoriť repozitár veľmi rýchlo bez nutnosti písania príkazov do príkazového riadku. Stačí sa prihlásiť pod GitHub účtom a zadať názov repozitára.



Obrázok 17. Vytvorenie GitHub repozitára

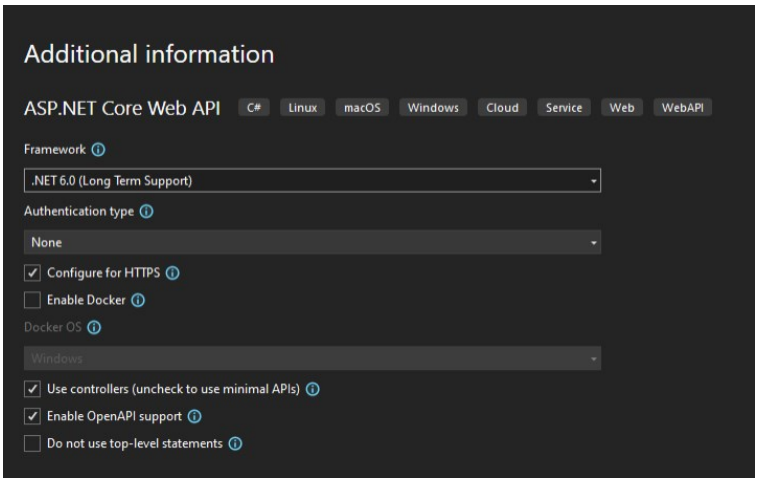
6.2 Vytvorenie mikroslužieb

Každá mikroslužba bude predstavovať samostatnú aplikáciu. Implementácie jednotlivých mikroslužieb sú vytvorené pomocou šablóny ASP.NET Core Web API.



Obrázok 18. Výber šablóny ASP.NET Core Web API

Je zvolená verzia frameworku .NET 6.0, možnosť Enable docker je vypnutá keďže kontajnerizáciu bude riešená manuálne.



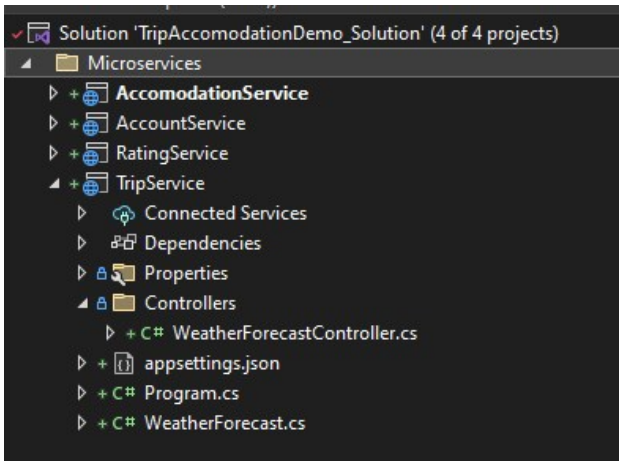
Obrázok 19. Vytvorenie projektu mikroslužby

Každá z mikroslužieb funguje ako samostatná aplikácia alebo služba, ktorá sa dá upravovať, debugovať (ladiť), spúšťať a neskôr aj nasadzovať. Ak by toto riešenie vytváral tým, v tomto momente by bolo možné začať priraďovať tieto aplikácie jednotlivým skupinám vývojárov.

Šablóna ASP.NET Core Web API je vlastne vzorový projekt, ktorý poskytuje základnú štruktúru a konfiguráciu pre vytvorenie aplikácie typu Web API pomocou frameworku ASP.NET Core.

Vygenerovaním vznikajú základné súbory a priečinky:

- **Program.cs**: Tento súbor obsahuje vstupný bod aplikácie, ktorý inicializuje hostiteľa ASP.NET Core.
- **Startup.cs**: Tento súbor obsahuje konfiguráciu aplikácie. Môžeme tu nastaviť služby, routovanie a podobne.
- **Controllers:** Tento priečinok obsahuje jednotlivé kontrolery, ktoré spracúvajú prichádzajúce HTTP požiadavky. V týchto kontroleroch budú definované rôzne akcie (actions) pre manipuláciu s dátami a logikou rozhrania API.
- **launchSettings.json**: Tento súbor obsahuje konfiguráciu spúšťania aplikácie pre rôzne prostredia a profilové nastavenia. Môžete tu definovať porty, parametre príkazového riadka, predvolené spúšťacie profily a ďalšie detaily týkajúce sa spúšťania aplikácie.
- **appsettings.json**: Tento súbor obsahuje konfiguračné údaje pre aplikáciu, ako napríklad pripojenie k databáze alebo ďalšie nastavenia.



Obrázok 20. Vytvorené služby a vygenerované súbory

6.2.1 Logika prístupu k dátam - repository pattern

Pre oddelenie prístupu k dátam od zvyšku aplikácie je využitý repository pattern. Hlavným cieľom je poskytnúť jednotné a abstraktné rozhranie pre komunikáciu s databázou alebo iným zdrojom dát. Repository pattern pracuje s kolekciou objektov, na ktorej je možné vykonávať operácie, ako sú vytváranie, čítanie, aktualizácia a mazanie (CRUD operácie). Repository poskytuje jednotné rozhranie pre tieto operácie, čo umožňuje zvyšku aplikácie pracovať s dátami bez potreby poznania konkrétneho zdroja údajov alebo podrobností o ukladaní a získavaní údajov.

6.2.2 Použitie DTO

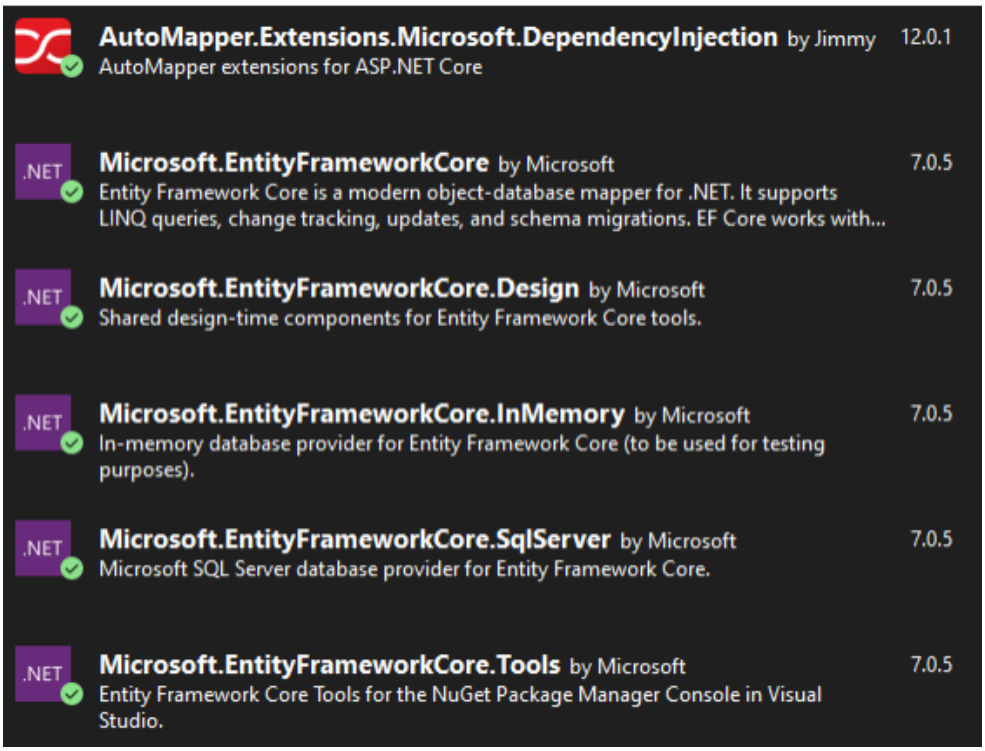
Na prenos dát sú vytvorené objekty určené výhradne pre tento účel. Takzvaný data-transfer objekt (DTO) sa používa na prenos dát medzi vrstvami alebo komponentami aplikácie. Hlavným cieľom DTO je zoskupiť relevantné dáta z pôvodného objektu pre konkrétny scenár alebo operáciu a neposkytovať nepotrebné dáta. Pomocou DTO je možné tvoriť objekty presne na mieru a nie je nutné posielať celé pôvodné objekty čo výrazne zníži objem prenášaných dát a taktiež volaní API. Pre jeden pôvodný objekt môže byť vytvorených viac DTO objektov pričom každý z nich bude slúžiť pre špecifické požiadavky.

6.3 Implementovanie logiky mikroslužieb

Táto časť popisuje implementáciu funkcionalít jednotlivých mikroslužieb. Každá z nich obsahuje svoj dátový model a API kontroler pre obsluhu požiadaviek. Rovnako sú vytvorené databázové spojenia a rozhrania pre oddelenie logiky prístupu k dátam od ostatných častí. Po implementácii budú služby spustené lokálne a otestované pomocou aplikácie Postman.

6.3.1 Služba pre správu používateľských účtov

V začiatku implementácie služby sú nainštalované potrebné balíky. Jednotlivé balíky sú pridané pomocou NuGet manažéra vo VS.



Obrázok 21. Pridané balíky do služby

Následne sa vytvára jednoduchý model typu Account ktorý predstavuje používateľský účet. Aj napriek tomu, že bude model viacmennej demonštračný, budem uchovaný v relačnej databáze. Model Account bude obsahovať nasledujúce údaje:

- **Id:** Jedná sa o identifikačný kľúč a bude generovaný automaticky v databáze. Typ property je Guid.
- **AccountName**: Slúži na uchovávanie názvu účtu. Typ property je string a je povinný ([Required]).
- **Email**: Slúži na uchovávanie emailovej adresy. Typ property je string a je povinný ([Required]).
- **PasswordHash**: Slúži na uchovávanie hesla v zahashovanom formáte. Typ property je byte[] (pole bytov) a je povinný ([Required]).

- **PasswordSalt**: Slúži na uchovávanie tzv. soli (angl. salt) (náhodne generovaný reťazec pridaný k heslu pred hashovaním). Typ property je byte[] (pole bytov) a je povinný ([Required]).
- **Role**: Slúži na uchovávanie role alebo oprávnenia používateľa. Typ property je string. Môže byť null.
- **Mobile**: Slúži na uchovávanie telefónneho čísla používateľa. Typ property je string. Môže byť null.

Pre prácu s relačnou databázou je využitý Entity Framework (EF) ktorý umožní pridať anotácie pre jednotlivé property. Okrem anotácií je možné využiť aj fluent api v metóde OnModelCreating pre kontrolu atribútov.

```
public class Account
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public Guid Id { get; set; }
    [Required]
    public string AccountName { get; set; }
    [Required]
    public string Email { get; set; }
    [Required]
    public byte[] PasswordHash { get; set; } = Array.Empty<byte>();
    [Required]
    public byte[] PasswordSalt { get; set; } = Array.Empty<byte>();
    public string? Role { get; set; }
    public string Mobile { get; set; }
}
```

Ukážka kódu 2. Model Account

Z dôvodu lokálneho testovania služba môže využiť in memory databázu použitím EF Core In-Memory Database Provider. V ďalšom kroku je definovaná trieda AccountDbContext ktorá v konštruktore prijme databázové nastavenia a vykoná pripojenie. V triede kontextu sa definuje aj DbSet<Account> ktorý reprezentuje kolekciu objektov Account a umožní prácu s nimi.

Následne sa pridá AccountDbContext kontext v Program.cs kde bude nastavené že databáza bude In-Memory.

```
builder.Services.AddDbContext<AccountDbContext>(options =>
    options.UseInMemoryDatabase("InMem"));
```

Ukážka kódu 3. Pridanie kontextu v Program.cs

S použitím repository pattern je vytvorené rozhranie IAccountRepository, ktoré implementuje trieda AccountRepository. Konštruktor injektuje databázový kontext AccountDbContext ako parameter ktorý slúži na komunikáciu s databázou a poskytuje prístup k entitám. V tejto triede budú vytvorené metódy pre správu účtov:

- **Metóda CreateAccount**: Slúži na vytvorenie nového záznamu v databáze pre entitu Account. Ak je vstupný parameter account null, vyvolá sa výnimka, inak sa entita account pridá do kolekcie Accounts v objekte AccountDbContext a teda aj do databázy.
- **Metóda GetAccountById**: Slúži na získanie záznamu z databázy podľa zadaného identifikátora id. Metóda vracia prvý záznam z kolekcie Accounts v databázovom kontexte AccountDbContext, ktorého Id sa zhoduje s daným identifikátorom id.
- **Metóda GetAllAccounts**: Slúži na získanie všetkých záznamov Account z databázy. Metóda vracia všetky záznamy z kolekcie Accounts v objekte AccountDbContext ako IEnumerable
- **Metóda SaveChanges**: Slúži na uloženie zmien v rámci objektu AccountDbContext do databázy. Volá metódu SaveChanges() na objekte AccountDbContext a vracia true, ak boli zmeny úspešne uložené, alebo false, ak sa vyskytla chyba pri ukladaní zmien.

```
public class AccountRepository : IAccountRepository
{
    private readonly AccountDbContext _context;

    public AccountRepository(AccountDbContext context)
    {
        _context = context;
    }

    public void CreateAccount(Account account)
    {
        if (account == null)
        {
            throw new ArgumentNullException(nameof(account));
        }
        _context.Accounts.Add(account);
    }

    public Account GetAccountById(Guid id)
    {
        return _context.Accounts.FirstOrDefault(x => x.Id == id);
    }

    public IEnumerable<Account> GetAllAccounts()
    {
        return _context.Accounts.ToList();
    }

    public bool SaveChanges()
    {
        return (_context.SaveChanges() >= 0);
    }
}
```

Ukážka kódu 4. Trieda AccountRepository

Trieda AccountRepository zatiaľ umožňuje pridávanie, vyhľadávanie a získavanie záznamov z databázy pre entity typu Account a tiež umožňuje uloženie zmien do databázy. Dalšie funkcie budú pridané neskôr keď pri implementácii vzájomnej komunikácie medzi službami.

Aby bolo možné použiť dependency injection pre injektovanie tohto rozhrania, je potrebné ho zaregistrovať v rámci konfigurácie v triede Program.cs

```
builder.Services.AddScoped<IAccountRepository, AccountRepository>();
```

Ukážka kódu 5. Registrovanie závislosti rozhrania IAccountRepository

V ďalšom kroku je vytvorený DTO pre požiadavku registrácie - AccountRegisterDto, ktorý bude odosielaný z vonkajšieho prostredia. Keďže model Account nebotrebuje pri vytvorení

všetky údaje, DTO bude obsahovať len nevyhnutné property. Obdobne je vytvorený DTO pre požiadavku Read - AccountReadDto, ktorý bude naopak odosielaný z vnútorného prostredia smerom von a bude obsahovať rovnaké property ako pôvodný Account avšak bez hesla čo by v reálnom nasadení znamenalo obrovské bezpečnostné zlyhanie.

```
public class AccountRegisterDto
{
    [Required]
    public string Email { get; set; }
    [Required]
    public string AccountName { get; set; }
    [Required]
    public string Password { get; set; }
}

public class AccountReadDto
{
    public Guid Id { get; set; }
    public string AccountName { get; set; }
    public string Email { get; set; }
    public string Mobile { get; set; }
    public string Role { get; set; }
}
```

Ukážka kódu 6. Objekty DTO

Keďže vo väčšine prípadov pôvodný objekt obsahuje totožné property ako DTO je možné využiť automatické translácie objektov alebo tzv. mapovanie. Pre tento účel je použitý AutoMapper. AutoMapper umožňuje jednoduchšie a čitateľnejšie mapovanie objektov ktoré majú podobné alebo zhodné vlastnosti, bez nutnosti vytvárať a udržiavať mapovacie metódy ručne. Pre mapovanie objektov v rámci služby bude slúžiť trieda AccountProfile ktorá implementuje Profile od AutoMapper. V triede sú definované vzájomné mapovania jednotlivých objektov.

```
public class AccountProfile : Profile
{
    public AccountProfile()
    {
        CreateMap<Account, AccountReadDto>();
        CreateMap<AccountRegisterDto, Account>();
    }
}
```

