Moderní architektury webových aplikací s mikrofrontendy

Bc. Filip Nesteš

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Filip Nesteš**
Osobní číslo:	**A22494**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Moderní architektury webových aplikací s mikrofrontendy**
Téma práce anglicky:	**Modern Web Applications Architectures With Microfrontends**

Zásady pro vypracování

1. Prostudujte architektonický koncept mikrofrontendů u moderních web aplikací.
2. Popište výhody a nevýhody tohoto konceptu ve srovnání s jinými architekturami.
3. Zpracujte teoretické a technické řešení alternativních přístupů, kterými lze mikrofrontendy nahradit.
4. Implementujte ukázkový projekt s mikrofrontendy, který bude obsahovat několik menších částí zpracovaných v různých JavaScriptových knihovnách.
5. Implementaci popište a vyhodnoťte, za jakých podmínek je výhodné investovat čas a prostředky do výstavby mikrofrontendové architektury.

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

Seznam doporučené literatury:

1. RAPPL, Florian. The Art of Micro Frontends: Build websites using compostional UIs that grow naturally as your application scales. Packt Publishing. 2021. ISBN 978-1800563568.
2. GEERS, Michael. Micro Frontends in Action. Manning. 2020. ISBN 9781617296871.
3. MEZZALIRA Luca. Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers. O'Reilly Media. 2021. ISBN 978-1492082996.
4. RUFUS, Vinci. Building Micro Frontends with React 18: Develop and deploy scalable applications using micro frontend strategies. Packt Publishing. 2023. ISBN 978-1804610961.
5. GHIDERSA, Michaela Roxana. Software Architecture for Web Developers: An introductory guide for developers striving to take the first steps toward software architecture or just looking to grow as professionals. Packt Publishing. 2022. ISBN 978-1803237916.

Vedoucí diplomové práce: **Ing. Tomáš Dulík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

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

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

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

Ve Zlíně dne 5. ledna 2024

Prehlasujem, že

- beriem na vedomie, že odovzdaním diplomovej práce súhlasím so zverejnením svojej práce podľa zákona č. 111/1998 Zb, o vysokých školách a o zmene a doplnení ďalších zákonov (zákon o vysokých školách), v znení neskorších právnych predpisov, bez ohľadu na výsledok obhajoby;
- beriem na vedomie, že diplomová práca bude uložená v elektronickej podobe v univerzitnom informačnom systéme dostupná k prezenčnému nahliadnutiu, že jeden výtlačok diplomovej práce bude uložený v príručnej knižnici Fakulty aplikovanej informatiky Univerzity Tomáša Baťu v Zlíne;
- bol som oboznámený s tým, že na moju diplomovú prácu sa vzťahuje zákon č. 121/2000 Sb, o autorskom práve, o právach súvisiacich s autorským právom a o zmene niektorých zákonov (autorský zákon) v znení neskorších právnych predpisov, a to § 35 odst. 3;
- beriem na vedomie, že podľa § 60 odst. 1 autorského zákona má UTB v Zlíne právo na uzavretie licenčnej zmluvy o použití školského diela v rozsahu § 12 odst. 4 autorského zákona;
- beriem na vedomie, že podľa § 60 odst. 2 a 3 autorského zákona môžem použiť svoje dielo – diplomovú prácu alebo poskytnúť licenciu k jej využitiu len ak to dovoľuje licenčná zmluva uzavretá medzi mnou a Univerzitou Tomáša Baťu v Zlíne s tým, že vyrovnanie prípadného primeraného príspevku na úhradu nákladov, ktoré boli Univerzitou Tomáša Baťu v Zlíne na vytvorenie diela vynaložené (až do ich skutočnej výšky) bude taktiež predmetom tejto licenčnej zmluvy;
- beriem na vedomie, že pokiaľ bol k vypracovaniu diplomovej práce použitý software poskytnutý Univerzitou Tomáša Baťu v Zlíne alebo inými subjektmi len k študijným a výskumným účelom (teda len k nekomerčnému využitiu), nie je možné výsledky diplomovej práce použiť na komerčné účely;
- beriem na vedomie, že pokiaľ je výstupom diplomovej práce akýkoľvek softwarový produkt, považujú sa za súčasť práce rovnako aj zdrojové kódy, prípadne súbory, z ktorých sa projekt skladá. Neodovzdanie tejto súčasti môže byť dôvodom k neobhájeniu práce.

Prehlasujem,

- že som na diplomovej práci pracoval samostatne a použitú literatúru som citoval. V prípade publikácie výsledkov budem uvedený ako spoluautor.
- že odovzdaná verzia diplomovej práce a elektronická verzia nahraná do IS/STAG sú totožné.

V Zlíne, dňa Bc. Filip Nesteš v.r.

ABSTRAKT

Cieľom tejto diplomovej práce je oboznámiť sa s technológiou mikrofrontendov a poskytnúť hlboký prehľad o možnostiach, ktoré mikrofrontendy ponúkajú, vrátane ich výhod a výziev. Získané informácie chceme následne demonštrovať a prakticky zapracovať v konkrétnom projekte. V teoretickej časti si objasníme koncepcie mikrofrontendov, ich historický vývoj od tradičných monolitických architektúr až po dekomponované riešenia, a rozoberieme si rôzne stratégie implementácie týchto architektúr. Práca nám priblíži aj technológie, ktoré v súčasnosti hýbu svetom vývoja webových aplikácií. Praktická časť je zameraná na návrh, vývoj a integráciu mikrofrontendov v projekte, ktorý kombinuje rôzne JavaScriptové rámce ako React, Angular, Vue a Svelte a ilustruje, ako tieto technológie môžu efektívne spolupracovať.

Kľúčové slova: mikrofrontendy, webové aplikácie, softvérová architektúra, JavaScript, React, Angular, Vue, Svelte, modulárny vývoj, škálovateľnosť

ABSTRACT

The aim of this thesis is to become familiar with the technology of micro frontends and to provide a deep overview of the possibilities that micro frontends offer, including their advantages and challenges. We want to subsequently demonstrate and practically implement the acquired information in a specific project. In the theoretical part, we will clarify the concepts of micro frontends, emphasizing their historical development from traditional monolithic architectures to decomposed solutions and we will also discuss various strategies for implementing these architectures. The thesis will also introduce technologies that are currently driving the development of web applications. The practical part is focused on designing, developing, and integrating micro frontends in a project that combines various JavaScript frameworks like React, Angular, Vue, and Svelte, illustrating how these technologies can effectively cooperate.

Keywords: microfrontends, web applications, software architecture, JavaScript, React, Angular, Vue, Svelte, modular development, scalability.

Chcel by som poďakovať vedúcemu diplomovej práce Ing. Tomášovi Dulíkovi, Ph.D. za odbornú pomoc, pripomienky a usmerňovanie pri tvorbe práce. Taktiež by som chcel poďakovať všetkým, ktorý mi svojimi radami a tipmi pomohli k dokončeniu tejto práce.

OBSAH

ÚVOD

V súčasnosti sa svet informačných technológií vyvíja mimoriadne rýchlym tempom, čo prináša stále nové výzvy v oblasti vývoja softvéru. Zvyšujúce sa požiadavky na flexibilitu, rýchlosť nasadzovania aktualizácií a škálovateľnosť aplikácií tlačia spoločnosti k hľadaniu inovatívnych prístupov k architektúre softvéru. V tomto dynamickom prostredí sa mikrofrontendy javia ako sľubné riešenie pre zefektívnenie vývoja a správy rozsiahlych webových aplikácií. Táto práca sa preto bude venovať podrobnej analýze a praktickej aplikácie mikrofrontendovej architektúry, ktorá umožňuje dekompozíciu klientskej časti webových aplikácií do menších, nezávisle fungujúcich častí.

Cieľom tejto práce je poskytnúť komplexný pohľad na mikrofrontendy ako na metódu organizácie webového vývoja.

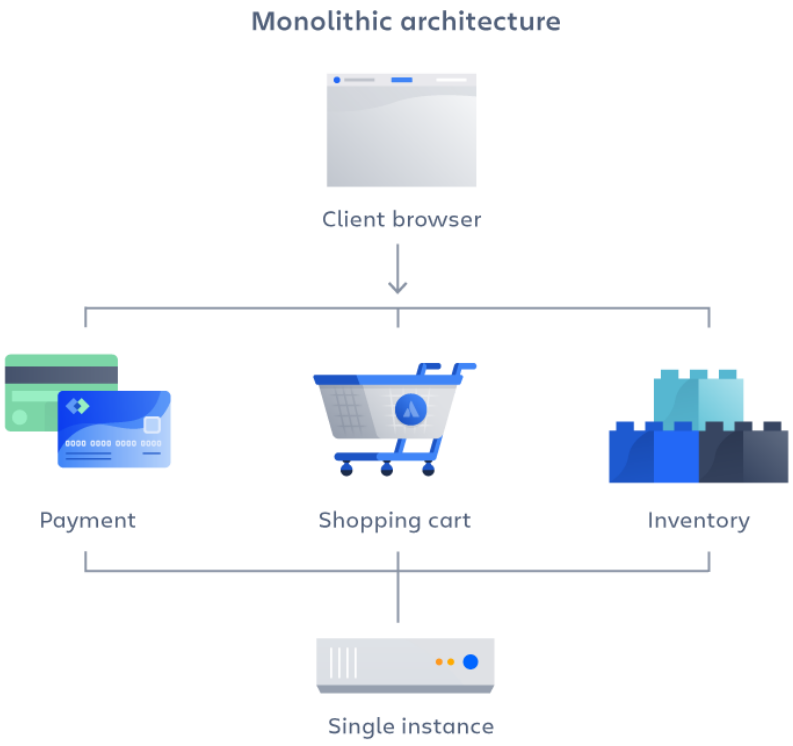
V teoretickej časti práce sa budeme zaoberať kľúčovými konceptmi a históriou vývoja softvérových architektúr, pričom špeciálny dôraz bude kladený na evolúciu od tradičných monolitických štruktúr až po moderné dekomponované riešenia. Budeme skúmať výhody a výzvy spojené s adopciou mikrofrontendov, ako aj rôzne stratégie ich implementácie. Priblížime si aj najmodernejšie technológie s ktorými sa pri vývoji modrených webových aplikácií môžeme stretnúť.

V praktickej časti sa sústredíme na návrh a implementáciu mikrofrontendov na konkrétnom projekte. Ukážeme, ako je možné integrovať rôzne JavaScriptové rámce, ako sú React, Angular, Vue a Svelte, do jednotného systému a ako môžu tieto technologicky rozdielne časti spolupracovať v rámci jedného projektu. Praktické ukážky nám poskytnú názorné príklady riešení, ktoré ilustrujú, ako mikrofrontendy prispievajú k lepšej modularite, zjednodušeniu procesov a zvýšenej efektivite celkového vývoja.

I. TEORETICKÁ ČASŤ

1 MONOLITICKÁ ARCHITEKTÚRA

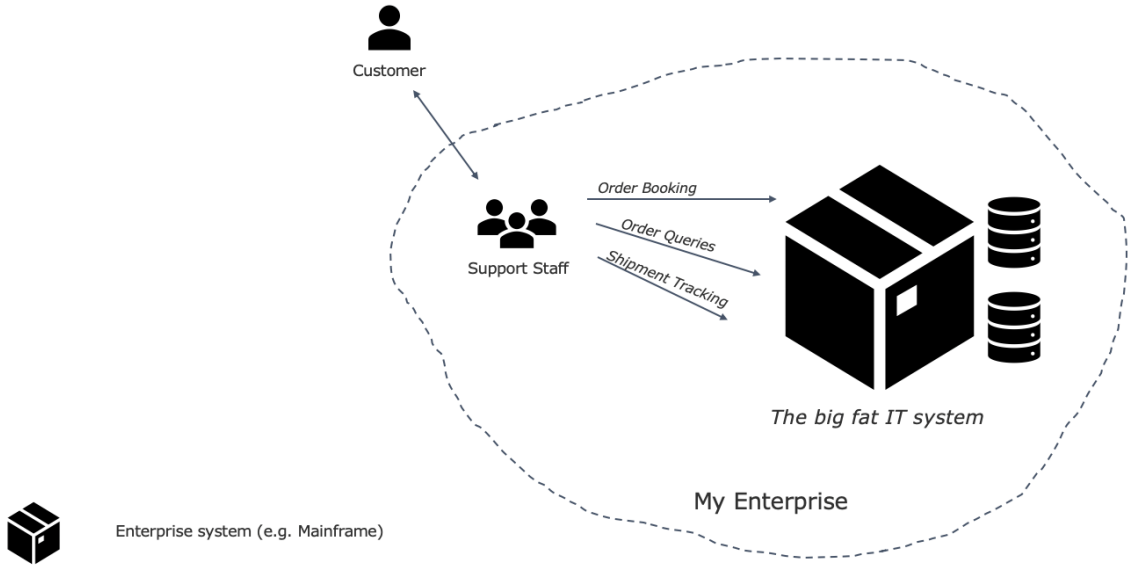
Monolitická architektúra je tradičný model softvérového programu, ktorý je postavený ako jednotný celok, samostatný a nezávislý od iných aplikácií. Slovo "monolit" je často pripisované niečomu veľkému a pomalému, čo nie je ďaleko od pravdy. Monolitická architektúra je jednotná, veľká výpočtová sieť s jednou základňou kódu, ktorá spája všetky obchodné záležitosti dohromady. Urobiť zmenu v takomto type aplikácie vyžaduje aktualizáciu celého zásobníka prístupom k základnému kódu a vytvorením a nasadením aktualizovanej verzie serverového rozhrania. To robí z aktualizácií obmedzujúce a časovo náročné operácie. Monolity sa môžu na začiatku projektu zdať byť omnoho pohodlnejšie z dôvodu ľahšej správy kódu, kognitívnej záťaže a nasadzovania. To umožňuje aby všetky súčasti monolitickej aplikácie boli nasadené súčasne. [1]



Obrázok 1: Príklad ako vyzerá monolitická architektúra [1]

1.1 História

Na začiatku boli monolitické aplikácie, ktoré by sme mohli charakterizovať ako napríklad systémy na spracovanie objednávok, cenotvorby, zásielok a podobne. Takéto monolity môžeme stále nájsť aj v súčasnosti pri projektoch kde nepotrebujeme žiadne sieťové volania, len jednoduchú komunikáciu medzi procesmi, poprípade spájanie databázových tabuliek do takzvaného „brokera“. Dobrým príkladom takýchto projektov sú startupy.



Obrázok 2: Dedičná monolitická aplikácia [2]

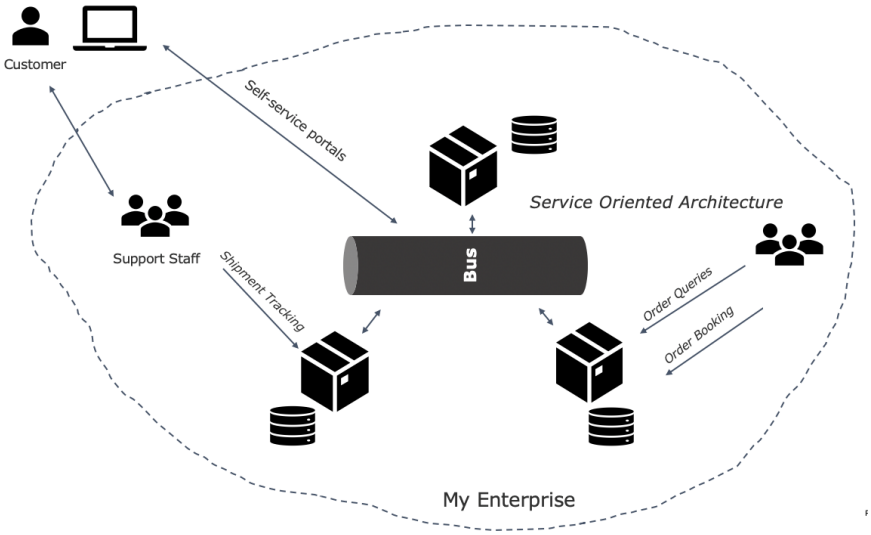
1.1.1 Rok 2000 a aplikácie MVC

Model-View-Controller (MVC) je návrhový vzor používaný pri vývoji softvéru, ktorý rozdeľuje aplikáciu na tri hlavné logické komponenty: Model, View a Controller. Tento vzor je navrhnutý tak, aby oddelil vnútornú reprezentáciu informácií (Model) od spôsobu, akým informácie prezentuje užívateľovi (View), s Controllerom ako medzičlánkom, ktorý spracováva vstupné dáta od užívateľa a transformuje ich na príkazy pre Model alebo View.

- Model predstavuje štruktúru dát, logiku a funkcie aplikácie. V podstate obsahuje všetky informácie a dáta, s ktorými aplikácia pracuje, ako aj pravidlá, podľa ktorých sa tieto dáta spracúvajú alebo menia.
- View je komponent, ktorý zodpovedá za zobrazenie dát alebo informácií užívateľovi. Môže existovať viacero pohľadov, ktoré predstavujú rôzne spôsoby, ako môžu byť dáta prezentované.

- Controller prijíma vstupy od užívateľa a rozhoduje o tom, ako má byť na vstupy reagované. To môže zahŕňať zmenu dát v Modeli alebo zmenu toho, ako sú dáta zobrazené vo View.

MVC umožňuje efektívne oddelenie zodpovedností v aplikácii, čo vedie k lepšej organizácii kódu, jednoduchšiemu manažmentu a údržbe softvéru a zjednodušuje spoluprácu v tímoch tým, že umožňuje, aby na rôznych častiach aplikácie pracovali rôzni ľudia nezávisle. Tento vzor bol obzvlášť populárny v počiatočných fázach vývoja webových aplikácií a je stále široko používaný v mnohých programovacích rámcoch a aplikáciách.



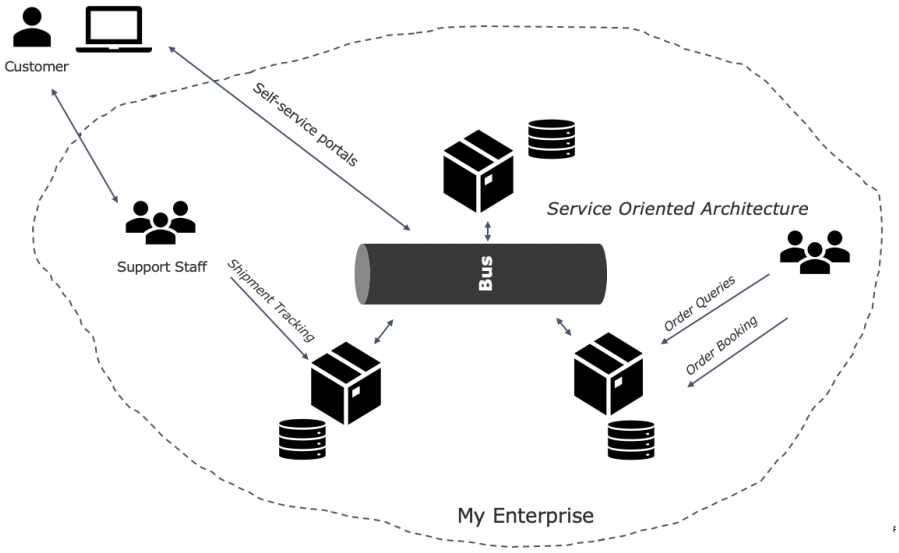
Obrázok 3: MVC aplikácia a dedičný monolit [2]

1.1.2 Architektúra orientovaná na služby

Tento prístup vznikol ako reakcia na rastúcu potrebu štandardizácie dát a komunikácie medzi rastúcim počtom vlastných aplikácií v rámci organizácií. Hlavnými bodmi SOA sú:

- Potreba štandardizácie a spoločného jazyka: S rastúcim počtom vlastných aplikácií vznikla potreba jednotnejšieho prístupu k popisu a spracovaniu podnikových dát a procesov. SOA reaguje na túto potrebu tým, že poskytuje spoločný jazyk a štandardy pre definovanie a využívanie podnikových služieb.
- Dekuplované služby: SOA propaguje vývoj služieb, ktoré sú navzájom oddelené (dekuplované), čo znamená, že môžu byť vyvíjané, nasadzované a spravované nezávisle. To umožňuje väčšiu flexibilitu a agilitu v procese vývoja softvéru.

- XML a správy založené na štandardoch JMS: V rámci SOA sa organizácie snažili vyvíjať vlastné riešenia založené na XML a správach, ktoré využívali štandardy JMS (Java Messaging Service) pre komunikáciu medzi službami. Tento prístup umožnil štandardizovanú a efektívnu výmenu dát medzi službami.
- Vznik a ciele SOA: SOA bola prijatá s ambicióznymi cieľmi, ako sú vývoj kanonických podnikových dátových modelov, zníženie závislosti služieb na konkrétnych implementáciách (napr. Java aplikácie závislé na službách, ktoré využívajú), zavedenie štandardizovanej bezpečnosti a vytvorenie registra služieb.
- Produkty podporujúce SOA: Napriek inovatívnemu prístupu SOA k dekuplácii a štandardizácii služieb, mnohé produkty, ktoré SOA podporovali, boli v skutočnosti monolitické systémy, ktoré bežali na aplikačných serveroch s jednotnou databázou. Tieto systémy sa snažili o virtualizáciu služieb a smerovanie založené na správach ale často boli obmedzené na transformáciu a smerovanie dát bez schopnosti riešiť komplexnejšie integračné výzvy. [2]



Obrázok 4: Príklad architektúry v modernom podniku [2]

1.2 Výhody a nevýhody monolitickej architektúry

1.2.1 Výhody monolitickej architektúry

To či organizáciám bude viacej vyhovovať používanie monolitickej alebo mikroslužbovej architektúry závisí od rôznych faktorov. Pri vývoji pomocou monolitickej architektúry je hlavnou výhodou rýchlosť vývoja vďaka jednoduchosti aplikácie, ktorá je založená na jednom základnom kóde.

- Jednoduché nasadenie – jeden vykonateľný súbor alebo adresár uľahčuje nasadenie.
- Vývoj – ak je aplikácia založená na jednom základnom kóde je ľahšie ju vyvíjať.
- Výkon – v centralizovanom kóde a repozitári môže jedno API často vykonávať rovnakú funkciu, ktorú vykonáva viacero API s mikroslužbami.
- Zjednodušené testovanie – keďže monolitická aplikácia je jednoliata, centralizovaná jednotka tak end-to-end testovanie môže byť vykonané rýchlejšie než s distribuovanou aplikáciou.
- Jednoduché ladenie – všetok kód umiestnený na jednom mieste takže je ľahšie sledovať požiadavku a nájsť problém.

