Pedagogická fakulta
Faculty of Education

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Vývoj webových Single Page Applications s využitím perspektivní techniky micro-frontendu

Development of Web Single Page Applications using a promising micro-frontend technique

Bakalářská práce

Vypracoval: Petr Vrba

Vedoucí práce: PaedDr. Petr Pexa, Ph.D.

České Budějovice 2023

Zadání bakalářské práce

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Pedagogická fakulta

Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Petr VRBA**
Osobní číslo:	**P20503**
Studijní program:	**B7507 Specializace v pedagogice**
Studijní obor:	**Informační technologie a e-learning**
Téma práce:	**Vývoj webových Single Page Applications s využitím perspektivní techniky micro-frontendu.**
Zadávající katedra:	**Katedra informatiky**

Zásady pro vypracování

Cílem bakalářské práce je zpracování problematiky tvorby webových Single Page Applications (dále SPA) s využitím perspektivních technologií micro-frontendu. Využití micro-frontendových technologií v rámci SPA umožňují práci s daty a obsahem webových stran, aniž by bylo třeba obnovovat stránku. Stejně tak jsou i perspektivní metodou, jak složitější aplikace díky dělbě práce na straně klienta i hostingu realizovat, aniž by docházelo k přetížení serveru. Teoretická část bude zaměřená na význam a specifikace SPA a micro-frontendu, dále pak budou představeny základní jazyky a technologie, se kterými se v rámci SPA pracuje, včetně konkrétních frameworků pro JavaScript. Součástí teoretické části bude také představení funkcionality a významu balíčkovacích systémů pro SPA. V praktické části bude hlavním výstupem webová stránka demonstrující SPA za použití micro-frontendových technologií na konkrétních příkladech a funkcionalitách. Webová stránka bude následně sloužit jako první veřejně dostupná dokumentace v českém jazyce, věnující se problematice Single Page Applications s využitím micro-frontendu, kterou budou moci využít IT odborníci i studenti.

Rozsah pracovní zprávy:	**40**
Rozsah grafických prací:	**interaktivní modely**
Forma zpracování bakalářské práce:	**tištěná**

Seznam doporučené literatury:

1. GANDHI, R. JavaScript next: your complete guide to the new features Introduced in JavaScript, starting from ES6 to ES9. United States: Apress, 2019. ISBN 978-1-4842-5393-9.
2. LANCIAUX, R. Modern Front-end Architecture, England: APress, 2021, ISBN 978-1-4842-6624-3.
3. DUCKETT, J. JavaScript and JQuery: Interactive front-end web development. Indianapolis, IN: Wiley, 2014. ISBN 978-1-118-53164-8.
4. Getting started. React.js official documentation [online]. 2013 [cit. 9.březen 2022]. Dostupné z: https://reactjs.org/docs/getting-started.html
5. Developer Guide. Angular.js official documentation [online]. 2010 [cit. 9.březen 2022]. Dostupné z: https://docs.angularjs.org/guide/

Vedoucí bakalářské práce: **PaedDr. Petr Pexa, Ph.D.**
Katedra informatiky

Datum zadání bakalářské práce: **17. března 2022**
Termín odevzdání bakalářské práce: **30. dubna 2023**

L.S.

doc. RNDr. Helena Koldová, Ph.D.
děkanka

doc. PaedDr. Jiří Vaníček, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 17. března 2022

Prohlášení

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval(a) pouze s použitím pramenů a literatury uvedených v seznamu použitých zdrojů.
V Českých Budějovicích dne 11. dubna 2023.

Petr Vrba

Anotace

Cílem bakalářské práce je zpracování problematiky tvorby webových Single Page Applications (dále SPA) s využitím perspektivních technologií micro-frontendu. Využití micro-frontendových technologií v rámci SPA umožňují práci s daty a obsahem webových stran, aniž by bylo třeba obnovovat stránku. Stejně tak jsou i perspektivní metodou, jak složitější aplikace díky dělbě práce na straně klienta i hostingu realizovat, aniž by docházelo k přetížení serveru. Teoretická část bude zaměřená na význam a specifikace SPA a micro-frontendu, dále pak budou představeny základní jazyky a technologie, se kterými se v rámci SPA pracuje, včetně konkrétních frameworků pro JavaScript. Součástí teoretické části bude také představení funkcionality a významu balíčkovacích systémů pro SPA. V praktické části bude hlavním výstupem webová stránka demonstrující SPA za použití micro-frontendových technologií na konkrétních příkladech a funkcionalitách. Webová stránka bude následně sloužit jako první veřejně dostupná dokumentace v českém jazyce, věnující se problematice Single Page Applications s využitím micro-frontendu, kterou budou moci využít IT odborníci i studenti.

Klíčová slova

Single Page Aplikace, Micro-Fronted, Frontend, JavaScript, JavaScript, React.js, Andular, Vue.js, framework, balíčkovací systém

Abstract

The point of the bachelors thesis is to process problematics of the Single Page Aplications (SPA further) websites development using promising micro-frontend technologies. Using of micro-frontend technologies for SPA allows processing of data and website content without need of webpage refresh. They are concurrently promising method, how to realize more complex aplications without server overload, thanks to division of labor between server and clients browser. The content of the theoretical part will be focused on meaning and specifications of SPA and micro-frontend, then presentation of specific JavaScript frameworks. In the theoretical part will be presented functionality and the importance of packaging systems for SPA also. The main result of the work will be SPA website demonstrating usement of micro-frontend technologies on specific examples and functionalities. The website will be used as first public documentation in czech language, dedicate with Single Page Applications using micro-frontend, which will be able to be used by IT professionals and students.

Keywords

Single Page Aplications, Micro-Frontend, Frontend, JavaScript, React.js, Andular.js, Vue.js, framework, package manager

Poděkování

Touto cestou bych rád poděkoval panu PaedDr. Petru Pexovi, Ph.D. za cenné rady, velice vstřícný přístup a trpělivost při vedení bakalářské praxe. Rád bych poděkoval i svým nejbližším, především rodičům, prarodičům a přítelkyni, kteří jsou mi oporou a pomáhají mi po celé studium.

Obsah

1 Úvod

Bakalářská práce se bude zabývat tvorbou Single Page Aplikací s použitím moderní technologie micro-frontendu, za využití nejpřednějších JavaScriptových frameworků React.js, Andular a Vue.js.

1.1 Východiska

Micro-frontend je moderní webovou technologií, která umožňuje pohodlně realizovat moderní aplikace. Zatím co v minulosti bylo přirozené, že se při interakci na webových stránkách musel načítat celý obsah znovu, tak potřeba moderních modulů jako jsou například chatová okna, vyžadovala mnohem efektivnější řešení.

Tím řešením je právě micro-frontend, který nahlíží na stavbu webové stránky zcela jinak. Dokáže ji efektivně rozdělit na jednotlivé moduly, v nichž neustále hlídá, zdali se změnil jejich obsah. V případě změny dokáže micro-frontendové pojetí jednoduše aktualizovat obsah daného modulu, aniž by musel stránku opět načítat. Micro-frontend je tak technologicky dokonalejší, a zároveň i uživatelsky mnohem přívětivější technologií. Stal se nepostradatelnou součástí celé řady oblíbených portálů, přičemž mnohé z nich pracují na vývoji vlastních frameworků pro realizaci micro-frontendových stránek.

1.2 Cíle

Cílem bakalářské práce bude v teoretické i praktické rovině zpracovat problematiku tvorby webových stránek s použitím micro-frontendových technologií. V rámci praktické části budu demonstrovat použití micro-frontend technologií na nejběžnějších modulech, které vyžadují aktivní DOM (Data Object Model), jako jsou chatová okna, kalkulačky a další podobné prvky. Všechny prvky v rámci demonstrace budu realizovat také pomocí standardního PHP bez použití frameworků, abych tím znázornil význam a výhody micro-frontend. Každý

modul navíc bude kompletně optimalizován pro co nejlepší výkon, přičemž použiji i balíčkovací systémy, které jsou především pro rozsáhlé portály základním nástrojem k optimálnímu načítání.

Výsledkem bakalářské práce bude webová prezentace, na které bude demonstrována funkcionalita micro-frontend na zmíněných modulech. Každému z modulů se budu věnovat ve všech zahrnutých frameworcích, abych dokázal ve výstupu porovnat náročnost provozu, vývoje
a budoucího zdokonalování webových prvků.

Od výsledné webové stránky očekávám, že dokáže fungovat jako efektivní a přehledná demonstrace micro-frontend, která zároveň dokáže i fakt, že jsou jednotlivé technologie
micro-frontend účelnější na různé druhy aplikací.

Doplňkem webové aplikace bude kompletní překlad dokumentací k frameworku React.js (od společnosti Meta) a frameworku Andular (od společnosti Google) do českého jazyka, včetně dokumentačních poznámek v kódu.

1.3 Metody

V teoretické části bakalářské práce se budu zabývat webovými technologiemi, které se v rámci micro-frontendu používají. K co nejlepší dokumentaci a demonstraci micro-frontendu jsem vybral trojici nejspecifičtějších frameworků, které v teoretické části detailně rozeberu, v praktické poté využiju. Mimo samotných JavaScript frameworků proberu i další technologie, mezi které spadají i balíčkovací systémy, které se hojně využívají k lepší organizaci a načítání jednotlivých funkčních modulů.

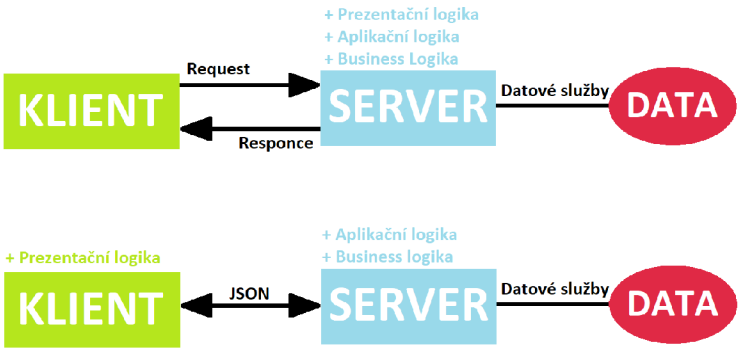
Hlavním výstupem praktické části bakalářské práce bude webová stránka, která bude fungovat jako dokumentace a zároveň bude demonstrovat výhody a využití micro-frontend. Vše bude demonstrováno na často používaných webových prvcích. Součástí výstupu je také porovnání náročnosti a výkonu kódu jednotlivých modulů za pomoci různých micro-frontendových technologií.

2 Single Page Aplikace

2.1 Specifikace

Základní charakteristikou Single Page Aplikací je fakt, že se celá aplikace načítá jako jedna jediná stránka. Díky tomu se tak server sám o sobě stará jenom o to, aby dokázal vytvořit aplikaci, ale následně ji spravuje především prohlížeč na uživatelském zařízení. Pro správné pochopení Single Page Aplikací je vhodné porovnat základní charakteristiky mezi tradičním webem a jednostránkovou verzí. [1]

U tradičního webu se vše odehrává pouze na straně serveru. Veškeré modely, business logika i prezentační logika vznikají zde. Prohlížeč tak odešle při návštěvě stránky HTTP request směrem k serveru. Ten jej přijme, kompletně zpracuje a pošle zpátky kompletní HTTP response v podobě statického HTML dokumentu. V moment, kdy uživatel bude chtít na stránce vykonat jakoukoliv úlohu, tak opět musí odesílat HTTP request a celý cyklus se opakuje. [1]



Obrázek 1: Komunikace u statické a Single Page aplikace

V případě Single Page Aplikací se ale práce dělí mezi klienta a server. Na straně serveru zůstávají veškeré datové, servis i business vrstvy včetně struktur řídících interakci prezentační vrstvy se zbytkem aplikace. Kompletní

prezentační vrstva se ale přesouvá do prohlížeče na straně klienta. Vykreslování webové stránky tak oproti tradičním webům probíhá u uživatele a komunikace mezi serverem a uživatelem se omezuje pouze na nutné datové operace. To rovněž umožňuje načtení celé stránky jako celku, což nakonec vyústí v absenci načítání každé stránky samostatně. [1]

2.2 Výhody Single Page Aplikací

2.2.1 Navigace bez obnovení stránky

Single Page aplikace nepoužívají stoprocentní HTML stránky tak, jako serverově orientované aplikace. Při načítání stránky dochází k načtení celého kódu, který je potřeba pro zobrazení uživatelského rozhraní. Jednotlivé části webu jsou ale generovány s pomocí objektově orientovaného přístupu k HTML dokumentu v podobě DOM. Veškeré nutné soubory pro generování view a jeho součástí jsou již načteny, takže v moment, kdy uživatel chce zobrazit nové view se stránka znovu nenačítá. JavaScript dynamicky upraví data v DOM.[1]

2.2.2 Prezentační logika u klienta

Zmíněný proces znamená, že se celá prezentační logika přesouvá do klientského prohlížeče. Potřebná HTML data tak jsou načtena v prohlížeči a samotný server obsahuje business logiku a reference na to, do jaké části HTML se data mají vložit. Nikdy ale nedochází k načítání kompletní stránky, ale pouze části view, která uživatele v daný moment zajímá.[1]

Problémem v takovém přístupu ale může mít nedostatečná kontrola životního cyklu aplikace a také směrování mezi jednotlivými view. O zajištění těchto funkcí se ale starají JavaScript frontend frameworky jako React.js, Vue.js, Angular a mnohé další, navržené na principu MV* architektur.[1]

2.2.3 Menší vytížení serveru

Při vykreslování webových stránek se pohybujeme v oblasti architektury klient-server. Ta sice má výhody jako je vysoká centralizace a ochrana dat, ale ve většině provedení znamená enormní vytížení serveru. Vykreslování statických webových stránek v tomto případě využívá stoprocentní kapacitu serveru a uživatelovo zařízení pouze vznáší a přijímá požadavky.[1]

Aplikace architektury klient-server, kterou aktivně Single Page Aplikace používají, se přesněji označuje jako klient-server se vzdálenou prezentací. Již zmíněné přenesení prezentační vrstvy do klientova zařízení totiž automaticky znamená snížení vytížení produkčního serveru. Přitom důležité prvky klient-server architektury jako je integrita dat, tím zůstávají zcela nenarušeny.[1]

2.2.4 Kratší odezva stránky

Pro kvalitní uživatelskou zkušenost s jakoukoliv aplikací je jedním ze zásadních parametrů rychlost odezvy. Mnohokrát zmíněná potřeba neustále načítat celou stránku znovu v případě statických dokumentů se tak dostává do pozice, kdy pro uživatele není ideální. Single Page aplikace díky načítání stránky jako souhrnu view s vloženými daty, dokáže zásadně snížit odezvu.[1]

To má nejen tu výhodu, že je stránka pro uživatele mnohem příjemnější na prohlížení, ale také může mít zásadní dopad na primární funkce stránky. V roce 2012 se této problematice v komerční sféře věnovala americká společnost Walmart v rámci studie. Výsledkem bylo, že zlepšení výkonu načítání webové stránky o přibližně 100ms se odrazilo také ve zvýšení prodejů z internetového portálu o 1 %.[1]

2.3 Nevýhody Single Page Aplikací

2.3.1 SEO optimalizace

Search engine optimization (SEO) je cestou, jak zvýšit návštěvnost stránky díky vyšším pozicím ve vyhledávačích. Obsahuje celou řadu onpage a offpage

faktorů, které dohromady určují čitelnost a relevanci stránky pro boty užívané vyhledávači. Na základě toho je tak pro SEO ideální stromová struktura jednoduchých a dobře čitelných dokumentů. To ale v případě Single Page Aplikací není ani zdaleka tak jednoduché. Většina botů, které užívají vyhledávače, není schopna proniknout hlouběji do architektury Single Page aplikace a dobře ji číst. I přesto je SEO optimalizace možná, ale jedná se o podstatně složitější úkol.[2]

2.3.2 Problémy na straně klienta

Na straně klienta může docházet k celé řadě problémů. JavaScript, bez kterého nemohou Single Page aplikace fungovat, vyžaduje mnohem vyšší výpočetní kapacitu než samotné načítání statických HTML dokumentů. Proto se může stát, že na slabších konfiguracích mohou masivní Single Page aplikace mít problém s rychlostí, nebo dokonce mohou končit pádem po vyčerpání přidělené výpočetní paměti.[1]

Dalším problémem může být podpora aplikace na starších, nebo méně používaných prohlížečích. Moderní prohlížeče jsou kvalitně vybaveny podporou nejen Single Page aplikací, ale i všech frameworků a nástrojů nutných pro jejich funkci. Na starších prohlížecích ale mohou aplikace projevovat nesprávné chování, nebo dokonce nemusí vůbec fungovat.[1]

2.3.3 Riziko XSS

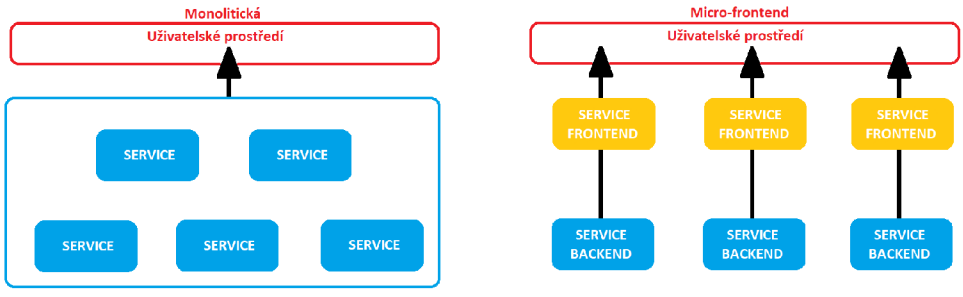
Zkratka XSS reprezentuje pojem „Cross Site Scripting Attack“. Jedná se o druh počítačového útoku, který funguje na bázi vkládání škodlivého kódu do těla dynamických webových stránek.[1] Díky vložení kódu stránka není schopna rozpoznat, že nepochází od ověřeného zdroje a může tak dojít k narušení datové integrity v rámci odposlouchávání komunikace. Jedná se ale o náročný útok, kterému lze do značné míry předejít kvalitně zabezpečenou Single Page aplikací.[1]

3 Micro-Frontend

3.1 Specifikace

Technologie tvorby webových aplikací pomocí micro-frontend patří mezi nejmodernější současné trendy. Jedná se o webové stránky, které jsou postaveny na mikroservis architektuře, která je do značné míry pravým opakem architektury monolitické.