Ukážka kódu 7. Objekt AccountProfile

```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

Ukážka kódu 8. Pre fungovanie AutoMapper je potrebná registrácia v Program.cs

Pre obsluhu prichádzajúcich požiadaviek je vytvorený kontroler, pomocou ktorého služba vykonáva rôzne operácie, napríklad registráciu alebo návrat zoznamu používateľov. Pre prácu s databázou kontroler injektuje repository IAccountRepository. Jednotlivé metódy – endpointy vytvoria odpoveď, ktorá sa pošle späť klientovi vo forme HTTP odpovede s vhodným kódom a dátami.

Každá metóda obsahuje anotáciu s formátom požiadavky, medzi najpoužívanejšie patria napríklad:

- [HttpGet] pre vrátenie objektu,
- [HttpPost] pre vytvorenie objektu
- [HttpDelete] pre zmazanie
- [HttpPut] pre editáciu celého objektu
- [HttpPatch] pre editáciu určitých parametrov objektu

Metóda **GetAllAccounts** slúži na získanie všetkých účtov. Volá metódu GetAllAccounts() z objektu _repository, ktorá vráti kolekciu účtov. Následne sa táto kolekcia mapuje na kolekciu objektov AccountReadDto pomocou _mapper a výsledok sa vráti s kódom 200 OK.

Metóda **GetAccountById** slúži na získanie účtu podľa zadaného identifikátora. Volá metódu GetAccountById(id) z objektu _repository, ktorá vyhľadá účet podľa ID. Ak je účet nájdený, je mapovaný na objekt AccountReadDto pomocou _mapper a výsledok sa vráti s kódom 200 OK. Ak účet nie je nájdený, je vrátený odpoveď s kódom 404 Not Found.

Metóda **CreateAccount** je označená ako HTTP POST a má definovanú cestu "Create". Slúži na registráciu nového účtu. Prijíma objekt AccountRegisterDto, ktorý obsahuje informácie pre vytvorenie účtu. Metóda mapuje tento objekt na objekt typu Account pomocou _mapper. Z prijatého hesla je potrebné vytvoriť hash hesla a salt keďže v database nemôže byť uložené heslo v textovej forme. Pre tento účel je vytvorená metóda Hashing.CreatePasswordHash(). Po vytvorení objektu účtu sa pomocou _repository.CreateAccount(accountModel) účet pridá do databázy. Nakoniec sa vráti odpoveď s kódom 201 spolu s vytvoreným objektom typu AccountReadDto.

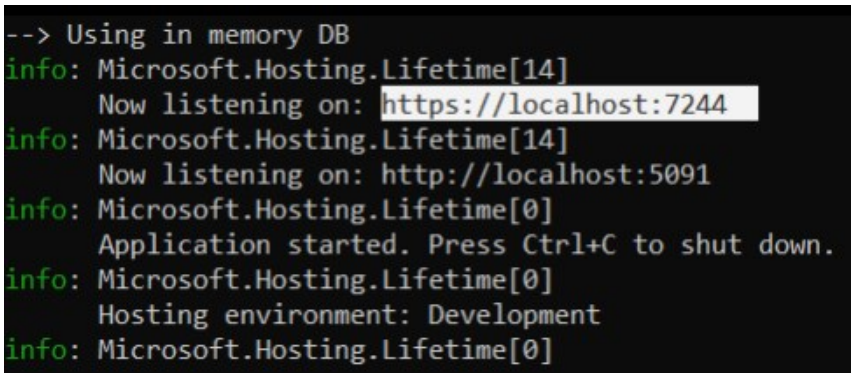
```
public static (byte[] passwordSalt, byte[] passwordHash)
CreatePasswordHash(string password)
{
    using var hmac = new System.Security.Cryptography.HMACSHA512();
    var passwordSalt = hmac.Key;
    var passwordHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
    return (passwordSalt, passwordHash);
}
```

Ukážka kódu 9. Funkcia na hashovanie hesla

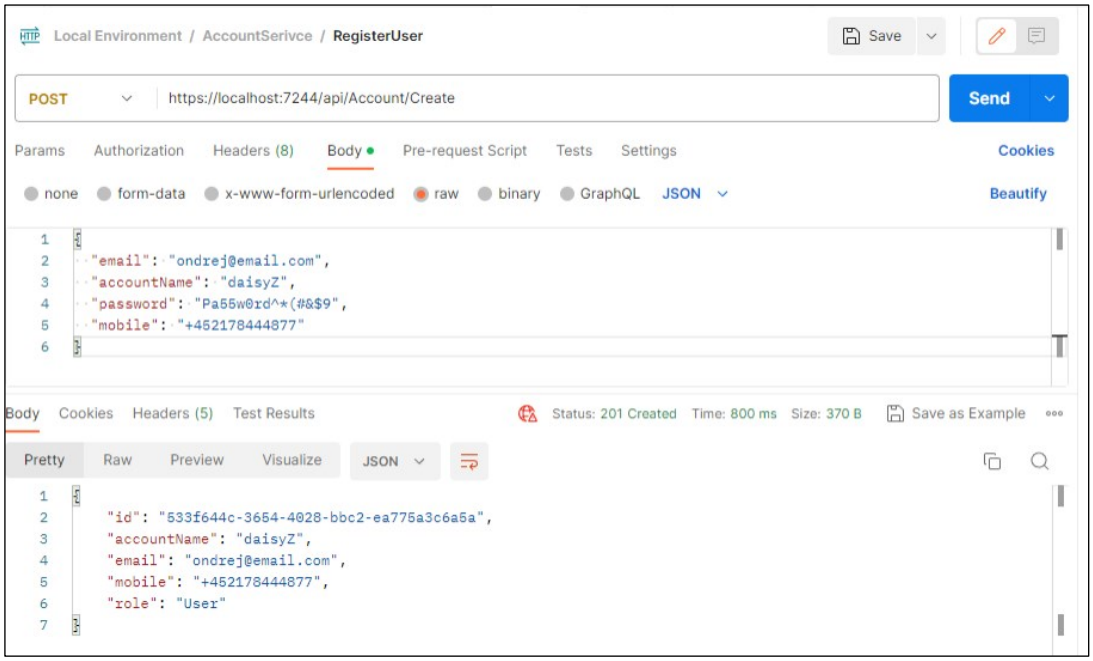
```
[Route("api/[controller]")]
[ApiController]
public class AccountController : ControllerBase
{
    private readonly IAccountRepository _repository;
    private readonly IMapper _mapper;

    public AccountController(IAccountRepository repository, IMapper mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }
    ...
    ...
    [HttpPost("Create")]
    public ActionResult CreateAccount(AccountRegisterDto accountCreateDto)
    {
        var accountModel = _mapper.Map<Account>(accountCreateDto);
        var (passwordSalt, passwordHash) =
        Hashing.CreatePasswordHash(accountCreateDto.Password);
        accountModel.PasswordSalt = passwordSalt;
        accountModel.PasswordHash = passwordHash;
        accountModel.Role = "User";

        _repository.CreateAccount(accountModel);
        _repository.SaveChanges();
        var accountReadDto = _mapper.Map<AccountReadDto>(accountModel);

        return CreatedAtRoute(
            nameof(GetAccountById), new { Id = accountReadDto.Id },
            accountReadDto);
    }
}
```

Ukážka kódu 10. Kontroler AccountController s metódou CreateAccount

Po vytvorení kontroleru je aplikácia spustená lokálne a otestovaná pomocou aplikácie Postman. Po spustení sa zobrazí konzolové okno v ktorom možno vidieť adresu a port na ktorom aplikácia beží.



Obrázok 22. Lokálne spustená služba AccountService



Obrázok 23. Požiadavka na registráciu v aplikácii Postman

6.3.2 Služba pre správu ubytovaní

V tejto službe bude využitá MongoDB databáza a preto sa v úvode nainštaluje balíček NuGet - MongoDB.Driver. Tento balíček pridá závislosti potrebné pri práci s MongoDB databázou.

Obrázok 24. NuGet balíček MongoDB.Driver

Následne je vytvorený dátový model typu Accomodation ktorý slúži na reprezentáciu dokumentov v kolekcii ubytovaní v MongoDB databáze a umožňuje manipuláciu s týmito

dokumentami pomocou MongoDB Driver. Atribúty tohto modelu budú obsahovať informácie o ubytovaní ako napríklad:

- **Id**: Slúži na identifikáciu jednotlivých objektov ubytovania.
- **HostUserId**: Určuje ID hostiteľa, ktorý vlastní toto ubytovanie. Pomocou tohto identifikátora je možné priradiť ubytovanie k určitému hostiteľovi. Tento parameter bude povinný.
- **TitleName**: Obsahuje názov ubytovania. Táto vlastnosť poskytuje informácie o tom, ako je ubytovanie identifikované alebo označené. Tento parameter bude povinný.
- **ShortDescription**: Poskytuje krátky popis ubytovania. Táto vlastnosť je voliteľná a slúži na poskytnutie stručného opisu ubytovania.
- **Country**: Určuje krajinu, v ktorej sa ubytovanie nachádza. Táto vlastnosť poskytuje informácie o mieste, kde sa ubytovanie nachádza. Tento parameter bude povinný.
- **City**: Určuje mesto, v ktorom sa ubytovanie nachádza. Pomocou tejto vlastnosti je možné špecifikovať presnú lokalitu ubytovania. Tento parameter bude povinný.
- **AwailableBedsCount**: Udáva počet dostupných postelí v ubytovaní. Táto vlastnosť slúži na informovanie o kapacite ubytovania a disponibilite postelí. Tento parameter bude povinný.

Pre jednotlivé atribúty možno použiť aj označenia ktoré definujú ich vlastnosti pre prostredie databázy MongoDB. Napríklad [BsonId] označuje primárny identifikátor objektu, [BsonRequired] označuje že konkrétny atribút je povinné zadať a [BsonRepresentation(BsonType.String)] označuje, že atribút je dátového typu string.

```
public class Accomodation
{
    [BsonId, BsonElement("_id"), BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; } = string.Empty;

    [BsonElement("HostUserId"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string HostUserId { get; set; }

    [BsonElement("TitleName"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string TitleName { get; set; }

    [BsonElement("ShortDescription"), BsonRepresentation(BsonType.String)]
    public string? ShortDescription { get; set; }

    [BsonElement("Country"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string Country { get; set; }

    [BsonElement("City"), BsonRequired, BsonRepresentation(BsonType.String)]
    public string City { get; set; }

    [BsonElement("AwailableBedsCount"), BsonRequired,
    BsonRepresentation(BsonType.Decimal128)]
    public decimal AwailableBedsCount { get; set; }
}
```

Ukážka kódu 11 Model Accomodation

Pre prácu s MongoDB databázou je vytvorené rozhranie IAccomodationsDbSettings ktoré implementuje trieda AccomodationsDbSettings. Táto trieda obsahuje potrebné informácie pre pripojenie k databáze – názov kolekcie, názov databázy a reťazec pripojenia. Tieto nastavenia budú uložené v súbore appsettings.Development.json pre lokálne nastavenie. Pomocou dependency injection ich budú injektované v AccomodationRepository repozitári.

```
"AccomodationsDbSettings": {
    "CollectionName": "accomodations",
    "ConnectionString": "mongodb://localhost:27017/dms_accomodations",
    "DatabaseName": "dms_accomodations"
  }
```

Ukážka kódu 12. Hodnoty nastavení uložené v súbore appsettings.Development.json

```
public class AccomodationsDbSettings : IAccomodationsDbSettings
{
    public string CollectionName { get; set; } = String.Empty;
    public string ConnectionString { get; set; } = String.Empty;
    public string DatabaseName { get; set; } = String.Empty;
}
```

Ukážka kódu 13. Objekt AccomodationsDbSettings

```
// set AccomodationsDbSettings to data from Configuration
builder.Services.Configure<AccomodationsDbSettings>(
    builder.Configuration.GetSection(nameof(AccomodationsDbSettings)));

// create singleton IAccomodationsDbSettings for Dep. Injection
builder.Services.AddSingleton<IAccomodationsDbSettings>(sp =>
sp.GetRequiredService<IOptions<AccomodationsDbSettings>>().Value);

// create singleton IMongoClient for Dep. Injection
builder.Services.AddSingleton<IMongoClient>(
s => new MongoClient
(builder.Configuration["AccomodationsDbSettings:ConnectionString"]));

builder.Services.AddScoped<IAccomodationRepository,
AccomodationRepository>();
```

Ukážka kódu 14. Registrovanie rozhraní v Program.cs pre využitie dependency injection

Na oddelenie logiky prístupu k dátam od ostatných častí aplikácie je využitý repository pattern. Hlavným cieľom je poskytnúť jednotné rozhranie pre prístup k dátam, čím sa v budúcnosti zjednoduší údržba, testovanie a výmena zdrojov dát. Toto rozhranie definuje metódy, ktoré poskytujú základné operácie prístupu k dátam.

Konštruktor triedy AccomodationRepository injektuje objekt nastavení databázy a pomocou IMongoClient sa získa kolekcia výletov z databázy.

Následne je vytvorená metóda na vytvorenie nového ubytovania a jeho uloženie do MongoDB kolekcie, metóda ktorá vráti ubytovanie s konkrétnym identifikátorom (Id) alebo aj asynchrónna metóda, určená na zmenu počtu dostupných postelí v ubytovaní ktorá vykoná update v MongoDB kolekcii na základe identifikátora ubytovania a nového počtu postelí.

```
public class AccomodationRepository : IAccomodationRepository
{
    private readonly IMongoCollection<Accomodation> _accomodationsCollection;

    public AccomodationRepository(IAccomodationsDbSettings dbSettings,
       IMongoClient mongoClient)
    {
        var database = mongoClient.GetDatabase(dbSettings.DatabaseName);
             _accomodationsCollection =
            database.GetCollection<Accomodation>(dbSettings.CollectionName);
    }
    public void CreateAccomodation(Accomodation accomodation)
    {
        _accomodationsCollection.InsertOne(accomodation);
    }
    public IEnumerable<Accomodation> GetAllAccomodations()
    {
       return
      _accomodationsCollection.Find(Builders<Accomodation>.Filter.Empty)
      .ToList();
    }
    public Accomodation GetAccomodationById(string accomodationId)
    {
            var filterDefinition = Builders<Accomodation>.Filter.Eq(x => x.Id,
            accomodationId);
            return
            _accomodationsCollection.Find(filterDefinition).SingleOrDefault();
    }
    public string ChangeBedCount(string accomodationId, int changeNumber,
         bool incoming)
    {
        Accomodation accomodation = GetAccomodationById(accomodationId);
        var currentBeds = accomodation.AwailableBedsCount;

        if (incoming) currentBeds -= changeNumber;
        else currentBeds += changeNumber;

        var filter = Builders<Accomodation>.Filter.Eq(x => x.Id,
         accomodationId);
        var update = Builders<Accomodation>.Update.Set(prop =>
         prop.AwailableBedsCount, currentBeds);
        UpdateResult updateResult = _accomodationsCollection.UpdateOne(filter,
         update);

        if (updateResult.ModifiedCount > 0)
        {
            return "OK";
        }
        else
        {
            return "No changes were made!";
        }
    }
}
```

Ukážka kódu 15. Trieda AccomodationRepository

Ďalším krokom je vytvorenie DTO pre požiadavku Create, ktorý bude odosielaný z vonkajšieho prostredia a bude obsahovať potrebné property pre vytvorenie objektu typu Accomodation.

Rovnako je vytvorený DTO pre požiadavku Read, ktorý bude odosielaný z vnútorného prostredia smerom von a bude obsahovať rovnaké property ako pôvodný Accomodation. V tomto prípade je možné použiť na prenos dát aj pôvodný objekt ale pre dodržanie vzoru použijem objekt DTO.

```
public class AccomodationCreateDto
{
    public string HostUserId { get; set; }
    public string TitleName { get; set; }
    public string Country { get; set; }
    public string City { get; set; }
    public int AwailableBedsCount { get; set; }
}
```

Ukážka kódu 16. DTO vytvorený pre požiadavku Create

```
public class AccomodationReadDto
{
    public string Id { get; set; }
    public string HostUserId { get; set; }
    public string TitleName { get; set; }
    public string? ShortDescription { get; set; }
    public string Country { get; set; }
    public string City { get; set; }
    public int AwailableBedsCount { get; set; }
}
```

Ukážka kódu 17. DTO vytvorený pre požiadavku Read

Pre prenos dát o zmene počtu lôžok v ubytovaní je vytvorený DTO ktorý bude niesť dáta potrebné pre vykonanie tejto operácie.

```
public class BedCountChangeDto
{
    public string Id { get; set; }
    public int ChangeValue { get; set; }
    public bool Incoming { get; set; }
}
```

Ukážka kódu 18. DTO BedCountChangeDto

Pre mapovanie objektov v rámci služby je vytvorená tireda AccomodationsProfile ktorá implementuje Profile zdedený od AutoMapper a bude niesť informácie o vzájomných mapovaniach jednotlivých objektov.