1.2.2 Nevýhody monolitickej architektúry

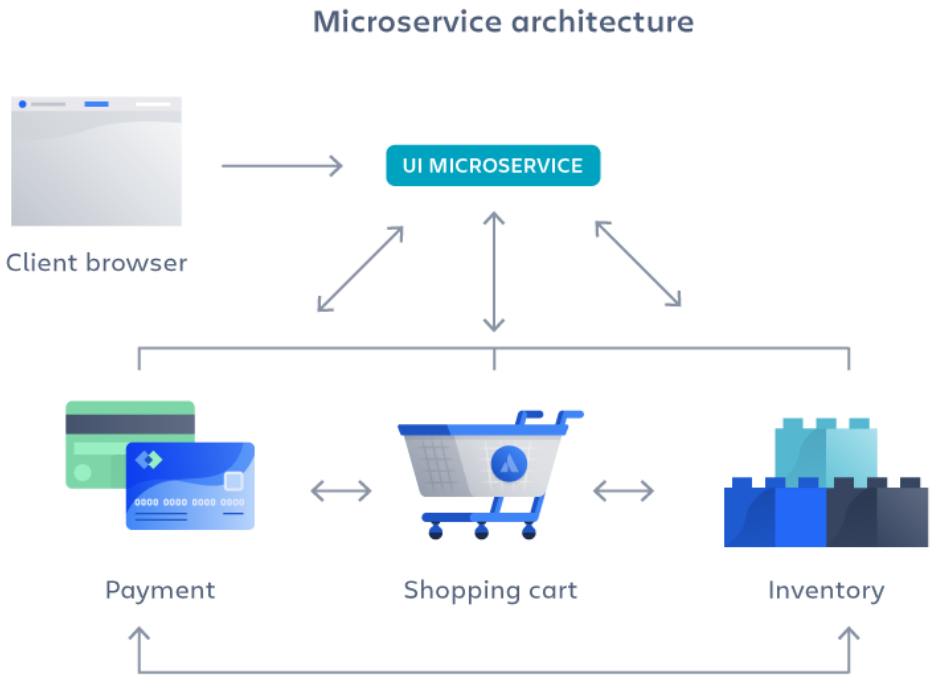
Monolitické aplikácie môžu byť veľmi účinné, až do momentu kým nevyrastú do príliš veľkých aplikácií a škálovanie sa stane pre vývoj doslova výzvou. Tento scenár v minulosti postihol viacerých veľkých technologických gigantov ako napríklad Netflix. A to preto, že snaha o urobenie aj malej zmeny v jednej funkcii si vyžaduje kompiláciu a testovanie celej platformy čo je v rozpore s agilným prístupom, ktorý dnešní vývojári uprednostňujú.

- Pomalšia rýchlosť vývoja – veľká, monolitická aplikácia robí vývoj zložitejším a pomalším
- Škálovateľnosť – nie je možné škálovať jednotlivé komponenty.
- Spoľahlivosť – ak je chyba v akomkoľvek module, môže to ovplyvniť dostupnosť celej aplikácie.

- Bariéra pre prijatie technológie – akékoľvek zmeny v rámca alebo jazyku ovplyvňujú celú aplikáciu čo robí zmeny často finančne drahšími a časovo náročnými.
- Nedostatok flexibility – monolit je obmedzený technológiami, ktoré už táto monolitická aplikácia používa.
- Nasadenie – aj malá zmena v monolitickej aplikácii si vyžaduje opätovné nasadenie celého monolitu [1]

2 MIKROSLUŽBY

Architektúra mikroslužieb, známa aj jednoducho ako mikroslužby je architektonická metóda, ktorá sa spolieha na sériu nezávisle nasaditeľných služieb. Tieto služby majú vlastnú obchodnú logiku a databázu s konkrétnym účelom. Aktualizácie, testovanie, nasadzovanie a škálovanie prebiehajú v rámci každej služby samostatne. Mikroslužby rozdeľujú hlavné obchodné, doménovo špecifické záležitosti do samostatných, nezávislých zdrojových kódov. Mikroslužby neznižujú komplexnosť ale robia akúkoľvek zložitú komplexnosť čitateľnejšou a lepšie zvládnuteľnou tým, že rozdeľujú úlohy na menšie procesy, ktoré fungujú nezávisle od seba. [1]

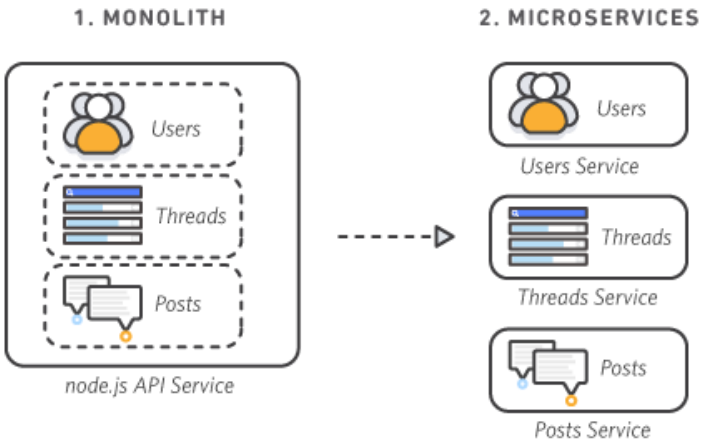


Obrázok 5: Príklad architektúry mikroslužieb [2]

2.1 Kľúčové rozdiely medzi mikroslužbami a monolitom

Monolitické aplikácie sa obvykle skladajú z rozhrania na strane klienta, takzvané UI, databázy a serverovej aplikácie. Vývojári vytvárajú všetky tieto moduly ako jeden súvislý kód. Na druhej strane v distribuovanej architektúre sa každá mikroslužba snaží správať tak

aby obsluhovala len kód danej funkcie alebo obchodnej logiky. Namiesto výmeny údajov v rovnakej kóde, mikroslužby komunikujú prostredníctvom API. [3]



Obrázok 6: Grafické znázornenie rozdielnej architektúry monolitu a mikroslužieb [3]

2.1.1 Proces vývoja

Monolitické aplikácie sú na začiatku jednoduchšie, pretože nie je potrebné vopred príliš veľa plánovať. Môžete začať a podľa potreby pridávať moduly. Rizikom ale je, že aplikácia sa môže stať časom príliš zložitá, ťažko aktualizovateľná alebo zmeniteľná.

Architektúra mikroslužieb vyžaduje viac plánovania a návrhov na začiatku. Vývojári musia navrhnúť rôzne funkcie, ktoré budú fungovať nezávisle a plánovať konzistentné API. Avšak tieto časové náklady na počiatočnú koordináciu robia údržbu kódu oveľa efektívnejšou. Je možné robiť zmeny počas vývoja a rýchlejšie hľadať vzniknuté chyby. Taktiež prepoužiteľnosť kódu časom rastie.

2.1.2 Nasadenie

Nasadenie monolitických aplikácií je jednoduchšie než nasadenie mikroslužieb. Vývojári nasadia celý kód a jeho závislosti v jednom prostredí.

Naopak nasadzovanie aplikácií založených na mikroslužbách je komplexnejšie, keďže každá mikroslužba je nezávisle nasaditeľný softvérový balíček. Vývojári zvyčajne kontajnerizujú mikroslužby pred ich nasadením. Takéto kontajnery zbalia kód a jeho súvisiace závislosti, aby bola dosiahnutá nezávislosť od platformy.

2.1.3 Ladenie

Ladenie je softvérový proces na identifikáciu chýb v kóde, ktoré spôsobujú, že aplikácia sa nespráva podľa očakávaní. Pri ladení monolitickej architektúry môže vývojár sledovať pohyb údajov naprieč aplikáciou, alebo skúmať správanie sa kódu v rámci rovnakého programovacieho prostredia.

Medzitým identifikácia problémov v kóde pri architektúre mikroslužieb vyžaduje preskúmanie viacerých samostatne stojacich služieb pripojených k projektu. Ladenie aplikácií postavenej na mikroslužbách môže byť náročnejšie aj preto, že za mnohé mikroslužby môže byť zodpovedných niekoľko vývojárov z viacerých tímov. Napríklad ladenie môže vyžadovať koordinované testy, diskusiu a spätnú väzbu medzi členmi tímu alebo tímov, čo zaberie viac času a zdrojov.

2.1.4 Modifikácie

Aj malá zmena v jednej časti monolitickej aplikácie ovplyvňuje viaceré funkcie kvôli úzko prepojenému štýlu programovania. Okrem toho, ak vývojári zapracujú nové zmeny v monolitickej aplikácii, musia znovu otestovať a znovu nasadiť celý systém na serveri.

Naopak, prístup mikroslužieb umožňuje väčšiu flexibilitu pretože je jednoduchšie robiť zmeny v aplikácii. Takže namiesto úpravy všetkých služieb stačí, aby sa zmenili len špecifické funkcie, ktorých sa zmeny týkajú. Tiež sa môžu konkrétne služby nasadzovať nezávisle. Takýto prístup je veľmi užitočný pri pracovnom postupe, kde sa často nasadzujú nové verzie, alebo kde sa pomerne často vykonávajú malé zmeny bez ovplyvnenia stability systému.

2.1.5 Škálovanie

Škálovanie v monolitických aplikáciách čelí niekoľkým výzvam. Monolitická architektúra totižto obsahuje všetky funkcie v jednom kódovom celku, takže celá aplikácia musí byť škálovaná podľa stále meniacich sa požiadaviek. Napríklad ak výkon aplikácie klesá, pretože funkcie slúžiace na komunikáciu s užívateľmi zažívajú nápor, tak treba zvýšiť výpočtovú kapacitu, aby ste akomodovali celú monolitickú aplikáciu. To vedie k plytvaniu zdrojmi, pretože nie všetky časti aplikácie sú tak vyťažené a stále majú dosť kapacity.

Medzitým architektúra mikroslužieb podporuje distribuované systémy. Každá softvérový komponent dostane v takomto distribuovanom systéme svoje vlastné výpočtové zdroje.

Tieto zdroje možno škálovať nezávisle na súčasných kapacitách a na požiadavkách, ktoré sa budú vykonávať. Takže napríklad môžete prideliť viac zdrojov jednej službe namiesto zvyšovaniu výpočtových kapacít celého systému.

2.1.6 Inovácie

Monolitická architektúra obmedzuje organizácie zavádzať nové mechanizmy a technológie do už existujúcich aplikácií. Vývojári nemôžu meniť určité časti kódu s využitým novými technológií čo brzdí organizáciu v zavádzaní moderných technologických trendov.

Medzitým mikroslužby sú nezávislé softvérové komponenty, ktoré vývojári môžu budovať s rôznymi rámcami a softvérovými technológiami. Voľné prepájanie medzi mikroslužbami umožňuje firmám inovovať určité komponenty a časti kódu rýchlejšie.

2.1.7 Znižovanie rizík

Aplikácie s monolitickou aj mikroslužbovou architektúrou sa stále stretávajú s takzvanými konfliktami kódov, chybami v kóde, či neúspešnými aktualizáciami. Rozdiel je v tom, že monolitická aplikácia nesie väčšie riziko pri snahe o aktualizovanie aplikácie, pretože pri celej aplikácií hrozí, že nesprávnym nasadením aktualizácie takáto aplikácia padne. Inými slovami aj menšia chyba v kóde môže spôsobiť zlyhanie celej aplikácie. Takéto incidenty majú potenciál spôsobiť vážne výpadky služieb a ovplyvniť všetkých aktívnych používateľov.

Aj preto pri väčších aplikáciách trendy smerujú skôr k vytváraniu aplikácií ako mikroslužieb, aby minimalizovali riziká pri nasadzovaní. Ak zlyhá mikroslužba, ostatné mikroslužby zostávajú funkčné, čo obmedzuje dopad na celú aplikáciu. Taktiež sa používajú nástroje na predchádzanie a opravu problémov, ktoré by mohli ovplyvniť chod mikroslužby a aby zlepšili možnú obnoviteľnosť aplikácie.

2.1.8 Urýchliť čas dokončenia aplikácie

Úsilie na dokončenie softvéru ako monolitickej aplikácie exponenciálne rastie s nárastom zložitosti kódu. Toto zvyčajne nakoniec vedie k tomu, že vývojári musia stráviť viac času na spravovanie už existujúceho kódu a knižníc na úkor budovania nových funkcií.

Naopak, organizácie so znalosťami mikroslužieb môžu vyvíjať a dokončovať aplikácie rýchlejšie. V takejto distribuovanej softvérovej architektúre sa každý vývojár zameriava na

menší kúsok kódu, namiesto programovania do jedného veľkého kódu. Keď vývojári vytvárajú špecifickú mikroslužbu, nepotrebujú rozumieť ako fungujú ostatné mikroslužby. Stačí len použiť vhodné API, ktoré sú rýchlejšie a jednoduchšie na pochopenie.

2.1.9 Celkové náklady na vývoj

Obe architektúry, či už mikroslužby tak aj monolitické aplikácie, počas svojej existencie generujú náklady na vývoj, následné nasadenia a aj neskoršiu údržby. Avšak prístup mikroslužieb je z dlhodobejšieho hľadiska nákladovo efektívnejší. Aplikácie používajúce mikroslužby možno škálovať na požiadanie a to horizontálnym pridaním výpočtových zdrojov. Stačí pridať zdroje len pre konkrétnu službu, nie pre celú aplikáciu. Na škálovanie monolitických systémov musia spoločnosti zvyšovať výpočtový výkon pre celú aplikáciu, čo je v konečnom zúčtovaní drahšie. Okrem nákladov na infraštruktúru rastú aj náklady na údržbu monolitických aplikácií, ktoré sa menia s prichádzajúcimi požiadavkami. Napríklad niekedy musia vývojári prevádzkovať staršie monolitické aplikácie na novšom hardvéri. To vyžaduje špeciálne znalosti a vývojári musia aplikáciu prebudovať tak, aby zostala stále funkčná.

Na druhej strane mikroslužby bežia nezávisle od konkrétneho hardvéru či platformy a to ušetrí organizáciám náklady na drahé inovácie. [3]

2.2 Kedy použiť monolitickú a kedy mikroslužbovú architektúru

Tak isto ako monolitická tak aj mikroslužbová architektúra pomáhajú a umožňujú vývojárom budovať aplikácie rôznymi prístupmi. Je tiež dôležité si uvedomiť, že mikroslužby nezničujú zložitosť aplikácie. Štruktúra mikroslužieb odhaľuje možnosti ako riešiť zložité problémy a umožňuje vývojárom efektívnejšie budovať, spravovať a škálovať veľké aplikácie. Keď sa rozhodujete medzi vývojom mikoslužbovej alebo monolitickej architektúry, môžete zvážiť nasledujúce faktory.

2.2.1 Veľkosť aplikácie

Monolitický prístup je vhodnejší pri navrhovaní menšej, jednoduchej aplikácie alebo prototypu. Keďže monolitické aplikácie využívajú jediný zdrojový kód a rámec, vývojári môžu softvér vyvíjať bez potreby integrácie viacerých služieb. Mikroslužbové aplikácie môžu vyžadovať značný čas a úsilie pri návrhoch a rozbehávaní služieb čo zbytočne

navyšuje náklady a čas pri veľmi malých projektov. Na druhej strane architektúra mikroslužieb je lepšia pre budovanie komplexného systému. Poskytuje pevný programovací základ pre tím a podporuje tak schopnosť pružne pridávať ďalšie funkcie.

2.2.2 Kompetencie tímu

Napriek svojej flexibilite vyžaduje vývoj mikroslužieb nutnosť mať určité znalosti z tejto oblasti a návrhové myslenie. Na rozdiel od monolitických aplikácií vyžaduje vývoj mikroslužieb porozumenie cloudovej architektúry, API, kontajnerizácii a iným špecifickým expertízam moderných cloudových aplikácií. Navyše riešenie problémov s mikroslužbami môže byť pre začínajúcich vývojárov v distribuovanej architektúre výzvou.

2.2.3 Infraštruktúra

Monolitická aplikácia beží na jednom serveri, ale aplikácie mikroslužieb viac využívajú cloudové prostredie. Hoci je možné prevádzkovať mikroslužby aj z jedného servera, vývojári obvykle hostia mikroslužby u poskytovateľov cloudových služieb, aby pomohli zabezpečiť škálovateľnosť, odolnosť voči chybám a vysokú dostupnosť. Predtým ako začneme s mikroslužbami je potrebné mať správnu infraštruktúru. To samozrejme pri mikroslužbách vyžaduje viac úsilia na nastavenie nástrojov a pracovného postupu, ale na druhej strane sú preferované kvôli budovaniu komplexnej a škálovateľnej aplikácie. [3]

3 MIKROFRONTEND

Pojem mikrofrontend sa prvýkrát objavil koncom roku 2016 v Technologickom radare spoločnosti Thoughtworks, čo je dokument, ktorý pravidelne vydáva táto konzultačná firma zameraná na softvérový vývoj. Dôvody, prečo sa vývojári frontendových aplikácií začali zaoberať možnosťou vytvorenia mikrofrontendov, boli veľmi podobné ako v prípade mikroslužieb. Taktiež zápasili s problémami, ktoré má monolitická architektúra s veľmi obťažnými možnosťami škálovania aplikácie.

Martin Fowler definoval architektúru mikrofrontendu ako „architektonický štýl, kde samostatne dodávateľné frontendové aplikácie sú komponované do väčšieho celku. Jednoducho povedané, mikrofrontend je časť webovej stránky (nie celá stránka). V architektúre mikrofrontendu existuje „Hostiteľ“ alebo „Kontajner“, ktorý môže hostiť jednu alebo viacero mikrofrontendov. Hostiteľská/kontajnerová stránka môže tiež zdieľať niektoré zo svojich vlastných mikrofrontendových komponentov.[4]

Mikrofrontend teda môže byť kompletná stránka alebo tiež konkrétne fragmenty stránky, ktoré môžu iné tímy použiť hocikde na stránke, ktorú vyvíjajú. Na rozdiel od opakovane použiteľných komponentov, mikrofrontendy môžu byť implementované nezávisle ako individuálne projekty. Technika implementácie mikrofrontendov spočíva v tom, že všetko sa vyvíja samostatne, pričom ostatné komponenty sú extrahované a používané počas behu aplikácie. Mikrofrontend pozostáva z niekoľkých nezávislých a modulárnych komponentov, ktoré sa zobrazujú podľa potreby. To znamená, že pre konkrétnu stránku sú načítané len potrebné komponenty. Tieto komponenty priamo interagujú s údajmi a nevyžadujú centralizovaný server na smerovanie požiadaviek, alebo spracovanie údajov. Okrem komponentov zobrazujúcich obsah môže mikrofrontend obsahovať aj niekoľko pomocných komponentov na interakciu s prostredím aplikácie.[5]

3.1 Výhody

Podobne ako mikroslužby aj mikrofrontendy prišli hlavne ako reakcia na nevýhody vyplývajúce zo stále narastajúcich monolitov. Fakt, že na strane backendu sa mikroslužby osvedčili ako dobrá stratégia vývoja, začalo sa čoraz viac špekulovať o podobnom prístupe

aj na strane frontendu. V tejto kapitole sa preto budeme sústrediť na vlastnosti mikrofrontendov a na výhody, ktoré prinášajú, alebo z nich vyplývajú.

3.1.1 Postupné zlepšovanie

Pre mnohé organizácie predstavuje práca s mikrofrontendami začiatok novej éry. Staré, rozsiahle frontendové monolitické aplikácie sa stávajú čoraz viac zastaranými kvôli používaniu či už starších technológií alebo často krát kvôli kódu, ktorý bol neraz písaný pod časovým tlakom. Takéto aplikácie časom prichádzajú k bodu kedy je celé prepísanie kódu aplikácie lákavou ale riskantnou možnosťou. Aby sme sa vyhli rizikám spojeným s prepísaním celého kódu, volíme najskôr postup kde postupne prepisujeme starú aplikácie kúsok po kúsku. Zároveň ale chceme naďalej prinášať nové funkcie našim zákazníkom bez toho, aby nás spomaľovali nevýhody monolitu. A toto sú zvyčajne najčastejšie situácie, ktoré vedú k zavádzaniu architektúry mikrofrontendov. Ak sa podarí jedenému tímu úspešne nasadiť nové funkcie s minimálnymi zásahmi do starého systému, môže to spustiť domino efekt kde aj ostatné tímy sa budú tiež chcieť pripojiť k takémuto novému prístupu. V takomto prípade stávajúci kód je stále potrebné udržiavať a v niektorých prípadoch je rozumné pridávať doň aj nové funkcie avšak teraz už existuje možnosť voľby prístupu inak ako zväčšovanie monolitu. Naším cieľom je teda získať väčšiu slobodu pri rozhodovaní o jednotlivých častiach nášho produktu a postupne vylepšovať našu architektúru, závislosti a celkový zážitok pre používateľa. Ak dôjde k výraznej zmene v našom hlavnom rámci, každý mikrofrontend môže byť aktualizovaný v konkrétnom správnom čase, namiesto toho aby sme museli celú aplikáciu naraz zastaviť a aktualizovať. Ak chceme experimentovať s novou technológiou alebo novými formami interakcie môžeme tak urobiť oveľa izolovanejším spôsobom, než to bolo možné doposiaľ.

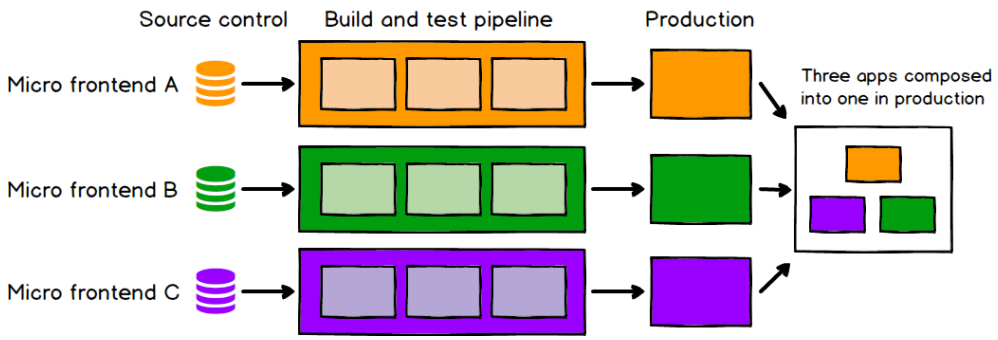
3.1.2 Jednoduché a nezávislé kódy

Je priam istota, že zdrojový kód každého mikrofrontendu bude vždy oveľa menší než zdrojový kód jedného monolitického frontendu. Tieto menej rozsiahle kódy sú zvyčajne jednoduchšie a pre vývojárov je s nimi ľahšie pracovať. Zvlášť sa snažíme vyhýbať takým situáciám, ktoré vznikajú neúmyselným či nevhodným prepojením komponentov, ktoré by nemali o sebe ani vedieť. Vytýčením jasných hraníc okolo ohraničených kontextov aplikácie sťažujeme vznik takéhoto náhodného prepojenia. Samozrejme ani pri rozhodnutí o používaní mikrofrontendovej architektúry sa nezbavíme nutnosti tieto nové časti

naprogramovať. Nie je to ani našou snahou vyhnúť sa programovaniu, avšak investujeme viac času a energie do úsilia, aby sme takýto kód zlepšili.