Monolitická architektura se vyznačuje jednotou kódu. Jedná se tak o aplikace, které jsou standardně vyvíjeny v úzkém okruhu technologií a jazyků. Celý životní cyklus od začátku až do konce u monolitické architektury počítá se vším, co je její součástí a na rozdíl od jiných architektur zde nenajdeme žádné dělení do vrstev nebo servis.[1] Výhodou je rozhodně jednodušší vývoj a možnost komplexně testovat celou aplikaci najednou. Problémem ale je, že jedna změna se může promítnout na celé řadě míst v rámci monolitické aplikace. I kvůli tomu je monolitická architektura správnou volbou jen na drobné projekty, protože se s rostoucí aplikací stává vývoj a udržitelnost kódu velice nereálnou.[2]



Obrázek 2: Porovnání Monolitické a Mikroservis architektury

Micro-Frontend přináší díky mikroservis architektuře jiný pohled na vývoj

celé aplikace. Mikroservis architektura funguje tak, že dokáže pro každou komponentu oddělit vše od dat, přes aplikační vrstvu až do prezentační vrstvy. Přitom ale na potřebném výstupu mohou jednotlivé komponenty plnit úlohu jako celek.[2] Micro-Frontend se v tomto dělení ocitá jen v samotné prezentační vrstvě. Motivací pro vznik mikroservis architektury je především neustále se zvětšující rozsah webových aplikací. Již zmíněný problém s udržováním a rozšiřováním monolitické aplikace přitom je jeden z mnoha. K němu se připojuje fakt, že pokud by bylo potřeba monolitickou aplikaci přesunout na modernější jazyk či framework, tak je to nemožné. Zcela naopak v rámci mikroservis architektury není problém přidat novou komponentu ve zcela odlišném jazyce.[3]

3.2 Výhody Micro-Frontend

3.2.1 Jednodušší údržba rozsáhlých aplikací

U stále rostoucích aplikací se může ukázat problém s dostatečnou údržbou a dalším rozšiřováním funkcionalit. Pokud je ale celá aplikace rozdělená do jednotlivých nezávislých částí, tak je mnohem jednodušší aplikaci dále vyvíjet i rozšiřovat. Přehlednost zcela oddělených modulů je tak velkou výhodou.[2]

3.2.2 Nezávislost jednotlivých technologií

U monolitických aplikací je poměrně omezené množství jazyků a technologií, které se používají. Spojení různých frontend frameworků je tak nereálné. V případě micro-frontend je ale možnost použít na jedné aplikaci téměř neomezené množství technologií zásadní výhodou. Jednotlivé technologie a frameworky mohou jen částečně splňovat specifika dané aplikace. Pokud tak existuje více možných technologií, tak je lze v rámci oddělených mikroservis vložit do jediného projektu.[4]

3.2.3 Škálovatelnost aplikace

Jedná se o jednu ze specifických funkcí, které mikroservis architektura umožňuje. Každá z mikroservis v případě standardního provozu stránky funguje v jediné instanci. Může se ale stát, že některá část aplikace bude přetížena vysokou návštěvností. V ten moment lze aplikaci škálovat za pomoci virtualizace. V praxi to znamená, že přetížená služba je spuštěna v další instanci a o příchozí požadavky se instance starají odděleně. Jakmile se sníží vytížení služby, tak controller ukončí všechny duplicitní instance.[5]

3.2.4 Jednodušší a bezpečnější práce na projektu

Práce na monolitické aplikaci vyžaduje vysokou míru porozumění celému systému a vysokou koordinaci mezi vývojáři. Mikroservis architektura ale umožňuje, aby na každém komponentu mohl pracovat nezávislý tým bez nutnosti znát zbytek aplikace a komponentů. To zásadně ulehčuje vývoj i údržbu aplikace. Z pohledu bezpečnosti je výhodou mikroservis fakt, že chyba jedné komponenty nemusí mít žádný dopad na funkci zbytku aplikace a lze tak zamezit systémovým chybám.[5]

3.3 Nevýhody Micro-Frontend

3.3.1 Obtížné testování aplikace

Monolitické aplikace mají výhodu v tom, že jsou jednoduše testovatelné. Standardní jednotkové testy jsou totiž koncipovány na konkrétní technologii. V případě mikroservis architektury je ale potřeba každou komponentu testovat samostatně, a navíc je nutné testovat i framework nebo router, který se stará o finální kompozici celé aplikace.[5]

3.3.2 Vysoké náklady u rozsáhlých aplikací

Aplikace s použitím mikroservis architektury se mohou rozšiřovat víceméně neustále. Přitom je ale potřeba myslet na to, že množství odlišných komponent s sebou nese také množství nákladů. Při použití různých technologií je potřeba zajistit kvalifikovaný vývojářský tým, webové prostředí i datové prostředky pro každou zvlášť. Tím se může vývoj aplikace oproti monolitické výrazně prodražit.[5]

4 Architektonické a návrhové vzory

4.1 Architektonické vzory

Pro vývojáře znamenají architektonické vzory předpis pro vývoj software podle daného vzoru. Jedná se o abstraktní předpisy, které nevypovídají nic o konkrétní implementaci a funkci systému. [6]

Základní úkoly architektonických vzorů jsou:

- Definice struktury aplikace – jednotlivé architektonické vzory rozdělují aplikace do struktury modulů a vrstev. Mohou být rovněž monolitické, nebo tvořené mikroservisy.[7]
- Definice chování – nejedná se o konkrétní implementaci nebo jazyk. Jak ale bude popsáno v konkrétních vzorech, tak jednotlivé části aplikace mají vždy svůj jasný účel.[6]
- Definice vztahů – jednotlivé části aplikace mezi sebou mohou nabývat různých úrovní vztahů a komunikace, ale běžně mezi sebou také žádný vztah nemají.[7]
- Zajištění udržitelnosti – díky striktnímu oddělení konkrétních částí aplikace je kód mnohem lépe udržitelný a rozšiřitelný.[6]
- Možnost přenositelnosti – mnohé architektonické vzory nabízí možnost jednoduché přenositelnosti aplikace mezi systémy a možnost komunikace s jinými, zcela odlišnými aplikacemi.[7]
- Zefektivnění vývoje – díky striktnímu rozdělení mohou na částech aplikace pracovat jedinci i týmy bez závislosti na ostatních.[7]

Mimo svých základních funkcí a výhod staví architektonické vzory před vývojáře také mnohá rozhodnutí. Použití konkrétních jazyků a technologií, včetně jejich implementace, nemusí být vždy vhodné pro vybraný vzor. Dá se říci, tak definují i množinu prostředků, které je rozumné pro vývoj použít.[8]

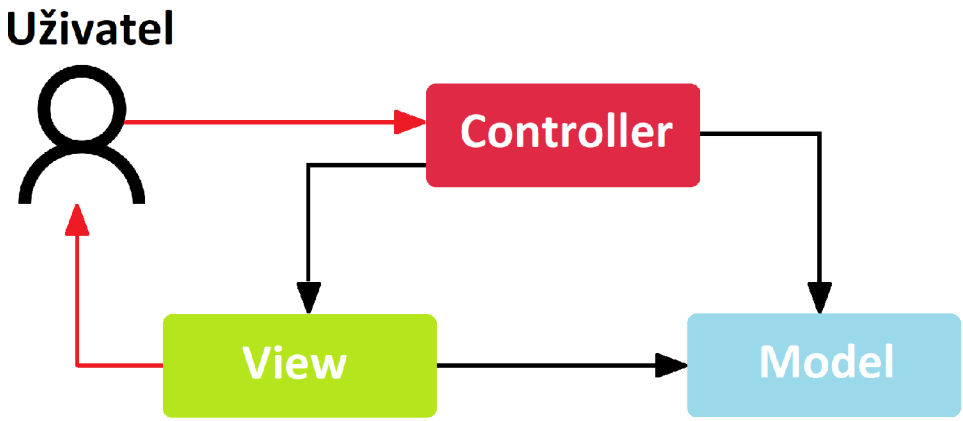
4.1.1 Model-View-Controller

Jedná se o architektonický vzor, který umožňuje rozdělit aplikaci na vzájemně oddělené části, které oddělují datovou, aplikační a prezentační vrstvu. Model-View-Controller (dále jen MVC) je hojně používaným architektonickým vzorem právě u webových aplikací. MVC umožňuje rozdělit data, jejich zpracovávání a zobrazení do jednotlivých částí, které mezi sebou mají minimální vazbu.[9]

- Model obsahuje interpretaci dat, v případě webových stránek většinou databáze, nebo jejich předpis v podobě datových tříd.[9] Model je zcela nezávislá část aplikace. Nemá žádné povědomí o controller a view, ani žádné závislosti. Díky tomu se tak mohou měnit zdroje dat, ale aplikace přitom může zůstat plně funkční bez zásahu do dalších částí.[9]

- View je zobrazení uživatelského prostředí a dat. View dokáže nahlížet do modelu, protože zobrazuje data získávaná právě z modelu.[9]

- Controller je nad view i modelem. Dokáže nahlížet do obou částí aplikace a dokáže s oběma spolupracovat. Controller čeká na uživatelskou interakci, která proběhne ve view. Na základě toho pak nabídne jiné view (např. přechod ze stránky na stránku), nebo změní model (např. přidání kontaktu do databáze).[9]



Obrázek 3: Diagram MVC Architektury

Komunikace v aplikaci založené na MVC probíhá tak, že jakékoliv akce v uživatelském prostředí eviduje controller. Zde se vždy rozhoduje o tom, jaká akce se stane. V případě potřeby controller přistoupí k modelu a aktualizuje jej na základě konkrétní operace. Následně view nahlédne do modelu a zobrazí konkrétní data v rámci uživatelského rozhraní.[10]

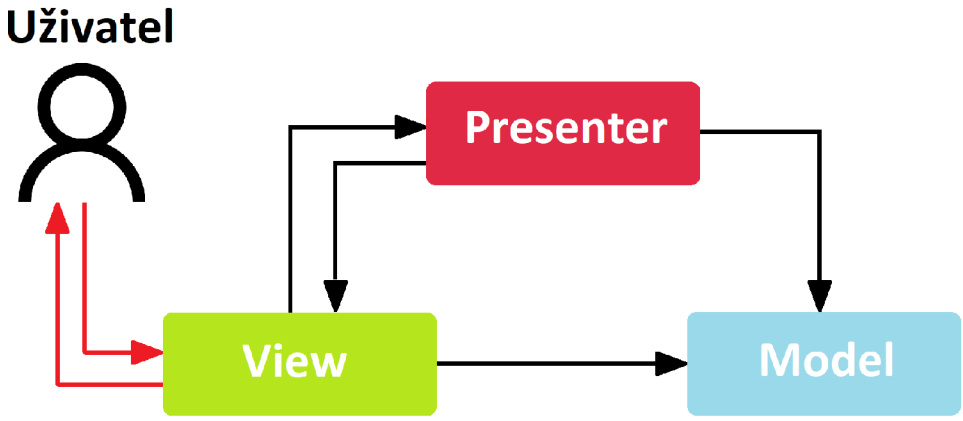
4.1.2 Model-View-Presenter

MVP je architektonickým vzorem, který je odvozený od předchozího vzoru MVC. Rozdíl je ale v prezentační vrstvě samotné. Rozdělení jednotlivých částí na Model-View-Presenter je si s MVC podobné, ale zodpovědnosti jsou zde jiné. V rámci MVC se používá co nejprimitivnější view, které pouze prezentuje obdržená data, ale nevyužívá se zde koncept návrhového vzoru data binding.[11]

- Model obsahuje data specifická pro danou doménu. I v tomto případě je pouze prezentací dat samotných, bez jakékoliv další logiky.[11]
- View je zde co nejjednodušší zobrazení, většinou se tak jedná o HTML, XML dokumenty. V rámci MVC view reaguje na změny v modelu, zatím co zde s view nemá žádnou vazbu, a tudíž ani nereaguje na jakékoliv

změny. To je zásadní rozdíl, který zcela mění princip MVC a MVP.[11]

- Presenter obsahuje veškerou business logiku a vidí jak do model, tak i do view. Jakmile se ve view stane událost, tak ji presenter obdrží a upraví náležitě model. Následně zaktualizuje view podle nového modelu.[11]



Obrázek 4: Diagram MVP Architektury

Aplikace postavená na MVC funguje tak, že jako první se hlásí view o data do presenteru. Presenter následně nahlédne do model, vybere a popřípadě upraví data podle potřeby, a následně je odešle do view k zobrazení. Pokud na stránce dojde k události, tak o ní opět view informuje presenter, který manipuluje s modelem a jeho novou podobu opět vrací do view.[11]

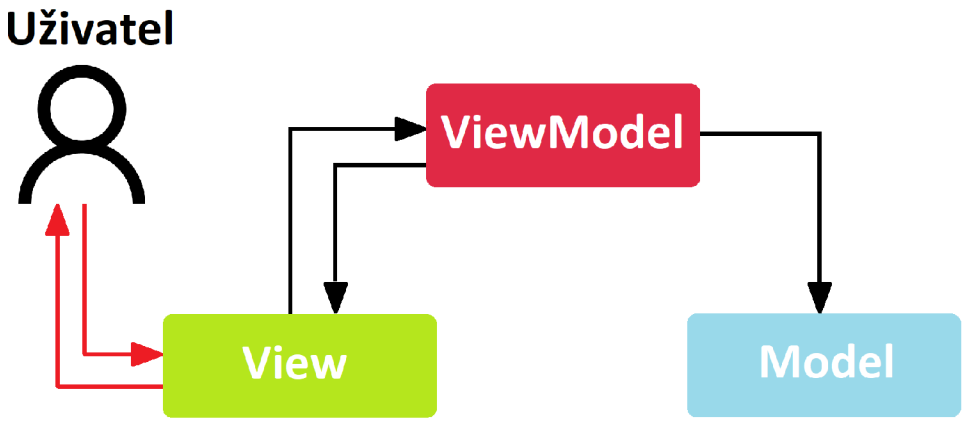
Specifikem MVP je možnost několika provedení presenteru. Ta se liší podle toho, o jakou konkrétní implementaci se jedná. Mimo standardního presenteru se používá také Supervising Controller, který stále funkcí zůstává MVP architekturou, ale díky aplikaci návrhového vzoru data binding může model a view částečně práci původního presenteru obejít. Tím se tato verze MVP výrazně blíží MVC a MVVM architekturám.[11]

4.1.3 Model-View-ViewModel

Architektonický vzor Model-View-ViewModel (dále jen MVVM) je hojně používaným architektonickým vzorem pro tvorbu Single Page Aplikací vytvořený pro potřeby vývoje na platformě ASP.NET. MVVM umožňuje plně oddělit business logiku a prezentační logiku od zobrazení uživatelského prostředí, což zjednodušuje vývoj a údržbu jednotlivých částí aplikace. Také MVVM je změněnou variantou MVC, ale se zásadním rozdílem v interaktivitě aplikace.[12] Původní název MVVM, který se občas používá dodnes, je Presentation Model.[10]

MVVM aktivně využívá návrhového vzoru data binding. Díky tomu je tak vhodnou volbou pro data, která jsou zobrazena ve statickém view, ale přitom se dynamicky mění díky přiřazení dat z viewModel.[10]

- Model i v tomto případě obsahuje konkrétní data, která jsou v rámci aplikace používána.[12]
- View se v rámci MVVM stará o statické části uživatelského rozhraní aplikace. Typickými příklady jsou statické HTML nebo XAML dokumenty. Do view jsou pomocí návrhového vzoru data binding vložena data.[12]
- ViewModel pracuje s view a modelem zároveň. Jsou v něm obsaženy všechny reakce na uživatelem vyvolané události ve view a také to, jaká data a jak lze pomocí data binding do view promítnout. Nezanedbatelnou funkcí viewModel je rovněž dohled nad synchronizací zobrazených dat ve view s těmi, které obsahuje model.[12]



Obrázek 5: Diagram MVVM Architektury

V praxi MVVM funguje následujícím způsobem. Model obsahuje konkrétní reprezentaci dat pro danou aplikaci. ViewModel nahlédne do model, vybere a upraví data, která následně pomocí data binding předá do view. View pouze zobrazuje data a uživatelské prostředí. ViewModel pak přes zmíněný data binding sleduje změny ve view. V případě uživatelské interakce viewModel na základě obsažených funkcí aktualizuje model i view současně.[12]

4.2 Návrhové vzory

Při vývoji software a aplikací se vývojáři běžně potkávají se stejnými výzvami a překážkami. Návrhové vzory jsou cestou, jak se jim vyhnout. Jednoduchou definicí návrhových vzorů je opakovatelně použitelné řešení problémů, se kterými se denně v rámci softwarového inženýrství potkáváme. Jsou ověřeným řešením konkrétních situací, které disponuje možností jednoduché aplikace na daný problém u více i méně robustních aplikací.[10]

- Vytvářející (Creational Design Patterns) jsou takové vzory, které jsou zaměřené na kontrolu a řízení tvorby objektů. Mohou ovlivňovat podobu objektů, ale také možný počet instancí daného objektu.[10]

Příklady: Factory, Singleton

- Strukturální (Structural Design Patterns) se starají o rozdělení aplikace do přehledné struktury. Nejčastějším důvodem užití strukturálních vzorů je zpřehlednění aplikace.[10]

  Příklady: Facade, Adapter

- Chování (Behavioral Design Patterns) umožňují kontrolovat chování tříd a podtříd včetně řízení dědičnosti a zajistit komunikaci mezi třídami.[10]

  Příklady: Iterator, Observer

Součástí výhod, které návrhové vzory nabízí, je stejně jako u architektonických vzorů vysoká míra standardizace díky kvalitní dokumentaci. Většina populárních návrhových vzorů je používána ve velkém měřítku napříč celou řadou jazyků a vývojářských nástrojů. Často se tak u jednoho návrhového vzoru setkáme s více možnostmi konkrétní implementace. Díky předem dané konstrukci navíc zjednodušují refaktorizaci a zpřehledňují kód.[10]

4.2.1 Singleton

Díky návrhovému vzoru Singleton, v překladu „jedináček“, je možné kontrolovat počet vzniklých instancí dané třídy. V některých případech je totiž potřeba, aby daný proces byl aktivní pouze jednou. Jedná se především o observační nebo autorizační služby. Základním principem návrhového vzoru Singleton je fakt, že jím aplikované třídy mohou nabýt pouze jediné instance. První založení instance funguje stejně jako bez aplikace Singleton, ale jakmile dojde k zakládání druhé instance, tak se vzniku zamezí a namísto spuštění standardně dává k dispozici referenci na již fungující instanci.[10]

V rámci tvorby Single Page aplikací pomocí micro-frontend je Singleton standardně používaný u frameworků, které se starají o směrování mezi jednotlivými komponenty stránky.

4.2.2 Data Binding

Jedná se o návrhový vzor, který zajišťuje možnost spolupráce mezi model a view. Je základem pro technologie, kdy je view prezentováno jako co nejprimitivnější uživatelské prostředí. V něm se na konkrétních místech zobrazují data, která jsou pomocí binding spojena s modelem. Data binding je nedílnou součástí architektury MVVM.[13]

Z použitých technologií v rámci práce má podporu návrhového vzoru data binding framework Angular. Jedná se navíc o podporu v obou směrech, takzvané „two-way binding“. To znamená, že jsou spojena data ve view a modelu, přičemž jsou oboustranně reaktivní. Pokud se změní model, automaticky se změní i view, a naopak pokud se změní view, tak se s ním mění i model. Díky tomu jsou tak obě části architektury neustále synchronizované. Jiné frameworky, například React.js, podporují jednostranné spojení dat.[14]

4.2.3 Dependency Injection

Návrhový vzor ve volném překladu zvaný jako vkládání závislostí, je jednou z cest, jak zajistit závislosti v aplikacích tvořených nezávislými komponentami. Ze základů objektově orientovaného programování je známo, že každá třída má být zapouzdřená. Pokud je ale potřeba některou její část poskytnout jiné třídě, tak se musí využít speciální třídy jménem rozhraní (v programovacích jazycích psaná jako interface), metody umožňující převzít data ze zapouzdřené třídy (gettery a settery) nebo implementovat dependency injection do konstruktoru zodpovědného za vnik dané třídy.[15]

Základním rozdílem mezi použitím dependency injection a předáním kódu je ten, že injection nabízí pouze samotný výstup. To znamená, že druhá komponenta, která přijímá injektovaná data z první komponenty zná výsledek, ale netuší nic o vnitřní implementaci kódu. Dependecy injection je návrhový vzor, který řeší problémy spojení mezi komponentami, které jsou jinak samostatně zapouzdřené. Celý proces funguje na bázi producenta a konzumenta, kdy pro-

ducent je komponenta nabízející přes návrhový vzor výstup vnitřních funkcí a konzument přijímá výsledky. Možných implementací Dependency Injection je celá řada v závislosti na konkrétní technologii. Například JavaScript sám o sobě nedokáže využít nejčastější implementaci pomocí rozhraní, protože jako jazyk třídu typu rozhraní nezná.[15]

4.3 Data

4.3.1 Document Object Model

Document Object Model (dále už jen DOM) je objektovou interpretací XML nebo HTML dokumentů. Z logického hlediska DOM tvoří stromovou strukturu, která hierarchicky navazuje na samotnou strukturu webové stránky.[16] Mezi jednotlivými větvemi stromu jsou uzly, které mohou reprezentovat element na stránce, atributy elementu, nebo textová data. Přitom hierarchie mezi jednotlivými uzly je pevná podle toho, jak je psaný samotný XML/HTML dokument.[16]

Základem funkce DOM je fakt, že lze s HTML stránkou manipulovat pomocí JavaScript editací jednotlivých uzlů. Každý uzel je z pohledu DOM objektem a lze tak k němu přistupovat a manipulovat s ním několika způsoby.[16]

Nejpoužívanější tři způsoby jsou:

