Univerzita Hradec Králové
Fakulta informatiky a managementu

Využití technologie WebAssembly pro tvorbu SPA

Bakalářská práce

Autor: Tomáš Trávníček
Studijní obor: AI3-p

Vedoucí práce: Ing. Jakub Beneš

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně s použitím uvedené literatury.

……

V Hradci Králové dne 28.4.2022 Tomáš Trávníček

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Jakubovi Benešovi za metodické vedení práce, věcné připomínky a vstřícnost při konzultacích a vypracování bakalářské práce.

Anotace

Bakalářská práce se zabývá problematikou webových technologií, které se dnes využívají pro tvorbu webových stránek. Porovnávají se tradiční statické stránky, dále hojné využití JavaScriptu pro tvorbu single page aplikací a v neposlední řadě je práce zaměřena na web assembly jako technologii a konkrétně její využití pro tvorbu single page aplikací jako alternativu pro JavaScript. V teoretické části jsou vysvětleny pojmy, které jsou klíčové pro pochopení zmíněných technologií, a dále jsou v praktické části technologie porovnány podle vybraných kritérií.

Praktická část se skládá z dvou identických aplikací - jedna napsaná v JavaScriptovém frameworku React a druhá pomocí frameworku Yew pro jazyk Rust. Aplikace jsou dále porovnány z hlediska výkonu a rychlosti vývoje.

Annotation

Title: Using WebAssembly to create Single Page Applications

This Bachelor Thesis contemplates the area of web technologie which are currently used to build websites. Furthermore, comparison between static, JavaScript-heavy and WebAssembly pages is discussed. Webassembly is specifically compared as an alternative to current mainstream JavaScript technologies. The theoretical part of this thesis explains the basic terms which need to be understood in order to fully cover the topic of WebAssembly and modern JavaScript frameworks. Different user perspectives are also discussed. The practical part of this thesis consists of two applications - one is built using a JavaScript framework called React and the other is built using Yew which is a framework built on top of a programming language called Rust. These two applications are then compared regarding their performance and write-time effectiveness.

Obsah

1 Úvod

Píše se druhé desetiletí 21. století a technologický vývoj se nadále urychluje. To není výjimkou ani v oboru webových aplikací. Vznik nových knihoven a frameworků se dá přirovnat k frekvenci tiku vteřinových hodinek a mnoho z těchto nových směrů má dle různých odborníků růst i nadále [4]. Jeden z těchto směrů je právě WebAssembly. Stávající trend JavaScriptových frameworků pro tvorbu stránek je v aktuální situaci jednoznačně dominantní, ale na povrch se dostávají nové možnosti každým dnem.

Téma bylo vybráno právě proto, že je technologie WebAssembly objektivně vnímána s vysokým potenciálem, což je možné podložit různými projekty, které tuto technologii implementují. Jedním příkladem může být cloud-based nástroj pro designéry a grafickou práci - Figma. Figma pracuje čistě v rámci prohlížeče a pro většinu efektů, které mohou být výpočetně náročné, se spoléhá na C++ ve formě WebAssembly. Jako další příklad lze uvést online platformu pro stolní hru šachy - Lichess. Platforma ve svém jádru využívá engine Stockfish, který je celý napsán v C++ a jeho implementace je díky WebAssembly vysoce efektivní.

Práce se svém obsahu pokusí odpovědět na otázky, zda je WebAssembly skutečně technologie, které by měla být věnována pozornost a zda je vhodné ji začít využívat na reálných projektech.

2 Cíl práce

Cílem práce je prozkoumat, seznámit a rozšířit znalosti čtenáře v oblasti vývoje Single Page Aplikací (SPA).

Práce si dále klade za cíl porovnat tradiční technologie pro tvorbu SPA a nově vznikající technologie WebAssembly na základě předem definovaných kritérií. Následně se v praktické části práce porovnají vybrané technologie na praktických příkladech.

3 Single Page Aplikace (SPA)

Samotný koncept SPA v oboru webových technologií existoval již kolem roku 2003, ale ke konkrétním implementacím a hojnému využití došlo až přibližně v druhém desetiletí 21. století. Do té doby bylo pro psaní webových stránek využíváno převážně principu statických webových stránek. Statické stránky spoléhají na funkcionalitu, ve které musí mít každá samostatná stránka vlastní soubor a pro přechod mezi individuálními stránkami dochází vždy k načtení nového (samostatného) html souboru [5].

3.1 Úvod do SPA(s)

Na rozdíl od statických stránek mají SPA (Single Page Applications) již dle názvu jen "jednu" stránku, na které probíhá veškerá aktivita. Jednou stránkou je myšleno, že oproti stránkám statickým mají jen jediný html soubor, který se obvykle nazývá index. Nicméně to nutně neznamená, že na stránce neexistuje žádná strukturovaná navigace. Tato funkcionalita je naopak velmi častá a dá se říci, že je v dnešní době standardem. Je to stručně řečeno abstraktní vrstva, která leží až nad vrstvou fyzickou a umožňuje mnohem komplexnější a flexibilnější řešení než jsou fyzické (statické) html stránky. Tuto vrstvu infrastruktury SPA je možné aplikovat díky JavaScriptu v prohlížečích.

3.1.1 JavaScript

JavaScript je především známý jako událostmi řízený skriptovací jazyk , který se těší využití převážně ve webových prohlížečích. Nicméně jedná se o multiplatformní a také objektově orientovaný jazyk, který našel využití i na straně serveru pro psaní API a jiných back-endových služeb (Deno/Node.js, Express atd.) [6]. Pro účely této práce je ale nejdůležitější využití právě ve zmíněných prohlížečích.

Z bezpečnostních důvodů a soukromí uživatele je ale práce JavaScriptu v prohlížeči omezená a není tedy možné například pracovat s místními soubory počítače.
Výjimku tvoří specifická úložiště (localStorage, sessionStorage) prohlížeče, která dokáží ukládat textové řetězce nebo zachovat data v konkrétním strukturovaném

formátu. Tato úložiště typicky nacházejí využití v aplikacích, které mají vyšší nároky na uživatelskou přívětivost a rychlost.

V kontextu webových stránek se JavaScript načítá skrze specifický objekt v html stromu hlavního html souboru - obvykle index.html. Konkrétně se jedná o script tag, který přímo obsahuje chtěný JavaScript kód - viz Výpis kódu 3.1.

```
<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

Výpis kódu 3.1: Inline script tag. Zdroj: JavaScript. W3Schools [online]. [cit. 2022-04-09]. Dostupné z: https://www.w3schools.com/js/js_whereto.asp

Dále je v případě komplexnějšího chování možné využít odkaz přes relativní adresu souboru ve formátu `*.js` . Odkaz se specifikuje pomocí atributu src, který zkratkou označuje source - česky zdroj - viz Výpis kódu 3.2.

```
<script src="myScript1.js"></script>
<script src="myScript2.js"></script>
```

Výpis kódu 3.2: Script tag včetně source atributu. Zdroj: JavaScript. W3Schools [online]. [cit. 2022-04-09]. Dostupné z: https://www.w3schools.com/js/js_whereto.asp

Zdroj script tagu může mimo jiné vést i na externí soubory dostupné na internetu - viz Výpis kódu 3.3.

```
<script src="https://www.w3schools.com/js/myScript.js"></script>
```

Výpis kódu 3.3: Externí script tag. Zdroj: JavaScript. W3Schools [online]. [cit. 2022-04-09]. Dostupné z: https://www.w3schools.com/js/js_whereto.asp

V kontextu celého HTML souboru se obvykle script tag vkládá až na konec hlavičky (`head tag`) nebo na konec těla (`body tag`).

3.1.2 JavaScript routing

Router v JavaScriptu je objekt, který jednoduše mapuje adresy URL k specifickým funkcím a při navštívení dané URL se odpovídající funkce zavolá. Jedná se o klíčovou komponentu při vytváření moderních webů pomocí populárních frontendových frameworků. Umožňuje efektivně spravovat interní stav aplikace, zobrazovat různé pohledy stránky (například administrátor/uživatel) a skrýt určité části webu za přihlášení. Kontrastně proti statickému přístupu dělení stránky umožňuje větší flexibilitu - například dynamické URL adresy s parametry. Tyto parametry mohou specifikovat pro ilustraci například identifikátor entity, která se váže na právě navštívenou stránku. Scénář pro vizualizaci by se mohl skládat z URL adresy, která zobrazuje články a parameter by obsahoval ID článku, který se má zobrazit [7].

