Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Porovnání mikroservisové a monolitické architektury pro menší webové aplikace

Bc. Martin Kustl

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Martin Kustl

Systémové inženýrství a informatika
Informatika

Název práce

Porovnání mikroservisové a monolitické architektury pro menší webové aplikace

Název anglicky

Comparison of microservice architecture and monolithic architecture for smaller web applications

---

Cíle práce

Práce se zabývá problematikou mikroservisové a monolitické softwarové architektury webových aplikací. Hlavním cílem této práce je porovnat mikroservisovou a monolitickou architekturu pro menší aplikace. Dílčí cíle jsou:
- Charakterizovat technologie a možnosti, které jsou v současné době při tvorbě webových aplikací k dispozici.
- Vytvořit experimentální webovou aplikaci.
- Vybrat vhodná kritéria pro porovnání těchto dvou architektur.
- Provést experimentální měření aplikace, a na základě jeho výsledků architektury porovnat.

Metodika

Metodika zpracování teoretické části spočívá ve studiu a analýze odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska práce.

V praktické části bude navržena a implementována webová aplikace. Tato aplikace bude implementována ve dvou verzích pro každou architekturu zvlášť. Následně budou tyto dvě verze aplikace sloužit k porovnání obou softwarových architektur dle předem stanovených kritérií.

Na základě teoretických východisek a výsledků praktické části budou formulovány závěry práce.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

webová aplikace, JavaScript, mikroservisová architektura, monolitická architektura, TypeScript, NodeJS

Doporučené zdroje informací

FLANAGAN, David. JavaScript – The Definitive Guide. 7th Edition. Newton (Massachusetts): O'Reilly Media, 2020. ISBN 978-1491952023

NEWMAN, Sam. Building Microservices: Designing Fine-Grained Systems. Newton (Massachusetts): O'Reilly Media, 2015. ISBN 978-1491950357

NEWMAN, Sam. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. Newton (Massachusetts): O'Reilly Media, 2019. ISBN 978-1492047841

RODGER, Richard. The Tao of Microservices. New York (New York): Manning Publications, 2017. ISBN 978-1617293146

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Jan Masner, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 27. 7. 2021

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 23. 03. 2022

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Porovnání mikroservisové a monolitické architektury pro menší webové aplikace" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 25.3.2022 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce, panu Ing. Janu Masnerovi, Ph.D., za odborné vedení, ochotu, věcné připomínky a cenné rady. Také bych chtěl poděkovat katedře informatiky za několikaletou spolupráci, díky které jsem mohl pracovat na řadě projektů, včetně vývoje webové aplikace popsané v této práci a získat tak velké množství praktických zkušeností s technologiemi, které se dnes v praxi zcela běžně využívají. Získané znalosti a zkušenosti mi mimo jiné umožnily zabývat se takto náročným tématem, na které bych si jinak netroufl.

Porovnání mikroservisové a monolitické architektury pro menší webové aplikace

Abstrakt

Diplomová práce se zabývá problematikou porovnání architektur webových aplikací. Hlavním záměrem této práce je co nejlépe a nejobsáhleji porovnat mikroservisovou a monolitickou architekturu pro menší webové aplikace. Pro splnění tohoto cíle bylo provedeno několik dílčích kroků. V teoretických východiscích jsou popsány technologie a možnosti, které jsou v současnosti dostupné pro vývoj webových aplikací. Některým technologiím je věnováno více prostoru, a to zejména se záměrem lepšího pochopení vlastní práce a jejích výsledků.

V rámci vlastní práce je popsána tvorba experimentální aplikace pro obě architektury zvlášť, načež jsou stanovena kritéria pro porovnání architektur. Poté je popsán průběh měření vytvořené experimentální webové aplikace. Zbylá část práce se zabývá porovnáním architektur na základě výsledků z měření vytvořené webové aplikace a poznatků získaných během psaní této práce.

Klíčová slova: webová aplikace, JavaScript, mikroservisová architektura, monolitická architektura, TypeScript, Node.js

Comparison of microservice architecture and monolithic architecture for smaller web applications

Abstract

The diploma thesis is focused on a comparison of web application architectures. The main goal is to comprehensively compare microservices and monolithic architecture for smaller web applications. Multiple sub-steps have been performed to accomplish the main goal. Technologies and options that are currently available for web applications development are described in the theoretical part of this thesis. Some of these technologies are described in more detail for a better understanding of the practical part and its results.

Development of the experimental web application is described individually for both architectures in practical part of this thesis and then criteria for comparison of both architectures are selected. Afterwards the process of measurement of the created experimental web application is described. The rest of this thesis is focused on the comparison of both architectures based on the results of the measurements of the experimental web application and findings acquired during creation of this thesis.

Keywords: web application, JavaScript, microservices architecture, monolithic architecture, TypeScript, Node.js

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Internet vznikl v šedesátých letech minulého století, přičemž původně se jednalo o armádní projekt s názvem ARPANET (1). Od té doby se mnohé změnilo a v současnosti představuje internet nedílnou součást našeho každodenního života. Bez internetu si zkrátka nelze představit dnešní život, protože ho využíváme k zábavě, komunikaci, práci, či ke studiu. Samotný internet však představuje pouze síť propojující obrovské množství zařízení celého světa.

To, co dělá z internetu v současnosti užitečnou technologii, jsou zejména webové stránky a webové aplikace, které lze díky němu používat. První webové stránky vznikaly v 90 letech minulého století. Jednalo se o statické textové dokumenty s nulovou interakcí vytvořené v jazyce HTML (2). Od dob prvních webů se však mnohé změnilo a dnes je možnost interakce s webovou stránku či aplikací považována za samozřejmost. V současnosti lidé používají webové aplikace na denním pořádku. Také svět byznysu hojně takové aplikace využívá, protože řada firem opustila od nativních desktopových či mobilních aplikací, a přetvořila tyto aplikace na webové. (3)

Dnešní webové aplikace musí mít přívětivé a intuitivní uživatelské rozhraní, ale také musí být například bezpečné, spolehlivé a rychlé. Tyto požadavky stojí za značným zvýšením náročnosti vývoje takových aplikací bez ohledu na jejich velikost. Jednou z otázek, kterou při vývoji tak musí v současnosti vývojářské týmy řešit je jakou architekturu pro vyvíjenou aplikaci zvolí. Jedná se o velice důležité rozhodnutí, které má přímý vliv na náročnost a dobu vývoje, ale také na následnou bezpečnost, možnosti škálování, či spolehlivost.

V současnosti je volba zejména mezi dvěma architekturami, a to monolitickou architekturou, ve které je aplikace napsána jako jeden celek a mikroservisovou architekturou, ve které je aplikace rozdělena na vícero menších nezávislých logických celků.

Tato práce se zaměřuje na problematiku porovnání těchto dvou architektur, a to zejména se zaměřením na menší webové aplikace.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na architektury menších webových aplikací. Konkrétně se zbývá problematikou porovnáním mikroservisové a monolitické softwarové architektury webových aplikací. Hlavním cílem je porovnat mikroservisovou a monolitickou architekturu pro menší webové aplikace. Za účelem splnění hlavního cíle, byly zvoleny 4 dílčí cíle, kterými jsou:

- Charakterizovat technologie a možnosti, které jsou v současné době při tvorbě webových aplikací k dispozici.
- Vytvořit experimentální webovou aplikaci.
- Vybrat vhodná kritéria pro porovnání těchto dvou architektur.
- Provést experimentální měření aplikace, a na základě jeho výsledků architektury porovnat.

2.2 Metodika

Metodika zpracování teoretické části spočívá ve studiu a analýze odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska práce.

V praktické části bude navržena a implementována experimentální webová aplikace. Tato aplikace bude implementována ve dvou verzích pro každou architekturu zvlášť. Obě vytvořené verze aplikace budou sloužit k porovnání mikroservisové a monolitické architektury na základě stanovených vhodných kritérií. Tato kritéria budou v praktické části práce také stanovena. Poté bude provedeno měření vytvořené webové aplikace. Následně budou na základě teoretických východisek, poznatků získaných během tvorby aplikace a výsledků z měření aplikace formulovány závěry práce.

3 Teoretická východiska

Svět webových technologií je v současnosti opravdu rozsáhlý. V teoretické části jsou tak charakterizovány základní možnosti při vývoji webových aplikací. V některých částech je však věnována větší pozornost určitým technologiím, které se pro vývoj webových aplikací hojně využívají. Tyto technologie představují zástupce možných řešení různých oblastí, kterými se vývojáři při tvorbě webových aplikací zabývají. Mimo to je však těmto technologiím věnován větší prostor s ohledem na praktickou část, kde jsou následně využity při vývoji experimentální webové aplikace. V této části je tak vytvořen teoretický základ, aby se čtenář následně lépe orientoval v praktické části a chápal některé závěry a souvislosti.

3.1 Webové aplikace

Pojem webová aplikace je v této práci hojně využíván. Jeho porozumění je tak naprosto stěžejní pro pochopení celé diplomové práce. Z toho důvodu byly vybrány 4 definice tohoto pojmu.

„Webová aplikace je taková aplikace, kterou není nutno instalovat na zařízení uživatele (počítač, tablet, smartphone) a můžete ji spustit z kteréhokoliv zařízení pomocí webového prohlížeče, protože je spuštěna na straně serveru.“ (4)

„Webová aplikace je souhrnné označení pro software zprostředkovaný internetovým prohlížečem. Na rozdíl od desktopového softwaru ho nemusíte instalovat, stačí, když do prohlížeče (Opera, Google Chrome, Safari) zadáte požadovanou adresu – nezabírají tak žádné místo na disku počítače.“ (5)

„Webová aplikace je software, který nevyžaduje instalaci a namísto toho k němu může být přistoupeno ze vzdáleného serveru skrze webový prohlížeč. Webové aplikace jsou vytvořeny pro interakci, která umožňuje uživatelům posílat a konzumovat data mezi prohlížečem a webovým serverem. Tato interakce může být jednoduché přihlášení k účtu, nebo něco složitého jako provedení platby kreditní kartou.“ (6)

„Webová aplikace je aplikační program, který je uložen na vzdáleném serveru a doručen přes internet skrze rozhraní webového prohlížeče.“ (7)

3.2 Typy webových aplikací

Svět internetových technologií jde skutečně rychle dopředu a možností, jak tvořit webové stránky a webové aplikace je dnes mnoho. V základu se však dělí na dvě hlavní skupiny a sice statické a dynamické, přičemž ty dynamické se následně dělí na řadu dalších typů.

3.2.1 Statické webové aplikace

Statické webové aplikace jsou nejjednodušším typem webových aplikací. Obvykle mají malé množství obsahu a málo interaktivních prvků, které by uživatelům, jakkoliv umožňovaly měnit obsah. (8) (9)

Statické webové aplikace jsou obvykle vytvořeny pouze pomocí HTML, CSS a JavaScriptu na straně klienta. Díky tomu je vcelku snadné je vyvíjet, na druhou stranu ale jakákoliv změna jejich obsahu není snadným úkolem. Pro změnu obsahu je totiž třeba upravovat přímo dané HTML soubory, ve kterých se nachází obsah stránek. To znamená, že je tyto soubory třeba stáhnout, následně modifikovat a poté upravenou verzi opět nahrát na server, přičemž tyto kroky musí provést administrátor stránky. (8) (10)

3.2.2 Dynamické webové aplikace

Dynamické webové aplikace jsou z technického pohledu v porovnání s těmi statickými mnohem komplexnější. Forma a typ dat, která jsou uživateli v aplikaci zobrazována jsou dynamicky utvářena na základě naprogramované logiky aplikace. (8)

K uchování dat, která jsou v aplikaci zobrazována, používají nějakou formu databáze. Dynamické webové aplikace tato data zpravidla také umožňují spravovat skrze administrátorský panel, kde administrátoři mohou provádět úpravy obsahu. Díky tomu jsou úpravy obsahu mnohem snazší než u statických webových aplikací, protože pro změnu obsahu nemusí administrátor přistoupit k serveru a přehrávat soubory s kódem. (8) (10)

Dynamické webové aplikace představují velkou skupinu, kterou lze dělit na další typy, o kterých se lze dočíst ve zbytku této kapitoly.

3.2.3 Multi-page aplikace

Multi-page, nebo česky vícestránkové aplikace, jsou tvořeny vícero stránkami. Tento typ aplikací je jakýmsi protikladem jednostránkových aplikací, o kterých je psáno v následující kapitole.

Vícestránkové aplikace jsou oproti jednostránkovým tradičnější, protože ty jednostránkové zažívají rozmach až v posledních pár letech. (11)

Multi-page aplikace funguje tak, že když si uživatel vyžádá nějakou stránku, tak je mu ze serveru poslána se vším všudy. To znamená, že je poslána struktura, vzhled a veškerý obsah potřebný k jejímu zobrazení. Ve chvíli, kdy uživatel interaguje se stránkou, a obsah stránky se má překreslit, tak je překreslena celá stránka, i když je skutečná změna pouze nějaká drobnost. (12) V současnosti i vícestránkové aplikace umí překreslovat pouze určitou část stránky. (11) Nutno ale uvést, že implementace tohoto chování je v porovnání s jednostránkovými aplikacemi složitější, a pokud je vyžadováno na vícero místech v aplikaci, tak se kód může stát opravdu rychle nepřehledný a náročný na údržbu.

Vícestránkové aplikace také mají další nevýhodu a sice, že serverová a klientská část jsou úzce propojeny. (11)

Na druhou stranu nepopiratelnou výhodou vícestránkových aplikací je, že je všechen obsah generován na serveru. Díky tomuto chování jsou takové aplikace lépe optimalizovány pro prohlížeče, díky čemuž se obecně snáz nastavují tak, aby se uživatelům při vyhledávání zobrazovali na předních příčkách. (13)

3.2.4 Single-page aplikace

Jednostránkové aplikace se od těch vícestránkových liší zejména v samotném principu, jakým fungují. Jednostránkové aplikace totiž pro změnu svého obsahu nevyžadují znovunačtení celé stránky. Díky tomu jsou aplikace rychlejší, čímž se uživatelům lépe používají. Klientská a serverová část aplikace si tak zpravidla vyměňují pouze samotná data, a to většinou v JSON formátu, což je velice lehký formát pro výměnu dat. (14) (15) Jedná se totiž pouze o text, který se skládá buďto z kolekce páru jméno/hodnota, nebo ze seřazeného listu hodnot a je tak snadno pochopitelný jak pro lidi, tak pro počítače. (15)

Z výše zmíněného vyplývá, že klient a server jsou od sebe z velké části odděleny a mohou tak být vyvíjeny vcelku nezávisle na sobě. (16)

Jednostránkové aplikace však nemají pouze klady. Jejich hlavním neduhem je komplikovanější optimalizace pro prohlížeče. To je díky tomu, že většina obsahu je vykreslena až na straně klienta a ze serveru tak chodí pouze data. S tímto principem mají problémy web crawlery, které procházejí internet a indexují obsah. (17) Postupem času však toto přestává být problém, protože dnes existuje řada technologií, které umožňují tvorbu jednostránkových aplikací a zároveň vykreslování určité části obsahu na serveru. I tak je třeba na optimalizaci pro prohlížeče mnohem více myslet v porovnání s vícestránkovými aplikacemi. (11)

3.2.5 Portálové webové aplikace

Portálové webové aplikace umožňují přístup k různým kategoriím na domovské stránce. Portály jsou často využívaný firmami, či státními podniky. Portály umožňují tvorbu osobních profilů, a obsahují funkcí, například chat, emaily, fóra, registrace uživatelů atd. Podstatné je, že značná část obsahu těchto portálů je přístupná pouze členům oněch portálů. (8) (10)

3.2.6 E-commerce webové aplikace

Tento typ aplikací využívají podniky k tomu, aby své zboží mohly nabízet a prodávat na internetu, čímž se zvýší tržby podniků. Do tohoto typů webových aplikací spadají zejména e-shopy. E-commerce aplikace jsou velice komplexní, protože je v nich nutno řešit integraci různých metod elektronických plateb, finanční transakce, objednávky, správu obsahu a v neposlední řadě musí být celá aplikace uživatelsky přívětivá a snadno ovladatelná. (8) (18)

3.2.7 Progresivní webové aplikace

Progresivní webové aplikace jsou vytvořeny za pomocí webových technologií čili za pomocí HTML, CSS a JavaScriptu. Oproti ostatním typům webových aplikací, však působí nativně. To znamená, že pokud je uživatel otevře na mobilním zařízení, tak působí jako mobilní aplikace vytvořené pro Android či iOS. (19)

Aby byla webová aplikace považována za progresivní, tak by měla být:

1. **Objevitelná** – to znamená, že aplikaci lze nalézt skrze vyhledávač. (20)

2. **Instalovatelná** – aplikaci lze nainstalovat do zařízení. Webové aplikace se samozřejmě skutečně neinstalují, takže nezabírají v zařízení žádné místo. Stažením aplikace si uživatel vlastně jen na domovskou stránku svého zařízení uloží odkaz k aplikaci. Tento odkaz se však v zařízení zobrazí úplně stejně jako nativní mobilní aplikace. Uživatel tak vidí ikonku aplikace společně s jejím názvem což budí dojem, že jde o skutečnou mobilní aplikaci. (20) (21)
3. **Odkazovatelná** – aplikaci lze sdílet a otevřít skrze klasický URL odkaz. (20)
4. **Nezávislá na připojení k síti** – aplikace by měla fungovat i bez připojení k internetu. (20)
5. **Progresivně vylepšená** – Její základ by měl fungovat i ve starších prohlížečích, zatímco v těch moderních prohlížečích by měla zcela fungovat a využívat progresivní funkce (20). Jako příklad si lze představit aplikaci, která uživatelům umožňuje nahrávat obrázky. Základní úroveň fungování aplikace tak může být to, že uživatel může nahrát obrázek skrze klasický souborový input, který je podporovaný všemi prohlížeči. Dnešní moderní prohlížeče však umožňují i přístup ke kameře zařízení. Progresivní webová aplikace by tak ke klasickému souborovému inputu mohla přidat i další funkci, kdy má uživatel možnost z aplikace přímo přistoupit ke kameře svého telefonu, pořídit fotku a tu nahrát.
6. **Responzivní** – aplikace by měla být použitelná na jakémkoliv zařízení s prohlížečem bez ohledu na to, jak velkou má toto zařízení obrazovku. (19) (20)
7. **Bezpečná** – Připojení mezi uživatelem, aplikací a serverem by měla být zabezpečená proti případným útokům. Aplikace by tam měla určitě používat ke komunikaci protokol HTTPS. (19) (20) (22)
8. **Působit jako nativní** – Aplikace by měla vypadat a chovat se jako nativní aplikace pro dané zařízení. Také by měla být vyvíjena ve stylu jednostránkové aplikace, aby nedocházelo k častému obnovování celé stránky. (19) (22)
9. **Znovu upoutatelná (Re-engageable)** – Aplikace by měla být schopna opět zaujmout své uživatele, aniž by ji uživatelé museli otevírat. Moderní prohlížeče umožňují svým uživatelům posílat upozornění pomocí tzv. push notifikací.

Progresivní webová aplikace by měla tuto funkci používat například když je v aplikaci nový obsah, či proběhla nějaká aktualizace stávajícího obsahu. (20)

3.2.8 Webové aplikace se systémy pro správu obsahu (CMS)

V současné době není nutné tvořit webové aplikace pouze čistě pomocí psaní kódu, a to právě díky CMS. (23)

CMS (Content management systém), neboli česky systém pro správu obsahu je nástroj, který umožňuje tvorbu webové aplikace bez nutnosti psát všechen kód. Těchto nástrojů existuje celá řada, přičemž mezi nejznámější patří WordPress či Joomla (23) (24)

Tvorba aplikací pomocí CMS je tak značně odlišná od programování aplikace. V CMS je namísto psaní kódu, aplikace tvořena skrze uživatelské rozhraní. Díky tomuto uživatelskému rozhraní je tak možné vytvářet veškerý obsah, jako například měnit texty, vkládat obrázky pomocí jednoduchého drag and dropu, či lehce implementovat jinak komplexní funkce jako například platby. V konečném důsledku je samozřejmě aplikace vytvořena pomocí kódu, avšak tvůrce aplikace je díky uživatelskému rozhraní od tohoto kódu ze značné části odstíněn. (24)

Aplikace pomocí CMS lze tvořit skutečně rychle, avšak má to i své značné nevýhody. Základní princip CMS je ten, že jsou pro tvorbu aplikace předpřipraveny všechny funkce, které jsou při tvorbě aplikace potřeba a tvůrce aplikace následně aplikaci tvoří pouze skrze poskládání těchto předpřipravených funkcí. Samotné CMS poskytuje pouze určitý základ a řada dalších funkcí je tak poskytována od třetích stran. To znamená, že je aplikace následně závislá na tom, jestli jsou dané funkce správně implementovány a udržovány. Navíc některé funkce jsou také zpoplatněny a stojí nemalé částky, což může značně prodražit tvorbu a běh aplikace. Dále jsou zde značné limity, kdy je zkrátka potřeba využívat již předpřipravené funkce, takže pokud má vyvíjená aplikace nějaké specifické požadavky, tak je zde vysoká šance, že nebude možné se zcela vyhnout psaní kódu. V neposlední řadě jsou tyto aplikace zpravidla pomalejší a obtížně optimalizovatelné v porovnání s aplikacemi, které jsou celé naprogramovány. (25)

3.2.8.1 Headless CMS

Klasické CMS umožňuje pouze tvorbu celých aplikací čili klientské i serverové části, nebo také frontendu a backendu. To v současnosti, kdy jsou populární optimalizované, svižné a plynulé jednostránkové webové aplikace není vždy žádoucí. Headless CMS, mezi které patří například Strapi, či Sanity.io, umožňuje kompletní oddělení frontendové a backendové části aplikace. Oproti klasickému CMS, se totiž Headless CMS stará pouze o správu samotných dat. Poskytuje tak rozhraní pouze pro správu dat, a následně prostřednictvím aplikačního programovacího rozhraní (API) zpřístupňuje data frontendu aplikace. Toto řešení tak umožňuje pro frontend aplikace zvolit jakoukoliv technologií, či případně mít vícero aplikací. To je dobré zejména ve chvíli, kdy je potřeba vytvořit webovou i mobilní aplikaci. (26) (27) (28)

3.3 Model klient server

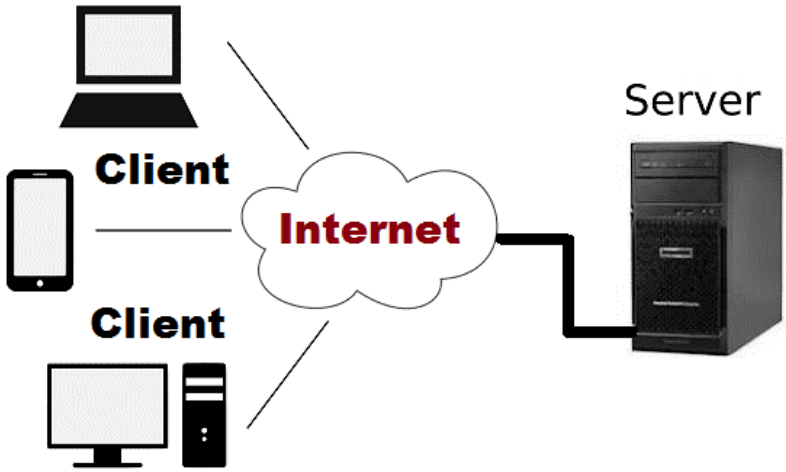
Z předchozího textu vyplývá, že webové aplikace se skládají ze dvou částí a sice z klienta a serveru. Model klient server tak popisuje, jak klientská zařízení vytvářejí požadavky na vzdálená zařízení schopná plnit tyto požadavky (servery). Je dobré zmínit, že na tomto modelu dnes funguje většina služeb internetu. (29)

Klientem je myšlen počítač (host), neboli zařízení, které je schopné obdržet informace, či používat nějakou službu od poskytovatelů služeb (serverů). Na druhou stranu serverem je myšlen vzdálený počítač, který poskytuje informace, nebo umožňuje přístup k nějakým službám. (29) (30)

Interakce v rámci modelu klient server probíhá následovně:

1. Uživatel zadá URL (adresu) webové stránky/aplikace, či souboru o prohlížeče. Čím vznese požadavek na získání dat dané stránky. (29)
2. Prohlížeč díky kontaktování DNS (systém doménových jmen) zjistí IP adresu příslušného webového serveru. (29) (31)
3. Prohlížeč následně skrze komunikační protokol HTTP/HTTPS pošle požadavek na příslušný webový server. (29)
4. Server následně pošle odpověď na daný požadavek. V odpovědi bývají soubory, či data vyžádaná prohlížečem. (29)

5. Prohlížeč obdržená data vykreslí uživateli, díky čemuž se například zobrazí daná webová stránka či aplikace. (29)



Obrázek 1 - interakce v rámci modelu klient-server. Zdroj: (32)

3.3.1 Typy modelů klient server

Model klient server se dále dělí dle toho, na kolik je rozdělen vrstev, či úrovní. (30) (32)

3.3.1.1 Jednovrstvý model

Jednovrstvý model se skládá z nějaké aplikace, která běží na jednom zařízení, které obsahuje vše potřebné k chodu aplikace. (30) (32) (33)

3.3.1.2 Dvouvrstvý model

Tento model se skládá ze dvou vrstev, a sice z uživatelského rozhraní, které se nachází na klientské části a serverové části, která obsahuje data. (30) (32) (33)

3.3.1.3 Třívrstvý model

Přechozí dvouvrstvý model má jednu hlavní nevýhodu, a sice, že dané dvě vrstvy jsou úzce propojeny a každá aplikace tak vyžaduje svého vlastního klienta. Jako adresace tohoto problému vznikl třívrstvý model, který je dnes používán. (30) (32) (33)

Obsahuje následující tři vrstvy:

- **Prezentační** – Tato vrstva se nachází na straně klienta a obsahuje uživatelské rozhraní, což je tedy zpravidla nějaký webový prohlížeč. (30) (32) (33)
- **Aplikační** – Aplikační vrstva už je na straně serveru a obsahuje veškerou aplikační logiku, která je specifická pro danou aplikaci. Stará se tak například o validaci požadavků od klienta, vytváří na tyto požadavky odpovědi, či provádí různé kalkulace. (30) (32) (33)
- **Datovou** – V této vrstvě jsou uchovávána data aplikace. (30) (32) (33)

3.3.1.4 N-vrstvý model

N-vrstvý model představuje nadstavbu třívrstvého modelu. Zásadní rozdíl je, že jednotlivé vrstvy jsou odděleny jak logicky, tak fyzicky. Přičemž některé vrstvy, zejména tedy aplikační, mohou být ještě dále rozvrstveny. (30) (34)

3.3.2 REST a RESTful API

Každá aplikace má tedy dvě hlavní části. Klientskou a serverovou, nebo také frontend a backend. Jednou z možností, jak vytvořit onu serverovou část, konkrétně aplikační vrstvu je právě REST API. Tento typ serverové části, nebo také jinak řečeno typ backendu se dnes hojně využívá při tvorbě backendu pro jednostránkové aplikace, a mimo jiné je následně použit v praktické části práce. Pro pochopení praktické části je nutné si RESTful API představit.