- K uzlům ve stromě můžeme přistupovat voláním objektu document, který v kódu automaticky identifikuje kořenový prvek (root) a použije jej jako startovní bod k hledání konkrétního uzlu ve stromu.[16]
- Výchozí bod lze identifikovat získáním konkrétního uzlu, se kterým uživatel manipuloval. Například pozorování stisku tlačítka voláním metody eventListener.[16]
- Voláním objektu document se zaměřením konkrétního uzlu jako vstupního bodu pomocí metody getElementById (zde se vracíme k již zmíněnému ID identifikátoru v HTML).[16]

Zmíněné způsoby jsou platné v případě, kdy se pracuje s DOM vycházejícího z HTML dokumentu. V takovém případě je k dispozici rozhraní Document, Element a HTMLElement.[17] Všechny umožňují identifikovat standardní HTML tagy a jejich atributy. HTML DOM API je pak cestou k dalším vlastnostem jako je získání přístupu k oknům, záložkám v prohlížeči, historii nebo vlastnostem prohlížeče a mnohým dalším.[17]

5 Webové technologie

5.1 HTML

HTML (zkratka angl. HyperText MarkUp Language) je základním jazykem pro tvorbu stránek v prostředí World Wide Web. Část názvu „Markup“ upozorňuje na dovednost HTML identifikovat pomocí značek (tagů) jednotlivé elementy v rámci webové stránky. Hypertext následně označuje dovednost užívání odkazů (angl. hyperlink, nebo zkráceně jenom link), umožňujícím odkazovat mezi jednotlivými webovými stránkami.[16]

HTML oproti všem dalším jazykům užívaným v práci nepotřebuje žádné překladače. Jedná se o jazyk, kterému operační systémy rozumí bez potřeby webového serveru, nebo podpory prohlížečů. Díky tomu je tak možné jej spouštět a vykreslovat bez omezení.[16]

5.1.1 ID selektor

ID je jedním ze základních selektorů, které slouží k identifikaci prvků na webové stránce. Vedle používaného selektoru class je ale zásadně rozdílné v tom, že ID musí být vždy unikátní v rámci konkrétní webové stránky.[16]

V ukázce zdrojového kódu je tag pro tvorbu textového odstavce. Má přiřazenou class odstavec, která může být shodná i pro jakoukoliv jinou, a také unikátní ID. Pokud bude chtít kód odkazovat na odstavec, nebo s ním pracovat v rámci micro-frontend frameworků, bude užívat právě ID.[18]

V rámci tvorby základních jednostránkových webů je ale zásadní jiná dovednost, kterou selektor ID má. Může být jednoduše použitý k tomu, aby kód dokázal odkazovat na konkrétní část webové stránky přes hypertextový odkaz. Prohlížeč přitom nemusí v žádném případě stránku znovu načítat. Při kliknutí na odkaz stránka doslova „skočí“ na pozici, na které je konkrétní ID umístěno. Z toho důvodu by se nikdy nemělo ID na stránce opakovat, aby odkaz dobře věděl, na který prvek se má stránka posunout.[16]

Zásadním rozdílem, který ale oproti Single Page aplikacím má použití HTML je fakt, že zde stránka pracuje pouze se statickými daty. Implementaci ID tak lze jednoduše použít k přecházení po jednostránkovém webu. Vlastnosti HTML už ale nebudou stačit k tomu, aby stránka dokázala pracovat s dynamickým obsahem na základě eventů, které se na stránce odehrají.[16]

5.2 Javascript

Historicky byly před příchodem JavaScript webové stránky pouze statické. HTML ani CSS totiž nemá žádné možnosti, jak může dynamicky pracovat se změnou obsahu stránky, s výjimkou vizuálních efektů. JavaScript se tak stal prvním nástrojem, který umožnil udělat stránky interaktivní a zlepšil tak výrazně uživatelskou zkušenost.[16]

Jako unikátní jazyk byl v roce 1998 JavaScript uveden společností Ecma International mezi standardy, které by měl splňovat každý prohlížeč. Na rozdíl od HTML totiž JavaScript neoperuje striktně na serveru, ale může pracovat přímo v prohlížeči uživatele, čímž je zásadně odlišný.[16] JavaScript je skriptovací jazyk, který umožňuje vyvíjet webové aplikace jak procedurální cestou, tak i pomocí objektově orientovaného návrhu. Práce s objekty je přitom jednou z nejzásadnějších vlastností také pro Single Page Aplikace, protože se jako objektová interpretace dat počítá i DOM.[16]

5.3 Front-end Frameworky

Frameworkem se v programování rozumí rozšíření pro konkrétní jazyk, který podle svého určení napomáhá k jednoduššímu vývoji. Základem pro úspěch při používání frameworku je ale dostatečná znalost jazyka jako takového.[20] Vývojář nemusí vše psát, protože framework zpravidla dokáže nabídnout kompletní základ. Časová úspora je tak jednou z hlavních výhod. Mimo samotné úspory je pak důležitou vlastností frameworků i fakt, že jsou vytvářeny a testovány zkušenými vývojáři a organizacemi. Díky tomu tak použití frameworků

značně napomáhá i zvýšení bezpečnosti, vyvarování se chybám během vývoje a také standardizaci aplikací.[20]

V rámci Single Page Aplikací s micro-frontend a všeobecně vývoje interaktivního frontend jsou podstatné frameworky pro jazyk JavaScript. Konkrétně se jedná o takzvané client-side frameworky, které umožňují vykreslovat uživatelské rozhraní přímo v prohlížeči klienta a frameworky zajišťující komunikaci mezi komponentami v rámci micro-frontend architektury.[21]

5.3.1 React.js

React.js je velice populární JavaScript framework pro tvorbu uživatelského prostředí. Vyvinut byl v roce 2013 společností Facebook (dnes již Meta). Důvodem byl fakt, že žádný dobový framework neodpovídal přesně potřebám společnosti. Výhodou React.js je poměrně jednoduché pochopení pro uživatele s předchozími zkušenostmi s JavaScript.[22]

React.js totiž efektivně spojuje možnost psát HTML kód a JavaScript do jediného JSX dokumentu. React.js jako knihovna navíc obsahuje velké množství již připravených komponent pro standardní úkony. Základním účelem React.js je už podle jeho tvůrců budování rozsáhlých webových aplikací s častou změnou dat. K tomu přistupuje velice jednoduchou syntaxí. Stejně jako další zmíněné frameworky využívá také React.js DOM.[23] Velkým rozdílem ale je přístup k celému systému DOM, který je zde odlišný. Proto se také často setkáme s názvem ReactDOM, protože v tomto provedení je od původního DOM podstatně odlišný. Důvodem je fakt, že framework necílí pouze na vývoj čistokrevných webových aplikací, ale umí s DOM pracovat také v rámci nativních aplikací.[23]

Nevýhodou React.js je potřeba práce s dalšími knihovnami jako je Redux nebo React Router, aby dokázal konkurovat frameworkům jako je Angular. Mimo toho je u React.js problematické realizovat „two-way binding“ mezi modelem a view. Základní funkcionalitou je jednosměrná reaktivita z modelu do view.[22]

5.3.2 Angular

Angular je jedním z nástrojů, které nám umožňují tvořit plně škálovatelné a udržitelné Single Page aplikace. Vychází z původního frameworku Angular.js, ale je jeho kompletním přepisem pro potřeby budování moderních webových aplikací, se zachováním základních funkcí a přístupu k datům.[24]

Oproti standardnímu JavaScript kódu přináší Angular celou řadu výhod. Mezi ty nejpřednější patří možnost tvořit vlastní komponenty, které mohou být opakovaně užívány napříč aplikací. Kvalitně psaná Angular aplikace tak značně eliminuje duplicitní kód. Jednotlivé komponenty jsou v Angular definovány pomocí direktivy. Umožňují nejen vyvarování se duplicitnímu kódu, ale také načítání dat jako souboru mikroservis komponent. Každá komponenta by v kvalitním návrhu měla mít jednoznačné a jednoúčelové chování. Jednotlivé komponenty na sebe přitom mohou navazovat voláním a komunikovat zprávami.[24] Angular dále používá návrhové vzory data binding a dependency injection. Ty umožňují práci s daty tak, aby bylo možné oddělit view od zbylé logiky aplikace. V neposlední řadě je potom Angular nástrojem, který umožňuje automatické testování (Unit Test v angl. originále).[24]

Stejně jako celá řada JavaScript frameworků se i Angular opírá o již existující technologie. Pro vývoj tak programátor potřebuje pracovat s Node.js, TypeScript a samotná instalace Angular probíhá přes balíčkovací systém NPM.[24]

5.3.3 Vue.js

Hlavním znakem frameworku Vue.js je jeho jednoduchost. Díky tomu je tak samotné jádro frameworku mnohem jednodušší a méně komplexní, než je tomu u konkurenčních řešení. To ale neznamená, že je Vue.js pouze technologií pro co nejjednodušší prezentační vrstvu. Lze jej jednoduše rozšířit dalšími komponentami jak od autorů Vue.js, tak i komponentami psanými v čistém JavaScript.[25]

Vue.js podporuje vývoj na základě vykreslování jednotlivých komponent.

Všestrannost Vue.js je podpořena jeho širokým využitím, které zahrnuje nejen kompletní Vue.js frontend, ale také vkládání komponent na jinak statickou stránku bez závislosti na tom, jestli se o vykreslování frontend stará klient nebo server.[26]

K vývoji ve framework Vue.js můžeme přistupovat dvojicí odlišných způsobů, Options API a Composition API.[26] Options API je bližší vývojářům s předchozími zkušenostmi z objektově orientovaného programování, protože aplikuje velice podobnou strukturu. Komponenty jsou zde prezentovány jako objekty, které mají svoje vlastní hodnoty a metody. Stejně jako v objektově orientovaném návrhu komponenty tvoří instance.[26] Oproti tomu Composition API představuje efektivnější metodu vývoje ve Vue.js, který dokáže plně využít potenciál frameworku, ale je vhodnější pro pokročilé uživatele. Přístup vyžaduje pochopení reaktivity ve Vue.js a aktivně využívá integrované metody základní API.[26]

5.4 Package Managers

Správci balíčků jsou nástroje, které umožňují instalovat softwarové balíčky, spravovat jejich uložení v systému, aktualizovat jejich obsah a popřípadě mohou být i prostředkem na sdílení svých vlastních balíčků. Mimo těchto funkcí správci balíčků kontroluji i vytvoření duplicitních instalací. Dále mají podstatnou výhodu v tom, že dokáží spravovat celý ekosystém daného software, díky čemuž máte jistotu funkce od nainstalování.[27]

Aby mohl být balíček stahovatelný, tak musí být zapsaný v některém z registrů balíčků. Nepopulárnější je registr NPM, ale existují i další registry jako jsou kupříkladu GitHub Registry Service nebo správa balíčků přes Microsoft Azure.[27]

Důležitým pojmem pro pochopení funkce správců balíčků jsou závislosti (dependencies). Libovolné projekty nemusí mít žádné závislosti, ale také jich mohou mít nespočet. Závislost v tomto případě znamená napojení vyvíjené

aplikace na software třetí strany.[27] Typickým příkladem je závislost Single Page aplikace na použitém frameworku. Přidání závislostí do projektu lze realizovat i jinými cestami, v případě JavaScript například vložením zdrojových skriptů přes HTML tag <script>. Problémem ale je, že v tomto případě může být komplikované hlídání nových verzí software třetí strany. Oproti tomu správci balíčků dokáží software aktualizovat a jeho případné přidání a odebrání probíhá na bázi několika příkazů. A to s jistotou, že se jedná o čistou instalaci i odinstalaci, což u hledání odkazů na software třetích stran není ve zdrojovém kódu vždy jednoduché.[27]

5.4.1 Npm

Je správce balíčků pro JavaScript, konkrétně pro aplikace pracující s prostředím Node.js. Používá se k bezpečné instalaci a nastavení různých balíčků, mezi které patří i frameworky React.js, Angular a Vue.js.[28] Npm je největším světovým registrem softwarových balíčků. Pro běžný vývoj nabízí Open Source řešení a stejně tak prostředí npm Organizations pro volně dostupné veřejné sdílení balíčků, ale i pro placenou funkci soukromých balíčků.[29]

První funkcí npm je při instalaci nastavení závislostí v aplikaci (dependencies). Díky tomu tak mohou vývojáři okamžitě využívat funkce frameworků, protože se o všechnu direktivu a prevenci chyb stará npm. Následně pak v průběhu instalace npm samo vytvoří v adresáři, ve kterém pracujeme kompletní adresářovou strukturu, do které následně ukládá data konkrétního balíčku nebo frameworku.[28]

Stažení aplikačních balíčků ale není jedinou funkcí systému. Další důležitou funkcí je možnost tvořit vlastní balíčky, které pak přes prostředí verzovacího systému Git mohou instalovat další uživatelé. To značně ulehčuje sdílení aplikací v JavaScript mezi pracovními týmy i při konečném spouštění na ostrém serveru. Součástí balíčku mimo zdrojových kódů projektu běžně bývá i potřebný framework a všechny komponenty, díky čemuž tak npm může zaručit dostupnost prostředků na místě stažení.[29]

6 Praktická část

6.1 Příprava pracovního prostředí

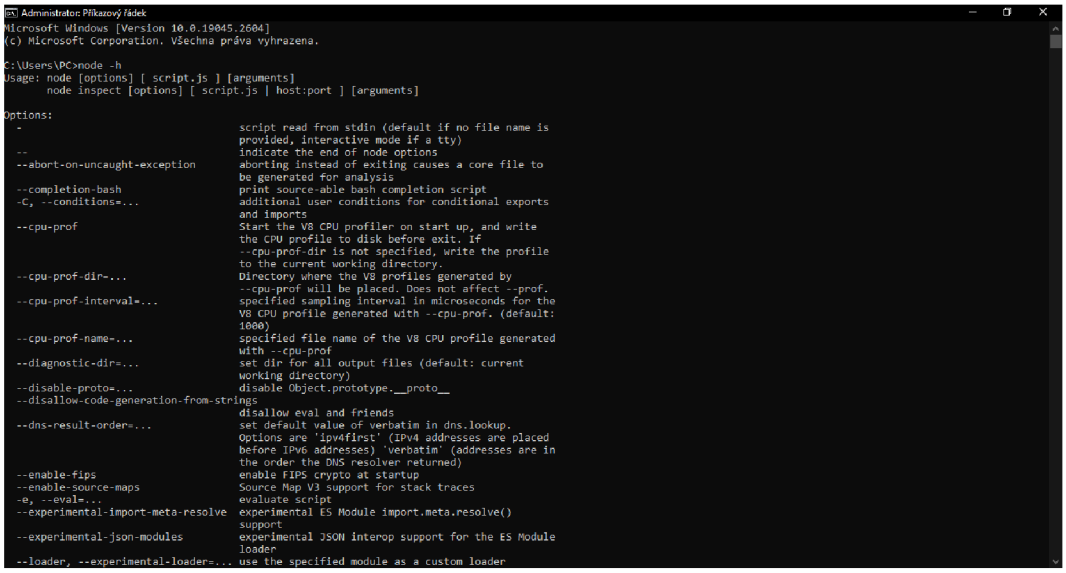
6.1.1 Instalace Node.js

Node.js je JavaScript platforma umožňující vytvářet škálovatelné aplikace. Základ Node.js funguje jako server-side. Pro moji práci je podstatný proto, abych dokázal na lokálním pracovním prostoru vytvořit ekosystém pro běh JavaScript frameworků, které jsou na jeho existenci závislé.

Jedná se o open source prostředí, které lze z oficiálních stránek stáhnout v podobě instalátoru pro systémy Windows, Linux i MacOS. Po instalaci je prostředí Node.js připraveno k práci. Pro ověření případné funkcionality stačí v terminálu vývojového prostředí, nebo v příkazovém řádku použít příkaz níže.[30]

```
node –h
```

Příklad 1: Výpis možností Node.js



Obrázek 6: Ověření úspěšné instalace Node.js

6.1.2 Instalace npm

V praktické části zmíněný správce balíčků npm pro moji práci hraje zásadní roli v tom, že s jeho pomocí dokážu instalovat aplikace přes terminál se zajištěním automatizace všech závislostí. Další možností je vložení odkazů přes HTML na skripty daných frameworků, ale v tomto případě by budoucí aplikace nebyla schopna ohlídat aktuální verze. Npm naopak samo informuje o aktuálních verzích a nabízí rychlou aktualizaci. Správce npm lze instalovat jednoduchým příkazem, ale v systému by již měla být existující instalace Node.js.[31]

```
npm install -g npm
```

Příklad 2: Instalační příkaz pro npm

V příkazovém řádku pak lze ověřit přítomnost npm výpisem jeho verze.[31]

```
npm -v
```

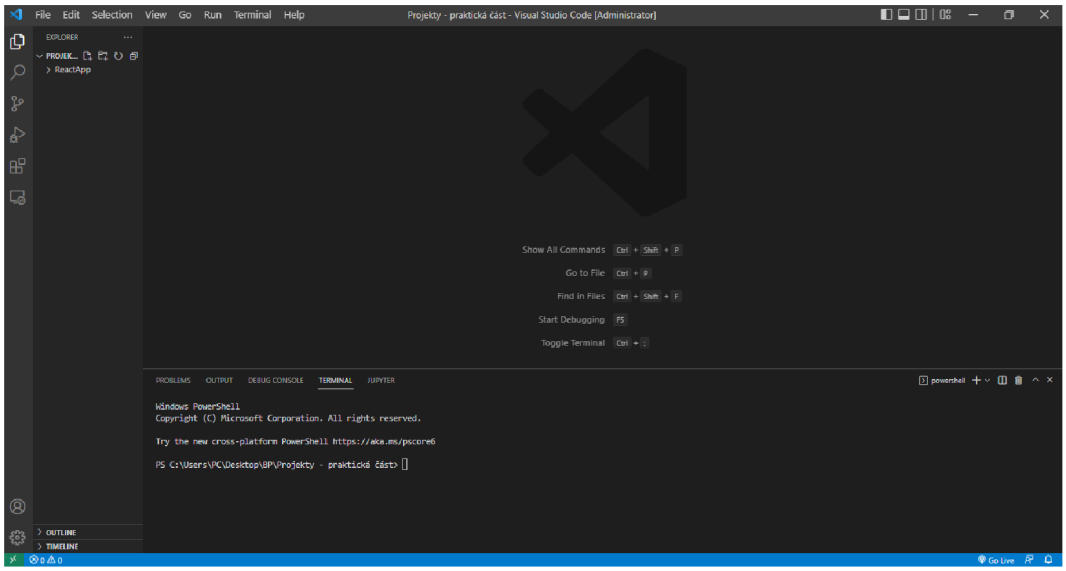
Příklad 3: Zjištění verze instalovaného npm

Obrázek 7: Ověření úspěšné instalace npm

6.1.3 Vývojové prostředí

Jako hlavní vývojové prostředí pro práci jsem zvolil Visual Studio Code. Jedná se o volně dostupné vývojové prostředí od společnosti Microsoft se širokou podporou jazyků. Prostředí umožňuje rozšířit základní funkce pomocí manažera rozšíření. Ten umožňuje instalovat podporu nových jazyků, frameworků a funkcí jako jsou Debuggery, Live Server, grafická rozhraní a mnohá další.[32]

V rámci bakalářské práce aktivně využiji rozšíření pro JavaScript frameworky a Live Server, který slouží k vytvoření lokálního serveru pro Node.js aplikace.