3.1.3 Samostatné nasadenie

Podobne ako to bolo pri mikroslužbách aj pri mikrofrontendoch je kľúčové, aby sa dali jeho časti nasadiť nezávisle. To znamená, že znižovaním veľkosti nasadzovaného balíka zmien znižujeme aj s tým spojené riziko vzniknutia akýchkoľvek chýb. Bez ohľadu na to, kde je umiestnený náš frontendový kód, každý mikrofrontend by mal mať svoj vlastný proces nepretržitého dodávania, ktorý zabezpečí jeho zostavenie, testovanie a nasadenie až do finálneho prostredia. Takže by sme mali byť schopní nasadiť každý mikrofrontend s minimálnym ohľadom na aktuálny stav ostatných kódov alebo pipeline. Nemalo by teda ani záležať na tom či je starý monolit aktualizovaný manuálnym fixným štvrťročným cyklom vydávania nových zmien, či iný tím nasadil chybnú funkcionalitu alebo nasadil nedokončené zmeny do svojej hlavnej vetvy. Ak je daný mikrofrontend pripravený ísť do produkcie, mal by mať možnosť to urobiť a takéto rozhodnutie by malo byť na tíme, ktorý ho vyvíja a udržiava.

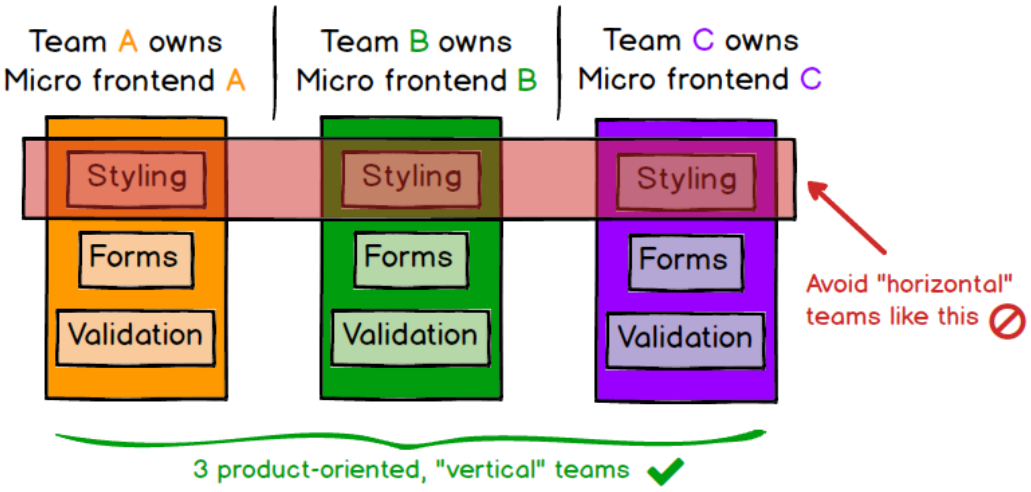


Obrázok 7: Každé mikrofrontendové rozhranie sa nasadzuje do produkcie nezávisle [6]

3.1.4 Samostatné tímy

Vďaka oddeleniu našich základných kódov a cyklov vydávania sa nám podarí dosiahnuť nezávislosť tímov, ktoré môžu mať zodpovednosť za určitú časť produktu od počiatočnej myšlienky až po uvedenie do prevádzky. To dovoľuje tímom mať plnú zodpovednosť za všetko čo potrebujú na to aby zákazníkom dodali produkt podľa požiadaviek a to všetko rýchlo a efektívne. Aby to fungovalo, naše tímy by mali byť štruktúrované podľa vertikálnych segmentov biznis funkcií (prístup kde je každý tím zodpovedný za celý životný

cyklus jednej alebo viacerých funkcií produktu), nie podľa technických znalostí. Jednoduchý spôsob ako to dosiahnuť je rozdeliť produkt podľa toho čo uvidia koncoví používatelia, takže každý mikrofrontend obsahuje jednu stránku aplikácie, ktorá je od začiatku do konca v zodpovednosti jedného tímu. To prináša vyššiu súdržnosť práce medzi tímami ako keby boli organizované okolo technických alebo „horizontálnych“ záležitostí, ako sú štýlovanie, písanie formulárov alebo validácia. Stručne povedané, mikrofrontendy sú o rozdelení veľkých celkov na menšie, ľahšie zvládnuteľné kúsky a potom o jasnom vymedzení závislostí medzi nimi.



Obrázok 8: Každá aplikácia by mala byť v správe jedného tímu [6]

3.1.5 Zhrnutie

Stručne povedané mikrofrontendy sú o rozdelení veľkých a zložitých celkov na menšie, ľahšie zvládnuteľné časti a tiež o jasnom vymedzení závislostí medzi nimi. Náš výber technológie, naše základné kódy, naše tímy a naše procesy vydávania by mali byť schopné fungovať a vyvíjať sa nezávisle od seba a bez potreby nadmernej koordinácie. [6]

3.2 Nevýhody mikrofrontendov

Mikrofrontendová architektúra, hoci prináša mnohé výhody, ako sú nezávislosť vývojových tímov a flexibilita v technologických rozhodnutiach, prichádza aj s radom nevýhod, ktoré môžu ovplyvniť celkovú efektívnosť projektov. Táto kapitola sa zameria na potenciálne problémy a výzvy spojené s implementáciou mikrofrontendov.

3.2.1 Zvýšená komplexnosť riadenia

Jednou z najvýraznejších nevýhod mikrofrontendov je zložité riadenie. Vývoj aplikácie rozdelený medzi viaceré tímy vyžaduje koordináciu a synchronizáciu, ktorá môže byť časovo náročná a náchylná na chyby. Tímy musia efektívne komunikovať, aby zabezpečili konzistenciu a integráciu rozličných častí aplikácie.

3.2.2 Vyššie nároky na zdroje

Mikrofrontendy vyžadujú rozdelenie zdrojov ako sú vývojári, testeri a infraštruktúra medzi množstvo samostatných tímov. Každý tím potrebuje príslušné nástroje a prístupy pre vývoj, čo môže viesť k značným nákladom a zložitosti v správe projektu. Pre malé projekty alebo tímy môžu byť mikrofrontendová architektúra nepraktická a finančne náročná.

3.2.3 Výzvy pri nasadení a integrácii

Každý mikrofrontend môže mať svoje vlastné požiadavky na nasadenie a závislosti, čo môže komplikovať proces nasadzovania aplikácie ako celku. Integrovanie rôznych mikrofrontendov, ktoré môžu používať rôzne knižnice a rámce, si vyžaduje robustné integračné a testovacie procesy, aby sa predišlo konfliktom a chybám vo finálnej podobe aplikácii

3.2.4 Riziko fragmentácie kódu a duplikácie závislostí

Používanie rôznych technológií a verzií knižníc v rôznych častiach aplikácie môže viesť k duplicite závislostí a zvýšeným nárokom na prenos dát. Toto riziko môže znížiť efektivitu a zvýšiť celkovú veľkosť aplikácie, čo môže mať negatívny vplyv na výkon a dobu načítania.

3.2.5 Zložitosť vývoja

Rozdelenie aplikácie na menšie samostatné jednotky môže teoreticky zjednodušiť vývoj ale v praxi často vedie k potrebe dodatočnej koordinácie a vypracovaniu technickej dokumentácie. Zložitosť môže rásť s počtom mikrofrontendov, ktoré je potrebné spravovať, najmä ak komunikujú alebo si zdieľajú spoločné stavy.

3.2.6 Potenciál pre nekonzistentné užívateľské rozhrania

Rôzne tímy pracujúce na rôznych častiach aplikácie môžu používať odlišné štýly a prístupy, čo môže viesť k nekonzistentnému užívateľskému zážitku. Aby sa zachovala konzistentnosť

a kvalita užívateľského rozhrania, je potrebná dôkladná koordinácia a prísne dodržiavanie dizajnových štandardov.

3.2.7 Zhrnutie nevýhod

Mikrofrontendová architektúra, napriek svojim mnohým výhodám, so sebou prináša aj výzvy, ktoré môžu komplikovať vývojové a nasadzovacie procesy. Zvýšená komplexnosť riadenia, vyššie nároky na zdroje, potenciál pre fragmentáciu a nekonzistentnosť sú len niektoré z otázok, ktoré je potrebné riešiť pri zavádzaní tejto architektúry. Efektívne využitie mikrofrontendov vyžaduje jasne definované rozhrania, prísnu koordináciu medzi tímami a robustné procesy správy projektu. [7]

3.3 Komunikácia a závislosti medzi Mikrofrontendami

V tejto kapitole o komunikácii a závislostiach medzi mikrofrontendami je kľúčové zvážiť niekoľko základných princípov pre efektívnu spoluprácu a údržbu systému:

- Izolácia a oddelenie: Je nevyhnutné zabezpečiť nezávislosť mikrofrontendov. Chceme im umožniť samostatný vývoj a nasadzovanie, znížiť závislosti a zvýšiť flexibilitu.
- Správa zložitosti údajov: Podľa potreby zdieľania jednoduchých alebo zložitých údajov využívame príslušné nástroje od vlastných udalostí až po knižnice pre správu zdieľaného stavu.
- Výkon a komunikačná záťaž: Hodnotenie vplyvu komunikácie na výkon aplikácie, vyvažovanie efektívneho zdieľania údajov s udržaním optimálneho výkonu.
- Centralizovaná vs. distribuovaná kontrola: Rozhodnutie medzi centralizovanou správou zdieľaného stavu a distribuovanou kontrolou vyberáme pomocou vlastných udalostí alebo GraphQL podľa špecifických potrieb projektu.
- Konzistentnosť a synchronizácia: Zabezpečenie dátovej konzistentnosti a zvládnutie výziev spojených so synchronizáciou, najmä pri asynchrónnych aktualizáciách.
- Bezpečnosť a súkromie: Prioritizácia ochrany citlivých údajov a dodržiavanie osvedčených postupov v zdieľaní údajov pre zachovanie dôvernosti.
- Testovanie a údržba: Uľahčenie ladenia a dlhodobej údržby cez jasne štruktúrované mechanizmy komunikácie a robustné testovacie procedúry.

- Škálovateľnosť a budúca rozšíriteľnosť: Zabezpečenie, že vybraný prístup k zdieľaniu údajov bude škálovateľný a pružný na podporu budúceho rastu a integrácie nových mikrofrontendov.

Zdôrazňuje sa dôležitosť strategického plánovania, jasnej dokumentácie a adaptácie k zmenám v požiadavkách projektu, pričom hlavným cieľom je udržať súdržný, výkonný a bezpečný systém.

3.3.1 Metódy na komunikáciu

Pri navrhovaní a vývoji mikrofrontendových aplikácií je dôležité zvoliť efektívnu metódu pre komunikáciu a zdieľanie stavu. Existuje niekoľko prístupov:

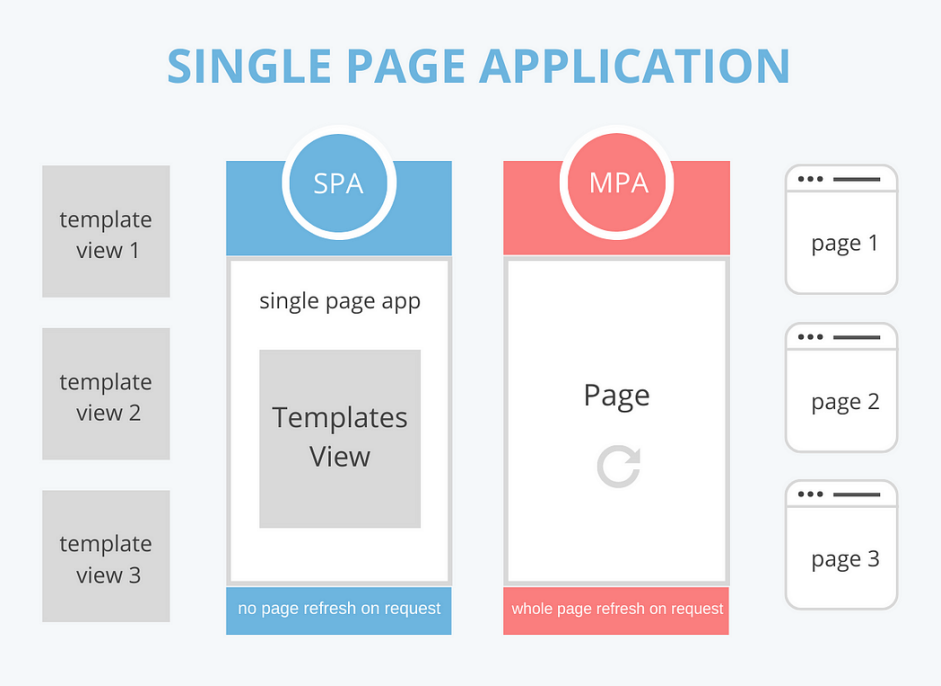
- Tento model umožňuje mikrofrontendom komunikovať prostredníctvom publikovania a odoberania udalostí. Ide o flexibilný mechanizmus pre zdieľanie dát a notifikácie medzi nezávislými komponentami.
  - Výhody:
    - Umožňuje nezávislý vývoj a aktualizácie.
    - Znižuje priamu závislosť medzi mikrofrontendami.
  - Nevýhody:
    - Môže spôsobiť konflikty v názvoch udalostí,
    - Ladenie môže byť zložitejšie v rozsiahlejších aplikáciách.
- Zdieľaný Stav (State Management): Používa centralizovaný úložný systém, ako sú Redux alebo Mobx na správu a zdieľanie stavu naprieč mikrofrontendami. Tento prístup umožňuje jednotné a štruktúrované riadenie stavových dát.
  - Výhody:
    - Centralizované a štruktúrované riadenie stavu
    - Podporuje modulárny kód a predvídateľný tok dát.
  - Nevýhody:
    - Môže byť zbytočne komplexný pre menšie aplikácie
    - Vyžaduje čas na naučenie sa práce s knižnicami pre správu stavu.

- Lokálne úložisko: Umožňuje jednoduché ukladanie a načítavanie dát pre zdieľanie medzi mikrofrontendami pomocou lokálneho úložiska prehliadača. Tento prístup je vhodný pre zdieľanie malých množstiev dát.
    - Výhody:
        - Jednoduché na implementáciu
        - Funguje naprieč rôznymi oknami alebo kartami prehliadača
    - Nevýhody:
        - Obmedzené na reťazce kľúč-hodnota, môže byť nevhodné pre zložité dátové štruktúry
        - Problémy s výkonom pri veľkom objeme dát
- GraphQL: GraphQL je dotazovací jazyk pre API, ktorý umožňuje efektívne načítavanie a zdieľanie dát medzi mikrofrontendami. Ponúka flexibilitu pri vyžiadaní konkrétnych dát potrebných pre aplikáciu.
    - Výhody:
        - Umožňuje efektívne získavanie a aktualizáciu dát
        - Redukuje nadmerné a nedostatočné načítavanie dát.
    - Nevýhody:
        - Vyžaduje backend, ktorý podporuje GraphQL
        - Môže predstavovať výzvy s asynchrónnosťou a koordináciou medzi mikrofrontendami.
- Kompozitný Prístup (Single SPA): Využíva knižnicu ako Single SPA na vytvorenie "shell" aplikácie, ktorá riadi integráciu a komunikáciu medzi viacerými mikrofrontendami. Každý mikrofrontend zostáva nezávislým ale sú integrované do koherentného celku.
    - Výhody:
        - Poskytuje centralizovanú kontrolu nad vykresľovaním a komunikáciou
        - Uľahčuje integráciu mikrofrontendov.

- Nevýhody:
  - Zavádza dodatočnú zložitosť pretože mikrofrontendy musia dodržiavať životný cyklus definovaný "shell" aplikáciou
  - Môže byť náročné na správu pri veľkých aplikáciách.

Pri výbere metódy pre komunikáciu a závislosti medzi mikrofrontendami je dôležité zvážiť potreby projektu, veľkosť a zložitosť aplikácie, požiadavky na izoláciu a výkon. Každá z uvedených metód ponúka jedinečné výhody a nevýhody, pričom vhodný výber závisí od špecifických potrieb a architektonických rozhodnutí projektu. Je dôležité zamerať sa na udržanie čistého a udržateľného kódu, zabezpečenie efektívnej a bezproblémovej komunikácie medzi komponentmi a minimalizáciu potenciálnych výkonnostných problémov. [8]

3.4 SPA a MPA



Obrázok 9: Model SPA a MPA aplikácie [8]

SPA (Single PageApplications) a MPA (MultiplePageApplications) predstavujú dva hlavné prístupy k vývoju webových aplikácií. SPA je aplikácia, ktorá načíta jedinú HTML stránku a dynamicky ju aktualizuje ako reakcia na interakcie užívateľa, čím poskytuje plynulú užívateľskú skúsenosť podobnú desktopovým aplikáciám. Na druhej strane MPA načíta

viacero HTML stránok z servera čo môže byť vhodné pre rozsiahlejšie webové projekty s komplexnejšou štruktúrou a obsahom. Mikrofrontendový prístup umožňuje tímom vyvíjať, nasadzovať a škálovať časti týchto aplikácií nezávisle či už ide o SPA alebo MPA, čím sa zvyšuje flexibilita a efektivita vývojových procesov.

Jednou vetou povedané: SPA a MPA sa zameriavajú na štruktúru webovej aplikácie z pohľadu počtu načítaných stránok (jedna stránka pre SPA vs. viacero stránok pre MPA) a na spôsob, akým užívateľ naviguje a interaguje s aplikáciou. [9]

3.4.1 Jednostránkové aplikácie (SPA)

Single-PageApplication (SPA) je webová aplikácia alebo webová stránka, ktorá funguje v našom webovom prehliadači a načíta všetko z jednej stránky. To znamená, že interaguje so všetkým v aplikácii na jednej obrazovke. To je najväčší rozdiel od prechádzania z jednej stránky na druhú ako tomu je pri tradičných webových stránkach alebo MPA. SPA pomocou JavaScriptu dynamicky aktualizuje to čo vidíme na obrazovke. Príkladom SPA sú sociálne platformy ako Facebook a X (predtým Twitter), kde aplikácia neustále aktualizuje a zobrazuje viac obsahu bez potreby opätovného načítania stránky.

3.4.2 Viacstránkové aplikácie (MPA)

Multi-PageApplication (MPA) je typ webovej aplikácie, ktorá pri každej špecifickej akcii načíta novú stránku. Tieto aplikácie sú považované za tradičné webové aplikácie, fungujú ako tradičné webové stránky a pozostávajú z viacerých samostatných statických HTMLwebových stránok. MPA je vhodné pre podniky, ktoré predávajú viacero produktov a služieb, pretože im umožňuje prispôsobiť vzhľad každej stránky podľa svojich potrieb.

3.4.3 Výhody a nevýhody SPA a MPA

- SPA výhody:
    - Rýchle načítavanie a výkon: SPA načíta všetok potrebný obsah vopred čo znamená rýchlejšie načítanie ako pri MPA.
    - Ukladanie dát do vyrovnávacej pamäte: Inicializácia SPA ukladá veľké množstvo dát do cache čo zlepšuje časy načítavania a umožňuje aj prezeranie si stránky offline

- Rýchlejší a jednoduchší vývojový cyklus: Vývoj SPA môže byť rýchlejší a lacnejší vďaka použitiu rovnakého API pre webové a mobilné aplikácie

- SPA nevýhody:
  - Slabá podpora SEO: SPA nie sú tak dobre optimalizované pre vyhľadávacie algoritmy ako MPA.
  - Potreba podpory JavaScriptu: Bez JavaScriptu SPA nemôže využívať dynamické načítavanie.
  - Bezpečnostné obavy: SPA sú náchylnejšie na bezpečnostné riziká ako napríklad skriptovanie medzi stránkami (XSS).
- MPA výhody:
  - Vynikajúca škálovateľnosť: MPA môže obsahovať neobmedzený počet stránok, čo je ideálne pre online obchody a spravodajské portály.
  - Lepšie SEO: Vyhľadávacie algoritmy ľahšie zhromažďujú informácie z MPA.
  - Podpora dátových analýz: Vďaka viacerým stránkam je jednoduchšie analyzovať výkon každej stránky a optimalizovať ju.

- MPA nevýhody:
  - Pomalší výkon: Interakcia s MPA si vyžaduje načítanie nových HTML stránok, čo je pomalšie ako dynamické načítavanie v SPA.
  - Vyššie náklady a dlhší čas vývoja: MPA môže byť zložitejšie a drahšie na vývoj a udržiavanie v porovnaní so SPA čo môže viesť k dlhšiemu času uvedenia na trh.
  - Menej opakovateľne využiteľného kódu: Na rozdiel od SPA kde môžete použiť rovnaké API pre webové a mobilné verzie aplikácie, pri MPA to nie je možné bez dodatočného vývoja a ďalších zdrojov.

3.4.4 Ako si vybrať medzi SPA a MPA

Výber medzi SPA a MPA závisí od špecifických požiadaviek vášho projektu. Pozrieme sa teda na niektoré ukazovatele, ktoré by nám mohli pomôcť pri rozhodovaní:

- SPA použijeme v prípade ak:
  - Vyhľadávanie na internete nie je pre nás prioritou.
  - Naša aplikácia má veľa interaktivity a funkcií.
  - Náš vývojový tím má skúsenosti s JavaScriptom a bezpečnosťou SPA.
  - Máme obmedzený rozpočet a krátky čas na doručenie aplikácie.
- Použitie MPA prípade ak:
  - Vyhľadávanie na internete je pre náš prioritou.
  - Naša aplikácia je prevažne na čítanie s obmedzenými interaktívnymi komponentmi.
  - Naša aplikácia ponúka širokú škálu produktov, služieb alebo obsahu, ktoré vyžadujú samostatné stránky.
  - Chceme, aby naša aplikácia fungovala bez podpory JavaScriptu.

SPA a MPA predstavujú dva rôzne prístupy k vývoju webových aplikácií, každý s vlastnými výhodami a nevýhodami. SPA ponúkajú plynulú užívateľskú skúsenosť a rýchle načítavanie ale môžu mať slabú podporu SEO a určité bezpečnostné riziká. MPA poskytujú lepšie SEO a analytické možnosti ale môžu byť pomalšie a náročnejšie na vývoj. Výber medzi nimi by mal byť preto založený na potrebách daného projektu, cieľovej skupine a technických požiadavkách.[10]

3.5 CSR a SSR

Vysvetlením si toho čo sú SPA a MPA sa nám otvárajú dvere k lepšiemu porozumeniu rozličných prístupov k renderovaniu v modernom webovom vývoji. A to konkrétne k rozdielom medzi server-side renderingom (SSR) a client-side renderingom (CSR). Tieto

techniky renderovania sú základnými stavebnými kameňmi SPA a MPA, keďže ovplyvňujú ako a kde sa webový obsah generuje a zobrazuje.

Ako sme už spomenuli, v súčasnosti existujú dva hlavné prístupy k renderovaniu webových stránok a aplikácií: server-siderendering (SSR) a client-siderendering (CSR). Tieto prístupy majú zásadný vplyv na výkon alebo aj na SEO a celkovú užívateľskú skúsenosť s webovou aplikáciou vrátane tých, ktoré využívajú architektúru mikrofrontendov.