```
your-webpage.com/articles/123456
```

Výpis kódu 3.4: Příklad route cesty. Zdroj: Autor práce.

Výpis kódu 3.4 popisuje URL adresu webu, který se jmenuje your-webpage.com . Domovská stránka se tedy nachází na / . Route /articles specifikuje, že by se měly zobrazit články a číslo za specifikací článků určuje unikátní identifikátor, který zobrazí jeden článek. Jedná se pouze o ilustrativní příklad, v realitě by pravděpodobně unikátní ID vypadalo složitěji, například uuid[1] formát.

Důležitý rozdíl mezi routingem a statickými HTML stránkami je v implementaci. Kde statickým stránkám stačí nový HTML soubor a objekt v DOMu, který se na něj odkáže, routing naopak spoléhá čistě na JavaScript.

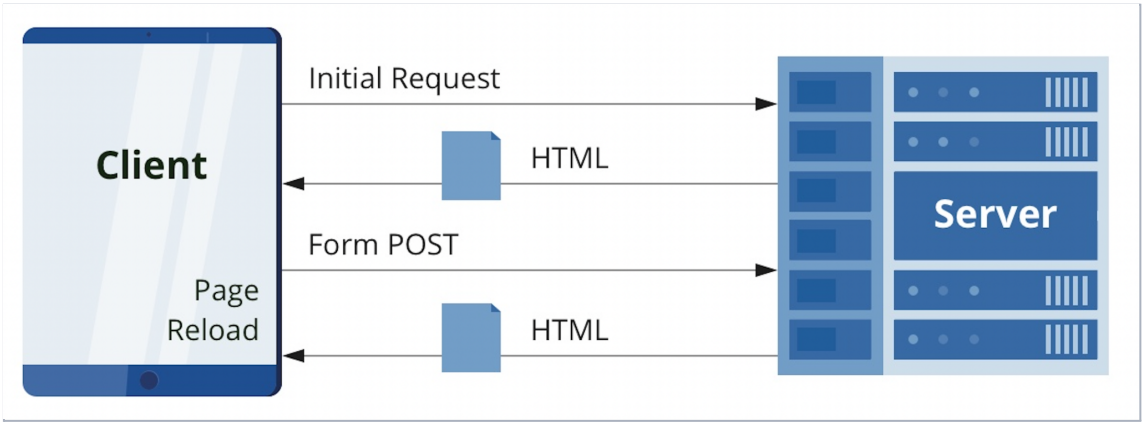
Dalším kritickým rozdílem je způsob, jakým se odlišné stránky mění.

3.1.3 Komunikace klient - server

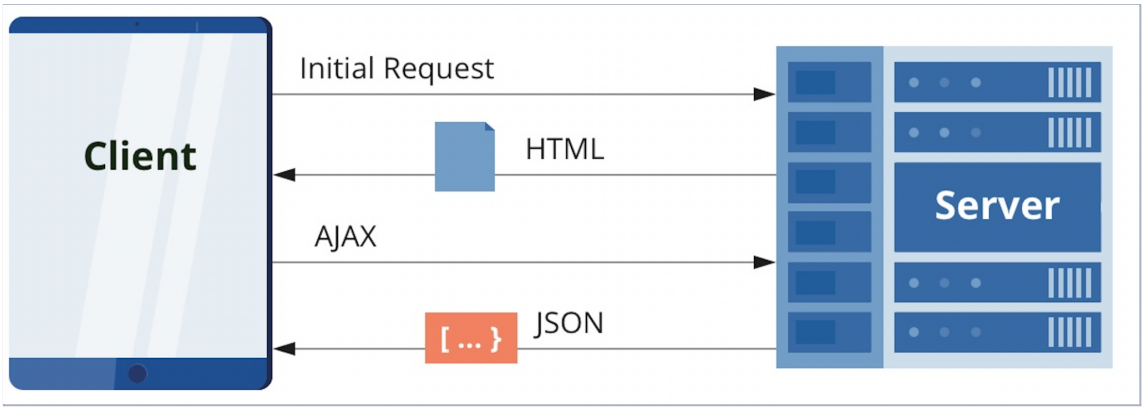
V rámci naprosté většiny webových stránek, které jsou dnes dostupné na internetu, je na pozadí komunikační model, který umožňuje strukturovaně posílat zprávy, požadavky a odpovědi. Tento model do určité míry ovlivňuje architekturu

[1] uuid (anglicky Universally Unique Identifier) je unikátní a univerzální identifikátor používaný k identifikaci v informačních systémech

aplikací/webů, kterými se tato práce zabývá a pro další odlišení funkcí statických webů oproti single page aplikacím je tato komunikace klíčová [8].



Obr. 3.1: Komunikace klient-server u statických webů. Zdroj: VALUY, Sergey. SPA and Static sites comparison. DZone [online]. Jun. 17, 2020 [cit. 2022-04-09]. Dostupné z: https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli



Obr. 3.2: Komunikace klient-server u SPA. Zdroj: VALUY, Sergey. SPA and Static sites comparison. DZone [online]. Jun. 17, 2020 [cit. 2022-04-09]. Dostupné z: https://dzone.com/articles/the-comparison-of-single-page-and-multi-page-appli

Na Obr. 3.1: Komunikace klient-server u statických webů (dále jen Obr. 3.1) a Obr. 3.2: Komunikace klient-server u SPA (dále jen Obr. 3.2) můžeme pozorovat 2 entity. Jednou je klient (Client) a druhou je analogicky server (Server). Ve své podstatě se jedná o 2 počítače, které mají k dispozici prostředky umožňující komunikaci po síti.

Entity jsou geograficky typicky odděleny - klient je osobní počítač, případně telefon s přístupem na internet a server je naopak počítač uložený v serverovně, případně datovém serveru. Nicméně server může být i osobní počítač uložený u kohokoliv doma v obývacím pokoji. Jedná se tedy pouze o termín, který specifikuje, kdo je na kterém konci komunikace. Klient představuje stranu, která posílá požadavky a server je obsluhuje a odpovídá.

První krok je u obou případů (Obr. 3.1, Obr. 3.2) stejný. Klient posílá dotaz (request), zde se jedná o dotaz na zobrazení jisté url. Server má namapované porty na běžící procesy (webové aplikace) a za předpokladu, že má klientův požadavek všechny požadované kvality, požadavek je zpracováván a posílá se odpověď. Zde se jedná o HTML soubor, který obsahuje základní strukturu stránky a typicky odkazy na další externí soubory (stylování, JavaScript). V tento moment je načtena celá stránka.

Rozdíl mezi situacemi výše nastává až v moment, kdy má dojít k přechodu na jinou stránku.

Na Obr. 3.1 je tato situace popsána požadavkem “Form POST” - na server se posílá formulář, server odpovídá s novým HTML souborem (přesměrování) a dochází k přechodu na jinou URL.

Na Obr. 3.2 taktéž dochází k dalšímu dotazu na serveru, ale odpověď není nová HTML stránka, nýbrž strukturovaná zpráva, která obsahuje potřebné informace, s kterými pracuje JavaScript na straně klienta. Zde se jedná o data ve formátu JSON.

Jakým způsobem JavaScript naloží s daty se může lišit scénář od scénáře, ale pro replikaci chování statické stránky dochází k změně UI - přechod na jinou stránku pomocí routeru (viz předchozí kapitola - JavaScript routing).

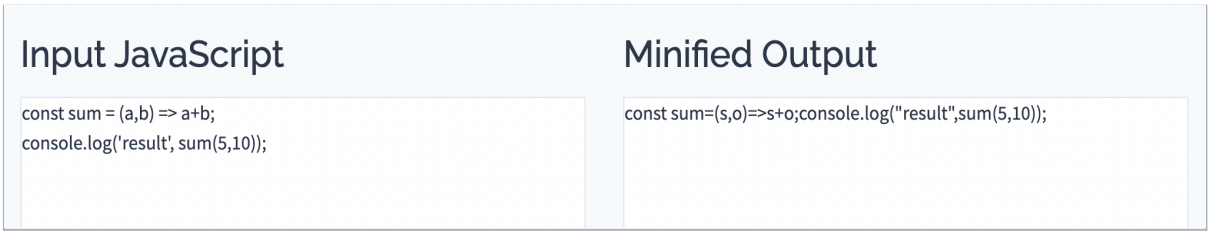
Pro změnu stránky u SPA není nutné posílat dotaz na server. Běžný routing je starost klienta, tudíž je možné změnit stránku jen v rámci lokálního chování, ale je typické načítat data v případě změny URL. Důvodem je fakt, že není vhodné načítat veškerá data, která web má k dispozici, hned na úvodní stránce. Data se načítají v závislosti na stávající URL - pro příklad lze opět uvést situaci s články. Stránka s články by měla načítat jen články a ne například informace o novinkách. Mimo to

může samozřejmě na každé stránce docházet k aktualizaci obecných informací, které jsou společné pro všechny stránky - například zda má uživatel stále platný přihlašovací token.

Důležité je také zmínit, že při přechodu na jinou stránku u statických webů dojde k “reloadu” stránky - obdoba zmáčknutí F5 v prohlížeči. To má za následek různé nevýhody - například restart session úložiště prohlížeče a horší uživatelský zážitek.

3.1.4 Minified JavaScript

Pro efektivní komunikaci mezi dvěma lokacemi po síti je vhodné, aby byl obsah zpráv co nejmenší. K tomu účelu slouží primárně různé komprimovací algoritmy - například gzip. Výsledkem takové komprimace je menší soubor, což ve výsledku znamená rychlejší přenos dat z jedné lokace do druhé.
V rámci komunikačního protokolu se obě strany domlouvají, jakým způsobem se obsah komprimuje. Typicky se domlouvají zabalením této informace do hlavičky dotazu. Pokud tedy přijímací strana nalezne header “content-encoding: gzip”, tak ví, že musí nejdříve na obsah aplikovat de-komprimovací algoritmus gzip a teprve potom je možné s daty pracovat.
Pokud se ale jedná o komunikaci, kde se posílají soubory s JavaScriptem, je možné pro redukci velikosti souborů uvažovat i nad takzvanou minifikací.



Obr. 3.3: Formátovaný JavaScript vs Minifikovaný JavaScript. Zdroj: Autor práce

Jednoduchou formou je možné říci, že minifikovaný JavaScript je upraven tak, aby byl velmi složitě čitelný lidmi. Na Obr. 3.3: Formátovaný JavaScript vs Minifikovaný JavaScript můžeme pozorovat, jakým způsobem se minifikovaný JavaScript liší od toho originálního. Na ilustraci je jen velmi jednoduchý příklad, ale s každým

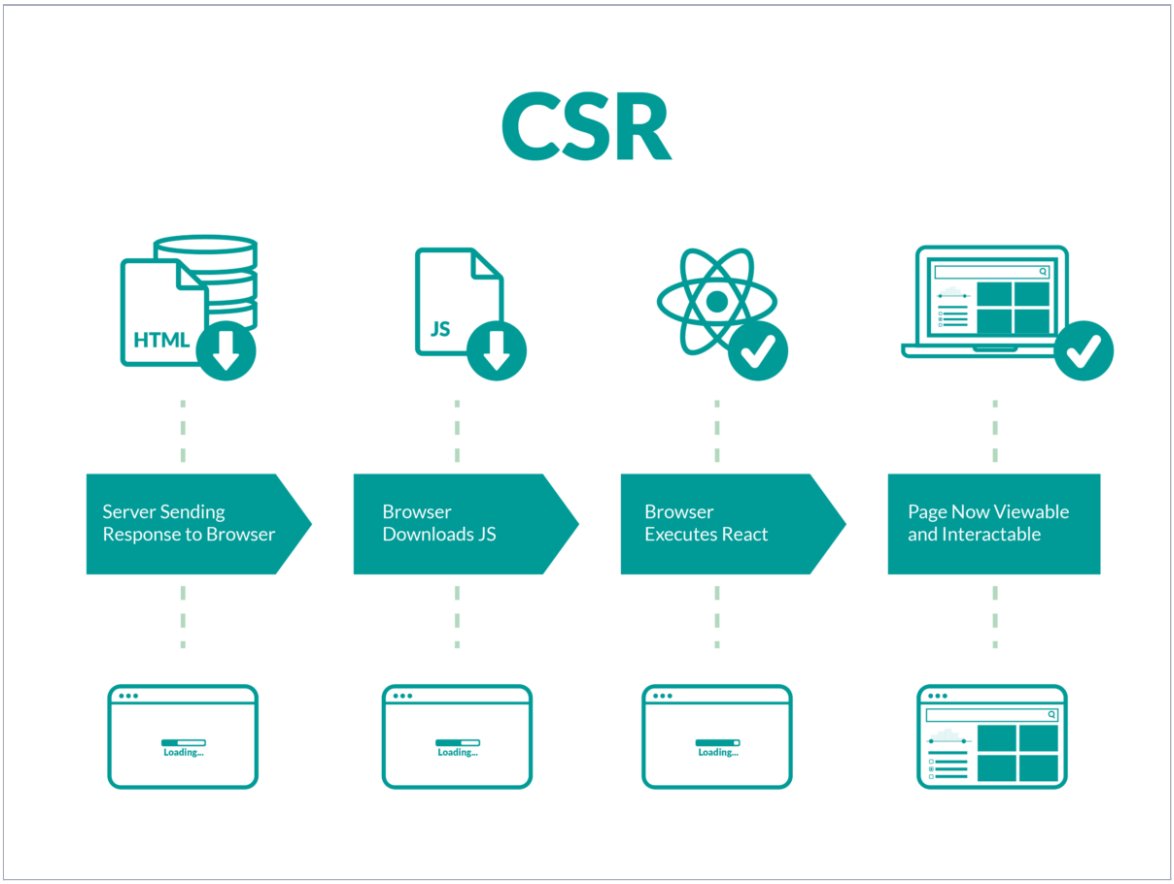
řádkem kódu přibývá komplexnost až do situace, kdy je téměř nemožné přečíst jediný řádek lidským okem.

Hlavním důvodem minifikace je, jak bylo zmíněno výše, ušetření místa. Mohlo by se zdát, že je obsah stejný, jen posunutý, ale ve skutečnosti k redukci velikosti skutečně dochází. Je to z toho důvodu, že i mezery se počítají jako validní symboly reprezentující místo na disku. Dalším důležitým krokem minifikace je takzvaný Obfuscation (zatemnění). Krok obfuscation je aplikován, aby se co do největší možné míry skryla business logika kódu. Daného skrytí je dosaženo přejmenováním a přesunutím proměnných v kódu. Mimo jiné tento krok znovu zmenší velikost výsledného souboru, ale také ztíží život těm, kteří by chtěli reverzně konstruovat cizí kód.

Minifikace dokáže v některých případech snížit velikost souboru až o 90% oproti originálu.

3.1.5 Client side rendering - CSR

V rámci SPA aplikací jsou k dispozici různé trendy, ale mezi ty nejpopulárnější se řadí způsob, kterým se obsah renderuje/zobrazuje na obrazovce klienta. Tím zdaleka nejvíce používaným je takzvaný Client side rendering (dále jen CSR) [9].