Obrázek 8: Prostředí Visual Studio 1.70.1

6.2 Single-spa framework

6.2.1 Proč single-spa?

Jedná se o JavaScript framework, který je navržený přesně pro potřeby tvorby Single Page aplikací s technikou micro-frontendu. Ve své podstatě se jedná o jednoduchý router, který umožňuje rychle propojit části micro-frontend z různých modulů na stránce.[33]

Z mého úhlu pohledu je ale zásadní výhodou možnost používat na jediné stránce více micro-frontend frameworků, přičemž mezi nimi single-spa umožňuje i realizovat komunikaci. Z této výhody single-spa o to více čerpá tím, že má širokou podporu moderních JavaScript technologií a frameworků. Setkáme se tak s podporou Webpack, Module Federation, SystemJS, TypeScript a dalších technologií. Pro nejpopulárnější frontend frameworky jako jsou mnou užité React.js, Angular a Vue.js navíc nabízí single-spa vlastní balíčky funkcí, které dokonce umožňují jednoduše zajistit kvalitní směrování a hlídání životního cyklu každé komponenty i celé aplikace jako celku.[33]

Funkce single-spa přitom věrně kopíruje zásady Single Page aplikací, jak

byly popsány v úvodu praktické části. Při prvním načtení framework odešle vše potřebné pro prezentační vrstvu směrem ke klientovi a následně pomocí vazby na konkrétní pozice v HTML dokumentu a dynamické DOM mění zobrazovaná data. Dalším důvodem, proč jsem si vybral práce single-spa je možnost jednoduchého založení prostředí pro Single Page aplikace, přičemž ale praktické výsledky frameworku mluví samy za sebe.[33]

Mezi dobře známé aplikace postavené na single-spa patří kupříkladu:

- Gmail
- Google Maps
- Facebook
- GitHub

6.2.2 Instalace frameworku

Je několik možností, jak instalovat single-spa na lokálním serveru. Nejpraktičtější cestou je využití správce balíčků npm pomocí následujícího příkazu do terminálu Visual Code ve složce, kde si přeji instalovat balíček.

```
1 npm i -g create-single-spa
```

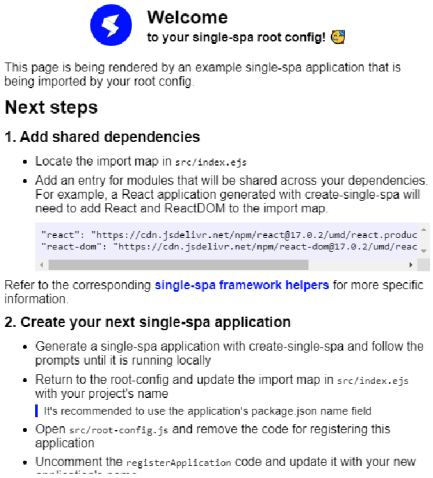
Příklad 4: Příkaz pro stažení instalačního balíčku přes npm

Tím dojde ke stažení všech potřebných souborů pro vytvoření aplikace v single-spa. Následným příkazem pak tvořím první single-spa projekt. Během instalace se skript automaticky zeptá na podrobnosti. Nedůležitější volbou je typ generování. Na výběr jsou možnosti single-spa application / parcel, in-browser utility module nebo single-spa root config. Poslední možnost slouží k nastavení aplikace od úplného začátku, což je v tento moment můj případ. Během instalace se instalační skript doptá zdali chci užívat v novém projektu layout designer, který je součástí single-spa a je jednou z možností, jak spravovat rozložení budoucí Single Page aplikace.[33]

```
create-single-spa
? Directory for new project single_app
? Select type to generate single-spa root config
? Which package manager do you want to use? Npm
? Will this project use Typescript? No
? Would you like to use single-spa Layout Engine Yes
? Organization name (can use letters, numbers, dash or
    underscore) Bakalarska-Prace
```

Příklad 5: Instalace kořenové aplikace single-spa

Po dokončení instalace jsem přešel do složky s aplikací a zde otestoval její funkčnost pomocí příkazu, který má za úkol spustit balíček ukrytý v daném adresáři. V souboru webpack.congif.js lze identifikovat odkaz na kterém bude aplikace aktivní. Standardně to je localhost na portu 9000, kde se nám zobrazí úvodní obrazovka aplikace.



Obrázek 9: Úvodní stránka single-spa

6.3 Instalace serverových mikroservis

6.3.1 Angular komponenta

Angular je z použitých frameworků bezesporu nejrozsáhlejší. Proto se tak odlišně přistupuje i k jeho instalaci. I zde je možnost využít instalátor samotného single-spa, ale pro opravdu kompletní framework je lepší instalovat jej samostatně. Průvodce instalací samotné aplikace požaduje jenom název a určení toho, zdali budu chtít využívat Angular router a kaskádové styly.

```
> ng new angular-app
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
```

Příklad 6: Instalace čistého Angular

Tím je nainstalovaný samotný základ aplikace Angular, ale ještě potřebuji zařídit, aby mohla komunikovat s frameworkem single-spa. Toho docílím pomocí následujících příkazů a instalace.

```
> ng add single-spa-angular
? Using package manager: npm
Need to install the following packages:
  @angular/cli@15.2.2
Ok to proceed? (y) y
? Would you like to add Angular routing? Yes
? What port should your project run on? 8081
```

Příklad 7: Instalace single-spa do Angular

Během instalace se skript ptá dokonce i na port na kterém bude aplikace fungovat, čímž mi dokáže ulehčit práci při konfiguraci jednotlivých komponent. V průběhu jsou do původního Angular nainstalovány především všechny závislosti mezi Angular a single-spa.

Po instalaci jsem následně měl problémy se samotnou kompilací programu přes spuštěním. Důvodem byla chybějící část aplikace environment. Následně

jsem zjistil, že v instalacích posledních verzí Angular se jedná o poměrně dost častý problém, který lze vyřešit za pomoci následujícího příkazu.[34]

```
ng generate environments
```

Příklad 8: Vytvoření environment pro Agular

Po spuštění příkazu byla vytvořena v sekci app, která reprezentuje samotnou aplikaci, složka environment a v ní dva dokumenty s prázdnými metodami. Samotné metody přitom pro spuštění potřebují předat správné hodnoty a z dokumentace jsem tak doplnil metody následujícím způsobem.[34]

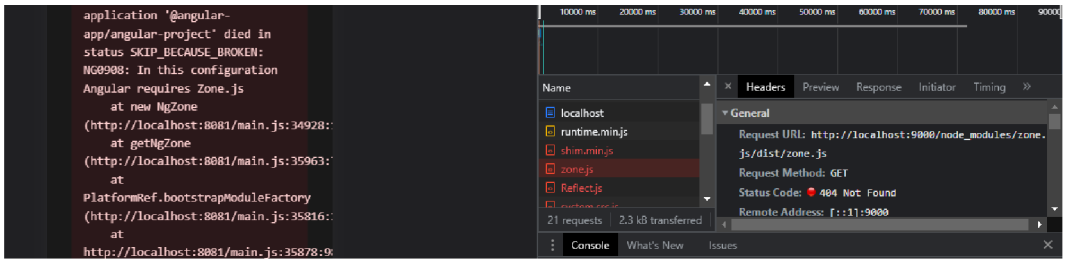
```
Soubor: environment.development.ts
export const environment = {
  production: false,
  apiUrl: 'http://my-api-url'
};

Soubor: environment.ts
export const environment = {
  production: true
};
```

Příklad 9: Podoba metod v src/environment

Po doplnění environment již samotná kompilace proběhla v pořádku. Po nastartování lokálního serveru jsem se ale setkal s další komplikací, která se u Angular ukazuje poměrně běžně. A to byla absence reference na vyžadované Zone.js.

Obrázek 10: Hlášení kompilačních chyb v Angular

Angular disponuje příjemným prostředím hlášení chyb. Zatím co jiné frameworky většinou nabídnou jen prázdnou obrazovku a důvody je potřeba vyčíst ve vývojářské konzoli, tak Angular kompilační chyby sám okamžitě nahlásí. I přesto jsem musel nakonec přes network tab hledat v rámci vývojářské konzole problém. Zone.js sice nainstalované v aplikaci je, ale zřejmě kvůli chybnému routování Angular nehledá daný soubor na svém serveru localhost:8081, ale na serveru single-spa localhost:9000.



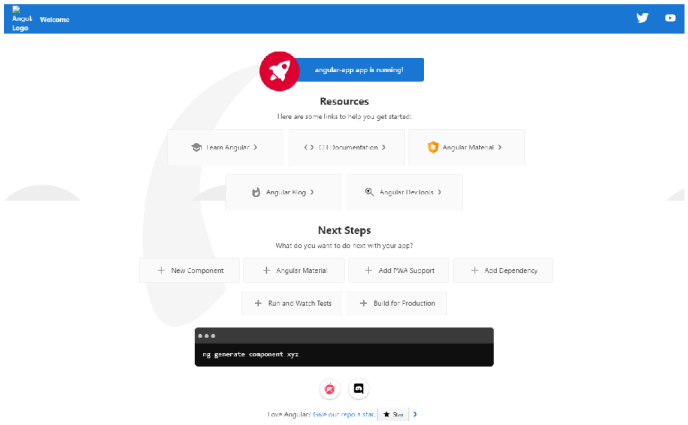
Obrázek 11: Chybné routování Angular

```
<script src="https://unpkg.com/zone.js@0.6.23?main=
    browser"></script>
```

Příklad 10: Externí zdroj chybějícího Zone.js

Po několika pokusech o nastavení odlišného routování jsem navíc přidal víc problémů, než vyřešil. Rozhodl jsem se proto vyhledat online zdroj, kde je knihovna Zone.js trvale dostupná. HTML kód výše, vložený přímo do hlavní stránky generující aplikaci, nakonec problém dokázal vyřešit a Angular aplikace

je již připravena.



Obrázek 12: Uvítací obrazovka Angular aplikace

6.3.2 React.js komponenta

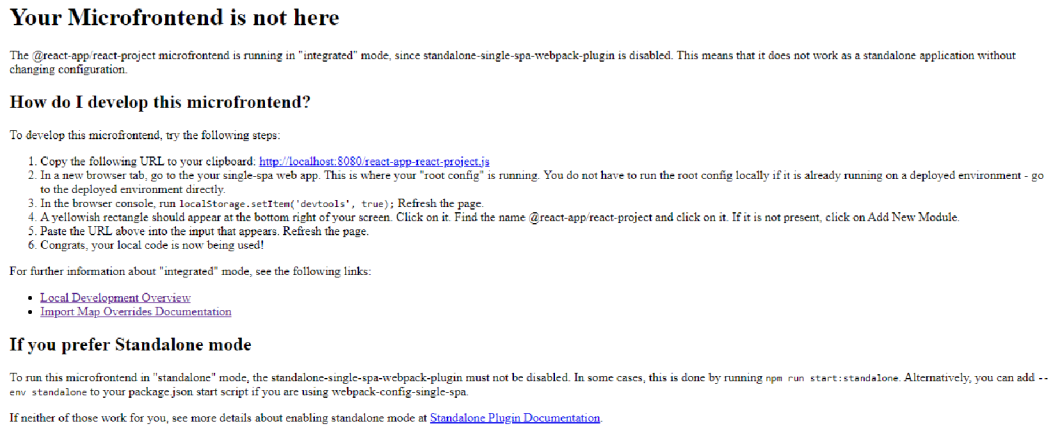
Instalace nového modulu se realizuje stejným způsobem, jako v předchozí kapitole instalace kořenové struktury single-spa aplikace. V prvním kroku ale zvolím namísto single-spa root config možnost single-spa application / parcel. Podle toho se také změní i výstup instalačního skriptu na výběr dostupných frameworků. [35]

```
create-single-spa
? Directory for new project react-module
? Select type to generate single-spa application /
   parcel
? Which framework do you want to use? React
? Which package manager do you want to use? Npm
? Will this project use Typescript? No
? Organization name (can use letters, numbers, dash or
   underscore) bakalarskaprace
```

```
8 ? Project name (can use letters, numbers, dash or
    underscore) reactmodule
```

Příklad 11: Instalace komponenty React.js

Instalace aplikace React.js proběhla bez potřeby stahování dodatečných balíčků. Důvodem je, že s React.js již pracuji, a tak v systému všechny potřebné balíčky mám dávno nainstalované. Rozdíl mezi klasickou instalací React.js a instalací React.js jako single-spa parcel je ten, že v rámci frameworku jsou již kompletně integrované závislosti na samotné single-spa. Defaultní hodnota pro spuštění aplikace je na localhost s portem 8080, která mi vyhovuje. Proto ji v tomto případě nijak neupravuji a rovnou se přesouvám k otestování funkčnosti. Tu lze otestovat jednoduchým příkazem npm start, po kterém se ukáže na adrese localhost:8080 informační stránka komponenty. Ta v daný moment sice hlásí, že je komponenta nedostupná, ale to je v pořádku. Komponenty totiž lze spouštět více metodami, které představím až v následujících kapitolách. Tím je samotná instalace komponenty React.js jako takové kompletní.



Your Microfrontend is not here

The @react-app/react-project microfrontend is running in "integrated" mode, since standalone-single-spa-webpack-plugin is disabled. This means that it does not work as a standalone application without changing configuration.

How do I develop this microfrontend?

To develop this microfrontend, try the following steps:

1. Copy the following URL to your clipboard: http://localhost:8080/react-app-react-project.js
2. In a new browser tab, go to the your single-spa web app. This is where your "root config" is running. You do not have to run the root config locally if it is already running on a deployed environment - go to the deployed environment directly.
3. In the browser console, run `localStorage.setItem('devtools', true);` Refresh the page.
4. A yellowish rectangle should appear at the bottom right of your screen. Click on it. Find the name @react-app/react-project and click on it. If it is not present, click on Add New Module.
5. Paste the URL above into the input that appears. Refresh the page.
6. Congrats, your local code is now being used!

For further information about "integrated" mode, see the following links:

- Local Development Overview
- Import Map Overrides Documentation

If you prefer Standalone mode

To run this microfrontend in "standalone" mode, the standalone-single-spa-webpack-plugin must not be disabled. In some cases, this is done by running `npm run start:standalone`. Alternatively, you can add `--env standalone` to your package.json start script if you are using webpack-config-single-spa.

If neither of those work for you, see more details about enabling standalone mode at Standalone Plugin Documentation.

Obrázek 13: Nainstalovaná React.js komponenta

6.3.3 Vue.js komponenta

Poslední aplikací, kterou instaluji jako mikroservis je aplikace ve Vue.js. Zde se opět vracím k jednodušší instalaci přímo přes příkaz create-single-spa a následné volby podle samotné aplikace. Při instalaci se skript doptá nejen na tradiční parametry, ale dokonce i na verzi Vue.js, kterou si přeji nainstalovat. Zde trochu netradičně nevolím nejaktuálnější verzi, ale naopak Vue 2 namísto Vue 3. Důvodem je, že je Vue 3 ještě poměrně nestabilní a přechod na ní bude díky zajištěné kompatibilitě zcela bezproblémový.[35]

```
create-single-spa
? Directory for new project components/vue-app
? Select type to generate single-spa application /
   parcel
? Which framework do you want to use? Vue
? Organization name (can use letters, numbers, dash or
   underscore) vueapp
? Project name (can use letters, numbers, dash or
   underscore) vueapp
Need to install the following packages:
  @vue/cli@5.0.8
Ok to proceed? (y) y
Vue CLI v5.0.8
? Please pick a preset: Default ([Vue 2] babel, eslint)
```

Příklad 12: Instalační příkazy pro Vue.js

U instalace Vue.js na první pokus narážím na zajímavost, že pro aplikační jméno jako jediný framework nepodporuje camel notation a celý název musí být jen malými písmeny. Na druhý pokus ale instalace probíhá v pořádku, včetně aktualizace balíčků.

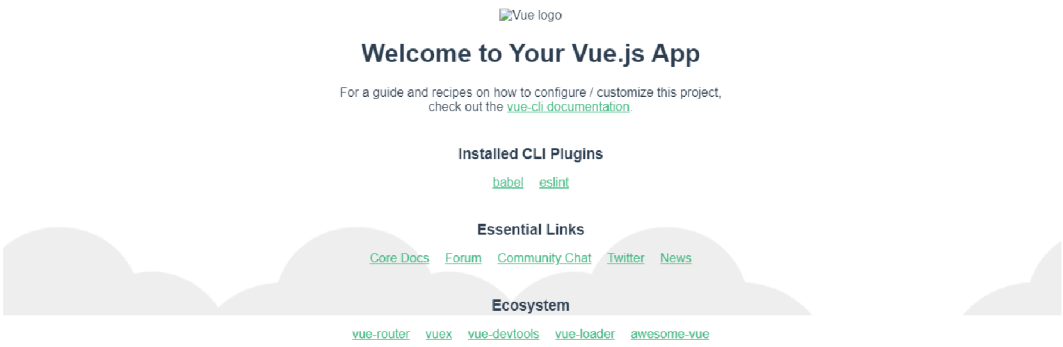
Aplikace je připravená víceméně k okamžitému použití, ale v rámci svého micro-frontend projektu ji musím lehce upravit. Důvodem je, že Vue.js se při

instalaci neptá na port ve kterém bude působit a defaultní port je nastavený na 8080. Ten ale už zabírá React.js komponenta a došlo by ke konfliktu. Nastavení aplikací se skrývá vždy v package.json, které je v kořenové složce každé komponenty. Podrobněji se jim budu teprve věnovat. Pro daný moment, ale potřebuji upravit příkaz z prvních řádků souboru, který spouští Vue.js do následující podoby.

```
1  "scripts": {
2    "serve": "vue-cli-service serve --port 8082",
3    "build": "vue-cli-service build",
4    "lint": "vue-cli-service lint",
5    "serve:standalone": "vue-cli-service serve --mode
     standalone --port 8082"
6  },
```

Příklad 13: Serve skripty s připsaným portem 8082

Následně stačí už jen aplikaci spustit a otestovat. To se podařilo, když aplikace sama nabídla uvítací stránku s úvodními informacemi.



Obrázek 14: Uvítací obrazovka Vue.js aplikace

6.4 Konfigurace mikroservis v single-spa

6.4.1 Struktura routeru v single-spa

Již dříve jsem zmínil, že framework single-spa je ve své podstatě router, který umožňuje směrovat a propojovat aplikace komponent z různých mikroservis. K tomu dokáže nabídnout výbornou utilitu v podobě Layout Engine, s jehož použitím je sestavení Single Page aplikace s užitím micro-frontendu podstatně efektivnější. Pro porozumění funkce routeru přitom potřebujeme znát funkci tří hlavních souborů aplikace. Ty se nacházejí v kořenovém projektu singlepage-app, který jsem si vytvořil první před jednotlivými komponentami, ve složce src.[36]

Jedná se o soubory

- BakalarskaPrace-root-config.js
- index.ejs
- microfrontend-layout.html

6.4.2 index.ejs

Nejdříve představím soubor index.ejs. Jedná se o soubor, který v sobě propojuje HTML kód s JavaScriptem. Při spuštění aplikace je také prvním souborem, který se spouští a od něj se pak odvíjí vše ostatní. Nejpodstatnější součástí souboru je takzvaná Import Maps. Jedná se o funkci single-spa, která umožňuje registrovat a volat jednotlivé komponenty micro-frontendu. Pokud bych v práci nevyužíval Layout Engine, tak bych musel v budoucnosti vkládat router s komponentami do tohoto souboru. V mém případě ale níže zmíněný kód volá další soubory, které jsou pro mě podstatné.[36] Mapa aktuálně vypadá takto:

```
1  <script type="systemjs-importmap">
2    {
```

```
3       "imports": {
4         "single-spa": "https://cdn.jsdelivr.net/npm/
    single-spa@5.9.0/lib/system/single-spa.min.js"
5       }
6     }
7   </script>
8   <% if (isLocal) { %>
9   <script type="systemjs-importmap">
10    {
11      "imports": {
12        "@single-spa/welcome": "https://unpkg.com/single
    -spa-welcome/dist/single-spa-welcome.js",
13        "@BakalarskaPrace/root-config": "//localhost
    :9000/BakalarskaPrace-root-config.js"
14      }
15    }
16  </script>
17  <% } %>
```

Příklad 14: Defaultní Import Maps

V kódu je vidět, že Import Maps pracuje ve dvou módech. První se stará o import komponent v případě, kdy je potřeba, aby aplikace registrovala vzdálené zdroje. Naopak druhá část mapy schovaná pod podmínkou if(isLocal) a registruje komponenty na lokálním serveru. Při nahlédnutí do souboru package.json se spouštěcí konfigurací single-spa je totiž mód zahrnutý v příkazu npm start.[36]

```
1 "start": "webpack serve --port 9000 --env isLocal"
```

Příklad 15: Podoba skriptu pro spouštění v lokálním serveru

V rámci tohoto souboru ale budu registrovat jednotlivé komponenty, které si budu přát zobrazit na stránce. Komponentu připojím na základě dvou hod-

not. První je volací jméno komponenty, které nemusí být shodné s názvem aplikace v ní, ale pro přehlednost kódu je vždy rozumnější používat plošně stejné názvy. Druhou hodnotou pak je adresa vstupního souboru na každém z lokálních serverů, přes které se komponenta spouští.[36]