3.3.2.1 API a REST

Nejdříve co je tedy API? API (Application programming interface) je česky řečeno aplikační rozhraní, které je vlastně jakýmsi souborem definovaných pravidel, které říkají jak počítače nebo aplikace mají spolu komunikovat. Rozhraní se tak nachází například mezi jednostránkovou aplikací a webovým serverem, kde funguje jako jakýsi prostředník. Aplikace tak může například volat jen koncové body serveru a spouštět akce či získávat data, aniž by musela sama implementovat nějakou logiku pro provedení daných akcí. (35)

REST nebo také REpresentational State Transfer je soubor architekturních omezení. To znamená, že se nejedná přímo o nějaký standard či protokol, ale spíše o sadu doporučení, které vývojáři při tvorbě API často dodržují, protože to značně usnadňuje práci s vytvořeným API. (36)

3.3.2.2 RESTful API

RESTful API je typ API, které se řídí stylem REST architektury. Vytvořené RESTful API o sobě poskytuje informace v podobě informací o svých zdrojích. Zdroji jsou zde myšlena data, se kterými RESTful API pracuje. To znamená, že pokud je RESTful API zavoláno, tak server pošle (transfers) klientovi reprezentaci (representation) stavu (state) požadovaného zdroje. Reprezentace zdroje bývá zpravidla v JSON formátu, i když to není vyloženě nutné. (37) JSON je zkrátka používán hlavně díky tomu, jak lehce je srozumitelný pro lidi i pro počítače. (38) Mimo JSON však lze použít jiné formáty, například XML nebo CSV. (39)

Aby tedy API bylo považování za RESTful, tak musí splňovat následující kritéria:

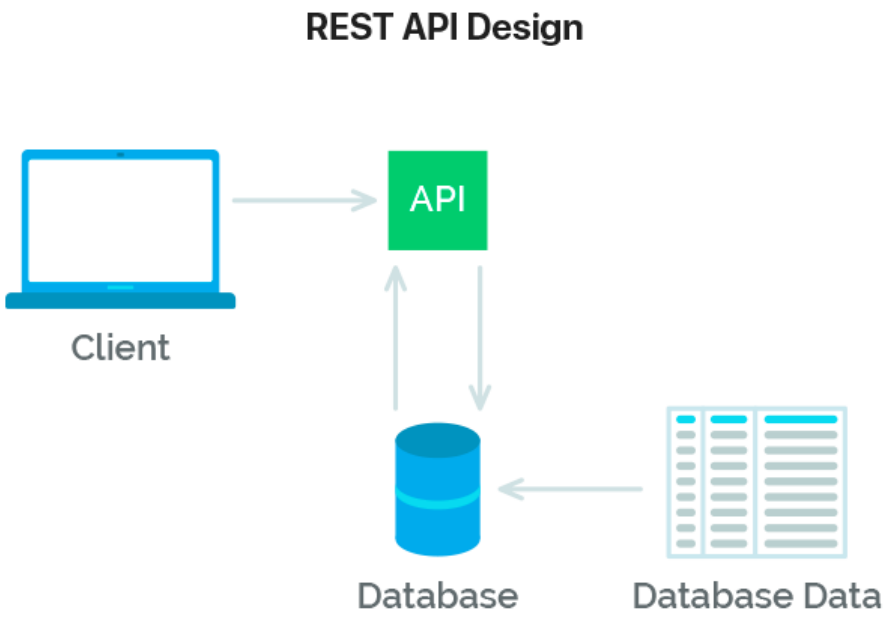
1. **Jednotné rozhraní**. Mělo by zde být jedno rozhraní, které je vždy stejné bez ohledu na to, z jakého zařízení či typu aplikace jsou data požadována. Rozhraní by mělo být založeno na zdrojích a požadavky na server tak musí obsahovat identifikátory těchto zdrojů. Dále odpověď serveru musí obsahovat dostatek informací, aby mohl klient tento zdroj modifikovat, pokud má klient k takovým akcím oprávnění. Každý požadavek také musí obsahovat veškeré informace, které server potřebuje pro vykonání požadavku a následně každá odpověď musí obsahovat veškeré informace, aby klient dané odpovědi porozuměl. Posledním požadavkem je v rámci jednotného rozhraní je, že by měl být dostupný hypertext či hypermedia. To znamená, že pokud lze daný zdroj nějak měnit, tak server by pomocí odkazů v rámci odpovědi měl informovat o možných způsobech změny stavu daného zdroje. Zde nutno říct, že většina běžných API se doporučení o odkazech příliš nedrží. (37) (40)
2. **Oddělení klienta a serveru**. Toto pravidlo vlastně říká, že klient a server mohou jednat nezávisle na sobě. Veškerá komunikace probíhá pouze pomocí požadavků, které přichází od klienta a odpovědí od serveru, které vznikají jako reakce na požadavky. (37)

3. **API musí být bez stavové (stateless)**. To se týká samotné komunikace klienta a serveru, kdy veškerá interakce nesmí být závislá na nějakém stavu. Jinými slovy server neukládá nic o požadavcích od klienta a jedná tak s každým požadavkem, jako by byl úplně nový. Kvůli tomu se v RESTful API nepoužívají žádné sessions či historie. Každý požadavek musí obsahovat vše, co je k obdržení požadovaných zdrojů nezbytné. Toto se týká i přihlašování a následné autorizace. (41)
4. **Cachovatelné**. Data, která proudí ke klientovi by měla být cachovatelná kdykoliv je to možné. (36)
5. **Systém vrstev**. To znamená, že REST umožňuje například nasadit API na jeden server, ukládat data na jiném serveru a ověřovat požadavky na zcela jiném serveru. (41) To je pouhým příkladem, a jednotlivé vrstvy tak mohou obsahovat dále například vrstvu cáchování, vrstvu pro vyvažování zátěže a další. Důležité je, že bez ohledu na počet vrstev, pro klienta je chování stále stejné a nijak toto vrstvení nezasahuje do komunikace mezi klientem a serverem. (37)
6. **Kód na vyžádání (Code-on-demand).** Tento požadavek není povinný. Jedná se o to, že klient může od serveru vyžádat kód, přičemž odpověď obsahuje spustitelný kód. (37)

3.3.2.3 Princip RESTful API

Když je použito RESTful API, tak spolu klient a server komunikují skrze požadavek a odpověď, konkrétně skrze protokol HTTP. Klient je tak zcela odstíněn od databáze a veškeré serverové logiky nutné k získání dat. Klient pouze žádá data, která mu jsou vystavena pomocí prostředníka ve formě API, konkrétně pomocí RESTful API.

Pokud klient například potřebuje data o konkrétním uživateli, tak pro jejich získání musí provést GET požadavek na server, přičemž v daném požadavku musí v URL zahrnout identifikátor zdroje, čili například "/api/uživatelé/<id uživatele>". Tímto způsobem klient zkontaktuje koncový bod (endpoint), který server v rámci RESTful API vystavuje, načež server spustí logiku daného koncového bodu a následně pošle odpověď zpravidla ve formě již zmíněného JSONu.



Obrázek 2 - Schéma komunikace klienta a RESTful API. Zdroj: (42)

3.3.2.4 Validace v RESTful API pomocí tiketů (tokenů)

Zabezpečení je důležitou otázkou každé aplikace. API, a obzvláště RESTful API k této mají problematice trochu jiný přístup, než bývá zvykem u klasických mnohostránkových aplikací, protože nepoužívají sessions. RESTful API mají pro zabezpečení samozřejmě trochu víc než jednu možnost, ale za účelem neodvracení pozornosti od skutečného tématu této práce je tu přiblížena pouze ta nejpoužívanější z nich, protože je také využita v rámci praktické části. (43)

RESTful API tedy nemají držet stav. To znamená, že přihlášení nemůže být drženo v rámci sessions a navíc, každý požadavek musí obsahovat vše potřebné k získání zdrojů, a to včetně nějakého prokázání o oprávnění k datům.

Takové oprávnění je například platné přihlášení do aplikace, bylo by však vysoce nebezpečné za prvé vůbec ukládat přihlašovací údaje na straně klienta a za druhé posílat tyto údaje s každým požadavkem. (43)

Hojně využívaným řešením tohoto problému je zabezpečení založené na tiketech. Princip je vcelku jednoduchý, uživatel se přihlásí do aplikace, a pokud je přihlášení úspěšné, tak server vygeneruje unikátní tiket, který uživatelovi pošle, a uživatel se následně prokazuje již jen vygenerovaným tiketem. Tiket je vygenerován pomocí kombinace dat o uživateli a nějakého tajemství, které zná pouze server, díky čemuž je pak pouze server schopen ověřit validitu tiketu. Dobré je, že pro ověření tiketu server potřebuje znát pouze to tajemství, a nemusí se vůbec dívat do databáze. Nevýhodou je, že kdokoliv se k tiketu dostane, má přístup k datům aplikace. Existuje řada způsobů, jak se neoprávněnému přístupu bránit, nebo alespoň minimalizovat jeho dopad. Populárním způsobem je zkrátka nastavení krátké životnosti tiketů. Délka platnosti tiketu záleží na rozhodnutí vývojářů, avšak často to bývá v řádu maximálně desítek minut. U krátké životnosti samozřejmě musí přibýt mechanismus obnovení tiketu, aby uživatel nebyl neustále odhlašován. To se provádí vcelku jednoduše, zkrátka přibude další tiket, který slouží jako obnovovací a funguje stejně jako ověřovací tiket, akorát se používá pouze pro obnovení platnosti ověřovacího tiketu. (44)

Co se samotného tiketu týče, tak dnes nejvyužívanější formou je tzv. JSON Web Token. Tento tiket může obsahovat řadu informací, například kdy má expirovat, samotné údaje o uživateli, nebo kdo tiket vystavil. JSON Web Token je digitálně podepsán, což umožňuje serveru poznat, zdali s tiketem někdo neoprávněně manipuloval. (44)

3.4 Technologie využívané pro vývoj webových aplikací

3.4.1 Jazyky

Nedílnou součástí vývoje webových aplikací jsou samozřejmě programovací jazyky. Těch, které lze použít pro tvorbu webových aplikací je velká řada a jako například PHP, C#, Python, Java a další. (45)

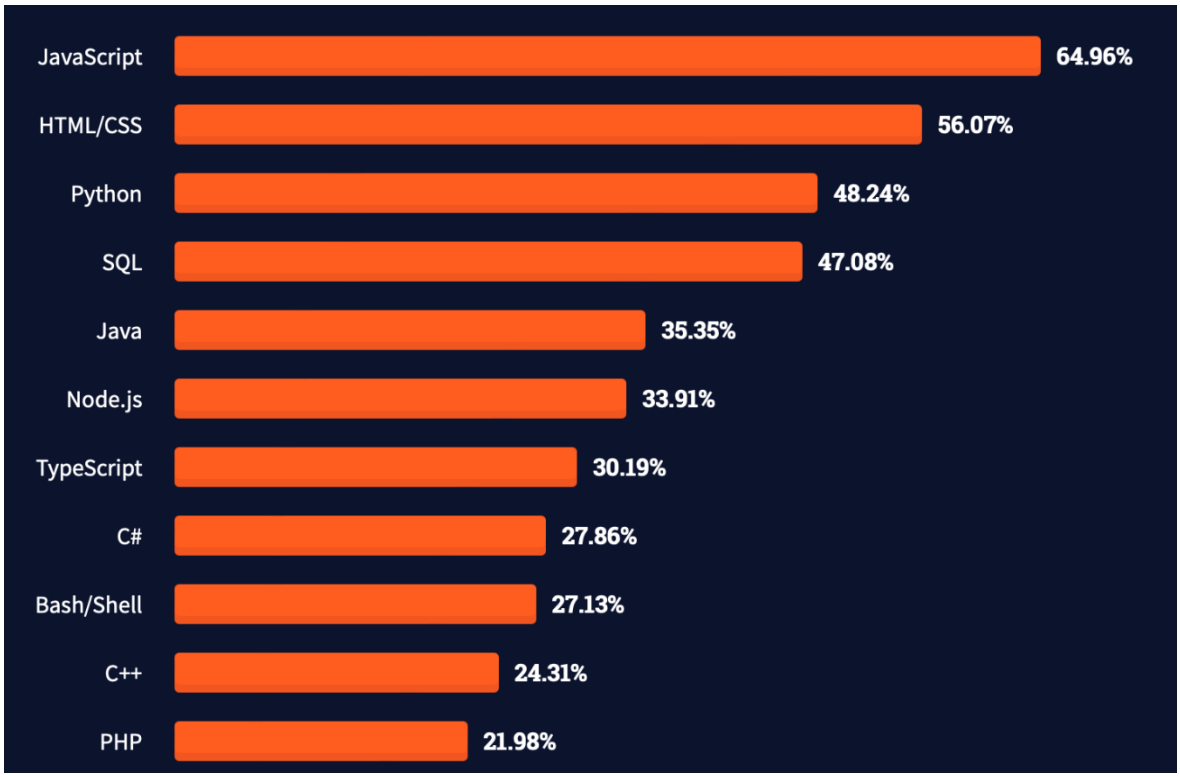
Na obrázku č. 3 jsou zobrazeny statistiky používaných jazyků v roce 2021. Tyto statistiky byly získány na základě dotazníkového průzkumu, kde odpovědělo více než 83 000 vývojářů z celého světa. Na dotazník odpovídali profesionální vývojáři, studenti, kteří se učí programovat, dále také lidé, kteří programují ve volném čase, či lidé, kteří v rámci práce programují pouze občas. Jedná se celkové statistiky, takže jsou v nich zahrnuty i jazyky, které nejsou přímo používány pro tvorbu webových aplikací. Konkrétně je tím myšlen Bash/Shell a C++. (46)

Lze vidět, že nejpoužívanějším jazykem je JavaScript. Tomuto jazyku se v současnosti při vývoji webových aplikací téměř nelze vyhnout. Jedná se o nejpoužívanější jazyk pro tvorbu dnes velice moderních jednostránkových aplikací. Lze jej také použít pro tvorbu serverové části aplikace, ve které je nejvíce používán v rámci prostředí Node.js, které lze ve statistice také nalézt a bude mu věnováno více prostoru v kapitole „3.4.3 Node.js“. V statistice lze vidět také jazyk TypeScript, který představuje pouhou nadstavbu JavaScriptu. (47) (48)

Druhé místo obsadily HTML/CSS. Je to proto, že společně s JavaScriptem jde o technologie, které jsou v drtivé většině používány pro tvorbu frontendové části webových aplikací. (49)

Ostatní jazyky, zejména Python, Java, C# či PHP jsou totiž v rámci webových aplikací používány pouze na serveru. Pomocí těchto jazyků se tak buďto tvoří pouze čistě backend webových aplikací a práci s daty je následně nejčastěji umožněna pomocí tzv. RESTful API, které bude více rozebráno v kapitole „3.3.2 REST a RESTful API“, nebo používají tzv. šablony. Tyto šablony jsou jakousi strukturou stránky, která má být uživateli vykreslena. Šablona obsahuje HTML/CSS, případně JavaScript a proměnné daného jazyka. Při skutečném vykreslení obsahu uživateli, jsou dané proměnné nahrazeny skutečnými hodnotami, čímž je šablona transformována do skutečného HTML souboru. (50) (51)

Čtvrté místo obsadil jazyk SQL, kterému se také lze těžko vyhnout. Jedná se totiž o jazyk využívaný pro práci s relačními databázemi, které jsou dnes jedním z nejpoužívanějších typů databází. (52)



Obrázek 3 - statistiky nejpoužívanějších programovacích jazyků v roce 2021 získaných na základě dotazníkového průzkumu. Zdroj: (46)

3.4.1.1 JavaScript a TypeScript

JavaScript je tedy nejpopulárnějším jazykem v oblasti webových aplikací. Je třeba zde zohledňovat i jazyk TypeScript, který je pouhou nadstavbou JavaScriptu. Ostatně zejména TypeScript je použit i v rámci praktické části, a tak je vhodné těmto dvěma jazykům věnovat alespoň krátký úvod.

JavaScript je multiplatformní, objektově orientovaný skriptovací jazyk, který se v současnosti používá zejména pro vývoj webových aplikací, a to jak na straně klienta, tak na straně serveru. (53) (54)

Přes značné rozšíření a popularitu JavaScriptu má tento jazyk řadu úskalí. Jedním z největších je, že je dynamicky typovaný. To znamená, že typy proměnných, či návratové hodnoty funkcí se v rámci kódu nedefinují a jsou kontrolovány až za běhu programu. Takový přístup umožňuje velice rychle psát spustitelný kód. Ve větších aplikacích je však velice nepraktický, protože s přibývajícím množstvím kódu značně komplikuje jeho přehlednost a předvídatelnost. (55)

Proto vznikl TypeScript, který je staticky typovaným jazykem, kde je ona definice proměnných či návratových hodnot funkcí vyžadována a kontrolována během psaní kódu. (55) Jak již bylo řečeno, TypeScript je pouze nadstavbou JavaScriptu. Z toho vyplývá, že kód napsaný v TypeScriptu nelze hned spustit a musí být nejdříve komplikován do JavaScriptu. (56)

3.4.2 Frameworky

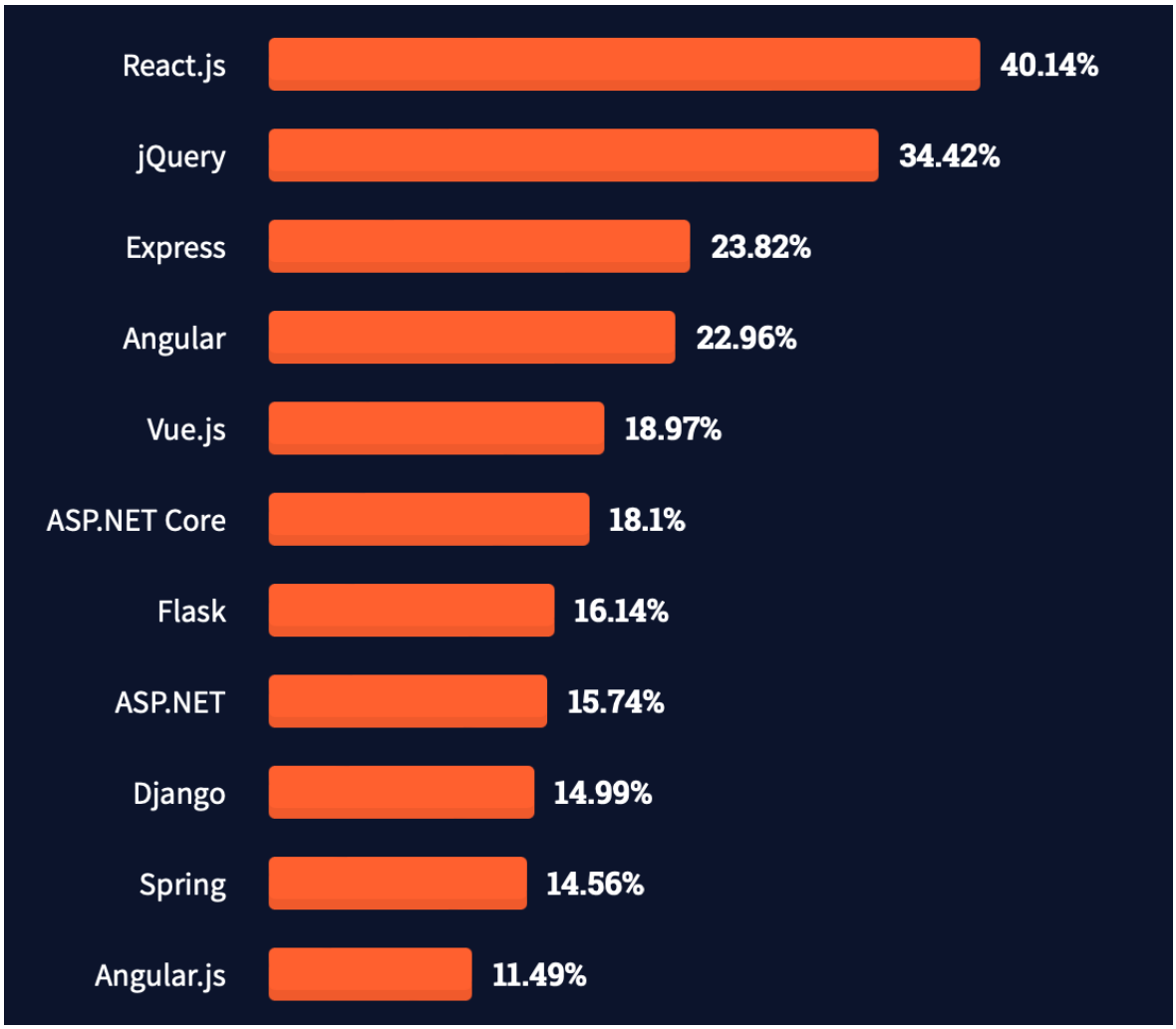
Webové aplikace se v dnešní době netvoří bez jakéhokoliv základu. Bylo by to velice náročné a řada problémů se při vývoji aplikací neustále opakuje. Proto se k tvorbě aplikací používají různé frameworky. Framework je jakýsi základ, který obsahuje řadu nástrojů a předpřipravených funkcí, díky kterým je tvorba aplikací mnohem snazší a rychlejší. Frameworky bývají úzce spojeny s programovacím jazykem, ve kterém byly vytvořeny a jsou vždy použitelné pro specifickou sadu úkolů čili například pro tvorbu webových aplikací. (57)

Na obrázku č. 4 lze vidět statistiky nejpoužívanějších Frameworů pro tvorbu webových aplikací. Jedná se o výsledky získané na základě stejného dotazníkového průzkumu jako výsledky zobrazené na obrázku č. 3. Rozdíl je pouze v počtu respondentů, protože zde odpovědělo 73 tisíc vývojářů. (46)

S ohledem na předchozí statistiky, kdy vyšel nejpopulárnější jazyk JavaScript není žádným překvapením, že 6 příček obsadily JavaScript frameworky. React.js, jQuery, Angular, Angular.js a Vue.js jsou frameworky používané pro tvorbu zejména jednostránkových webových aplikací. Express je backendovým JavaScript frameworkem používaným na straně serveru. (58)

Jsou zde také C# frameworky ASP.NET Core a ASP.NET. Oba tyto frameworky jsou vyvíjeny Microsoftem a lze je použít pro tvorbu webových aplikací. Zásadním rozdílem je, že ASP.NET Core lze použít i na jiných operačních systémech, než je Windows, a sice na MacOS a Linuxu. Mezi těmito frameworky je samozřejmě řada dalších rozdílů, avšak tyto rozdíly nejsou v rámci této práce nijak podstatné. (59)

Dále jsou zde Python frameworky Flask a Django a neposlední řadě je zde Java Framework Spring. (60) (61)



Obrázek 4 - statistiky nejpoužívanějších frameworků pro tvorbu webových aplikací v roce 2021 získaných na základě dotazníkového průzkumu. Zdroj: (46)

3.4.3 Node.js

Node.js je tedy v současnosti jednou z nejpoužívanějších technologií pro tvorbu webových aplikací. Společně s již popsaným jazykem TypeScript je také použit v rámci praktické části. Za účelem lepšího pochopení praktické části a přiblížení důvodů stojících za popularitou této technologie, je Node.js v této kapitole blíže popsán.

Node.js je asynchronní, událostmi řízené jedno vláknové prostředí pro JavaScript. (62) (63) Definice Node.js je krátká, ale každé slovo je pro pochopení vhodné vysvětlit.

3.4.3.1 Prostředí

JavaScript ke svému spuštění potřebuje vhodné prostředí. V rámci předchozí kapitoly bylo řečeno, že JavaScript lze použít jak na klientovi, tak na serveru. V rámci klientské části je tím prostředím prohlížeč. Na straně serveru je tímto prostředím právě Node.js, který používá V8 JavaScriptí engine, který je kromě Node.js využíván i v prohlížeči Google Chrome. Tento engine je totiž navržen tak, aby byl nezávislý na prohlížeči, ve kterém je hostován. (64)

3.4.3.2 Jedno vláknové

Node.js je jedno vláknové, čímž je myšleno, že Node.js aplikace ke svému běhu využívají jedno vlákno procesoru. To znamená, že Node.js je schopen zpracovávat v jednom čase pouze jednu událost. Jedná se o jakousi poloviční pravdu, což bude vysvětleno v následujících kapitolách o event-driven a asynchronní povaze této technologie.

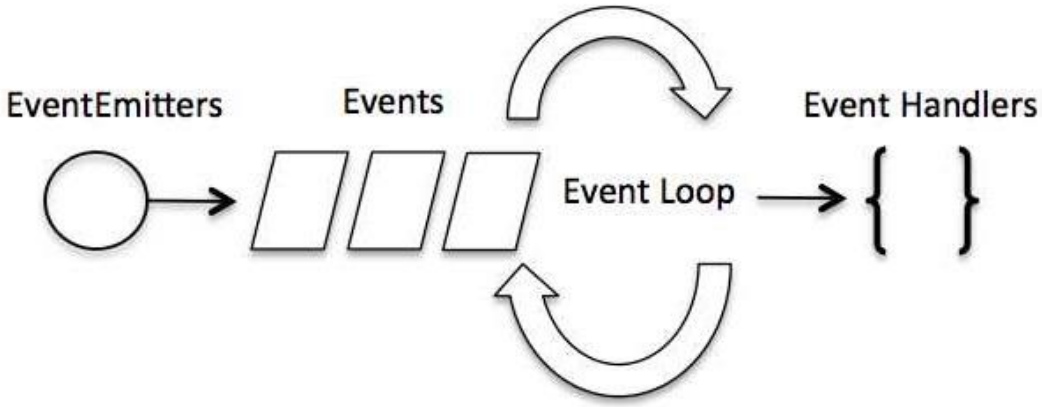
3.4.3.3 Event-driven neboli událostmi řízené

Node.js opravdu hojně využívá události (eventy), a je to také jedním z důvodů, proč je Node.js vcelku rychlé. Při spuštění Node.js serveru se tak nejdříve nadefinují proměnné, deklarují funkce a následně se čeká, až se stane nějaká událost, která dle zmíněných definic spustí některou z deklarovaných funkcí. Aby to celé fungovalo, tak je v této událostmi řízené technologii tzv. událostní smyčka, která naslouchá událostem a pokud se nějaká událost stane, tak spustí předem deklarovaný kus kódu, což bývá často nějaká funkce neboli “event handler”. (65)

Jsou zde tak dvě části:

- Funkce, které se mají při události spustit, což jsou již zmíněné „event. handlers“
- Smyčka akcí (event loop), která naslouchá událostem a spouští zmíněné „event. handlers“

Na obrázku č. 5 je zobrazeno, jak smyčka událostí funguje. Události jsou emitovány, ty jsou pak zpracovány smyčkou událostí, která spouští příslušně funkce neboli „event. handlers“.