Obr. 3.4: CSR. Zdroj: Client-side rendering vs. server-side rendering: which one is better. rockcontent [online]. Jun. 17, 2020 [cit. 2022-04-09]. Dostupné z: https://rockcontent.com/blog/client-side-rendering-vs-server-side-rendering/

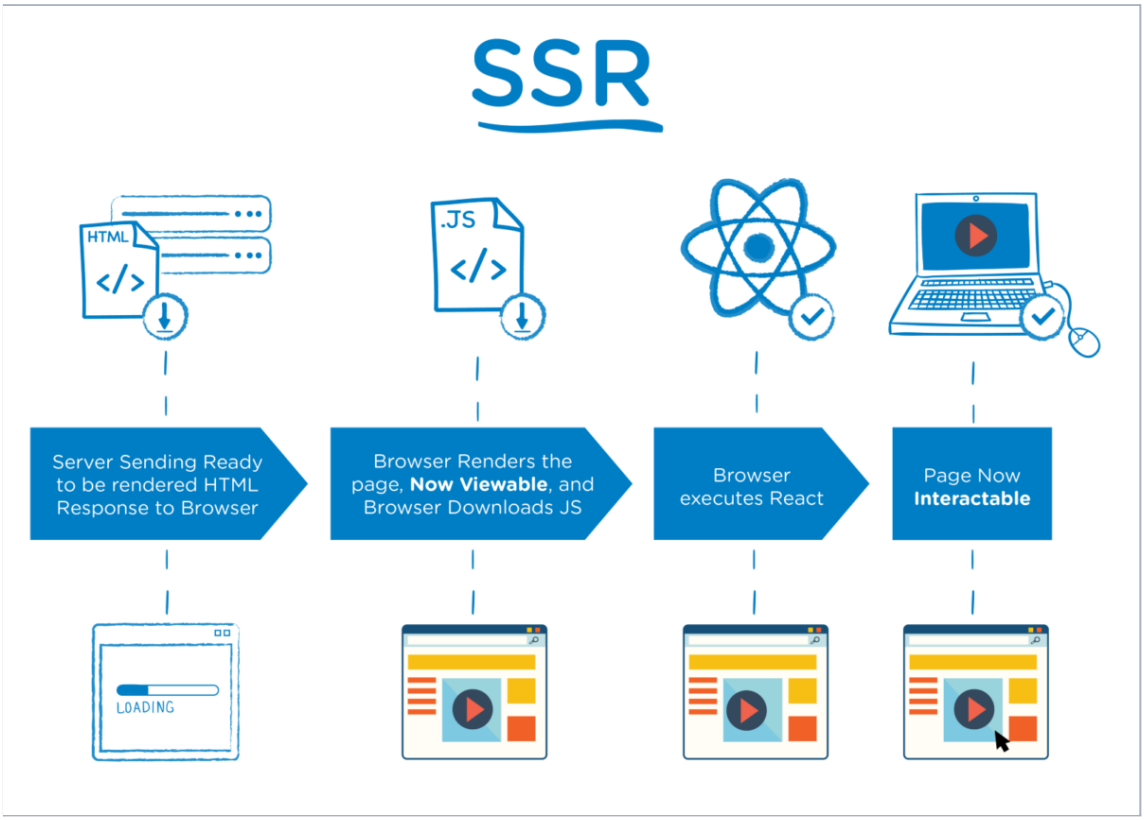
CSR lze ve své podstatě rozdělit na několik základních kroků. Prvním je navázání komunikace mezi serverem a klientem. Server posílá klientovi html soubor, který obsahuje potřebné linky. Nicméně v tomto kroku je `body` html souboru stále prázdné. Soubor obsahuje jen hrubou strukturu a je nutné načíst další soubory (js, obrázky, css). V tomto okamžiku má u sebe klient (prohlížeč) potřebné odkazy na statické soubory na serveru a posílá další dotazy statické soubory, které jsou na

serveru. Jakmile dostane prohlížeč “druhou” dávku souborů, práce samotné aplikace začíná a až tehdy je vidět obsah stránky.

Kontrastně oproti jiným způsobům renderování je důležité, že je obsah stránky viditelný až po načtení veškerého JavaScriptu, včetně obsahu, který načítá samotný JavaScriptu až v prohlížeči (například články na stránce, kde se nacházíte).

3.1.6 Server side rendering - SSR

Dalším z populárních metod jak zobrazit obsah v prohlížeči je takzvaný Server side rendering (dále jen SSR).



Obr. 3.5: SSR. Zdroj: Client-side rendering vs. server-side rendering: which one is better. rockcontent [online]. Jun. 17, 2020 [cit. 2022-04-09]. Dostupné z: https://rockcontent.com/blog/client-side-rendering-vs-server-side-rendering/

Oproti CSR se tato metoda primárně liší ve způsobu, kterým se načítá obsah html souboru. Kdežto v CSR se obsah do body načítá až po načtení samotného JavaScriptu, v SSR je tento obsah poslán už v úvodním dotazu na html soubor.

Takového chování lze docílit tím, že se potřebný obsah načte na straně serveru a obsah se tedy nemusí načítat až na straně klienta. Nicméně ke správnému fungování včetně validní autorizace klienta je nutné přidat extra vrstvu abstrakce a využití má tato metoda převážně u SEO senzitivních webů.
SEO je anglicky Search Engine Optimizations a znamená to vylepšení efektivity vyhledávacích enginů jako je třeba Google. Nicméně search enginy se stále vylepšují a SEO rozdíly mezi CSR a SSR weby se nadále zmenšují.
Hlavním benefitem SSR je tedy user experience. Pokud má klient pomalé připojení, tak hned po úvodním dotazu vidí základní strukturu stránku. Tato struktura sice nemá k dispozici žádnou interakci s DOM elementy, ale oproti CSR, kde je k dispozici jen prázdná stránka, je vizuálně intuitivnější [9].

3.1.7 Static site generation - SSG

Další, relativně letitou, ale v posledních letech opět hojně používanou metodou renderování obsahu v prohlížeči je Static site generation (dále jen SSG).
Tato metoda je velmi podobná SSR - podobná je v tom ohledu, že se veškerý obsah stránky renderuje už na straně serveru. Nicméně oproti SSR je tu jeden velký rozdíl. SSG stránky jsou všechny dopředu připravené - proto název static. SSG popisuje proces konstrukce stránek, které se renderují už při “build” fázi aplikace - build time. Výsledkem je kompletní html soubor, který má v sobě zakomponovanou základní funkcionalitu i stylování. Je to tedy velmi rychlý a efektivní způsob, kterým lze zobrazit obsah na webu. Nicméně to není vhodný způsob pro renderování stránek, které mají rozsáhlý a dynamický obsah. Proto je normou tyto přístupy kombinovat a na straně serveru načítat jen některé části webu.

3.2 Konkrétní SPA technologie

V dnešní době každým dnem vychází nová technologie napsaná v JavaScriptu, pomocí které lze obsluhovat a renderovat SPA na serveru. Nicméně stejně jako mají lidé v Česku jasného favorita ve výběru auta, stejně tak je ve světě SPA pár oblíbenců.

React.js

React byl vyvinut společností Facebook (dnes Meta) a jeho původní verze byla vydána v roce 2013. Od té doby mu interní tým Facebooku věnoval opravdu hodně péče a byly vydány různé verze, které přinesly spoustu změn, díky kterým framework našel místo v srdcích spousty vývojářů. V dnešní době se objektivně jedná o číslo jedna ve vývoji webu, co se JavaScriptu týče. Díky tomu pro něj existuje široký ekosystém podpůrných knihoven a je proto vhodný jak pro menší, tak velké projekty [10].

Vue.js

Vue byl vyvinut ex-zaměstnancem Googlu - Evanem You v roce 2014. Jedná se o framework, který se typicky těší popularity u startupů a obecně menších projektů. Nicméně dle průzkumu komunity v posledních 3 letech popularita frameworku vzrostla a ke konci roku 2020 byla vydána nová stabilní verze 3.0 [10].

Angular

Angular je framework, který byl vyvinut pod střechou Googlu a jedná se o nejstarší framework ze všech 3 zmíněných v této práci. První verze byla vydána v roce 2010 a nesla jméno Angular.js, kde chyběla nativní podpora TypeScriptu[2]. V roce 2016 došlo k vydání nové verze, která již plně podporovala Typescript. Angular nachází využití převážně v korporátních prostředích díky svému stáří a dlouhodobé podpoře [10].

[2] TypeScript je nadstavba nad jazykem JavaScript, která jej rozšiřuje o statické typování

4 WebAssembly (WASM)

WebAssembly, také wasm, případně WA, je webový standard, který definuje binární formát a umožňuje na webových stránkách (webových prohlížečích) spustit strojový kód.
Umožňuje běh aplikací téměř stejně rychle, jako nativní strojový kód. Představuje tedy doplněk k JavaScriptu, který nabízí zrychlení kritických částí webu, na které JavaScript není nejlepší volbou. Díky binární formě je tedy možné vyvíjet webové aplikace pro prohlížeč i v jiných jazycích než je JavaScript. Standard je momentálně pod křídly organizace W3C (webový standard) a je vyvíjen inženýry z firem Mozilla, Microsoft, Google a Apple [11].

4.1 Úvod do WASM

Pro pochopení principu a využití WASM je v první řadě nutné pochopit jakým způsobem procesor interpretuje kód, který píšeme v tradičních (pro lidi srozumitelných) programovacích jazycích.

4.1.1 Strojová instrukce

Strojová instrukce označuje kódovaný příkaz, který vyvolá elementární = základní logickou operaci, kterou je procesor počítače přímo schopen vykonat.

Každý počítač (jeho procesor) dokáže zpracovávat jen určitý soubor konkrétních instrukcí - závislost na architektuře. V dnešní době jsou naštěstí zavedeny standardy a většina počítačů sdílí způsob, jakým instrukcím rozumí.

Instrukce se většinou skládá z dvou základních částí - první částí je samotný kód operace. Ten určuje, jaká akce se má udělat. Druhou částí je adresa (někdy i více adres), která specifikuje, s jakými daty se má operace provést. Analogicky by tato myšlenka mohla být přirovnána k funkci a parametrům, které určují chování této funkce.

Instrukce jsou zapsány čísly (typicky v dvojkové nebo šestnáctkové soustavě), což je hlavním důvodem, proč je strojový kód relativně nesrozumitelný pro člověka. Z

toho důvodu se v něm manuálně programuje jen zcela výjimečně a existují speciální programy - tzv. compilers = česky překladače, které mají za úkol přeložit programy napsané ve vyšším programovacím jazyce do jazyka nižšího. Například program v C++ právě do strojového kódu.