Jednou vetou povedané: SSR a CSR sa zaoberajú technickým mechanizmom renderovania stránok teda či sa obsah generuje na serveri a odošle sa ako kompletné HTML (SSR), alebo či sa na klientovi dynamicky generuje z minimálneho HTML a JavaScriptu (CSR).

3.5.1 Server-side rendering (SSR)

SSR znamená, že webová stránka je predvytvorená na serveri a užívateľovi je poslaná už kompletná HTML stránka. Tento prístup je ideálny pre statické webové stránky alebo stránky, ktoré nevyžadujú veľa interaktivity. SSR má pozitívny vplyv na SEO, pretože vyhľadávače lepšie indexujú obsah, ktorý je okamžite dostupný pri načítaní stránky. Navyše, SSR môže ponúknuť rýchlejšie prvotné načítanie stránky, čo zlepšuje celkovú užívateľskú skúsenosť.

- Výhody SSR:
  - Rýchlejšie prvotné načítanie: Obsah je predvytvorený na serveri čo skracuje dobu načítania úvodnej stránky.
  - Lepšie SEO: Vyhľadávače lepšie indexujú obsah, ktorý je ihneď dostupný ako statické HTML.
  - Zlepšená dostupnosť: Stránky môžu byť dostupné aj pre užívateľov s obmedzenou alebo vypnutou funkcionalitou JavaScriptu v prehliadači.
  - Spoľahlivosť pri slabom pripojení: Užívatelia s pomalým internetovým pripojením môžu rýchlejšie vidieť obsah, keďže hlavná stránka je načítaná zo serveru.

- Nevýhody SSR:
  - Vyššie zaťaženie servera: Generovanie obsahu na serveri môže zvýšiť nároky na výpočtový výkon, najmä pri veľkom počte užívateľov.
  - Obmedzená interaktivita: Na dynamické zmeny na stránke je často potrebné odoslať požiadavku na server čo môže spomaliť interakciu s užívateľom.
  - Komplexnejší vývoj: SSR môže vyžadovať zložitejšiu architektúru a koordináciu medzi backendom a frontendom.

3.5.2 Client-sideRendering (CSR)

CSR znamená, že obsah stránky sa dynamicky generuje a aktualizuje na strane klienta pomocou JavaScriptu. Tento prístup umožňuje vytvárať dynamické a interaktívne webové aplikácie, kde zmeny na stránke sa môžu diať bez nutnosti opätovného načítania celej stránky. CSR je vhodný pre aplikácie, ktoré vyžadujú časté aktualizácie obsahu a vysokú úroveň interaktivity.

- Výhody CSR:
  - Plynulá užívateľská skúsenosť: Po načítaní aplikácie môže užívateľ interagovať s dynamickým obsahom bez nutnosti načítavania nových stránok z servera.
  - Menšie zaťaženie servera: Počiatočné načítanie úvodnej stránky môže znížiť počet požiadaviek na server.
  - Flexibilný vývoj: Umožňuje rýchlejšie iterácie a zmeny na frontende bez nutnosti zásahov do backendu.
  - Lepšie využitie cache: Po načítaní aplikácie môžu byť dáta a zdroje efektívnejšie ukladané do pamäte prehliadača na strane klienta.
- Nevýhody CSR:
  - Pomalšie prvotné načítanie: Načítanie celej aplikácie vrátane JavaScriptu môže spôsobiť oneskorenie pri prvom zobrazení obsahu.
  - Výzvy pre SEO: Dynamicky generovaný obsah môže byť pre vyhľadávače ťažšie indexovateľný.

- Vyššie požiadavky na prehliadač: Všetko renderovanie a vykonávanie aplikácie závisí od schopností a výkonu klientovho zariadenia.
- Bezpečnostné obavy: Aplikácie bežiace na strane klienta môžu byť viac náchylné určitým typom útokov ako je cross-site scripting (XSS).

3.5.3 Ako si vybrať medzi CSR a SSR

Výber medzi CSR a SSR závisí od viacerých kritérií, ktoré sú spojené s cieľmi našej webovej aplikácie, požiadavkami na výkon, SEO a užívateľskou skúsenosťou. Tu sú niektoré kľúčové faktory, ktoré by sme mali zvážiť pri rozhodovaní:

- CSR použijeme v prípade ak:
  - Našou prioritou je väčšiu interaktivita a dynamické aktualizácie obsahu bez potreby opätovného načítania.
  - Uprednostňujeme rýchle vývojové cykly a flexibilitu v iteráciách s dôrazom na skúsenosti s JavaScriptom.
  - Sme schopní riešiť prvotné dlhšie načítanie aplikácie výmenou za plynulejšiu navigáciu a interakcie v ďalšom používaní.
  - Máme stratégie na prekonanie výziev spojených s SEO, ktoré prináša CSR.
- Použijeme MPA prípade ak:
  - Je pre nás pomerne dôležitá optimalizácia vyhľadávania a zabezpečenie, že obsah je ľahko indexovateľný.
  - Kladieme dôraz na rýchlosť prvotného načítania a dostupnosť obsahu ihneď po príchode užívateľa na stránku.
  - Naša cieľová skupina zahŕňa užívateľov s obmedzeným prístupom k internetu alebo zariadeniami s nižšou výkonnosťou.
  - Chceme zabezpečiť dostupnosť obsahu aj v prípade vypnutého JavaScriptu alebo pre používateľov so špecifickými potrebami.

Pri rozhodovaní medzi CSR a SSR je dôležité zvážiť nielen technické aspekty, ale aj potreby vašich užívateľov a obchodné ciele. V niektorých prípadoch môže byť efektívne kombinovať oba prístupy, využívajúc SSR pre rýchle načítanie a SEO a CSR pre zlepšenie

interaktivity a užívateľskej skúsenosti po prvotnom načítaní. Tento hybridný prístup môže ponúknuť to najlepšie z oboch svetov, ak je správne implementovaný.[11] [12]

4 TECHNOLOGICKÝ EKOSYSTÉM MIKROFRONTENDOV

V rýchlo sa meniacom prostredí webového vývoja predstavujú mikrofrontendy prístup, ktorý umožňuje tímom efektívnejšie spravovať a iterovať rozsiahle webové aplikácie. Táto kapitola nás prevedie technologickým ekosystémom, ktorý mikrofrontendy obklopuje a ponúkne hlbší pohľad na rôzne nástroje, technológie a prístupy, ktoré tieto architektúry využívajú.

4.1 JavaScriptový rámec

JavaScriptové rámce predstavujú zbierku knižníc kódu JavaScript, ktoré nám poskytujú už existujúci kód pre rutinné programovacie úlohy. Sú to štruktúry so špecifickým kontextom, ktoré nám pomáhajú pri tvorbe webových aplikácií. JavaScript je silný programovací jazyk, ktorý nachádza uplatnenie ako vo frontendovom, tak aj v backendovom vývoji. Vytváranie robustných webových aplikácií bez použitia JavaScriptových rámcov síce je možné, avšak rámce nám poskytujú šablóny, ktoré nám výrazne uľahčujú prácu pri vytváraný nového kódu. Pri vytváraní aplikácie tak nemusíme kódovať každú funkciu od základov ale môžeme využívať existujúce sady funkcií. Všetky JavaScriptové rámce, podobne ako väčšina iných rámcov, stanovujú určité pravidlá a usmernenia. Tieto pravidlá a usmernenia nám umožňujú vytvárať komplexné aplikácie rýchlejšie a efektívnejšie než keby sme sa rozhodli začať úplne od nuly. Pravidlá a usmernenia tiež pomáhajú formovať a organizovať náš web alebo webovú aplikáciu. Tým, že JavaScriptové rámce poskytujú štruktúru nášmu kódu sa v súčasnosti už stávajú takmer nevyhnutnou súčasťou všetkých nových frontendových aplikácií. Existuje viacero užitočných JavaScriptových rámcov, ktoré vývojári pravidelne využívajú a niektoré z nich predstavíme v tejto kapitole. Väčšina rámcov je open-source, čo znamená, že ich komunita používateľov neustále zdokonaľuje, takže sú vždy aktuálne. Tieto rámce však nie sú nemenné. Máme slobodu prispôsobiť si vybraný rámec tak, aby vyhovoval potrebám nášho webu alebo aplikácie. [13]

4.1.1 Prečo používať JavaScriptové rámce

Autori JavaScriptových rámcov ich vytvorili hlavne preto aby sebe aj iným vývojárom, ktorí si ich vyberú, uľahčili prácu. Umožňujú programátorom využívať najnovšie funkcie a nástroje JavaScriptu bez potreby namáhavého kódovania často krát tých istých funkcií od základov. Tieto rámce fungujú ako šablóny, ktoré poskytujú základ pre softvérové aplikácie. Zbierajú spoločné zdroje ako sú knižnice, referenčné dokumenty, obrázky a ďalšie, a balia

ich dohromady, aby sme ich mohli efektívne využívať. Vďaka týmto rámcom môžeme do webovej stránky a celého webu pridať lepšiu funkcionalitu ako sú interaktívne užívateľské rozhrania, dynamické aktualizácie obsahu a tiež rozšíriť možnosti webu o pokročilé vizuálne efekty.

4.1.2 JavaScript rámec alebo JavaScript knižnica

Diskusia o rozdieloch medzi rámcom a knižnicou je bežná a častá téma v komunite softvérového vývoja. Hoci odborníci tvrdia, že hranica medzi nimi môže byť nejasná, rozlišovanie medzi týmito dvoma pojmy je užitočné. Kým JavaScriptový rámec predstavuje komplexný súbor nástrojov, ktorý pomáha formovať a organizovať náš web alebo aplikáciu, JavaScriptová knižnica ponúka zbierku predpripravených kódových blokov. Tieto bloky sú primárne určené na pridávanie špecifických funkcií do aplikácie podľa aktuálnej potreby, než na celkové formovanie štruktúry aplikácie. [14]

4.1.3 TypeScript

TypeScript, vyvinutý spoločnosťou Microsoft, je objektovo orientovaný programovací jazyk, ktorý od svojho vzniku v roku 2012 získal významnú pozornosť vo vývojárskej komunite. Ako striktná nadstavba JavaScriptu, TypeScript rozširuje možnosti existujúceho JavaScript kódu pridaním vylepšených funkcií a statickej typovej kontroly, čím sa stáva ideálnou voľbou pre rozsiahlejšie projekty. Dizajn TypeScriptu je zameraný hlavne na zlepšenie škálovateľnosti a udržateľnosti kódu, čo umožňuje efektívnejšiu integráciu do JavaScriptových projektov.

Jednou z kľúčových vlastností TypeScriptu je jeho podpora pre statickú typovú kontrolu, ktorá hoci si vyžaduje dodatočnú prácu pri deklarácii typov, výrazne zvyšuje kvalitu kódu a čitateľnosť. Tento jazyk tiež umožňuje pravé objektovo orientované programovanie s použitím tried, dedičnosti a prístupových modifikátorov ako sú public, private a protected, čím sa líši od prototypovo orientovaného JavaScriptu. Modulárnosť a podpora ES6 funkcií sú ďalšie silné stránky TypeScriptu, ktoré z neho robia atraktívnu voľbu pre moderný webový vývoj.

TypeScript nie je možné priamo interpretovať prehliadačmi a preto je nutné jeho kód skompilovať do JavaScriptu pomocou TypeScript kompilátora (tsc), čo je proces, pri ktorom sa .ts súbory premenia na .js súbory. Táto kompilácia umožňuje vývojárom využívať výhody TypeScriptu bez obmedzenia kompatibility s existujúcimi JavaScriptovými projektmi.

Porovnanie TypeScriptu a JavaScriptu odhalí, že TypeScript pridáva objektovo orientované vlastnosti a statickú typovú kontrolu, ktoré JavaScript prirodzene nepodporuje. Tieto vlastnosti uľahčujú prácu s komplexnými aplikáciami a zvyšujú efektivitu vývojového procesu. Vďaka tomu a podpore zo strany silnej komunity, TypeScript získava na popularite nielen medzi jednotlivými vývojármi ale aj vo veľkých technologických firmách.

Výhody používania TypeScriptu zahŕňajú nielen ľahké učenie a zvýšenú efektivitu vývoja ale aj lepšiu udržateľnosť kódu vďaka opakovateľne použiteľným komponentom a zlepšenému výkonu aplikácií vďaka virtuálnemu DOM. Na druhej strane nevýhodou môže byť potrebný čas na kompiláciu a počiatočná náročnosť pre úplných začiatočníkov bez pevných základov v JavaScripte.

TypeScript sa ukazuje ako výkonný nástroj pre vývoj robustných webových aplikácií a jeho použitie prichádza s mnohými výhodami vrátane zlepšenej kvality kódu, prevencie chýb a prístupu k aktívnej komunite. Jeho schopnosť integrovať sa s existujúcimi JavaScriptovými projektmi a podpora pre ES6 a novšie funkcie robí z TypeScriptu atraktívnu voľbu pre projekt každej veľkosti.[15]

4.1.4 Prehľad najpopulárnejších JavaScriptových rámcov

JavaScript patrí medzi najpopulárnejšie a najširšie používané programovacie jazyky na svete, pričom disponuje väčším počtom rámcov než akýkoľvek iný jazyk. Keďže JavaScriptové rámce sa využívajú tak na klientskej, ako aj na serverovej strane, existuje veľké množstvo rámcov, s ktorými môžeme pracovať. Medzi niektoré z najpopulárnejších rámcov patria React, Angular a Vue.js. Tieto rámce sú obzvlášť cenené vývojármi pre ich flexibilitu, výkonnosť a širokú podporu komunity. React, vytvorený spoločnosťou Facebook, je známy svojou efektívnosťou pri budovaní dynamických užívateľských rozhraní. Angular od Google zase ponúka komplexné riešenie pre vývoj jednostránkových aplikácií (SPA), zatiaľ čo Vue.js sa vyznačuje svojou jednoduchosťou a prispôsobivosťou. Každý z týchto rámcov prináša unikátne výhody a je navrhnutý tak, aby vyhovoval rôznym potrebám a preferenciám vývojárov.

4.1.5 React.js

V oblasti webového vývoja dochádza k neustálemu rastu a vzniku nových technológií, ktoré vývojárom uľahčujú tvorbu užívateľsky atraktívnych webových stránok a aplikácií. Medzi tieto technológie patrí aj React – v súčasnosti jedna z najpopulárnejších JavaScriptových

knižníc zameraných na vytváranie užívateľských rozhraní. React.js, často označovaný len ako React je nástroj, ktorý umožňuje stavbu webových aplikácií zložených z opakovateľne použiteľných komponentov čo zjednodušuje vývojový proces a prispieva k efektívnosti práce vývojárov.

React je knižnica JavaScriptu určená pre budovanie dynamických užívateľských rozhraní. Každá aplikácia vytvorená v Reacte sa skladá z komponentov, ktoré možno opakovane využívať na rôznych miestach aplikácie od navigačnej lišty, cez pätičku, až po hlavný obsah stránky. Tento prístup nielenže uľahčuje vývoj ale aj zvyšuje rýchlosť a výkon aplikácie vďaka možnosti načítavať obsah stránok bez nutnosti ich opätovného načítania.

React sa vyznačuje niekoľkými kľúčovými vlastnosťami a výhodami, ktoré ho odlišujú od iných knižníc a rámcov:

- Učenie a komunita: React je relatívne jednoduchý na naučenie, najmä ak máte základy JavaScriptu. Disponuje rozsiahlou dokumentáciou a aktívnou online komunitou, ktorá neustále vytvára nové zdroje a návody.
- Opakovateľne použiteľné komponenty: Vývojári môžu vytvárať samostatné komponenty s vlastnou logikou, ktoré sa dajú ľahko znovu použiť v rámci aplikácie čo znižuje potrebu opätovne písať rovnaký kód.
- Výkonnosť: Vďaka virtuálnemu DOM (Document Object Model) React efektívne spracúva zmeny v užívateľskom rozhraní čo umožňuje rýchlejšie renderovanie stránok bez ich celkového načítania.
- Rozšíriteľnosť: React samotný sa zameriava len na užívateľské rozhranie. Vývojári majú slobodu vybrať si ďalšie knižnice na spracovanie rôznych aspektov aplikácie ako je smerovanie alebo správa stavov.

React našiel uplatnenie u mnohých vývojárov pracujúcich pre startupy aj etablované spoločnosti, vrátane Facebooku, Netflixu, Instagramu a mnohých ďalších. Jeho schopnosť zvládať vysoký výkon, jednoduchosť použitia a škálovateľnosť sú hlavnými dôvodmi, prečo je React populárny nielen pri vývoji celých produktov ale aj jednotlivých stránok a funkcií.

Pred začatím prác s Reactom je nutné by sme mali dobré základy v JavaScripte, najmä v ES6 funkcionalitách ako sú šípkové funkcie, operátor rest a spread, moduly, deštrukturalizácia, metódy polí, šablónové reťazce a používanie kľúčových slov let a const.

Tieto znalosti sú nevyhnutné pre efektívnu prácu s Reactom a využívanie jeho plného potenciálu.

Medzi výhody a nevýhody používania Reactu by sme mohli zaradiť:

- Výhody: Jednoduchosť učenia, aktívna komunita, bohaté pracovné príležitosti, zvýšená výkonnosť aplikácií a rozšíriteľnosť.
- Nevýhody: Pre úplných začiatočníkov bez pevných základov v JavaScripte môže byť React náročnejší na pochopenie. Okrem toho, pre niektoré funkcie ako je správa stavov alebo smerovanie je potrebné doinštalovať externé knižnice.

S použitím Reactu môžeme budovať dynamické, rýchlo reagujúce webové aplikácie, ktoré zlepšujú užívateľský zážitok a posúvajú možnosti moderného webu do nových výšin. [16]

4.1.6 Angular

Angular je vývojová platforma postavená na TypeScripte, ktorá ponúka robustný základ pre tvorbu škálovateľných webových aplikácií. Ako platforma zahŕňa komponentovo založený rámec pre vývoj webov, kolekciu integrovaných knižníc pokrývajúcich široké spektrum funkcií od smerovania a správy formulárov po komunikáciu klient-server či sadu vývojárskych nástrojov na pomoc pri vývoji, testovaní a aktualizácii kódu. Angular umožňuje vytvárať aplikácie rôznej veľkosti a zložitosti od malých projektov po rozsiahle podnikové aplikácie a je podporený rozmanitou komunitou viac ako 1,7 milióna vývojárov, autorov knižníc a tvorcov obsahu.

Angular je platforma a rámec pre tvorbu single-page aplikácií v HTML a TypeScripte. Jeho architektúra stojí na základných konceptoch ako sú komponenty, ktoré definujú prvky aplikačnej obrazovky, medzi ktorými Angular môže vyberať a ktoré môže modifikovať podľa logiky programu a údajov.

Komponenty využívajú služby na poskytovanie funkcionality na pozadí, ktorá nie je priamo súčasťou užívateľských rozhraní ako je načítavanie údajov. Tieto služby môžu byť injektované do komponentov ako závislosti, čo robí kód modulárnym, opakovateľne použiteľným a efektívnym. Angular zahŕňa aj Router službu na definovanie navigačných ciest medzi súbormi čo umožňuje sofistikované navigačné možnosti v prehliadači.

Angular aplikácia má vždy aspoň jeden komponent nazývaný koreňový komponent, ktorý spája hierarchiu komponentov s DOM (Document Object Model) stránky. Komponenty a

služby sú triedy označené dekorátormi, ktoré Angularu poskytujú metadáta o tom ako ich používať.

Šablóny kombinujú HTML s Angular značkami, ktoré môžu modifikovať HTML elementy pred ich zobrazením. Angular podporuje dvojsmernú dátovú väzbu, čo znamená, že zmeny v DOM sú tiež odzrkadlené v údajoch programu. Šablóny môžu využívať rúry (pipes) na transformáciu hodnôt na zobrazenie, napríklad na formátovanie dátumov a mien podľa lokality užívateľa.

Pre dáta alebo logiku, ktorá nie je spojená s konkrétnym rozhraním ale chceme ju zdieľať medzi komponentmi, musíme vytvoriť triedu služieb. Dekorátor @Injectable() pred definíciou triedy služby poskytuje metadáta potrebné na to, aby bolo možné službu sprístupniť komponentom prostredníctvom injektáže závislostí (DI), čo udržiava triedy komponentov štíhle a efektívne.

Smerovač (router) služba Angularu umožňuje definovať navigačné cesty medzi rôznymi stavmi aplikácie a hierarchiami súborov. Router interpretuje URL cesty podľa pravidiel navigácie, stavu údajov aplikácie a umožňuje navigovať na súbory na základe užívateľských akcií alebo iných stimulov. Táto služba tiež zaznamenáva históriu prehliadača, čím zabezpečuje funkčnosť tlačidiel späť a vpred.

Medzi výhody a nevýhody používania Angularu by sme mohli zaradiť:

- Výhody: Angular poskytuje komplexnú vývojovú platformu s integráciou TypeScriptu pre statickú typovú kontrolu, vysokú modularitu a opakovateľnú použiteľnosť komponentov, pokročilé funkcie pre výkon a podporu pre PWA (podpora pre progresívne webové aplikácie), doplnené o rozsiahlu dokumentáciu a silnú komunitnú podporu.
- Nevýhody: zložitosť a učiaca krivka môžu byť náročné pre začiatočníkov, niektoré úlohy vyžadujú viac „boilerplate“ kódu, pri veľkých aplikáciách môže dôjsť k výkonovým problémom a časté aktualizácie znamenajú potrebu neustáleho doplňovania a obnovovania svojich už nadobudnutých vedomostí. [17]

4.1.7 Vue.js

Vue je moderný JavaScriptový rámec, ktorý sa zameriava na vytváranie užívateľských rozhraní. Jeho hlavnou silou je jednoduchosť a flexibilita, ktoré umožňujú rýchly vývoj aplikácií s použitím deklaratívneho a komponentovo založeného programovacieho modelu.

Vue je postavené na štandardoch HTML, CSS a JavaScriptu, čo z neho robí ľahko prístupné a intuitívne riešenie pre vývojárov rôznych úrovní. Vue umožňuje efektívne spravovať stav aplikácie a UI reaktívnym spôsobom, čo znamená, že akonáhle dôjde k zmene dát, automaticky sa aktualizuje aj zobrazenie bez nutnosti ďalších zásahov. Vue sa vyznačuje použitím Single-FileComponents (SFC), kde každý komponent má svoju logiku, šablónu a štýly definované v jednom súbore. Tento prístup zjednodušuje správu a organizáciu kódu a umožňuje efektívnejšiu prácu v tíme. Vue ponúka dva štýly API pre tvorbu komponentov: Options API a Composition API.

- Options API je založené na deklarácii rôznych možností komponentu (data, metódy, životné cykly), ktoré Vue spracuje.
- Composition API ponúka flexibilnejšie a funkcionálne riešenie pre definovanie logiky komponentov čo je vhodné pre pokročilejšie použitie a umožňuje lepšiu opätovnú použiteľnosť kódu.