Obrázek 5 - NodeJS smyčka událostí. Zdroj: (8)

3.4.3.4 Asynchronní

Node.js je asynchronní povahy, což je značně propojeno s jeho událostmi řízeným principem. V některých článcích se díky tomu lze dočíst, že je tzv. non-blocking, neboli neblokující. To je proto, že kód, který je asynchronní je také kód, který neblokuje spouštění dalšího kódu. (66) Node.js je tak psáno ve stylu, že se nadefinují tzv. spouštěče událostí a pro ně také asynchronní/nebklokující kód, který má být spuštěn při daných událostech.

Díky asynchronní povaze není tak zcela pravdou, že se v Node.js aplikacích zpracovává pouze jedna věc. Node.js jako takové je opravdu jedno vláknové a má tak pouze jedno hlavní vlákno, které pokud se zablokuje, tak se celá aplikace skutečně zasekne. Asynchronní kód ale není po celou dobu zpracováván tímto vláknem. Uveďme si příklad asynchronní funkce, která volá databázi. V takovém případě se funkce spustí, ale ve chvíli, kdy dojde na volání databáze, Node.js se nezastaví a nečeká na odpověď, ale odloží toto volání „stranou“ a pokračuje dál v exekuci jiného kódu a k této funkci se vrátí až ve chvíli, kdy z databáze přijde odpověď. (67)

Je tak otázkou, kam „stranou“ se asynchronní kód odkládá. Node.js není pouze ono hlavní vlákno, ale jsou zde i C++ API, která pak asynchronní úkoly řeší a ve chvíli, kdy jsou vyřešeny, tak je vrátí skrze smyčku událostí zpět do hlavního vlákna. (68) Je to samozřejmě trochu zjednodušené, a to zejména z důvodu, že tato práce není čistě o Node.js, a větší množství detailů by spíše odvádělo od hlavního tématu bez smysluplného přínosu. Nicméně

základní princip je dobré pochopit, protože to, jak Node.js funguje, značně ovlivňuje, jaké jsou jeho výhody, případy užití a výkon.

3.4.3.5 Výhody Node.js

Node.js je tedy jedno vláknové a pracuje asynchronně, čímž se liší od jazyků jako například PHP či Java. (69) Díky tomu je vhodné pro asynchronní úkoly například asynchronní vstupně/výstupní operace. (70) Mezi tyto operace se řadí například volání do databáze, kdy se zpustí nějaká funkce, ta zavolá databázi, následně Node.js pokračuje ve spouštění další logiky, a když se z databáze vrátí výsledek, tak se Node.js vrátí ke spuštění logiky, která na odpověď z databáze reaguje. (67) Vstupně/výstupní operace jsou operace, které zahrnují komunikaci mezi dvěma procesujícími systémy. Ve zmíněném příkladu je to Node.js a databáze. (71)

Síla Node.js je tedy v tom, že funguje jako jeden proces, bez vytváření nového vlákna pro každý požadavek. Node.js poskytuje sadu vstupně/výstupních primitiv ve své standardní knihovně za použití neblokujících paradigmat, díky čemuž je hlavní vlákno blokováno spíše výjimečně.

Pokud Node.js provádí vstupně/výstupní operaci, jako například čtení ze sítě, přístup do databáze nebo souborového systému, tak namísto blokování vláken procesoru, jsou tyto činnosti odloženy a je na ně navázáno, až když se vrátí odpověď. (69)

Výše zmíněné z něj dělá vhodného adepta pro řadu REST API.

3.4.3.6 Nevýhody NodeJS

Na druhou stranu Node.js není vhodný pro intenzivní úkoly, které vyžadují procesor, a to právě díky tomu, že má pouze jedno hlavní vlákno. To znamená, že pokud nějaká ze zavolaných funkcí spustí synchronní logiku, která zabere hodně času, tak po celou dobu spuštění této logiky Node.js nemůže dělat nic jiného a jsou tak zablokovány všechny požadavky na webovou aplikaci. (72)

Pokud tak tvořená aplikace vyžaduje nějaké dlouhé kalkulace, tak je třeba zvážit, jestli je Node.js pro danou aplikaci vhodné.

3.4.3.7 Využití Node.js

Node.js má vskutku širokou škálu využití, jen je samozřejmě třeba zvážit jeho silné a slabé stránky, jinak v něm lze provádět následující:

- Vytvářet celé webové aplikace (62)
- Tvořit RESTful API (73)
- Node.js umožňuje otevírání, čtení, zápis, mazání a zavírání složek na serveru (62)
- Lze skrze něj pracovat s databázemi, přičemž je dobré zmínit, že Node.js není nijak propojeno s konkrétním typem databází. Toto je zmíněno hlavně proto, že v řadě článcích a kurzech je Node.js používáno pouze ve spojení s databází MongoDB, což je dokumentová databáze, která místo tabulek využívá dokumenty. (74) Node.js lze zcela bez problémů využít i s jakýmkoliv relačním typem databází.
- Pomocí Node.js lze také provádět tzv. web scraping, což je technika získávání dat z různých internetových zdrojů. (73) (75)

3.5 Monolitická architektura

Vývoj aplikací nezahrnuje pouze volbu těch správných technologií, mimo to je třeba zvolit samotnou architekturu aplikace. Naprosto základní rozdělení aplikací dle jejich architektury je na monolitické a mikroservisové.

Aplikace napsaná v monolitické architektuře obsahuje většinu nebo všechny funkce dané aplikace v rámci jednoho velkého celku. (76)

Taková aplikace v sobě obsahuje:

- Autentizaci a Autorizaci – aplikace si řeší jak přihlašování, tak následné ověřování oprávnění k provedení požadovaných akcí. (77)
- Prezentační vrstvu – tou je myšlena vrstva, která se stará o HTTP požadavky a odpovídá na ně buďto vykreslením celých stránek v podobě XML či HTML, anebo v případě API v podobě JSON nebo XML. (77)
- Byznys logiku – tímto je vlastně myšlen veškerý kód který říká, jak mají být data vytvářena, ukládána a procesována. (77) (78)

- Databázovou vrstvu – objekty zodpovědné za přístup k databázi (77)
- Případnou integraci s dalšími službami, čímž je myšleno například propojení s nějakým dalším API (77)

3.5.1 Výhody monolitické architektury

- Vývoj je rychlejší a jednodušší. (79) To je hlavně proto, že na začátku není nutné aplikaci plánovat tolik do detailů, a pro samotný vývoj v nejjednodušších případech stačí znát jeden jazyk, například PHP. Tato jednoduchost je však zrádná a zejména to, že lze do jisté míry přeskočit plánování a návrh aplikace se nemusí vůbec vyplatit, o čemž bude ještě hovořeno ke konci praktické části. Také samotná jednoduchost vývoje přestává s růstem aplikace platit.
- Snáz se nasazuje do produkce. U některých aplikací dokonce stačí vzít napsaný kód a překopírovat jej na server. (80) Nelze to však říct obecně a například aplikace napsané v JavaScriptu vyžadují jisté kroky navíc. Také je zde v dnešní době Docker, který se hojně při nasazování aplikací používá a při jeho použití prosté kopírování souborů nestačí. O této technologii ještě bude hovořeno dále.
- Má jednu databázi, kterou sdílí celá aplikace (79)

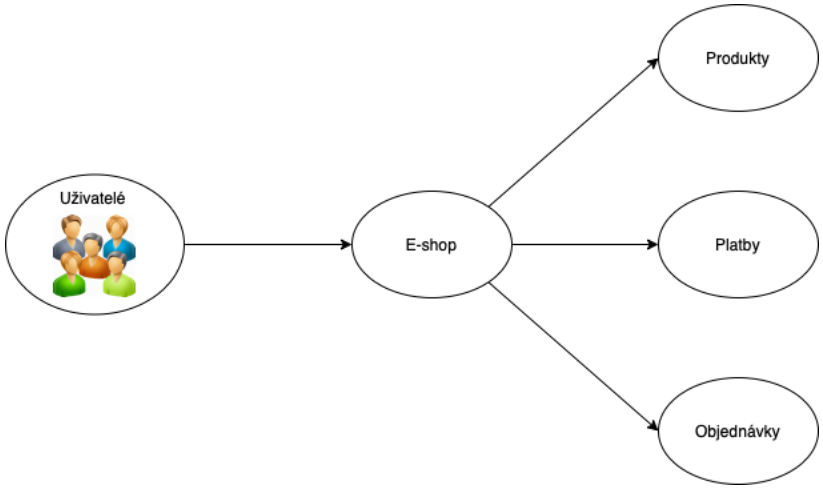
3.5.2 Nevýhody monolitické architektury

- Tím, že je aplikace napsána jako jeden velký celek, tak jsou také všechny její části úzce provázány. Tento fakt značně komplikuje řadu oblastí. Je těžké rozšiřovat aplikaci o nové funkce, protože to vyžaduje přepsání části aplikace a čím větší je změna, tím větší část aplikace se musí přepsat. (79) Díky provázanosti aplikace není tolik spolehlivá, protože malá změna může vyvolat pád kompletně celé aplikace. (77)
- Výše zmíněné problémy se s růstem aplikace jen zhoršují což ztěžuje i samotné udržování aplikace. (77)
- Nasazení i sebemenší změny vyžaduje nové nasazení celé aplikace. (77)
- Škálování aplikace je značně omezené, protože lze maximálně nasadit další instanci celé aplikace. (81)

3.6 Mikroservisová architektura

Při vývoji aplikací lze kromě monolitické architektury použít také mikroservisovou. V mikroservisové architektuře je aplikace rozdělena do menších logických celků, kterým se říká služby (services). Služby jsou na sobě nezávislé a jsou propojené pouze pomocí nějakého komunikačního řešení, avšak na venek se aplikace stále jeví jako jeden celek.

Například e-shop by mohl být rozdělen na několik částí, jmenovitě na produkty, platby a objednávky. Díky vzájemné nezávislosti těchto logických celků mohou být jednotlivé služby vyvíjeny, opravovány či vylepšovány nezávisle na sobě. (81)



Obrázek 6 - ilustrativní schéma mikroservisové architektury. Zdroj: vlastní zpracování

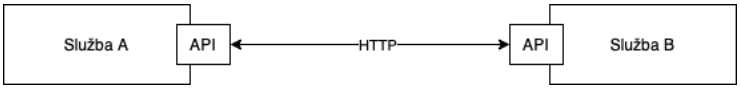
3.6.1 Způsoby komunikace

Je jasné, že služby v rámci mikroservisové architektury spolu nějak spolupracovat a komunikovat musí. Komunikace se zde tak stává naprosto klíčovým prvkem aplikace, přičemž její implementace není snadným úkolem a je na výběr z řady možností, avšak základní dělení komunikace je na synchronní a asynchronní.

3.6.1.1 Synchronní komunikace

Synchronní způsob komunikace je založen na principu požadavku a odpovědi zpravidla pomocí protokolu HTTP. Je to vcelku jednoduchý způsob komunikace, kdy klient zadá požadavek a služba na něj odpoví. Jedná se o způsob, který zná snad každý vývojář a tak se nabízí i pro komunikaci mezi službami. Synchronní komunikace by však měla probíhat pouze mezi uživatelem a aplikací. Pro komunikaci mezi službami navzájem zkrátka

není příliš vhodný. (82) Je to proto, že jakmile by se služby začaly napřímo volat, tak by musely vzájemně vědět o své existenci, což by je s přibývající komunikací stále více vzájemně propojovalo. (83)

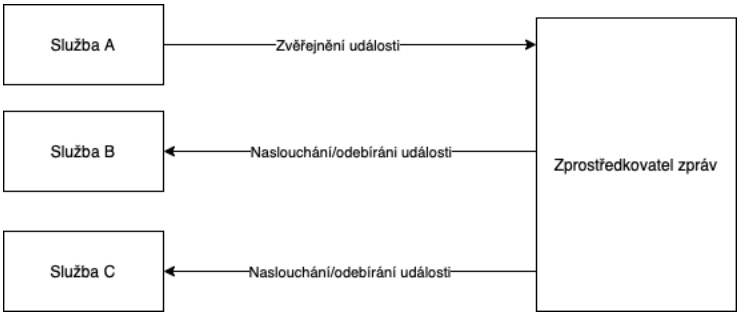


Obrázek 7 - ilustrační schéma průběhu synchronní komunikace přes protokol HTTP. Přeloženo z: (84)

3.6.1.2 Asynchronní komunikace

Asynchronní způsob komunikace je obecně vhodnějším způsobem pro vzájemnou komunikaci mezi službami. (82) Komunikace je označována také jako „event-based“, neboli komunikace založená na událostech. Je to kvůli principu jejího průběhu, služba zde totiž jen řekne, že se stala nějaká událost a očekává, že ostatní strany zúčastňující se komunikace vědí, jak na danou událost zareagovat. Služby jsou tak od sebe z velké části odděleny, protože pouze říkají, že se něco stalo a vůbec se nestarají o to, kdo daným událostem naslouchá či jak na ně reaguje. Aby to mohlo celé fungovat, tak je nutné řešit, jak přesně události vytvářet a jak jim naslouchat. Pro tyto účely jsou zde tzv. zprostředkovatelé zpráv. (83) Zprostředkovatelů zpráv existuje celá řada, přičemž mezi nejznámější se řadí například RabbitMQ, Apache Kafka či Red Hat AMQ. (85) Problematika asynchronní komunikace je vskutku komplexní a jedná se tak o mocné nástroje, které jsou náročné na naučení a značně zvyšují náročnost vývoje mikroservisových aplikací. (83)

Na následujícím obrázku je znázorněna situace, kdy služba A má nějakou událost, kterou zveřejňuje přes zprostředkovatele zpráv. Služba se nestará o nic jiného než samotné odeslání události zprostředkovateli zpráv. Vše ostatní tak má na starosti zprostředkovatel zpráv, který zabezpečuje rozeslání této události službám, které mají o tento typ události zájem. Na obrázku se jedná konkrétně o služby B a C.



Obrázek 8 - ilustrační schéma průběhu asynchronní komunikace založené na událostech. Inspirováno: (84)

3.6.2 Výhody mikroservisové architektury

- Rychlejší nasazení do produkce, protože není nutno nasazovat celou aplikaci při změně jedné služby jako je tomu u monolitických aplikací. (86)
- Aplikace je vcelku robustní, protože pokud přestane fungovat některá se služeb, tak ostatní jsou do jisté míry nadále schopny fungovat. (87)
- Díky tomu, že jednotlivé služby představují autonomní celky, tak lze každou službu vyvíjet pomocí jiných technologií a jazyků. (87)
- Pro vývojáře je snazší se v službách zorientovat. Tento klad vychází z toho, že jednotlivé služby jsou relativně malé. Pro vývojáře je zkrátka snazší pochopit jednotlivé malé na sobě nezávislé služby než jeden velký celek, který má v sobě řadu souvislostí. (79) Díky tomu je také snazší přidávat nové funkce, či rozšiřovat ty stávající.
- Chyby v aplikaci jsou zde izolovány na menší oblast, a to konkrétně na oblast služby, která obsahuje danou chybnou funkci. (77)
- Lze lépe škálovat, protože je možné zkrátka nasazovat nové instance pouze konkrétních vytížených služeb. (88)
- Poslední z výhod je rychlost aplikace. (81) Nad tou je ale otazník, protože mikroservisová aplikace není automaticky rychlejší než monolitická a občas tomu je spíše naopak. Rychlost totiž značně záleží na návrhu jednotlivých služeb, konkrétně na tom, jestli se podaří navrhnout služby tak, aby byla jejich zátěž co nejvíce rozložena a zároveň aby mezi službami probíhalo co nejméně komunikace. První kritérium je vcelku jasné, to druhé je spojeno s již popsanou komunikací mezi službami. Požadavek, který lze vyřešit v rámci jedné služby je zkrátka mnohem rychlejší než požadavek, který k vyřešení potřebuje propojení a komunikaci vícero služeb najednou.

3.6.3 Nevýhody mirkoservisové architektury

- Je nutné řešit komunikaci mezi službami, což přidává na obtížnosti vývoje aplikace. (77)

- Možnost kombinace různých technologií napříč službami je zrádné, a pokud není volba technologií hlídána, tak se může stát, že je každá služba vyvíjena pomocí něčeho jiného. To značně komplikuje údržbu aplikace, protože je pro ni třeba řada odborníků, kteří vzájemně nejsou zastupitelní. (86)
- Zatímco následné nasazování nových verzí služeb je svižné, tak prvotní nastavení je složitější, protože vyžaduje využití většího počtu nástrojů. Aplikaci totiž nelze jen zkopírovat na server. Pro jednotlivé služby je nutné nastavit jejich prostředí a nějak je orchestrovat, aby bylo možné například dynamicky nasazovat nové instance vytížených služeb. (81)
- Je náchylnější na bezpečnostní rizika. (79) Jednotlivé služby běží ve svých autonomních prostředích to znamená, že také musí být jednotlivě nastavena, přičemž nastavení těchto prostředí se odvíjí od technologií využitých pro vývoj dané služby. To znamená, že je vícero míst, kde lze udělat chybu nastavení, čímž se aplikace stává náchylnější k útokům.
- Prvotní investice do mikroservisového řešení je trochu dražší. To zkrátka souvisí s nutností propojení služeb, nastavením jejich prostředí, a provedením dalších nezbytných kroků, aby vůbec šlo s vývojem začít. Dále také samotné pokrytí nákladů na provoz z hlediska hardwaru může být dražší. Aplikace totiž sice nemusí, ale může být rozprostřena na vícero serverech. (81)
- Aby byly služby nezávislé na okolí, tak také musí mít svou databázi. To znamená, že je v rámci aplikace nutné spravovat vícero databází, což není lehký úkol. (81)

3.7 Docker

V předchozích kapitolách byly popsány typy aplikací, typy architektur a v návaznosti na praktickou část, také některé nástroje používané pro jejich samotný vývoj. Poslední problematikou, kterou je nutné v rámci vývoje aplikace řešit, je její nasazení, a s tím spojené prostředí, ve kterém aplikace poběží. U monolitických aplikací je samozřejmě již zmíněná možnost zkrátka překopírovat soubory na server. To ale obnáší řadu problémů a komplikací. Zejména fakt, že prostředí, ve kterém byla aplikace vyvíjena je zcela odlišné od produkčního serveru. To častokrát vyústuje ve velice známou situaci, kdy aplikace bezproblémově funguje na vývojářově počítači, ale na serveru ji nelze spustit. (89)

Velice moderním a hojně využívaným nástrojem, který tento problém adresuje je Docker.

Docker je open-source platforma pro vývoj, dodávání a běh aplikací. Díky Dockeru lze aplikace oddělit od zbytku infrastruktury, což elegantně řeší výše zmíněný problém. (90)

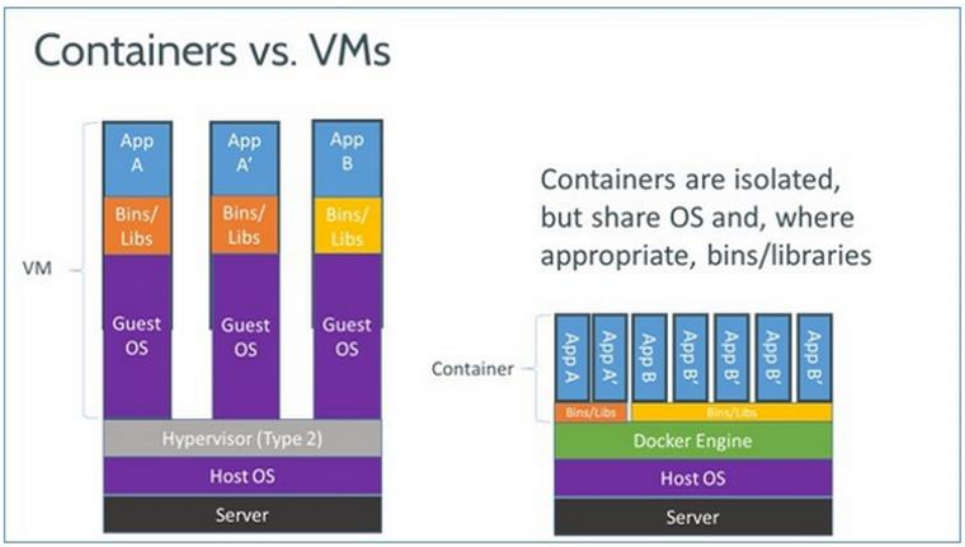
Docker platforma umožňuje:

- Vývoj aplikací a jejich podpůrných komponent s využitím tzv. kontejnerů. Kontejner je lehký, samostatný a spustitelný balíček softwaru, který obsahuje všechny komponenty nutné pro běh aplikace (90) (91)
- Výše zmíněné kontejnery pak slouží k distribuci a testování aplikace (90)

3.7.1 Princip Docker platformy

Docker tedy izoluje aplikaci i s jejím prostředím od všeho ostatního, což velice připomíná princip virtualizace. Od klasické virtualizace se však v jistých ohledech liší.

Docker umožňuje zabalit a spustit aplikaci v izolovaném prostředí nazvaném kontejner. Proto se také často v Docker mluví o kontejnerizaci a ne přímo o virtualizaci. Prvním zásadním rozdílem v Dockeru oproti klasické virtualizaci je, že Docker nepoužívá hypervisor, což je software, který vytváří a udržuje v běhu virtuální stroje. Docker namísto hypervisoru běží přímo na kernelu hostitelského stroje. Toto řešení umožňuje kontejnery izolovat, ale zároveň mezi nimi sdílet velké soubory operačního systému a rozdělit si výpočetní zdroje. (90) (92) (93)

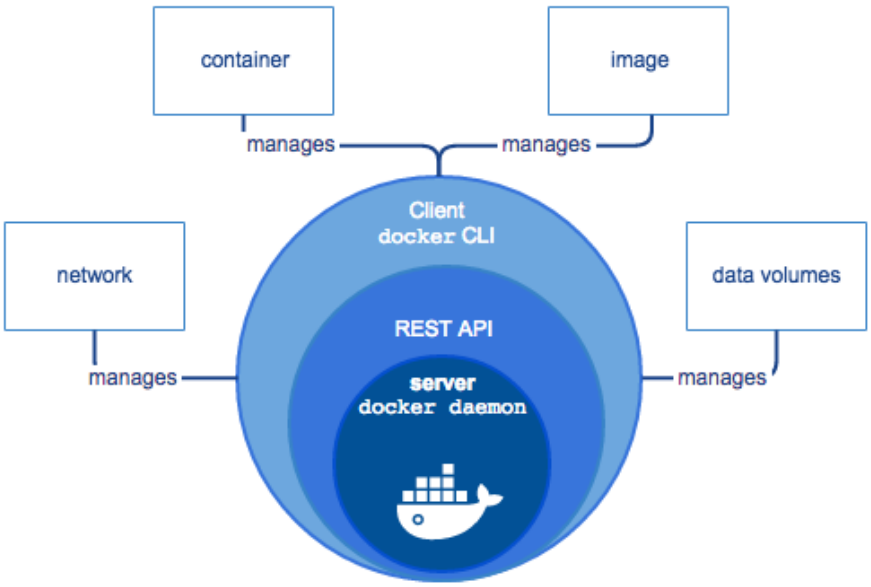


Obrázek 9 - Virtuální stroj vs Docker. Zdroj: (93)

3.7.2 Docker engine

Jádrem všeho je Docker engine, který je open source technologií pro tvorbu a kontejnerizaci aplikací. Jedná se o prostředí, ve kterém kontejnerizované aplikace běží společně se všemi jejich závislostmi. Díky tomu pak mohou kontejnerizované aplikace běžet konzistentně na jakékoliv infrastruktuře, na které je Docker engine nainstalován. Tento engine funguje na principu klient-server a obsahuje následující komponenty: (90) (94)

- Server, s dlouho běžícím daemon procesem (90)
- REST API, které umožňuje aplikacím komunikaci s daemonem (90)
- Rozhraní příkazové řádky klienta ve formě tzv. Docker CLI (příkazy určené pro Docker deamona začínají s klíčovým slovem **docker**) (90)



Obrázek 10 - Docker engine. Zdroj: (95)

3.8 Kubernetes

S příchodem kontejnerů a Dockeru vyvstala otázka, jak co nejefektivněji tyto kontejnery spravovat. Jedním ze známých a rozšířených řešení je Kubernetes, které je zejména při vývoji mikroservisových aplikací opravdu užitečným nástrojem.

Kubernetes je přenosná, rozšiřovatelná open-source platforma pro správu kontejnerizovaných aplikací a služeb. (96)

3.8.1 Funkce Kubernetes

- Kubernetes umožňuje vyvažování zátěže napříč kontejnery a také automatické škálování, které se odvíjí od aktuálního vytížení aplikace (96) (97)
- S jeho pomocí lze také orchestrovat uložiště. Je možné používat například lokální či cloudové úložiště, přičemž tato úložiště mohou být trvalá což znamená, že nezaniknou společně se zánikem kontejneru. (96)
- Kubernetes se také stará o případné restartování kontejnerů či jejich „zabití“, na základě uživatelem nastavených pravidel. (96)
- Drtivá většina aplikací potřebuje uchovávat tajemství, jako například nějaká hesla, či API klíče. Kubernetes umožňuje nastavení a správu tajemství, která tak nemusí být součástí samotné konfigurace chování jednotlivých kontejnerů. (96)

3.9 Obdobná řešení

Celá tato práce se zabývá porovnáním monolitické a mikroservisové architektury aplikací. Konkrétně se soustředí na porovnání v rámci menších webových aplikací. To znamená aplikací, které jsou vyvíjeny v malém týmu, nemají velký rozpočet, ani rozmanitou kódovou základnu.

Mikroservisy jsou dnes velice aktuálním tématem, na které lze najít řadu článků, či odbornějších prací. Zmíněné články jsou ale často pouze teoretické a jejich závěry se neopírají o nějaké skutečné testování aplikace. Na druhou stranu některé odbornější práce se zaměřují čistě na porovnání výkonu těchto architektur a nevěnují téměř žádný prostor samotnému procesu vývoje aplikace, čímž je značně omezen náhled do řešení aplikace, kterou využily pro svá měření (98) (99). Zatímco další odbornější práce se pro změnu zabývají hlavně vývojem, či využitím mikroservis v různých oblastech. (100) (101)

Tato práce se pokouší o zachycení širšího obrazu při porovnání těchto dvou architektur se zaměřením na menší webové aplikace. Pro dosažení cíle byla vyvinuta reálná webová aplikace, o které se lze dočíst v praktické části. Jedná se o aplikaci, kterou katedra informatiky skutečně využívá a má zájem o její další aktivní vývoj. (102) Podstatné části

aplikace byly vyvinuty pro každou architekturu zvlášť, aby bylo možné se při porovnání opírat o skutečné návrhové vzory, či o skutečné výsledky různých měření.