```
public class AccomodationsProfile : Profile
{
    public AccomodationsProfile()
    {
        //Source -> target
        CreateMap<Accomodation, AccomodationReadDto>();
        CreateMap<AccomodationCreateDto, Accomodation>();
    }
}
```

Ukážka kódu 19. Objekt AccomodationsProfile

```
//register auto mapper
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

Ukážka kódu 20. Pre fungovanie AutoMapper je nutné ho zaregistrovať v Program.cs

Pre obsluhu požiadaviek je vytvorený kontroler ktorý bude obsluhovať prichádzajúce požiadavky a vykonávať zmeny v databázy prostredníctvom _repository. V konštruktori sú injektované IAccomodationRepository a IMapper. Tieto závislosti sa používajú na komunikáciu s úložiskom (repository) a mapovanie dát.

```
[Route("api/[controller]")]
[ApiController]
public class AccomodationsController : ControllerBase
{
    private readonly IAccomodationRepository _repository;
    private readonly IMapper _mapper;

    public AccomodationsController(IAccomodationRepository repository, IMapper
    mapper)
    {
        _repository = repository;
        _mapper = mapper;
    }
}
```

Ukážka kódu 21. Konštruktor kontroleru AccomodationsController

Pre demonštráciu sú vytvorené metódy na pridanie ubytovania, získanie všetkých ubytovaní alebo jedného ubytovania na základe poskytnutého ID a metódu pre zmenu počtu lôžok v ubytovaní.

Metóda **GetAcomodations** vráti všetky ubytovacie zariadenia. Vo vnútri volá metódu GetAllAccomodations z IAccomodationRepository na získanie dát a následne používa IMapper na mapovanie výsledkov na AccomodationReadDto. Výsledok je vrátený ako ActionResult s kódom OK.

Metóda **GetAccomodationById** vráti ubytovacie zariadenie na základe zadaného ID v požiadavke. Po získaní výsledku z funkcie GetAccomodationById používa IMapper na mapovanie výsledku na AccomodationReadDto. Ak je zariadenie nájdené, jeho objekt je vrátený ako ActionResult s kódom OK. Ak zariadenie nie je nájdené, je vrátený kód NotFound.

Metóda **CreateAccomodation** vytvorí nové ubytovacie zariadenie na základe poskytnutých údajov. Používa IMapper na mapovanie AccomodationCreateDto na Accomodation model. Následne volá metódu CreateAccomodation z IAccomodationRepository na uloženie zariadenia do úložiska. Po vytvorení zariadenia je vrátený AccomodationReadDto ako ActionResult s kódom CreatedAtRoute ktorý vráti kód 201 spolu s novo vytvoreným objektom.

Metóda **ChangeBedCount** zmení počet lôžok pre ubytovacie zariadenie s konkrétnym id. Volá metódu ChangeBedCount z IAccomodationRepository na vykonanie zmeny. Ak je zmena úspešná, je vrátený ActionResult s kódom OK a správou "Reservation accepted!". Ak zmena nie je úspešná, je vrátený ActionResult s kódom BadRequest a príslušnou chybovou správou.

```
[HttpGet("{id}", Name = "GetAccomodationById")]
public ActionResult<AccomodationReadDto> GetAccomodationById(string id)
{
    var accomodationItem = _repository.GetAccomodationById(id);
    if (accomodationItem != null)
    {
        return Ok(_mapper.Map<AccomodationReadDto>(accomodationItem));
    }
    return NotFound();
}

[HttpPost("Create")]
public async Task<ActionResult<AccomodationReadDto>>
CreateAccomodation(AccomodationCreateDto accomodationCreateDto)
{
    var accomodationModel = _mapper.Map<Accomodation>(accomodationCreateDto);
    _repository.CreateAccomodation(accomodationModel);

    var accomodationReadDto =
    _mapper.Map<AccomodationReadDto>(accomodationModel);

    return CreatedAtRoute(nameof(GetAccomodationById), new { Id =
    accomodationReadDto.Id }, accomodationReadDto);
}

[HttpPatch("ChangeBedCount")]
public ActionResult ChangeBedCount(BedCountChangeDto changeBedCount)
{
    var returnMesage = _repository.ChangeBedCount(changeBedCount.Id,
    changeBedCount.ChangeValue, changeBedCount.Incoming);
    if (returnMesage == "OK")
    {
        return Ok("Reservation accepted!");
    }
    else return BadRequest(returnMesage);
}
```

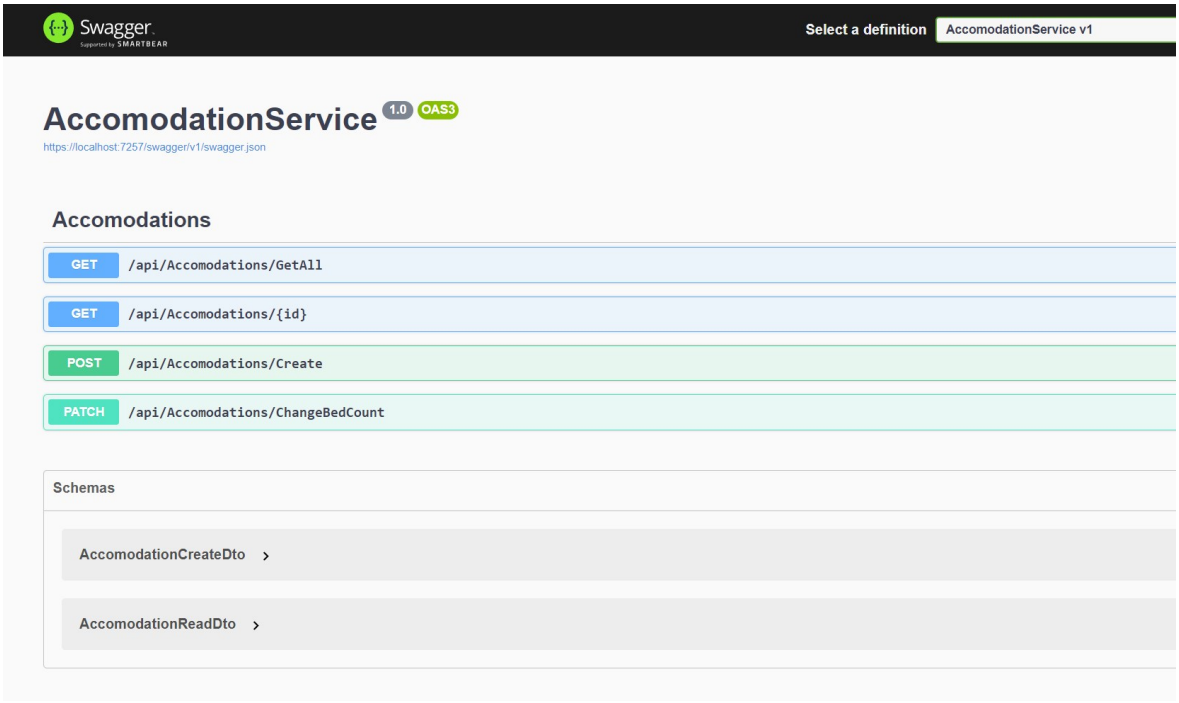
Ukážka kódu 22. Metódy na kontroleru služby Accomodation

Po vytvorení kontroleru je služba spustená lokálne. Zobrazí sa príkazový riadok v ktorom vidieť že služba beží a je možné vykonať požiadavky na adresu https://localhost:7257.

```
--> Seeding data
info: Microsoft.EntityFrameworkCore.Update[30100]
      Saved 2 entities to in-memory store.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7257
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5186
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
```

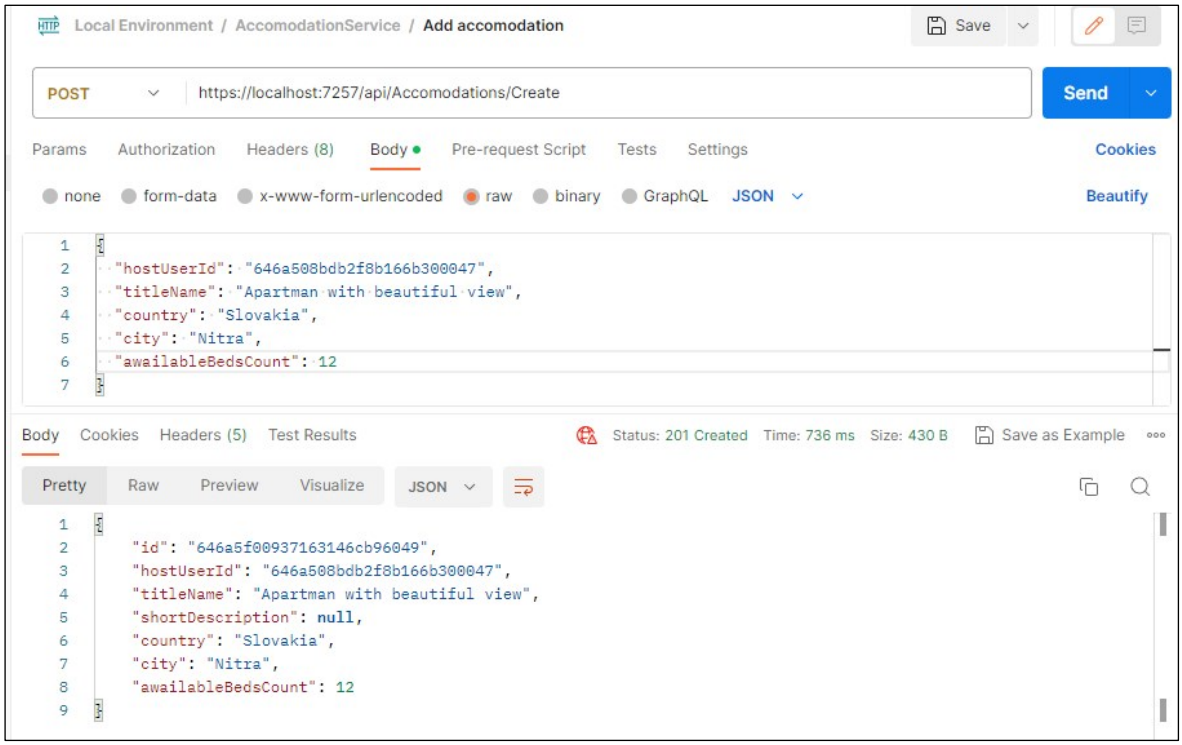
Obrázok 25. Spustenie služby AccomodationService

Keďže pri vytváraní služby bola ponechaná podporu OpenAPI, jednotlivé endpointy je možné skontrolovať aj cez prostredie Swagger.

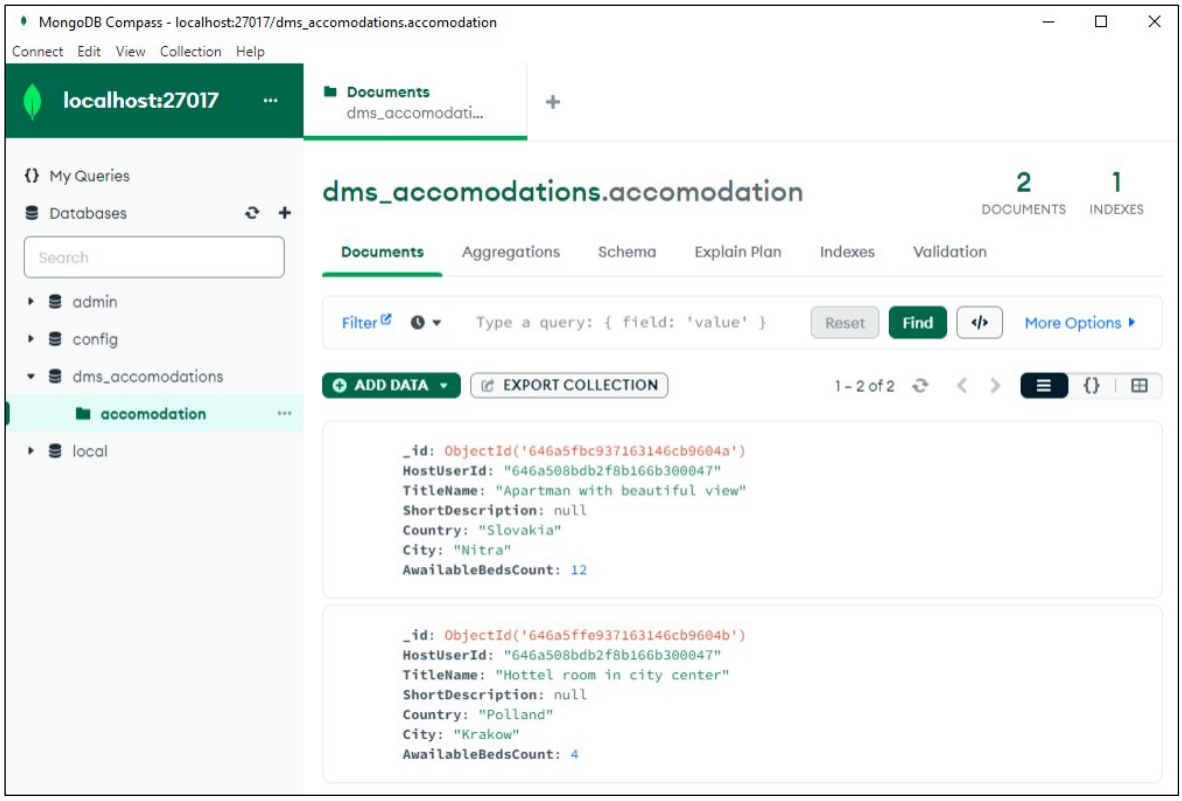


Obrázok 26. Prostredie Swagger s definovanými endpointami

Pre prehľadnejšie testovanie endpointov je použitá aplikácia Postman pomocou ktorej je možné nadefinovať jednotlivé požiadavky a uložiť ich pre budúce použitie. Po zvolení typu POST je zadaná príslušná adresa požiadavky. Do políčka Body sa vložia hodnoty pre AccomodationCreateDto DTO. Služba vráti odpoveď s kódom 201 spolu s novo vytvoreným objektom tak je definovaná v metóde.



Obrázok 27. Požiadavka na vytvorenie ubytovania v aplikácii Postman



Obrázok 28. Vytvorený záznam ubytovania je možné skontrolovať v databáze pomocou aplikácie MongoDB Compass.

6.3.3 Služba pre správu výletov

Implementáca služby pre správu výletov začína vytvorením modelu Trip ktorá bude obsahovať property ako napríklad:

- **Id**: Identifikátor výletu. Používa sa ako unikátny identifikátor v kolekcii MongoDB ktorý bude generovať databáza.
- **TitleName**: Názov výletu. Slúži na uchovávanie názvu výletu. Tento parameter bude povinný.
- **Description**: Popis výletu. Umožňuje uchovávanie textového popisu výletu.
- **Location**: Miesto konania výletu. Uchováva informáciu o mieste, kde sa výlet uskutočňuje.
- **FreeOfCost**: Indikátor, či je výlet bezplatný. True znamená, že výlet je bezplatný, false znamená, že nie je.
- **Price**: Cena výletu. Umožňuje uchovávanie hodnoty ceny výletu.
- **MaxPersons**: Maximálny počet účastníkov výletu. Určuje maximálny počet osôb, ktoré sa môžu zúčastniť výletu.
- **Highlights**: Zvýraznenia výletu. Je to zoznam reťazcov, ktoré obsahujú zvýraznené vlastnosti alebo body týkajúce sa výletu.
- **CreatorId**: Identifikátor tvorcu výletu. Umožňuje identifikovať tvorcu výletu pomocou unikátneho identifikátora. Tento parameter bude povinný.
- A ďalšie

Ďalšie property môžu byť pridané neskôr v priebehu implementácie. Keďže služba využíva databázu MongoDB, je možné použiť dostupné anotácie pre tento typ databázy.

```
[Serializable, BsonIgnoreExtraElements]
public class Trip
{
    [BsonId, BsonElement("_id"), BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }

    [BsonElement("title_name"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string TitleName { get; set; }

    [BsonElement("description"), BsonRepresentation(BsonType.String)]
    public string Description { get; set; } = string.Empty;

    [BsonElement("location"), BsonRepresentation(BsonType.String)]
    public string Location { get; set; } = string.Empty;

    [BsonElement("free_of_cost"), BsonRepresentation(BsonType.Boolean)]
    public bool FreeOfCost { get; set; } = false;

    [BsonElement("price"), BsonRepresentation(BsonType.Decimal128)]
    public decimal Price { get; set; } = decimal.Zero;

    [BsonElement("max_persons"), BsonRepresentation(BsonType.Decimal128)]
    public int MaxPersons { get; set; } = 4;

    [BsonElement("Highlights")]
    public List<string> Highlights { get; set; } = new List<string>();

    [BsonElement("CreatorId"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string CreatorId { get; set; } = string.Empty;
}
```

Ukážka kódu 23. Model Trip s anotáciami

Pre prácu s MongoDB databázou bolo vytvorené rozhranie ITripsDbSettings, ktoré je implementované triedou TripsDbSettings. Táto trieda obsahuje potrebné informácie pre pripojenie k databáze, ako je názov kolekcie, názov databázy a reťazec pripojenia. Tieto nastavenia sú uložené v súbore appsettings.Development.json pre lokálne nastavenie. pomocou dependency injection sú tieto nastavenia používané v repozitári TripRepository.

```
public class TripsDbSettings : ITripsDbSettings
{
    public string CollectionName { get; set; } = String.Empty;
    public string ConnectionString { get; set; } = String.Empty;
    public string DatabaseName { get; set; } = String.Empty;
}

-----------------------------------------------------------------------------

"TripsDbSettings": {
    "CollectionName": "trip",
    "ConnectionString": "mongodb://localhost:27017/dms_trips",
    "DatabaseName": "dms_trips"
  }
```