Vue je navrhnutý ako postupne adoptovateľný rámec čo znamená, že ho možno použiť pre jednoduché zlepšenia existujúcich stránok až po komplexné single-page aplikácie (SPA) a komplexné riešenia s použitím server-side renderingu (SSR). Táto flexibilita robí z Vue atraktívnu voľbu pre širokú škálu projektov a použití.

Medzi výhody a nevýhody používania Vue by sme mohli zaradiť:

- Výhody: Vue umožňuje rýchly a intuitívny vývoj s dôrazom na jednoduchosť a flexibilitu, vďaka čomu je vhodný pre začiatočníkov a zároveň je dostatočne výkonný aj pre pokročilých vývojárov. Jeho reaktívny systém a komponentovo založený model spolu s možnosťou postupnej integrácie a podporou Single-File Components (SFC) umožňujú efektívne vytváranie užívateľských rozhraní. Silná a aktívna komunita neustále rozširuje Vue o nové nástroje a knižnice čo samozrejme zvyšuje jeho atraktivitu a použiteľnosť.
- Nevýhody: Hoci Vue je navrhnuté tak aby bolo čo najjednoduchšie, existencia dvoch rôznych API (Options API a Composition API) môže byť pre niektorých vývojárov zmätočná a môže vyžadovať dodatočný čas na zorientovanie. Táto dvojitá ponuka API tiež môže viesť k nekonzistencii v kódových štandardoch a prístupoch v rámci tímov alebo projektov. Vue napriek svojej širokej adaptabilite, môže vyžadovať čas a úsilia navyše pri začleňovaní do veľmi špecifických alebo netradičných vývojových prostredí.

Vue poskytuje robustný základ pre vývoj webových aplikácií s dôrazom na jednoduchosť, efektivitu a flexibilitu. Jeho schopnosť prispôsobiť sa projektom rôznej veľkosti a zložitosti, spolu s bohatým ekosystémom a podporou komunity, robí z Vue výbornú voľbu pre vývojárov hľadajúcich efektívny a moderný prístup k vytváraniu webových rozhraní. [18]

4.1.8 Svelte.dev

Svelte je progresívny rámec pre webový vývoj, ktorý mení tradičný prístup k vytváraniu aplikácií. Na rozdiel od klasických rámcov ako React alebo Vue, ktoré vyžadujú načítanie celých knižníc v prehliadači, Svelte pôsobí ako kompilátor. Tento prístup znamená, že väčšina práce typicky vykonávaná prehliadačom sa presúva do kompilačného kroku počas výstavby aplikácie, čo výrazne znižuje nároky na výkon a zlepšuje rýchlosť aplikácií, najmä na mobilných zariadeniach. Svelte nám umožňuje písať stručné komponenty použitím bežných jazykov ako HTML, CSS a JavaScript. Vďaka jednoduchej správe stavov kde sa premenné aktualizujú priamo bez potreby použitia dodatočného kódu alebo háčikov (hooks), je práca s pravidelnými aktualizáciami a zmenami v projekte jednoduchšia. Svelte tiež integruje so Sanity - systém pre správu obsahu na backend-e, čo umožňuje prácu s dátami v reálnom čase a zjednodušuje ich spracovanie v aplikáciách.

Na rozdiel od Reactu, ktorý je známy svojou veľkou komunitnou podporou a rozsiahlou knižnicou, Angularu oceňovaného pre svoju stabilitu a silné nástroje a Vue, ktorý ponúka vyvážené silné stránky vo viacerých aspektoch webového vývoja, Svelte vyniká svojím kompilačným prístupom. Tento unikátny prístup, pri ktorom sa kompilácia uskutočňuje ešte pred spustením aplikácie v prehliadači umožňuje, že aplikácie vyvinuté v Svelte často fungujú rýchlejšie než tie, ktoré sú postavené na Reacte alebo Angulari. Dôvodom je ich menšia náročnosť na zdroje a vyššia efektívnosť.

- Výhody: Svelte poskytuje vysokú efektivitu a rýchlosť vykonávania vďaka svojmu modelu kompilácie, ktorý eliminuje potrebu načítať celú knižnicu v prehliadači. Jeho syntakticky jednoduchý a priamy spôsob správy stavu zjednodušuje vývoj a vedie k čistejšiemu základnému kódu, ktorý je jednoduchšie udržiavateľný a menej náchylný na chyby. Integrácia so systémom Sanity zase rozširuje možnosti práce s obsahom a real-time dátami.
- Nevýhody: Hoci Svelte prináša mnohé výhody jeho inovatívny prístup môže byť pre niektorých vývojárov bariérou, najmä ak sú zvyknutí na tradičnejšie JavaScriptové rámce. Absencia veľkej komunity ako majú napríklad React alebo Angular môže

znamenať menšiu dostupnosť externých zdrojov a podpory. Okrem toho, pri zložitejších aplikáciách môže byť náročnejšie implementovať niektoré pokročilé funkcionality, ktoré sú v iných rámcoch štandardom.

Celkovo Svelte predstavuje atraktívnu alternatívu k tradičným rámcom, s jedinečnými výhodami v rýchlosti, efektívnosti a jednoduchosti vývoja. Jeho schopnosť zjednodušiť vývojový proces a ponúknuť vysoko výkonné aplikácie robí z Svelte významného hráča na poli webového vývoja. [19]

4.2 Balíkovače JavaScriptových modulov

Balíkovač JavaScriptových modulov, známy aj ako module bundler je nástroj, ktorý výrazne zjednodušuje a zefektívňuje vývoj webových aplikácií. V modernom webovom vývoji kde aplikácie často obsahujú veľké množstvo JavaScriptových súborov a závislostí tieto nástroje hrajú kľúčovú rolu.

Balíčkovanie modulov je proces, pri ktorom sa množstvo malých, nezávislých a vymeniteľných častí kódu - modulov - zlúči do jedného alebo viacerých súborov optimalizovaných pre webové prostredie. Tento prístup pomáha riešiť problémy s riadením závislostí a uľahčuje distribúciu a načítavanie aplikácií v prehliadačoch.

Balíkovač modulov prechádza kód aplikácie od definovaného vstupného bodu, vytvára graf závislostí medzi modulmi a na základe toho generuje výstupné súbory podľa konfigurácie. Tento proces nezahŕňa len zlúčenie súborov ale často zahŕňa aj transformáciu kódu (napríklad transpiláciu nových JavaScriptových funkcií do staršej verzie, ktorá je kompatibilná s väčšinou prehliadačov).

Výhody použitia balíkovača modulov:

- Zjednodušenie správy závislostí: Automatizuje proces pridávania a aktualizácie externých knižníc a modulov.
- Optimalizácia výkonu: Znižuje množstvo HTTP požiadaviek potrebných na načítanie zdrojov aplikácie tým, že redukuje počet súborov, ktoré musí prehliadač stiahnuť.
- Zlepšená údržba kódu: Pomocou modularizácie a zlúčenia súvisiacich funkcií do jednotlivých súborov uľahčuje správu a aktualizáciu kódu.

Nevýhody použitia balíkovača modulov

- Náročnosť na nastavenie: Konfigurácia balíkovača modulov môže byť zložitá, najmä pri komplexných aplikáciách s mnohými závislosťami.
- Potreba udržiavania: Ako sa JavaScriptové ekosystémy neustále vyvíjajú je potrebné neustále aktualizovať balíkovače a ich pluginy na najnovšie verzie.
- Možné problémy s výkonom: Nesprávne nastavenie balíkovača môže viesť k nadmernému zväčšeniu výsledných súborov, čo môže negatívne ovplyvniť čas načítavania aplikácie.

Takto základná konfigurácia a pochopenie balíkovačov modulov uľahčuje vývojárom prácu a umožňuje efektívnejšie spravovať moderné webové aplikácie. Či už ide o základné statické stránky alebo komplexné webové aplikácie, balíkovanie modulov prináša značné výhody v optimalizácii a správe zdrojov.

4.2.1 Webpack

Webpack je vysoko pokročilý balíkovač modulov, ktorý sa stal neoddeliteľnou súčasťou moderného vývoja webových aplikácií. Jeho primárnym účelom je transformácia množstva malých, nezávislých súborov a modulov z ktorých je aplikácia zostavená do jedného alebo viacerých optimalizovaných súborov určených pre webové prostredie.

Webpack začína proces s jedným alebo viacerými vstupnými bodmi a na ich základe vytvára graf závislostí. Tento graf obsahuje všetky moduly potrebné pre aplikáciu a ich vzájomné vzťahy. Následne zabalí všetky tieto moduly do jedného balíka (alebo viacerých, ak je to konfigurované), čím minimalizuje potrebný kód a optimalizuje načítavanie aplikácií. Vďaka tomu sú aplikácie rýchlejšie a efektívnejšie, pretože prehliadač musí načítať menej súborov. Webpack je vybavený množstvom načítavačov (loaderov), ktoré umožňujú spracovanie nielen JavaScriptu, ale aj CSS, obrázkov a iných aktív. Tieto loadery transformujú zdroje do formátu, ktorý môže byť efektívne spracovaný a zabalený. Okrem toho, systém pluginov Webpacku rozširuje jeho funkcionalitu, čo umožňuje vývojárom prispôsobiť proces balenia presne podľa ich potrieb, vrátane optimalizácie balíka, správy prostredí a iných úprav.

Webpack tiež exceluje v podpore mikrofrontendovej architektúry, ktorá umožňuje rozdelenie frontendu veľkej aplikácie do menších, samostatne nasaditeľných častí. Toto je dosiahnuté pomocou funkcionalít ako je Module Federation plugin, ktorý umožňuje jednotlivým mikrofrontendom načítať zdieľané závislosti bez potreby duplikácie, čím

výrazne znižuje celkový objem prenesených dát a zlepšuje časy načítania. Module Federation umožňuje dynamické načítanie kódu z rôznych mikrofrontendov, ktoré môžu byť hostované na rôznych serveroch alebo súčastiach aplikácie a zdieľať medzi nimi knižnice alebo komponenty na spoločnom základe. Táto schopnosť robí Webpack ideálnym nástrojom pre vývoj rozsiahlych aplikácií, kde jednotlivé časti môžu byť vyvíjané paralelne rôznymi tímami.

Webpack je nielen robustné riešenie pre správu a balenie modulov v tradičných aplikáciách ale je tiež neoceniteľný v architektúre založenej na mikrofrontendoch, kde jeho flexibilita a konfigurovateľnosť umožňujú efektívne riešiť výzvy spojené s vývojom rozsiahlych webových aplikácií. Jeho schopnosť prispôsobiť sa rôznym vývojovým scenárom, kombinovaná s pokročilými funkciami na správu zdrojov, robí z Webpacku významný nástroj pre každého moderného webového vývojára. [20]

4.2.2 Vite

Vite je moderný nástroj pre vývoj webových aplikácií, ktorý je navrhnutý tak, aby ponúkol rýchlejší a efektívnejší spôsob balenia a vývoja. Jeho meno, ktoré vo francúzštine znamená "rýchlo", odráža hlavný cieľ tohto nástroja, teda poskytnúť bleskovo rýchly nástroj na balenie a načítavanie modulov počas vývoja. Na rozdiel od tradičných balíkovačov ako je Webpack, Vite využíva moderné webové štandardy a optimalizácie ako je ES Modules, čo umožňuje, že veľká časť jeho funkcionality beží priamo v prehliadači bez potreby prekompilácie.

Vite v zásade funguje ako dvojfázový builder. Počas vývoja používa natívne ES moduly pre bleskové načítavanie zdrojov a hot-reloading (okamžité aplikovanie zmien v kóde bez nutnosti obnoviť celú stránku), čím výrazne znižuje čakacie doby pri zmene kódu. Pre produkčné nasadenie paketuje zdroje s použitím Rollup, ktorý je vynikajúci v optimalizácii balíkov a minimalizácii konečného výstupu. Táto hybridná architektúra umožňuje vývojárom využívať rýchlosť a efektivitu počas vývoja, zatiaľ čo zároveň poskytuje robustné a optimalizované súbory pre produkciu. Vite ponúka bohatú sadu pluginov, vrátane podpory pre rôzne frontendové rámce ako React, Vue, a Svelte. Umožňuje tiež jednoduchú integráciu s inými nástrojmi a technológiami cez svoj flexibilný plugin systém, čo vývojárom umožňuje prispôsobiť si svoje vývojové prostredie podľa ich špecifických potrieb.

Vite umožňuje efektívnu prácu s mikrofrontendami prostredníctvom pluginu ako je Vite Plugin Federation. Tento plugin poskytuje podobnú funkčnosť ako Module Federation v Webpacku, čo umožňuje dynamické načítavanie modulov medzi rôznymi mikrofrontendovými projektmi. Tento prístup umožňuje oddelené časti aplikácie zdieľať spoločné knižnice a závislosti bez opätovného načítavania, čo výrazne znižuje veľkosť načítaných zdrojov a zlepšuje výkon aplikácie.

Vite teda predstavuje novú generáciu nástrojov pre vývoj webových aplikácií, ktoré usilujú o maximalizáciu rýchlosti a efektivity vývoja. S jeho pomocou môžu vývojári nielen rýchlejšie testovať a iterovať svoje projekty ale tiež zabezpečiť, že produkčné verzie aplikácií sú optimalizované a pripravené pre užívateľov. Jeho integrácia s modernými webovými štandardami a podpora pre mikrofrontendy cez pluginy ako Vite Plugin Federation ďalej rozširujú jeho použiteľnosť a robia z neho atraktívnu voľbu pre projekty rôznej veľkosti a komplexnosti.[21]

4.3 NPM balíčky

NPM (Node Package Manager) balíčky predstavujú základný stavebný prvok v ekosystéme JavaScriptu. Sú to moduly alebo knižnice kódu, ktoré možno opakovane používať v rôznych projektoch. Tieto balíčky sa zvyčajne distribuujú prostredníctvom NPM repozitára, ktorý je najväčšou svetovou databázou takéhoto softvéru.

Podľa dokumentácie NPM, balíček je kolekcia súborov, ktoré spolu fungujú a poskytujú určitú funkcionalitu. Každý balíček obsahuje package.json súbor, ktorý obsahuje dôležité informácie o projekte, vrátane závislostí, skriptov a konfiguračných detailov. Tieto balíčky možno použiť na vykonávanie špecifických úloh alebo na dodanie širokej škály funkcionalít, od jednoduchých nástrojov na manipuláciu s textom až po komplexné rámce pre vývoj aplikácií. [22]

4.3.1 Načítanie a integrácia balíčkov do aplikácií

NPM balíčky sú načítané do aplikácií cez proces zvaný inštalácia. Počas tejto fázy NPM komunikuje s repozitárom, sťahuje potrebné súbory a uloží ich do adresára node_modules v koreňovom adresári projektu. Tento proces je riadený konfiguračnými súbormi projektu, kde package.json hrá kľúčovú rolu v definovaní, ktoré balíčky a verzie sú potrebné.

Pri vytváraní aplikácií sú NPM balíčky zvyčajne prekladané alebo transformované, čo je proces, ktorý integruje externé moduly do aplikácie. Chyby v týchto balíčkoch môžu viesť

k zlyhaniu buildu, čo môže spôsobiť zdržania alebo vyžadovať dodatočné ladenie. Taktiež, nie je možné priamo použiť balíčky určené pre jednu platformu (napríklad React) v aplikáciách, ktoré sú postavené na inej platforme (napríklad Angular). Toto obmedzenie vyplýva z rozdielov v architektúre a závislostiach jednotlivých rámcov. Tieto špecifika si vyžadujú, aby vývojári boli obozretní pri výbere a integrácii externých balíčkov do svojich projektov a aby zabezpečili kompatibilitu a stabilitu aplikácie.

4.3.2 Lokálne NPM balíčky a Verdaccio

Pre situácie, kde tímy alebo organizácie potrebujú zdieľať a spravovať vlastné, privátne balíčky, môžu využiť nástroje ako Verdaccio. Verdaccio je ľahký súkromný NPM proxy server, ktorý umožňuje hosťovať vlastné balíčky, zatiaľ čo zároveň poskytuje prístup k verejnému NPM repozitáru. Toto je užitočné v prostrediach, kde bezpečnosť alebo interné politiky vyžadujú striktnú kontrolu nad používaným softvérom. Verdaccio podporuje autentifikáciu, takže prístup k balíčkom možno efektívne kontrolovať a regulovať, čo zaisťuje, že len oprávnení používatelia alebo systémy majú prístup k týmto zdrojom.[23]

4.3.3 Porovnanie NPM Balíčkov a Mikrofrontendov

NPM balíčky a mikrofrontendy sú dve technológie, ktoré sa na prvý pohľad môžu zdať podobné, keďže obidve slúžia na modularizáciu funkcií a kódu vo webových aplikáciách. Avšak ich využitie a účely sú značne odlišné a rozumieť týmto rozdielom je kľúčové pre efektívne využitie v projektových architektúrach.

NPM balíčky sú primárne zamerané na opätovné použitie kódu na úrovni závislostí alebo knižníc. Umožňujú vývojárom zdieľať a znovu používať kód cez rôzne projekty, čo znižuje duplikáciu a uľahčuje údržbu. Balíčky môžu obsahovať všetko od malých utility funkcii až po kompletné rámce. Táto modularizácia je však obmedzená na úroveň kódu a nezahŕňa vizuálnu alebo funkčnú izoláciu komponentov vo finálnej aplikácii.

Mikrofrontendy poskytujú metódu na rozdelenie frontendovej časti webovej aplikácie na menšie, nezávislé časti. Každý mikrofrontend môže byť vyvíjaný, testovaný a nasadzovaný nezávisle od ostatných častí systému, čo umožňuje väčšiu flexibilitu a efektivitu vo veľkých tímoch a projektových prostrediach. Mikrofrontendy sú obzvlášť užitočné pri rozsiahlych aplikáciách, kde rôzne tímy alebo oddelenia zodpovedajú za rôzne aspekty aplikácie.

Kedy použiť ktorý prístup?

Výber medzi NPM balíčkami a mikrofrontendami závisí od potrieb projektu:

- NPM balíčky sú ideálne, keď je potrebné znovu použiť logiku alebo funkcie naprieč viacerými projektmi alebo keď chcete integrovať externé knižnice bez nutnosti zaoberať sa komplexnosťou ich kódu. Sú tiež vhodné pre projekty, kde centralizácia závislostí znižuje komplexnosť a zlepšuje udržiavateľnosť.
- Mikrofrontendy sú preferované v prostrediach, kde je potrebné dosiahnuť vysokú mieru izolácie a nezávislosti medzi jednotlivými časťami aplikácie ako sú veľké e-commerce platformy alebo enterprise systémy, kde rôzne tímy pracujú na rôznych častiach aplikácie.

Hoci obidve technológie zlepšujú správu a údržbu kódu tým, že ho rozdeľujú na menšie časti, mikrofrontendy poskytujú dodatočnú flexibilitu a kontrolu na úrovni aplikácie, zatiaľ čo NPM balíčky sú viac zamerané na zdieľanie a opätovné použitie kódu na úrovni závislostí. Rozumieť týmto rozdielom je kľúčové pre správne rozhodovanie o architektúre a nástrojoch v projektoch softvérového vývoja.

II. PRAKTICKÁ ČASŤ

5 PRAKTICKÁ ČASŤ

V tejto kapitole sa podrobnejšie zameriame na praktické využitie technológie mikrofrontendov v rámci vývoja webovej aplikácie. Vývoj webových aplikácií prešiel dlhú cestu od jednoduchých webových aplikácií, ktoré používali HTML, CSS a JavaScript. V súčasnosti sa kladú vysoké nároky na vývojové prostredie webových aplikácií, ktoré sa musia vyrovnávať s rastúcimi požiadavkami na funkcionalitu, užívateľskú prívetivosť a možnosti rozširovania. Technológia mikrofrontendov prichádza ako vhodné riešenie na decentralizáciu zložitých monolitických riešení, zvýšenie efektivity vývoja a nasadenia aplikácií.

V rámci praktickej časti sme sa venovali návrhu a implementácii webovej aplikácie, ktorá je postavená na modernej JavaScriptovej knižnici React a balíkovači Webpack. Kľúčovou súčasťou architektúry tejto aplikácie je plugin Module Federation, ktorý umožňuje bezproblémovú integráciu rôznych mikrofrontendov do jednej hlavnej aplikácie. Tento prístup umožňuje jednotlivým frontendovým tímom pracovať nezávisle na častiach aplikácie, ktoré sú za ne zodpovedné, pričom zároveň zjednodušuje integráciu a testovanie celkovej aplikácie. Naša aplikácia integruje štyri rôzne mikrofrontendy, ktoré sú vyvinuté v populárnych rámcoch: React, Angular, Vue a Svelte. Táto diverzita umožňuje demonštrovať flexibilitu týchto nástrojov ako aj schopnosť technológie mikrofrontendov adaptovať sa na rôzne technologické stacky bez ovplyvnenia celkového chodu aplikácie alebo užívateľského zážitku.

Implementácia mikrofrontendovej architektúry poskytla našej hlavnej aplikácii významné výhody v podobe modularity, izolácie kódu a optimalizácie načítavania zdrojov. Každý mikrofrontend môže byť vyvíjaný, testovaný a nasadzovaný nezávisle, čo znižuje riziko konfliktov v závislostiach a umožňuje rýchlejšie iterácie vývojového cyklu. Zároveň táto architektúra zvyšuje možnosti opätovného použitia kódu a zlepšuje celkovú udržateľnosť projektu. Celkovo prístup využívajúci mikrofrontendy otvára nové perspektívy pre vývoj komplexných webových aplikácií a stáva sa významnou súčasťou moderných vývojových riešení, ktoré reagujú na neustále sa meniace požiadavky trhu digitálnych technológií. Táto kapitola poskytne detailný pohľad na to ako sme tieto technológie využili k vytvoreniu modulárnej a efektívne škálovateľnej aplikácie.

5.1 Hlavná aplikácia

Hlavná aplikácia v našom kóde pomenovaná ako MainApp je naprogramovaná v JavaScriptovom rámci React. Tento rámec sme zvolili preto, že React patrí medzi najrozšírenejšie a najpoužívanejšie JavaScriptové rámce. Jeho výhodou je relatívna jednoduchosť s akou umožňuje implementáciu špecifických požiadaviek. Na rozdiel od konkurenčných rámcov ako je napríklad Angular, React neukladá striktné postupy pre vývoj (bestpractices) a poskytuje väčšiu flexibilitu pri využití čistého JavaScriptu, čo zjednodušuje prispôsobenie aplikácie našim potrebám.

Ako balíkovač sme do aplikácie vybrali Webpack. Ide o nevyhnutný nástroj pre moderný vývoj webových aplikácií, obzvlášť ak pracujeme s mikrofrontendami. Jeho hlavná úloha spočíva v optimalizácii a organizácii zdrojových kódov do efektívne spravovateľných balíčkov, čo zjednodušuje načítavanie a zvyšuje výkon aplikácie. Pri mikrofrontendoch, kde každý modul môže byť vyvíjaný nezávisle, Webpack umožňuje každý mikrofrontend zabaliť do samostatného balíka, ktorý môže byť nasadený a aktualizovaný nezávisle od ostatných častí systému.