Strojové instrukce se dělí podle určení na 3 základní typy: přesunové, aritmeticko-logické a řídící. Mezi přesunové lze zařadit například alokaci paměti, aritmeticko-logické mají na starost práci s logickými daty - počty s čísly, rozhodování pravda-nepravda a řídící mají spravovací charakteristiku.

Každá instrukce má za úkol vyvolání základní aktivity, kterou je potřeba na počítači vykonat. Tyto instrukce se skládají v sekvence a vzniká tak zvaný strojový kód. Jméno strojový lze interpretovat tak, že v "textových" příkazech doslova vidíme, co se děje (nebo má dít) s hardwarem počítače - strojem [12].

- **`add r0, r2;`** – sčítání – přičte do registru r0 hodnotu uloženou v registru r2
- **`addc r1, r3;`** – sčítání s přenosem – přičte do registru r0 hodnotu registru r1 a příznaku přenosu C
- **`mov 1234h, r0;`** – přesun – uloží do paměti na adresu 1234h hodnotu z registru r0
- **`mov 1236h, r1;`** – přesun – uloží do paměti na adresu 1236h hodnotu z registru r1
- **`mov [r7], r0;`** – přesun – uloží na adresu určenou registrem r7 hodnotu z registru r0
- **`cmp r4, r5;`** – porovnání – porovná hodnoty registrů r4 a r5
- **`jmpr cc_UGT, 8100h;`** – podmíněný skok – pokud byla hodnota registru r4 vyšší, pokračuje program na adrese 8100h

Obr. 4.1: Strojové instrukce Siemens SAB. Zdroj: Strojové instrukce. Wikipedia [online]. [cit. 2022-04-09]. Dostupné z: https://cs.wikipedia.org/wiki/Strojov%C3%A1_instrukce

Na Obr. 4.1 výše můžeme pozorovat název a popis několika ukázkových strojových instrukcí, které jsou navrženy pro architekturu procesoru Siemens.

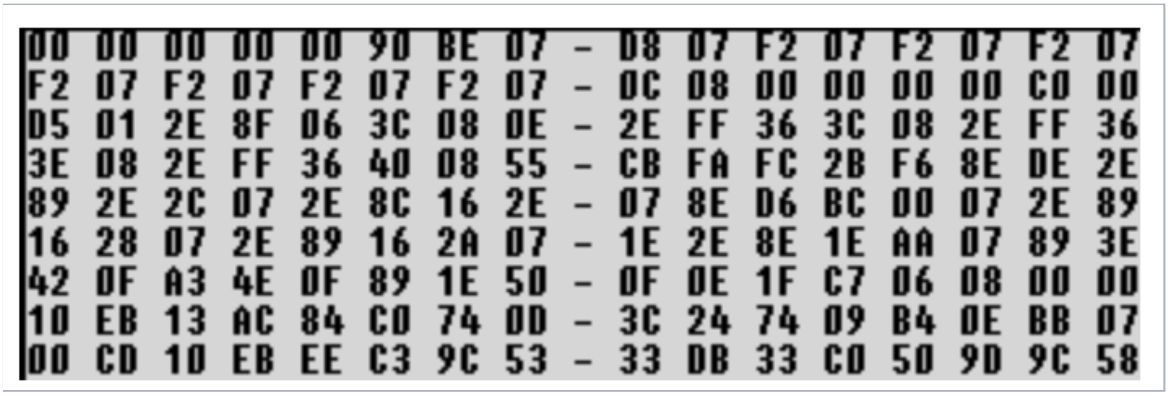
4.1.2 Strojový kód

Posloupnost strojových instrukcí se označuje jako strojový kód. Jak již bylo zmíněno v předchozí kapitole o strojových instrukcích, různé procesory mají různé sady strojových instrukcí. To znamená, že program s konkrétním strojovým kódem nebude nutně fungovat sám od sebe na druhém počítači. Některé procesory nicméně podporují více sad strojových instrukcí a kompatibilita je tak poměrně rozsáhlá. V dnešní době například většina osobních počítačů podporuje sadu x86,

což je hromadné označení pro podporu sad x86-16, IA-32 a x86-64 - jinými slovy 16, 32 a 64 bitové systémy - typicky systémy s procesorem od firmy Intel. Další populární architektura je například ARM, která je často spojována s produkty od firmy Apple a AMD, ale nejedná se o výhradní produkt těchto společností. Je to jen architektura procesoru, která je využívána v různých odvětvích [13].

V roce 2022 jsou 64bitové systémy považovány za průmyslový standard a jsou hlavním technologickým směrem počítačů, chytrých telefonů a dalších zařízení. K přechodu na tyto systémy došlo až přibližně v roce 2012, protože do té doby nebylo potřeba využívat v osobních zařízeních 64 bit dlouhých datových typů [14],[15].

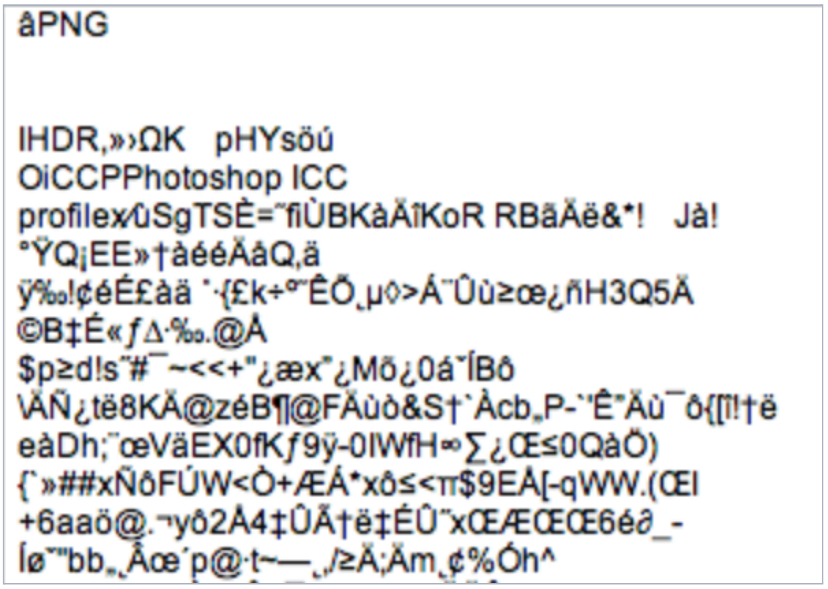
Pro vizuální ilustraci strojového kódu můžeme pozorovat Obr. 4.2: Strojový kód počítače s procesorem Intel. Na ilustraci je vyobrazena sada strojových instrukcí, která je určena pro procesor Intel. Tyto instrukce jsou tedy pro daný procesor exkluzivní.



Obr. 4.2: Strojový kód počítače s procesorem Intel. Zdroj: Od strojového kódu k programovacím jazykům [online]. [cit. 2022-04-09]. Dostupné z: https://www.fi.muni.cz/usr/jkucera/pv109/sl5.htm

4.1.3 Binární kód

Binární kód se nachází v binární souboru. Je to soubor, který se skládá z dat určených pro další zpracování počítačovým programem. To jinými slovy znamená, že kód samotný nelze hned spustit a musí se nejprve určitým způsobem interpretovat.



Obr. 4.3: PNG obrázek otevřený v textovém editoru. Zdroj: Binary vs text files [online]. Dostupné z: https://fileinfo.com/help/binary_vs_text_files

Binárními se tyto soubory nazývají proto, že obsahují čísla v dvojkové = binární soustavě, což odpovídá jedničkám a nulám. Jako binární se běžně označují ty soubory, které nemohou být jednoduše zobrazeny textovým editorem (notepad, vim atd.). Příkladem souborů, které textový editor nedokáže přečíst mohou být např. spustitelné programy obsahující strojový kód, hudba či obrázky - viz Obr. 4.3: PNG obrázek otevřený v textovém editoru.

Když se binární soubor pokusíme otevřít v textovém editoru, každou skupinu osmi bitů (kombinace osmi jedniček a nul) uvidíme typicky přeloženou jako jeden znak. Bude se velice pravděpodobně jednat o nesmyslné uspořádání náhodných textových znaků. Některé aplikace mohou dokonce soubor začít číst řadu barev - obrázek - což bude opět ve výsledku nesmyslné. Z toho důvodu je nutné před čtením binárního souboru znát jeho určení a jak se má interpretovat. Znalost této

interpretace spočívá v tom, že se ví, co přesně znamená konkrétní sekvence 8 bitů - tomu pomáhá hlavička binárního souboru.
Hlavička binárního souboru může data popisovat. Konkrétně se jedná o takzvaná metadata. Tyto aditivní informace se typicky nachází v záhlaví souboru a určují jaká data se v souboru nachází = jakým způsobem se mají interpretovat.

U některých formátů (binárních souborů) je všeobecně známo, jakou mají strukturu a jakým způsobem se mají číst, ale u některých to známé být nemusí - různé softwary mají vlastní binární formáty.

Mezi binární soubory, které mají známý formát, se řadí typicky multimediální soubory - například `GIF, JPEG, MPEG` atd. Dalším příkladem může být formát `DOC/DOCX`, který je spojován s editorem Microsoft Word. Tento konkrétní formát je převážně známý, ale interpretace není kompletní, proto formát často nemá dokonalý překlad v jiných kancelářských programech jako je třeba open source Open Office nebo Pages od firmy Apple.

Nicméně je důležité si uvědomit, že na úrovni filesystému je textový a binární soubor totéž - soubor dat (jedniček a nul) na disku. Primární rozdíl je v tom, že textový soubor obsahuje posloupnost bytů, která tvoří text v nějaké znakové sadě (ASCII, Unicode atd.) a binární soubor obsahuje typicky složitější věci, které musí být interpretovány specifickým způsobem.
Dalším typem souborů jsou pak soubory spustitelné, které nepotřebují interpretovat pomocí jiného programu. [16], [17]

4.2 WA jako binární formát

Jak bylo zmíněno v úvodu do WA, webassembly je standard, který definuje binární formát a umožňuje ve webovém prohlížeči spustit strojový kód. Další pasáž se věnuje problematice vytváření formátů, kterým rozumí prohlížeče s podporou pro WA standard.

4.2.1 Aplikace v binárním formátu

Abychom mohli použít WA v prohlížeči, musíme mít aplikaci v takzvaném binárním formátu. Tento formát získáme tím, že aplikaci nejdříve do binárního formátu

zkompilujeme. K datu psaní této práce existuje již celá řada jazyků, která nabízí podporu pro kompilaci do binárního formátu, nicméně ne všechny jazyky tuto možnost nabízí nativně.
Příkladem těch, které to umí nativně jsou například C/C++, C# nebo Rust.
Kontrastně taky ale existují jazyky, případně nadstavby nad některými jazyky, které používají specifické compilery, které WASM standard zaměřují aditivně [18].

4.2.2 Načtení WA v prohlížeči

Moderní prohlížeče dle úvodu kapitoly dokáží přeložit a přečíst aplikace v binárním formátu, ale abychom je mohli používat, je nutné tyto soubory nejdříve inicializovat v rámci samotného html souboru [19].
Pro ilustraci uvažujme funkci answer (viz Výpis kódu 4.1), která je napsána v jazyku C. Funkce se nachází v souboru **answer.c**