Ukážka kódu 24. Trieda TripsDbSettings ktorá odpovedá nastaveniam

```
// set TripDbSettings to data from Configuration
builder.Services.Configure<TripsDbSettings>(
    builder.Configuration.GetSection(nameof(TripsDbSettings)));

// create singleton ITripsDbSettings for Dep. Injection
builder.Services.AddSingleton<ITripsDbSettings>(sp =>
sp.GetRequiredService<IOptions<TripsDbSettings>>().Value);

// create singleton IMongoClient for Dep. Injection
builder.Services.AddSingleton<IMongoClient>(
    s => new
MongoClient(builder.Configuration["TripsDbSettings:ConnectionString"]));

builder.Services.AddScoped<ITripRepository, TripRepository>();
```

Ukážka kódu 25. Registrovanie rozhraní v Program.cs pre využitie dependency injection

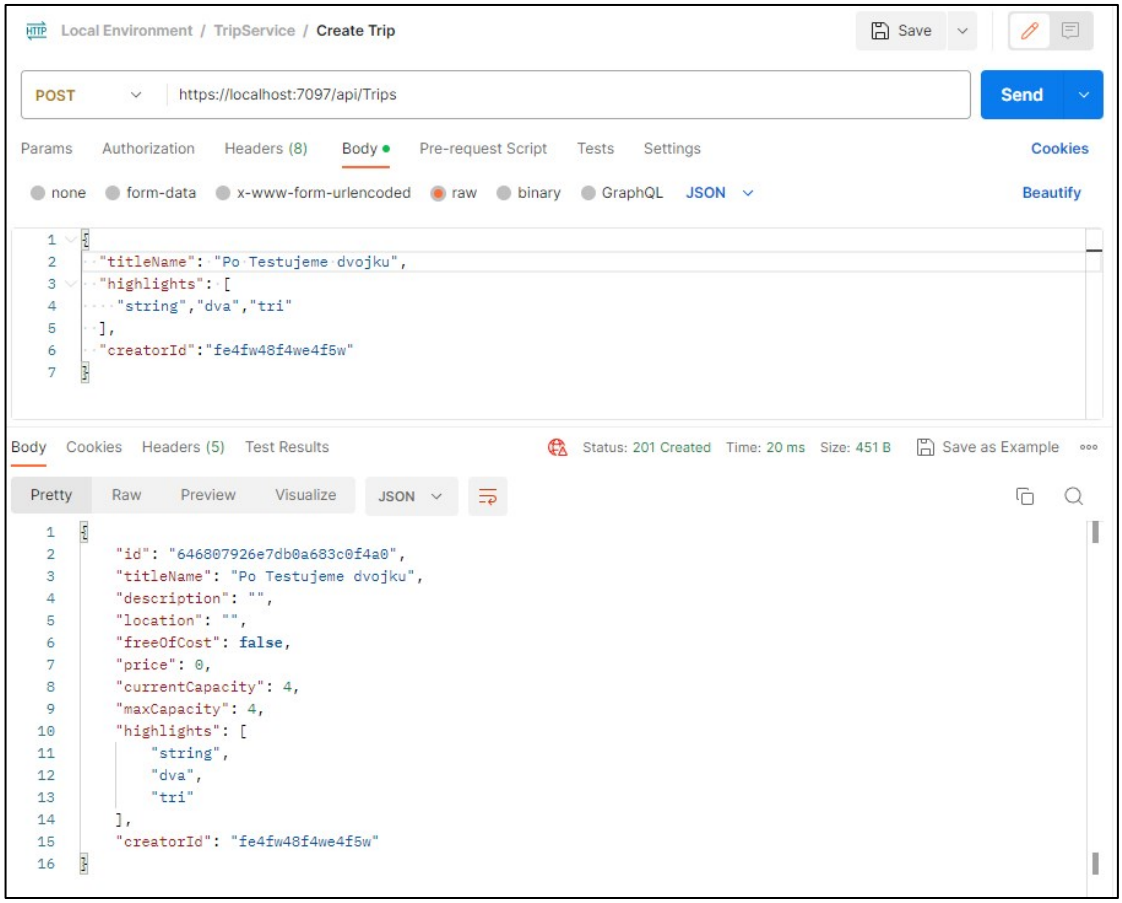
Na oddelenie logiky prístupu k dátam od ostatných častí aplikácie sa opäť využije repository. Je vytvorená tireda TripRepository ktorá implementuje rozhranie jednotlivých metód ktoré pre obsluhu práce s dátami a databázou. V konštruktore je injektovaný objekt nastavení databázy a pomocou IMongoClient sa získa z databázy kolekcia výletov.

```
public class TripRepository : ITripRepository
{
   private readonly IMongoCollection<Trip> _tripsCollection;

   public TripRepository(ITripsDbSettings dbSettings, IMongoClient mongoClient)
   {
       var database = mongoClient.GetDatabase(dbSettings.DatabaseName);
       _tripsCollection = database.GetCollection<Trip>(dbSettings.CollectionName);
   }

   ...
```

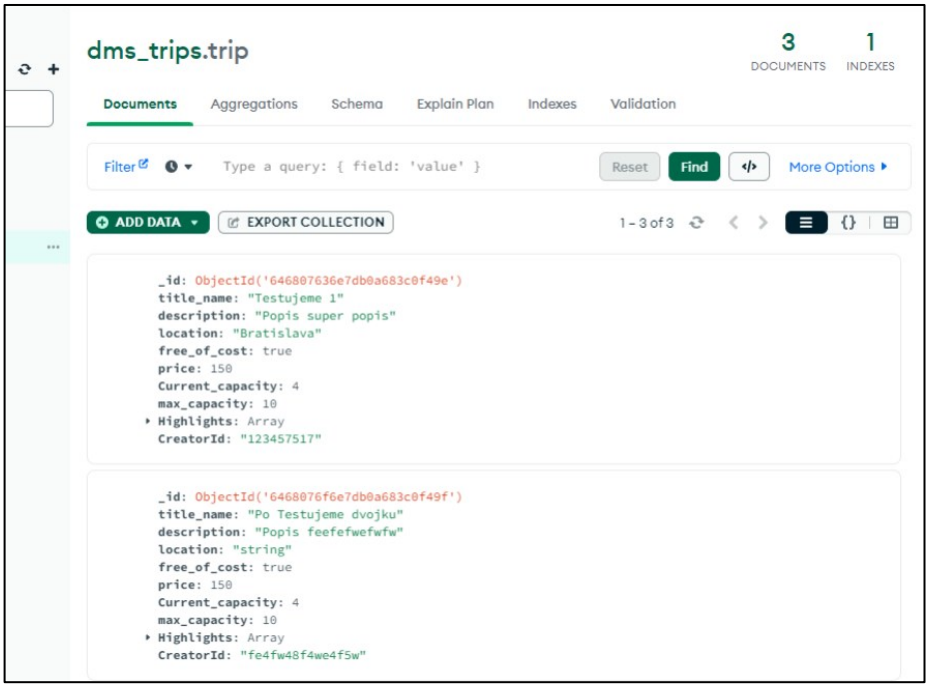
Ukážka kódu 26. Konštruktor TripRepository

Pre obsluhu požiadaviek je vytvorený kontroler, v ktorom sú vytvorené jednoduché metódy pre vytváranie, upravovanie ,mazanie výletov a získanie zoznamu všetkých výletov alebo jednotlivých podľa ID.

Po vytvorení kontroleru je spustená služba TripService lokálne. Pomocou Swager prostredia si možno skontrolovať či sú k dispozícii všetky endpointy a pomocou Postman budú otestované. Zvolí sa metóda POST a pridá sa príslušná adresa požiadavky. Do políčka Body sú vložené údaje pre vytvorenie výletu. Na obrázku 29. je vidieť že endpoint funguje aj bez poskytnutia všetkých údajov, čo je správne keďže vyžadované sú iba titleName a creatorId.



Obrázok 29. Testovanie endpointu vytvárania výletu.



Obrázok 30. Záznamy v databáze možno kontrolovať pomocou aplikácie MongoDB Compas

6.3.4 Služba pre správu rezervácií

Model služby pre správu rezervácií obsahuje property ako ID ubytovania, ID výletu – tieto môžu byť zadané oboje v prípade že chce používateľ absolvovať aj ponúkaný výlet k ubytovaniu. Ďalšie potrebné property sú ID hostiteľa ktorý ponúka výlet s ubytovaním, a ID používateľa ktorý vytvára rezerváciu. Model pri má pri vytvorení aj inicializovanú property s názvom HostAccepted na hodnotu false, ktorá slúži na prijatie rezervácie hostiteľom.

```
public class Reservation
{
    [BsonId, BsonElement("_id"), BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; } = string.Empty;
    [BsonElement("AccomodationId"), BsonRepresentation(BsonType.String)]
    public string? AccomodationId { get; set; }
    [BsonElement("TripId"), BsonRepresentation(BsonType.String)]
    public string? TripId { get; set; }
    [BsonElement("HostUserId"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string HostUserId { get; set; }
    [BsonElement("ReservingUserId"), BsonRequired,
    BsonRepresentation(BsonType.String)]
    public string VisitorUserId { get; set; }
    [BsonElement("PeopleCount"), BsonRequired,
    BsonRepresentation(BsonType.Decimal128)]
    public decimal PeopleCount { get; set; }
    [BsonElement("DateStart"), BsonRepresentation(BsonType.DateTime)]
    public DateTime DateStart { get; set; }
    [BsonElement("DateEnd"), BsonRepresentation(BsonType.DateTime)]
    public DateTime DateEnd { get; set; }
    [BsonElement("HostAccepted"), BsonDefaultValue(false),
    BsonRepresentation(BsonType.Boolean)]
    public bool HostAccepted { get; set; } = false;
}
```

Ukážka kódu 27 Model Reservation

Pre prácu s MongoDB databázou je vytvorené rozhranie pre nastavenia ktoré budú definované v súbore appsettings.Development.json pre lokálne nastavenie. Pomocou dependency injection sa budú používať v ReservationRepository repozitári.

```
"ReservationsDbSettings": {
    "CollectionName": "trip",
    "ConnectionString": "mongodb://localhost:27017/dms_trips",
    "DatabaseName": "dms_trips"
  }
```

Ukážka kódu 28. Nastavenia databázy pre službu rezervácií

Na oddelenie logiky prístupu k dátam od ostatných častí aplikácie bude slúžiť repository trieda ReservationRepository ktorá implementuje rozhranie jednotlivých metód určených na prácu s dátami a databázou..

Konštruktor triedy ReservationRepository prijíma dva parametre: dbSettings typu IReservationsDbSettings a mongoClient typu IMongoClient. Tieto parametre slúžia na nastavenie pripojenia k MongoDB databáze a získanie prístupu k požadovanej kolekcii rezervácií.

```
public class ReservationRepository : IReservationRepository
{
    private readonly IMongoCollection<Reservation> _reservationsCollection;

      public ReservationRepository(IReservationsDbSettings dbSettings,
      IMongoClient mongoClient)
      {
          var database = mongoClient.GetDatabase(dbSettings.DatabaseName);
            _reservationsCollection =
            database.GetCollection<Reservation>(dbSettings.CollectionName);
      }
      public void CreateReservation(Reservation reservation)
      {
          _reservationsCollection.InsertOne(reservation);
      }
      public async Task<Reservation> GetReservationById(string reservationId)
      {
          var filterDefinition = Builders<Reservation>.Filter.Eq(x => x.Id,
          reservationId);
          return await
      _reservationsCollection.Find(filterDefinition).SingleOrDefaultAsync();
      }

      public async Task<string> HostAcceptReservation(string reservationId)
      {
          var filter = Builders<Reservation>.Filter.Eq(x => x.Id,
          reservationId);
          var update = Builders<Reservation>.Update.Set(prop =>
          prop.HostAccepted, true);
          UpdateResult updateResult = await
          _reservationsCollection.UpdateOneAsync(filter, update);

          if (updateResult.ModifiedCount > 0) return "OK";
          else return "No changes were made!";
      }
}
```

Ukážka kódu 29. ReservationRepository

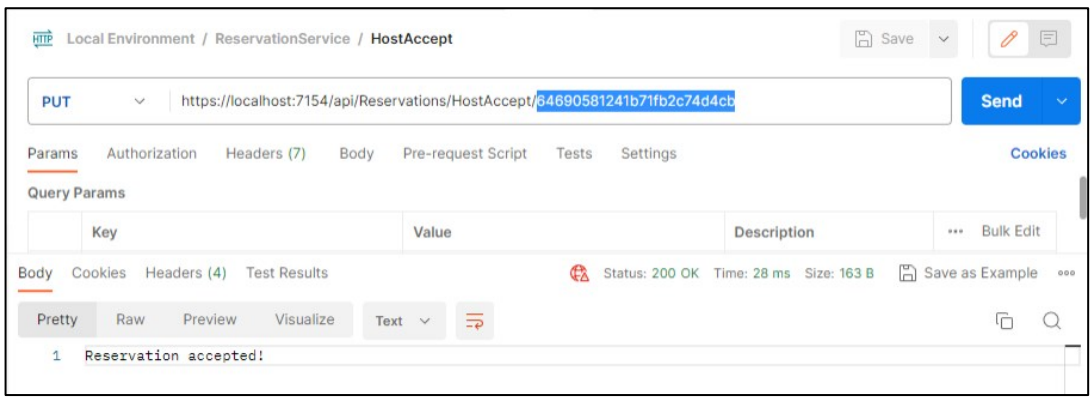
Následne je vytvorený kotroler pre obsluhu požiadaviek pre manipuláciu s rezerváciami. Konštruktor kontroleru ReservationsController prijíma závislosť na rozhraní IReservationRepository a injektuje ho pomocou dependency injection.

Metóda **GetReservationById** slúži na získanie rezervácie s konkrétnym identifikátorom reservationId. Metóda volá GetReservationById rozhrania IReservationRepository a vráti získanú rezerváciu ako výsledok HTTP požiadavky s kódom 200 OK. Ak rezervácia neexistuje, vráti sa odpoveď s kódom 404 Not Found.

Metóda **CreateTrip** je označená atribútom [HttpPost("Create")] a slúži na vytvorenie novej rezervácie. Metóda volá CreateReservation rozhrania IReservationRepository a následne vráti vytvorenú rezerváciu ako výsledok HTTP požiadavky s kódom 201 Created. Okrem toho je vráti dáta vytvorenej rezervácie.

Metóda na potvrdenie rezervácie zo strany hostiteľa s názvom **HostAcceptReservation** je označená atribútom [HttpPut("HostAccept/{reservationId}")]. Táto metóda volá HostAcceptReservation rozhrania IReservationRepository ktorá Metóda vykoná update dokumentu v kolekcii, kde nastaví vlastnosť HostAccepted na hodnotu true. Na základe vrátenej správy určuje, či zmena v databáze prebehla úspešne alebo nie. V prípade úspechu sa vráti odpoveď „No changes were made!“ s kódom 200 OK a správou „Reservation accepted!“. V opačnom prípade sa vráti odpoveď s kódom 400 Bad Request a vrátenou správou z metódy HostAcceptReservation rozhrania IReservationRepositorz.

Testovanie lokálne spustenej služby ReservationService je vykonávané v aplikácii Postman pomocou ktorej sa odošle požiadavka na vytvorenie rezervácie. Následne sa otestuje akceptovanie rezervácie pomocou požiadavky typu put, do ktorej sa zadá ID už vytvorenej rezervácie ktorá nebola zatiaľ akceptovaná. Na obrázku 31. možno vidieť že aplikácia odpovedala Reservation accepted a teda rezervácia bola úspešná.