Plugin pre správu mikrofrontendov ako je Module Federation plugin, ktorý sme použili aj v tomto projekte, zohráva kľúčovú úlohu v integrácii týchto nezávisle vyvíjaných modulov do jednej koherentnej aplikácie. Tento plugin umožňuje dynamické načítavanie jednotlivých mikrofrontendov za behu, čo znamená, že hlavná aplikácia môže začleniť rôzne mikrofrontendy bez potreby ich predbežného stiahnutia. Tým sa značne znižuje počiatočná veľkosť načítanej aplikácie, čím sa zlepšuje jej celková reaktívnosť a efektívnosť.

Vďaka Module Federation pluginu môžu rôzne tímy pracovať na rôznych častiach aplikácie nezávisle, čo umožňuje paralelný vývoj a minimalizuje riziko konfliktov v závislostiach medzi jednotlivými mikrofrontendami. Tento prístup nie len zvyšuje agilitu vývojového procesu ale tiež poskytuje flexibilitu v nasadzovaní a skúšaní nových funkcií bez rušenia celkovej funkčnosti aplikácie.

Takto Webpack spolu s Module Federation pluginom predstavujú základnú infraštruktúru pre efektívny vývoj a správu mikrofrontendových architektúr, čo umožňuje vytvoriť škálovateľné, modulárne a výkonné webové aplikácie.

5.1.1 Postup

Aplikácia využíva React pre konštrukciu užívateľského rozhrania a Webpack pre súborové zlúčenie a modulárnu správu. React Router Dom je implementovaný na správu navigácie v rámci SPA. Webpack Module Federation umožňuje hlavnej aplikácii načítať kód mikrofrontendov za behu z rôznych zdrojov, ktoré sú hostované na rôznych serveroch. Konfigurácia Webpacku definuje "main" ako hostiteľskú aplikáciu, ktorá vystavuje určité lokálne moduly a načíta externé mikrofrontendy ako sú React, Angular, Vue a Svelte moduly, špecifikované vo svojich konfiguráciách. Tieto externé zdroje sú prístupné cez definované URL. Všetky zdieľané závislosti sú centralizované a manažované cez package.json, čo znižuje redundanciu a uľahčuje správu verzii.

Prostredníctvom skriptov v package.json je možné spustiť vývojový server s hot-reloadingom, ktorý podporuje rýchly vývoj a produkčný build, ktorý optimalizuje aplikáciu pre nasadenie. Verdaccio, lokálny NPM registry, je použitý na interné spravovanie privátnych balíkov.

Hlavná aplikácia poskytuje rámcový layout, do ktorého sa dynamicky vkladajú mikrofrontendy. Pomocou ReactSuspense a Lazy loading je možné asynchrónne načítať komponenty, čo zlepšuje časy načítania a efektivitu vykonávania. Každý mikrofrontend je zodpovedný za svoju časť aplikácie a môže byť nezávisle aktualizovaný a udržiavaný.

Aplikácia efektívne rieši chyby pri načítavaní mikrofrontendov pomocou fallback komponentov, ako "OopsComponent", ktoré zobrazujú chybové hlásenia a poskytujú užívateľské rozhranie pre možné riešenia problémov. Toto je kľúčové pre zabezpečenie robustnosti aplikácie v produkčnom prostredí.

5.1.2 Ukážka najdôležitejších častí Main aplikácie

Prvý, najhlavnejší a zároveň najdôležitejší bod, ktorý si ukážeme je ten ako aplikácia môže pracovať s mikrofrontendami. Ako sme už spomínali v tejto práci sme si zvolili balíkovač Webpack a jeho rozšírenie o Module Federation Plugin, ktorý nám dovoľuje jednotlivé časti kódu ponúkať ako mikrofrontend alebo na druhej strane prijímať mikrofrontendy z vonku. V nasledujúcom obrázku si ukážeme ako vyzerá kód takejto konfigurácie.

```
new ModuleFederationPlugin({
  name: "main",
  filename: "remoteEntry.js",
  remotes: {
    reactMf: "ReactMicrofrontend@http://localhost:3001/remoteEntry.js",
    angularMf: "angularMicrofrontend@http://localhost:3002/remoteEntry.js",
    vueMf: "vueMicrofrontend@http://localhost:3003/remoteEntry.js",
    svelteMf: "svelteMicrofrontend@http://localhost:3004/remoteEntry.js",
  },
  shared: {
    ...packageJson.dependencies,
    ...packageJson.sharedDependencies,
  },
}),
```

Obrázok 10: Konfigurácia Module Federation Pluginu v MainApp

Tento kód je súčasťou súboru webpack.config.js v main aplikácií, ktorá v našom projekte slúži ako hlavná aplikácia, ktorá spája všetky mikrofrontendy a iné časti do jednej aplikácie. Ako môžeme vidieť najskôr si inicializujeme tento ModuleFederationPlugin a následne do jeho tela vložíme konfiguráciu.

Prvé sme si definovali meno tohto hostiteľa alebo kontajnera, v tomto prípade "main".Toto meno slúži ako identifikátor v ekosystéme Module Federation pre konkrétne rozhranie, kde sú hostované alebo zdieľané moduly.

Druhý riadok určuje názov súboru "remoteEntry.js", ktorý obsahuje informácie o vystavených moduloch a zdieľaných závislostiach. Tento súbor slúži ako manifest a vstupný bod pre načítavanie modulov z tohto hostiteľa.

Objekt "remotes" definuje kde jednotlivé kľúče a k nim priradené hodnoty, ktoré odkazujú na externé mikrofrontendové aplikácie. V tomto prípade máme štyri pripojené mikrofrontendy k hlavnej aplikácií a to reactMicrofrontend ktorý beží na localhost:3001, angularMicrofrontend na porte 3002, vueMicrofrontend na porte 3003 a svelteMicrofrontend na porte 3004.

Ako posledný v našej konfigurácií je "shared". Tento riadok definuje zdieľané závislosti medzi rôznymi mikrofrontendami. Takéto zdieľanie využívame z dôvodu, že zdieľanie závislostí znižuje celkovú veľkosť balíčkov a zabezpečuje, že všetky aplikácie používajú rovnaké verzie zdieľaných knižníc, čím sa optimalizuje načítavanie a synchronizuje správanie medzi aplikáciami.

Na ďalšom obrázku si ukážeme taktiež konfiguračný kód Module Federation Pluginu z Webpacku avšak zo strany vkladaného mikrofrontendu.

```
new ModuleFederationPlugin({
  name: "ReactMicrofrontend",
  filename: "remoteEntry.js",
  exposes: {
    "./LayoutTemplate": "./src/layout/LayoutTemplate.js",
    "./MiniReactApp": "./src/miniApp/miniReactApp.js",
    "./MiniAppInput": "./src/miniApp/miniAppInput.js",
  },
  remotes: {
    footerVue: "FooterVue@http://localhost:3010/remoteEntry.js",
  },
  shared: {
    ...packageJson.dependencies,
    ...packageJson.sharedDependencies,
  },
}),
```

Obrázok 11: Konfigurácia Module Federation Pluginu v React mikrofrontende

Tu môžeme vidieť, že konfigurácia na strane mikrofrontendu vyzerá veľmi podobne len s tým rozdielom, že nám pribudol záznam "exposes“. Tento záznam definuje, ktoré moduly (komponenty alebo súbory) má náš projekt vystaviť, aby boli dostupné pre iné projekty alebo aplikácie v rámci pluginu Module Federation. V tomto konkrétnom príklade vidíme, že náš ReactMicrofrontend vystavuje tri moduly, ktoré budú následne k dispozícii hlavnej aplikácií ale aj ostatným modulom (LayoutTemplate, MiniReactApp, MiniAppInput). ReactMicrofrontend sme spravili tak, že nie je len vysielateľom svojich komponentov ale aj prijímateľov. A to konkrétne modulu footerVue z mikrofrontendu FooterVue bežiaceho na porte 3010. Týmto sme si chceli deklarovať, žehoci ktorý modul či aplikácia môžu byť tak ako vystavovateľ svojich komponentov tak aj prijímateľom iných.

Na začiatku sme si definovali "name“, ktorý je v tomto ekosystéme dôležitý na identifikáciu konkrétneho mikrofrontendu. Ak sa pozrieme späť na konfiguráciu v hlavnej aplikácií môžeme vidieť, ako sa toto meno používa vo volaní konkrétneho mikrofrontendu pred znakom @ po ktorom nasleduje adresa servera:

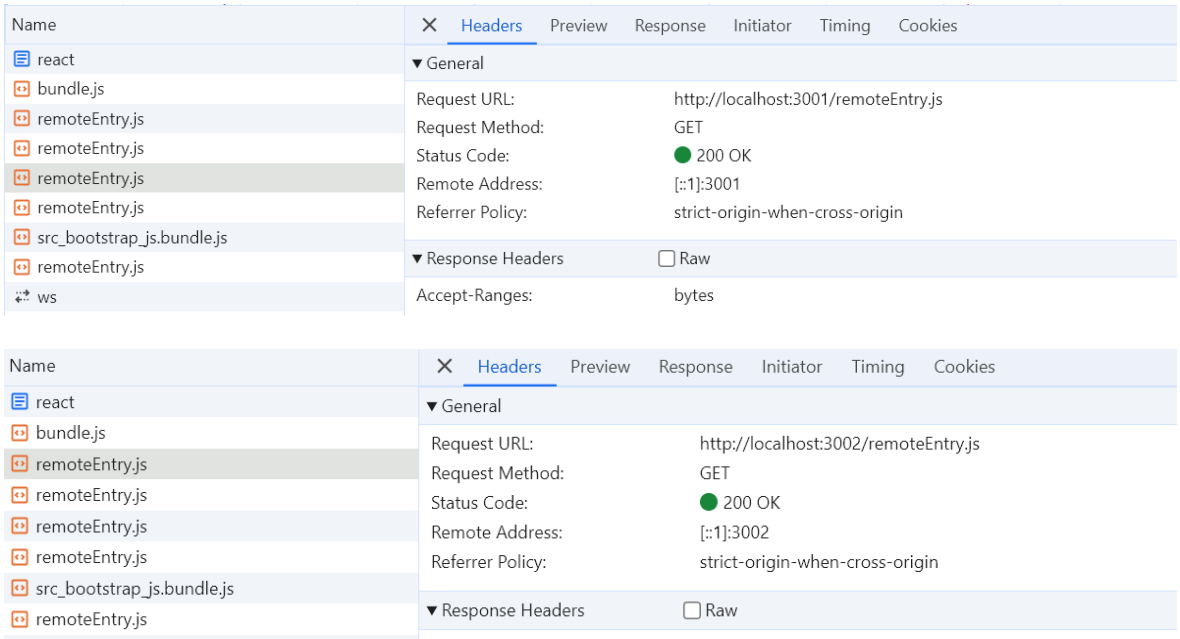
```
reactMf:"ReactMicrofrontend@http://localhost:3001/remoteEntry.js",
```

Kľúč "filename" tak isto ako v hlavnej aplikácií definuje súbor (remoteEntry), ktorý slúži ako centrálny bod.

Taktiež "shared" má rovnakú funkciu ako v konfigurácií hlavnej aplikácie a to definovať zdieľané závislosti.

Po nastavení ModuleFederationPlugin je dôležité overiť, či je integrácia mikrofrontendov správne nakonfigurovaná a funguje ako očakávame. Jedným z najefektívnejších spôsobov ako to môžeme urobiť je využitie Network panela vo vývojárskych nástrojoch v našom prehliadači. Tento nástroj umožňuje vizualizovať a analyzovať všetky sieťové požiadavky, ktoré naša aplikácia vykonáva, vrátane načítania skriptov z mikrofrontendov.

Ukážka :



Obrázok 12: Ukážka načítavania remoteEntry v Network aplikácie v prehliadači

Na základe zobrazených obrázkov z Network panela prehliadača môžeme vidieť, že každý mikrofrontend úspešne posiela svoj remoteEntry.js súbor, čo potvrdzuje kľúčový aspekt architektúry mikrofrontendov. Tento prístup nám umožňuje efektívne škálovať našu aplikáciu tým, že jednotlivé časti aplikácie môžeme načítať z rôznych zdrojov. Vďaka tomu, že každý mikrofrontend má svoj vlastný vstupný bod, vieme nezávisle aktualizovať a nasadiť rôzne časti nášho systému bez toho, aby sme museli prekompilovať alebo reštartovať celú aplikáciu. Táto modularita a flexibilita sú hlavnými výhodami používania

mikrofrontendov, keďže nám umožňujú efektívne rozdeliť aplikáciu na menšie, ľahko spravovateľné a nezávisle rozvíjateľné časti.

5.2 React Mikrofrontend

V predošlej kapitole sme si už spomenuli mikrofrontendy s ktorými pracujeme a ako ich vystavujeme alebo načítavame z iných zdrojov. Teraz si priblížime jednotlivé špecifiká konkrétnych rámcov a pozrieme sa na to čo potrebujeme vedieť ak v nich chceme naprogramovať funkčný mikrofrontend.

Ako prvý si ukážeme mikrofrontend v Reacte. V tomto prípade máme výhodu, že aj samotná hlavná aplikácia beží na rámci React, takže komunikácia medzi nimi je jednoduchšia a priamejšia. Nie je potreba zaobaľovania tohto mikrofrontendu, pretože ho vieme pohodlne vložiť do hlavnej aplikácie pomocou React.lazy, čo umožňuje dynamické načítavanie bez potreby dodatočných konfigurácií. V prípade mikrofrontendov založených na iných technológiách, ako sú Angular, Vue alebo Svelte, je potrebné realizovať špeciálne 'bootstrapovanie', teda inicializáciu týchto aplikácií v rámci React aplikácie. Tento proces často zahŕňa vkladanie základného root elementu, (napríklad app-root pre Angular) a následné spustenie bootstrapovacej funkcie, ktorá sprístupní a integruje mikrofrontend do hlavnej aplikácie. Tento krok je zložitejší a vyžaduje špecifickú prípravu a správu interakcií medzi rôznymi technológiami preto si o ňom povieme a ukážeme riešenie v kapitolách o rámcoch Angular, Vue a Svelte.

ReactMicrofrontend, ako sme si už ukázali na obrázku 11 vystavuje tri komponenty

- LayoutTemplate
- MiniReactApp
- MiniAppInput

a jeden modul, nazvaný FooterVue si načítava z adresy 3010.

5.2.1 LayoutTemplate

Tento komponent slúži ako hlavný UI obaľovač aplikácie. Konkrétne v sebe zahŕňa hlavičku stránky, ktorej súčasťou je aj navigačné menu, ďalej obsahovú časť umiestnenú v strede obrazovky a pätičku stránky umiestnenú na spodku obrazovky.

```
return (
  <div className="layout-root">
    <div>
      <div className="layout-content-division">
        <Navbar items={props}></Navbar>
        <div className="page-view-wrapper">{props.children}</div>
        <Footer />
      </div>
    </div>
  </div>
);
```

Obrázok 13: Html časť kódu v súbore Layout

Hlavička obsahuje nadpis, ktorý nás informuje o tom, že sa jedná o časť naprogramovanú v rámci React a napravo od tohto nadpisu sa nachádza navigačné menu cez ktoré sa môžeme pohybovať po stránke a prepínať ním jednotlivé podstránky našej aplikácie. Smerovanie po stránke je zabezpečené použitím knižnice React Router DOM, ktorý je v súčasnosti najpoužívanejším smerovačom pre rámec React.

Pri programovaní časti LayoutTemplate sme sa snažili zachovať univerzálnosť a možné znovu použitie tejto komponenty v inej aplikácií a preto sme sa rozhodli nepísať v rámci tohto mikrofrontendu cesty pre smerovač priamo do kódu ale nechať si ich posielať z hlavnej aplikácie a tým aj demonštrovať možnosti komunikácie medzi hlavnou aplikáciou a mikrofrontendom v Reacte. Keďže sa jedná o rovnaký rámec tak posielanie a čítanie dát medzi týmito dvoma samostatnými časťami nie je problém a je možne takúto komunikáciu viesť cez takzvané "props“ (skratka od properties, preklade premenné). Využitým preposielaných dát so smerovacími adresami dosiahneme, že hlavná aplikácia bude vždy rozhodovať o tom aké sú jej podstránky a distribuovaná služba LayoutTemplatetak bude len vykresľovať a štýlovať navigačné menu. Takýto spôsob je vhodný pre veľké portály, ktoré majú viacero menších aplikácií bežiacich na rôznych doménach, avšak chceme aby ich vzhľad vždy vychádzal z rovnakých pravidiel a aktív (assetov).

Ukážka, ako vyzerajú dáta v hlavnej aplikácií ktoré chceme preposielať do mikrofrontendu zodpovedného za rozloženie (layout) stránky. V našom prípade do LayoutTemplate súboru.

```
const menuItems = [
  {
    id: 0,
    label: "React",
    link: "/react",
  },
  /* ... Ďalšie podstránky */
  {
    id: 5,
    label: "NPM",
    link: "/npm",
  },
];
export default menuItems;
```

Obrázok 14: Konfigurácia navigačného menu v menuItems

Na obrázku vidíme klasické pole objektov v ktorom si posielame údaje ako id (identifikátor), label (zobrazený text v menu) a link (adresa presmerovania).

Toto pole následne načítame do súboru App.js v MainApp a vložíme ho ako parameter do pre LayoutTemplate

```
import React, { lazy, Suspense } from "react";
import "./main-app.scss";
import LayoutTemplateFallback from "./components/LayoutTemplateFallback";
import menuItems from "./layout/menuItems.jsx";
import Routes from "./layout/Routes.js";

const LayoutTemplate = lazy(() =>
  import("reactMf/LayoutTemplate").catch(() => ({
    default: LayoutTemplateFallback,
  }))
);

export default function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <div className="main-app">
        <div className="title">
          <div className="img" />
          <p>React Main Application</p>
        </div>
      </div>
      <LayoutTemplate menuItems={menuItems}>
        <Routes />
      </LayoutTemplate>
    </Suspense>
  );
}
```

Obrázok 15: Súbor App.js v MainApp

Na priloženom obrázku je možnosť aj vidieť spôsob ako sa dá v rámci React aplikácie vkladať React mikrofrontend o ktorom sme si písali v úvode kapitoly. Import časť lazy funkcie nám hovorí z akej adresy načítava konkrétny komponent (LayoutTemplate komponent) a druha časť .catch nám zabezpečuje, že v prípade ak na adrese reactMf/LayoutTemplate nenájde čo hľadá alebo daná adresa (server) neodpovedá tak načíta default. Do tejto predvolenej časti si vložíme komponent ktorý sa zobrazí ak LayoutTemplate nie je dostupný. Zvyčajne sa používajú ospravedlňujúce hlášky ako „Oops something wrong“ o ktorých si povieme v ďalšej časti.

Ďalej na obrázku môžeme vidieť vloženie LayoutTemplate komponentu do HTML časti aj pridanými vstupnými dátami menuItems. Takto jednoducho si vieme naše pole objektov poslať z hlavnej aplikácie do mikrofrontendu (iba ak sú obe strany rovnaký rámec). Spracovanie takýchto dát na druhej strane potom vyzerá nasledovne :

```
import React from "react";
import App from "../components/App";

export default function Template(props) {
  return <App {...props} />;
}
```

Obrázok 16: Súbor LayoutTemplate v React mikrofrontende

Tu môžeme vidieť že Template funkcia tieto dáta očakáva ako parameter props, ktorý následne pošle do súbor App.js kde je definovaný react-router-dom.

```
<Router>
  <Switch>
    <Route exact path="/">
      <Redirect to="/all" />
    </Route>
    <Layout {...props} />
  </Switch>
</Router>
```

Obrázok 17: Definovanie Routra v App React mikrofrontendu

V tejto časti kódu sme si definovali Router, ktorý nám obaľuje Layout súbor a posiela do neho dáta. Ako vyzerá Layout súbor sme si ukázali na obrázku 13.

Ako posledný súbor pre vykreslenie nášho navigačného menu v mikrofrontende teda zostal Navbar ktorý obsahuje funkciu .map na spracovanie poľa dát a vykreslenie jeho obsahu

```
<nav className="navbar-nav">
  {menuItems.map((item) => (
    <NavLink
      key={item.id}
      to={item.link}
      className="nav-link"
      activeClassName="active"
    >
      <span>{item.label}</span>
    </NavLink>
  ))}
</nav>
```

Obrázok 18: Vykreslovanie jednotlivých položiek menu v Navbare

Takto sme docielili, že naša MainApp rozhoduje o tom, aké položky menu budú na stránke, avšak o ich vykresľovanie a vzhľad sa stará mikrofrontend v Reacte. Týmto sme si aj ukázali, že škálovanie aplikácie na samostatne modulárne časti nemusí byť prekážkou, ak potrebujeme aby tieto jednotlivé dielčie časti spolupracovali a komunikovali medzi sebou. Ako posledný kúsok skladačky k funkčnému smerovaču nám zostáva súbor Routes, ktorý nám definuje čo majú jednotlivé adresy zobrazovať. Doterajší postup nám zabezpečil, aby sa nám stránka prepínala medzi jednotlivými podstránkami, avšak potrebujeme ešte priradiť, aký obsah sa na danej podstránke má zobraziť. Na tento účel nám bude slúžiť spomínaný Routes.js ktorý je súčasťou MainApp a jeho jednotlivé cesty vyzerajú nasledovne.

```
<Route
  path="/angular"
  render={useCallback(
    (props) => (
      <Suspense fallback={<div>Loading Microfrontend...</div>}>
        <AngularMicrofrontend></AngularMicrofrontend>
      </Suspense>
    ),
    []
  )}
/>
```

Obrázok 19: Ukážka smerovania podstránky pri adrese /angular v Routes

Táto časť nám priradí obsah Angular mikrofrontendu k adrese /angular. Týmto istým spôsobom postupujeme aj pri smerovaní na ostatné mikrofrontendy.

5.2.2 MiniReactApp a MiniAppInput

MiniReactApp a MiniAppInput sú dva menšie komponenty, ktoré využívame na podstránke AllExamples. MiniAppInput je komponent, ktorý spravuje input na tejto podstránke. Tento input slúži na vkladanie reťazcov znakov, ktoré sa následne preposielajú do jednotlivých mini komponentov naprogramovaných vo všetkých rámcoch s ktorými v tejto práci pracujeme. MiniReactApp je na druhej strane jedným z prijímateľov týchto reťazcov znakov vkladaných cez input predchádzajúcej komponenty. Viac o tom ako to funguje si povieme v kapitole pre podstránku AllExamples.