4 Vlastní práce

Vlastní práce se zabývá porovnáním mikroservisové a monolitické architektury. Za účelem porovnání byly vytvořeny dvě verze experimentální aplikace. V úvodu je tak aplikace představena. Poté byly popsány oblasti implementace, ve kterých se obě verze aplikace značně liší. Vytvořené verze aplikace byly následně podrobeny testování výkonu a vytížení hardwaru serveru. V závěru byla stanovena kritéria pro porovnání obou architektur na základě nastudování problematiky a konzultací s odborníky z praxe.

4.1 Experimentální webová aplikace

Jedním z dílčích cílů této práce je vytvoření experimentální webové aplikace. O tuto aplikaci se značně opírá následné porovnání architektur a je tak dobré uvést, o čem aplikace je a jaké má využití.

Aplikace s názvem Wolno zobrazuje na interaktivní mapě počty připojených uživatelů. V současném stavu zobrazuje pouze Provozně ekonomickou fakultu České zemědělské univerzity, ale je v plánu pokrytí rozšířit a postupně začlenit všechny fakulty. (102) Aplikace je dostupná na adrese wolno.pef.czu.cz.

Díky vizualizaci počtu připojených uživatelů je možné vidět v reálném čase obsazenost fakulty. Aplikace umožňuje podívat se na obsazenost v konkrétním čase, či v určitém časovém rozmezí, pro které dokáže přehrát průběh v čase. (102)

Aplikace mimo jiné slouží i jako demonstrace toho, co lze dokázat při propojení webových technologií a chytrých zařízení internetu věcí. Zatím je pracováno pouze s daty z přístupových bodů, ale do budoucna by bylo možné aplikaci rozšířit o vizualizaci dat z dalších zařízení, například z různých senzorů teploty, hluku, vlhkosti, a dalších.

V současném stavu se jedná o malou aplikaci, která nemá velkou uživatelskou základnu a za jejím vývojem stojí opravdu malý tým, přičemž drtivou část tvorby aplikace má na starosti autor této práce. Nicméně nelze popřít jistý potenciál růstu této aplikace, a to zejména její serverové části, kterou by bylo možné postupně rozšiřovat a využívat pro různé analýzy dat, či jiné projekty. Je tak otázkou, zdali v tomto brzkém stadiu pokračovat ve vývoji monolitické verze, či přejít rovnou od začátku na mikroservisový přístup, i když není jasné, jak moc se aplikace v budoucnu rozšíří. Jinými slovy je otázkou, jestli se vyplatí tvořit aplikaci mikroservisově, i když je a zůstane vcelku malá. Díky

této otázce bylo usouzeno, že je aplikace vhodným adeptem pro porovnání mikroservisové a monolitické architektury.

Obrázek 11 - Aplikace Wolno. Zdroj: (103)

4.2 Tvorba monolitické verze

Jako první je popsána monolitická verze aplikace, a to zejména díky předpokladu, že monolitická architektura je známá většímu množství lidí.

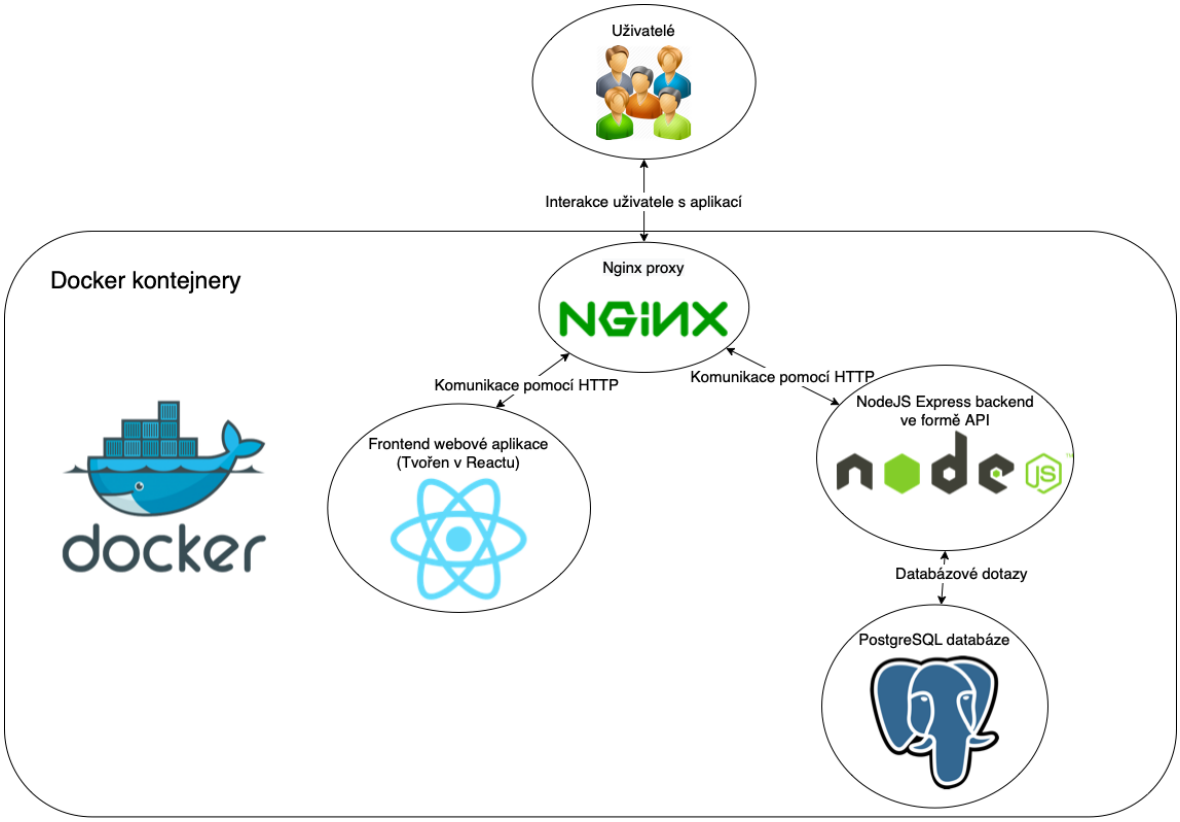
4.2.1 High level schéma architektury

V první řadě je třeba říct, že mikroservisy jsou zejména serverovou (backendovou) záležitostí, protože i mikroservisová aplikace se na venek jeví jako jeden velký celek a z pohledu klientské (frontendové) části se tak v ideálním případě vůbec nic nemění. Klientskou část (Frontend) lze samozřejmě také dělit, ale u klientské části se jedná o takzvané mikrofrontendy, které nejsou předmětem této práce. (104)

Z tohoto důvodu v práci není věnován téměř žádný prostor klientské části aplikace a je zmíněn pouze na začátku pro kompletní představení aplikace.

Na obrázku č. 12 je znázorněno high level schéma architektury monolitické verze aplikace. Celá aplikace je v Docker kontejnerech. Klientská i serverová část aplikace je schovaná za Nginx proxy, která všechny dotazy, které mají v URL /api, posílá na Node.js

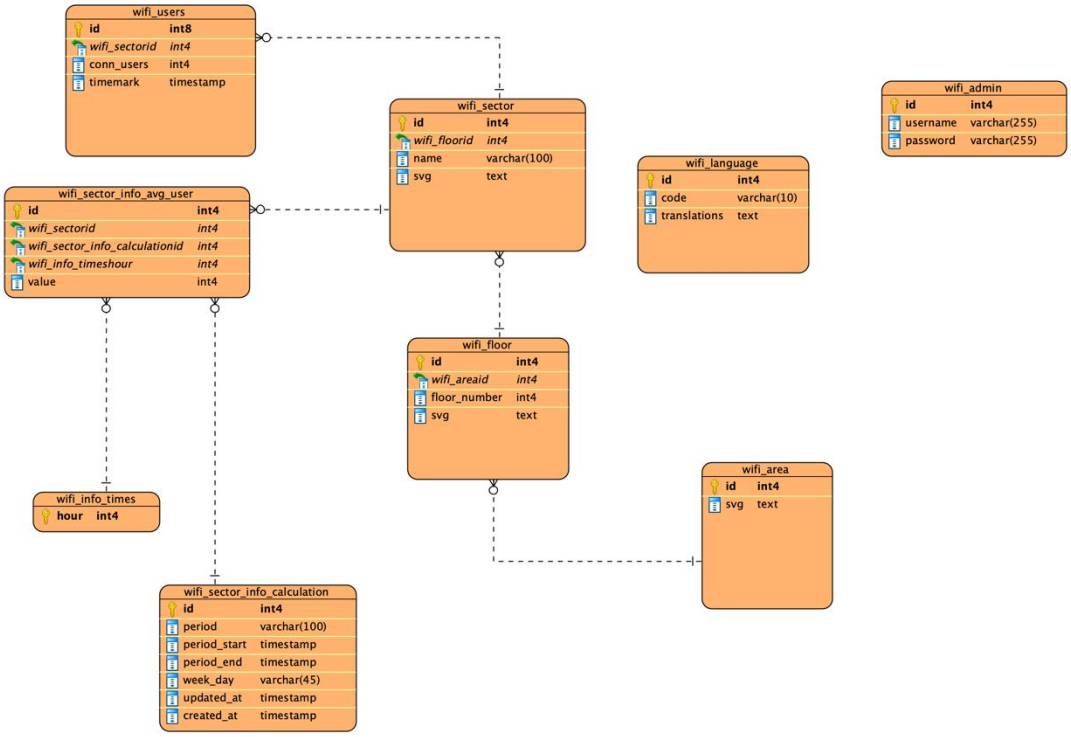
(konkrétně Express.js) backend a vše ostatní posílá na klientskou část napsanou v JavaScriptové knihovně React. Serverová část je vcelku standardní RESTful API napsané v TypeScriptu. Jako takové tedy k chodu vyžaduje prostředí Node.js a nějaký framework. Frameworků pro tvorbu serverové části aplikace je ve světě JavaScriptu a TypeScriptu opravdu hodně, přičemž mezi nejrozšířenější a nejznámější patří jednoznačně Express.js, který byl použit i pro tvorbu tohoto API. Pro databázovou vrstvu bylo zvoleno PostgreSQL. Pro doplnění je třeba říct, že pro snazší tvorbu databázových dotazů byla použita Prisma ORM, která umožňuje pracovat s databází v rámci jazyka TypeScript.



Obrázek 12 - High level schéma architektury monolitické verze aplikace. Zdroj: vlastní zpracování

4.2.2 ER Diagram

Jedná se o klasický ER diagram, který znázorňuje schéma použité v databázi pro aplikaci.



Obrázek 13 - ER diagram monolitické verze aplikace. Zdroj: vlastní zpracování

4.2.3 Přihlašování do aplikace a ověřování identity

Aplikace má svou administraci, do které je samozřejmě nezbytné přihlášení. Implementace tohoto přihlášení je v rámci diplomové práce záměrně trochu zjednodušená a odlišná od skutečnosti, nicméně princip je stejný. Je zde vyvíjeno API, a tak je přihlašování implementováno pomocí tiketů, jejichž princip je popsán v teoretické části práce, konkrétně v kapitole „Validace v RESTful API pomocí tiketů (tokenů)“.

Koncové body, které vyžadují administrátorský přístup obsahují middleware, která ověří, jestli uživatel, který chce provést danou akci, má validní token a je tak oprávněn k provedení této akce.

Middleware je součástí složky „commons“, která je zmíněna v následující kapitole.

4.2.4 Sdílení kódu v aplikaci

Sdílení kódu v rámci monolitické verze je řešeno vcelku jednoduše. Pro sdílený kód byla vytvořena složka „commons“, kam byl vložen veškerý sdílený kód napříč aplikací. Jedná se o různé middlewary, například již zmíněnou middleware ověřující tiket uživatele,

nebo middleware pro ověření validity dat požadavku. Dále jsou v této složce třídy chybových hlášek, či různé pomocné funkce.

4.2.5 Nasazení vývojové a produkční verze aplikace

Monolitickou verzi aplikace lze samozřejmě vyvíjet a následně nasadit bez Dockeru. V dnešní době je však vcelku moderní Docker využívat, a to bez ohledu na velikost aplikace. Je to z důvodů zmíněných v teoretické části, zejména v kapitole „Docker“. Nicméně to samozřejmě obnáší nutnou znalost Dockeru a pro optimální nastavení kontejnerů je potřeba si tuto technologii dobře prostudovat.

Co se vývoje týče, tak má vývojář možnost zvolit variantu, kdy aplikaci vyvíjí bez Dockeru, a Docker tak využije pouze pro produkční verzi. Tento fakt je zmíněn s ohledem na mikroservisovou architekturu, kde toto není dost dobře možné a bude o tom ještě psáno. Avšak nastavit Docker i pro vývojovou verzi není příliš náročné, když je vývojář schopen nastavit kontejnery pro produkční verzi. Je jen nutné uvědomit si, jaké prostředí aplikace pro svůj běh potřebuje a jaké kroky jsou potřeba k jejímu spuštění.

Docker kontejnery lze spustit buďto pouze pomocí Docker příkazů, ale to se příliš nedoporučuje. Je spíše upřednostňován Docker Compose, který slouží pro definování a spouštění aplikace, kde je vícero kontejnerů a zapisuje se v podobě YAML souboru. (105) Pro spuštění Docker kontejnerů je totiž nutné nastavit řadu věcí, například jméno kontejneru, síť, do které bude kontejner připojen, port, ze kterého bude kontejner přístupný, či kontext, ze kterého má být převzat Dockerfile, ve kterém jsou uvedeny kroky pro spuštění kontejneru. Docker Compose tedy slouží jako jakýsi zápis kroků, které je potřeba udělat pro spuštění celé aplikace.

Níže je uveden příklad části Docker Compose souboru

```
api:
  container_name: wifi-monolith-api
  restart: always
  build:
    context: ./wifi-api
  ports:
    - "5001:5000"
```

Kód 1 - úryvek Docker-compose souboru. Zdroj: vlastní zpracování

A k němu příslušný Docker soubor

```
FROM node:16.13-alpine
```

```
WORKDIR "/usr/src/app"
COPY ./package.json ./
COPY ./package-lock.json ./
COPY ./tsconfig.json ./
RUN npm install
COPY . .
RUN npm run build

FROM node:16.13-alpine
WORKDIR "/usr/src/app"
COPY ./package.json ./
RUN npm install --only=production
COPY --from=0 /usr/src/app/dist /usr/src/app
COPY --from=0 /usr/src/app/prisma/schema.prisma
/usr/src/app/prisma/schema.prisma

CMD ["npm","run","prod"]
```

Kód 2 - Docker soubor. Zdroj: vlastní zpracování

Je jasné, že Docker je mocný nástroj, ale pro vývojáře představuje určitou nadstavbu, se kterou je nutno se naučit pracovat. Při nastavování Dockeru je nezbytné uvědomit si všechny kroky potřebné pro nasazení vývojové, či produkční verze. Avšak i bez Dockeru je potřeba řadu těchto kroků provést. V Dockeru je akorát zásadní rozdíl v tom, že je nutné všechny kroky sepsat v rámci Docker souborů. Následně je tato komplexita navíc vykoupena tím, že nasazení produkční verze je více bezproblémové v porovnání s nasazením aplikace bez Dockeru.

4.3 Tvorba mikroservisové verze

4.3.1 High level schéma architektury

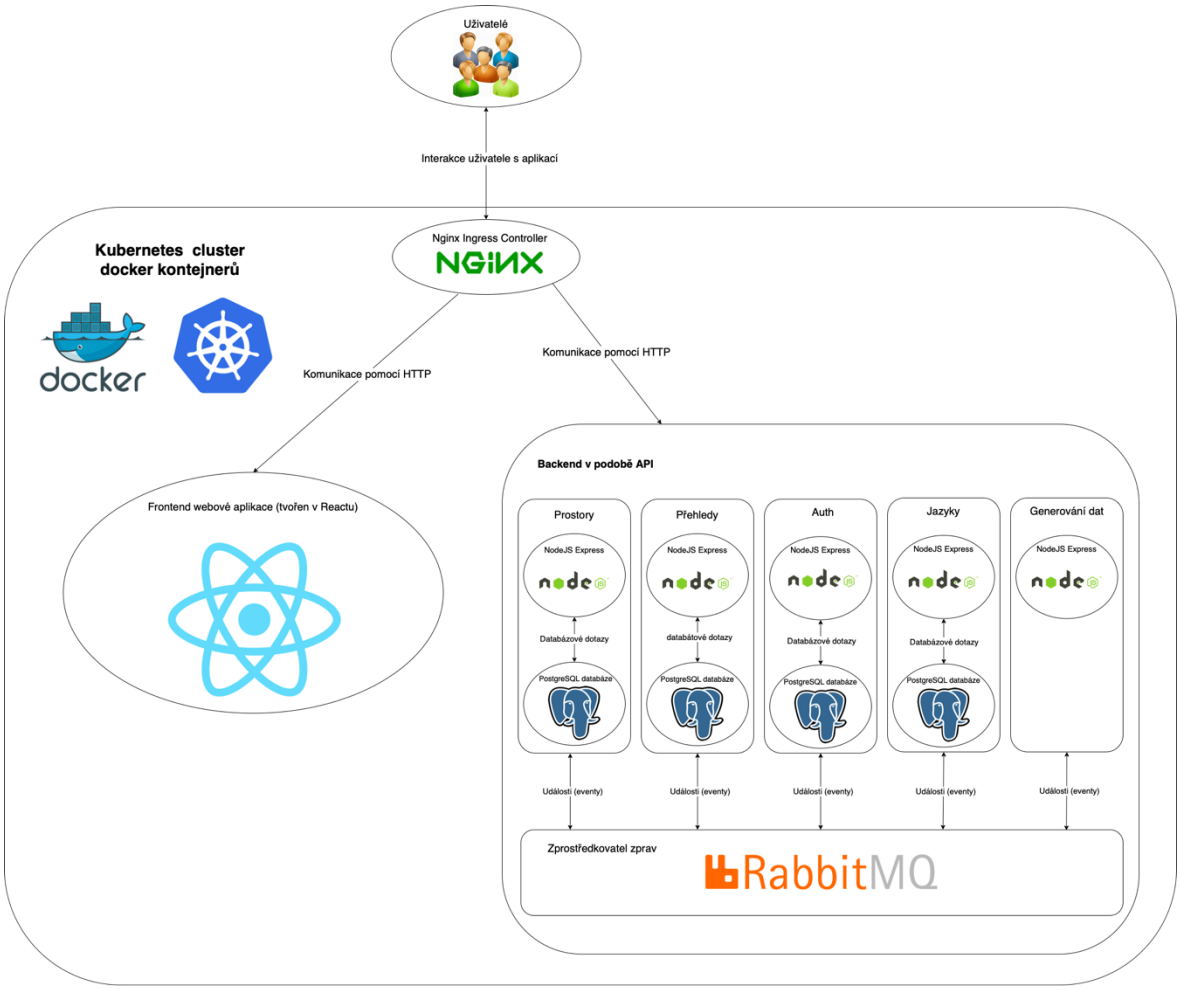
Na obrázku č. 14 je možné vidět high level schéma mikroservisové verze aplikace. Jak již bylo řečeno, klientské části aplikace se změny v architektuře vůbec netýkají a je tak stále stejná. Velký rozdíl je možné vidět v rámci serverové části aplikace a také ve změně proxy, která rozesílá požadavky od uživatelů na správná místa. Už ze samotného obrázku je jasné, že mikroservisová architektura značně přidává na komplexitě samotného vývoje aplikace. V kapitole „Mikroservisová architektura“ bylo řečeno, že jde vlastně o menší samostatné části většího celku. Aby mohl být nějaký kus aplikace nezávislý na okolí, tak je vhodné, aby obsahoval vše, co ke svému chodu potřebuje. Toto se týká i dat, proto mají všechny služby, které potřebují uchovávat data, svou vlastní databázi.

Z toho pramení jistá duplicita dat a nutnost zajištění správnosti dat napříč jednotlivými službami.

V rámci teoretické části práce, konkrétně v kapitole „Způsoby komunikace“ bylo řečeno, že jednotlivé služby by spolu neměly komunikovat napřímo a ideálně by ani vzájemně neměly vědět o své existenci. Určitý způsob komunikace je však samozřejmě nezbytný. Je řada způsobů, jak komunikace docílit. V rámci aplikace byl použit zprostředkovatel zpráv pro asynchronní komunikaci služeb RabbitMQ.

Další změnou je samotná proxy. Aplikace se musí na venek jevit jako jeden celek. Za tímto účelem byl zvolen Nginx Ingress Controller. V rámci něj se nastaví host, například wifi.com a následně se již uvádí jen jednotlivé tvary URL, a kam se má daný dotaz v rámci serverové části aplikace směrovat. Mimo to, ale také vyvažuje zátěž. Jednotlivé mikroservisy totiž zpravidla nemají pouze jednu repliku dané služby. Nginx Ingress Controller toho umí využít a rozkládá zátěž mezi jednotlivé repliky služeb tak, aby repliky byly co nejlépe vytíženy. (106) (107)

Poslední zásadní změnou, která značně přidává na komplexitě, je využití Kubernetes, o kterých již byla řeč v kapitole „Kubernetes“.



Obrázek 14 - High level schéma mikroservisové verze aplikace. Zdroj: vlastní zpracování

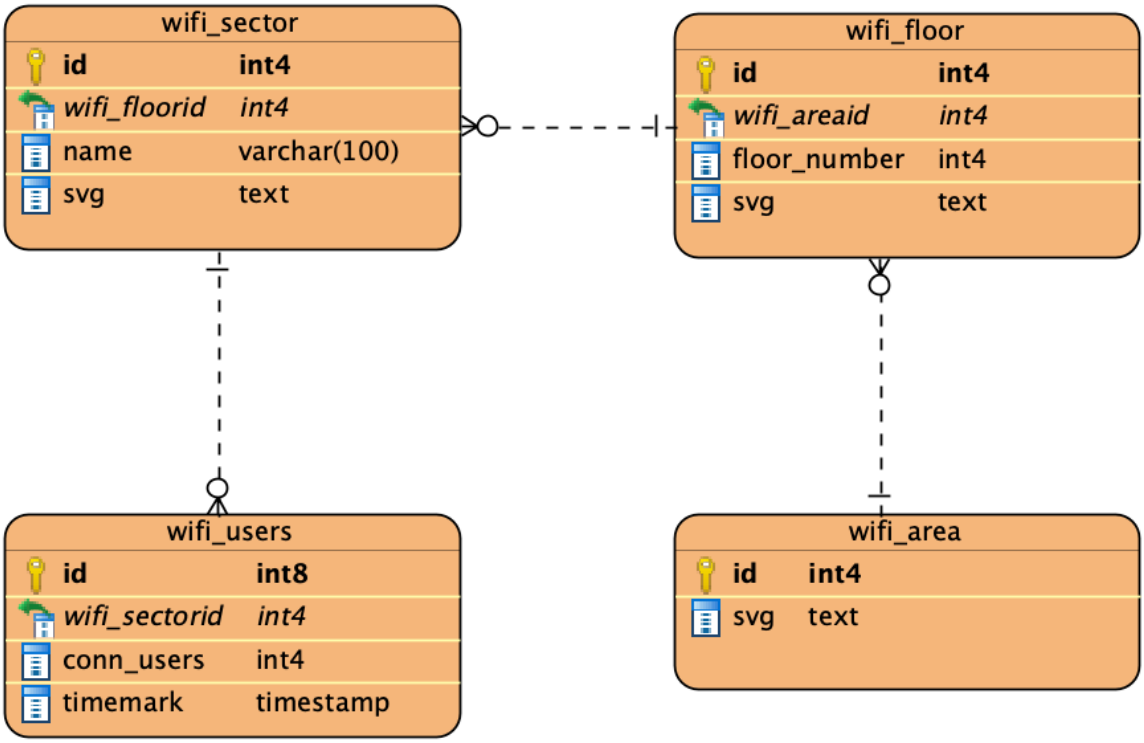
4.3.2 Jednotlivé služby a jejich ER diagramy

V této kapitole se lze dočíst o tom, do jakých částí byla aplikace v rámci mikroservisové architektury rozdělena. Jednotlivé celky, nebo také služby byly rozděleny dle doporučení odborných článků a knih tak, aby byly co nejvíce nezávislé, a aby byla vyžadována co nejmenší nutná komunikace mezi nimi. (82) (83) Je důležité zdůraznit, že již jen samotný návrh jednotlivých služeb není snadný a je potřeba se nad ním opravdu zamyslet. U menších aplikací je samozřejmě snazší zohlednit všechny funkcionality aplikace a veškerou nutnou komunikaci, nicméně i tak jde o problematiku, kterou v rámci monolitické verze nebylo nutné příliš řešit. Pro představu je dobré uvést, jak moc navrhování služeb prodloužilo vývoj. V této aplikaci návrh trval 4 dny. Dobu návrhu samozřejmě nelze nijak zobecnit, protože každá aplikace je zkrátka jiná. Na uvedeném časovém údaji je zajímavé hlavně to, že nejde o zanedbatelnou časovou dotaci.

V časové dotaci je nutné zohlednit i fakt, že autor práce neměl před tímto projektem téměř žádné zkušenosti s mikroservisovou architekturou, ale v současné době obecně není příliš mnoho programátorů, kteří by s mikroservisami měli bohaté zkušenosti. Na druhou stranu navrhování služeb zkrátila skutečnost, že autor práce aplikaci dobře zná a ví, co má aplikace dělat. Navíc i její monolitická verze byla vyvinuta jako první, což ještě více usnadnilo náhled do logiky aplikace a značně to v této fázi pomohlo.

4.3.2.1 Prostory

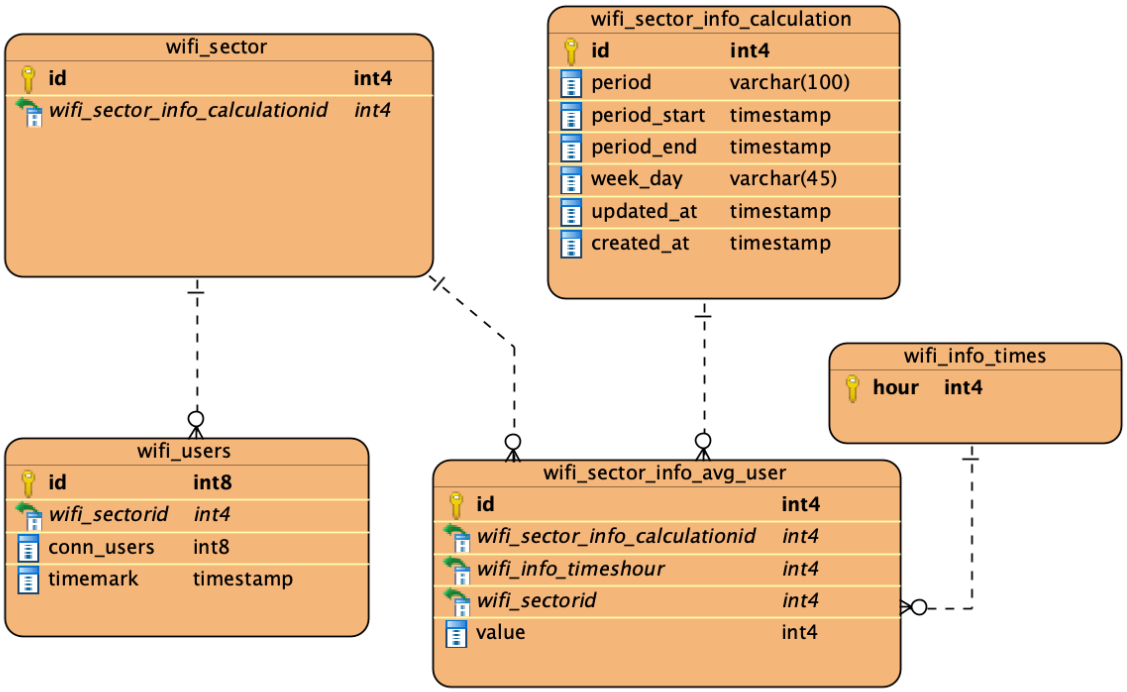
Služba prostor obsahuje vše potřebné pro získání dat o jednotlivých patrech v budovách školy.