Pro aplikaci React.js se jedná o soubor react-app-react-project.js na portu 8080, pro Angular se jedná o soubor main.js s portem 8081 a v případě Vue.js je spouštěcím souborem app.js na 8082. Název aplikace jako takové je plošně umístěn vždy na prvním řádku package.json uvnitř souborů každé komponenty. Po přidání závislostí na komponenty tak vypadá nová podoba lokálních Import Maps takto. První položkou je uvítací stránka, kterou ponechávám do finálního refactoringu kódu. Druhá položka je import pro konfigurační soubor, kterému se budu věnovat v další kapitole, a poslední tři položky jsou pak mnou přidané závislosti na jednotlivé komponenty.

```
"imports": {
        "@single-spa/welcome": "https://unpkg.com/single-spa-welcome/dist/single-spa-welcome.js",
        "@BakalarskaPrace/root-config": "//localhost:9000/BakalarskaPrace-root-config.js",
        "@react-app/react-app": "http://localhost:8080/react-app-react-project.js",
        "@angular-app/angular-app": "http://localhost:8081/main.js",
        "@vue-app/vue-app": "http://localhost:8082/js/app.js"
    }
```

Příklad 16: Přidání závislostí do isLocal mapy

Oproti ostatním frameworkům užívajícím DOM v jeho základní podobě má React.js odlišný přístup a často se tak objektová interpretace nazývá React-DOM. Z toho důvodu bude potřeba do Import Maps přidat závislost nejen na

React.js aplikaci, ale také na soubory umožňující funkci s ReactDOM. Soubory s podporou pro ReactDOM neimportuji z lokálního serveru, ale z trvalého online úložiště. Neukládám je tak pod isLocal mapu, ale pod mapu pro vzdálené zdroje.

```
"imports": {
        "single-spa": "https://cdn.jsdelivr.net/npm/
    single-spa@5.9.0/lib/system/single-spa.min.js",
        "react": "https://cdn.jsdelivr.net/npm/react@17
    .0.2/umd/react.production.min.js",
        "react-dom": "https://cdn.jsdelivr.net/npm/react
    -dom@17.0.2/umd/react-dom.production.min.js"
      }
```

Příklad 17: Přidání závislostí do online mapy

Po přidání všech závislostí soubor ukládám a mohu se přesunout k další důležité části routeru. A to přímo k jeho konfiguračnímu souboru, který je importován skriptem níže.

```
<script>
  System.import('@BakalarskaPrace/root-config');
</script>
```

Příklad 18: Import konfiguračního souboru aplikace single-spa

6.4.3 BakalarskaPrace-root-config.js

V tomto souboru najdeme víceméně celou logiku, která se odehrává na pozadí routeru v rámci single-spa. Soubor se stará o založení a funkci celého routeru. Kód vypadá takto a pro potřeby své aplikace do něj nebudu zasahovat.

```
import { registerApplication, start } from "single-spa";
import {
  constructApplications,
```

```
  constructRoutes,
  constructLayoutEngine,
} from "single-spa-layout";
import microfrontendLayout from "./microfrontend-layout.
    html";

const routes = constructRoutes(microfrontendLayout);
const applications = constructApplications({
  routes,
  loadApp({ name }) {
    return System.import(name);
  },
});
const layoutEngine = constructLayoutEngine({ routes,
    applications });

applications.forEach(registerApplication);
layoutEngine.activate();
start();
```

Příklad 19: Layout Engine v konfiguračním souboru

Tento kód ve své podstatě plní jednoduchou funkci, ale plní jí velice účinně. V úvodu importuje poslední důležitý soubor routeru. V něm pak identifikuje jednotlivé aplikace a cesty k nim v rámci Import Map z předešlého souboru, a následně pak navrací import aplikace v podobě jejího jména, což bude po celou dobu sloužit jako identifikátor.

6.4.4 microfrontend-layout.html

V posledním souboru se už nachází samotný router, který je díky oddělení veškeré logiky a mapování velice přehledný. I to je důvod, proč jsem se rozhodl

pro užívání Layout Engine. V prvním zmíněném souboru se mapují jednotlivé soubory, v druhém pak lze pracovat s konfigurací jednotlivých importů i celého routeru, a v posledním HTML dokumentu se již píše čistý kód a vkládají v rámci routeru komponenty. Soubor nyní vypadá takto.

```
<single-spa-router>
  <main>
    <route default>
    <application name="@single-spa/welcome"></
    application>
    </route>
  </main>
</single-spa-router>
```

Příklad 20: Základní podoba routeru

Vidíme zde sekci route default. Default pojmenování značí ty komponenty, které se zobrazují již při prvním načtení stránky. Z teoretické části už moc dobře víme, že podstatou Single Page aplikací je načtení celé aplikace naráz. To znamená, že v tomto souboru mohou být uvedeny i komponenty mimo default router, které tak jsou načteny a připraveny, ale jejich view se generuje později. Jednotlivé aplikace jsou zde identifikovány pomocí názvu, který jsem již vyplnil do Import Maps v rámci index.ejs. Do routeru tak přidám všechny tři aplikace, protože chci mít na stránce aktivní všechny tři komponenty naráz. Finální router vypadá takto:

```
<single-spa-router>
  <main>
    <route default>
      <application name="@react-app/react-app"></
    application>
      <application name="@vue-app/vue-app"></application
    >
```

```
6       <application name="@angular-app/angular-app"></
    application>
7     </route>
8   </main>
9 </single-spa-router>
```

Příklad 21: Finální podoba routeru

6.5 První spuštění stránky

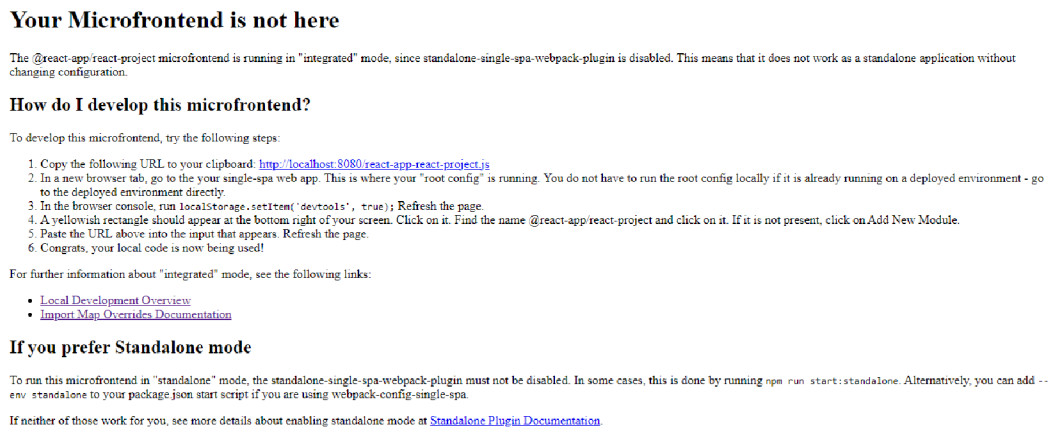
6.5.1 Skripty start a start:standalone

Před samotným spuštěním stránky je potřeba zjistit metody spouštění, které se nabízejí. Jednou ze zajímavých funkcí single-spa je možnost odlišného spouštění jednotlivých komponent. Všechny skripty jsou uloženy vždy v package.json dané komponenty a lze si všimnout, že jsou k dispozici skripty jak pro klasické spuštění, tak i pro spuštění s parametrem standalone.

```
1     "start": "webpack serve",
2     "start:standalone": "webpack serve --env standalone"
    ,
```

Příklad 22: Možnosti spouštění v package.json

Mezi nimi je zásadní rozdíl právě ve vazbě k samotné single-spa. Pokud dojde ke spuštění přes start:standalone, tak se pouští aplikace samostatně. Lze na ni v rámci daného odkazu přistoupit, živě na ní pracovat, pozorovat změny a dělat vše jako kdyby nikdy nebyla se single-spa propojena. V ten moment i kdyby byla aplikace připojena na micro-frontend, pracuje sama za sebe. Zcela naopak pokud dojde ke klasickému spuštění bez parametru –env standalone, tak se aplikace stává samostatně nepřístupnou, ale v rámci celku jako komponenty v Layout Engine single-spa je viditelná a funkční. O tom nás informuje i automatická obrazovka.



Your Microfrontend is not here

The @react-app/react-project microfrontend is running in "integrated" mode, since standalone-single-spa-webpack-plugin is disabled. This means that it does not work as a standalone application without changing configuration.

How do I develop this microfrontend?

To develop this microfrontend, try the following steps:

1. Copy the following URL to your clipboard: http://localhost:8080/react-app-react-project.js
2. In a new browser tab, go to the your single-spa web app. This is where your "root config" is running. You do not have to run the root config locally if it is already running on a deployed environment - go to the deployed environment directly.
3. In the browser console, run `localStorage.setItem('devtools', true);` Refresh the page.
4. A yellowish rectangle should appear at the bottom right of your screen. Click on it. Find the name @react-app/react-project and click on it. If it is not present, click on Add New Module.
5. Paste the URL above into the input that appears. Refresh the page.
6. Congrats, your local code is now being used!

For further information about "integrated" mode, see the following links:

- Local Development Overview
- Import Map Overrides Documentation

If you prefer Standalone mode

To run this microfrontend in "standalone" mode, the standalone-single-spa-webpack-plugin must not be disabled. In some cases, this is done by running `npm run start:standalone`. Alternatively, you can add `--env standalone` to your package.json start script if you are using webpack-config-single-spa.

If neither of those work for you, see more details about enabling standalone mode at Standalone Plugin Documentation.

Obrázek 15: Obrazovka při vstupu do komponenty bez standalone módu

6.5.2 Spuštění komponent a single-spa

Samostatné spouštění komponent a single-spa může probíhat několika způsoby. Díky tomu že užívám Visual Studio Code mám k dispozici vestavěný terminál. Ten pokud spustím ve složce s danou aplikací, tak přes npm a ng mohu spouštět skripty obsažené v package.json. Stejně tak ale lze přistoupit do samotného package.json a prostředí samo nabízí možnost spuštění skriptu kliknutím myší. Jako první tak nejdříve startuji lokální servery pro všechny tři komponenty. Jako poslední pak startuji server pro samotnou single-spa.

Při spouštění se mohou vyskytnout potíže, pokud není některý z balíčků kompletní. To například bylo problémem v případě instalace Angular, kde chyběla část aplikace environment. Aby vše fungovalo správně, tak musí být všechny komponenty spuštěny v režimu start či serve bez parametru standalone.

```
1 React.js: npm start
2 Angular : npm run serve:single-spa:angular-app
3 Vue.js  : npm run serve
4 single-spa: npm start
```

Příklad 23: Spouštěcí příkazy na lokálním serveru

6.5.3 Import Map v prohlížeči

Po spuštění všech komponent přecházím na adresu http://localhost:9000/, kde se na lokálním serveru ocitá samotná aplikace single-spa. První pohled na aplikaci mi nabízí kombinaci všech uvítacích oken aplikací. Mimo ověření pohledem ale nabízí single-spa další možnost, jak zjistit, které aplikace jsou úspěšně připojeny. Pomocí příkazu níže lze vygenerovat tabulku přímo v prohlížeči:

```
1 run local.Storage.setItem('devtools', true);
```

Příklad 24: Spuštění přehledu importů v prohlížeči

Příkaz zadávám přímo do konzole v prohlížeči. Přes F12 tedy zapínám konzoli, vkládám příkaz a posléze obnovuji okno aplikace. V pravém dolním rohu se ukáže oranžová ikona složených závorek se třemi tečkami. Při kliknutí nabízí přímo v prohlížeči přehled aplikací, které jsou importovány. Jsou zde nejen komponenty, ale i utility jako je ReactDOM v importu online. U každé z nich je uvedený název a také adresa, ze které se komponenta připojuje.

Obrázek 16: Viditelné komponenty v prohlížečovém režimu

Okno nabízí celou řadu možností, jak s komponentami pracovat a testovat jejich připojení bez potřeby zasahovat nutně do Import Maps. Změny, které v tomto módu uživatel udělá jsou platné pouze po dobu funkce stránky, nepřepisují zdrojový kód. Rozhodně se mi ale tento mód osvědčil, když jsem poprvé testoval připojování komponent na začátku zkušeností se single-spa. Mezi možnostmi je přidání nového modulu, přidání celé mapy nebo přepsání adresy, ze které se vybraný modul načítá.

Díky kontrole připojených modulů přímo v prohlížeči, prohlédnutí importovaných souborů přes network tab ve vývojářských nástrojích, a také díky

vizuálnímu zobrazení tak už vím, že se mi úspěšně podařilo realizovat micro-frontendový přístup na Single Page aplikaci. V následujících kapitolách se budu věnovat tvorbě obsahu v jednotlivých komponentách.

6.6 Angular komponenta

6.6.1 Úprava vstupních souborů

V případě Angular komponenty jsem aplikoval instalaci samostatného frameworku, do kterého jsem následně přidal závislosti na single-spa router. V adresáři s Angular aplikací tak v tento moment mám složku src, kde se ocitá samotná aplikace. Stejně tak ale v této složce lze najít adresář single-spa. V tomto adresáři se ale vyskytuje jenom samotná konfigurace, která je již nastavená od instalace pomocí npm a není potřeba do ní v tento okamžik nijak zasahovat. Mimo zmíněných adresářů je zde ještě adresář app, ve kterém budou data budoucí aplikace a soubory index.html a main.ts. Koncovka souboru .ts se připojuje k souborům psaným v jazyce TypeScript, který Angular užívá namísto obyčejného JavaScriptu.

```
1 import 'zone.js';
2 import 'zone.js/dist/long-stack-trace-zone.js';
3 import { platformBrowserDynamic } from '@angular/
    platform-browser-dynamic';
4 import { AppModule } from './app/app.module';
5
6 platformBrowserDynamic().bootstrapModule(AppModule)
7   .catch(err => console.error(err));
```

Příklad 25: Nastavení main.ts v Angular

Na ukázce zdrojového kódu výše je vidět podoba souboru tak, jak je po přidání důležitých částí. V rámci importů se přidává do main.ts modul aplikace, který je již vytvořený z instalace a je na něm čistě jen uvítací obrazovka

Angular. V kódu níže pak lze vidět mechanismus pro zachytávání a vypisování chyb do konzole. První dva řádky pak importují potřebné soubory pro zone.js, které v původní instalaci chyběly.

```
<app-root></app-root>
```

Příklad 26: Vstupní bod Angular aplikace

V souboru index.html stačí jediný řádek. Tento řádek je kořenovým místem pro vygenerování aplikace jako takové. Pokud by aplikace byla přejmenována, tak by se musela změnit i podoba těchto tagů tak, aby vždy aplikace našla vstupní bod pro vykreslení.

6.6.2 Adresář app

Jakmile je vše nastaveno, tak je možné se přesunout do adresáře app, kde se již bude ocitat samotná aplikace. V rámci Angular komponenty vytvořím dynamický blok s obsahem věnujícím se úvodu do Single Page aplikací a micro-frontend. V této složce se zároveň odehrává poměrně složitá logika Angular, jak se dostat k samotnému psaní kódu. Po pochopení je ale nutné říct, že jsem se díky této logice lépe vyznal ve stavbě samotné aplikace. Dále se tak budu věnovat konkrétní souborům:

- app.module.ts
- app.component.ts
- app.component.html

Ve složce se nachází i další adresáře, které mohou být užitečné pro Angular aplikace. Jsou to soubory app.component.css nebo app-routing.module.ts. V obou případech ale nevyužiji tyto soubory. Routování v Angular mezi moduly nebudu užívat, protože o to se již stará single-spa. V případě kaskádových stylů pak design budu psát do kořenové single-spa aplikace, namísto do jednotlivých komponent.

6.6.3 App-module.ts

V tomto souboru dochází především k importu všech důležitých součástí, které Angular aplikace bude využívat. Kód po úpravách v tento moment vypadá takto:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Příklad 27: Nastavení modulu v Angular

V horní části dochází k importování důležitých závislostí pro aplikace. NgModule umožňuje realizovat reaktivitu, kterou použiji. BrowserModule se užívá při vypisování dat do prohlížeče. Routování se stará o životní cykly a změny dat a samotný poslední import pak odkazuje na data aplikace jako takové.

6.6.4 App.component.ts

V této části již dochází ke skládání samotné komponenty. Ještě zde nebudu psát konkrétní funkcionality nebo šablony, ale jednoduše poskládám vše dohromady a dám komponentně informaci i tom, kde se má vypisovat.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-app';
}
```

Příklad 28: Konstruktor Angular komponenty

V první části dochází k defaultnímu importu funkce komponent z jádra Angular. Následně se komponenta spojuje z důležitých částí. V rámci konstruktoru komponenty můžeme najít tři parametry. V prvním se propojuje komponenta s tagy <app-root><app-root/> v index.html. Druhým parametrem je odkaz na šablonu, kde bude uložený HTML šablona a také veškeré dynamické funkce. V posledním parametru pak je uložený defaultní styl komponenty. Následně se celá komponenta exportuje pod názvem AppComponent.

6.6.5 App.component.html

V této části aplikace se již bude psát samotná šablona. K vytvoření dynamického textového bloku přitom využiji integrovanou funkci Angular s názvem ngSwitch. Jedná se o způsob directivy v Angular, která se používá k přičleňováním dat ke konkrétním částem šablony. Z logického pohledu je přitom stejná, jaké kterýkoliv jiný switch. To znamená, že si určím místo, kam budu chtít

vkládat data v jednotlivých případech a následně zde jednoznačně pojmenuji každý případ. O směrování mezi jednotlivými případy se mohou starat různé formulářové prvky. I z designových důvodů zde zvolím několik odkazů.

```
<a href="#" (click)="selection.value = 'spa'">O Single
    Page Aplikacich</a>
<a href="#" (click)="selection.value = 'mf'">O Micro-
    Frontend</a>
```

Příklad 29: Odkazy s listenery v Angular

V ukázce zdrojového kódu jsou dva z několika buttonů, které jsem do aplikace přidal. Uvnitř odkazu je sledování eventu (click), které čeká na stisknutí odkazu. V okamžiku, kdy dojde ke stisku, tak se proměnná selection.value změní na jednu ze zkratek textových bloků.

```
<div [ngSwitch]="selection.value">
</div>
```

Příklad 30: Konstrukce funkce ngSwitch

V další částí kódu pak tvořím HTML rámec, do kterého se budou vkládat data podle aktuální hodnoty. Zde implementuji funkci Angular ngSwitch, která se dynamicky bude měnit podle kliknutí na daný odkaz.

```
<div *ngSwitchDefault>
<p>text</p>
</div>
<div *ngSwitchCase="'spa'">
<p>text</p>
</div>
```

Příklad 31: Hodnoty funkce ngSwitch

V další části kódu pak už jsou konkrétní případy, které funkce ngSwitch vyvolává. V prvním rámci je vidět případ ngSwitchDefault. Jedná se o případ, kdy ještě není žádná hodnota určena, takže při načtení samotné aplikace.

Nejedná se o povinnou součást ngSwitch. Podle druhého rámce již píši div s parametrem ngSwitchCase, který se řídí podle rámce, ve kterém je zabalený. Pokud se v něm změní proměnná selection.value, tak se okamžitě vykreslí ten div, který je identifikován danou zkratkou. Uvnitř každého rámce mohou být libovolné HTML tagy, nebo také import dat z databází. Pro potřebu své aplikace jsem ale vytvořil statické texty, které se mění dynamicky na stránce podle rámce, ve kterém jsou uloženy.

6.7 React.js komponenta

V případě komponenty v React.js jsem se rozhodl demonstrovat další výhodu, kterou micro-frontend nabízí. Napojení zcela nezávislých služeb může mít tu výhodu, že se na projektu nebudou křížit prostředky a konfigurační soubory pro napojení k různým API.

API znamená rozhraní aplikace, které umožňuje na dálku komunikovat s jejími funkcemi, přijímat výstupy, a naopak odesílat data. Standardně se přitom může stát, že některá API již nebude vyhovovat business potřebám daného projektu, nebo zkrátka ukončí podporu a funkci. V takovém případě je u monolitické aplikace často problém ji nahradit, protože se závislosti a zdrojové kódy mohou ocitat na velkém množství míst v aplikaci.

V případě mojí aplikace využiji na napojení k API React.js komponentu. Veškeré závislosti a kódy potřebné ke komunikaci tak budou pouze v rámci dané komponenty a jejich změna nebude mít na aplikaci jako celek vliv. Pro demonstraci napojení k API přitom využiji rozhraní jedné z aktuálně nejfrekventovanějších online služeb, OpenAI API ke komunikaci s ChatGPT.

6.7.1 Instalace REST API serveru a OpenAI

Aby bylo možné komunikovat přes API, tak je potřeba zajistit možnost komunikace s REST API pomocí HTTP requestů. Pro svůj projekt přitom volím technologii Express.js, která umožňuje v rámci backendu vytvořit podmínky

pro aplikace komunikující s REST API v součinnosti s Node.js, na kterém mám postavenou celou aplikaci.