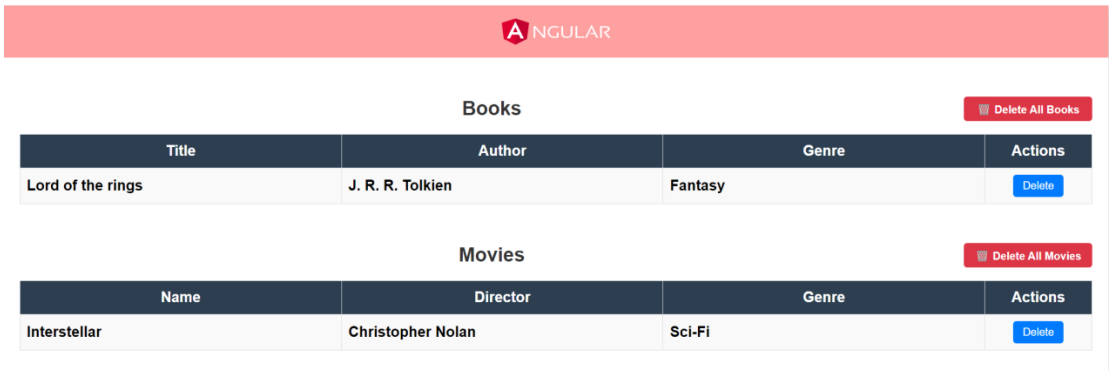
5.3 Angular Mikrofrontend

V nasledujúcej kapitole si popíšeme spôsob ako je možné využívať mikrofrontend, ktorý je naprogramovaný v inom rámci ako je hlavná aplikácia. Ako prvý sme si vybrali Angular. Aj pri tomto rámci sme si ako balíkovač vybrali Webpack a plugin Module Federation Plugin. Voľba rámca nemá na konfiguráciu tohto pluginu vplyv takže všetky nadstavenia spĺňajú pravidlá ako aj pri predošlom mikrofrontende. Rozdiel badať iba pri zdieľaných závislostiach, kde sme vložili závislosti, ktoré potrebujem na chod Angular projektu.

```
new ModuleFederationPlugin({
  name: "angularMicrofrontend",
  filename: "remoteEntry.js",
  exposes: {
    "./angularMf": "./src/loadApp.ts",
    "./miniAngularMf": "./src/loadMiniApp.ts",
  },

  shared: share({
    "@angular/core": {
      singleton: true,
      strictVersion: true,
      requiredVersion: "auto",
    },
    "@angular/common": {
      singleton: true,
      strictVersion: true,
      requiredVersion: "auto",
    },
    "@angular/common/http": {
      singleton: true,
      strictVersion: true,
      requiredVersion: "auto",
    },
    "@angular/router": {
      singleton: true,
      strictVersion: true,
      requiredVersion: "auto",
    },

    ...sharedMappings.getDescriptors(),
  }),
}),
sharedMappings.getPlugin(),
```

Obrázok 20: Konfigurácia Module Federation Pluginu v Angular mikrofrontende

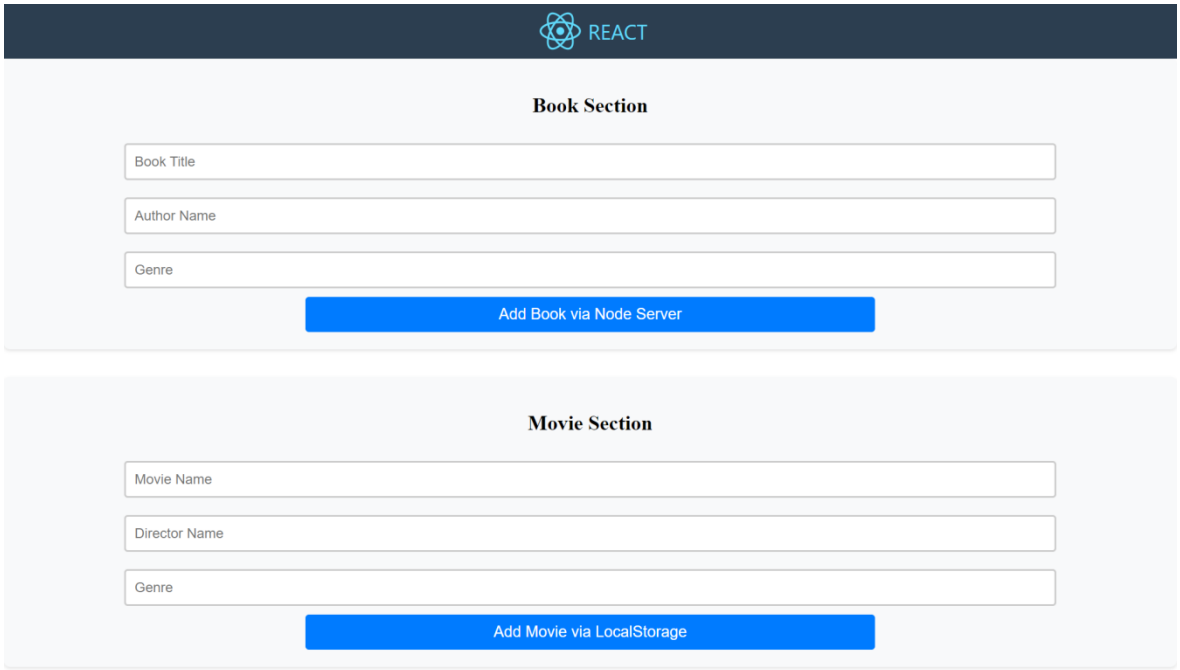
Z nasledovného obrázku vidíme, že AngularMikrofrontend vystavuje dva komponenty. Prvým je angularMf, ktorý je hlavným obsahom pre podstránku dostupnú na adrese /angular.



Obrázok 21: Podstránka Angular s hlavným obsahom Angular mikrofrontendu

Táto podstránka obsahuje dve tabuľky.

Tabuľka Books zobrazuje naše knihy, ktoré sme do tejto tabuľky pridali cez input element v reactovej časti v sekcii Book section, ktorú môžeme nájsť na adrese /react. Ako stĺpce má title (názov knihy), author (kto knihu napísal), genre (akého žánru kniha je) a actions (akcie ktoré vieme vykonávať s dátami v tabuľke). Akcie ktoré vieme vykonať nad touto tabuľkou sú vymazanie konkrétneho záznamu (Delete tlačidlo v stĺpci Actions na riadku daného filmu ktorý chceme vymazať) a vymazanie celej tabuľky (červené Delete All Books tlačidlo umiestnené vľavo hore nad konkrétnou tabuľkou, ktorú chceme vymazať). Tabuľka Movies zobrazuje naše filmy, ktoré sme do tejto tabuľky pridali taktiež cez input element v reactovej časti aplikácie v sekcii Movie Section na adrese /react. Táto druhá tabuľka má rovnakú funkcionalitu ako tabuľka kníh. Mazanie jej záznamov funguje rovnako.



Obrázok 22: Podstránka React s inputmi na pridávanie kníh a filmov

Okrem obsahu je hlavný rozdiel medzi týmito tabuľkami spôsob ukladania dát.

5.3.1 Ukladanie dát cez Node server

Prvým spôsobom, ktorý využívame pri vkladaní kníh je ukladanie dát do Node.js. Pre potreby tohto prístupu sme si museli vytvoriť Node server, ktorý sme si nazvali Api-server a beží na porte 3100. Ide o jednoduchý Node server ktorý obsahuje súbor server.js. V tomto súbore máme naprogramované všetky akcie ktoré chceme nad týmito tabuľami vykonávať ako get, post a delete.

```
app.delete("/data/:category/:index", (req, res) => {
  const { category, index } = req.params;
  if (dataStorage[category] && dataStorage[category].length > index) {
    dataStorage[category].splice(index, 1);
    res.status(200).json({ message: "Item deleted successfully" });
  } else {
    res.status(404).send("Item not found");
  }
});
```

Obrázok 23: Ukážka funkcie vymazania záznamu v tabuľke v kóde Api serveru

Ako môžeme vidieť, tak operácie nad prácou v tabuľke sú písané v čistom JavaScripte, čo nám dovoľuje používať ich naprieč všetkými mikrofrontendami bežiacimi na rôznych rámcoch bez nutnosti vytvárania obalovačov.

Aby sme vedeli v rámci Angular pristupovať k koncovým bodom (endpointom) nášho Api-servera museli sme si vytvoriť servis súbor (data-service) v ktorom definujeme všetky akcie ktoré chceme z api servera používať.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private baseUrl = 'http://localhost:3100';

  constructor(private http: HttpClient) {}

  fetchData(category: string): Observable<any> {
    return this.http.get(`${this.baseUrl}/data/${category}`);
  }

  deleteData(category: string, index: number): Observable<any> {
    return this.http.delete(`${this.baseUrl}/data/${category}/${index}`);
  }

  deleteAllData(category: string): Observable<any> {
    return this.http.delete(`${this.baseUrl}/data/${category}`);
  }
}
```

Obrázok 24: Service súbor pre spracovanie volaní na a zo serveru pre Angular aplikáciu

Keď už náš modul komunikuje so serverom prostredníctvom servisu, môžeme prejsť k jeho použitiu. Dosiahneme to tak, že do komponentu v ktorom chceme tieto akcie vykonávať, túto servisu najskôr načítame

```
import{DataService}from'./data-service.service';
```

a potom jednoducho zavoláme ako do našej funkcie v rámci nášho komponentu.

```
deleteItem(index: number): void {
  this.dataService.deleteData('books', index).subscribe({
    next: (response) => {
      this.data = this.data.filter((_: any, idx: number) => idx !== index);
      this.cdr.detectChanges();
    },
    error: (error) => {
      console.error('Error deleting the item:', error);
    },
  });
}
```

Obrázok 25: Funkcia deleteItem na vymazanie záznamu z tabuľky v Angular aplikácií

5.3.2 Ukladanie dát do LocalStorage

Druhý spôsob, ktorý používame na ukladanie dát je ukladanie do pamäte prehliadača, teda LocalStorage. Toto riešenie je značne jednoduchšie, pretože nevyžaduje vytváranie svojho úložiska pre dáta, ale využíva existujúce vyhradené miesto v prehliadači. Na ukladanie a čítanie dát nám stačia funkcie setItem a getItem. Funkcie setItem budeme používať v reactovej časti aplikácie kde z formulára zozbierame data a príkazom,

```
localStorage.setItem("movies",JSON.stringify(existingMovies));
```

kde "movies" je názov kľúča pod, ktorým budú v pamäti uložené naše dáta a "existingMovies" je zoznam našich filmov.

Následne na strane Angular modulu môžeme tieto filmy načítať z lokálnej pamäte prehliadača príkazom getItem

```
this.movies=JSON.parse(localStorage.getItem('movies') ||'[]');
```

5.3.3 Zhrnutie práce s dátami

Ukladanie dát na server aj ukladanie dát do pamäte prehliadača sú ilustračné spôsoby, ako by sa dali manažovať dáta medzi mikrofrontendami. V praxi by sme volili spôsob ukladania podľa konkrétnej situácie, čo potrebujeme spraviť a aké špecifické podmienky pri tom musíme dodržať.

LocalStorage je užitočný pre uchovávanie užívateľských preferencií a stavu rozhrania na webe, ako napríklad zapamätanie si vybraného jazyka alebo témy na webových stránkach. Veľa krát sa s takýmto riešením stretávame aj pri e-shope, kde môžeme použiť LocalStorage na uchovanie obsahu nákupného košíka, aby užívateľ pri ďalšej návšteve stránky našiel svoje vybrané produkty stále v košíku. Nevýhodou LocalStoragu je jeho obmedzená bezpečnosť, pretože dáta v ňom uložené nepodliehajú žiadnemu šifrovaniu a viditeľnosť len na jednom zariadení, čo ho robí nevhodným na ukladanie citlivých údajov, alebo pre aplikácie vyžadujúce synchronizáciu údajov medzi viacerými zariadeniami

Ukladanie na server (na backend) je štandardnou praxou pre uchovávanie dát, ktoré chceme mať trvalo zachované a bezpečne uložené. Tento prístup je obľúbený kvôli jeho schopnosti centralizovať údaje, ktoré sú potom prístupné z rôznych bodov aplikácie a zariadení, čo umožňuje ľahšiu správu a integráciu. Zároveň umožňuje robustné bezpečnostné opatrenia na ochranu týchto dát. Avšak, ukladanie na server nie je vždy ideálne pre dočasné údaje, alebo pre údaje, ktoré vyžadujú časté a rýchle aktualizácie. Každý prístup k týmto údajom vyžaduje žiadosť (request), teda sieťovú komunikáciu, ktorá môže byť časovo náročná a môže vytvárať zbytočnú záťaž na server. Toto môže viesť k spomaleniu reakcie aplikácie, obzvlášť ak server spracováva veľké množstvo požiadaviek súčasne, alebo ak sú dáta fyzicky uložené na vzdialenom serveri. V prípade, že vaša aplikácia vyžaduje rýchlu a častú manipuláciu s dočasnými údajmi, môže byť efektívnejšie ukladať takéto informácie lokálne na klientskom zariadení, alebo použiť iné technológie ako napríklad in-memory databázy, ktoré poskytujú rýchlejší prístup a menej záťaže na sieťové prostriedky.

Obe tieto riešenia, LocalStorage aj Node server sme vybrali, pretože klasické metódy správy stavu majú problém s udržiavaním kontinuity dát medzi izolovanými inštanciami rôznych mikrofrontendov. Keďže každý mikrofrontend môže byť postavený na inom JavaScriptovom rámci s vlastnými mechanizmami správy stavu, nedochádza pri navigácii medzi mikrofrontendami k prirodzenému zdieľaniu stavových informácií. Tento problém nás viedol k implementácii centralizovanejšieho prístupu, kde ukladanie dát na server alebo do LocalStorage poskytuje spoločnú vrstvu, cez ktorú môžu všetky časti aplikácie efektívne komunikovať a udržiavať užívateľské dáta konzistentné a prístupné naprieč rôznymi užívateľskými reláciami a rámci

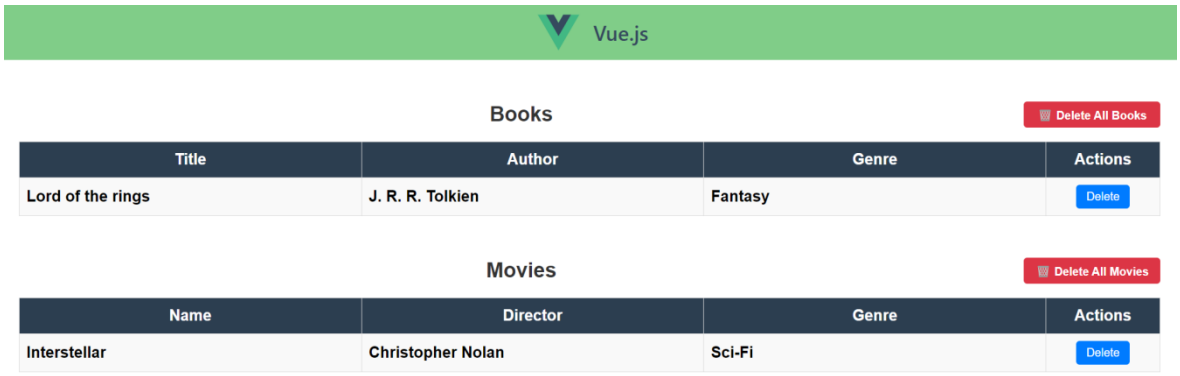
5.4 Vue Mikrofrontend

Vue mikrofrontend je ďalším mikrofrontendom postavenom na JavaScriptovom rámci Vue.js vo verzií 3. Aj pri tomto rámci využívame rovnaký balíkovač a plugin pre správu mikrofrontendov. Ako sme si už pri Angulare spomenuli výber rámca nemá zásadný vplyv na konfiguráciu Module Federation pluginu preto vysvetľovanie konfigurácie preskočíme.

```
new ModuleFederationPlugin({
  name: "VueMicrofrontend",
  filename: "remoteEntry.js",
  exposes: {
    "./VueMainMf": "./src/bootstrap",
    "./VueMiniApp": "./src/miniApp/miniBootstrap",
  },
  shared: packageJson.dependencies,
}),
```

Obrázok 26: Konfigurácia Module Federation Pluginu vo Vue mikrofrontende

Z obrázku konfigurácie vidíme, že VueMicrofrontend podobne ako AngularMicrofrontend vystavuje dva komponenty. Prvým je VueMainMf, ktorý distribuuje podobne ako v Angulare tabuľku kníh a filmov. Rozhodli sme sa zvoliť tento koncept, že každý mikrofrontend bude v hlavnej časti aplikácie na samostatnej adrese spracovávať tie isté dáta do tabuľky aby bolo možné porovnať technologické odlišnosti medzi riešeniami v jednotlivých rámcoch.



Obrázok 27: Podstránka Vue s hlavným obsahom Vue mikrofrontendu

Druhým vystavovaným komponentom je VueMiniApp o ktorom si povieme v kapitole AllExamples.

Čo je však kľúčovým bodom pri správe mikrofrontendov načítavaných do aplikácie, ktorá beží na inom rámci ako daný mikrofrontend je proces, ktorý by sme mohli nazvať ako dynamické nasadenie (dynamicmounting). Tento proces zabezpečuje, že mikrofrontend môže byť načítaný, inicializovaný a správne integrovaný do hostiteľskej aplikácie bez toho, aby došlo k konfliktom v rámci alebo k problémom so závislosťami.

Na úrovni kódu Vue, funkcia mount v súbore bootstrap je zodpovedná za inicializáciu Vue aplikácie v špecifikovanom DOM elemente, ktorý dostane ako argument. Toto umožňuje Vue aplikácii "pripevniť sa" k akémukoľvek elementu v DOM hlavnej React aplikácie, čo je dôležité pre flexibilné integrovanie rôznych mikrofrontendov do jedného užívateľského rozhrania.

```
import { createApp } from "vue";
import App from "./App";

const mount = (el) => {
  const app = createApp(App);
  app.mount(el);
};

const devRoot = document.querySelector("#vue-microfrontend");
if (devRoot) {
  mount(devRoot);
}

export { mount };
```

Obrázok 28: Ukážka súboru vykonávajúceho dynamické nasadenie

Na strane Reactu, hlavná aplikácia využíva React hooky ako useRef a useEffect na ovládanie načítania a životného cyklu Vue mikrofrontendu. Komponent VueMicrofrontend vytvára kontajner (DOM element), do ktorého sa mikrofrontend načíta a inicializuje. Pomocou Reactref sa tento kontajner sprístupní a v useEffect hooku sa dynamicky naimportuje mount funkcia z Vue mikrofrontendu, ktorá pripevní Vue aplikáciu k tomuto kontajneru. Tento prístup umožňuje React aplikácii kontrolovať a spravovať životný cyklus mikrofrontendu, zatiaľ čo zachováva jeho izoláciu a nezávislosť.

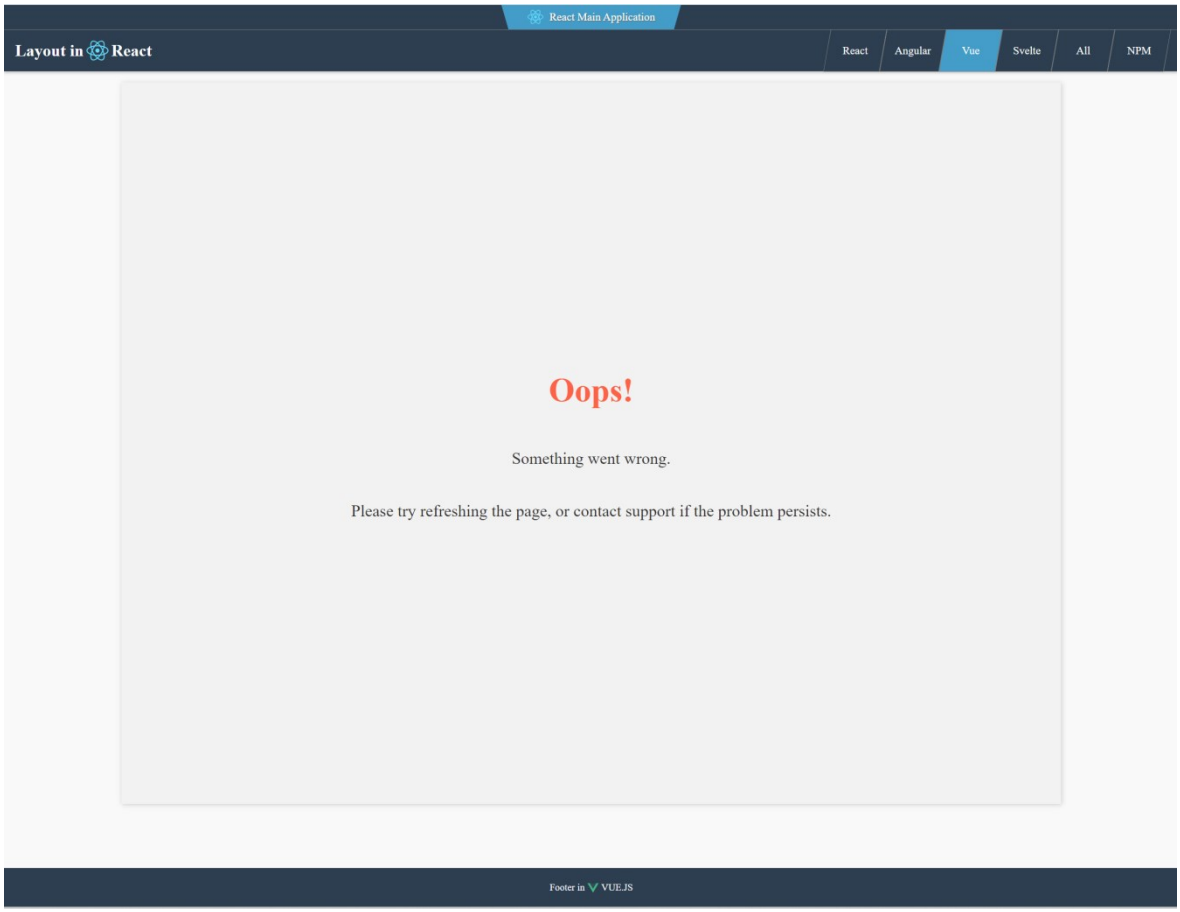
```
import React, { useRef, useEffect, useState } from "react";
import OopsComponent from "./OopsComponent";

const MicroFrontendLoader = () => {
  const ref = useRef(null);
  const [hasError, setHasError] = useState(false);

  useEffect(() => {
    if (!ref.current) return;

    const loadMicroFrontend = async () => {
      try {
        const { mount } = await import("vueMf/VueMainMf");
        mount(ref.current);
      } catch (error) {
        console.error("Microfrontend loading failed:", error);
        setHasError(true);
      }
    };

    loadMicroFrontend();
  }, []);

  if (hasError) {
    return <OopsComponent />;
  }

  return <div ref={ref} />;
};

export default MicroFrontendLoader;
```

Obrázok 29: Kontajner pre Vue hlavnú časť Vue mikrofrontendu

Takéto riešenie vyžaduje dôkladné plánovanie a koordináciu medzi rôznymi časťami aplikácie, aby sa zabezpečila hladká a efektívna funkcionalita celej aplikácie. Pri správnej implementácii však poskytuje vysokú flexibilitu a umožňuje vývojárom využívať najlepšie aspekty rôznych technológií v jednom koherentnom riešení.

Z daného obrázku je možné vidieť aj ako je riešené prípadne nenačítanie mikrofrontendu VueMainMf. V takomto prípade ak príde k chybe pri načítavaní mikrofrontendu, je dôležité mať pripravený spôsob, ako zachovať stabilitu a funkčnosť hlavnej aplikácie. V našom príklade je na tento účel využitý komponent OopsComponent. Tento komponent slúži ako záložný plán (fallback), ktorý sa zobrazí, ak proces načítania mikrofrontendu zlyhá z akéhokoľvek dôvodu ako napríklad sieťové chyby, problémy so závislosťami alebo chyby v kóde mikrofrontendu. Integrácia OopsComponent do procesu načítavania je realizovaná prostredníctvom stavovej premennej hasError v React komponente VueMicrofrontend. Ak proces pri načítavaní vyvolá výnimku, chybový stav je zachytený a premenná hasError je nastavená na true, čo spustí renderovanie OopsComponent namiesto pôvodného mikrofrontendu. Tým sa užívateľom zabezpečí aspoň základná funkčnosť alebo informácia

o probléme, čo výrazne zlepšuje užívateľský zážitok a znižuje frustráciu z možných technických problémov.



Obrázok 30: Reverzná obrazovka Oops

Tento model zabezpečuje, že chyby pri načítaní mikrofrontendov neovplyvnia celkovú stabilitu a dostupnosť hlavnej aplikácie a to dokonca aj v prípade, že časť jej funkcionality je dočasne nedostupná.

5.5 Svelte Mikrofrontend

Ako posledný zo štvorice JavaScriptových rámcov je Svelte.dev. Konfigurácia Module Federation Pluginu vo Webpacku je rovnaká, ako v predchádzajúcich mikrofrontendoch len s pozmenenými zdieľanými závislosťami.