Obrázek 15 - ER diagram pro službu "Prostory". Zdroj: vlastní zpracování

4.3.2.2 Přehledy

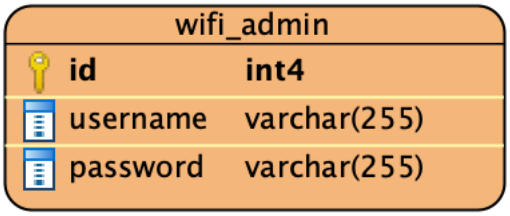
Přehledy obsahují vše potřebné k vytvoření a získání průměrné vytíženosti jednotlivých sektorů v budovách školy.



Obrázek 16 - ER diagram pro službu "Přehledy". Zdroj: vlastní zpracování

4.3.2.3 Auth

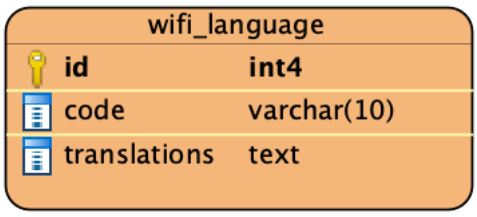
Služba auth obsahuje vše potřebné k přihlášení do administrace aplikace a vytvoření ověřovacího tiketu, kterým se následně přihlášený uživatel prokazuje.



Obrázek 17 - ER diagram pro službu "Auth". Zdroj: vlastní zpracování

4.3.2.4 Jazyky

Služba jazyky obsahuje vše potřebné k získání textace aplikace.



Obrázek 18 - ER diagram pro službu "Jazyk". Zdroj: vlastní zpracování

4.3.2.5 Generování dat

Tato služba je v rámci porovnání architektur značně upravena pro potřeby diplomové práce. Jejím účelem je v daných časových intervalech generovat uměle vytvořená data, která ve skutečné aplikaci přichází z přístupových bodů na fakultě. Tento přístup byl zvolen proto, aby následná měření výkonu aplikace byla v domácích podmínkách autora práce co nejvěrnější skutečnému produkčnímu prostředí.

Služba poté co vygeneruje data, pošle událost do RabbitMQ, která v sobě obnáší vygenerovaná data. RabbitMQ následně rozešle tuto událost do služeb, které tomuto typu události naslouchají.

4.3.3 Komunikace mezi službami

Komunikace mezi službami je jednou z klíčových částí mikroservisové architektury. V kapitole „3.6.1 Způsoby komunikace“ bylo popsáno, jak taková komunikace teoreticky probíhá, nicméně pro lepší pochopení a porovnání komplexity vývoje, je dobré uvést praktický příklad přímo z aplikace.

Z předchozí kapitoly je jasné, že několik služeb potřebuje ukládat data o počtu připojených uživatelů v rámci sektorů.

Služby by spolu neměly komunikovat napřímo, aby od sebe byly co nejvíce oddělené. (108). To znamená, že služba, která generuje data o připojených uživatelů by neměla napřímo posílat tato data do ostatních služeb, které je potřebují.

Je tak vhodné využít nějakého zprostředkovatele zpráv. Pro tyto účely existuje řada možností, přičemž každý možný nástroj řeší problematiku trochu jinak a již jen o samotné volbě toho správného nástroje lze napsat samostatnou diplomovou práci. (109)

V této aplikaci byl tedy použit RabbitMQ, od čehož se odvíjí nutné kroky pro zprovoznění, které vychází přímo z oficiální dokumentace.

Úkolem je vytvořit komunikační vzor, kde služba pro generování dat vyšle zprávu (událost) a RabbitMQ ji pak rozešle mezi všechny služby, které této události naslouchají. Komunikační vzor, kdy je jedna zpráva (událost) rozeslána mezi mnoho služeb, se nazývá Publish/Subscribe, neboli česky publikovat/odebírat. (110)

V první řadě je třeba, aby všechny služby byly připojeny k RabbitMQ. To je provedeno pomocí níže uvedeného kódu. Třída EventBusWrapper obsahuje kód nutný pro připojení k RabbitMQ. Asynchronní metoda connect připojí službu k RabbitMQ, a privátní proměnná channel pak umožňuje přístup k tomuto připojení.

```
import amqp, { Channel } from 'amqplib';

class EventBusWrapper {
  private _channel?: Channel;

  get channel() {
    if (!this._channel) throw new Error('Cannot access RabbitMQ channel');

    return this._channel;
  }

  async connect() {
    const connection = await amqp.connect(process.env.RABBITMQ_URL!);
    this._channel = await connection.createChannel();
  }
}

export const eventBusWrapper = new EventBusWrapper();
```

Kód 3 - třída pro připojení k RabbitMQ. Zdroj: vlastní zpracování

Samotné připojení je provedeno následovně.

```
await eventBusWrapper.connect();
```

Kód 4 - připojení k RabbitMQ. Zdroj: vlastní zpracování

Pro zveřejnění dat je potřeba udělat ještě dva kroky. V první řadě se musí ještě připojit k tzv. Výměně (Exchange). V RabbitMQ vydavatel (Publisher) zpráv nezveřejňuje zprávy napřímo, ale přes již zmíněnou „výměnu“. Výměna dělá pouze to, že přijímá na jedné straně zprávy a na druhé straně je posílá do RabbitMQ front. Tato výměna potřebuje jméno, typ

a případně další nastavení. (110) Jméno je zde obsaženo v rámci Exchange.WifiUsers, typ je „direct“, kterým je řečeno, že má být zpráva poslána pouze do front, které ji chtějí. (111)

```
    await    eventBusWrapper.channel.assertExchange(Exchange.WifiUsers,
'direct', { durable: false });
```

Kód 5 - Připojení k RabbitMQ výměně. Zdroj: vlastní zpracování

V neposlední řadě je nutné vytvořená data publikovat pro ostatní služby. Pro lepší znuvupoužitelnost byla pro tyto účely nejdříve vytvořena třída, která umožňuje publikovat události.

```
    import { Exchange } from './exchange';
    import { RoutingKey } from './routing-key';
    import { Channel } from 'amqplib';

    interface Event {
      exchange: Exchange;
      routingKey: RoutingKey;
      data: any;
    }

    export abstract class Publisher<T extends Event> {
      abstract exchange: T['exchange'];
      abstract routingKey: T['routingKey'];
      constructor(protected channel: Channel) {}

      publish(data: T['data']) {
        const stringifyData = JSON.stringify(data, (key, value) =>
          typeof value === 'bigint' ? value.toString() + 'n' : value
        );

        console.log(`Event    published,    exchange:${this.exchange},
routingKey: ${this.routingKey} `);

        this.channel.publish(this.exchange,           this.routingKey,
Buffer.from(stringifyData));
      }
    }
```

Kód 6 - třída pro publikování zpráv (událostí). Zdroj: vlastní zpracování

Konkrétní publikování zpráv je pak implementováno tak, že je pro každý publisher vytvořena speciální třída, která dědí z výše uvedené základní třídy pro publikování. Tato třída v sobě nese navíc jméno výměny a název směrovacího klíče. Výměna již byla popsána, a směrovací klíč slouží jako identifikátor, kterým RabbitMQ pozná, kdo o dané

zprávy stojí. Zprávy tak obdrží pouze ti odběratelé, kteří používají pro směrovací klíč stejnou hodnotu. (111)

```
import { Publisher,Exchange ,RoutingKey,NewUsersEvent } from '@skepter/wifi-micro-common'

export class NewUsersPublisher extends Publisher<NewUsersEvent> {
  exchange: Exchange.WifiUsers = Exchange.WifiUsers;
  routingKey: RoutingKey.NewWifiUsers = RoutingKey.NewWifiUsers;
}
```

Kód 7 - publikace zprávy (události) "new users" do RabbitMQ. Zdroj: vlastní zpracování

Vytvořenou třídu pro publikování pak už jen stačí zavolat v kódu následovně.

```
new NewUsersPublisher(eventBusWrapper.channel).publish(generatedData);
```

Kód 8 - volání metody pro publikaci zprávy (události) "new users". Zdroj: vlastní zpracování

Nyní je ještě potřeba vyřešit odběr těchto zpráv. Pro odběr je opět vytvořena znovupoužitelná základní třída. Ta obsahuje dvě podstatné asynchronní metody. První je consume, která obsahuje logiku pro odběr zpráv. Je v ní jedno důležité nastavení a sice „noAck: false“. RabbitMQ totiž v základu pouze rozesílá zprávy a nestará se, zda byly zprávy skutečně doručeny. Je to sice rychlejší způsob, nicméně některé zprávy se tak mohou ztratit. To je díky výše zmíněnému nastavení vypnuto, a je tak vyžadováno potvrzení, jinak je zpráva následně odeslána znovu. (112)

Druhou důležitou metodou je parseMessage. Ta je potřeba, protože zprávy chodí v JSON formátu, a pro jejich přečtení je tak nejdříve potřeba je přetvořit z JSON formátu na JavaScript objekty.

```
import { Exchange } from './exchange';
import { RoutingKey } from './routing-key';
import { Channel, ConsumeMessage } from 'amqplib';

interface Event {
  exchange: Exchange;
  routingKey: RoutingKey;
  data: any;
}

export abstract class Listener<T extends Event> {
  abstract exchange: T['exchange'];
  abstract routingKey: T['routingKey'];
```

```
        abstract onMessage(data: T['data'], channel: Channel, msg:
ConsumeMessage): void;
        abstract queueName: string;

        constructor(protected channel: Channel) {}

        async consume() {
          await this.channel.bindQueue(this.queueName, this.exchange,
this.routingKey);
          await this.channel.consume(
            this.queueName,
            (msg) => {
              const parsedData = this.parseMessage(msg);
              if (!msg) throw new Error('Null message in consume!');
              console.log(`Message received, queue: ${this.queueName},
exchange: ${this.exchange}, routingKey: ${this.routingKey}`)
              this.onMessage(parsedData, this.channel, msg);
            },
            { noAck: false }
          );
        }

        private parseMessage(msg: ConsumeMessage | null) {
          if (!msg) return;
          return JSON.parse(msg.content.toString(), (key, value) => {
            if (typeof value === 'string' && /^\d+n$/.test(value)) {
              return BigInt(value.substring(0, value.length - 1));
            }
            return value;
          });
        }
      }
```

Kód 9 - znovupoužitelná třída "Listener". Zdroj: vlastní zpracování

Nyní již zbývá jen začít odebírat zprávy. K tomu je vytvořena třída NewUsersListener, která dědí z třídy Listener, a navíc do ní přidává jméno výměny, směrovací klíč a jméno fronty. Fronta je zde důležitá. Protože zatímco díky výměně jsou zprávy rozesílány všem, kteří ji odebírají, tak v rámci fronty je zpráva poslána pouze jednomu členovi fronty. Zde je totiž potřeba si uvědomit, že některé služby budou mít vícero replik, to znamená vícero členů fronty, přičemž zprávu o počtu připojených uživatelů je potřeba zpracovat pouze jednou.

```
      import { Listener, NewUsersEvent, Exchange, RoutingKey } from
'@skepter/wifi-micro-common';
```

```
import { queueName } from './queue-name';
import { Channel, ConsumeMessage } from 'amqplib';
import prisma from '../../db';

export class NewUsersListener extends Listener<NewUsersEvent> {
  exchange: Exchange.WifiUsers = Exchange.WifiUsers;
  routingKey: RoutingKey.NewWifiUsers = RoutingKey.NewWifiUsers;
  queueName = queueName;
  async onMessage(data: NewUsersEvent['data'], channel: Channel,
msg: ConsumeMessage) {
    // Logika, která se má provést při obdržení zprávy
    channel.ack(msg);
  }
}
```

Kód 10 - odebírání zprávy (události) "new users". Zdroj: vlastní zpracování

4.3.4 Přihlašování do aplikace a ověřování identity

Princip je stejný jako u monolitické verze. Je zde ale velký problém. Jednotlivé služby vystavují různé koncové body, přičemž některé z nich mají být přístupné pouze přihlášeným uživatelům. Jak ale řešit přihlašování a ověřování identity, v rámci architektury, ve které je řada nezávislých logických celků a například služba „Prostory“, vůbec nemá ponětí o existenci služby „Auth“?

Zabezpečení mikroservisových aplikací je opravdu zásadní téma, které přidává na komplexitě tvorby tohoto typu aplikací. Existuje pro něj řada možností, přičemž žádná není jednoznačně správná a každá má svá úskalí.

Po nastudování problematiky byla zjištěna tři nejčastější řešení:

1. Jedna globální služba, která řeší přihlašování, ale i ověřování identity, přičemž ostatní služby se u každého dotazu musí dotázat na Auth službu, aby ověřila, jestli má uživatel přístup k požadovaným datům. Výhodou je, že zde není žádná duplicita kódu. Na druhou stranu je celá aplikace zpomalena, protože požadavky nejdřív musejí projít přes globální Auth službu, což přidává na latenci. Navíc jsou zde všechny služby závislé na Auth službě. To znamená, že pokud Auth spadne, přestane fungovat vše, co ke své funkci potřebuje ověření identity. (113) (114) (115)
2. Mikroservisová architektura často používá API bránu, která posílá požadavky dle nastavených kritérií do patřičných služeb. Tuto bránu lze využít i pro přihlašování či ověřování identity a brána tak pošle do patřičných míst pouze validní požadavky. Mezi hlavní výhody patří to, že jde o globální řešení, které nevyžaduje duplicitu kódu

a nepřidává na latenci. Jedná se o vcelku čisté řešení, které ale naráží na problém ve chvíli, kdy je potřeba mít v různých službách různé uživatelské role. (113) (114) (116)

3. Globální služba, která řeší pouze přihlašování a ověřování přístupu k datům si řeší ostatní služby samy. Hlavní výhodou řešení je, že je nejvíce nezávislé na svém okolí, nepřidává na latenci a také v případě nutnosti umožňuje různé nastavení uživatelských rolí v každé službě zvlášť. Nevýhodou je samozřejmě nutnost duplikování kódu, a pokud nejsou všechny služby napsány ve stejném jazyce, je nutné přepisovat logiku ověření identity do patřičných jazyků (113) (114)

V aplikaci bylo zvoleno třetí řešení, a to zejména z důvodu, že toto řešení zachovává největší nezávislost jednotlivých služeb.

Bylo tak nutné se vypořádat s hlavní nevýhodou tohoto řešení, a sice s nutností kopírování kódu pro ověření identity do každé služby. Tento problém je adresován v následující kapitole.

4.3.5 Sdílení kódu

V mikroservisové architektuře se doporučuje spíše kód duplikovat a sdílet tak naprosté minimum. Přesněji a lépe řečeno, v rámci služeb cílit na co největší znuvupoužitelnost kódu, ale tolik ji neřešit napříč službami. (83) Toto doporučení může znít opravdu zvláštně a v očích řady programátorů to může zprvopočátku vyvolávat pochybnosti.

Stojí za ním však opravdu dobré důvody. Prvním je fakt, že jednotlivé služby nutně nemusí být napsány v jednom jazyce a znovupoužití například nějakých pomocných funkcí tak stejně není dost dobře možné. Druhým a závažnějším důvodem je ten, že čím více je sdíleného kódu, tím více jsou služby vzájemně provázány. To je problém zejména na větších projektech, kdy bývá běžné, že každou službu vyvíjí jiný tým. V takovém případě, pokud některá služba potřebuje provést změny ve sdíleném kódu, tak na to musí zareagovat i ostatní služby, což se může velice snadno vymknout kontrole. (83)

Otázkou tedy je, co by měly služby sdílet a jak? Nad touto problematikou je nutno ve srovnání s monolitickým přístupem, mnohem více přemýšlet a opět neexistuje jedno univerzální a ideální řešení.

Nejzásadnější, co je nutné sdílet, je struktura zpráv, které mohou přes RabbitMQ služby obdržet. V případě této konkrétní aplikace je to struktura zprávy, která v sobě nese nová data o připojených uživatelích.

Bylo nalezeno několik způsobů řešení:

1. Definovat tuto strukturu pomocí schéma. Konkrétně například pomocí JSON schéma. (117) Hlavní výhodou řešení je fakt, že není závislé na jednom konkrétním jazyce. Nevýhodou je však to, že je nutno použít nějakou překládací vrstvu.
2. Kopírování kódu do všech služeb. Manuální kopírování je samozřejmě nejsnazším naivním řešením. To ale s sebou nese několik problémů. Tím nejzásadnějším je, že při kopírování lze udělat chybu, přičemž provedení nějaké změny vyžaduje promítnutí této změny do všech služeb. Bylo by samozřejmě možné kopírování zautomatizovat a změny rozkopírovat pomocí nějakého skriptu, i to však s sebou nese obrovské problémy. Za prvé je specifické pro jazyk, ve kterém je onen kód napsán. Za druhé, což je také největším problémem, změna sdíleného kódu se promítne do všech služeb, a ty na ně následně musí reagovat.
3. Vytvořit sdílenou knihovnu, a tu pak následně do jednotlivých služeb instalovat v podobě závislosti. Sdílená knihovna má tu nevýhodu, že je specifická pro daný jazyk. To znamená, že pokud bude napsaná v TypeScriptu, tak ji lze použít pouze v službách, které jsou rovněž napsané v TypeScriptu. (118) A pokud jsou některé služby napsány v jiném jazyce, je třeba přepsat sdílenou knihovnu pro daný jazyk a udržovat tak větší množství sdílených knihoven. Výhodou tohoto řešení je to, že konkrétně TypeScript je typově vcelku silným jazykem a definice struktury zpráv pomocí TypeScriptu je tak o něco lepší než pomocí JSONu. Navíc samozřejmě není nutné používat nějakou překládací vrstvu, protože knihovna i služba je napsána v jednom jazyce. Ve spojení s knihovnou je také nutné zmínit zásadní důvod, proč je vlastně knihovna lepší než jednoduché kopírování kódu do každé služby zvlášť. Výhodou je, že změna v kódu knihovny vyžaduje vydání nové verze knihovny, přičemž ta stará je stále přístupná. Díky tomu mohou změny sdíleného kódu adaptovat pouze služby, které tyto změny chtějí a potřebují.

V aplikaci byla zvolena třetí možnost, a to proto, že jsou v současnosti všechny služby napsané v jazyce TypeScript. Dále také proto, že jde o menší aplikaci, kterou vyvíjí malý tým, a je tak v porovnání s velkými projekty snazší udržet přehled o tom, co je a co není

sdíleno, a při případné nutnosti změny kódu není tak náročné, aby měl o změnách přehled celý tým. V neposlední řadě byl tento přístup zvolen kvůli dříve vybranému způsobu zabezpečení aplikace, kdy přihlášení obstarává Auth služba, ale ověření identity si řeší služby samostatně.

Kód obstarávající tuto logiku tak může být přesunut do sdílené knihovny. Služby následně mohou použít onen sdílený kus kódu, ale v případě nutnosti si mohou implementovat vlastní řešení naplňující potřeby dané služby.

Sdíleného kódu je samozřejmě trochu víc, například middleware pro validaci dat v požadavcích. To ale není nezbytné a je třeba opravdu promyslet, co bude sdíleno a volit pouze logiku, u které se nepředpokládá, že se bude často měnit. (118) (83)

4.3.6 Nasazení vývojové a produkční verze aplikace

Nasazení a vývoj mikroservisové verze aplikace je složitější v porovnání s monolitickou verzí, a to i když je v monolitické verzi použit Docker. V mikroservisové verzi aplikace je totiž Docker již nutností a k němu je ještě opravdu vhodné použít Kubernetes.

V samotném vývoji již není možné vyvíjet lokálně bez Dockeru či Kubernetes, protože jednotlivé služby pro svůj chod potřebují databáze, také potřebují funkční RabbitMQ a Ingress Nginx Controller, což by bylo značně nepraktické vše spouštět lokálně.

Ke spuštění jsou potřeba opět Docker soubory, ale již zde není použit Docker Compose. Namísto něj přibyly Kubernetes YAML soubory, které jako Docker Compose umožňují také nastavit jméno či port, na kterém daná služba bude, ale navíc je zde například kolik má být spuštěno replik dané služby. Pro každou službu je ideálně třeba mít jeden takový soubor s nastavením. Pro představu v této aplikaci je 20 konfiguračních souborů. Do určité míry jsou si tyto soubory samozřejmě podobné a je tak možné řadu věcí kopírovat a měnit pouze detaily, nicméně i tak to přináší větší náročnost při nastavení a vícero míst, kde se může vyskytnout chyba.

Další komplexitu přináší otázka, jak těchto 20 souborů spustit? Samozřejmě lze aplikovat jednotlivé konfigurační soubory jeden po druhém, to však není příliš ideální. Ke Kubernetes je tak třeba využít ještě jeden nástroj, který se jmenuje Skaffold. Skaffold umožňuje automatizaci úkonů pro nasazení aplikace a lze tak celou aplikaci spustit pomocí jednoho příkazu (119). Se Skaffoldem však přijdou další problémy, například nutnost

stanovit pořadí, ve kterém se mají jednotlivé služby spustit. RabbitMQ totiž musí fungovat dříve než ostatní služby, aby se k němu mohly při spuštění připojit. Dále jednotlivé databáze musí fungovat dříve než aplikační služby. Nejde o snadné nastavení a dle zkušeností z vývoje aplikace často není snadné najít správný zdroj informací, kde by bylo popsáno, jak tyto záležitosti správně nastavit.

Z výše zmíněného je jasné, že mikroservisovou verzi je v porovnání s monolitickou verzí náročnější spustit a následně nasadit do produkce. Lze v tomto nastavení udělat ještě více chyb, než v monolitické verzi a Kubernetes dokumentace dle názoru autora práce není špatná, ale má své mezery a v některých částech obsahuje zastaralé informace, které nejsou aplikovatelné v nových verzích Kubernetes. Vývojář je tak občas odkázán pouze na různá fóra, kde také není snadné najít aplikovatelnou odpověď. To vše vyúsťuje ve fakt, že nastavení zabírá nemalou část času vývoje a stálo by za zváženou na toto využít nějakého odborníka, kterých není mnoho a může se to značně podepsat na konečné ceně aplikace, což je zejména na menších aplikacích s malým rozpočtem, těžko obhajitelné.

4.4 Kritéria porovnání architektur

Na základě studia odborné literatury a konzultací s odborníky z praxe byla pro porovnání mikroservisové a monolitické architektury vybrána následující kritéria:

- Časová náročnost vývoje
- Výkon aplikace
    - Průměrná doba odezvy na dotazy v milisekundách
    - Průměrné množství obdržených v KB/sec
    - Průměrné množství poslaných v KB/sec
    - Datová propustnost v req/sec
- Vytížení serveru
    - Vytížení procesoru v procentech
    - Vytížení paměti v MB
- Počet použitých technologií
- Počty řádků kódu
    - Celkový počet řádků kódu
    - Počet řádků kódu v konfiguračních souborech

- Počet řádků kódu sloužících k implementaci komunikace mezi mikroservisami

4.5 Časová náročnost vývoje

Následující tabulky č. 1 a č. 2 zobrazují, jak je časově náročné pro jednoho vývojáře vyvinout jednotlivé verze aplikace. Čas je uváděn v jednotkách „man-day“, které vyjadřují pracovní čas jedné osoby odpovídající jednomu pracovnímu dni. Pracovním dnem je zde myšleno 8 hodin. (120)

4.5.1 Monolitická verze

Činnost	**Délka trvání v man-days**
Volba technologií pro vývoj	1
Návrh databáze aplikace	1
Čas strávený tvorbou vývojové a produkční verze Docker kontejnerů	1
Vývoj aplikace	10
Nasazení aplikace	1
Celkem	**14**

Tabulka 1 - Časový průběh tvorby monolitické verze aplikace. Zdroj: vlastní zpracování

4.5.2 Mikroservisová verze

Vývoj této verze značně prodloužil návrh služeb a pak také nastavení Kubernetes clusteru. Zejména čas pro nastavení Kubernetes clusteru je třeba brát trochu s rezervou s ohledem na zkušenosti autora práce, nicméně během vývoje se vyskytly problémy například díky tomu, že se zrovna Kubernetes zaktualizovalo na novou verzi. Na to začal reagovat Ingress Nginx Controller odlišným chováním, což vyústilo v nefunkčnost celé aplikace. Podobných problémů bylo víc a nelze se jim zcela vyhnout, protože Kubernetes je vskutku komplexní nástroj, který je pod velice aktivním vývojem a je často aktualizován. Dále také nástrojů nutných k chodu aplikace je víc, a právě ona kombinace těchto nástrojů nevyhnutelně vede k občasným problémům, které komplikují a prodlužují vývoj.

Také samotný vývoj byl prodloužen, protože bylo třeba řešit řadu problémů, jejichž řešení je ve srovnání s monolitickou verzí zkrátka složitější. Jedná se například o zmíněné řešení zabezpečení aplikace, komunikaci mezi mikroservisami, či sdílení kódu.

Činnost	**Délka trvání**
Volba technologií pro vývoj	3
Návrh jednotlivých mikroservis aplikace a tvorba jejich ER diagramů	4
Čas strávený s tvorbou vývojové a produkční verze Kubernetes clusteru	4
Vývoj aplikace	15
Nasazení aplikace	1
Celkem	**27**

Tabulka 2 - Časový průběh tvorby mikroservisové verze aplikace. Zdroj: vlastní zpracování

4.6 Testování výkonu a zátěže

4.6.1 Nástroje využité při testování

Pro testování výkonu API existuje řada nástrojů. API je možné testovat například pomocí velmi známého nástroje Postman. Nicméně bylo zjištěno, že Postman neposílá dotazy najednou, ale jeden po druhém a API tak vlastně není moc zatíženo a otestováno. (121)

Po bližším prozkoumání možností byl zvolen nástroj JMeter. A to proto, že je zadarmo, je velice populární, umožňuje širokou škálu nastavení, a má vcelku dobré uživatelské rozhraní.