Obrázok 31. Testovanie end pointu určeného na akceptovanie rezervácie

```
[Route("api/[controller]")]
[ApiController]
public class ReservationsController : ControllerBase
{
   private readonly IReservationRepository _reservationRepository;

   public ReservationsController(IReservationRepository
       reservationRepository)
   {
       _reservationRepository = reservationRepository;
   }
   [HttpGet("{reservationId}", Name = "GetReservationById")]
   public async Task<ActionResult<Reservation>> GetReservationById(string
       reservationId)
   {
       var reservationItem = await
       _reservationRepository.GetReservationById(reservationId);
       if (reservationItem != null)
       {
           return Ok(reservationItem);
       }
       else return NotFound();
   }

   [HttpPost("Create")]
   public async Task<ActionResult<Reservation>>
       CreateReservation(Reservation reservation)
   {
       _reservationRepository.CreateReservation(reservation);
       return CreatedAtRoute(nameof(GetReservationById), new { reservationId
       = reservation.Id }, reservation);
   }

   [HttpPut("HostAccept/{reservationId}")]
   public async Task<ActionResult> HostAcceptReservation(string
       reservationId)
   {
       string resultMessage = await
       _reservationRepository.HostAcceptReservation(reservationId);
       if (resultMessage == "OK")
       {
           return Ok("Reservation accepted!");
       }
       else return BadRequest(resultMessage);
   }
}
```

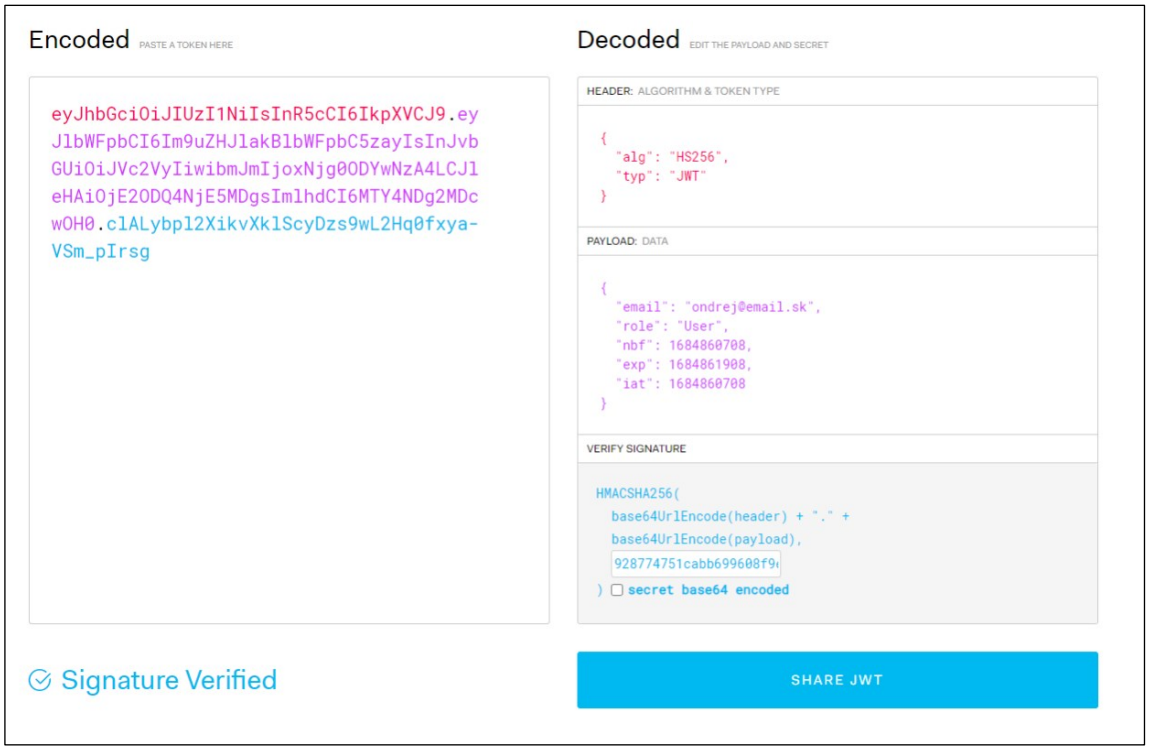
Ukážka kódu 30. Kontroler ReservationsController

6.4 Autentifikácia

V tejto časti je popísaná implementácia autentifikačnej služby ktorá bude fungovať na princípe poskytovania JWT (JSON Web Token) bearer tokenu. Tento token sa používateľovi vytvorí po úspešnom prihlásení. V reálnej situácii by sa tento token uložil v prehliadači a používateľ by sa ním autentifikoval pokým token nevyprší.

Pre zašifrovanie a dešifrovanie celého obsahu tokenu sa používa podpisový kľúč. Token vo svojom obsahu môže niesť ľubovoľné dáta ktoré je možné získať a použiť priamo bez volania ďalších služieb. Token taktiež nesie údaj o svojej exspirácii, čo možno využiť na kontrolu, či čas autentifikácie nevypršal.

Pre využitie autentifikácie je nutné nastaviť autentifikačnú metódu v nastaveniach programu. Definujú sa pravidlá a zadá sa podpisový kľúč. Jednotlivé obslužné metódy potom prijímajú požiadavky spolu s autentifikačným tokenom pred spustením na základe nastavení overia.



Obrázok 32. Zloženie JWT tokenu, overenie tokenu prebieha na základe zadaného podpisového kľúča

6.4.1 Služba pre správu autentifikácie

Pre prácu s Jwt tokenom je použitý NuGet balíček System.IdentityModel.Tokens.Jwt

Obrázok 33. NuGet balíčk System.IdentityModel.Tokens.Jwt

Na začiatku je vytvorená repository trieda JwtAuthenticationManager ktorá implementuje rozhranie IJwtAuthenticationManager. Táto trieda obsahuje jedinú metódu - Authenticate(string userEmail, string userPassword), ktorá slúži na autentifikáciu používateľa na základe poskytnutého e-mailu a hesla pri prihlásení. Predpokladá sa, že používateľ bude validovaný pomocou databázy avšak pre demonštračné účely je táto validácia, či sa používateľ správne prihlásil, vynechaná a nahradená komentárom v mieste validácie.

Ďalšie kroky v metóde zahŕňajú vytvorenie JWT tokenu. Vytvára sa objekt JwtSecurityTokenHandler, ktorý je zodpovedný za správu a generovanie JWT tokenov.

Kľúč pre podpisovanie tokenov sa získava zo IJwtSettings objektu, ktorý je injektovaný pomocou dependency injection. Okrem kľúča obsahuje tento objekt aj údaj o expiračnej dobe. Oba tieto údaje sú uložené v nastaveniach a inicializujú sa pri spustení programu v triede Program.cs. Podpisový kľúč sa používa na vytvorenie SigningCredentials, ktoré vytvorí podpis algoritmu HMAC-SHA256.

```
// set JwtSettings to data from Configuration
builder.Services.Configure<JwtSettings>(
    builder.Configuration.GetSection(nameof(JwtSettings)));

// create singleton IJwtSettings for Dep. Injection
builder.Services.AddSingleton<IJwtSettings>(sp =>
sp.GetRequiredService<IOptions<JwtSettings>>().Value);

builder.Services.AddScoped<IJwtAuthenticationManager,
JwtAuthenticationManager>();
```

Ukážka kódu 31. Injektovanie objektov v programových nastaveniach

```
public class JwtSettings : IJwtSettings
{
    public int JwtTokenValidityMins { get; set; }
    public string JwtSecurityKey { get; set; }
}
```

Ukážka kódu 32. Objekt JwtSettings

Následne sa vytvára objekt SecurityTokenDescriptor, ktorý obsahuje informácie o používateľovi, exspiráciu tokenu a podpis čo vlastne tvorí celý dátový obsah tokenu. Objekt JwtSecurityTokenHandler potom tento token vytvorí na základe deskriptora.

Nakoniec sa výsledný vygenerovaný token zapíše do reťazca pomocou jwtSecurityTokenHandler.WriteToken(securityToken). Funkcia vráti vygenerovaný token spolu s e-mailom používateľa a časom exspirácie pomocou DTO JwtAuthResponse ako výsledok autentifikácie.

```
public JwtAuthResponse Authenticate(string userEmail, string userPassword)
{
      //v tomto mieste validácia používatela//

      DateTime tokenExpiryTimeStamp =
      DateTime.Now.AddMinutes((double)_jwtSettings.JwtTokenValidityMins);
      var jwtSecurityTokenHandler = new JwtSecurityTokenHandler();
      var tokenKey = Encoding.ASCII.GetBytes(_jwtSettings.JwtSecurityKey);
      var securityTokenDescriptor = new SecurityTokenDescriptor
      {
            Subject = new System.Security.Claims.ClaimsIdentity(new
            List<Claim>
            {
                new Claim(ClaimTypes.Email, userEmail),
                new Claim(ClaimTypes.Role, "User") //-> rolu priradiť z DB
            }),
            Expires = tokenExpiryTimeStamp,
            SigningCredentials = new SigningCredentials(new
            SymmetricSecurityKey(tokenKey),
            SecurityAlgorithms.HmacSha256Signature)
      };
      var securityToken =
      jwtSecurityTokenHandler.CreateToken(securityTokenDescriptor);
      var token = jwtSecurityTokenHandler.WriteToken(securityToken);

      JwtAuthResponse responseObject = new JwtAuthResponse
      {
         Token = token,
         Email = userEmail,
         ExpireIn = (int)tokenExpiryTimeStamp.Subtract(DateTime.Now)
         .TotalSeconds
      };
      return responseObject;
}
```

Ukážka kódu 33. Funkcia Authenticate v JwtAuthenticationManager

V konštruktore kontroleru AuthController je injektovaný JwtAuthenticationManager. Konštruktor obsahuje spomínanú metódu Login ktorá prijíma DTO AccountLoginDto. V tejto metóde sa volaním _authManager.Authenticate overí existencia používateľa a vráti

sa objekt DTO JwtAuthResponse. V prípade že je vrátený objekt prázdny vráti sa odpoveď Unauthorized.

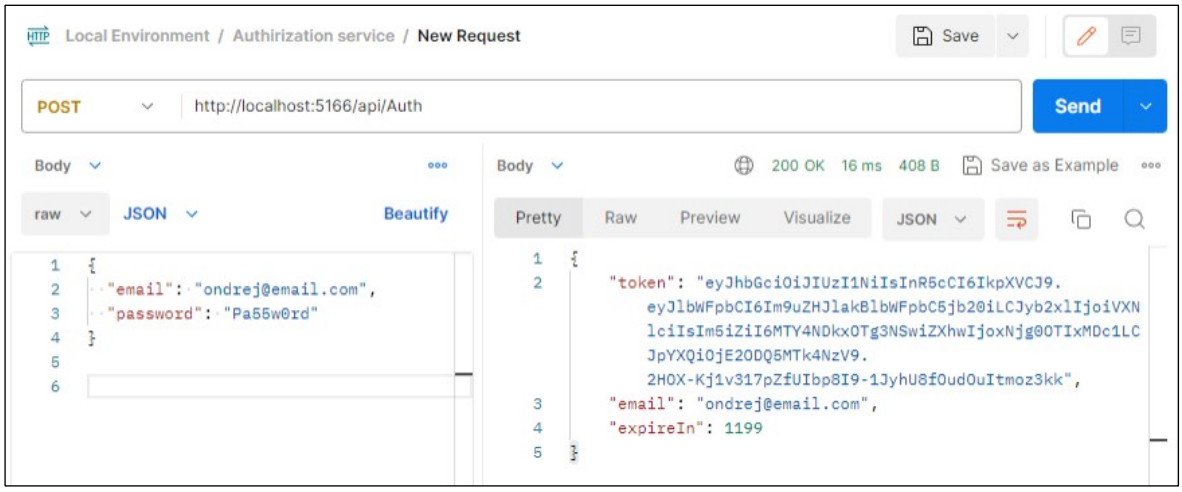
```
public class AccountLoginDto
{
    [Required]
    public string Email { get; set; }
    [Required]
    public string Password { get; set; }
}
```

Ukážka kódu 34. Objekt slúžiaci na prijatie údajov prihlásenia – AccountLoginDto

```
[Route("api /[controller]")]
[ApiController]
public class AuthController : Controller
{
    private readonly IJwtAuthenticationManager _authManager;

    public AuthController(IJwtAuthenticationManager authManager)
    {
        _authManager = authManager;
    }

    [HttpPost]
    public ActionResult Login(AccountLoginDto loginRequest)
    {
         var authResult = _authManager.Authenticate(loginRequest.Email,
         loginRequest.Password);

        if (authResult == null)
        {
            return Unauthorized();
        }
        else return Ok(authResult);
    }
}
```

Ukážka kódu 35. Auth kontroler s metódou Login

Overenie funkčnosti je vykonané v aplikácii Postman kde sa odošle správa s prihlasovacími údajmi. Vrátená správa bude obsahovať vygenerovaný token.



Obrázok 34. Požiadavka na autentifikáciu vracia objekt s tokenom, emailom, a časom exspirácie

6.4.2 Použitie autentifikácie v službách

Pre použitie autentifikácie v jednotlivých službách je potrebné pridať autentifikačnú službu. Keďže sa jedná o repetetívnu činnosť bola je vytvorená abstraktná triedu CustomJwtAuthExtension s abstraktnou metódu AddCustomJwtAuthentication. Táto metóda pridáva autentifikačnú službu do kolekcie služieb IServiceCollection v rámci konfigurácie aplikácie. V rámci tejto metódy je vykonaná konfigurácia autentifikácie pomocou JWT. Taktiež sú nastavené rôzne možnosti autentifikácie a parametre overovania tokenov.

V bloku AddAuthentication je konfigurované, že autentifikácia prebieha pomocou schémy JwtBearerDefaults.AuthenticationScheme, čo znamená, že sa očakáva použitie JWT tokenu pri autentifikácii.

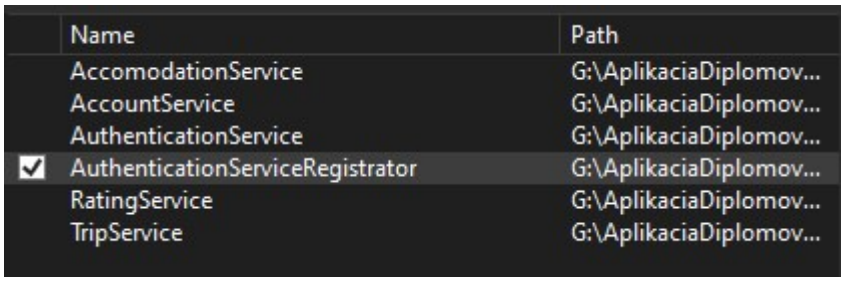
V bloku AddJwtBearer sú nastavené rôzne možnosti overovania a spracovania JWT tokenu:

- **RequireHttpsMetadata** nastavené na false umožňuje autentifikáciu aj v prípade, že komunikácia neprebieha cez HTTPS.
- **SaveToken** nastavené na true znamená, že pri úspešnej autentifikácii sa token uloží do príslušného autentifikačného tokenu.
- **TokenValidationParameters** obsahuje nastavenia pre overovanie tokenu, v tomto prípade sa overuje platnosť podpisového kľúča (IssuerSigningKey) pomocou symetrického kľúča získaného zo JwtSecurityKey.

```
public static void AddCustomJwtAuthentication(this IServiceCollection
services, string JwtSecurityKey)
{
    services.AddAuthentication(options =>
    {
      options.DefaultAuthenticateScheme =
      JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
      JwtBearerDefaults.AuthenticationScheme;
    }).AddJwtBearer(options =>
    {
        options.RequireHttpsMetadata = false;
        options.SaveToken = true;
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            ValidateIssuer = false,
            ValidateAudience = false,
            IssuerSigningKey = new
            SymmetricSecurityKey(Encoding.ASCII.GetBytes(JwtSecurityKey))
        };    });
}
```

Ukážka kódu 36. Statická metóda AddCustomJwtAuthentication

Pre použitie tejto metódy v ostatných službách je potrebné pridať do projektu mikroslužby referenciu na statickú triedu CustomJwtAuthExtension.



Obrázok 35. Pridanie referencie

Následne je potrebné pomocou dependency injection službu registrovať v nastaveniach v Program.cs a taktiež pridť nastavenie app.UseAuthentication(). Podpisový kľúč bude uložený v konfiguračných súboroch pod parametrom JwtSecurityKey.

```
builder.Services.AddCustomJwtAuthentication(builder.Configuration["JwtSecurity
Key"]);
```

```
app.UseAuthentication();
```

Ukážka kódu 37. Registrovanie konfigurácie pridaním AddCustomJwtAuthentication

Pre otestovanie autentifikácie je využitá služba rezervácií kde je v metóde GetAllReservation pridaná anotácia [Authorize]. Metóda sa nevykoná pokiaľ nebude poskytnutý bearer token odpovedajúci nastaveniam a vráti 401 Not authorized.

Keďže Jwt token umožňuje prenášať rôzne dáta je často využívaný aj na autorizáciu. Pri overovaní je používateľovi nastavená rola na základe ktorej sa môže pred spustením tejto metódy autorizovať. Pri procese autentifikácie bola používateľovi zadaná rolu s názvom „User" a teda ak bude potreba využiť autorizáciu možno tak urobiť použitím anotácie so špecifikáciou role [Authorize(Roles = "User")]. Pri takomto nastavení musí byť požívateľ nielen autentifikovaný ale musí spĺňať aj autorizačnú podmienku – v tomto prípade musí mať rolu User. Ak nespĺňa autorizačnú podmienku služba vráti 403 Forbidden. Týmto spôsobom je možné ošetriť jednotlivé end pointy proti neoprávnenému používaniu nielen pomocou rolí ale aj rôznych parametrov ([Authorize(UserEmail = "stefan@test.sk")]), čísla, a iných ľubovoľných parametrov.

```
[Authorize(Roles = "User")]
[HttpGet]
public ActionResult<IEnumerable<Reservation>> GetAllReservations()
{
    var reservations = _reservationRepository.GetAllReservations();
    return Ok(reservations);
}
```

Ukážka kódu 38. Pridaná anotácia autentizácie spolu s autorizačnou podmienkou

6.5 Implementácia komunikácie medzi službami

Komunikácia medzi službami je demonštrovaná pomocou synchrónneho posielania správ. Demonštrácia je zameraná na situáciu kedy hostiteľ prijme požiadavku na výmenu a tým pádom bude musieť byť znížený počet lôžok daného ubytovania.