```
npm install express body-parser cors
```

Příklad 32: Instalační příkaz Express.js

Pomocí jednoduchého příkazu dojde k instalaci všeho, co je pro provoz Express.js serveru potřebné. Přitom se nemusí instalovat nové soubory, protože všechna komunikace prochází pouze skrze instalaci závislostí do současné aplikace. Pro budoucí provoz serveru potřebuji vytvořit konfigurační soubor, odkud se bude server spouštět. Ten umísťuji do kořenové složky React.js aplikace, tedy do složky src pod jménem server.js, kde vytvářím celou aplikaci. Důvodem je jednodušší import souborů.

Dříve, než se přesunu k dalšímu kroku, tak potřebuji ještě instalovat závislosti umožňující komunikaci s aplikacemi OpenAI. Společnost přitom nabízí pro většinu populárních správců balíčků možnost jednoduché instalace jediným příkazem.

```
npm install openai
```

Příklad 33: Instalační příkaz OpenAI závislostí

6.7.2 Konfigurace server.js

V doposud prázdném souboru server.js se bude odehrávat veškerá logika, která je na pozadí komunikace s OpenAI. V prvním kroku tak nejdříve vytvořím proměnné, ke kterým přiřadím závislosti na již nainstalované služby.

```
const express = require("express");
const cors = require("cors");
const bodyParser = require("body-parser");
const configuration = require("openiai");
const api = require("openai");
```

Příklad 34: Závislosti v server.js

Po přidání všech závislostí se přesouvám k samotnému vytvoření spojení. V prvním kroku vytvořím instanci pro konfiguraci, které předám tajný klíč, který jsem si již vygeneroval ve svém účtu u společnosti OpenAI. Po předání tajného klíče pak tvořím instanci samotné aplikace s tím, že jako parametr předávám konfigurační proměnnou.

```
const configuration = new configuration({
    apiKey: "sk-
   sGeBDWZdPQB4q46oqm9DT3BlbkFJV0xRN5UNF6IXbf43UlX3"
});

const openai = new api(configuration);
```

Příklad 35: Vytvoření aplikace v server.js

Nyní už mám importované všechny potřebné závislosti a zároveň už mám připravenou instanci pro ověření přístupu přes apiKey. Na následujících řádcích tak tvořím připravuji samotný server. První tři řádky implementují Express.js a zároveň pro něj přidávají využití prostředků body-parser z JSON a cors. Body-parser po mě je podstatný proto, že budu potřebovat z těla requestu, který je posílaný v podobě JSON vybrat data co potřebuji. Cors je pak takzvaný middleware, který využívá ke zprostředkování komunikace Express.js.

Posledním krokem konfigurace serveru jako takového je vytvoření takzvaného endpointu. Endpoint je konečným místem, odkud a kam se odehrává komunikace se vzdálenou aplikací. Ten předává hodnoty jak v případě requestu, tak i respondu a je nastavený asynchronně, protože se komunikace tam a zpět může odehrávat v časových rozestupech podle složitosti requestu. Poslední částí kódu z pohledu konfigurace Express.js je pak stanovení provozního portu. Ten je standardně na 8080, ale pro moji aplikaci je nejbližší volný HTTP port 8084.

```
const app = express();
app.use(bodyParser.json());
```

```
app.use(cors())

app.post("/chat", async (req, res) => {
  const { prompt } = req.body;

  // Konfigurace komunikace s OpenAI
});
const port = 8084;
```

Příklad 36: Proměnná značící konkrétní port

6.7.3 Konfigurace komunikce s OpenAI

Poslední část skriptu pak využívá vlastnosti OpenAI. Zde jako ve velké části aplikace syntaxe značně závisí na použitých závislostech, které umožňují standardizované psaní kódu. Komunikace s OpenAI přes API nabízí mnoho parametrů, přičemž některé stojí za představení.

- model - tato vlastnost dává na výběr z mnoha modelů, které jsou k dispozici. Všiml jsem si, že různé modely se opravdu i odlišně chovají a tak stojí za vyzkoušení.
- prompt - je označení hodnoty, pod kterou se předávají data v podobě objektu i řetězce.
- max_tokens - maximální délka odpovědi ve znacích. Defaultní je 16, přičemž většina modelů podporuje 2048 a některé dokonce 4096 znaků.
- temperature - hodnotou od 0 do 2 včetně desetin oddělených tečkou lze určit, jak moc bude model zaměřovat svoji odpověď "doslovně"v závislosti na otázce.

```
  const completion = await openai.createCompletion({
    model: "text-davinci-003",
```

```
    max_tokens: 1024,
    prompt: prompt,
    temperature: 1,
  });
  res.send(completion.data.choices[0].text);
```

Příklad 37: Závislosti v server.js

6.7.4 Nastavení formuláře v React.js

Pro dokončení kompletní funkcionality mi chybí ještě jedna podstatná část. Mám již připravený HTTP server a závislosti na samotnou aplikaci OpenAI. Chybí stále ale nástroj, který v prohlížeči dokáže zpracovávat HTTP requesty, aby s nimi mohl pracovat React.js v živém okně. Pro tento účel jsem vybral populární Axios. Umožňuje jednoduchou a rychlou komunikaci, má přehledné nastavení a instalace probíhá rovněž přes správce balíčků npm. Mimo toho je Axios skvělý nástroj pro práci s JSON daty. Instalace i zde probíhá jedním příkazem.

```
npm install axios
```

Příklad 38: Instalační příkaz pro Axios

Instalace netvoří další soubory přímo ve složce, ale místo toho instaluje podstatné závislosti. V kořenovém React.js souboru react-app-react-project.js tak nejdříve naimportuji Axios. Rovnou naimportuji také užitečný nástroj useState.

```
import { useState } from "react";
import axios from "axios";
```

Příklad 39: Import služby Axios a React.js useState

V této komponentě budu pracovat s ReactHooks. Samotný useState je jednou z možných implementací této funkce. Ve své podstatě useState znamená, že ke konkrétnímu prvku na stránce připojím "hák", pomocí kterého sleduji

změny jeho stavů. Jeden useState tak bude hlídat vstup ve formuláři, kterým se odesílají data. Druhý pak bude vkládat na stránku odpověď, která se ze serveru vrátí. Proměnné pro tyto funkce vypadají takto.

```
const [prompt, setPrompt] = useState("");
const [response, setResponse] = useState("");
```

Příklad 40: Přiřazení hodnot k useState

Následně potřebuji nastavit samotné chování Axios, které je již nainstalované. HandleSubmit v tomto případě hlídá odeslání formuláře. Následná metoda preventDefault se stará o to, aby formulář nahradil defaultní chování tím, které využiji ve skriptu.

Samotný Axios pak v první řadě směřuji do endpointu, který je nastavený i v souboru server.js. Tedy na lokální server s portem 8084. Přes setResponse dojde k uchycení odpovědi od API. Celý proces je přitom umístěný v then, catch konstrukci, která umožňuje předejít pádu aplikace zachycením chyby a případným vypsáním do konzole.

```
const handleSubmit = (e) => {
    e.preventDefault();

    axios
      .post("http://localhost:8084/chat", { prompt })
      .then((res) => {
        setResponse(res.data);
      })
      .catch((err) => {
        console.error(err);
});
```

Příklad 41: Nastavení nestandardního chování formuláře

Poslední částí stránky je pak samotný formulář. Funkce onSubmit automaticky volá chování, které obsahuje odeslání přes Axios. V hodnotě value

sleduji obsah textového pole, kam uživatel bude psát své dotazy. Po následném odeslání formuláře pak asynchronně v tagu pro odstavec čeká {response} na vypsání odpovědi od API.

```
return (
  <div>
    <form onSubmit={handleSubmit}>
      <input type="text" value={prompt} onChange={(e)
 => setPrompt(e.target.value)} />
      <button type="submit">Zeptej se!</button>
    </form>
    <p>{response}</p>
    </div>
);
```

Příklad 42: Odeslání requestu a přijetí odpovědi

6.8 Vue.js komponenta

Komponentu ve frameworku Vue.js jsem se rozhodl využít na přehlednou prezentaci technologií, které za celým tímto projektem stojí. Mojí představou je tak komponenta s dynamickým vkládáním informací o konkrétním frameworku podle toho, jaký si uživatel vybere jako první. V případě této komponenty tak nebudu již potřebovat instalace dodatečných balíčků a půjde více o kódování, nežli o konfiguraci serverových služeb.

6.8.1 Konfigurace základního App.vue

Ačkoliv je Vue.js sama o sobě komponentou aplikace, tak se bude stejná hierarchie odehrávat také uvnitř komponenty jako takové. Složkou odkud se načítají veškerá data je /src. Uvnitř složky pak najdeme výchozí soubor App.vue, odkud se již větví celá aplikace. Ta již má přednastavenou uvítací stránku kterou mažu a začínám psát vlastní strukturu kódu.

```
<template>
  <div id="app">
    <MenuLogoes />
    </div>
</template>
```

Příklad 43: Podoba šablony v App.vue

V první části kódu aplikace nejdříve tvořím základní šablonu. Veškeré HTML elementy stránky musí být uzavřeny uvnitř kořenových tagů nebo jako v mém případě se dají použít <template></template>. Mezi nimi jsem pak ponechal základně stylizovaný div aplikace a element <MenuLogoes /> značí místo, kde se bude vykreslovat komponenta.

```
<script>
import MenuLogoes from './components/MenuLogoes.vue'

export default {
  name: 'App',
  components: {
    MenuLogoes
  }
}
</script>
```

Příklad 44: Import komponenty v App.vue

Pod samotným HTML kódem pak už přichází na řadu sekce script, kde se odehrává kompletní logika kolem vykreslování aplikace. V prvním řádku přes import pod konkrétním jménem importuji soubor s komponentou MenuLogoes.vue, kterou umišťuji do defaultní složky components. Tato celá Vue.js komponenta tak může být opět tvořena jednou komponentou, nebo i desítkami jednotlivých komponent. Striktní oddělení od kořenového souboru přitom zásadně zvyšuje přehlednost celé aplikace.

Funkce export default pak automaticky bez potřeby volání vykresluje samotnou aplikaci. Zde vidíme jméno pod kterou se bude aplikace vykreslovat a v sekci components jsem zapsal jmenný název importovaných komponent. Dále už je součástí tohoto souboru jenom základní inline kasádový styl, který ale není v tento moment nijak zajímavý ani podstatný.

6.8.2 Komponenta MenuLogoes.vue

Nejdříve přecházím do složky src/components a zde tvořím novou komponentu s názvem MenuLogoes.vue. V současnou chvíli přitom nejde aplikace renderovat, protože se Vue.js samo dotazuje po doposud neexistující komponentě importované v App.vue.

V rámci této komponenty přitom využiji tu nejsilnější zbraň, kterou dokáže Vue.js nabídnout. A tou je skvělý systém reaktivity s prvky na webové stránce, což využiji k vykreslování informací o jednotlivých komponentách. První částí bude i v tomto případě nejdříve psaní samotné template, kterou ale v rámci ukázky zkrátím.

```
1  <template>
2      <div class="component_wrapper">
3          <div class="menu_wrapper">
4              <div class="content-row">
5                  <h2> Pouzite technologie </h2>
6                  <p class="technology_description"></p>
7              </div>
8              <div class="content-row">
9                  <a href="#" @click="nodeDetail" class="
       technology_item"><img src="../assets/node.png" class=
       "technology_picture"> Node.js </a>
10                 ...
11                 <a href="#" @click="vueDetail" class="
```

```
    technology_item"><img src="../assets/vue.png" class="
    technology_picture"> Vue.js </a>
12              </div>
13              <div class="technology_detail">
14                  <h3>{{ techHeader }}</h3>
15                  <p class="technology_desc">
16                      {{ techDescription }}
17                  </p>
18                  <p v-html="techLinks"></p>
19              </div>
20          </div>
21
22      </div>
23  </template>
```

Příklad 45: Šablona komponenty MenuLogoes

V HTML kódu výše se ocitá základní HTML šablona, která ale již obsahuje důležité prvky pro její budoucí funkci. Vzhledem k tomu, že jde o obsahově složitější komponentu než předchozí React.js, tak ji rovnou v průběhu stylizuji, abych mohl kontrolovat její funkci. Zmíněné důležité prvky jsou elementy ve dvojitých složených závorkách a HTML elementy, které obsahují parametr v-html. Obě formy zápisu jsou reaktivní částí šablony. Ve své podstatě se jedná o obdobu odkazů, které jsou uvedeny v samotné šabloně. Dokud ale nedojde k volání metody zodpovědné za jejich naplnění, tak stále bez obsahu čekají.

Zmíněné metody jsou vidět v jednotlivých odkazech, kde je operátor @click. Zavináč je pouze zkráceným zápisem příkazu, který by jinak vypadal takto : "v-on:click". Jedná se o funkci zvanou Event Handling, která slouží k pozorování událostí na stránce a k reakcím na ně. V mém případě hlídám událost v podobě kliknutí na odkaz, přičemž jako parametr je zde volána funkce. Dané funkce přitom představím v následující sekci skript.

```
export default {
  name: 'MenuLogoes',
  data() {
  return {
    techHeader: '',
    techDescription: '',
    techLinks: ''
  }
},
methods: {
  nodeDetail() {
    this.techHeader = 'Node.js',
    this.techDescription = 'Popis',
    this.techLinks = '<a href="https://nodejs.org/en"
   target="_blank" class="technology_link"> Oficialni
   web </a><a href="https://nodejs.org/en/docs" target="
   _blank" class="technology_link"> Dokumentace </a>'
  },
...
  reactDetail() {
    this.techHeader = 'React.js',
    this.techDescription = 'Popis',
    this.techLinks = '<a href="#" target="_blank" class
   ="technology_link"> Oficialni web </a><a href="#"
   target="_blank" class="technology_link"> Dokumentace
   </a>'
  }
}
}
```

```
24 </script>
```

Příklad 46: Funkcionalita komponenty MenuLogoes

Ve skriptech této komponenty již není žádný import, protože je samotná komponenta již finální a nebude se dále větvit. Opět zde píši jméno komponenty, které musí být shodné s volacím jménem v App.vue. Oproti App.vue zde ale připisuji v podobě metody data reaktivní komponenty, které se v tento moment navracejí prázdné. Níže pak píši v sekci methods jednotlivé metody, které jsou přiřazeny k jednotlivým odkazům a v moment události @click se metody zavolají.

Vnitřek metod zde za pomoci operátoru this přiřazují do reaktivních částí šablony jednotlivé hodnoty. Zajímavostí, která stojí za zmínku, je ale dvojí možnost zápisu. V šabloně totiž mám zapsané reaktivní hodnoty pomocí zápisu hodnota , ale také pomocí <p v-html="hodnota»</p>. Mezi nimi je zásadní rozdíl. Pokud totiž předávám data do první varianty zápisu, tak i když předám správně zapsaný HTML element, vygeneruje se pouze jako plain text. V případě, kdy ale dosazuji hodnoty přes v-html funguje reaktivita jinak a pokud předám HTML element, tak se místo textu opravdu vykreslí HTML element. Dvojí možnosti reaktivity tady tak užívám proto, že v případě názvu a popisu mi stačí pouze předat text, ale u odkazů už si přeji vykreslit plnohodnotné hyperlinky.

6.9 Design aplikace

V tento moment již mám kompletní Single Page aplikaci i se všemi komponenty. Prozatím ale nemá žádný jednotný design a má podobu rozmístěných HTML elementů a formulářů. Pro tvorbu stylů jsem zvolil běžně užívaný CSS framework Bootstrap, který doplním vlastními styly.

6.9.1 Instalace Bootstrap

V rámci webových aplikací za použití frontend frameworků může být instalace Bootstrap zásadně odlišná od klasické statické webové stránky. Pomocí správce balíčků se dá instalovat Bootstrap jako rozšíření každé aplikace zvlášť. Pro potřeby své aplikace ale toto řešení rozhodně nezvolím. Důvodem je, že jsou veškeré komponenty přes import mapy spojeny pod single-spa jako celek. To mi tak dává šanci pracovat s Bootstrap pouze na úrovni single-spa. Přitom v každé komponentě budu používat třídy kaskádových stylů, které ale nebudou přímo v komponentách, ale právě v single-spa.

Instalace Bootstrap je tak v tomto případě velice jednoduchá. Pro aktuální verzi jsem zvolil přímé vložení kaskádových stylů přes HTML prvek link. Níže je tak vidět kód, který importuje do aplikace vše potřebné k užívání kaskádových stylů Bootstrap v projektu.

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/
    npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
    integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/
    j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="
    anonymous">
```

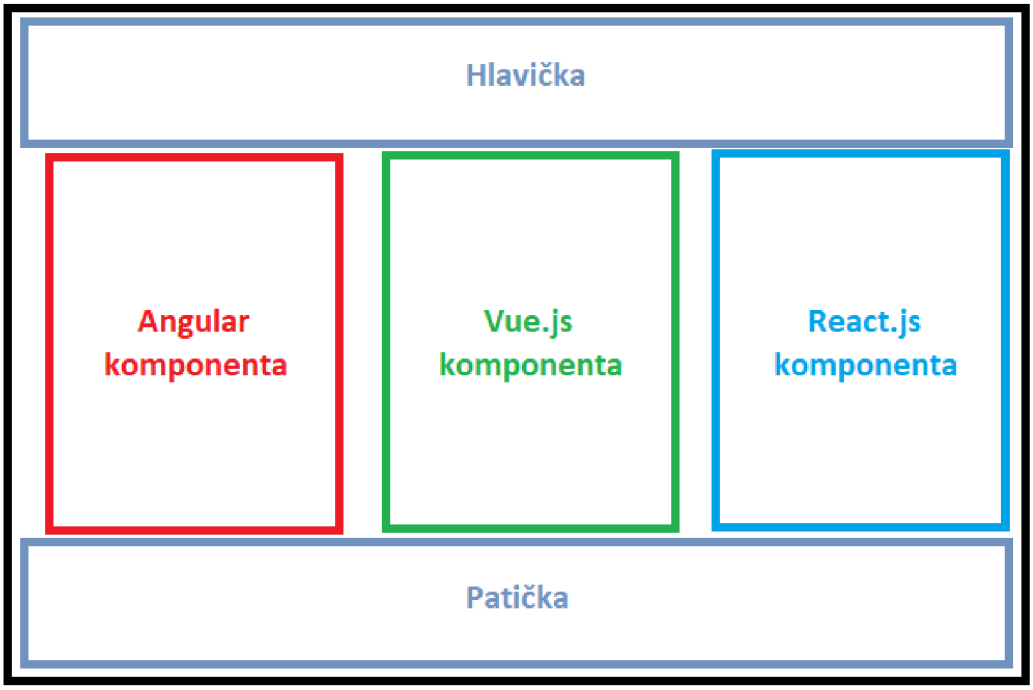
Příklad 47: Instalace Bootstrap do single-spa

Tento kód přitom vkládám do kořenového souboru single-spa, do souboru index.ejs. Dříve jsem již v práci zmínil, že jsou právě zde uloženy import mapy a jedná se tak o nejdůležitější část celého routeru, který spojuje komponenty do funkčního celku.

6.9.2 Návrh designu

Bootstrap je frameworkem, který umožňuje díky skvělému grid a flex systému stylů tvorbu webových stránek s přístupem mobile first. Tento přístup znamená preferování obsahu pro mobilní zařízení. Přitom dynamické break pointy umožňují, aby jediný styl mohl fungovat na různých velikostech monitorů, i na

mobilních zařízeních zároveň. Na obrázku níže je vidět návrh layout webové stránky.