```
int answer() {
   return 42;
}
```

Výpis kódu 4.1: Funkce answer v jazyku C. Zdroj: What is WebAssembly. StackPath [online]. Dostupné z: https://www.stackpath.com/edge-academy/what-is-webassembly

V kontextu načtení dané funkce do prohlížeče je nutné tento soubor a jeho obsah zkompilovat a přeložit do souboru formátu .wasm - tedy WebAssembly. V případě jazyku C bereme v potaz například compiler emscripten.
S vygenerovaným souborem answer.wasm lze přejít ke spuštění v prohlížeči. K tomu je nutné vytvořit nový objekt ve scopu objektu window a následně ho vyvolat. K tomuto účelu mají všechny moderní prohlížeče implementované API, které práci s touto oblastí usnadňují.

Pro samotné načtení .wasm souborů je nutná inicializace pomocí JavaScriptu - uvnitř script tagu (viz Úvod do SPA - JavaScript).

```
fetch('answer.wasm').then(response =>
 response.arrayBuffer())
 .then(bytes =>
 WebAssembly.instantiate(bytes))
 .then(result =>
 alert(result.instance.exports.answer()));
```

Výpis kódu 4.2: Import wasm souboru do JavaScriptu. Zdroj: What is WebAssembly. StackPath [online]. Dostupné z: https://www.stackpath.com/edge-academy/what-is-webassembly

Na ukázce výše (Výpis kódu 4.2) je demonstrován způsob, pomocí kterého se načte soubor answer.wasm a následně se ukáže číslo 42 pomocí vestavěné funkce alert prohlížeče.

Podobným způsobem je také možné využívat funkce prohlížeče přímo v nativním kódu. Takové chování je možné díky takzvaným napojením (bindings) mezi jazyky. Jedná se tedy o umožnění high-level interakcí mezi moduly Wasm a JavaScriptem.

4.3 Rozdělení WA aplikací

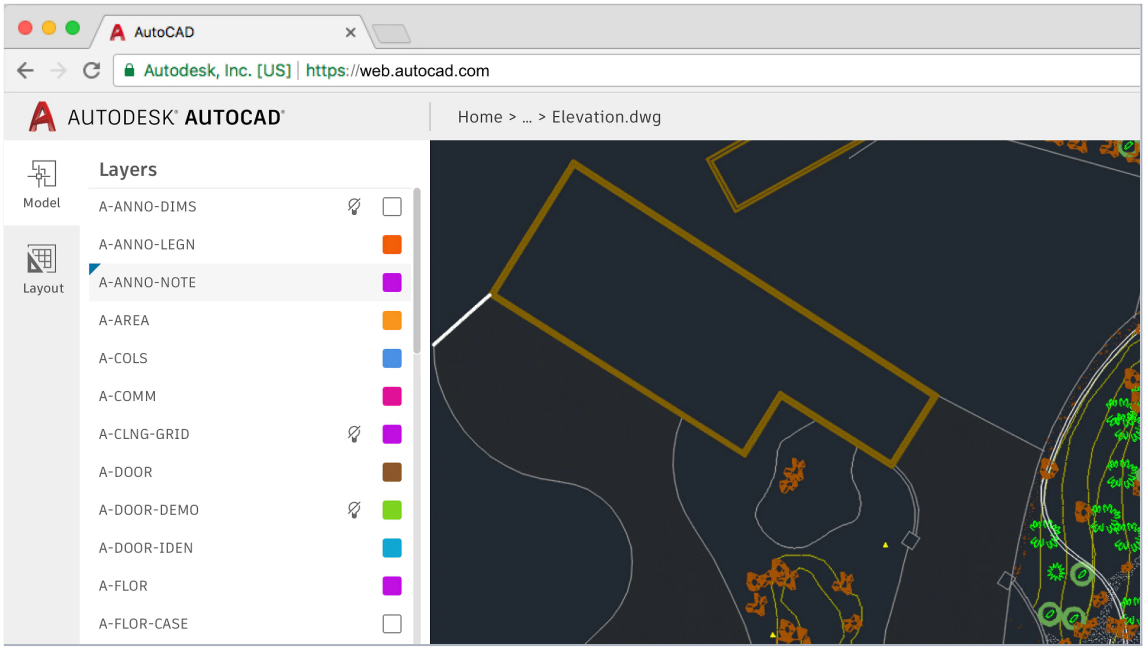
WebAssembly je široký a obecný koncept a dá se s ním uvažovat nad celou řadou aplikací.

Z pohledu softwarových firem a individuálních vývojářů je mnoho důvodů, proč jít cestou WA, ale tyto důvody lze z pohledu vývojáře zjednodušit na několik primárních směrů.

4.3.1 Výkonnost

V případě výkonnosti se ke slovu dostávají aplikace, u kterých je zmíněná výkonnost na prvním místě. Mezi takové je možné zařadit například software na rozpoznávání obrazu, živou editaci médií, fyzikální simulace, CAD (Computer-Aided Software), případně pokročilé 3D hry.

Jedná se tedy o typy softwaru, které ke svému chodu využívají velké množství výpočetního výkonu a pro tyto účely není JavaScript díky své implementaci vhodnou volbou. Historicky bylo nutné takové aplikace nainstalovat na pevný disk počítače, ale dnes je možné spustit danou aplikaci v prohlížeči online - viz Obr. 4.4: AutoCAD Online.



Obr. 4.4: AutoCAD Online. Zdroj: AutoCAD Web App. Made with WebAssembly [online]. Dostupné z: https://madewithwebassembly.com/showcase/autocad/

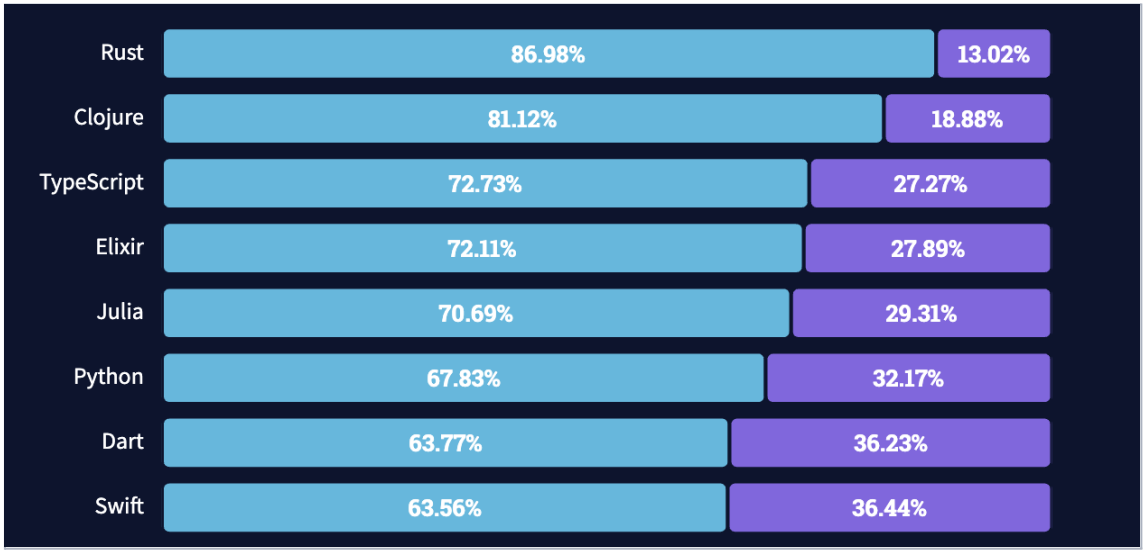
4.3.2 Stabilita a bezpečnost

Vývoj softwaru v JavaScriptu, ať už jde o webovou stránku nebo službu na backendu, bývá tradičně doprovázen celou řadou nástrojů, která má za úkol samotný proces vývoje usnadnit a dále předejít drobným, ale i větším chybám v produkci. Jedním z takových nástrojů je například Typescript - nadstavba, která přidává do světa JavaScriptu možnost typování. Nadstavba byla vyvinuta společností Microsoft a svojí implementací se v některých ohledech podobá moderním jazykům z rodiny C. Dá se také říci, že se u rozsáhlejších projektů stal Typescript standardem.

Byť má tedy JavaScript za opaskem různé nástroje, které bezpečnost a stabilitu vylepšují, nedají se v tomto ohledu stále porovnat s jinými low-level jazyky, které jsou typované staticky a zároveň typicky kompilované. Jako příklad lze uvést Rust - jazyk, který spravuje paměť pomocí modelu vlastnictví. Na rozdíl od JavaScriptu, kde neodpovídající typy proměnných pro běh aplikace nevadí, se v Rustu nesrovnalosti dostanou na povrch už při kompilaci programu a není možné aplikaci spustit.

4.3.3 Developer experience

Byť je JavaScript velmi používaným a chtěným jazykem, dle průzkumů StackOverflow se nejedná se o jazyk, který byl v posledních letech tím nejpopulárnějším jazykem mezi vývojáři.



Obr. 4.5: Průzkum nejvíce milovaných a nenáviděných jazyků 2021. Zdroj: 2021 Developer survey. StackOverflow [online]. Dostupné z: https://insights.stackoverflow.com/survey/2021

Na Obr. 4.5 výše jsou v jednoduchém grafu zobrazeny ty nejvíce milované a zároveň nenáviděné programovací jazyky v roce 2021. Hodnota vlevo u každého řádku značí procento vývojářů, kteří daný jazyk již použili v rozsáhlém hledisku a chtějí jazyk používat nadále. Hodnota vpravo naopak značí procento recipientů, kteří jazyk nadále používat nechtějí. V grafu nejsou zobrazeny všechny hodnoty, z původních dat byly vybráno jen několik prvních hodnot, přibližně do úrovně oblíbenosti 60 procent. Z dat je tedy možné odvodit, že je zážitek vývoje webových aplikací například již ve zmíněném Rustu statisticky přívětivější než samotný JavaScript.

4.4 SPAs pomocí WebAssembly

Byť je WA k dispozici přibližně od roku 2017, je pro tvorbu SPA pomocí klasických jazyků k dispozici relativně omezené množství nástrojů. Je dále důležité uvést, že

většina existujících řešení založených na klasických jazycích a kompilací do WA nemá plnou podporu a není považována za takzvaně production ready. Nicméně existují i řešení, které se naopak za production ready považují a jsou v některých případech hojně využívány [18]. Právě takové technologie budou dále blíže popsány.

4.4.1 WA technologie pro tvorbu SPA

Blazor

Mezi jednu z production ready technologií lze zařadit Blazor. Jedná se o framework určený pro vývoj s jazykem C# a rozsáhlou platformou .NET. Dále existuje varianta Blazor Server, která replikuje SSR praktiky - analogicky identické k JavaScript řešením [23].

Blazor typicky nachází využití v případech, kdy se backendové služby vyvíjí pomocí technologie .NET a dává tedy smysl psát i frontend ve stejném jazyku.

Blazor je založen na syntaxu, kterému se říká Razor (viz Výpis kódu 4.3: Blazor kód). Razor umožňuje vkládání kódu založeného na .NET platformě do HTML kódu určeného pro webové stránky. Syntaxe samotná se tedy skládá ze speciálních Razor značek, jazyka C# a HTML. Ukázkový kód demonstruje jednoduchou aplikaci, která zobrazuje v paragrafu `<p>` počet, který lze pomocí tlačítka `<button>` navyšovat.