V službe pre správu rezervácií je vytvorená repository trieda DataClient ktorá bude vykonávať posielanie správ s využitím HTTP klienta.

V konštruktore je injektovaný HTTP klient a rozhranie IConfiguration. Trieda obsahuje metódu s názvom ChangeBedCountOfAccomodation ktorá odošle službe pre správu ubytovaní požiadavku na upravenie počtu lôžok. Požiadavka bude vlastne serializovaný JSON BedCountChangeDto ktorý je vytvorený pre tento účel. Následne sa zachytí odpoveď a vyhodnotí sa pomocou kontroly IsSuccesCode.

```
public class DataClient : IDataClient
{
    private readonly HttpClient _httpClient;
    private readonly IConfiguration _configuration;

    public DataClient(HttpClient httpClient, IConfiguration configuration)
    {
        _httpClient = httpClient;
        _configuration = configuration;
    }

    public async Task ChangeBedCountOfAccomodation(BedCountChangeDto
    bedCountChange)
    {
        var httpContent = new StringContent(
            JsonSerializer.Serialize(bedCountChange),
            Encoding.UTF8,
        "application/json");

        var response = await
        _httpClient.PatchAsync(_configuration["ChangeBedCountUrl"],
        httpContent);

        if (response.IsSuccessStatusCode)
        {
            Console.WriteLine("--> Sync PATCH was OK");
        }
        else
        {
            Console.WriteLine("--> Sync PATCH was not OK");
        }
    }
}
```

Ukážka kódu 39. Trieda DataClient

```
public class BedCountChangeDto
{
    public string Id { get; set; }
    public int ChangeValue { get; set; }
    public bool Incoming { get; set; }
}
```

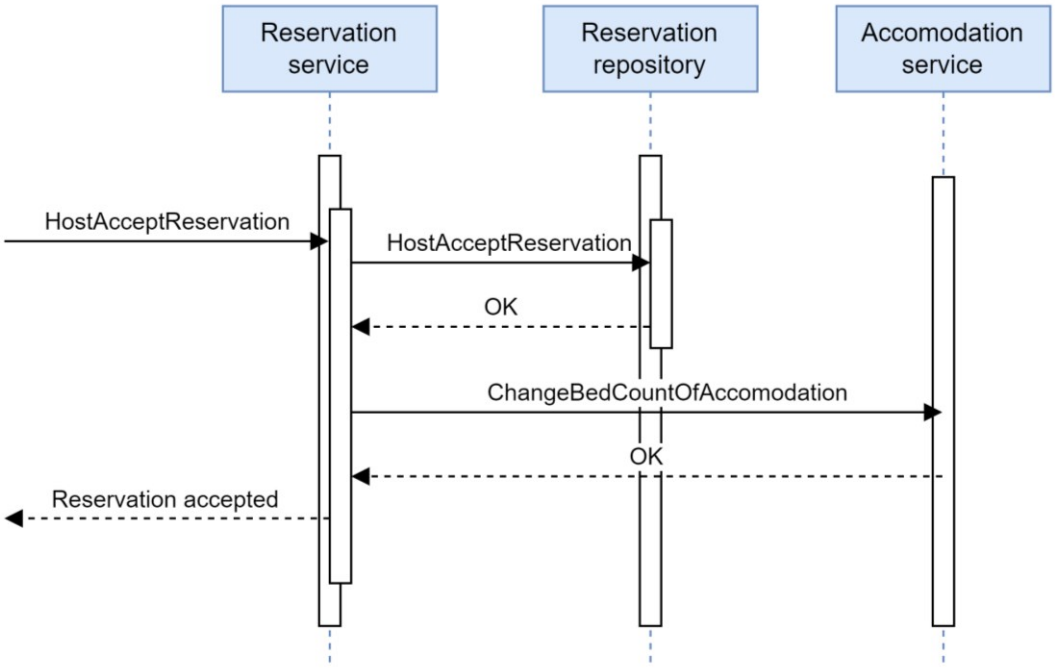
Ukážka kódu 40. BedCountChangeDto

V ďalšom kroku je vytvorená metóda HostAcceptReservation v kontroleri ReservationsController. Táto metóda obslúži požiadavku na akceptovanie rezervácie. Pomocou metódy _repository.HostAceptReservation zmení parameter HostAccepted na true a následne odošle požiadavku na zmenu počtu lôžok v ubytovaní pomocou metódy

_dataClient.ChangeBedCountOfAccomodation ktorá odošle požiadavku do služby pre správu ubytovaní.

```
[HttpPut("HostAccept/{reservationId}")]
public async Task<ActionResult> HostAcceptReservation(string reservationId)
{
    string resultMessage = await
    _reservationRepository.HostAcceptReservation(reservationId);
    if (resultMessage == "OK")
    {
        try
        {
            var reservation = await
            _reservationRepository.GetReservationById(reservationId);

            var changeValue = await
            _reservationRepository.GetReservationGuestCount(reservationId);
            var bchDto = new BedCountChangeDto { Id =
            reservation.AccomodationId, ChangeValue = changeValue };

            await _dataClient.ChangeBedCountOfAccomodation(bchDto);
        }
        catch (Exception ex)
        {
            throw ex;
        }
        return Ok("Reservation accepted!");
    }
    else return BadRequest(resultMessage);
}
```

Ukážka kódu 41. Metóda HostAcceptResrevation využíva DataClient pre odosielenie správ do služby obsluhy ubytovaní

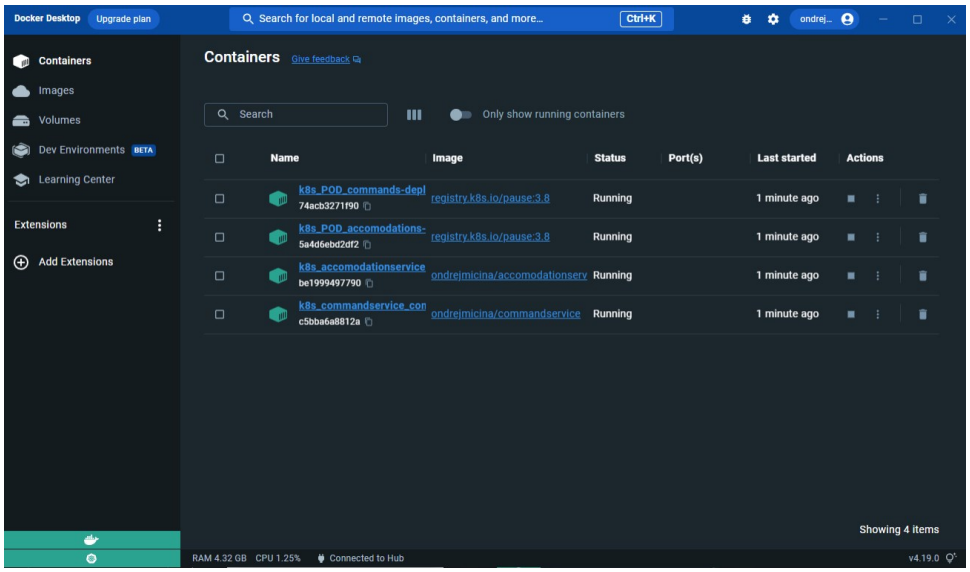


Obrázok 36. Priebeh dátovej komunikácie

6.6 Implementácia kontajnerizácie a orechestrácie služieb

Všetky služby jednotlivých mikroslužieb sa mi podarilo implementovať, spustiť a otestovať v lokálom prostredí. V nasledujúcich krokoch teda z týchto služieb vytvorím Docker kontajnery pomocou definovania potrebných nastavení.

Pre prácu s Dockerom je potrebná aplikácia Docker Desktop ktorá bude slúžiť ako virtuálne prostredie v ktorom budú bežať spustené kontajnery definovaných mikroslužieb.



Obrázok 37. Aplikácia Docker Desktop

6.6.1 Nastavenie inštrukcií pre budovanie image

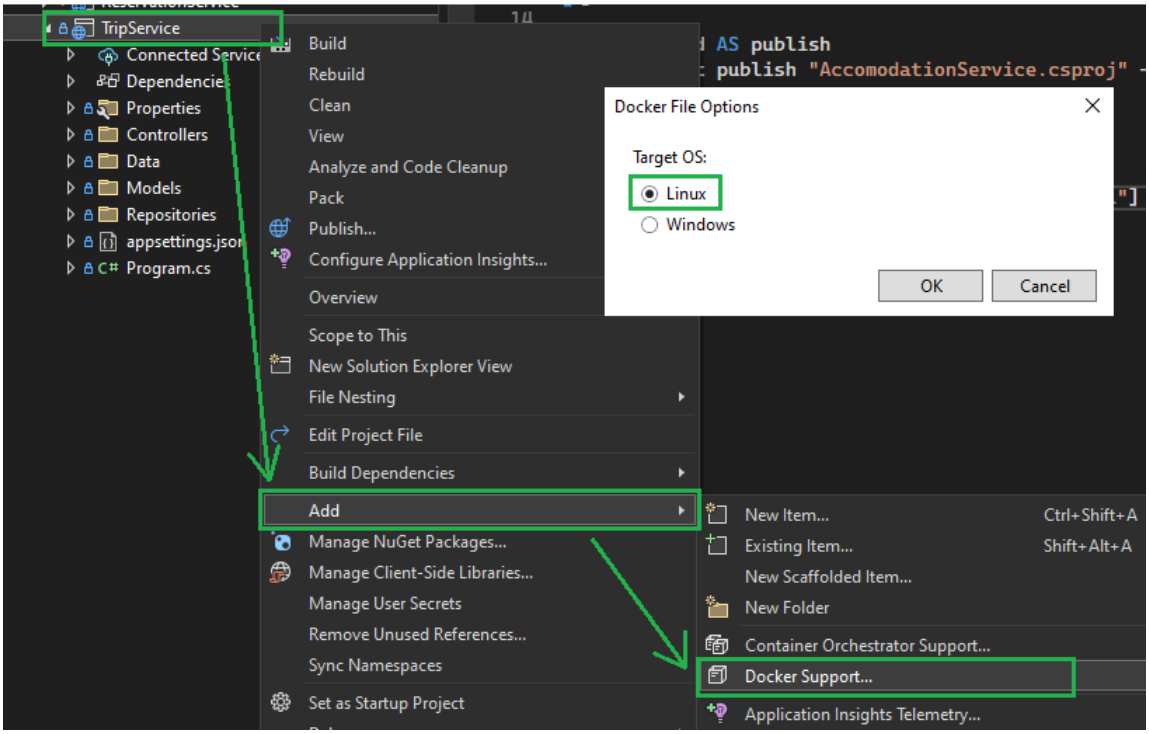
Docker nespúšťa aplikácie priamo z nahraných súborov. Na to aby bolo možné spustiť aplikáciu v prostredí Docker je potrebné z nej vytvoriť image. Image v Dockeri je statický balík, ktorý obsahuje všetky potrebné súbory a závislosti na spustenie konkrétnej aplikácie alebo prostredia ako kontajnera.

Na to, aby Docker mohol balík image vytvoriť, je potrebné definovať inštrukcie na zostavenie tohto balíka. Tieto inštrukcie sa zapíšu do súboru s presným názvom „Dockerfile“ ktorý je vhodné vytvoriť v priečinku danej mikroslužby. Inštrukcie je možné zapísať manuálne podľa Docker dokumentácie alebo automaticky pomocou VS.

Následne pomocou príkazu docker build je vybudovaný image kontajneru mikroslužby.

Keďže postup tvorby kontajneru je pre všetky služby rovnaký, vytvorenie Dockerfile a samotného kontajneru bude demonštrované s použitím služby určenej pre správu výletov.

Pre generovanie je potrebné kliknúť pravým na TripService - Add - Docker Support a zvoliť operačný systém Linux. VS následne vygeneruje potrebné inštrukcie pričom zahrnie aj všetky závislosť projektu. Inštrukcie v súbore Dockerfile sú určené pre prekladač ktorý bude zostavovať DockerImage image.