Obrázek 17: Návrh layout aplikace

Na desktopu tak bude stránka i designem tvořena jako přehledná Single Page. V případě mobilních zařízení se následně jednotlivé prvky poskládají pod sebe podobným stylem, jakým se čte procedurální kód. Tedy od shora podle pořadí v kódu, pokud není selektory dáno jinak. O toto rozložení se postará jednoduchý systém grid tříd.

6.9.3 Tvorba kaskádových stylů

Hlavní rámec je v CSS označen třídou "container". A to nejen v single-spa, ale také u každé komponenty odděleně. V případě umístění prvků v řádku se používá třída "row". V rámci každého řádku následně lze rozdělit pomocí breakpointů celistvý řádek na různě velké části. Ty jsou přitom rozděleny do dvanáctin, takže umožňují v řádku až dvanáct oddělených rámců. Pro záhlaví a zápatí stylizuji rámce s plnou šířkou pro každé zařízení. Proto zde vůbec

vnitřní rámce neužívám a navigaci ukládám do rámce s třídou "row". Samotné tělo aplikace již ale grid systém využije plnohodnotně. Vzhledem k tomu, že chci rozdělit řádek s aplikacemi na tři třetiny, tak upravuji třídy v kódu takto.

```
<div class="row">
        <div class="col-md-4 angular_row justify-content-center">
          <application name="@angular-app/angular-project"></application>
        </div>
        <div class="col-md-4 vue_row justify-content-center">
          <application name="@vueapp/vue-app"></application>
        </div>
        <div class="col-md-4 react_row justify-content-center">
          <application name="@react-app/react-project"></application>
        </div>
</div>
```

Příklad 48: Grid layout

Následující ukázka kódu již obsahuje aplikaci grid systému včetně vlastních stylů. Ty jsou uloženy ve variantách angular_row, vue_row a react_row. Třída col-md-4 značí třetinu řádku s tím, že md je middle breakpoint. Poslední třída justify-content-center je další připravený styl Bootstrap, který zarovnává obsah daného rámce do středu. Uvnitř rámců jsou pak vstupní body pro mikroservisy, do kterých styly nijak nezasahují.

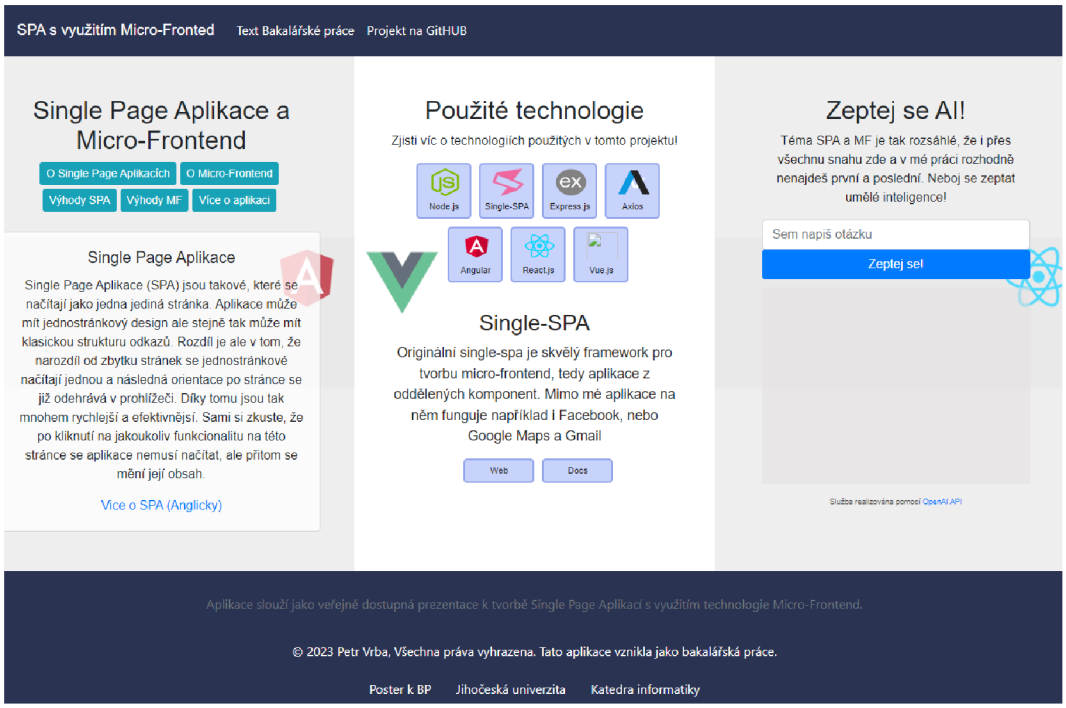
Z Bootstrap tak plnohodnotně využívám celý grid systém, který umožňuje pohodlně navrhnout responsivní design. O jednotlivé prvky se pak starám sa-

mostatně pomocí vlastních stylů, které linkuji i v tomto případě v souboru index.ejs pod single-spa.

```
.mybgcolor{
  background-color: #2b3352;
}
.mylinkscolor{
  color: #ffffff !important;
}
.mylinkscolor:hover{
  color: rgb(193, 190, 190) !important;
}
.openai_realized{
  font-size: 0.6rem;
  color: #585454;
}
```

Příklad 49: Ukázka kaskádových stylů v aplikaci

Po dokončení stylů tak mám plnohodnotnou aplikaci od funkčnosti až po plně responsivní design. Na obrázcích níže jsou přiloženy snímky desktop i responsivní verze stránky.



Obrázek 18: Finální desktop design aplikace

Obrázek 19: Finální responsive design aplikace

6.10 Tvorba PHP prezentace

Jedním z cílů bakalářské práce je právě vytvoření prezentace, která bude volně dostupná a bude fungovat jako dokumentace k technologiím Single Page aplikací a Micro-Frontend. Vzhledem ke specifickým serverovým potřebám pro ostrý běh aplikací postavených na mikroservis architektuře, bude PHP prezentace tou variantou, která bude stále dostupná. Jejím úkolem bude informování návštěvníků o obou přístupech a technologiích. Mimo toho bude obsahovat i pokyny k instalaci aplikace a vyzkoušení na lokálním serveru, včetně volně dostupného kódu a textu práce.

Druhým důvodem, proč tato PHP prezentace vzniká, je zároveň zřetelné porovnání SPA a PHP aplikací, stejně tak jako monolitní a mikroservis aplikace, které je rovněž jedním z praktických výstupů mé bakalářské práce.

6.10.1 Volba technologií

Aplikaci píši jako čistou PHP stránku bez využití PHP frameworků. Základem stránky tak bude "vanilla"PHP, které bude rozšiřovat Bootstrap k realizaci responzivního designu aplikace.

Na stránce tak budou tyto technologie:

- HTML 5 (obsah aplikace)
- CSS 3 (Bootstrap + vlastní CSS)
- PHP 7.x (tvorba kostry a struktury aplikace)

6.10.2 Struktura aplikace

Aplikace má jednoduchou strukturu. Jako vstupní bod zde pochopitelně tvoří index.php. Veškeré součásti webové stránky ale budou obsaženy ve složce assets. Ve své podstatě tak budou základní stránky pouze šablonou, která bude spojovat jednotlivé komponenty. To je výhodné hlavně kvůli tomu, že pokud

budu potřebovat v budoucnu zasahovat do společné části aplikace, tak budu měnit jediný soubor. Po napsání šablony tak vypadá index.php takto.

```
<?php
    $title="Bakalarska prace - SPA a Micro-Frontend";
    $breadcrumb = "<li class='breadcrumb-item active'
   aria-current='page'>Uvod</li>";
?>
<!DOCTYPE html>
<html>
    <head>
        <?php require_once 'assets/parts/header.php'; ?>
    </head>
<body>
<div class="container-fluid">
        <?php require_once 'assets/parts/navigation.php
   '; ?>
        <?php require_once 'assets/parts/pages/
   content_index.php'; ?>
        <?php require_once 'assets/parts/footer.php'; ?>
</div>
</body>
</html>
```

Příklad 50: Kostra PHP stránky

V první PHP sekci se v rámci proměnných předávají unikátní data pro každou stránku, kterou jsou v mém případě nadpisy a struktura drobečkové navigace. Pomocí require_once jsou pak jednorázově volány jednotlivé komponenty ve složce parts, které pak skládají celou webovou stránku. Stejnou kostru mají i další PHP stránky, jen se změnou jiného obsahu ze složky parts/pages.

Tento přístup k tvorbě webové aplikace v PHP volím z toho důvodu, že

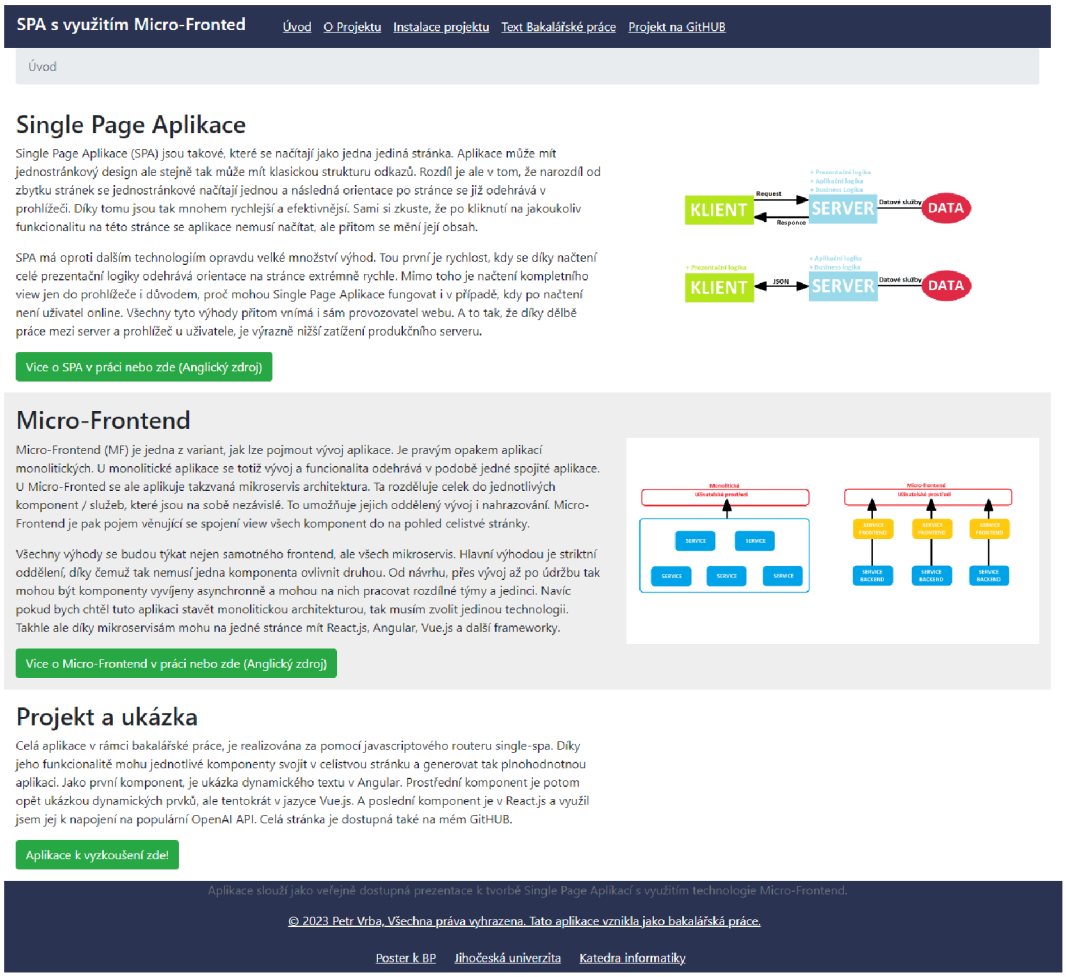
je praktický pro potřeby jednoduché informační stránky a zároveň i proto, že může zdánlivě připomínat práci s komponentami v mikroservis architektuře. Jak ale bude vidět v porovnání, tak se opravdu jedná o zdánlivou podobu.

Kompletní struktura webové stránky je taková:

- root - zde je uložen index.php, sitemap, robots a vstupní soubory stránek
- root/assets - veškeré komponenty a součásti stránky
- root/assets/css - vlastní kaskádové styly
- root/assets/design_img - multimédia v designu
- root/assets/docs - místo uložení dokumentů a souborů (např. text práce)
- root/assets/img - multimédia v obsahu webu
- root/assets/parts - komponenty webové aplikace (např. hlavička, patička)

6.10.3 Podoba a umístění aplikace

Podrobnosti o psaní PHP aplikace schválně nezmiňuji, protože se jedná o základní HTML a PHP kód, který víceméně ani není zaměřením této bakalářské práce. Po dokončení webové stránky vypadá design takto:



Obrázek 20: Vzhled PHP aplikace na produkci

Tato webová stránka je trvale přístupná na mé doméně z adresy `https://bp.vrbapetr.cz/`. Nepovinnou součástí webové prezentace jsou nejen vypracované texty k představení konceptu Single Page aplikací a Micro-Frontend, ale také komentované video, vysvětlující práci v aplikaci jako takové. Design PHP i SPA aplikace je zcela schválně podobný.

7 Porovnání technologií

Ke konci své práce jsem se rozhodl i v rámci svých cílů každou technologii, se kterou se v práci setkám, ohodnotit na základě mnou získaných zkušeností. Hodnocení se u všech bude skládat ze slovního vyjádření hlavních výhod a nevýhod. Zakončením každého hodnocení pak bude souhrnná tabulka s plusy a mínusy ve zkratkách.

7.1 Frontend frameworky

7.1.1 Single-spa

Volba frameworku či metody, kterou budu realizovat samotný micro-frontend, pro mě byla víceméně tou nejsložitější. Existuje celá řada technologií, které spojení výhod Single Page aplikací a micro-frontend umožňují, ale většinou jsem setkal s opravdu vysokými nároky na pochopení rozsáhlé aplikace. Reálně tak pro jedince nebyla tato cesta ideální.

Zcela naopak single-spa z mého úhlu pohledu nabízí velice pochopitelné řešení. Důvodem je hlavně to, že jsou jednotlivé části aplikace přehledně rozděleny. Kořenová aplikace se stará o routování mezi jednotlivými komponentami, zatím co nutné závislosti pro funkci se single-spa, jsou součástí každé komponenty a konfigurují se v ní. S tím je spojen také fakt, že instalace populárních frameworků probíhají v odlehčené verzi přímo pro účely single-spa. Pokud by tak aplikace byla opravdu rozsáhlá a počítala s plným potenciálem každého frameworku, tak se jako možná nevýhoda může jevit nutnost doinstalovat chybějící nutné části.

Hlavní výhodu v podobě zjednodušené implementace frontend frameworků podporuje i fakt, že lze do single-spa přidat i další nástroje, které umožňují pohodlnou realizaci mikroservis architektury. Velice silnou zbraní může být kupříkladu v kombinaci s Module Federation, což opět může posunout výhody single-spa v případě masivních aplikací mnohem dál. Posledním důleži-

tým postřehem, který chci zmínit, je možnost na produkci i v lokálním prostředí spouštět aplikace v klasickém režimu, či režimu standalone. Tento režim jsem již popsal v praktické části. Z mého úhlu pohledu ale možnost spouštět komponentu jako samostatně nepřístupný fragment aplikace zvyšuje bezpečnost celé webové aplikace.

Výhody	Nevýhody
jednoduchá instalace	odlehčené varianty frameworků
kvalitní routování	vysoké nároky na produkci
implementace frameworků	pro zkušené vývojáře
přehledná struktura	
open source	
užívaná u masivních aplikací	
režimy standard/standalone	
zaměření na jeden problém	

Tabulka 1: Výhody a nevýhody single-spa

7.1.2 Angular

V případě Angular jsem začínal tento projekt s víceméně nulovými zkušenostmi, tak pro mě bylo složitější se v jeho problematice orientovat. Namísto klasického JavaScript totiž Angular pracuje s TypeScript, což pro mě byla novinka. S porozuměním původnímu JavaScript ale bylo jen otázkou chvíle, nežli jsem se v problematice dostatečně vyznal k vytvoření jednoduchého modulu.

Užívání TypeScript jsem tak postupem času hodnotil spíše pozitivně, protože v rámci aplikace dává stoprocentně smysl. Oproti ostatním frameworkům je Angular rozhodně rozsáhlejší. Sám o sobě tak může stačit k vývoji masivních aplikací, přičemž další frameworky pro podobnou funkcionalitu potřebují instalovat doplňující služby a balíčky. S rozsáhlostí frameworku ale pramení také jeho menší přehlednost, kterou u Angular hodnotím ze všech tří rozhodně nejhůř. Mimo toho jsem se setkal v rámci instalace a konfigurace s celou řadou

potíží, leč jsem užíval oficiální produkci správce balíčků npm. Tyto problémy jsem pak při hledání řešení našel v mnoha dotazech i na podpoře Angular a odborných fórech, takže jde nejspíše o systematický problém.

V rámci práce jsem si mohl vyzkoušet reaktivitu přes dynamické zobrazování prvků, která je výborně propracovaná. Opravdu největším překvapením pro mě byl fakt, že se v modulech tvořených pomocí Angular zalamoval text v HTML tazích pro odstavec po zmáčknutí enteru (při psaní kódu uvnitř Visual Studio Code). To jinak v rámci programovacích jazyků většinou nic neznamená a řádkování se řeší spíše jen kvůli přehlednosti zdrojového kódu. Angular navíc neustále kontroluje syntaxi kódu na všech úrovních. Neumožňuje tak například HTML prvky umisťovat mimo tělo kořenového rámce, nebo zanechávat neuzavřené rozličné tagy. Tím rozhodně může Angular pomoci s čistotou kódu. Nalezené chyby se ukazují v prohlížeči jako chyba při vykreslování aplikace s popisem řádku, většinou dokonce i pozice znaku.

Výhody	Nevýhody
mohutný framework	složitá struktura
užívání TypeScript	obtížný na pochopení
oboustranná reaktivita	náročná instalace
připravený pro MongoDB	
hlídání syntaxe	
přehledné hlášení chyb	

Tabulka 2: Výhody a nevýhody Angular

7.1.3 React.js

V rámci React.js jsem před touto bakalářskou prací měl nejvíce zkušeností. Proto jsem se i rozhodl napojit externí API právě v této aplikaci, ačkoliv nakonec byla implementace velice jednoduchá. React.js má rozhodně největší nevýhodu v tom, že v mnoha ohledech odchází od řady standardních prvků,

které se objevují v jiných frameworcích. Při ladění aplikace jsem se kupříkladu setkal s tím, že klasický identifikátor class v HTML elementech je sice vykreslený, ale v konzoli se propisují chyby, protože React.js požaduje změnu identifikátoru na className.

Jedná se o jednu z mnoha komplikací, které se mohou při vývoji v React.js ukázat, pokud již má vývojář vrozené syntaxe a postupy z jiných frameworků. Stejně tak i DOM zde funguje jinak. Aby aplikace mohla být opravdu soběstačná a plnohodnotná, tak je potřeba mimo integrace React.js také integrovat další nástroje jako je ReactRouter, ReactDOM nebo dokonce architektonický prvek Redux.

Z pozitivních stránek je potřeba nejvíce zvýraznit psaní JSX kódu. Jedná se o možnost psát kombinovaný kód, v jehož těle se na jednom místě nachází HTML a JavaScript. To může snížit přehlednost jednoho modulu, ale naopak zmenšuje počet importů a závislostí, které v aplikaci jinak musí být. Pro mnohá užití může být React.js výhodný díky svojí menší velikosti a poměrně rychlému pochopení, pokud uživatel má předchozí zkušenosti s JavaScript. Z pokročilých možností React.js je pak zásadní výhodou používání nástrojů state a hook, které umožňují jednoduchou, leč jednostrannou reaktivitu.

Výhody	Nevýhody
jednoduchá instalace	horší hlášení chyb
JSX kód	méně standardizované přístupy
funkce state a hook	potřebuje rozšíření pro velké aplikace
menší velikost	
dobré porozumění	

Tabulka 3: Výhody a nevýhody React.js

7.1.4 Vue.js

S prací ve Vue.js jsem na začátku práce měl minimální zkušenosti. Jedná se o framework, který má velice přehlednou syntaxi a hlavní výhodou je dokonalé

kontrolování použitých zdrojů a samotné syntaxe kódu. Stejně tak, jako je tomu například u Angular, je i zde striktní hlídání užívání párových tagů v HTML šabloně a celé syntaxe kódu.