Kromě JMeteru, který nakonec zobrazuje výsledky samotného měření s ohledem zejména na rychlost vyřizování požadavků, či datovou propustnost, bylo usouzeno, že je potřeba měřit i vytížení hardwaru serveru, a to konkrétně vytížení procesoru a paměti.

Pro tyto účely byly vybrány dva nástroje. Prvním je Portainer, který slouží zejména jako uživatelské rozhraní pro práci s Dockerem a Kubernetes, nicméně je také schopen v rámci Dockeru zobrazovat i metriky využitých zdrojů jednotlivých kontejnerů. (122)

V rámci monitorování Kubernetes clusteru bohužel bylo zjištěno, že Portainer zobrazuje dobře metriky pro Docker kontejnery, ale zobrazení metrik pro Kubernetes cluster je opravdu omezené. Kubernetes cluster tak byl monitorován pomocí Kubernetes

Dashboard. Kubernetes Dashboard plní obdobnou funkci jako Portainer, jen s tím rozdílem, že lépe zobrazuje využití zdrojů serveru. (123)

4.6.2 Podmínky testování

Aby bylo odstíněno co nejvíce faktorů zkreslujících výsledky testování, tak byl použit vlastní server, který běžel v rámci lokální sítě. Server je vcelku primitivní. Jedná se o starý notebook přetvořený na server. Na výkonném hardwaru by byly výsledky nejspíš trochu jiné a odezvy rychlejší. Nicméně vychází se z předpokladu, že rozdíly mezi oběma architekturami budou viditelné i zde, protože obě verze aplikace budou otestovány v rámci stejného prostředí. Pro autora práce bylo toto řešení přijatelnější a vhodnější oproti možnosti si například pronajmout virtuální privátní server a riskovat tak, že bude testování zkresleno případným momentálním zpomalením rychlosti internetu v rámci testování jedné z verzí, či nějakým momentálním větším vytížením serverů poskytovatele.

Testovací stroj má 8 GB paměť, 512 GB SSD disk a procesor Intel core i7-3630QM 2,40GHz. Jako operační systém byl zvolen Ubuntu Server.

V rámci mikroservisové aplikace byli použity 2 repliky pro všechny služby obsahující koncové body aplikace.

4.6.3 Sledované veličiny

- **Průměr** – průměrný čas potřebný pro vyřízení požadavků. Čím je hodnota menší, tím lépe. (124)
- **Obdržené KB** – udává, kolik kilobajtů za sekundu je obdrženo klientem. Čím je hodnota větší, tím lépe. (124)
- **Poslané KB** – udává, kolik kilobajtů za sekundu je posláno serveru. Čím je hodnota větší, tím lépe. (124)
- **Datová propustnost** – udává, kolik požadavků za sekundu (req/sec) bylo v průběhu testování zpracováno. Čím větší je toto číslo, tím lépe si aplikace vede.

Také je zde absolutní rozdíl mezi mikroservisovou a monolitickou architekturkou, který byl vypočítán dle vzorce:

$$|\text{mikroservisová} - \text{monolitická}|$$

4.6.4 Testovací scénáře

1. Scénář - 1000 dotazů od 1 uživatele pro získání aktuálních informací o sektorech daného patra
2. Scénář - 1000 dotazů od 100 uživatelů pro získání aktuálních informací o sektorech daného patra
3. Scénář - 1000 dotazů od 300 uživatelů pro získání aktuálních informací o sektorech daného patra
4. Scénář - 1000 dotazů od 100 uživatelů pro získání aktuálních informací o sektorech daného patra a také o detailních informacích o konkrétním sektoru

4.6.4.1 1000 dotazů od 1 uživatele pro získání aktuálních informací o sektorech daného patra

V tabulce č. 3 lze vidět, že si mikroservisová verze vedla o trochu lépe než monolitická ve všech měřených hodnotách. Je to hezkým příkladem toho, co bylo řečeno v teoretické části a sice, že vše závisí na samotném návrhu aplikace a typu funkcionalit aplikace. Aplikace v diplomové práci byla navržena tak, aby při odbavování požadavků vyžadovala co nejmenší interní komunikaci mezi službami. Tomu samozřejmě nahrává i fakt, že v aplikaci značně převažují GET koncové body, které díky zmíněnému návrhu nemusí pro potřebná data komunikovat s dalšími službami.

Dále je to také díky tomu, že celá serverová část aplikace není příliš složitá a neobsahuje řadu entit například ve stylu „položka“ a „objednávka“. V takové aplikaci by totiž nejspíše každá z entit představovala jednu službu, a například vytvoření objednávky, která se skládá z řady položek, by nešlo uskutečnit bez využití obou služeb, což by vyžadovalo jejich komunikaci a aplikaci by to zpomalovalo.

Kombinace návrhu aplikace, implementace vyvažovače zátěže v podobě Nginx Ingress Controlleru a využití dvou replik příslušné služby umožňuje rychlejší a výkonnější běh aplikace.

Verze	Průměr	Obdržených KB	Poslaných KB	Datová propustnost
Mikroservisová	14 ms	193,39 KB/sec	8,78 KB/sec	68,6 req/sec
Monolitická	15 ms	183,76 KB/sec	8,32 KB/sec	64,6 req/sec

Rozdíl	1 ms	9,63 KB/sec	0,46 KB/sec	4 KB req/sec

Tabulka 3 - Výsledky testování 1000 dotazů od 1 uživatele. Zdroj: vlastní zpracování

Je vhodné uvést náhled i z druhé strany, konkrétně se podívat na to, jak moc zdrojů serveru obě verze vyžadují. Mikroservisová verze během testu využila 799MB paměti 4,85 % procesoru. Oproti tomu monolitická verze ke svému chodu potřebovala pouze 128,7MB paměti a 5,88 % procesoru. Je tak jasné, že zejména s ohledem na využití paměti, je mikroservisová verze hardwarově náročnější, přičemž procesor je zde využit přibližně stejně. Značný rozdíl ve využití paměti je zapříčiněn zejména tím, že v mikroservisové verzi aplikace server hostuje vícero aplikací. Konkrétně každá služba představuje jednu aplikaci, která již jen pro svůj chod bez vytížení využívá určité množství zdrojů. Dále jak již bylo řečeno, tak některé služby mají dvě repliky, přičemž každá z nich opět využije určité zdroje.

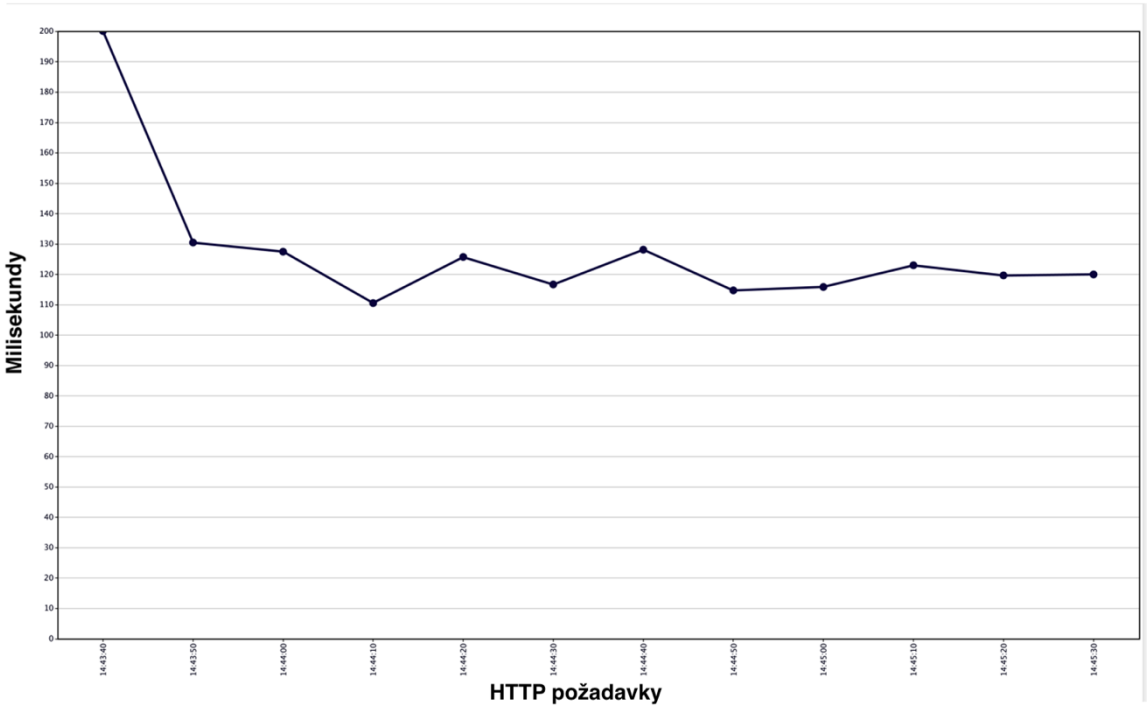
4.6.4.2 1000 dotazů od 100 uživatelů pro získání aktuálních informací o sektorech daného patra

V tabulce č. 4 lze vidět, že mikroservisová verze si opět vedla lépe než monolitická. V průměru posílala odpovědi rychleji, mezi klientem a serverem proudilo více dat a odbavování požadavků bylo dle datové propustnosti také lepší.

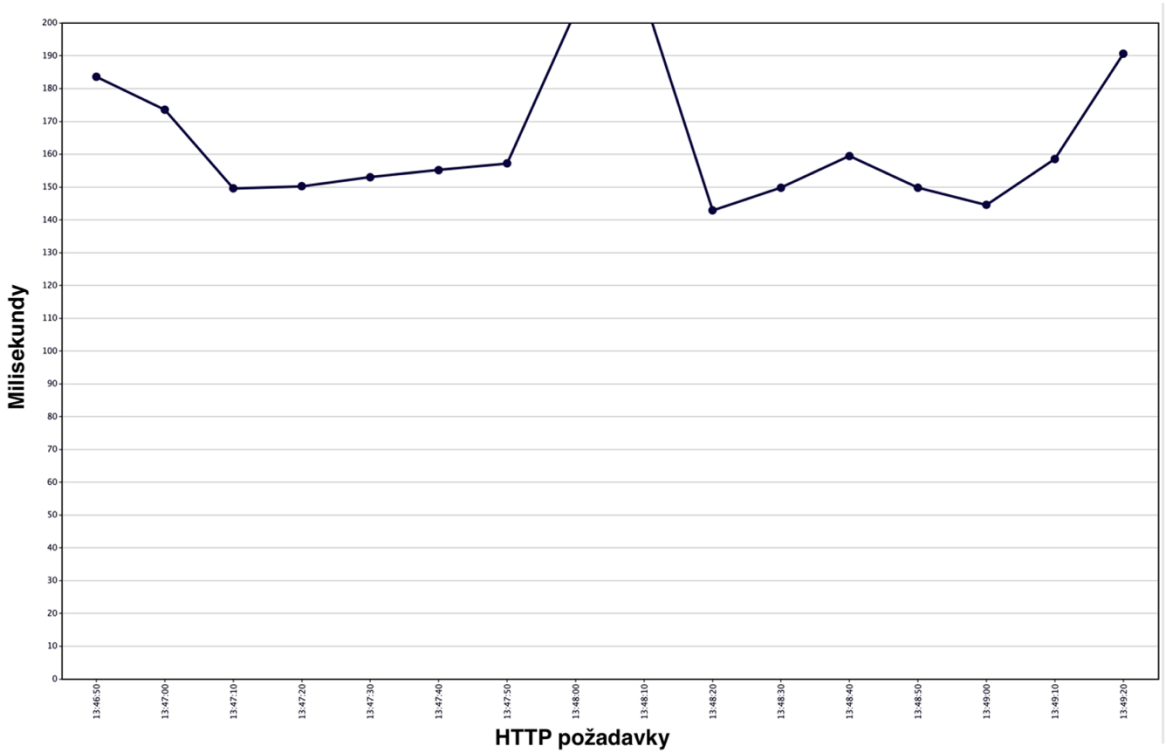
Verze	Průměr	Obdržených KB	Poslaných KB	Datová propustnost
Mikroservisová	120 ms	2309,26 KB/sec	104,70 KB/sec	818,4 req/sec
Monolitická	159 ms	1756,17 KB/sec	79,58 KB/sec	617 req/sec
Rozdíl	39 ms	553,09 KB/sec	25,12 KB/sec	201,4 req/sec

Tabulka 4 - Výsledky testování 1000 dotazů od 100 uživatelů. Zdroj: vlastní zpracování

Na obrazcích č. 19 a č. 20 je možné vidět grafy znázorňující odbavování požadavků v milisekundách během celého testování. Pokud pomineme počátek, tak je možné si povšimnout, že mikroservisová verze je v čase mnohem stabilnější a časy odpovědí nemají oproti monolitické verzi tak velké výkyvy.



Obrázek 19 - Graf zpracování odpovědí s milisekundách v mikroservisové verzi. Zdroj: výstup z aplikace JMeter



Obrázek 20 - Graf zpracování odpovědí s milisekundách v monolitické verzi. Zdroj: výstup z aplikace JMeter

Během tohoto testu mikroservisová verze využívala přibližně 46,5 % procesoru a 1 360MB paměti. Monolitická využívala 24,5 % procesoru a 222,6MB paměti. Rozdíl je zde již opravdu znatelný a mikroservisová verze zatěžuje procesor i paměť podstatně více. Oproti předchozímu scénáři je tedy i procesor značně více vytížen. Kromě samotného faktu, že server musí v rámci mikroservisové verze hostovat vícero aplikací, je zde také to, že příslušná služba odbavující požadavky má dvě repliky. To umožňuje Ingress Nginx Controlleru vyvažovat zátěž a rozdělovat požadavky mezi těmito dvěma replikami.

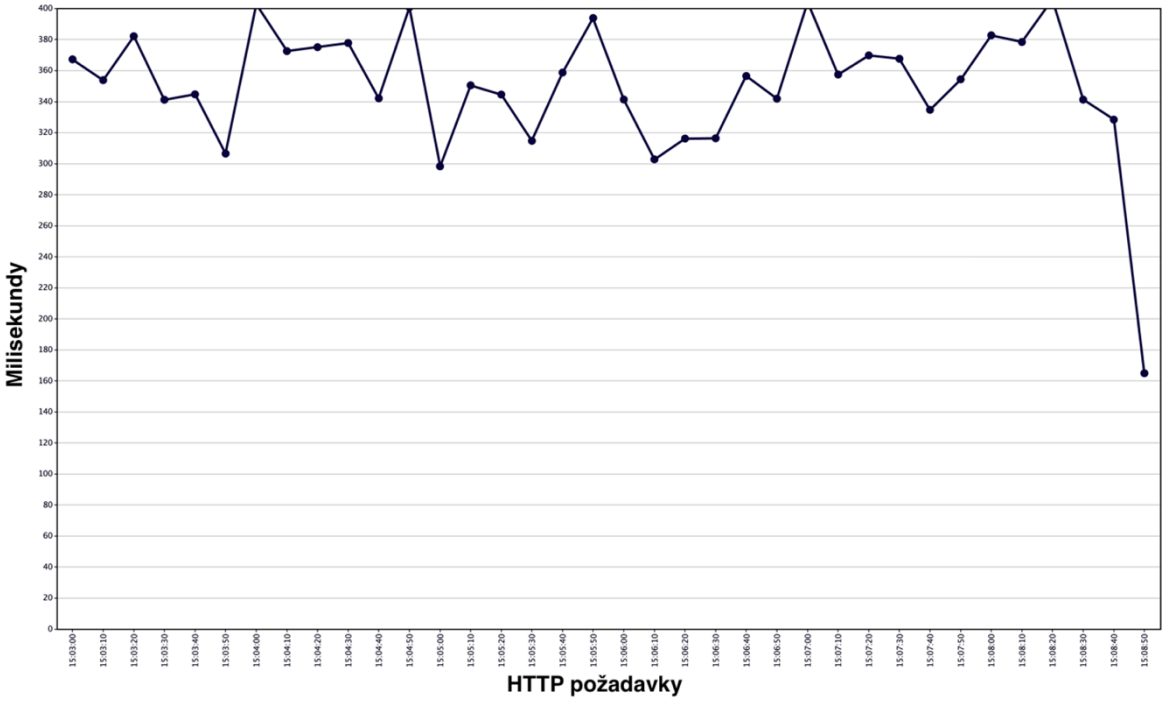
4.6.4.3 1000 dotazů od 300 uživatelů pro získání aktuálních informací o sektorech daného patra

Jedná se o prakticky stejný scénář jako ten předešlý s tím rozdílem, že je zde více uživatelů. Bylo testováno, zdali nebude s počtem přirůstajících uživatelů narůstat také rozdíl mezi výkonem. Oproti předchozímu scénáři se zvýraznil rozdíl hlavně v rámci průměrné odezvy v milisekundách, která se zde liší o 104 milisekund ve prospěch mikroservisové verze. V předchozím scénáři tento rozdíl činil 39 milisekund, což napovídá tomu, že mikroservisová verze zvládá přibývající zátěž lépe než monolitická verze.

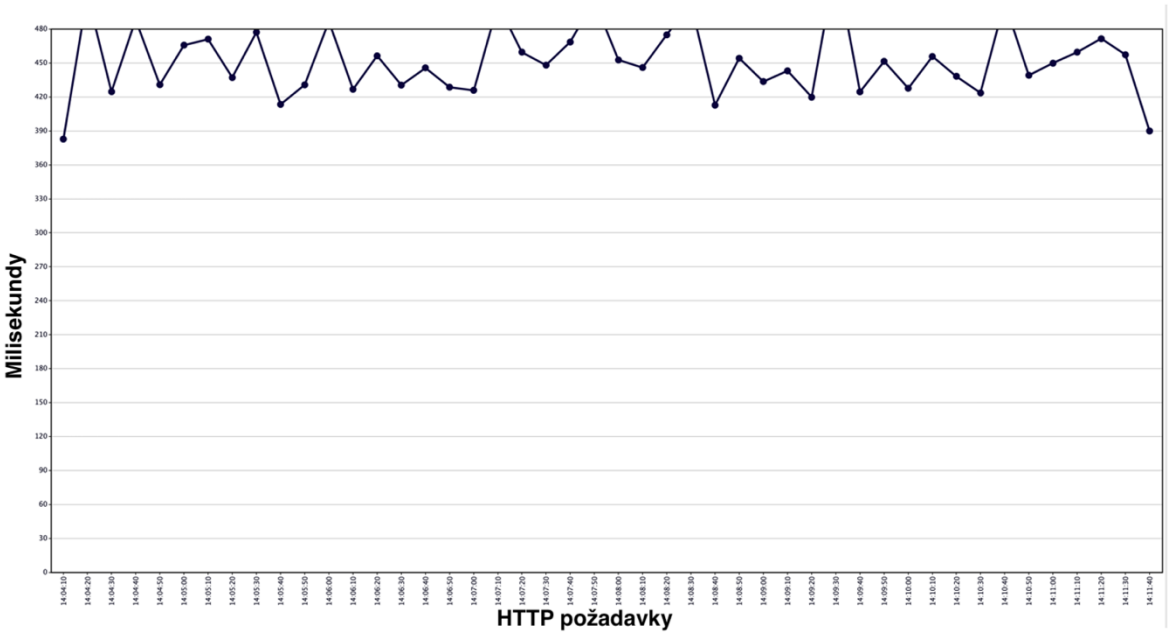
Verze	Průměr	Obdržených KB	Poslaných KB	Datová propustnost
Mikroservisová	346 ms	2382,39 KB/sec	107,95 KB/sec	843,8 req/sec
Monolitická	450 ms	1879 KB/sec	85,11 KB/sec	660,3 req/sec
Rozdíl	104 ms	503,39 KB/sec	22,84 KB/sec	183,5 req/sec

Tabulka 5 - Výsledky testování 1000 dotazů od 300 uživatelů. Zdroj: vlastní zpracování

Na obrázcích č. 21 a č. 22 je možné opět vidět grafy znázorňující odbavování požadavků v milisekundách během celého testování. Zde nelze konstatovat, že by jeden průběh byl stabilnější než ten druhý.



Obrázek 21 - Graf zpracování odpovědí s milisekundách v mikroservisové verzi. Zdroj: výstup z aplikace JMeter



Obrázek 22 - Graf zpracování odpovědí s milisekundách v monolitické verzi. Zdroj: výstup z aplikace JMeter

S ohledem na zdroje serveru, mikroservisová verze využila 50 % procesoru a 882MB paměti, zatímco monolitická verze využila 24,13 % procesoru a 252 MB paměti.

Mikroservisová verze je tak opět náročnější na zdroje. Oproti předchozímu testu si mikroservisová verze vzala ještě více zdrojů procesoru, ale méně paměti.

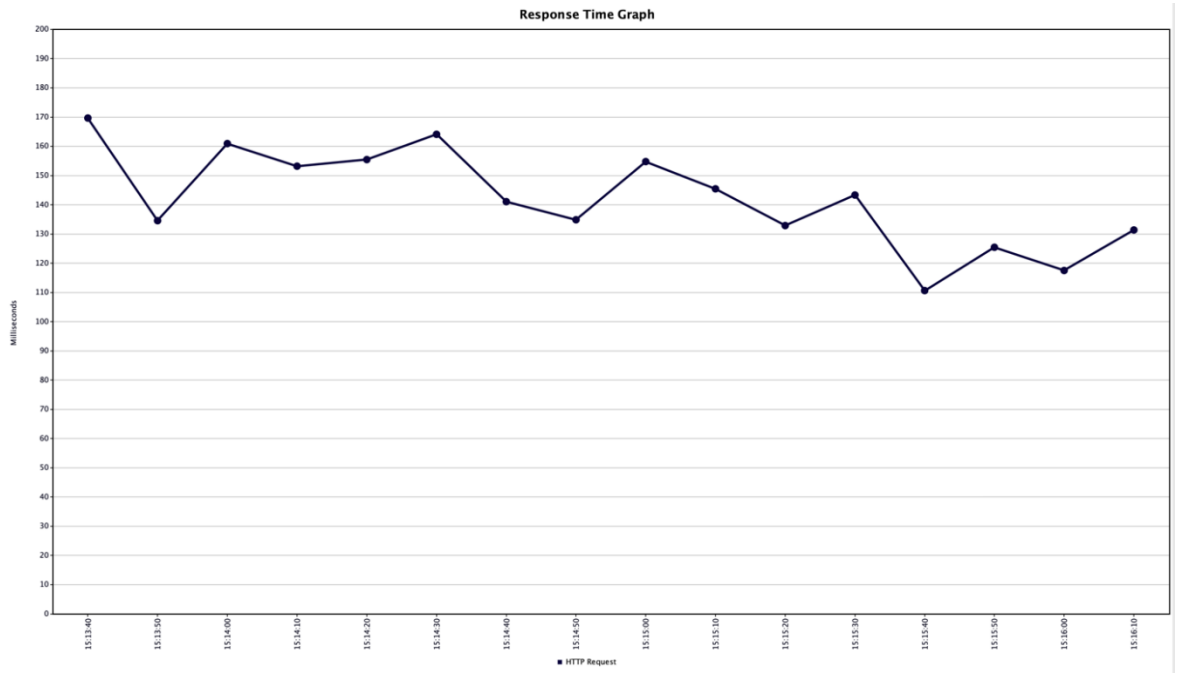
4.6.4.4 1000 dotazů od 100 uživatelů pro získání aktuálních informací o sektorech daného patra a také o detailních informacích o konkrétním sektoru

Tento scénář testoval, jak se bude lišit výkon v případě, že budou probíhat dva odlišené dotazy najednou, přičemž v rámci mikroservisové architektury je každý dotaz v jiné mikro službě a zátěž by tak měla být ještě lépe rozdělena. V tabulce č. 6 je vidět, že toto se potvrdilo, a výkonnostní rozdíl je zde opravdu ze všech testů nejvyšší ve prospěch mikroservisové architektury.

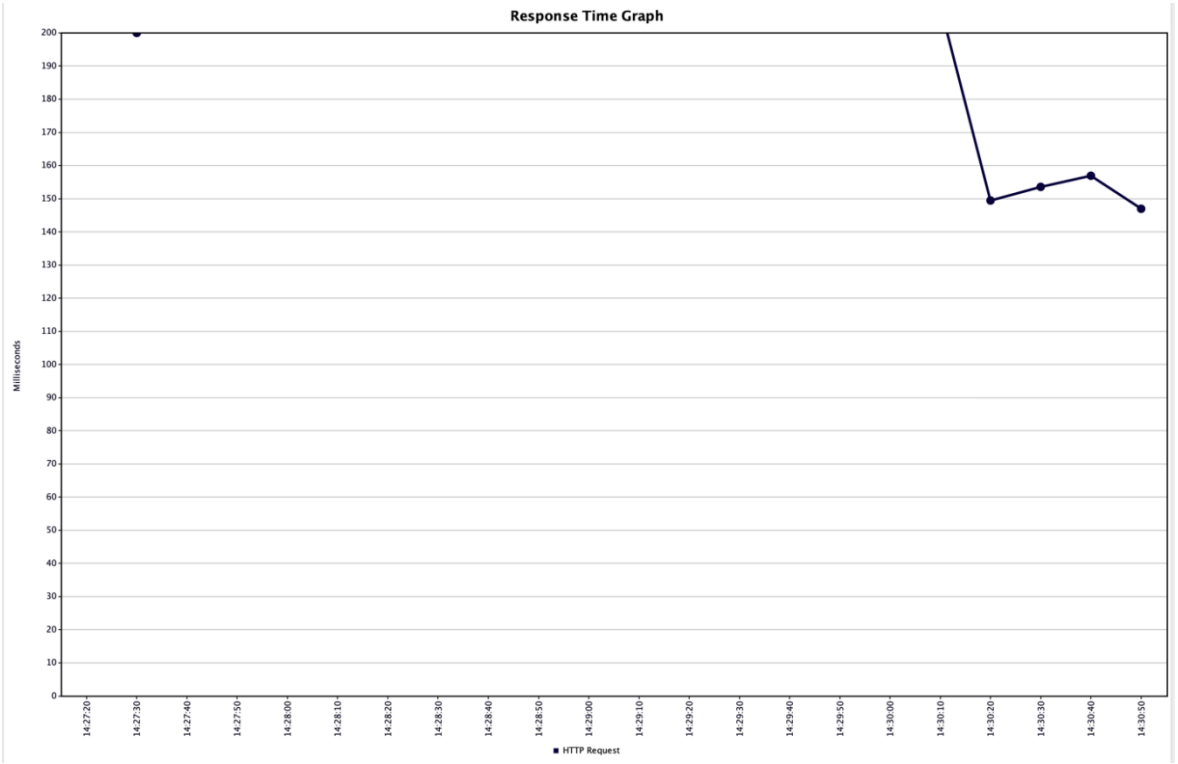
Verze	Průměr	Obdržených KB	Poslaných KB	Datová propustnost
Mikroservisová	142 ms	2050,69 KB/sec	182,12 KB/sec	1191,6 req/sec
Monolitická	202 ms	1532,75 KB/sec	135,19 KB/sec	878,9 req/sec
Rozdíl	60 ms	517,94 KB/sec	46,93 KB/sec	312,7 req/sec

Tabulka 6 - Výsledky testování 1000 dotazů na dva koncové body od 100 uživatelů. Zdroj: vlastní zpracování

Obrázek 23 a Obrázek 24 obsahuje grafy znázorňující odbavování požadavků v milisekundách během celého testování. Graf monolitické verze je velice zvláštní. Pro jistotu bylo toto testování raději několikrát zopakováno, nicméně pokaždé s obdobným výsledkem. Zde se opravdu značně projevuje lepší rozložení zátěže v mikroservisové verzi, zatímco v monolitické je server po většinu času opravdu hodně vytížen a vyřizování požadavků tak zkrátka trvá dlouho.