Obrázok 38. Generovanie Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["TripService/TripService.csproj", "TripService/"]
COPY
["AuthenticationServiceRegistrator/AuthenticationServiceRegistrator.csproj",
"AuthenticationServiceRegistrator/"]
RUN dotnet restore "TripService/TripService.csproj"
COPY . .
WORKDIR "/src/TripService"
RUN dotnet build "TripService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "TripService.csproj" -c Release -o /app/publish
/p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "TripService.dll"]
```

Ukážka kódu 42. Inštrukcie v súbore Dockerfile

Význam jednotlivých inštrukcií:

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS base

- Tento riadok určuje základný obraz (base-image) pre prvú fázu zostavenia obrazu. V tomto prípade sa používa obraz s názvom "dotnet/sdk:6.0" poskytovaný spoločnosťou Microsoft. Tento obraz obsahuje nástroje a závislosti potrebné na zostavenie aplikácie s Dotnet SDK verzie 6.0.

WORKDIR /app

- Týmto riadkom sa nastavuje pracovný priečinok v kontajneri, kde sa budú vykonávať následujúce príkazy.

COPY ["TripService/TripService.csproj", "TripService/"]

- Tento príkaz kopíruje súbor TripService/TripService.csproj z aktuálneho adresára hostiteľského systému do pracovného priečinka TripService/ v kontajneri.

COPY
["AuthenticationServiceRegistrator/AuthenticationServiceRegistrator.csproj",
"AuthenticationServiceRegistrator/"]

- Tento príkaz kopíruje závislosť AuthenticationServiceRegistrator.csproj z aktuálneho adresára hostiteľského systému do pracovného priečinka AuthenticationServiceRegistrator v kontajneri.

RUN dotnet restore

- Tento príkaz vykonáva príkaz "dotnet restore" v kontajneri, čo spôsobí obnovenie balíčkov a závislostí pre projekt na základe súboru ".csproj".

COPY . .

- Tento príkaz kopíruje všetky súbory a adresáre z aktuálneho adresára hostiteľského systému do pracovného priečinka v kontajneri.

RUN dotnet publish -c Release -o out

- Tento príkaz vykonáva príkaz "dotnet publish" s cieľom "Release" a výstupným adresárom "out". Týmto sa vytvorí skompilovaná a optimalizovaná verzia aplikácie, ktorá je pripravená na spustenie.

FROM mcr.microsoft.com/dotnet/aspnet:6.0

- Tento riadok určuje základný obraz pre druhú fázu zostavenia obrazu, ktorá je založená na runtime obrazu ASP.NET verzie 6.0 poskytovaného spoločnosťou Microsoft. Tento obraz obsahuje všetko potrebné na spustenie aplikácie ASP.NET.

WORKDIR /app

- Opäť sa nastavuje pracovný priečinok v kontajneri, tentoraz pre druhú fázu zostavenia obrazu.

COPY --from=build-env /app/out .

- Tento príkaz kopíruje obsah výstupného adresára z prvotnej fázy (označenej ako "build-env") do aktuálneho pracovného priečinka v kontajneri. Tým sa aplikácia, ktorá bola skompilovaná a optimalizovaná v predchádzajúcej fáze, dostane do tohto obrazu.

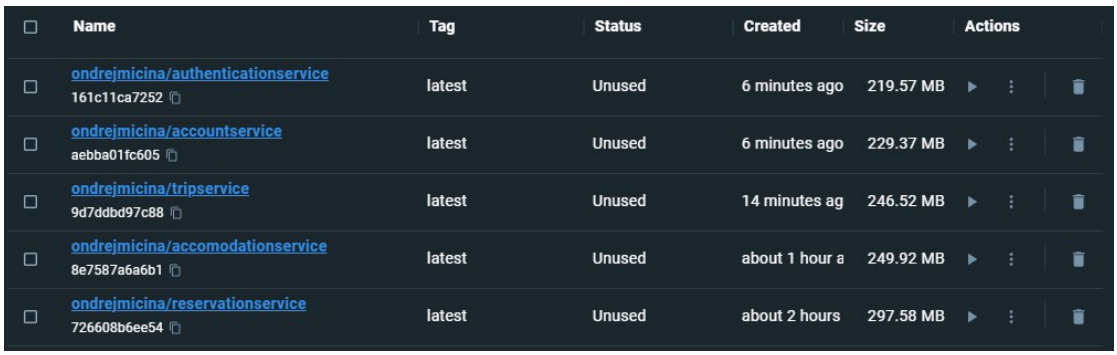
ENTRYPOINT ["dotnet", "TripService.dll"]

- Tento riadok určuje spúšťací príkaz pre kontajner. Pri spustení kontajneru sa vykoná príkaz "dotnet TripService.dll", čím sa spustí aplikácia.

6.6.2 Vybudovanie jednotlivých image

Keďže jednotlivé mikroslužby obsahujú závislosti, build jednotlivých kontajnerov je nutné vykonávať z rodičovského adresára aby prekladač mohol pridávať jednotlivé závisloti. Image sa vybuduje pomocou príkazu ***docker build -f .\TripService\Dockerfile -t ondrejmicina/tripservice .*** pričom parameter –f označuje cestu Dockerfile súboru, parameter –t vyjadruje zadanie vlastného názvu image – „ondrejmicina/tripservice“ a bodka znamená kontext budovania (celý priečinok). Docker následne stiahne požadované image súbory uvedené v súbore Dockerfile a poskladá výsledný image so zadaným názvom.

Build služieb možno zadať ako viacero príkazov naraz. Pre všetky služby platí rovnaký postup, rozdiel bude v jednotlivých .csproj súboroch a v .dll súbore danej služby. Vytvorené image jednotlivých mikroslužieb možno skontrolovať aj v Docker Desktop aplikácii.



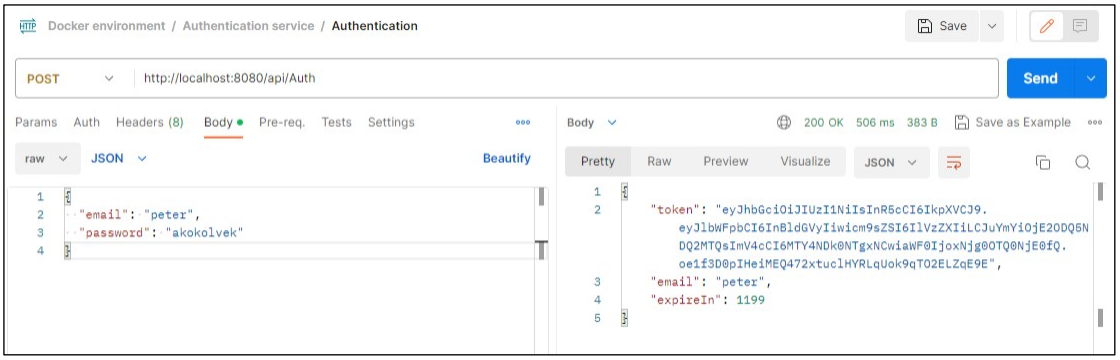
Obrázok 39. Vytvorené image jednotlivých mikroslužieb

6.6.3 Vytvorenie kontajnerov služieb z image

Pre prvé vytvorenie a spustenie kontajneru je nutné zavolať príkaz ***docker run -p 8080:80 -d ondrejmicina/authenticationservice*** ktorý vytvorí kontajner z image a spustí ho na porte 8080. Následne pomocou príkazu ***docker ps*** možno zobraziť bežiace kontajnery a ich ID. Toto ID je možné použiť v príkazoch ***docker start [id]*** a ***docker stop [id]*** pre spúštanie a vypínanie kontajnera.

Keďže v ďalšom kroku bude použitá platforma Kubernetes ktorá jednotlivé image získava z internetu, je potrebné aby bol image služby pridaný do online registra Docker. Takzvaný „push“ sa vykonáva pomocou príkazu ***docker push ondrejmicina/authenticationservice*** ktorý odošle vytvorený image do registra.

Keďže služby obsahujú nastavenia v appsetings.json súboroch je nutné pred samotným spustením definovať nastavenia pre produkčné prostredie a to tak že sa vytvorí súbor s názvom appsettings.Production.json do ktorého sa tieto nastavenia umiestnia.



Obrázok 40. Spustený kontajner so službou Authentication teraz beží na porte 8080

6.7 Orchestrácia služieb

Cieľom orchestrácie je dosiahnuť väčšiu flexibilitu, škálovateľnosť a spoľahlivosť aplikácie s architektúrou mikroslužieb a automatizované riadenie služieb.

Na orchestráciu mikroslužieb sa používa platforma Kubernetes. Pre demonštráciu bude použitá služba AccountService.

V prvom kroku pomocou sa vytvorí konfiguračný súbor danej služby v ktorom budú definované nastavenia deploymentu. Súbor bude vytvorený v samostatnom priečinku \K8S v rodičovskom adresári. Z tohto priečinku budú volané jednotlivé príkazy pre prácu Kubernetes.

Následne sa vytvorí súbor accounts-depl.yaml ktorý bude obsahovať nastavenia deploymentu. Význam jednotlivých nastavení:

- **apiVersion: apps/v1** určuje verziu API pre Deployment objekt v Kubernetes.
- **kind: Deployment** definuje typ objektu, v tomto prípade Deployment.
- **metadata** obsahuje metadáta objektu, vrátane názvu (name). V tomto prípade je názov nastavený na accounts –depl.
- **spec** špecifikuje konfiguráciu nasadenia.
- **replicas: 3** určuje počet replík (inštancií) aplikácie, ktoré majú byť nasadené. Vyšší počet replík sa používa v prípade veľkého dopytu po službe.

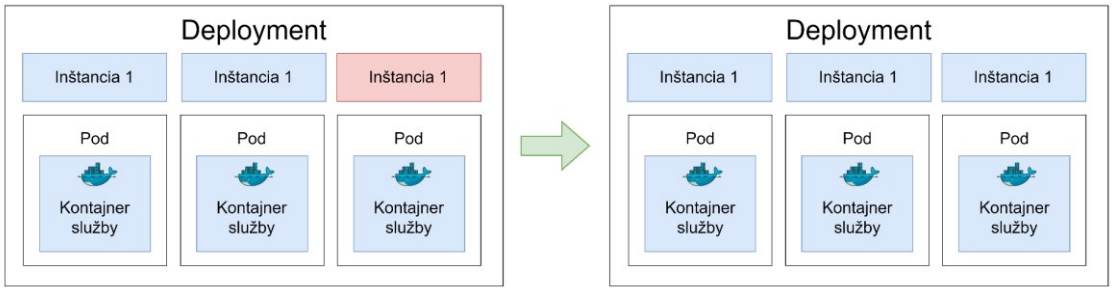
- **selector** definuje, ako sa majú vybrať ***pod*** objekty, ku ktorým sa vzťahuje Deployment.
- **matchLabels** určuje, aké značky musia mať ***pod*** objekty, aby boli vybrané. V tomto prípade sa vyberajú ***pod*** objekty s hodnotou **app: accountservice**.
- **template** definuje šablónu pre vytváranie ***pod*** objektov.
- **metadata** obsahuje metadáta ***pod*** objektu, vrátane značiek. V tomto prípade je značka nastavená na **app: accountservice**.
- **spec** špecifikuje konfiguráciu ***pod*** objektu, ktorý obsahuje kontajner s aplikáciou.
- **containers** určuje zoznam kontajnerov v ***pod*** objekte.
- **name: accountservice** určuje názov kontajnera.
- **image: ondrejmicina/accountservice:latest** určuje image kontajnera, ktorý sa má použiť z online registra Docker. V tomto prípade sa používa image s názvom ondrejmicina/accountservice a najnovšou verziou označenou ako latest.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: accounts-depl
spec:
  replicas: 3
  selector:
    matchLabels:
      app: accountservice

  template:
    metadata:
      labels:
        app: accountservice

    spec:
      containers:
      - name: accountservice
        image: ondrejmicina/accountservice:latest
```

Ukážka kódu 43 Nastavenie deploymentu

Po definovaní nastavení je možné vytvoriť takzvaný **pod** objekt pomocou aplikovania deploymentu služby. V rámci podu budú spustené jeden alebo viac kontajnerov danej služby. Deployment sa vykoná pomocou príkazu ***kubectl apply -f .\accounts-depl.yaml***. Vytvorené deploymenty je možné zobraziť príkazom ***kubectl get deployments*** pomocou ktorého je možné zistiť ich stav. Cieľom podov bude zabezpečiť neustály chod tejto služby

alebo počet replík danej služby. V prípade že služba zlyhá, pod okamžite spustí novú čím udržiava chod aplikácie. Pre prehľad podov možno použiť príkaz ***kubectl get pods***.



Obrázok 41. Pod zabezpečí nové spustenie služby v prípade jej zlyhania

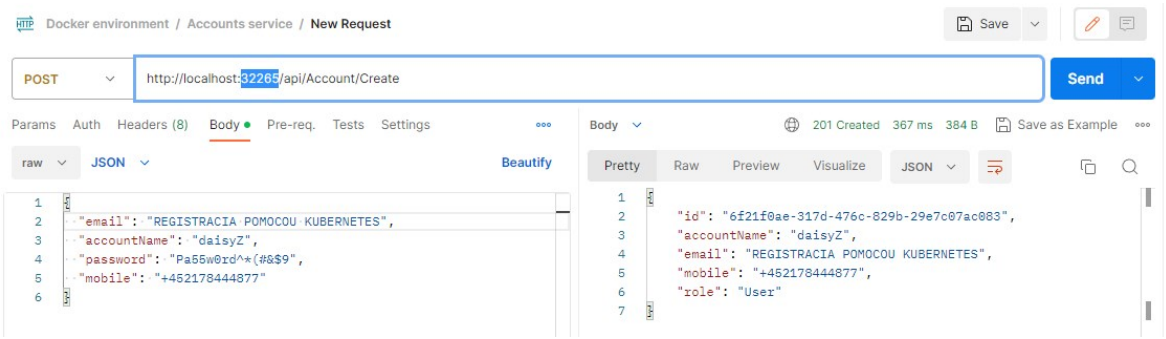
V tomto momente beží služba avšak nemožno k nej pristúpiť. Pre demonštračné účely bude vytvorený node port pomocou ktorého bude vykonávaná komunikácia s vonkajším prostredím.

V priečinku K8S sa vytvorí súbor s názvom accounts-np-srv.yaml ktorý bude obsahovať nastavenia služby node port.

```
apiVersion: v1
kind: Service
metadata:
  name: account-np-service-srv
spec:
  type: NodePort
  selector:
    app: accountservice
  ports:
    - name: accountservice
      protocol: TCP
      port: 80
      targetPort: 80
```

Ukážka kódu 44. Nastavenia služby node portu

Následne sa vykoná deployment node portu pomocou príkazu ***kubectl apply -f .\accounts-np-srv.yaml*** . Pomocou príkazu ***kubectl get services*** sa zobrazia aktuálne bežiace služby, medzi nimi aj node port. Každá bežiaca služba obsahuje informácie vrátanie IP adresy a portu. V tomto prípade má služba node port priradený port 32265 ktorý bude použitý pre požiadavku v aplikácii Postman.



Obrázok 42. Spracovanie požiadavky pomocou kontajnera spravovaného platformou Kubernetes

7 ZHODNOTENIE A MOŽNOSTI DAĽŠIEHO ROZVOJA

V predchádzajúcej kapitole bol predstavený návrh demonštračnej aplikácie a príslušný postup implementácie vybraných častí aplikácie. Cieľom tohto postupu bolo poskytnúť základné znalosti a zručnosti z oblasti návrhu a implementácie mikroslužieb.

V prípade záujmu o rozšírenie aplikácie, alebo o jej uvedenie do reálneho komerčného prostredia je nutné aby boli dôkladne zvážené všetky kroky návrhu a implementácie. Táto kapitola rozoberie niektoré z predstavených riešení a popíše ďalšie pohľady a možnosti rozvoja danej problematiky.

7.1 Mikroslužby alebo monolit

Pri návrhu aplikácie sa berie v úvahu jej veľkosť a členitosť. Oplatí sa začínať s architektúrou mikroslužieb alebo začať radšej monolitom a neskôr ho prestavať na mikroslužby – to je dôležitá otázka.

Implementácia aplikácie na výmenu ubytovaní a výletov bola od začiatku vedená v architektúre mikroslužbe. Bolo to však správne?

Zo začiatku sa môže zdať že aplikácia je veľmi malá a architektúra mikroslužieb je pre ňu zbytočne náročnou aj z pohľadu nevyužitia potenciálu tejto architektúry. Pre jednoduché funkcie by stačila aj monolitická architektúra ktorá by bola vyvíjaná ako jeden celok, nemusela by sa riešiť komunikácia a podobne.

Ak sa však zhodnotí, že táto aplikácia môže mať v budúcnosti náročnejšie a komplexnejšie funkcie (čo v tomto prípade má – poskytuje funkciu ubytovania a výletov zároveň) alebo bude vyžadovať škálovanie, odlišné úložiská dát alebo väčšiu dostupnosť, voľba mikroslužieb hneď od začiatku je vhodným riešením.

Na druhej strane, ak sa neočakáva nárast komplexnosti aplikácie, ak bola aplikácia zameraná na jeden typ služby (napr. ponuka výletov) a nie je úplne jasné či sa uchytí na trhu, je vhodné zvoliť na začiatku monolitickú architektúru. Vďaka monolitickej architektúre je možné vyvinúť aplikáciu oveľa rýchlejšie, keďže odpadá nutnosť implementovať náročné funkcie ako napr. komunikáciu medzi službami pri mikroslužbách.

7.2 Nové požiadavky aplikácie

V prípade rozšírenia aplikácie pre zdieľanie výletov a ubytovania možno uvažovať nad pridaním nových požiadaviek. Medzi nové funkčné požiadavky by sa mohli zaradiť nasledovné:

- **Notifikácie**: Aplikácia by mala posielať notifikácie používateľom o nových ponukách, odpovediach na ich ponuky a iných dôležitých udalostiach.
- **Správa používateľského profilu**: Používatelia by mali mať možnosť upraviť a spravovať svoje osobné údaje, vrátane profilovej fotky, kontaktných informácií a preferencií.
- **Vyhľadávanie a filtrovanie**: Používatelia by mali mať možnosť vyhľadávať a filtrovať ponuky ubytovania a výletov na základe určitých kritérií, ako napríklad miesto, dátum, typ ubytovania a ďalšie parametre.
- **Integrácia mapy**: Aplikácia by mala integrovať mapové služby, aby používateľom umožnila vizualizovať umiestnenie ponúk ubytovania a výletov a poskytla navigáciu k cieľovým bodom.
- **Správa platby**: Ak by aplikácia ponúkala možnosť platby za niektoré služby (napr. diaľničné známky, vstupenky), mala by zahrňovať bezpečné a spoľahlivé metódy spracovania platby.
- **Ukladanie obľúbených položiek**: Používatelia by mali mať možnosť označovať ponuky ubytovania a výletov ako obľúbené a ukladať ich pre ľahší prístup a neskoršie prehliadanie.
- **Systémové upozornenia**: Používatelia by mali dostávať upozornenia o dôležitých udalostiach, ako napríklad potvrdenie rezervácie alebo blížiace sa termíny výletov.
- **Sledovanie histórie**: Používatelia by mali mať prístup k histórii svojich aktivít, vrátane predchádzajúcich ponúk, výmen ubytovania a výletov

Okrem funkčných požiadaviek s rastúcou aplikáciou bude potrebné navrhnúť aj nefunkčné požiadavky. Môžu to byť napríklad:

- **Výkon**: Aplikácia by mala byť rýchla a spoľahlivá aj pri veľkom množstve používateľov a ponúk. Odpovede a aktualizácie by mali byť vykonávané v reálnom čase.

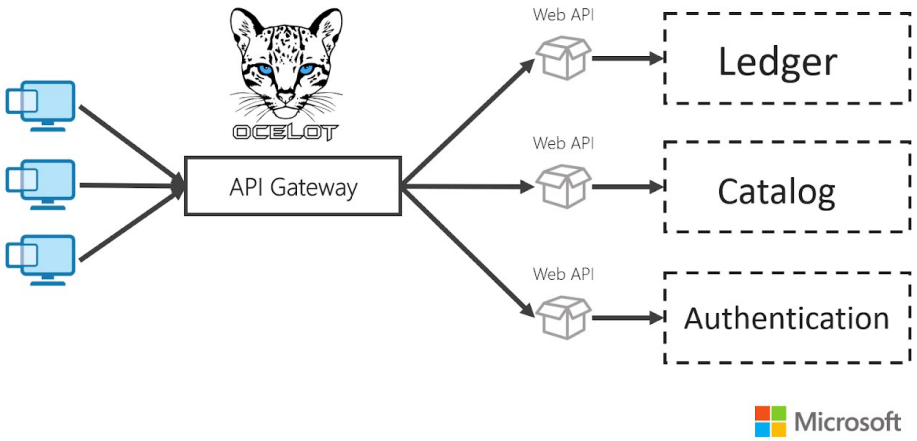
- **Podpora rôznych platform**: Aplikácia by mala byť poskytovať prispôsobené odpovede API pre jednotlivé platformy, ako napríklad mobilné zariadenia (Android, iOS) a webové prehliadače.
- **Dostupnosť dát**: Aplikácia by mala mať spoľahlivý prístup k dátam o ponukách ubytovania a výletov, ako aj k používateľským údajom, aby sa zabezpečila konzistentnosť a aktualizácia informácií.
- **Zálohovanie a obnovenie**: Aplikácia by mala mať mechanizmus zálohovania a obnovenia dát, aby sa minimalizovala straty údajov v prípade výpadku alebo havárie systému.
- **Multimediálna podpora**: Aplikácia by mala byť schopná spracovať a zobrazovať multimediálne obsahy, ako napríklad fotografie a videá, s adekvátnou kvalitou a rýchlosťou na podporu prehliadania ponúk a prípadného zdieľania obsahu medzi používateľmi.
- **Súlad s reguláciami**: Aplikácia by mala byť v súlade s platnými zákonnými a regulatívnymi požiadavkami týkajúcimi sa ochrany osobných údajov, online transakcií a zdieľania obsahu.