```
new ModuleFederationPlugin({
  name: "SvelteMicrofrontend",
  filename: "remoteEntry.js",
  exposes: {
    "./SvelteMainMf": "./src/loadApp.js",
    "./SvelteMiniApp": "./src/loadMiniApp.js",
  },
  shared: {
    svelte: {
      singleton: true,
      eager: true,
    },
    "some-shared-lib": {
      singleton: true,
      strictVersion: true,
    },
  },
}),
```

Obrázok 31: Konfigurácia Module Federation pluginu vo Svelte

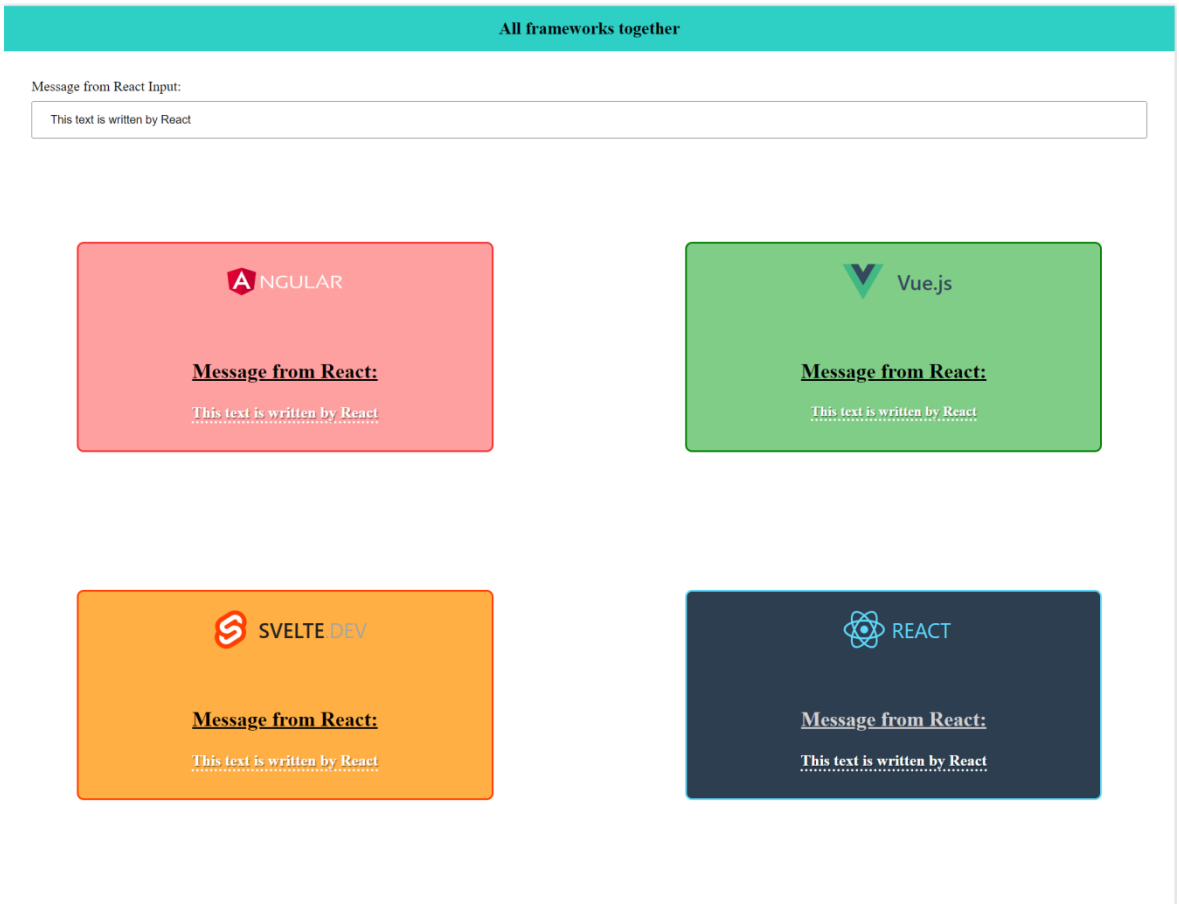
Taktiež vystavuje dva komponenty a to konkrétne SvelteMainMf a SvelteMiniApp, kde prvý menovaný slúži na zobrazovanie dát načítavaných zo servera a z pamäte prehliadača do dvoch tabuliek (knihy a filmy) a možno ju nájsť na adrese /svelte a druhý menovaný je mini aplikácia o ktorej si povieme v AllExamples.

Súbor loadApp,js a loadMiniApp.js sú rovnako ako v prípade súboru vo Vue (bootstrap) súbory, ktoré vykonávajú funkciu dynamického nasadenia o ktorom sme si už písali predchádzajúcej kapitole.

5.6 Podstránka AllExamples

Naša aplikácia na podstránke "AllExamples" predstavuje centrálny bod pre načítanie a interakciu s mikrofrontendami, postavených na všetkých štyroch JavaScriptových rámcoch, ktoré v aplikácií využívame (Angular, Vue, Svelte a React). Táto stránka demonštruje, ako

je možné v rámci jednej hostiteľskej aplikácie integrovať a spravovať mikrofrontendy v rámci jednej inštancie.



Obrázok 32: Obrazovka AllExamples

Hlavným účelom tejto podstránky je umožniť prenos dát medzi mikrofrontendami, bez potreby ich ukladania na server, alebo do lokálnej pamäte prehliadača, ako to bolo nutné v predchádzajúcich prípadoch pri navigácii medzi podstránkami reprezentujúcimi jednotlivé mikrofrontendy. Toto je dosiahnuté prostredníctvom centralizovaného vstupného komponentu, MiniAppInput, ktorý prijíma vstupy od užívateľa a rozosiela ich do všetkých načítaných mikrofrontendov pomocou vlastného eventu.

```
const handleInputChange = (event) => {
  const newValue = event.target.value;
  setInputValue(newValue);

  const eventDetail = new CustomEvent("fromReact", {
    detail: { message: newValue },
  });
  window.dispatchEvent(eventDetail);
};
```

Obrázok 33: Ukážka vkladania údajov z formulára do udalosti window

Vstupy od užívateľa prijaté prostredníctvom komponentu MiniAppInput sú rozosielané do jednotlivých mikrofrontendov pomocou vytvorenia vlastného objektu udalostí, nazývaného CustomEvent, ktorý je následne vyvolaný na globálnom objekte window. Tento spôsob umožňuje, aby všetky mikrofrontendy načúvali na tejto spoločnej udalosti a reagovali na ňu v reálnom čase, čím zabezpečujú konzistentnú a efektívnu komunikáciu medzi rozdielnymi technológiami použitými v aplikácii.

Kód na stránke "AllExamples" dynamicky načítava Reactové mikrofrontendy využitím technológie Suspense a lazy loading, ktoré sú súčasťou Reactu, čo zabezpečuje efektívnosť a optimalizáciu načítavania týchto komponentov. Každý mikrofrontend je reprezentovaný vlastným modulom, ktorý je načítaný asynchrónne prostredníctvom svojich špecifických funkcií na načítavanie, známych ako mount funkcie, definovaných v ich príslušných konfiguračných súboroch. Tento prístup minimalizuje počiatočné zaťaženie aplikácie a zároveň umožňuje integrovať mikrofrontendy zo všetkých našich JavaScriptových rámcov do hostiteľskej React aplikácie.

Mikrofrontendy na tejto stránke sú schopné prijímať a zobrazovať dáta poslané z MiniAppInput prostredníctvom globálnych eventov. Tento prístup umožňuje dátovú komunikáciu medzi rôznymi časťami aplikácie bez toho, aby sa spoliehala na spoločné globálne stavové riadenie alebo zdieľanie stavu na úrovni servera. Tieto komponenty prijímajú dáta posielané z React komponentu, spracovávajú ich a zobrazujú v svojom užívateľskom rozhraní.

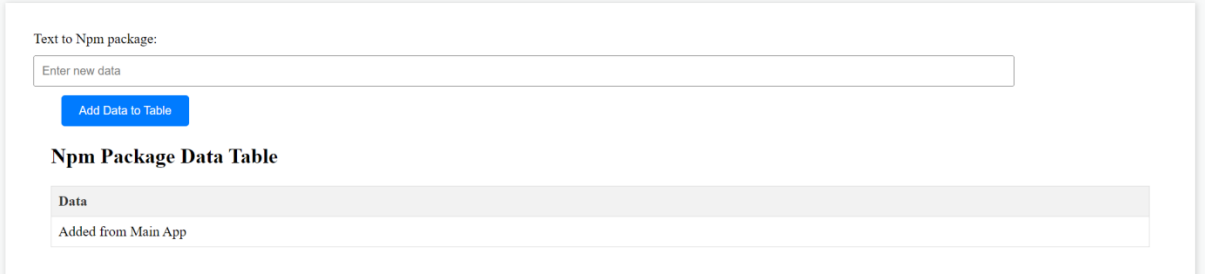
```
methods: {
  handleReactMessage(event) {
    this.messageFromReact = event.detail.message;
  },
},

mounted() {
  window.addEventListener("fromReact", this.handleReactMessage);
},

beforeDestroy() {
  window.removeEventListener("fromReact", this.handleReactMessage);
},
```

Obrázok 34: Načítavanie dát z udalosti window

Celkovo, podstránka "AllExamples" predstavuje výkonné riešenie pre demonštráciu a testovanie kompatibility medzi rôznymi mikrofrontendami a zároveň poskytuje platformu pre zdieľanie dát v reálnom čase medzi rôznymi časťami aplikácie v rámci jednej inštancie.

5.7 Alternatíva NPM balíčka

Na podstránke NPM v našej hlavnej aplikácii demonštrujeme spôsob, ako môže aplikácia interagovať s komponentami, ktoré sú distribuované ako samostatné NPM balíčky. V tomto prípade máme TableComponent, ktorý je importovaný z lokálneho NPM balíčka s názvom npmpackage. Tento komponent slúži na zobrazovanie dát v tabuľke, pričom dáta sú dynamicky pridávané užívateľom prostredníctvom textového vstupu na webovej stránke.



Obrázok 35: Obrazovka NPM zobrazujúca obsah NPM balíčka

Užívateľ má možnosť vložiť text do vstupného poľa a pomocou tlačidla "AddData to Table" (v preklade: pridať tento text do tabuľky). Tento proces funguje tak, že užívateľský vstup je najprv uložený v lokálnom stave React komponentu a po akcii pridania je tento vstup prenesený do stavu, ktorý obsahuje dáta pre TableComponent. Každé nové dáta pridané užívateľom sa tak zobrazujú v tabuľke.

Podstránka je teda praktickou ukážkou využitia NPM balíčka pre správu a prezentáciu užívateľských dát v rámci React aplikácie. Tento prístup umožňuje oddeliť časti funkcionality do samostatných balíčkov, ktoré môžu byť nezávisle vyvíjané a spravované. Používame vlastný Verdaccio server, ktorý je lokálnym a privátnym NPM registrom balíčkov a beží nám na porte 4873. Tento server zjednodušuje správu a distribúciu našich balíčkov v rámci vývojového prostredia, umožňujúc nám efektívne riadiť závislosti a verzie softvéru.

5.7.1 Porovnanie mikrofrontendov s NPM balíčkami v rámci aplikácie

Mikrofrontendy a NPM balíčky predstavujú dve rôzne metódy modularizácie a izolácie funkcionalít v moderných webových aplikáciách.

Mikrofrontendy umožňujú vytvárať veľké aplikácie ako súbor malých, nezávislých častí, ktoré môžu bežať na rôznych technologických zásobníkov (stackoch) a komunikovať medzi sebou, napríklad prostredníctvom zdieľaných stavov alebo eventov. Tento prístup je ideálny pre veľké tímy a aplikácie s rôznorodými technologickými požiadavkami, pretože každý mikrofrontend môže byť vyvíjaný a nasadzovaný nezávisle. Jednou z hlavných výhod mikrofrontendov oproti NPM balíčkom je škálovateľnosť zdrojov, keďže jednotlivé mikrofrontendy môžu byť načítané dynamicky podľa potreby, čo umožňuje efektívnejšie využitie zdrojov a optimalizáciu výkonu aplikácie

Na druhej strane, NPM balíčky poskytujú spôsob, ako izolovať a znovu používať kód na úrovni funkcionalít alebo komponentov v rámci JavaScriptových aplikácií. NPM balíčky sú zvyčajne technologicky homogénnejšie, čo znamená, že sú určené pre špecifické prostredie alebo rámec, ako je React alebo Angular. Tento prístup je vhodný pre zdieľanie a znovu použitie kódu medzi projektami alebo medzi rôznymi časťami rovnakej aplikácie. V prípade NPM balíčkov sa však čelí limitácii v škálovateľnosti, keďže celý balíček sa musí stiahnuť a integrovať do aplikácie počas procesu budovania (buildu), čo môže viesť k väčšiemu zaťaženiu zdrojov a zvýšeniu času načítania aplikácie.

Oba prístupy prispievajú k efektívnosti vývoja, avšak ich výber závisí od špecifických potrieb projektu, štruktúry tímu a infraštruktúry. Mikrofrontendy sú lepšie pre komplexné systémy s heterogénnymi technológiami, zatiaľ čo NPM balíčky sú ideálne pre znovu použitie kódu a funkcionality v homogénnom technologickom ekosystéme.

ZÁVER

Z výsledkov práce na projekte vyplýva, že implementácia mikrofrontendovej architektúry prináša významné výhody pre moderné webové aplikácie. Tento prístup značne zvyšuje modularitu a umožňuje lepšiu škálovateľnosť a flexibilitu vývojových tímov pri práci na rozličných častiach aplikácie. Nezávislosť jednotlivých mikrofrontendov zjednodušuje testovanie a nasadzovanie nových funkcií, čo vedie k rýchlejšiemu vývoju a efektívnejšiemu manažmentu projektov. Táto architektúra je najmä výhodná pre veľké projekty a platformy, kde rôzne tímy môžu pracovať na odlišných funkciách alebo produktoch a potrebujú flexibilné riešenie pre nezávislý vývoj.

Avšak, výber tejto technológie by mal byť urobený s ohľadom na špecifiká projektu. V prípade menších projektov, kde sa kladie dôraz na úzke prepojenia komponentov a jednoduchosť správy týchto komponentov, môže byť výhodnejšie zostať pri tradičnejšej architektúre ako predstavujú klasické monolity, alebo jednoduchšie rozdelené moduly. Integrácia mikrofrontendov môže totiž viesť k zvýšenej technickej a organizačnej komplexnosti, čo pri menších aplikáciách môže byť zbytočné.

Pri implementácii mikrofrontendov je nevyhnutné venovať pozornosť správnemu návrhu a riadeniu komunikačných protokolov medzi mikrofrontendami, aby sa zabránilo problémom s konzistentnosťou a výkonom aplikácie. Pri správnej implementácii mikrofrontendov môžu organizácie dosiahnuť významné zlepšenie v agilnosti a adaptabilite ich softvérových riešení, čo je v dnešnom rýchlo sa meniacom svete technológií neoceniteľnou výhodou.

Záverom teda možno konštatovať, že mikrofrontendy predstavujú silný a perspektívny nástroj pre organizácie, ktoré hľadajú spôsoby ako zlepšiť a zefektívniť svoje softvérové vývojové procesy a sú pripravené prijať zvýšenú úroveň technickej komplexnosti. S dobrým plánovaním a správnym nasadením môžu mikrofrontendy poskytnúť strategickú výhodu tým, že umožňujú rýchlejšie inovovať a lepšie reagovať na požiadavky trhu.

ZOZNAM POUŽITEJ LITERATÚRY

[1] HARRIS, CHANDLER. Microservices vs. Monolithic architecture. Atlassian [online]. [cit. 2024-05-05]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[2] MISHRA, Alok. What's the difference between monolithic and microservices architecture? [online]. 2020, 26.2.2020 [cit. 2024-05-05]. Dostupné z: https://alok-mishra.com/2020/02/26/how-did-we-get-to-microservices/

[3] AMAZON. What's the difference between monolithic and microservices architecture? Amazon [online]. [cit. 2024-05-05]. Dostupné z: https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/

[4] ELHOUSIENY, Rany. Micro-Frontends: What, why, and how [online]. 2021, 11.3.2021 [cit. 2024-05-05]. Dostupné z: https://levelup.gitconnected.com/micro-frontends-what-why-and-how-bf61f1f0a729

[5] APLYCA. MicroFrontends: What are They and When to UseThem? Aplyca [online]. 2023, 12.1.2023 [cit. 2024-05-05]. Dostupné z: https://www.aplyca.com/en/blog/micro-frontends-what-are-they-and-when-to-use-them

[6] JACKSON, Cam. MicroFrontends. Martinfowler [online]. 2019, 19.6.2019 [cit. 2024-05-05]. Dostupné z: https://martinfowler.com/articles/micro-frontends.html

[7] BROOKS, Gary. Micro frontends: pros and cons. Fabrity [online]. 2023, 22.8.2023 [cit. 2024-05-05]. Dostupné z: https://fabrity.com/blog/micro-frontends-pros-and-cons/

[8] SHARMA, Abhinav. Connecting The Dots: Data Communication Methods for Microfrontends. [online]. 2023, 23.9.2023 [cit. 2024-05-05]. Dostupné z: https://www.fabricgroup.com.au/blog/connecting-the-dots-data-communication-methods-for-micro-frontends

[9] FERGUSON, Natasha. Single Page Applications (SPA). Medium [online]. 2023, 1.1.2023 [cit. 2024-05-05]. Dostupné z: https://medium.com/@teamtechsis/single-page-applications-spa-48b1b845b446

[10] JACKSON-BARNES, Shannon. SPA Vs. MPA Explained: Differences Between Single-Page Applications and Multi-Page Applications [online]. 2023, 17.11.2023 [cit. 2024-05-05]. Dostupné z: https://www.orientsoftware.com/blog/spa-vs-mpa/

[11] SCHEPENAAR, Wilbert. Server-side vs Client-side Routing. Medium [online]. 2017, 29.5.2017 [cit. 2024-05-05]. Dostupné z: https://medium.com/@wilbo/server-side-vs-client-side-routing-71d710e9227f

[12] VINCI, Abhinav. Microfrontends? What and Why. Medium [online]. 2023, 20.05.2023 [cit. 2024-05-05]. Dostupné z: https://medium.com/@vinciabhinav7/micro-frontends-what-and-why-d7bbe67d3c6f

[13] TOAL, Rory. What Is JavaScript Framework? Code institute [online]. [cit. 2024-05-05]. Dostupné z: https://codeinstitute.net/global/blog/javascript-framework/

[14] KHAN, Shahzad and General Assembly. What is a javascript framework? [online]. [cit. 2024-05-05]. Dostupné z: https://generalassemb.ly/blog/what-is-a-javascript-framework/

[15] GEEKS FOR GEEKS. Introduction to TypeScript. Geeks for geeks [online]. 2024, 11.3.2024 [cit. 2024-05-05]. Dostupné z: https://www.geeksforgeeks.org/introduction-to-typescript/?ref=lbp

[16] KINSTA. What Is React.js? A Look at the Popular JavaScript Library. Kinsta [online]. 2023, 11.7.2023 [cit. 2024-05-05]. Dostupné z: https://kinsta.com/knowledgebase/what-is-react-js/

[17] GOOGLE, INC. What is Angular? Angular [online]. 2023, 15.8.2023 [cit. 2024-05-05]. Dostupné z: https://angular.io/guide/what-is-angular

[18] What is Vue? Vuejs [online]. [cit. 2024-05-05]. Dostupné z: https://vuejs.org/guide/introduction

[19] What is Svelte? Sanity [online]. 2024, 16.1.2024 [cit. 2024-05-05]. Dostupné z: https://www.sanity.io/glossary/svelte

[20] OPENJS Foundation, Concepts. Webpack [online]. [cit. 2024-05-05]. Dostupné z: https://webpack.js.org/concepts

[21] Getting Started. Vitejs [online]. [cit. 2024-05-05]. Dostupné z: https://vitejs.dev/guide/

[22] GITHUB. About packages and modules. Npmjs [online]. 2023, 23.10.2023 [cit. 2024-05-05]. Dostupné z: https://docs.npmjs.com/about-packages-and-modules

[23] Verdaccio. Verdaccio [online]. [cit. 2024-05-05]. Dostupné z: https://verdaccio.org

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

API	An application Programming Interface
CSR	Client Side Rendering
CSS	Cascading Style Sheets
DOM	Document Object Model
ES6	ECMA Script version 6
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JMS	Java Message Service
js	JavaScript
MPA	Multiple Page Applications
MVC	Model View Constroller
NPM	Node Package Manager
PWA	Progressive Web App
ref	reference
SEO	Search Engine Optimization
SFC	Single File Components
SOA	Service Oriented Architecture
SPA	Single Page Applications
SSR	Server Side Rendering
Ts	TypeScript
tsc	typescript compilator
UI	User Inerface
URL	Uniform Resource Locator
XML	eXtensible Markup Language
XSS	Cross Site Scripting

ZOZNAM OBRÁZKOV

ZOZNAM PRÍLOH

Príloha I: CD

PRÍLOHA P I: NÁZOV PRÍLOHY

- DP_Netes_Filip.pdf