```
<h1>Blazor code example</h1>
<p>count: @count</p>
<button class="btn btn-primary" @onclick="IncCount">Click to
increment</button>

@code {
    private int count = 0;
    private void IncCount()
    {
        count++;
    }
}
```

Výpis kódu 4.3: Blazor kód. Zdroj: ASP.NET Core Blazor [online]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/blazor/?view=aspnetcore-6.0

Yew

Dalším příkladem WA technologie, pomocí které lze vytvářet Single Page Aplikace a pomocí které je v této práci dále napsaná ukázková aplikace, se jmenuje Yew.

Yew je framework napsaný nad jazykem Rust a je určený pro vytváření vícevláknových webových aplikací.

Jedním ze základním prvků syntaxu Yew je makro, které umožňuje deklarovat interaktivní html pomocí Rust výrazů. Analogicky je toto chování možné přirovnat syntaxu JSX, který je původně spojený s JavaScript frameworkem React (viz Výpis kódu 4.4: Yew kód).

```
use yew::prelude::*;

#[function_component]
fn App() -> Html {
   let counter = use_state(|| 0);
   let onclick = {
       let counter = counter.clone();
       move |_| {
           let value = *counter + 1;
           counter.set(value);
       }
   };

   html! {
       <div>
           <button {onclick}>{ "+1" }</button>
           <p>{ *counter }</p>
       </div>
   }
}

fn main() {
   yew::Renderer::<App>::new().render();
}
```

Výpis kódu 4.4: Yew kód. Zdroj: docs.rs [online]. Dostupné z: https://docs.rs/yew/0.17.4/yew/

Ukázkový kód stejně jako u příkladu Blazoru po kompilaci zobrazí aplikaci, která má v UI 2 prvky. Prvním je číslo reprezentující počet kliků a druhým tlačítko, kterým lze počet zvyšovat.

Mezi další klíčové prvky frameworku patří vysoká optimalizace render cyklů (aktualizace DOMu[3]) a jednoduchá **integrace JavaScriptových knihoven z npm**.

4.4.2 Integrace JavaScriptu v nativním kódu

Nativní aplikace nebyly historicky stavěné s myšlenkou, aby je bylo možné pouštět v prohlížeči. Nicméně tato realita se díky WA změnila a vznikla poptávka po nových rozšířeních, které komunikaci s prohlížečem umožňují. Konkrétně se jedná o funkčnost, která umožňuje napojení na API prohlížeče. Prohlížeč má totiž k dispozici různé funkce, které jsou momentálně exkluzivní pro JavaScript, který je s ním úzce spojený. Jako příklad lze uvést funkci alert, která má za úkol zobrazit upozornění v popup okně. Dalším příkladem může být přístup k objektu document, který je v prohlížeči též dostupný globálně a umožňuje interakci s DOMem. V neposlední řadě lze zmínit API, které umožňuje přepínat fullscreen režim obrazovky.

Problém je tedy obecný pro všechny aplikace, které chtějí komunikovat s prohlížečem, případně využít prvky JavaScriptu a zároveň jsou napsané v jiném jazyce než je JavaScript.

Jelikož je dále v praktické části důraz kladen primárně na Rust s frameworkem Yew a protože pro tuto obecnou problematiku existují v různých jazycích různá řešení, bude v následujících kapitolách probírán způsob propojení JavaScriptu jen pro jazyk Rust. Nicméně řešení vazeb JavaScriptu v jiných jazycích jsou typicky podobné tomu pro Rust.

Wasm-bindgen

Wasm bindgen (korektně wasm-bindgen) je knihovna a hojně využívaný nástroj, který poskytuje interakce a napojení mezi moduly Wasm a JavaScriptem [20].

Je napsán v jazyce Rust a je tedy taktéž určen pro použití na projektech, který využívají Rust jako programovací jazyk. Například již zmíněný framework Yew využívá tento nástroj pro interakci s prohlížečem a napojení interních funkcí.

[3] DOM (Document Object Model) je objektově orientovaná reprezentace XML nebo HTML dokumentu

Jednoduché využití této knihovny můžeme pozorovat na ukázkách níže (viz Výpis kódu 4.5, 4.6). Na první ukázce (Rust kód) se nejprve importuje funkce alert z prohlížeče. Dále je v témže souboru definována funkce **greet**, která využívá již importované funkce alert. Klíčovým slovem **pub** je funkce greet označena pro export a využití externími soubory. V následujícím (druhém) obrázku (JavaScript kód) je funkce greet importována a spuštěna. Tímto jednoduchým principem byla tedy zajištěna komunikace a propojení mezi dvěma jazyky.

```rust
use wasm_bindgen::prelude::*;

// Import funkce `window.alert` do Rustu
#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

// Export funkce `greet` do JavaScriptu
#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {}!", name));
}
```

Výpis kódu 4.5: Rust využití wasm-bindgen. Zdroj: Wasm-bindgen [online]. Dostupné z: https://github.com/rustwasm/wasm-bindgen

```javascript
import { greet } from "./hello_world";