Obrázek 23 - Graf zpracování odpovědí s milisekundách v mikroservisové verzi. Zdroj: výstup z aplikace JMeter



Obrázek 24- Graf zpracování odpovědí s milisekundách v monolitické verzi. Zdroj: výstup z aplikace JMeter

Během testu mikroservisová verze využila 65,38 % procesoru a 945MB paměti. Monolitická verze využila 22 % procesoru a 224,7MB paměti. Opět tak využila

mikroservisová verze více zdrojů, přičemž rozdíl zde byl ze všech provedených testů největší.

4.7 Získání počtů řádků kódu

Počty řádků kódu byly zjištěny pomocí rozšíření do kódového editoru VS Code, které se jmenuje VS Code Counter. Je třeba uvést, že TypeScriptové aplikace automaticky generují značné množství řádků kódu, zejména v rámci souboru package-lock.json, který v sobě drží informace o všech závislostech v aplikaci včetně jejich verzí. Také je zde složka node_modules, která obsahuje veškeré knihovny použité v aplikaci. V mikroservisové verzi aplikace se nachází pro každou službu separátní package-lock.json a node_modules, což by značně zkreslovalo rozdíly mezi oběma verzemi aplikace. Z toho důvody byly z celkových počtů odečteny řádky kódu v rámci package-lock.json souborů a node_modules.

V následující tabulce lze vidět výsledné počty. Vzhledem k tomu, že je největší množství kódu napsáno v rámci klientské části, které se však změna architektury netýká, tak byly zjištěny i počty řádků kódu v rámci REST API, konfiguračních souborů a také řádky implementující komunikaci mezi službami. Ona komunikace mezi službami je problémem pouze u mikroservisové verze, jedná se však o velice důležitou část aplikace, kterou je třeba v počtech zahrnout.

	Mikroservisová	**Monolitická**	**Rozdíl**
Celkový počet	24612	23131	1481
Počet v rámci REST API	3788	2307	1481
V konfiguračních souborech (Kubernetes a Docker)	797	82	715
Pro vytvoření komunikace mezi mikroservisami	203	0	203

Tabulka 7 - Získané počty řádků kódu v obou verzích aplikace. Zdroj: vlastní zpracování

5 Výsledky a diskuse

5.1 Výsledky porovnání objektivních kritérií

5.1.1 Počet použitých technologií

Větší počet technologií znamená větší nároky na znalosti vývojářů, vícero míst, kde se mohou objevit chyby, či také větší časové nároky na tvorbu aplikace. Detailnější popis využitých technologií lze najít zejména v kapitolách „4.2.1 High level schéma architektury“ a „4.3.1 High level schéma architektury“.

Pro monolitickou verzi bylo použito následující:

- Node.js
- Framework Express.js
- React
- Jazyk TypeScript
- Nginx, který byl využit jako proxy
- PostgreSQL
- Prisma ORM
- Docker

Pro mikroservisovou verzi bylo použito to samé jako pro monolitickou, ještě ale přibylo následující:

- Kubernetes
- Skaffold
- RabbitMQ
- Ingress nginx

V tabulce č. 8 lze vidět, že mikroservisová verze experimentální aplikace používá o 4 technologie více než monolitická verze. Lze tak říct, že monolitická verze je na tom lépe.

	Mikroservisová	**Monolitická**	**Absolutní rozdíl**
Počet	12	8	4

Tabulka 8 - Přehled počtu použitých technologií v obou verzích aplikací. Zdroj: vlastní zpracování

5.1.2 Počty řádků kódu

V tabulce č. 9 lze vidět celkový přehled řádků kódu. Jak již bylo dříve řečeno, klientské části se změny architektury vůbec netýkají a celkový počet řádků kódu je zde uveden pouze pro úplnost. V rámci porovnání jsou zajímavé zejména další řádky tabulky. V mikroservisové verzi bylo napsáno o 1481 řádků kódu více a lze tak říct, že je na tom opět monolitická verze lépe. Značná část tohoto rozdílu pochází ze samotné konfigurace aplikace. To bylo způsobeno tím, že v mikroservisové verzi bylo třeba vytvořit konfigurační yaml soubory pro jednotlivé služby, aby mohly být následně spuštěny v rámci Kubernetes clusteru. Také bylo třeba nastavit Skaffold, který již byl zmíněn v rámci kapitoly „4.3.6 Nasazení vývojové a produkční verze aplikace“. Na druhou stranu v monolitické verzi stačilo pouze napsat Docker soubory a docker-compose, které v součtu mají 82 řádků. Další rozdíl v neprospěch mikroservisové verze je nutnost psaní kódu pro komunikaci jednotlivých služeb, kterou následně obstarává již zmíněný RabbitMQ. Rozdíl není tak markantní, jedná se o pouhých 203 řádků kódu. To je ale způsobeno tím, že v aplikaci není příliš vzájemné komunikace mezi službami a v jiných aplikacích může být toto číslo výrazně vyšší.

	Mikroservisová	**Monolitická**	**Rozdíl**
Celkový počet	24612	23131	1481
Počet v rámci REST API	3788	2307	1481
V konfiguračních souborech (Kubernetes a Docker)	797	82	715
Pro vytvoření komunikace mezi mikroservisami	203	0	203

Tabulka 9 - přehled počtu řádků kódu v obou verzích aplikace. Zdroj: vlastní zpracování

5.1.3 Časová náročnost aplikace

Tabulka č. 10 obsahuje finální rozdíl časové náročnosti tvorby mikroservisové a monolitické verze aplikace. Blíže byl časový průběh vývoje popsán v kapitole „4.5 Časová náročnost vývoje“. Vývoj mikroservisové verze trval téměř jednou tolik man days, což znamená, že s ohledem na časovou náročnost je na tom monolitická verze výrazně lépe.

Pro úplnost je dobré uvést, že na tomto značném rozdílu má svůj podíl fakt, že autor této práce má více zkušeností s tvorbou monolitických aplikací. Nicméně zejména technologie jako Kubernetes, a RabbitMQ jsou zkrátka opravdu komplexní, a jejich správná implementace zkrátka zabere značné množství času.

	Mikroservisová	**Monolitická**	**Absolutní rozdíl**
Strávený čas v man-days	27	14	13

Tabulka 10 - předhled času stráveného tvorbou obou verzí aplikace. Zdroj: vlastní zpracování

5.1.4 Výkon aplikace

Tabulka č. 11 obsahuje veškeré výsledky měření výkonu aplikace, které bylo blíže popsáno v kapitole „4.6 Testování výkonu“. Ve všech testech na vyšla mikroservisová verze lépe.

	Parametr	**Mikroservisová**	**Monolitická**	**Rozdíl**
1. scénář	Průměr v ms	14	15	1
	Obdržených KB/sec	193,39	183,76	9,63
	Poslaných KB/sec	8,78	8,32	0,46
	Datová propustnost req/sec	68,6	64,6	4
2. scénář	Průměr v ms	120	159	39
	Obdržených KB/sec	2309,26	1756,17	553,09
	Poslaných KB/sec	104,70	79,58	25,12
	Datová propustnost req/sec	818,4	617	201,4
3. scénář	Průměr v ms	346	450	104
	Obdržených KB/sec	2382,39	1879	503,39
	Poslaných KB/sec	107,95	85,11	22,84
	Datová propustnost req/sec	843,8	660,3	183,5
4. scénář	Průměr v ms	142	202	60
	Obdržených KB/sec	2050,69	1532,75	517,94

	Poslaných KB/sec	182,12	135,19	46,93
	Datová propustnost req/sec	1191,6	878,9	312,7

Tabulka 11 - přehled výsledků ze všech testovacích scénářů. Zdroj: vlastní zpracování

Pro shrnutí lze v tabulce č. 12 lze nalézt průměrné hodnoty za všechny testovací scénáře. Lze si všimnout, že výkonnostní rozdíly jsou opravdu znatelné ve prospěch mikroservisové verze.

Parametr	Mikroservisová	Monolitická	Rozdíl
Průměr v ms	155,5	206,5	51
Obdržených KB/sec	1733,9325	1337,92	396,0125
Poslaných KB/sec	100,8875	77,05	23,8375
Datová propustnost req/sec	730,6	555,2	175,4

Tabulka 12 - průměry naměřených hodnot výkonu aplikace ze všech testovacích scénářů. Zdroj: vlastní zpracování

5.1.5 Hardwarové vytížení serveru

V tabulce č. 13 jsou uvedeny výsledky vytížení hardwaru serveru během všech testovacích scénářů. Kromě vytížení procesoru v prvním scénáři vychází všude lépe monolitická verze aplikace. Rozdíl je značný, a to jak při vytížení procesoru, tak při vytížení paměti. Jak již bylo zmíněno v rámci testování aplikace, tak je toto vytížení způsobeno tím, že server musí v rámci mikroserisové verze hostovat vícero aplikací, a následně u některých služeb ještě vícero jejich replik. Díky použitým dvěma replikám služeb, může Ingress Nginx Controller vyvažovat jejich zátěž, což umožňuje rychlejší odezvy, ale zároveň to ještě více zvyšuje využití zdrojů serveru při zátěži aplikace.

	Parametr	Mikroservisová	Monolitická	Rozdíl
1. scénář	Vytížení procesoru v %	4,85	5,88	1,03
	Vytížení paměti v MB	799	128,7	670,3
2. scénář	Vytížení procesoru v %	46,5	24,5	22
	Vytížení paměti v MB	1360	222,6	1137,4
3. scénář	Vytížení procesoru v %	50	24,13	25,87
	Vytížení paměti v MB	882	252	630

4. scénář	Vytížení procesoru v %	65,38	22	43,38
	Vytížení paměti v MB	945	224,7	720,3

Tabulka 13 - přehled vytížení hardwarových zdrojů ve všech testovacích scénářích. Zdroj: vlastní zpracování

Stejně jako u výkonu aplikace, lze v tabulce č. 14 vidět průměrné hodnoty za všechny testovací scénáře.

Parametr	**Mikroservisová**	**Monolitická**	**Rozdíl**
Vytížení procesoru v %	41,68	19,13	22,55
Vytížení paměti v MB	996,5	207	789,5

Tabulka 14 - průměry naměřených hodnot vytížení serveru ze všech testovacích scénářů. Zdroj: vlastní zpracování

5.2 Subjektivní hodnocení architektur na základě získaných poznatků

Kromě objektivních kritérií, které lze kvantifikovat a při jejich porovnání je tak možné se opřít o naměřené hodnoty, ze kterých lze udělat objektivní závěry, je tu ještě řada dalších faktorů, které je nutné při volbě architektury zvážit. V této kapitole jsou tak shrnuty všechny poznatky o mikroservisové a monolitické architektuře, které byly získány díky nastudování problematiky, tvorbě experimentální aplikace a konzultacemi s odborníky z praxe. Jedná se však o subjektivní hodnocení, které je nutné brát s jistou rezervou a konečné rozhodnutí o volbě architektury je tak vždy na samotném vývojářském týmu.

5.2.1 Shrnutí poznatků o mikroservisové architektuře

Mikroservisová architektura má v řadě dnešních aplikací bez pochyb své místo. Avšak nelze pominout, že tato architektura s sebou nese řadu problémů a výzev, které při monolitickém vývoji menších aplikací buďto není vůbec nutné řešit, anebo je jejich řešení o poznání snazší. V teoretické části bylo řečeno, že jednou z výhod mikroservisů je, že lze jednotlivé služby vyvíjet pomocí různých technologií. To je jistě pravda, avšak při samotném vývoji bylo zjištěno, že využití různých jazyků má svá úskalí a přináší komplikace zejména s ohledem na sdílení kódu napříč službami.

Další z výhod mikroservis je, že jednotlivé služby mohou být vyvíjeny na sobě nezávislými týmy a také mohou být měněny a nasazovány nezávisle na okolí, přičemž pro menší aplikace je samozřejmě výhodná zejména ona nezávislost jednotlivých logických částí, díky které je zredukována šance, že jedna malá změna rozbije celou aplikaci. Skutečně tomu tak je, avšak jistá úroveň komunikace a koordinace je mezi týmy nezbytná a míra autonomie jednotlivých týmů silně závisí na tom, jak nezávislé skutečně jednotlivé služby jsou. Tato nezávislost se značně odvíjí mimo jiné i od samotné povahy aplikace a některých rozhodnutích při jejím návrhu. Celý vývojářský tým se určitě musí dohodnout na tom, jak mezi sebou budou služby komunikovat, případně jakou strukturu tyto zprávy budou mít. Dále záleží, jak moc kódu se bude mezi službami sdílet a samozřejmě čím více bude sdíleného kódu, tím více budou služby propojeny. U povahy aplikace zase záleží, zdali se její funkcionality vůbec dají rozdělit na smysluplné samostatné celky, či nikoliv. Ve vyvíjené aplikaci bylo rozdělení vcelku dobře možné a ani nebylo nutné řešit posloupnost zpracování zpráv. To ale není samozřejmostí, a například nějaký rezervační systém hotelů by vyžadoval mnohem větší provázanost služeb, výrazně větší množství posílaných zpráv mezi službami a nutnost zohlednění posloupnosti zpracování zpráv. Pokud by totiž v takovém systému nebyla posloupnost zohledněna, mohlo by se stát, že uživatel si zarezervuje pokoj a následně rezervací zruší, přičemž by ale při vysokém vytížení sítě mohla nastat situace, kdy zpráva o zrušení byla zpracována dřív než zpráva o rezervaci. Taková aplikace by pak v porovnání se zde vyvíjenou aplikací byla výrazně komplexnější a přinášela by další výzvy.

Nepopiratelnou výhodou je škálovatelnost aplikace, kdy je při vyšší zátěži možné nasadit další repliky jen oněch vytížených částí. Zde je pouze jeden neduh, a to sice fakt, že každá služba si zkrátka vyžádá své hardwarové zdroje, které jak bylo zjištěno v testování zátěže opravdu nejsou malé. V posledním zátěžovém testu, kdy byly zatíženy dvě služby najednou bylo zjištěno, že vytížení procesoru bylo bezmála trojnásobné a vytížení paměti téměř pětinásobné. Aby této škálovatelnosti bylo dostatečně využito, je nutné nasadit aplikaci na patřičně silný server, či dokonce na vícero serverů. To by zejména u menších aplikací s malým rozpočtem mohl být problém.

Mikroservisová architektura je oproti té monolitické značně robustnější. To pramení zejména z již zmíněné autonomie jednotlivých služeb a pokud tak přestane fungovat jedna služba, tak to hned neznamená, že nefunguje celá aplikace. Toto je výhodou zejména

u větších aplikací, protože u menších aplikací, kde je méně služeb, značně záleží na tom, který kus aplikace přestane fungovat. V experimentální aplikaci je to krásně vidět. Zatímco například pád „Přehledů“, či „Auth“ není tak hrozný, tak pád „Prostorů“ by způsobil nefunkčnost té nejpodstatnější části aplikace.

5.2.1.1 Případy použití mikroservisové architektury v rámci menších webových aplikací

Na základě všeho zjištěného nutno konstatovat, že pro většinu menších aplikací nedává smysl použití mikroservisové architektury, a to z následujících důvodů:

- Naprosto zásadním důvodem jsou výzvy a problémy, které mikroservisy přináší. Je potřeba oproti monolitické verzi více řešit zabezpečení v rámci aplikace, zabezpečení v rámci nastavení Kubernetes, problematiku sdílení kódu, návrh jednotlivých mikroservis, a další. To klade mnohem větší nároky na znalosti vývojáře.
- S předchozím bodem je spojen fakt, že vývoj aplikace je značně prodloužen. V experimentální aplikaci to bylo konkrétně téměř na dvojnásob. To je problém, protože u menších aplikací bývá často nátlak na co nejrychlejší vytvoření a také jejich rozpočet nebývá velký.
- Na základě měření výkonnosti aplikace bylo ověřeno tvrzení, že mikroservisové aplikace vyžadují o poznání více hardwarových zdrojů. Také u této architektury vůbec nepřipadá v úvahu nejlevnější způsob nasazení aplikace, kterým je sdílený hosting. Je určitě férové uvést, že monolitická verze byla vyvíjena v Dockeru a také k jejímu vývoji byl využit Node.js. Takové aplikace vyžadují alespoň virtuální privátní server, avšak pro jejich chod stačí minimální nabízené konfigurace. V této konfiguraci dle průzkumu známých českých poskytovatelů, mezi které patří Wedos, Váš Hosting, či Český Hosting, je nabízeno 2 GB RAM a jeden procesor. (125) (126) (127) Při provedených testech by to dostačovalo, ale při větší zátěži, či rozšíření počtu služeb již by zejména paměť nemusela být dostačující a bylo by nutné si připlatit.

Určitě nelze říct, že mikroservisy pro menší aplikace nikdy nedávají smysl. Jsou aplikace, které jsou malé s ohledem na množství napsaného kódu, či na počet vývojářů,

avšak jejich zátěž je velká. V takovém případě je škálování v rámci mikroservis dobrým způsobem, jak se s vysokou zátěží vyrovnat.

Také určitě záleží na tom, kdo aplikaci vyvíjí. Pokud se jedná o vývojáře, který má s mikroservisami již nějaké zkušenosti, tak může mít na náročnost vývoje jiný názor než autor této práce a může být schopen aplikaci bez větších problémů vytvořit v rozumném čase.

Dalším adeptem typu menší aplikace vhodné pro adaptaci mikroservis je právě aplikace vyvíjená v rámci této práce a sice aplikace vyvíjena pro potřeby univerzity. Na univerzitě je řada schopných učitelů a studentů, kteří mají potenciál k tvorbě tohoto typu aplikací. V takovém prostředí je také větší tolerance při výskytu bezpečnostních chyb a vývoj je i při delším trvání vcelku levný. To je z prostého důvodu. Financovat projekt vyvíjený studenty je zkrátka levnější než financovat komerční projekt vyvíjený profesionálními programátory, přičemž se od studentů neočekává stejná míra profesionality a bezchybnosti výsledného produktu.

Posledním vhodným případem užití je situace kdy, již aplikace existuje v monolitické verzi, má úspěch a je očekáván její růst jak v uživatelské základně, tak v množství nabízených funkcí. Tato situace již samozřejmě hraničí s tím, že menší aplikace přerůstá ve větší, a je tak možné namítnout, že zcela nespadá do záběru této diplomové práce. Je však nutno říct, že pokud má aplikace opravdu růst, tak je dobré ji přepsat do této architektury, dokud je ještě relativně malá, což znamená, že nějaký čas ještě také malá zůstane. Taková situace může vývojáře svádět k prosté snaze o udržení monolitické architektury a pouhé přepsání některých jejích částí, které původně nebyly dobře napsány. To se může jevit jako levnější a optimálnější řešení, protože je možné nový kód jen nabalovat na již existující řešení. Dle získaných zkušeností však nutno konstatovat, že je v takové situaci vhodnější přejít na mikroservisovou architekturu. Zejména kvůli tomu, že se řada jejích nevýhod pojí hlavně s počátečním vývojem aplikace, kdy je nutné udělat mnoho rozhodnutí a zavést řadu nových technologií. Po této počáteční fázi však výhody začínají převažovat nad nevýhodami. Aplikace je tak po přechodu stabilnější, snáze spravovatelná a rozšiřitelná.

Navíc při brzkém přechodu je eliminováno riziko, že aplikace značně přeroste svůj původní záměr, přičemž přepsání velkých monolitických aplikací je nelehkým a nákladným úkolem. Dokonce se v takovém případě doporučuje postupná adaptace, která může trvat dlouho. (83)

5.2.2 Shrnutí poznatků monolitické architektuře

V porovnání s mikroservisami, jsou monolitické aplikace s ohledem na vývoj menších aplikací snazší. Odpadá zde řada problémů, které urychlují vývoj. Na druhou stranu monolitický vývoj tolik vývojářský tým nenutí k dobrému promyšlení aplikace a jejich funkcionalit. Na začátku je samozřejmě vždy dobré sepsat detailně co má aplikace dělat, jak konkrétně má fungovat, vytvořit různá schémata a diagramy, také vytvořit návrh aplikace, vytvořit scénáře užití, a s tím spojené testovací scénáře. To je však naprosto ideální scénář průběhu návrhu a vývoje. Nutno konstatovat, že dle zkušeností z praxe není popsaný postup vždy dodržován. Monolitická architektura zkrátka umožňuje velice rychle začít psát kód bez hlubšího přemýšlení, z čehož pramení nevyhnutelné a mnohdy opakované přepisování různých částí aplikace, které naopak vývoj komplikuje a prodlužuje. Oproti tomu mikroservisovou aplikaci ani nelze bez dobré přípravy začít vyvíjet. To bylo názorně v této práci demonstrováno již při popisu jednotlivých služeb a jejich ER diagramů. Bez dobrého promyšlení funkcí aplikace by vývojář nebyl schopen jednotlivé služby ani navrhnout.

Kód je v monolitické verzi mnohem více provázaný a změna jedné části aplikace může vyústit v neočekávané chování zcela odlišné části aplikace. U menších aplikací samozřejmě kódu není tolik a chyby se hledají snáz než ve větších projektech, nicméně i tak je občas opravování chyb díky této provázanosti obtížné. To je zejména díky tomu, že je v porovnání s mikroservisami obtížné vůbec v prvním kroku vymezit rozsah oblasti, ve které je nutné chybu hledat.

5.2.2.1 Případy použití monolitické architektury v rámci menších webových aplikací

Z předchozí kapitoly vyplývá, že použití monolitické architektury je vhodné pro většinu menších aplikací. Je však nutné brát v potaz rozpočet, znalosti vývojáře, předpokládané vytížení aplikace a samotné zadání. Zejména to zadání je podstatné, protože jak již bylo řečeno, u vývoje monolitické aplikace to občas svádí k přeskočení důkladné analýzy aplikace a následnému dobrému zpracování dokumentace. To je však zrádné, protože se může stát, že je aplikace malá a jednoduchá pouze na první pohled díky přehlédnutí složitosti některých jejích funkcí, či nedostatečnému domyšlení, jak mají některé funkce reálně fungovat. Aplikace tak může značně přerůst vývojářskému týmu přes hlavu

jen proto, že na samotném začátku nebyla dostatečně zanalyzovaná a zdokumentovaná, načež na tyto chyby nebylo včas upozorněno.

6 Závěr

V teoretické částí práce byly charakterizovány technologie a možnosti, které jsou v současné době při tvorbě webových aplikací k dispozici. Nejdříve byly představeny typy webových aplikací, načež následoval popis modelu klient-server. Poté byly v kapitole „3.4 Technologie využívané pro vývoj webových aplikací“, popsány jazyky a frameworky využívané pro tvorbu webových aplikací. Dále byly teoreticky popsány obě porovnávané architektury, jmenovitě mikroservisová a monolitická. V závěru teoretické části byly přiblíženy technologie, které se dnes hojně využívají pro vytvoření prostředí, ve kterém jsou webové aplikace vyvíjeny, a následně také hostovány na produkčních serverech. Na úplném konci teoretické části lze také najít kapitolu o obdobných řešeních, ve které jsou uvedeny práce zabývající se problematikou monolitické a mikroservisové architektury.

První část vlastní práce byla věnována popisu tvorby experimentální webové aplikace, a to konkrétně v kapitolách „4.2 Tvorba monolitické verze“ a „4.3 Tvorba mikroservisové verze“, ve kterých se lze dočíst o implementaci obou verzí aplikace. Následně byla vybrána vhodná kritéria pro porovnání architektur.

Po zvolení vhodných kritérií bylo provedeno měření aplikace. Nejdříve byla v kapitole „4.5 Časová náročnost vývoje“ popsána časová náročnost vývoje aplikace. Poté bylo provedeno měření výkonu a zátěže aplikace, které je popsáno v kapitole „4.6 Testování výkonu a zátěže“. Následně byly také spočítány počty řádků kódu a počty použitých technologií.

Na základě změřených výsledků byly porovnány obě architektury. V rámci kapitoly „5.1 Výsledky porovnání objektivních kritérií“ bylo provedeno porovnání architektur dle již zmíněných vybraných kritérií. Na základě těchto výsledků tak lze konstatovat, že mikroservisové aplikace jsou náročnější technologicky, hardwarově i časově. Na druhou stranu je lze lépe škálovat, díky čemuž je při dobrém návrhu možné dosáhnout lepšího výkonu. Pro celistvé porovnání architektur, však tato objektivní kritéria nestačí, a proto bylo toto porovnání v kapitole „5.2 Subjektivní hodnocení architektur na základě získaných poznatků“ doplněno o subjektivní názory získané na základě konzultace s odborníky z praxe a poznatků získaných nastudováním problematiky a tvorbou experimentální aplikace.

7 Citovaná literatura

1. **Zprávičky .** ARPANET, předchůdce internetu, byl spuštěn před 50 lety. *ITBIZ.* [Online] 2. Září 2019. [Citace: 27. Únor 2022.] https://www.itbiz.cz/zpravicky/arpanet-predchudce-internetu-byl-spusten-pred-50-lety.

2. **Uryutin, Oleg.** A brief history of web app . *Medium.* [Online] 13. Září 2018. [Citace: 28. Únor 2022.] https://oleg-uryutin.medium.com/a-brief-history-of-web-app-50d188f30d.

3. **Kariyawasam, Mehani.** The Evolution of the Web: How Application Development Changed as a Result. *Medium.* [Online] 17. Červen 2020. [Citace: 28. Únor 2022.] https://medium.com/@mehani_kariyawasam/the-evolution-of-the-web-how-application-development-changed-as-a-result-84354e6349bf.

4. **ManagementMania.com.** Webová aplikace (Web Application). *Managementmania.* [Online] 18. Říjen 2018. [Citace: 19. Únor 2022.] https://managementmania.com/cs/webova-aplikace-web-application.

5. **Koďousková, Barbora.** WEB, WEBOVÁ STRÁNKA A WEBOVÁ APLIKACE, V ČEM JE ROZDÍL? *Rascasone.* [Online] 22. Červen 2021. [Citace: 19. Únor 2022.] https://www.rascasone.com/cs/blog/web-webova-aplikace-rozdil.