7.3 Použitie API Gateway

Aj keď návrh aplikácie pre zdieľanie výletov a ubytovaní nezahŕňa implementáciu API Gateway určite stojí za predstavenie. API Gateway poskytuje rozhranie (API) pre komunikáciu s vonkajším prostredím, pričom vytvára vrstvu abstrakcie a centralizácie pre mikroslužby. Tento prístup prináša niekoľko výhod.

API Gateway poskytuje jednotné rozhranie, čo uľahčuje klientom interakciu so službami pričom skrýva internú komplexnosť systému. Ďalšou výhodou je, že API Gateway umožňuje implementáciu škálovateľnosti a výkonnosti. Vďaka tejto vlastnosti je možné riadiť tok požiadaviek, implementovať vyrovnávaciu pamäť alebo load balancer, čo prispieva k optimalizácii výkonu a zvýšeniu odolnosti systému voči prípadnej vysokej záťaži.

K návrhu implementácie by bolo teda možné pridať API Gateway v rámci aplikácie. Jednou z nich je napríklad Ocelot Gateway, ktorý je open-source nástroj pre .NET ekosystém. Ocelot poskytuje množstvo funkcií pre správu a konfiguráciu API Gateway, ako sú routovanie, autentifikácia, autorizácia a zabezpečenie.



Obrázok 43. API Gateway Ocelot[32]

Ďalšou možnosťou je použitie NGINX, populárneho webového servera a proxy servera. NGINX ponúka robustné možnosti konfigurácie a výkonu a môže byť nasadený ako API Gateway. Keďže v návrhu aplikácie pre zdieľanie výletov a ubytovaní je využívaná platforma Kubernetes, môže byť NGINX dodatočne nakonfigurovaná konfigurovaný a spravovaný práve pomocou Kubernetes ingress controlleru.

7.4 Použitie asynchrónnej komunikácie

V návrhu aplikácie pre zdieľanie výletov a ubytovaní je pre komunikáciu využitá synchrónna komunikácia pomocou HTTP protokolu. V architektúre mikroslužieb sa používa taktiež asynchrónna komunikácia.

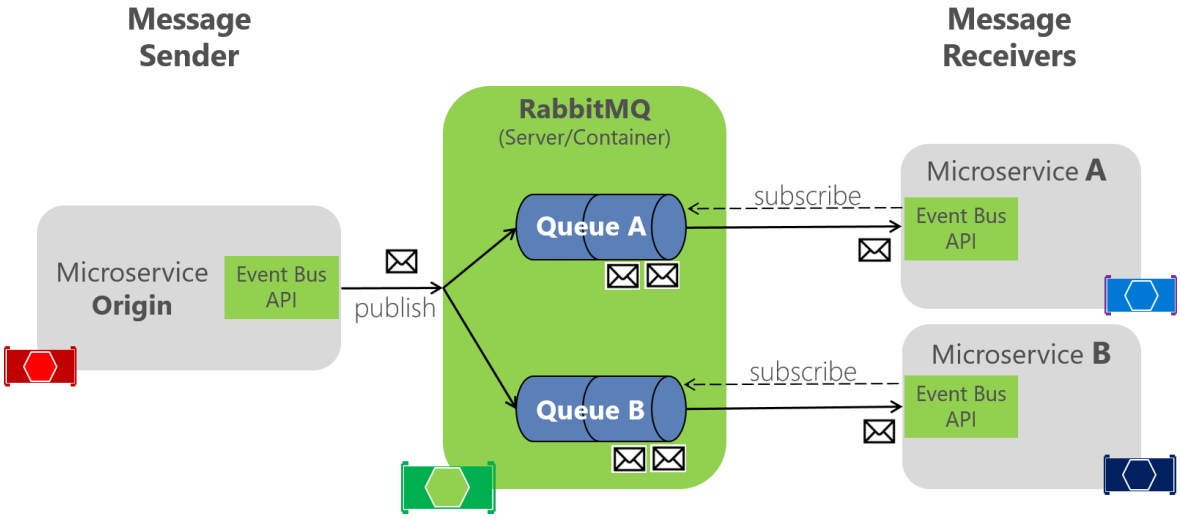
Asynchrónna komunikácia v mikroslužbách je spôsob komunikácie, v ktorom sa výmena správ medzi službami neuskutočňuje prostredníctvom tradičného cyklu „request-response“ (žiadosť-odpoveď), ale na základe vzoru „publish-subscribe“ (publikovanie-odber). Tento spôsob umožňuje komunikáciu medzi službami bez toho, aby jedna služba čakala na okamžitú odpoveď od druhej.

Pri asynchrónnej komunikácii sa správy publikujú (publish) na určitý kanál alebo tematickú frontu - mesasge bus alebo mesasge broker. Služby, ktoré sú zainteresované, si tieto správy odberajú (subscribe). Publikovanie a odoberanie správ sa môže diať nezávisle od seba a správy môžu byť spracované asynchrónne, čo umožňuje väčšiu paralelizáciu a odolnosť voči výpadkom.

Jedným z nástrojov, ktorý sa často používa pre asynchrónnu komunikáciu v mikroslužbách, je RabbitMQ. RabbitMQ je open-source správny systém (message broker), ktorý umožňuje publikovanie a odoberanie správ medzi službami. Je schopný spracovať veľké množstvo správ a zabezpečuje rôzne mechanizmy doručenia a spracovania správ, ako sú fronty, výmeny a routovanie.

RabbitMQ môže byť použitý aj v kontajnerovom prostredí pomocou Kubernetes čo znamená že by mohol byť použitý aj v návrhu aplikácie pre zdieľanie výletov a ubytovaní. RabbitMQ môže byť nasadený ako samostatná služba v Kubernetes clusteri a môže byť nakonfigurovaný na komunikáciu medzi mikroslužbami pomocou publish-subscribe modelu.

Vďaka asynchrónnej komunikácii v mikroslužbách aplikácie nemusia o sebe vedieť, čo znižuje ich vzájomnú závislosť. Každá služba môže zverejniť správy, ktoré sú relevantné pre iné služby, a tieto správy môžu byť odoberané a spracovávané nezávisle. To umožňuje väčšiu flexibilitu, škálovateľnosť a jednoduchšie rozširovanie systému.



Obrázok 44. Implementácia RabbitMQ [33]

ZÁVER

Cieľom tejto práce je prispenie k lepšiemu porozumeniu softvérovej architektúry založenej na mikroslužbách a poskytnutie konkrétneho príkladu implementácie na tvorbe aplikácii určenej pre zdieľanie výletov a ubytovaní.

V teoretickej časti boli popísané rôzne architektonické prístupy, ako je monolitická architektúra, objektovo orientovaná architektúra a architektúra orientovaná na služby. Bol preskúmaný koncept mikroslužieb a výhody, ako je modularita, nezávislosť, odolnosť a škálovateľnosť. Taktiež boli popísané rozdiely medzi monolitickými riešeniami a riešeniami založenými na mikroslužbách. Porovnané boli napríklad škálovateľnosť, obtiažnosť vývoja a nasadzovania, technologická flexibilita, izolácia porúch a bezpečnosť. Na základe týchto porovnaní je možné si vybrať vhodnú architektúru pre rôzne riešenia.

V praktickej časti bola navrhnutá a implementovaná aplikácia pre zdieľanie výletov a ubytovaní. Boli vytvorené detailné postupy implementácie mikroslužby pre správu používateľských účtov, ubytovaní, výletov a rezervácií. V jednotlivých riešeniach boli popísané použité vývojové metódy ako napríklad repository pattern, DTO objekty, dependency injection a podobne. Taktiež boli popísané použité nástroje ako VisualStudio, Postman, Docker či Kubernetes. Rovnako sa venovala pozornosť autentifikácii s využitím Jwt tokenu, komunikácii medzi službami prostredníctvom http komunikácie a kontajnerizácie a orchestrácii služieb pomocou platforiem Docker a Kubernetes.

Záver práce naznačuje, že mikroslužby ponúkajú množstvo výhod voči tradičným monolitickým riešeniam. Ich flexibilita a škálovateľnosť prispievajú k lepšiemu výkonu aplikácie a používateľskému zážitku. Zhodnotenie možností o ďalšom rozvoji aplikácie poukazuje na možnosti, ako ďalej vylepšiť implementáciu, ako napríklad využitie API Gateway pre centralizovanú správu komunikácie alebo implementáciu asynchrónnej komunikácie pre zvýšenie výkonu a škálovateľnosti.

Celkovo je táto práca prínosná pre lepšie porozumenie softvérovej architektúry založenej na mikroslužbách Implementácia aplikácie pre zdieľanie výletov a ubytovania s využitím mikroslužieb je výsledkom tejto práce a predstavuje funkčnú názornú ukážku.

ZOZNAM POUŽITEJ LITERATÚRY

[1] Chapter 1: What is Software Architecture? [online]. [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ee658098(v=pandp.10)

[2] RICHARDS, Mark a Neal FORD, 2020. Fundamentals of software architecture: an engineering approach. Beijing: O‘Reilly. ISBN 978-149-2043-454.

[3] FOWLER, Martin, 2003. Patterns of enterprise application architecture. Boston: Addison-Wesley. Addison-Wesley signature series. ISBN 03-211-2742-0.

[4] FOWLER, Martin, 13 May 2015n. l. MicroservicePremium [online]. [cit. 2023-05-26]. Dostupné z: https://martinfowler.com/bliki/MicroservicePremium.html

[5] RICHARDSON, Chris, MAY 25, 2014. Microservices: Decomposing Applications for Deployability and Scalability [online]. [cit. 2023-05-26]. Dostupné z: https://www.infoq.com/articles/microservices-intro/

[6] KALSKE, Miika, 2017. Transforming monolithic architecture towards microservice ar-chitecture. Helsinki. Dostupné také z: https://core.ac.uk/reader/157587910. M.Sc. Thesis. UNIVERSITY OF HELSINKI Department of Computer Science.

[7] RICHARDSON, Chris. Monolithic Architecture [online]. [cit. 2023-05-26]. Dostupné z: https://microservices.io/patterns/monolithic.html

[8] RAJ, Pethuru, Anupama RAMAN a Harihara SUBRAMANIAN, 2017. Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture. 2017. ISBN 978-1787287495.

[9] KOTUĽ, Matúš, 2022. Použití mikroslužeb pro vývoj aplikace. Zlín. Diplomová práce. Univerzita Tomáše Bati ve Zlíně. Vedoucí práce Ing. Pavel Vařacha, Ph.D.

[10] GILLIS, Alexander S. a Sarah LEWIS. Object-oriented programming (OOP) [online]. [cit. 2023-05-26]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

[11] NEWMAN, Sam, 2015. Building microservices. Sebastopol: O'Reilly. ISBN 978-149-1950-357.

[12] Microservices: yesterday, today, and tomorrow [online]. [cit. 2023-05-26]. Dostupné z: https://www.researchgate.net/publication/315664446_Microservices_yesterday_today_and_tomorrow

[13] MOHAPATRA, Biswa Pujarini, Baishakhi BANERJEE a Gaurav ARORAA, 2019. Microservices by examples using .NET Core: A book with lot of practical and architectural styles for Microservices using .NET Core (First). ISBN 9387284581.

[14] LEWIS, James a Martin FOWLER. Microservices [online]. [cit. 2023-05-26]. Dostupné z: https://martinfowler.com/articles/microservices.html

[15] The Publish-Subscribe Pattern on Rails: An Implementation Tutorial [online]. [cit. 2023-05-26]. Dostupné z: https://www.toptal.com/ruby-on-rails/the-publish-subscribe-pattern-on-rails

[16] GOETSCH, Kelly, 2017. Microservices for Modern Commerce. ISBN 9781491970874.

[17] NEWMAN, Sam, 2021. Building microservices: designing fine-grained systems. Second edition. Beijing: O'Reilly. ISBN 978-149-2034-025.

[18] Microservices Orchestration vs Choreography | (Technology) [online]. [cit. 2023-05-26]. Dostupné z: https://medium.com/geekculture/microservices-orchestration-vs-choreography-technology-5dbe612cf7e9

[19] SCHABOWSKY, Jonathan. The Benefits of Microservices Choreography vs Orchestration [online]. [cit. 2023-05-26]. Dostupné z: https://solace.com/blog/microservices-choreography-vs-orchestration/

[20] Choreography pattern [online]. [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography

[21] KIZILPINAR, Dilfuruz. Data Consistency in Microservices Architecture [online]. [cit. 2023-05-26]. Dostupné z: https://dilfuruz.medium.com/data-consistency-in-microservices-architecture-5c67e0f65256

[22] Microservices Choreography Best Practices and Tools [online]. [cit. 2023-05-26]. Dostupné z: https://levelup.gitconnected.com/microservices-choreography-best-practices-and-tools-8e46f12bf8f1

[23] Microservice Architecture: Eventual Consistency [online]. [cit. 2023-05-26]. Dostupné z: https://dev.to/evanhameed99/microservice-architecture-eventual-consistency-3ckp

[24] Scaling Monolithic Applications [online]. [cit. 2023-05-26]. Dostupné z: https://medium.com/swlh/scaling-monolithic-applications-3c69193f942a

[25] PETYCHAKIS, Michael. Scalability Design with MicroServices [online]. [cit. 2023-05-26]. Dostupné z: https://apilama.com/2018/07/18/scalability-design-with-microservices/

[26] AWATI, Rahul a Ivy WIGMORE. Monolithic architecture [online]. [cit. 2023-05-26]. Dostupné z: https://www.techtarget.com/whatis/definition/monolithic-architecture

[27] HARRIS, CHANDLER. Microservices vs. monolithic architecture [online]. [cit. 2023-05-26]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[28] NADAREISHVILI, Irakli et al., 2016. Microservice Architecture: Aligning Principles, Practices, and Culture. ISBN 978-1491956250.

[29] Microservices vs Monolithic Architecture for DevOps: 8 Differences Explained [online]. [cit. 2023-05-26]. Dostupné z: https://www.spiceworks.com/tech/devops/articles/microservices-vs-monolithic-architecture/

[30] Monoliths vs. Microservices: Differences, Pros & Cons, and Choosing the Right Architecture [online]. [cit. 2023-05-26]. Dostupné z: https://www.cortex.io/post/monoliths-vs-microservices-whats-the-difference

[31] ELLIOTT, Roxana. Monolithic vs microservice architecture: Which is best? [online]. [cit. 2023-05-26]. Dostupné z: https://www.digitalocean.com/blog/monolithic-vs-microservice-architecture#how-to-choose-between-a-monolithic-and-microservices-application

[32] Ocelot [online]. [cit. 2023-05-26]. Dostupné z: https://github.com/PasinduUmayanga/Ocelot-Gateway-Sample

[33] Implementing an event bus with RabbitMQ for the development or test environment [online]. [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/rabbitmq-event-bus-development-test-environment

ZOZNAM POUŽITÝCH SKRATIEK A SYMBOLOV

CRUD	Create Read Update Delete
REST	Representational state transfer
RPC	Remote procedure call - Vzdialené volanie procedúr
ACID	Atomicity, Consistency, Isolation, and Durability
SQL	Structured Query Language - relačná databáza
NoSQL	non SQL – nerelačná databáza
OOA	Objektovo orientovaná architektúra
OOP	Objektovo orientované programovanie
OS	Operačný systém
MVC	Model View Controller
SOA	Service oriented architecture / Architektúra orientovaná na služby
SOAP	Simple Object Access Protocol
MVP	Minimum viable product
tzv.	takzvaný/takzvane
DevOps	Development a Operations
ORM	Object-Relational Mapping
DTO	DataTransferObject

ZOZNAM OBRÁZKOV

ZOZNAM UKÁŽOK KÓDU

ZOZNAM PRÍLOH

Príloha P I: CD-ROM

PRÍLOHA P I: CD-ROM

Priložené CD obsahuje:

- Zdrojové kódy aplikácie uložené v archive .zip
- Diplomovú prácu vo formáte .pdf