greet("World!");
```

Výpis kódu 4.6: JavaScript využití wasm-bindgen. Zdroj: Wasm-bindgen [online]. Dostupné z: https://github.com/rustwasm/wasm-bindgen

Byť je příklad výše jednoduchý a nereprezentuje reálné využití všech funkcí, které jsou v rámci knihovny k dispozici, nabízí náhled do principů, na kterých jsou WA SPA frameworky postaveny.

Pokud by se jednalo o komplexní příklad, kde je struktura projektu rozložena na mnohdy desítky různých souborů, je nutné představit další řadu nástrojů, které mají za úkol tyto souboru uspořádat a vytvořit hierarchii zkompilovaných a

minifikovaných souborů. Takové soubory je dále možné představit serveru, který je umí odesílat na požadavek (request) klienta.

4.4.3 Bundlery (nejen) pro WA

Jak již bylo zmíněno v kapitole Načtení WA, pro spuštění WA v prohlížeči je nutné načíst potřebné soubory pomocí komplementárních .js souborů, které se odkazují na zkompilované .wasm soubory. Jelikož by ale u větších projektů bylo velice obtížné všechny soubory správně importovat a spravovat, existují právě pro tento účel takzvané bundlery.

Bundlery mají primárně za úkol efektivně sledovat a shromažďovat (dále jen bundlovat) všechny moduly a závislosti (např. externí knihovny) aplikace. Tyto moduly jsou transformovány na statické soubory (assety) připravené pro odesílání na serveru. Bundlery se typicky využívají, když se aplikace přesouvá do produkční formy, případně se spouští na lokálním serveru pro účely vývoje. Mezi další typické charakteristiky lze v rámci bundlerů zařadit i například takzvaný code splitting. Code splitting zefektivňuje komunikaci mezi serverem a klientem tím, že se potřebné soubory rozdělí na více takzvaných chunků a není nutné mít pro úvodní načtení aplikace k dispozici veškerý kód, který se v aplikaci nachází.

Webpack

Webpack je jeden z nejpopulárnějších bundlerů s primárním zaměřením na JavaScript. Umí ale pracovat i s jinými soubory určenými pro front-end. Jedná se o volně dostupný a open-source nástroj a mezi jeho přednosti patří možnost používat externí doplňky (pluginy) [24]. Tyto doplňky jsou často vytvořené komunitou a v kontextu Rustu se jedná například o **wasm-pack-plugin**. Plugin pro webpack, který umožňuje bundlování právě Rust kódu .

Parcel

Parcel je bundler s hlavním zaměřením pro webové aplikace a pyšní se mimo jiné svým developer-friendly postojem. Od svých konkurentů se dle autorů liší ve své jednoduchosti implementace v projektech a také samotnou rychlostí exekuce [25].

Trunk

Trunk je bundler webových aplikací WASM napsaných v jazyku Rust. Ve svém jádru spoléhá na jednoduchou konfiguraci, pomocí které dále umí sdružovat WASM kód, JavaScript a další statické soubory. Mimo bundlování se od ostatních alternativ liší tím, že sám od sebe obsahuje podporu pro spuštění development serveru [26].

Jedná se o jeden z doporučených bundlerů pro projekty, které v jádru využívají framework Yew (viz kapitola SPAs pomocí WebAssembly - Konkrétní WA technologie).

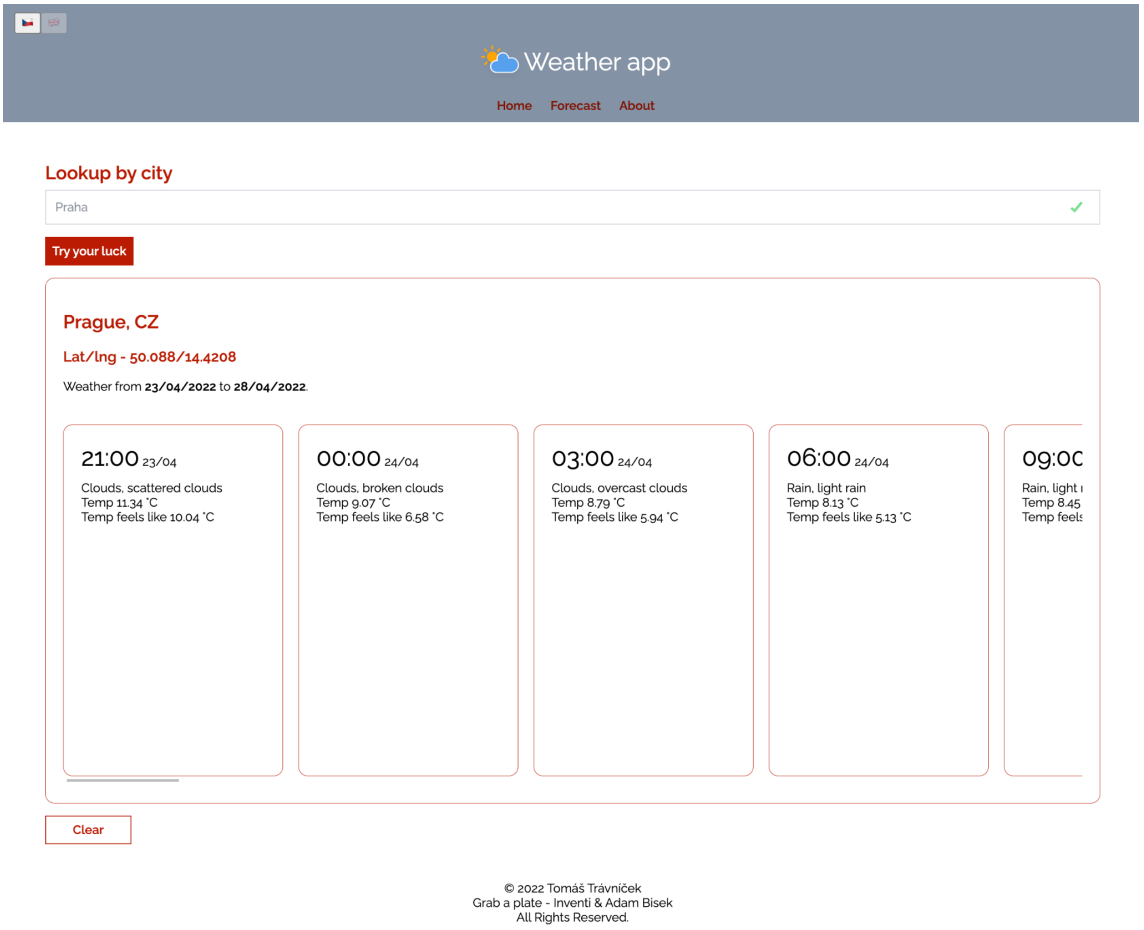
5 Srovnání vybraných technologií pro tvorbu SPA

5.1 Společné téma pro aplikace

Při výběru typu aplikace byl kladen důraz na princip SPA - aplikace mají více stránek, sjednocenou správu interního stavu, a v neposlední řadě komunikují s externím serverem - provolávání API.

V rámci těchto kritérií bylo pro aplikace zvoleno ukázkové téma - předpověď počasí. Aplikace obsahují 3 hlavní routy. První je domovská stránka s uvítáním uživatele. Druhá je stránka se samotnou předpovědí počasí, kterou je možné vyhledávat pomocí názvu města v různých jazycích a poslední stránkou je stránka o autorovi a projektu samotném.

5.2 Aplikace s technologiemi React a Typescript



Obr. 5.1: UI React aplikace. Zdroj: Autor práce

Ekosystém JavaScriptu a konkrétně Reactu je mohutný a je tedy pro vývoj SPA možné vybírat z velkého množství různých kombinací knihoven a nástrojů. Nicméně pro práci na ukázkové React aplikaci byly zvoleny převážně technologie, které jsou často využívány na velkých projektech, mají rozsáhlou podporu komunity a jsou v některých případech vnímány jako technologický standard ve vývoji JavaScriptových SPA.

Pro úvodní sestavení projektu bylo využito šablony (boilerplate), která již obsahuje veškeré potřebné knihovny a je tedy snadné aplikaci začít hned vyvíjet. Použitý boilerplate je **Grab a plate**, který byl interně vyvinut ve firmě INVENTI Development s.r.o.

Boilerplate jako základ pro bundlování aplikace využívá nástroj **Razzle**. Razzle je bundler určený primárně pro vývoj React aplikací, ale umí pracovat i jinými JS frameworky. Plně podporuje SSR a Hot Module Replacement (změny v aplikaci při vývoji se hned připíší na development server). Ve svém jádru spolehá na Webpack a jedná se tedy o nadstavbu, která práci s Webpackem usnadňuje. Na ilustrace níže (viz Výpis kódu 5.1: Soubor razzle.config.js) můžeme pozorovat jak vypadá nastavení pro nástroj razzle.

```
const alias = {
 '@app': path.resolve(__dirname, 'src/App'),
 '@components': path.resolve(__dirname, 'src/components'),
 '@utils': path.resolve(__dirname, 'src/utils'),
 // ... aliasy pro zjednodušené importy
}

module.exports = {
 alias, // pro externí využití, např. storybook
 modifyWebpackConfig({
   env: {
     target,
     dev, // typ buildu aplikace (bool)
   },
   webpackConfig, // webpack config
   webpackObject, // webpack node module
   options: {
     pluginOptions,
     razzleOptions,
     webpackOptions,
   },
   paths, // upravené paths pomocí aliasu
 }) {
   // aliases
   webpackConfig.resolve.alias = {
     ...webpackConfig.resolve.alias,
     ...alias,
   }

   // code splitting
   if (target === 'web') {
     const filename = path.resolve(__dirname, 'build')
     webpackConfig.plugins.push(
       new LoadableWebpackPlugin({
         outputAsset: false,
         writeToDisk: { filename },
       }),
     )
   }
   }

   // .. další nastavení
 }
}
```

Výpis kódu 5.1: Soubor razzle.config.js. Zdroj: Autor práce

Dále byla pro správu stavu aplikace zvolena robustní knihovna **Redux**, která je doprovázena knihovnou **Redux Saga**. Redux je centralizovaný kontejner, který v sobě drží interní stav aplikace v takzvaném storu. Data ve storu je možné měnit pomocí akcí (dispatch actions), které obsahují název akce a payload = data pro

změnu. Redux saga má následně za úkol spravovat vyslané redux akce pomocí postranních efektů (side effects). Kombinace těchto knihoven umožňuje relativně jednoduše a elegantně udržovat strukturovanou a čitelnou architekturu kódu. Na obrázku níže (viz Výpis kódu 5.2: Setup redux storu) je k dispozici ukázka kódu, která Redux včetně ság inicializuje.

```
export default (initialState: object, history: History): Store => {
 const sagaMiddleware = createSagaMiddleware()
 const middlewaresToApply = [sagaMiddleware, ...createMiddlewares(history)]
 const enhancers = [applyMiddleware(...middlewaresToApply)]
 const store = createStore(createReducers(history), initialState,
composeEnhancers(...enhancers))
 sagaMiddleware.run(rootSaga)
 return store
}
```

Výpis kódu 5.2: Setup redux storu. Zdroj: Autor práce

JSX

Pro psaní React aplikací se typicky využívá syntax JSX. Jedná o se rozšíření JavaScriptu, které rozumí kombinaci html elementů, vnořených dat a React komponent. Syntax byl historicky exkluzivní pro React, ale dnes ho lze najít i v jiných technologiích, například Vue.js (viz kapitola Konkrétní SPA technologie). Na ilustraci níže (viz Výpis kódu 5.3: Root aplikace) můžeme pozorovat samotné jádro renderování aplikace, které obsahuje abstraktní komponenty, které do sebe zanořují další komponenty. Takovému vztahu se říka parent - children. Například nejvyšší komponenta `FaultBarrier` začíná tagem `<FaultBarrier>` a končí až tagem `</FaultBarrier>`. Všechny komponenty mezi těmito dvěma tagy tedy lze tedy označit jako children `FaultBarrier` komponenty.

```
const App = (): JSX.Element => (
 <FaultBarrier>
   <LinguiProviderRedux>
     <ThemeProvider theme={defaultTheme}>
       <Layout>
         <ScrollToTop />
         <Switch>
           {routesArray.map(([key, route]) => (
             <Route key={key} {...route.extraProps} path={route.route.path}
render={route.render} />
           ))}
           <RouteStatus statusCode={404}>
             <Route component={Error404} />
           </RouteStatus>
         </Switch>
       </Layout>
     </ThemeProvider>
   </LinguiProviderRedux>
 </FaultBarrier>
)
```

Výpis kódu 5.3: Root aplikace. Zdroj: Autor práce

Komplexnější příklad JSX můžeme pozorovat na komponentě WeatherCard, která má za úkol zobrazit kartu se základními informacemi z objektu weather - viz Výpis kódu 5.4: Weather komponenta. Stylování je zajištěno pomocí knihovny x-styled-components, která umožňuje specifikovat styly přímo v html elementech.

```
const WeatherCard = ({ weather }: Props) => {
 const { city, list } = weather
 const listStartIdx = 0
 const listEndIdx = list.length - 1
 const dateFrom = dayjs(list[listStartIdx].dt_txt).format('DD/MM/YYYY')
 const dateTo = dayjs(list[listEndIdx].dt_txt).format('DD/MM/YYYY')

 return (
   <Card>
     <x.h2>
       {city.name}, {city.country}
     </x.h2>
     <x.h3>
       Lat/lng - {city.coord.lat}/{city.coord.lon}
     </x.h3>

     <x.span>
       <Trans
         id="weatherCard.dateFromTo"
         message="Weather from {from} to {to}."
         values={{ from: <strong>{dateFrom}</strong>, to: <strong>{dateTo}</strong> }}
       />
     </x.span>
     <x.div display="flex" overflowY="auto" mt={10}>
       {list.slice(listStartIdx, listEndIdx).map((entry) => {
         const date = dayjs(entry.dt_txt)
         return (
           <Card key={entry.dt} minWidth="250px" minHeight="400px" mr={5} mb={3}>
             <x.div mb={3}>
               <x.span fontSize="2rem">{date.format('HH:mm')}</x.span>
               <x.span> {date.format('DD/MM')}</x.span>
             </x.div>
             {entry.weather.map((w) => (
               <div key={w.id}>
                 {w.main}, {w.description}
               </div>
             ))}

             <div>
               <Trans id="weatherCard.temp" message="Temp {temp} °C" values={{ temp:
entry.main.temp }} />
             </div>
             <div>
               <Trans id="weatherCard.tempFeelsLike" message="Temp feels like {temp} °C"
values={{ temp: entry.main.feels_like }} />
             </div>
           </Card>
         )
       })}
     </x.div>
   </Card>
 )
}
```

Výpis kódu 5.4: Weather komponenta. Zdroj: Autor práce

Dále byla v aplikaci využita podpora více jazyků, konkrétně angličtina a čeština. Pro tuto funkčnost je využito knihovny **lingui**. Lingui je nástroj, který umožňuje pomocí

specifických html tagů (viz Trans - Výpis kódu 5.4) a souborů obsahující katalogy (viz Výpis kódu 5.5, 5.6) přepínat aplikaci mezi různými jazyky.