Vue.js je automaticky navrženo pro práci s komponentami. To znamená, že i když jsem v něm dělal jednoduchou aplikaci, tak to musela být i tak komponenta, namísto jediného kódu, jako je tomu třeba u Angular. Tento princip ale má spoustu výhod, jako je třeba předávání parametrů do komponenty během vykreslování. Každá komponenta má oddělené CSS, výkonný JavaScript a HTML šablonu, leč jsou v jediném souboru. Díky tomu jsou komponenty velice přehledné.

V této oblasti se ale najdou i zajímavosti a úskalí, která Vue.js má. Při práci jsem řešil nefunkčnost reaktivity, která fungovala voláním metody v komponentě. Přitom metoda byla v pořádku. Následně jsem zjistil, že byl problém v jiné metodě, což spustilo systémovou chybu a celá komponenta nefungovala, aniž by ale konzole či kompilační výstup, jakkoliv vývojáře informovali. To považuji za velký nedostatek Vue.js. Naopak jako velkou výhodu v tomto ohledu považuji fakt, že Vue.js požaduje využití všeho, co je v komponentně implementované. Pokud jsem tak přidal proměnnou, nebo metodu, tak automaticky Vue.js vyvolal kompilační chybu, pokud nebyla následně v šabloně nějakým způsobem užita. To mě přesvědčilo o tom, že vývoj ve Vue.js může být cestou k opravdu jednoduchému a přehlednému kódu bez zbytečných částí.

`Výhody`	`Nevýhody`
výběr verze při instalaci	špatné hlášení chyb
jednoduchý kód	nestabilní Vue.js 3
striktní oddělení	nucená práce s komponentami
hlídání syntaxe	
hlídání užití zdrojů	

Tabulka 4: Výhody a nevýhody Vue.js

7.1.5 Vzájemné porovnání

Jedním z cílů bakalářské práce také bylo po dokončení projektu zhodnocení vlastností a porozumění všem třem frontend frameworkům, které jsou použity. Abych docílil opravdu co největší objektivity, tak jsem v předchozí kapitole výhody a nevýhody specifické pro každý framework již shrnul. V této části se pokusím nastínit, nakolik mi přišly jednotlivé frameworky v porovnání s ostatními konkurenceschopné tabulkovou formou.

Kritérium	Angular	React.js	Vue.js
Instalace	Problematická	Bezproblémová	Bezproblémová
Konfigurace	Problematická	Bezproblémová	Bezproblémová
Přehlednost	Obtížně přehledný	Přehledný	Přehledný
Porozumění	Obtížné	Mírně obtížné	Jednoduché
Čistota kódu	Částečně hlídaná	Nehlídaná	Plně hlídaná
Hlídání chyb	Detailní	Detailní	Základní
Soběstačnost	Plně	Částečně	Plně
Reaktivita	Oboustranná	Jednostranná	Jednostranná

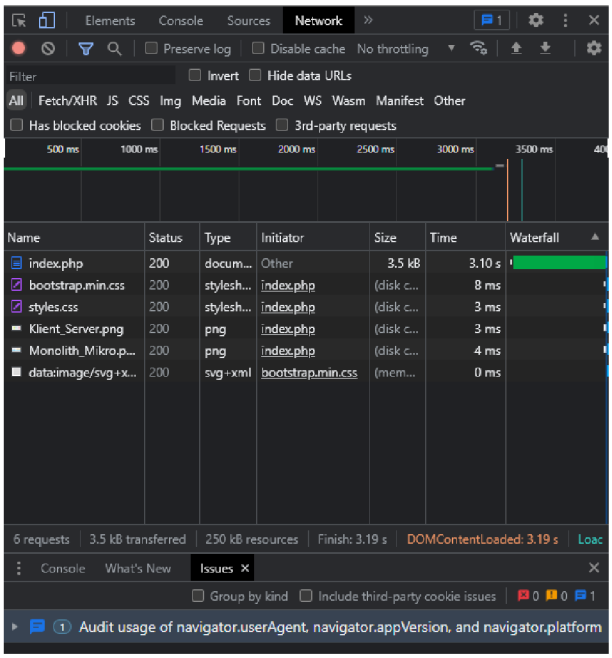
Tabulka 5: Porovnání frontend frameworků

V rámci hodnocení jsem schválně nevolil číselnou škálu, protože se mi pro jednotlivá kritéria nedařilo najít správný popis každého stupně. V rámci porovnání v tabulce výše vychází, že je Angular náročný framework, který ale nabídne také nejvíce funkcionalit. V případě Vue.js a React.js jsou pak frameworky hodnoceny podobně. Každý má své jasné výhody, ale ani jeden z této dvojice druhý framework výrazně neporáží.

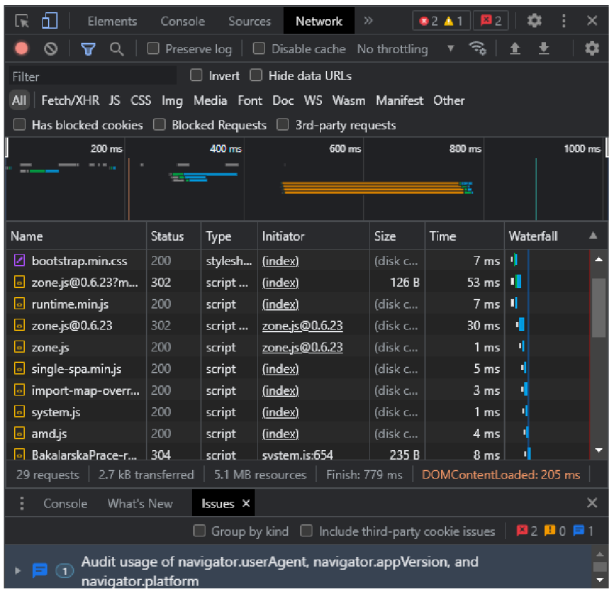
7.2 SPA versus PHP aplikace

Single Page aplikace a PHP aplikace jsou zásadně rozdílné. V teoretické i praktické části jsem několikrát zmiňoval zásadní výhody Single Page aplikací proti statickým webovým stránkám. Při dokončování projektu jsem ale narazil na problém, který mě donutil částečně změnit plány v rámci publikace aplikace a také mi odhalil zásadní komplikaci. Propojení Single Page aplikací s Micro-Frontend totiž vyžaduje velice specifické prostředí serveru. Jednotlivé mikroservisy tak pracují na platformách jako Microsoft Azure, Amazon S3+, Google Cloud Services a mnohé další. Nejedná se ale o levná řešení webového prostoru. Naopak PHP aplikace mohou fungovat víceméně na všech základních hostingových variantách a případné problémy mohou vznikat hlavně z přetížení serveru.

Na následujících dvou obrázcích totiž představím velké výkonnostní rozdíly obou aplikací. SPA i PHP aplikace byly testovány ve stejném prostředí, se stejným výkonem a byla použita i stejná média. K měření jsem použil vývojářská rozšíření pro Google Chrome.



Obrázek 21: Výkon načítání PHP aplikace



Obrázek 22: Výkon načítání SPA aplikace

Z výsledků je patrný velký rozdíl v načítaných datech. PHP stránka načítá pouze aktuální view, tedy dokument, který přišel jako odpověď od serveru. Vedle toho SPA naopak načetlo všechna view, ale generovala se jenom ta, která jsou aktuální. Velikost načítaných dat je tak menší u PHP stránky, což může být výhoda u pomalejších nebo nestabilních připojení. Oproti tomu SPA aplikace díky rychlému vykreslování dokáže nabídnout zásadně menší odezvu, navzdory většímu množství dat. PHP aplikace navíc obsahuje v moment načítání jen zlomek dat, která obsahuje SPA aplikace. Viditelným rozdílem je také rychlost načítání hlavních částí stránky. V případě PHP aplikace byla několikanásobně vyšší doba načítání statického souboru index. Vedle toho SPA aplikace dokázala načítat velké množství souborů rychle díky asynchronnímu zpracování. To je do značné míry zaviněno nejen podstatou SPA aplikace jako takové, ale také aplikací mikroservis architektury. Jednotlivé komponenty jsou sice vykreslovány jako celek, jenže načítají se z vícero zdrojů a díky tomu tak nedochází k hromadění requestů na jedinou adresu.

Závěrem slovního porovnání je pak nutné říct, že v moment kdy na PHP aplikaci budu chtít vidět nová data, tak se vždy musí proces odesílání requestu

a přijímání odpovědi ze serveru opakovat. U SPA aplikace je již ale nahraná kompletní funkce stránky v prohlížeči. Díky tomu je tak konkrétně má vlastní aplikace zcela nezávislá na serveru s výjimkou odpovědí od OpenAI API.

Pro finální porovnání aplikací jsem se rozhodl pro číselnou škálu od 1 do 5 v následující podobě: 1 - excelentní; 2 - nadprůměrné; 3 - průměrné; 4 - nedostatečné; 5 - nereálné. V obou případech přitom zvažuji nejen mé vlastní aplikace, ale všeobecně aplikace podle specifikací SPA a PHP.

Kritérium	SPA aplikace	PHP aplikace
Náročnost vývoje	3	1
Nároky na hosting	4	1
Odezva aplikace	1	3
Offline funkce	1	5
Vytížení serveru	1	3
Škálovatelnost	1	2
SEO Optimalizace	3	1
Bezpečnost	2	2
Podpora v ČR	4	1
Plánování vývoje	1	2
Volnost užitých technologií	1	4
Údržba masivních aplikací	2	4

Tabulka 6: Porovnání PHP a SPA aplikací

8 Závěr

Bakalářská práce se zabývala tvorbou webových aplikací za použití technologií Single Page aplikace a Micro-Frontend zároveň. Od tohoto spojení jsem očekával největší náročnost hlavně co se týče pochopení problematiky na úrovni mikroservis architektury, což se mi také potvrdilo. V přípravě na práci jsem procházel četné technologie pro realizaci mikroservis, ze kterých v mých očích nakonec vzešli jako vítězové Module Federation a single-spa. Volbou pro realizaci se nakonec stalo single-spa proto, že mi přišlo jako nejčistší řešení problematiky, a navíc je užíváno na mohutné aplikace globálního významu jako Gmail, Facebook a další.

V teoretické části jsem se zaměřil především na popsání Single Page aplikací a Micro-Frontend v širším měřítku, stejně tak jsem nastínil zásadní rozdíly oproti statickým stránkám a projektům s monolitickou architekturou. K tomu jsem přidal i základní povědomí o funkci architektonických a návrhových vzorů, společně s užívanými webovými technologiemi.

V praktické části jsem vytvořil Single Page aplikaci, na kterou jsem aplikoval mikroservis architekturu a demonstroval jsem její hlavní výhody. Součástí aplikace jsou tak tři moduly – po jednom v React.js, Vue.js a Angular, přičemž vše zastřešuje router single-spa. Tyto moduly tvoří demonstraci dynamických prvků na SPA a především ukazují, že mikroservis architektura umožňuje použít na jednom frontend několik frameworků, které by u monoliticky psané aplikace nemohly fungovat pospolu. Z důvodu vysokých nároků na produkční server není aplikace spuštěna živě, ale ve své finální podobě je veřejně dostupná v Git repozitáři `https://github.com/VrbaPetr/BP_Project`.

Ke konci praktické části práce jsem pak vytvořil PHP aplikaci bez použití frameworků na základě stejného obsahu jako Single Page aplikaci. Tato aplikace bude na mém hostingu fungovat jako slíbená volně dostupná prezentace seznamující s problematikou SPA i Micro-Frontend a bude zde i volně dostupný samotný projekt. Vše na adrese `https://bp.vrbapetr.cz/`. Posledním důvo-

dem pro zpracování PHP aplikace bylo poskytnutí porovnání vlastností a metrik mezi PHP a SPA aplikacemi, které rovněž byly jedním z výstupů bakalářské práce společně s porovnáním užitých frontend frameworků.

Z mého úhlu pohledu jsou Single Page aplikace a mikroservis architektura budoucností vývoje moderních aplikací. A to jak odděleně, tak i společně, kde mohou využít dohromady celého spektra výhod. O svých kvalitách mě rovněž přesvědčil Micro-Frontend router single-spa. S užitím zpracovaných přístupů a technologií je možné aplikaci dlouhodobě rozvíjet a zdokonalovat, přičemž se mohou týmy vývojářů vyhnout celé řadě jinak typických problémů. Jedinou nevýhodu stále vidím v ekonomických nárocích na vývoj takové aplikace. Spojení několika týmů vývojářů schopných získat maximum z rozdílných technologií bude vždy nákladné. Stejně tak je problémem i vysoká náročnost na ostrou produkci oproti PHP aplikacím. Rozhodně jsem ale během práce zjistil, že propojení těchto technologií nevyužije každá aplikace. Zcela naopak dává smysl u aplikací, od kterých se očekává velký rozsah funkcí a dlouhá doba existence, kde se pak i počítá s vyššími investicemi do zmíněných oblastí. Já sám jsem v této oblasti našel během psaní bakalářské práce zalíbení a rád se v pozdějších pracích, nejen akademického charakteru, vrátím do oblasti mikroservis architektury a Single Page aplikací.

Seznam použité literatury a zdrojů

[1] SCOTT, Emmit. SPA design and architecture: understanding single-page web applications. Shelter Island, New York: Manning Publications Co., 2016. ISBN 978-161-7292-439.

[2] HARRIS, Chandler. Microservices vs. monolithic architecture: When monoliths grow too big it may be time to transition to microservices. Atlassian [online]. Atlassian, 2023 [cit. 2023-03-13]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[3] GILL, Navdeep Singh. Micro Frontend Architecture and Best Practices. XenonStack [online]. 2022 [cit. 2023-03-13]. Dostupné z: https://www.xenonstack.com/insights/micro-frontend-architecture

[4] EIJGERMANS, Peter. Micro Frontends With Example. DZone [online]. 2021 [cit. 2023-03-04]. Dostupné z: https://dzone.com/articles/micro-frontends-by-example-8

[5] RADY, Ben. Serverless Single Page Apps: Fast, Scalable, and Available. USA: The Pragmatic Programmers, 2016. ISBN 978-1-68050-149-0.

[6] Fundamentals of Software Architecture. Geeks for Geeks [online]. Noida, Uttar Pradesh: Geeks for Geeks, 2022 [cit. 2023-03-10]. Dostupné z: https://www.geeksforgeeks.org/fundamentals-of-software-architecture/

[7] Software Architecture & Design Introduction. TutorialsPoint [online]. 2023 [cit. 2023-03-10]. Dostupné z: https://www.tutorialspoint.com/software_architecture_design/introduction.htm

[8] COCCA, Germán. The Software Architecture Handbook. FreeCodeCamp [online]. freeCodeCamp, 2022 [cit. 2023-03-10]. Dostupné z: https://www.freecodecamp.org/news/

an-introduction-to-software-architecture-patterns/#what-s-your-infrastructure-like

[9] Overview of ASP.NET Core MVC. Learn Microsoft - ASP.NET [online]. Microsoft, 2022 [cit. 2023-03-04]. Dostupné z: https://learn.microsoft.com/en-gb/aspnet/core/mvc/overview?WT.mc_id=dotnet-35129-website&view=aspnetcore-7.0

[10] F. DOOLEY, John. Software Development, Design and Coding: With Patterns, Debugging, Unit Testing, and Refactoring. 2. vydání. Galesburg, Illinois: Apress, 2017. ISBN 978-1-4842-3152-4.

[11] OSMANI, Addy. Learning: Javascript Design Patterns. 2. vydání. Sebastopol, CA: O'Reilly Media, 2012. ISBN 978-1-449-33181-8.

[12] Model-View-ViewModel (MVVM). Learn Microsoft - ASP.NET [online]. Microsoft, 2022 [cit. 2023-03-04]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

[13] Binding syntax. Angular [online]. 2023 [cit. 2023-03-10]. Dostupné z: https://angular.io/guide/binding-syntax

[14] AngularJS Data Binding. W3 Schools [online]. W3 Schools, 2023 [cit. 2023-03-10]. Dostupné z: https://www.w3schools.com/angular/angular_databinding.asp

[15] DEURSEN, Steven Van a Mark SEEMANN. Dependency Injection: Principles, Practices, and Patterns. Shelter Island, NY: Manning Publications, 2019. ISBN 9781617294730.

[16] DEAN, John. Web Programming: with HTML5, CSS, and Javascript. Burlington, Massachusetts: Jones & Barlett Learning, 2019. ISBN 9781284091793.

[17] Document Object Model (DOM). MDN Web Docs [online]. Mozilla Corporation, 1998–2023 [cit. 2023-03-08]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`

[18] LAURENČÍK, Marek. Tvorba www stránek v HTML a CSS. Praha: Grada Publishing, 2019. Průvodce. ISBN 978-80-271-2241-7.

[19] ESPOSITO, Dino. Modern Web Development: Understanding domains, technologies, and user experience. Redmond, Washington: Microsoft, 2016. ISBN 978-1-5093-0001-3.

[20] Codecademy Team. What Is a Framework?. Codecademy [online]. Codecademy, 2021 [cit. 2023-03-08]. Dostupné z: `https://www.codecademy.com/resources/blog/what-is-a-framework/`

[21] Understanding client-side JavaScript frameworks. MDN Web Docs [online]. 1998–2023 [cit. 2023-03-08]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks`

[22] BANKS, Alex a Eve PORCELLO. Learning react: functional web development with react and redux. Sebastopol: O'Reilly, 2017. ISBN 978-1-491-95462-1.

[23] MARDAN, Azad. React Quickly: PAINLESS WEB APPS WITH REACT, JSX, REDUX, AND GRAPHQL. Shelter Island, NY: Manning Publications, 2017. ISBN 978-1-61729-334-4.

[24] SESHADRI, Shyam. Angular: Up and Running. California: O´Reilly Media, 2018. 1. vydání. ISBN 978-1-491-99983-7.

[25] NELSON, Brett. Getting to Know Vue.js. Minnesota: Apress, 2018. 1. vydání. ISBN 978-1-4842-3781-6

[26] Introduction | Vue.js. Vue.js [online]. © 2014-2023 [cit. 2023-02-06]. Dostupné z: `https://vuejs.org/guide/introduction.html`

[27] Package management basics. MDN Web Docs [online]. Mozilla Corporation, 2023 [cit. 2023-03-10]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management`

[28] Npm: javascript package manager. Npm Docks [online]. [cit. 2023-03-09]. Dostupné z: `https://docs.npmjs.com/cli/v9/commands/npm`

[29] Npm: About npm. Npm Docks [online]. [cit. 2023-03-09]. Dostupné z: `https://docs.npmjs.com/about-npm`

[30] About Node.js®. Node.JS [online]. [cit. 2023-03-14]. Dostupné z: `https://nodejs.org/en/about/`

[31] Downloading and installing Node.js and npm. Npm Docks [online]. [cit. 2023-03-14]. Dostupné z: `https://docs.npmjs.com/downloading-and-installing-node-js-and-npm`

[32] Getting Started. Visual Studio Code [online]. Seattle: Microsoft, 2023 [cit. 2023-03-14]. Dostupné z: `https://code.visualstudio.com/docs`

[33] Getting Started with single-spa: JavaScript Microfrontends. Single-spa [online]. 2023 [cit. 2023-03-14]. Dostupné z: `https://single-spa.js.org/docs/getting-started-overview/#create-a-single-spa-application`

[34] Building and serving Angular apps. Angular [online]. 2023 [cit. 2023-04-11]. Dostupné z: `https://angular.io/guide/build`

[35] Concept: Applications: Building single-spa applications. Single-spa [online]. 2023 [cit. 2023-04-11]. Dostupné z: `https://single-spa.js.org/docs/building-applications`

[36] Layout Engine: Introduction. Single-spa [online]. 2023 [cit. 2023-04-11]. Dostupné z: `https://single-spa.js.org/docs/layout-overview`

Seznam příkladů

Seznam tabulek

Seznam obrázků

A Příloha

Repozitář s praktickou částí: `https://github.com/VrbaPetr/BP_Project`

B Příloha

PHP prezentace: `https://bp.vrbapetr.cz/`