6. **Codecademy.** What is a Web App? *codeacademy.* [Online] [Citace: 19. Únor 2022.] https://www.codecademy.com/article/what-is-a-web-app.

7. **TechTarget Contributor.** Web application (Web app). *TechTarget.* [Online] Srpen 2019. [Citace: 19. Únor 2022.] https://searchsoftwarequality.techtarget.com/definition/Web-application-Web-app.

8. **Angular Minds .** 10 Different Types of Web App Development . *Medium.* [Online] 13. Říjen 2020. [Citace: 19. Únor 2022.] https://medium.com/angularminds/10-different-types-of-web-app-development-a9bedb2fdcee.

9. **HELLO DARWIN.** What are the different types of web apps? *hellodarwin.* [Online] 23. Květen 2019. [Citace: 19. Únor 2022.] https://hellodarwin.com/blog/different-web-apps.

10. **Editorial Team.** Web Application Development – Ultimate Guide to 10 Different Types in 2022. *Clustox.* [Online] 2022. [Citace: 19. Únor 2022.] https://www.clustox.com/web-application-development-ultimate-guide-to-10-different-types/.

11. **Neoteric.** Single-page application vs. multiple-page application. *Neoteric.* [Online] 2. Prosinec 2016. [Citace: 28. Leden 2022.] https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.

12. **Shah, Krunal.** Single-Page Apps vs Multi-Page Apps: What To Choose For Web Development. *ThirdRockTechno.* [Online] 17. Červen 2020. [Citace: 28. Leden 2022.] https://www.thirdrocktechkno.com/blog/single-page-apps-vs-multi-page-apps-what-to-choose-for-web-development/.

13. **Nishu_Dissanayake.** When to use Multi-Page Apps? *Bits and Pieces.* [Online] 25. Listopad 2021. [Citace: 28. Leden 2022.] https://blog.bitsrc.io/when-to-use-multi-page-apps-587030b0f37b.

14. **Valuy, Sergey.** A Comparison of Single-Page and Multi-Page Applications. *DZone.* [Online] 17. Červen 2020. [Citace: 28. Leden 2022.] https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli.

15. **Refsnes Data.** What is JSON? *W3 schools.* [Online] [Citace: 25. Leden 2022.] https://www.w3schools.com/whatis/whatis_json.asp.

16. **ASPER BROTHERS.** Single Page Application (SPA) vs Multi Page Application (MPA) – Two Development Approaches. *ASPER BROTHERS.* [Online] 12. Listopad 2019. [Citace: 28. Leden 2022.] https://asperbrothers.com/blog/spa-vs-mpa/.

17. **Braun, Tyson.** Optimizing Single-Page Applications for Crawling and Indexing Purposes. *seoClarity.* [Online] 16. Říjen 2020. [Citace: 30. Leden 2022.] https://www.seoclarity.net/blog/single-page-applications.

18. **Faisal, Fazmeena.** How e-Commerce Applications Help Your Business Grow? *Mindster.* [Online] 29. Listopad 2019. [Citace: 19. Únor 2022.] https://mindster.com/ecommerce-applications/.

19. **Nyakundi, Hillary.** What is a PWA? Progressive Web Apps for Beginners. *freeCodeCamp.* [Online] 6. Duben 2021. [Citace: 20. Únor 2022.] https://www.freecodecamp.org/news/what-are-progressive-web-apps/.

20. **Mozilla and individual contributors.** Introduction to progressive web apps. *MDN Wev Docs.* [Online] [Citace: 20. Únor 2022.] https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#discoverability.

21. —. Installing and uninstalling web apps. *MDN Web Docs.* [Online] [Citace: 20. Únor 2022.] https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Installing.

22. **Farrugia, Kevin.** A Beginner’s Guide To Progressive Web Apps . *Smashing magazine.* [Online] 11. Srpen 2016. [Citace: 20. Únor 2022.] https://www.smashingmagazine.com/2016/08/a-beginners-guide-to-progressive-web-apps/.

23. **Kinsta Inc.** What Is a Content Management System (CMS)? *kinsta.* [Online] 2. Září 2021. [Citace: 19. Únor 2022.] https://kinsta.com/knowledgebase/content-management-system/.

24. **Fitzgerald, Anna.** What Is a CMS and Why Should You Care? *HubSpot.* [Online] 2. Prosinec 2021. [Citace: 19. Únor 2022.] What Is a CMS and Why Should You Care?.

25. **Rentech digital.** Advantages & Disadvantages of Using a Content Management System. *rentechdigital.* [Online] 24. Prosinec 2021. [Citace: 19. Únor 2022.] https://rentechdigital.com/advantages-and-disadvantages-of-cms.

26. **Łuksza, Rafał.** Should I use Headless CMS? Use Cases, Pros and Cons. *Soft Kraft.* [Online] [Citace: 19. Únor 2022.] https://www.softkraft.co/headless-cms-pros-cons/.

27. **Eastwood, Georgina.** 10 Most Popular Headless CMS of 2021. *Wiredelta.* [Online] 6. Listopad 2021. [Citace: 19. Únor 2022.] https://wiredelta.com/10-most-popular-headless-cms-of-2021/.

28. **Strapi.** What is a Headless CMS? *strapi.* [Online] 2022. [Citace: 19. Únor 2022.] https://strapi.io/what-is-headless-cms.

29. **syedmodassirali.** Client-Server Model . *GeeksforGeeks.* [Online] 15. Listopad 2019. [Citace: 27. Únor 2022.] https://www.geeksforgeeks.org/client-server-model/.

30. **Ingalls, Sam.** What Is a Client-Server Model? A Guide to Client-Server Architecture. *ServerWatch.* [Online] 17. Listopad 2021. [Citace: 27. Únor 2022.] https://www.serverwatch.com/guides/client-server-model/.

31. **Amazon Web Services.** What is DNS? *aws.* [Online] [Citace: 27. Únor 2022.] https://aws.amazon.com/route53/what-is-dns/.

32. **Teach Computer Science .** Client-Server Architecture. *Teach Computer Science.* [Online] 2022. [Citace: 27. Únor 2022.] https://teachcomputerscience.com/client-server-architecture/.

33. **OmniSci, Inc.** omni sci. *Client-Server.* [Online] 2021. [Citace: 27. Únor 2022.] https://www.omnisci.com/technical-glossary/client-server.

34. **EdPrice-MSFT, a další.** N-tier architecture style. *Microsoft.* [Online] 2. Červenec 2022. [Citace: 27. Únor 2022.] https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier.

35. **IBM Cloud Education.** Application Programming Interface (API). *IBM.* [Online] 19. Srpen 2020. [Citace: 27. Leden 2022.] https://www.ibm.com/cloud/learn/api.

36. **Red Hat, Inc.** What is a REST API? *Red Hat.* [Online] 8. Květen 2020. [Citace: 27. Leden 2022.] https://www.redhat.com/en/topics/api/what-is-a-rest-api.

37. **Avraham, Shif Ben.** What is REST — A Simple Explanation for Beginners, Part 1: Introduction. *Medium.* [Online] 5. Září 2017. [Citace: 27. Leden 2022.] https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f.

38. **Farcic, Viktor.** REST API with JSON. *Technology Conversations.* [Online] 12. Srpen 2014. [Citace: 27. Leden 2022.] https://technologyconversations.com/2014/08/12/rest-api-with-json/.

39. **Buckler, Craig.** What Is a REST API? *sitepoint.* [Online] 5. Únor 2020. [Citace: 27. Leden 2022.] https://www.sitepoint.com/rest-api/.

40. **geeksforgeeks.** REST API Architectural Constraints . *GeeksforGeeks.* [Online] 1. Červenec 2020. [Citace: 27. Leden 2022.] https://www.geeksforgeeks.org/rest-api-architectural-constraints/.

41. **Lokesh.** REST Architectural Constraints. *REST API Tutorial.* [Online] 27. Září 2021. [Citace: 27. Leden 2022.] https://restfulapi.net/rest-architectural-constraints/#client-server.

42. **Redka, Vasyl.** A Beginner's Tutorial for Understanding RESTful API. *MLSDev.* [Online] 12. Listopad 2021. [Citace: 27. Leden 2022.] https://mlsdev.com/blog/81-a-beginner-s-tutorial-for-understanding-restful-api.

43. **Levin, Guy.** 4 Most Used REST API Authentication Methods. *RestCase.* [Online] 26. Červenec 2019. [Citace: 28. Leden 2022.] https://blog.restcase.com/4-most-used-rest-api-authentication-methods/.

44. **Bui, Steven.** Securing REST APIs with Token-based Auth. *Medium.* [Online] 17. Květen 2021. [Citace: 28. Leden 2022.] https://medium.com/codex/securing-rest-apis-with-token-based-auth-5db91471dea6.

45. **Moyers, Stephen.** Common Web Design Languages, What They Do and Why You Need Them. *Spinx.* [Online] [Citace: 29. Leden 2022.]

https://www.spinxdigital.com/blog/common-web-design-languages-what-they-do-and-why-you-need-them/.

46. **Stackoverflow.** 2021 Developer Survey. *stackoverflow.* [Online] [Citace: 20. Únor 2022.] https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies.

47. **James.** Frameworks for Developing Single Page Applications in 2022. *toobler.* [Online] 25. Leden 2022. [Citace: 20. Únor 2022.] https://www.toobler.com/blog/frameworks-for-developing-single-page-applications.

48. **Mozilla and individual contributors.** What is JavaScript? *MDN Web Docs.* [Online] [Citace: 20. Únor 2022.] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

49. **Brewster, Cordenne.** 11 Best Front-End Technologies To Use In 2022. *Trio.* [Online] [Citace: 20. Únor 2022.] https://trio.dev/blog/front-end-technologies.

50. **Mozilla and individual contributors.** Introduction to the server side. *MDN Web Docs.* [Online] [Citace: 20. Únor 2022.] https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction.

51. **StrongLoop, IBM, and other expressjs.com contributors.** Using template engines with Express. *Express.* [Online] [Citace: 20. Únor 2022.] https://expressjs.com/en/guide/using-template-engines.html.

52. **Refsnes Data.** Introduction to SQL. *w3schools.* [Online] [Citace: 20. Únor 2022.] https://www.w3schools.com/sql/sql_intro.asp.

53. **Mozilla and individual contributors.** Introduction. *MDN Web Docs.* [Online] [Citace: 29. Leden 2022.] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction.

54. **lakshita.** Introduction to JavaScript. *GeeksForGeeks.* [Online] 26. Říjen 2021. [Citace: 29. Leden 2022.] https://www.geeksforgeeks.org/introduction-to-javascript/.

55. **Pavey, Cameron.** Understanding Types; Static vs Dynamic, & Strong vs Weak. *Medium.* [Online] 24. Únor 2019. [Citace: 29. Leden 2022.] https://medium.com/@cpave3/understanding-types-static-vs-dynamic-strong-vs-weak-88a4e1f0ed5f.

56. **Dubey, Bishal Kumar.** Difference between TypeScript and JavaScript. *GeeksForGeeks.* [Online] 31. Srpen 2021. [Citace: 29. Leden 2022.] https://www.geeksforgeeks.org/difference-between-typescript-and-javascript/.

57. **Codecademy Team.** What Is a Framework? *codeacademy.* [Online] 23. Září 2021. [Citace: 20. Únor 2022.] https://www.codecademy.com/resources/blog/what-is-a-framework/.

58. **Dhaduk, Hiren.** Top 15 JavaScript Frameworks You Should Consider in 2022. *Simform.* [Online] 2. Prosinec 2021. [Citace: 20. Únor 2022.] https://www.simform.com/blog/javascript-frameworks/.

59. **Microsoft.** volba mezi ASP.NET 4. x a ASP.NET Core. *Microsoft.* [Online] 7. Únor 2022. [Citace: 20. Únor 2022.] https://docs.microsoft.com/cs-cz/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-6.0.

60. **Singh, Vijay.** Best Python Frameworks. *hackr.io.* [Online] 18. Únor 2022. [Citace: 20. Únor 2022.] https://hackr.io/blog/python-frameworks.

61. **Goel, Aman.** 10 Best Web Development Frameworks. *hackr.io.* [Online] 18. Únor 2022. [Citace: 20. Únor 2022.] https://hackr.io/blog/web-development-frameworks.

62. **Refsnes Data.** Node.js Introduction. *W3schools.com.* [Online] [Citace: 27. Leden 2022.] https://www.w3schools.com/nodejs/nodejs_intro.asp.

63. **OpenJS Foundation.** Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. *nodejs.* [Online] [Citace: 27. Leden 2022.] https://nodejs.org/en/.

64. —. The V8 JavaScript Engine. *nodejs.* [Online] [Citace: 27. Leden 2022.] https://nodejs.dev/learn/the-v8-javascript-engine.

65. **tutorialspoint.** Node.js - Event Loop. *tutorialspoint.* [Online] [Citace: 27. Leden 2022.] https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm.

66. **geeksforgeeks.** Blocking and Non-Blocking in Node.js. *GeeksforGeeks.* [Online] 21. Květen 2021. [Citace: 27. Leden 2022.] https://www.geeksforgeeks.org/blocking-and-non-blocking-in-node-js/.

67. **Reales, Andres.** Is Node.js Single-Threaded or Multi-Threaded? and Why? *DEV.* [Online] 7. Září 2021. [Citace: 27. Leden 2022.] https://dev.to/arealesramirez/is-node-js-single-threaded-or-multi-threaded-and-why-ab1.

68. **Roberts, Philip.** Philip Roberts: What the heck is the event loop anyway? *JSCONF.EU.* [Online] 13. Září 2014. [Citace: 27. Leden 2022.] https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html.

69. **OpenJS Foundation GitHub.** Introduction to Node.js. *nodejs.* [Online] [Citace: 27. Leden 2022.] https://nodejs.dev/learn.

70. **AltexSoft.** The Good and the Bad of Node.js Web App Development. *altexsoft.* [Online] 21. Říjen 2019. [Citace: 22. Leden 2022.] https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-node-js-web-app-development/.

71. **SuperHosting.BG.** I/O (Input/Output Operations). *SuperHosting.BG.* [Online] 2. Červenec 2019. [Citace: 27. Leden 2022.] https://help.superhosting.bg/en/what-is-io-iops.html.

72. **Mozilla and individual contributors.** Express/Node introduction. *MDN web docs.* [Online] [Citace: 27. Leden 2022.] https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction.

73. **Dhaduk, Hiren.** Node.JS Use Case: When & How Node.JS Should be Used. *SIMFORM.* [Online] 27. Říjen 2019. [Citace: 27. Leden 2022.] https://www.simform.com/nodejs-use-case/.

74. **MongoDB, Inc.** Introduction to MongoDB. *MongoDB.* [Online] [Citace: 27. Leden 2022.] https://docs.mongodb.com/manual/introduction/.

75. **Kenny, Colm.** Web scraping. *Zyte.* [Online] [Citace: 27. Leden 2022.] https://www.scrapinghub.com/what-is-web-scraping/.

76. **nishanil a olprod.** Monolitické aplikace. *Microsoft.* [Online] 14. Prosinec 2021. [Citace: 28. Leden 2022.] https://docs.microsoft.com/cs-cz/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications.

77. **Haq, Siraj ul.** Introduction to Monolithic Architecture and MicroServices Architecture. *Medium.* [Online] 2. Květen 2018. [Citace: 28. Leden 2022.] https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

78. **Frankenfield, Jake a Estevez, Eric.** Business Logic. *Investopedia.* [Online] 28. Srpen 2020. [Citace: 28. Leden 2022.] https://www.investopedia.com/terms/b/businesslogic.asp.

79. **Kotia, Neeti.** Monolithic vs Microservices: Which Web App Architecture to Prefer? *Konstant Infosolutions.* [Online] 26. Květen 2021. [Citace: 28. Leden 2022.]

https://www.konstantinfo.com/blog/monolithic-vs-microservices-which-architecture-prefer/.

80. **Karwatka, Piotr.** Monolithic architecture vs microservices. *divante.* [Online] 14. Leden 2020. [Citace: 28. Leden 2022.] https://www.divante.com/blog/monolithic-architecture-vs-microservices.

81. **Newizze.** Microservices vs Monolithic Architecture: Which is Right for You? *newizze.* [Online] [Citace: 28. Leden 2022.] https://newizze.com/microservices-vs-monolithic-architecture-which-is-right-for-you/.

82. **nishanil a olprod.** Komunikace v architektuře mikroslužeb. *Microsoft.* [Online] 27. Září 2021. [Citace: 29. Leden 2022.] https://docs.microsoft.com/cs-cz/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

83. **Newman, Sam.** *Building Microservices: Designing Fine-Grained Systems.* Newton (Massachusetts) : O'Reilly Media, 2015. ISBN 978-1491950357.

84. **Reselman, Bob.** Synchronous vs. asynchronous microservices communication patterns. *TheServerSide.* [Online] 10. Březen 2021. [Citace: 29. Leden 2022.] https://www.theserverside.com/answer/Synchronous-vs-asynchronous-microservices-communication-patterns.

85. **SPEC INDIA.** Top 15 Kafka Alternatives Popular In 2021 . *Spec India.* [Online] 14. Září 2021. [Citace: 29. Leden 2022.] https://www.spec-india.com/blog/kafka-alternatives.

86. **tutorialspoint.** Microservice Architecture - Introduction. *tutorialspoint.* [Online] [Citace: 29. Leden 2022.] https://www.tutorialspoint.com/microservice_architecture/microservice_architecture_introduction.htm.

87. **raman_257.** Monolithic vs Microservices architecture. *GeeksForGeeks.* [Online] 17. Srpen 2021. [Citace: 29. Leden 2022.] https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/.

88. **ILANY, RAN.** Microservices vs. Monoliths: Which is Right for Your Enterprise? *DevOps.com.* [Online] 26. Červen 2020. [Citace: 29. Leden 2022.] https://devops.com/microservices-vs-monoliths-which-is-right-for-your-enterprise/.

89. **Kwan, Zac.** But, it works on my machine. *Hackernoon.* [Online] 3. Květen 2018. [Citace: 29. Leden 2022.] https://hackernoon.com/but-it-works-on-my-machine-74b6875ab4e7.

90. **Docker.** Docker overview. *docker docs.* [Online] [Citace: 29. Leden 2022.] https://docs.docker.com/get-started/overview/.

91. **Arvind.** Docker Architecture: Why is it important? *edureka!* [Online] 26. Listopad 2019. [Citace: 29. Leden 2022.] https://www.edureka.co/blog/docker-architecture/.

92. **VMware, Inc.** What is a hypervisor? . *vmware.* [Online] [Citace: 29. Leden 2022.] https://www.vmware.com/topics/glossary/content/hypervisor.html.

93. **mberezin.** What is Docker and How Does it Work? *University of Massachusetts Amherst.* [Online] 9. Říjen 2018. [Citace: 29. Leden 2022.] https://blogs.umass.edu/Techbytes/2018/10/09/what-is-docker-and-how-does-it-work/.

94. **Docker Inc.** The Industry-Leading Container Runtime . *docker.* [Online] [Citace: 27. Únor 2022.] https://www.docker.com/products/container-runtime.

95. **Aqua Security Software Ltd.** Docker Architecture. *cloud native wiki.* [Online] [Citace: 29. Leden 2022.] https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/.

96. **The Kubernetes Authors.** What is Kubernetes? . *kubernetes.* [Online] 23. Červenec 2021. [Citace: 29. Leden 2022.] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

97. **VMware, Inc.** vmware. *What is Kubernetes?* [Online] [Citace: 29. Leden 2022.] https://www.vmware.com/topics/glossary/content/kubernetes.html.

98. **Leon, Freddy Tapia a Saransig, Alexis.** Research Gate. *Performance Analysis of Monolithic and Micro Service Architectures – Containers Technology: Proceedings of the 7th International Conference on Software Process Improvement.* [Online] January 2019. [Citace: 22. Leden 2022.] https://www.researchgate.net/publication/327918047_Performance_Analysis_of_Monolithic_and_Micro_Service_Architectures_-_Containers_Technology_Proceedings_of_the_7th_International_Conference_on_Software_Process_Improvement_CIMPS_2018.

99. **Flygare, Robin a Holmqvist, Anthon.** Performance characteristics between monolithic and microservice-based systems. *Diva portal.* [Online] 06 2017. [Citace: 22. Leden 2022.] https://www.diva-portal.org/smash/get/diva2:1119785/FULLTEXT03.pdf.

100. **Lejček, Miloslav.** Využití microservices architektury pro automatizaci firemních procesů. *Univerzitní informační systém.* [Online] 2021. [Citace: 22. Leden 2022.] https://is.czu.cz/zp/portal_zp.pl?podrobnosti_zp=276655;zpet=;prehled=pracoviste;pracoviste=121;dohledat=Dohledat;typ=1;typ=2;typ=3;obhajoba=2022;obhajoba=2021;obhajoba=2020;jazyk=1;jazyk=3;lang=cz.

101. **Apu, Md Foyjur Rahman.** Microservices and Serverless Architecture in Web Applications. *Univerzitní informační systém.* [Online] 2020. [Citace: 22. Leden 2022.] https://is.czu.cz/zp/portal_zp.pl?podrobnosti_zp=254761;zpet=;prehled=pracoviste;pracoviste=121;dohledat=Dohledat;typ=1;typ=2;typ=3;obhajoba=2022;obhajoba=2021;obhajoba=2020;jazyk=1;jazyk=3;lang=cz.

102. **Masner, Jan.** OBSAZENOST AREÁLU ČZU. *Life Sciences 4.0.* [Online] [Citace: 22. Leden 2022.] http://ls40.pef.czu.cz/obsazenost-arealu-czu.

103. **Katedra informačních technologií.** Wolno. *Wolno.* [Online] [Citace: 14. Únor 2022.] http://wolno.pef.czu.cz.

104. **Geers, Michael.** Extending the microservice idea to frontend development. *Micro Frontends.* [Online] [Citace: 22. Leden 2022.] https://micro-frontends.org.

105. **Docker Inc.** Overview of Docker Compose. *docker docs.* [Online] [Citace: 26. Leden 2022.] https://docs.docker.com/compose/.

106. **kubernetes / ingress-nginx.** Overview. *NGINX Ingress Controller.* [Online] [Citace: 30. Leden 2022.] https://kubernetes.github.io/ingress-nginx/.

107. **The Kubernetes Authors.** Ingress. *Kubernetes.* [Online] [Citace: 30. Leden 2022.] https://kubernetes.io/docs/concepts/services-networking/ingress/.

108. **nishanil, a další.** Microsoft. *Communication in a microservice architecture.* [Online] 2022. [Citace: 24. Leden 2022.] https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

109. **Bohmann, David.** Diplomová práce. *Srovnání implementací message brokerů.* [Online] 2020. [Citace: 2022. Leden 24.] https://dspace5.zcu.cz/bitstream/11025/41755/1/Bohmann_David_2020_DP.pdf.

110. **VMware.** Publish/Subscribe. *RabbitMQ.* [Online] [Citace: 24. Leden 2022.] https://www.rabbitmq.com/tutorials/tutorial-three-python.html.

111. —. Routing. *RabbitMQ.* [Online] [Citace: 24. Leden 2022.] https://www.rabbitmq.com/tutorials/tutorial-four-python.html.

112. —. Work Queues. *RabbitMQ.* [Online] [Citace: 24. Leden 2022.] https://www.rabbitmq.com/tutorials/tutorial-two-javascript.html.

113. **Strugo, Tzachi.** Authentication & Authorization in Microservices Architecture - Part I. *DEV.* [Online] 14. Únor 2021. [Citace: 24. Leden 2022.] https://dev.to/behalf/authentication-authorization-in-microservices-architecture-part-i-2cn0.

114. **Aggarwal, Bhavya.** Authentication and Authorization in Microservices. *DZone.* [Online] 19. Březen 2019. [Citace: 24. Leden 2022.] https://dzone.com/articles/authentication-and-authorization-in-microservices.

115. **Ayoub, Mina.** Microservices Authentication and Authorization Solutions. *medium.* [Online] 24. Duben 2018. [Citace: 24. Leden 2022.] https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a.

116. **Besic, Nera.** Microservices Security: Challenges and Best Practices. *NeuraLegion.* [Online] 28. Květen 2021. [Citace: 24. Leden 2022.] https://www.neuralegion.com/blog/microservices-security/.

117. **Orner, Daniel.** Schema and Topic Design in Event-Driven Systems (featuring Kafka!). *Medium.* [Online] 1. Květen 2020. [Citace: 25. Leden 2022.] https://medium.com/flippengineering/schema-and-topic-design-in-event-driven-systems-featuring-kafka-a555ddfdb8d8.

118. **Fernando, Ashan.** The Dilemma of Code Reuse in Microservices . *Medium.* [Online] [Citace: 25. Leden 2022.] https://blog.bitsrc.io/the-dilemma-of-code-reuse-in-microservices-a925ff2b9981.

119. **Kubala, Nick a Wolf, Russel.** Kubernetes development, simplified—Skaffold is now GA. *Google Cloud.* [Online] 7. Listopad 2019. [Citace: 26. Leden 2022.] https://cloud.google.com/blog/products/application-development/kubernetes-development-simplified-skaffold-is-now-ga.

120. **ManagementMania.com.** Člověkoden (Man-day). *Managementmania.* [Online] 11. Říjen 2018. [Citace: 9. Únor 2022.] https://managementmania.com/cs/clovekoden-manday.

121. **Despa, Valentin.** API Load-Testing / Performance-Testing with Postman. Does it Really Work? *Medium.* [Online] 14. Říjen 2021. [Citace: 26. Leden 2022.] https://medium.com/apis-with-valentine/api-load-testing-performance-testing-with-postman-does-it-really-work-1a2910c69c03.

122. **Portainer.** A centralized service delivery platform for containerized apps. *portainer.io.* [Online] 2021. [Citace: 26. Leden 2022.] https://www.portainer.io/.

123. **The Kubernetes Authors.** Deploy and Access the Kubernetes Dashboard. *kubernetes.* [Online] 31. Prosinec 2021. [Citace: 26. Leden 2022.] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

124. **SOFTWARETESTINGHELP.** JMeter Listeners: Analyzing Results With Different Listeners. *Software Testing Help.* [Online] 4. Leden 2022. [Citace: 25. Leden 2022.] https://www.softwaretestinghelp.com/jmeter-listeners/.

125. **Váš Hosting s.r.o.** váš hosting. *Tarify virtuálních serverů .* [Online] 2021. [Citace: 26. Leden 2022.] https://www.vas-hosting.cz/ceny-serveru.

126. **WEDOS Internet, a. s.** Virtuální privátní servery s SSD disky! . *Wedos.* [Online] 2022. [Citace: 26. Leden 2022.] https://www.wedos.cz/vps-ssd.

127. **THINline s.r.o.** Virtuální servery. *český hosting.* [Online] [Citace: 26. Leden 2022.] https://www.cesky-hosting.cz/servery/virtualni-servery/.