```
#: pages/Forecast/Forecast.tsx:29
msgid "forecast.heading"
msgstr "Vyhledávání podle města"
```

Výpis kódu 5.5: Lingui český překlad . Zdroj: Autor práce

```
#: pages/Forecast/Forecast.tsx:29
msgid "forecast.heading"
msgstr "Lookup by city"
```

Výpis kódu 5.6: Lingui anglický překlad . Zdroj: Autor práce

5.3 Aplikace s technologiemi Rust a Yew

Obr 5.2: UI Yew aplikace. Zdroj: Autor práce

Jak již bylo zmíněno v kapitole SPAs pomocí WebAssembly - Konkrétní technologie, vývoj SPA pomocí WA je v porovnání s JavaScriptem relativně nová záležitost a je tedy logické, že i výběr podpůrných technologií pro tvorbu takových aplikace bude omezený. Z toho důvodu byly použité technologie vybrány primárně s ohledem na dostupnost a podporu, ale také aby svými vlastnostmi a využitím byly v některých ohledech podobné technologiím, které byly využity v React aplikaci výše.

```
{
 "author": "Tomáš Trávníček",
 "name": "yew-app",
 "version": "0.1.0",
 "scripts": {
   "build": "cross-env WASM_PACK_PROFILE=release parcel build
static/index.html --public-url ./",
   "start": "cross-env WASM_PACK_PROFILE=dev parcel static/index.html -p
8000 --open",
   "test": "wasm-pack test --headless --chrome"
 },
 "devDependencies": {
   "cross-env": "^7.0.3",
   "parcel-bundler": "^1.12.5",
   "parcel-plugin-wasm.rs": "^1.3.0",
   "sass": "^1.45.1",
   "typescript": "^3.7.4"
 }
}
```

Výpis kódu 5.7: Soubor package.json Zdroj: Autor práce

Pro úvodní vytvoření projektu bylo využito šablony Create Yew App, která ve svém jádru spoléhá na bundler Parcel (viz kapitola SPAs pomocí WebAssembly - Bundlery (nejen) pro WA) a wasm-bindgen pro napojení JavaScriptu v WA kódu. V souboru package.json (viz Výpis kódu 5.7) se nachází základní informace o projektu, skripty, které je možné spouštět pomocí parcelu a následně závislosti na externí knihovny, které jsou pro běh aplikace nezbytné.

Díky Parcelu a knihovně wasm-bindgen je tedy v projektu zajištěna komunikace mezi Rustem a JavaScriptem. Dále je ale také nutné definovat samotnou Rust aplikaci a související Rust závislosti. K tomu účelu u Rustu a konkrétně frameworku Yew slouží soubor Cargo.toml. Analogicky je velmi podobný souboru package.json,

ale svými definicemi je zaměřený čistě na knihovny s Rust kódem. Viz Výpis kódu 5.8: Soubor Cargo.toml

```
[package]
authors = ["Tomáš Trávníček"]
version = "0.1.0"
... informace o projektu

[lib]
crate-type = ["cdylib", "rlib"]

[dependencies]
chrono = { version = "0.4", features = ["serde"] }
log = "0.4"
serde = "1"
reqwest = { version = "0.11", features = ["json"] }
yew = "0.19.3"
yew-router = "0.16.0"
yew-hooks = "0.1"
wasm-bindgen = "0.2"
wasm-logger = "0.2.0"
wee_alloc = "0.4.5"
stylist = { version = "0.10", features = ["yew_integration"] }
dotenv_codegen = "0.15.0"
thiserror = "1"

[dependencies.web-sys]
version = "0.3"
features = [
 "Document",
 "Element",
 "HtmlCollection",
 "HtmlInputElement",
]

[dev-dependencies]
wasm-bindgen-test = "0.3.14"
gloo-utils = "0.1.0"

[dev-dependencies.web-sys]
version = "0.3"
features = [
 "Document",
 "Element",
 "HtmlCollection",
 "HtmlInputElement",
]
```

Výpis kódu 5.8: Soubor Cargo.toml Zdroj: Autor práce

Aplikace obsahuje několik hlavních souborů, které specifikují základní chování a funkcionalitu. Prvním je takzvaný entry point aplikace, který slouží k definici všech využívaných modulů (viz klíčové spojení **pub mod** - Výpis kódu 5.9: Yew entry point) a následně inicializaci samotné webové aplikace - řádek **yew::start_app::<App>();**.

```
pub mod app;
pub mod components;
pub mod error;
pub mod routes;
pub mod services;
pub mod types;

use wasm_bindgen::prelude::*;

use app::App;

// Specifikace WASM formátu `wee_alloc`
#[global_allocator]
static ALLOC: wee_alloc::WeeAlloc = wee_alloc::WeeAlloc::INIT;

// Začátek WASM aplikace
#[wasm_bindgen]
pub fn run() -> Result<(), JsValue> {
   wasm_logger::init(wasm_logger::Config::default());
   yew::start_app::<App>();
   Ok(())
}
```

Výpis kódu 5.9: Yew entry point. Zdroj: Autor práce

Pro funkcionalitu dynamických stránek je v aplikaci implementováno router řešení yew-router. Jedná se o komplementární knihovnu frameworku Yew, která je definována jako peer závislost v seznamu využívaných knihoven (viz Obr. 34: Soubor Cargo.toml). Dále můžeme ukázce Výpis kódu 5.10: Yew routes pozorovat, jakým způsobem jsou individuální routy definované. Každá cesta má jeden parametr **at**, který značí na které url se bude stránka nacházet. Následně je hned po názvu definovaná komponenta, která se na url vyskytuje.

```rust
// Dostupné routy aplikace
#[derive(Routable, Debug, Clone, PartialEq)]
pub enum AppRoute {
    #[at("/")]
    Home,
    #[at("/forecast")]
    Forecast,
    #[at("/about")]
    About,
    #[not_found]
    #[at("/page-not-found")]
    PageNotFound,
}

/// Switch dostupných routes
pub fn switch(routes: &AppRoute) -> Html {
    match routes.clone() {
        AppRoute::Home => html! { <Home /> },
        AppRoute::Forecast => html! { <Forecast /> },
        AppRoute::About => html! { <About /> },
        AppRoute::PageNotFound => html! { "Page not found" },
    }
}
```

Výpis kódu 5.10: Yew routes. Zdroj: Autor práce

Struktura renderovaného obsahu u frameworku Yew spolehá na takzvané makro **!html**. Makra jsou obecný pojem spojený s jazykem Rust a umožňují definovat unikátní klíčová slova, která zaobalují rozsáhlou funkcionalitu do jednoho termínu. V případě makra !html se jedná o funkcionalitu, která umí pracovat s html elementy, daty a dalšími komponentami - viz Výpis kódu 5.11: Root aplikace. Svojí implementací je velmi podobný syntaxu JSX, které se váže s JavaScriptem a převážně Reactem.

```
/// Root
#[function_component(App)]
pub fn app() -> Html {
   html! {
       <BrowserRouter>
           <Layout>
               <Switch<AppRoute> render={Switch::render(switch)} />
           </Layout>
       </BrowserRouter>
   }
}
```

Výpis kódu 5.11: Root aplikace. Zdroj: Autor práce

Na ukázce výše (Výpis kódu 5.11) můžeme pozorovat, jakým způsobem je router "zanořený" ve stromu všech komponent. Nejdříve je nutné jako rodiče uvést BrowserRouter, který poskytne pro `Switch` níže potřebné data je formě properties (dále jen **props**).

Props jsou ve své podstatě data (vlastnosti, funkce, proměnné …), které jsou poskytnuty rodičem svému potomku - child. Například u komponenty button můžeme zpozorovat, že jsou k dispozici props `on_click`, `disabled` a `children` (viz Výpis kódu 5.12: Komponenta button).

```rust
#[derive(Properties, PartialEq)]
pub struct ButtonProps {
    pub on_click: Callback<MouseEvent>,
    pub disabled: bool,
    pub children: Children,
}

#[styled_component(Button)]
pub fn button(props: &ButtonProps) -> Html {
    html! {
        <button
            type="button"
            onclick={props.on_click.clone()}
            disabled={props.disabled}
            class={css!(r#"
                box-sizing: border-box;
                display: inline-flex;
                align-items: center;
                justify-content: center;
                min-width: 200px;
                padding: 0.2rem 0.5rem;
                margin-left: 0.3rem;
                border-style: solid;
                border-width: 1px;
                border-color: gray;
                border-radius: 10px;
                background-color: white;
                font-weight: bold;
                transition: 0.3s;
                &:hover:enabled {
                    background-color: gray;
                    color: white;
                    border-color: white;
                    cursor: pointer;
                }
            "#)} >
        { for props.children.iter() }
        </button>

    }
}
```

Výpis kódu 5.12: Komponenta button. Zdroj: Autor práce

`On_click` prop značí funkci, která je provolána zpětně k rodiči v případě, že je na dané tlačítko "kliknuto". Prop `disabled` dle svého názvu označuje vlastnost, zda je tlačítko vypnuté, případně zapnuté. A v neposlední řadě komponenta Button nabízí prop `children`. Tato prop značí potomky, které se mají dále předat jako potomek samotné Button komponentě. V realitě to značí html element, který má být

zobrazen jako text tlačítka (viz “Load”, Výpis kódu 5.13: Využití Button komponenty).

```
<Button on_click={on_submit} disabled={!can_submit}>{ "Load" }</Button>
```

Výpis kódu 5.13: Využití Button komponenty. Zdroj: Autor práce

Yew pro stylování aplikace uvádí jako standardní řešení používání statických CSS, případně SCSS[4] souborů, které jsou umístěny ve veřejných (public) složkách aplikace. V takovém případě jsou styly importovány jako jednoduché odkazy v hlavním html souboru aplikace. Nicméně existují i komplexnější řešení, které generují CSS třídy za běhu, případně během kompilace aplikace a jsou následně injectovány s unikátními názvy tříd. Tento přístup umožňuje psát styly přímo v rust kódu, takzvaně css-in-rust. Jedná se o termín odvozený ze zkratky css-in-js, která má analogicky identický význam ve světě JavaScriptu.

S ohledem na dostupné možnosti bylo pro stylování základních elementů stránky a dalších dílčích komponent využito externí knihovny **stylist**. Jedná se o řešení, které se řadí mezi již zmíněné css-in-rust nástroje a svojí implementací se snaží napodobit css-in-js knihovnu **styled-components**, která je napsaná v JavaScriptu a je hojně využívána v kontrastních JavaScriptových webových aplikacích.

Stylování je demonstrováno na ukázce (viz Výpis kódu 5.14), kde je styl aplikován přímo na element `div`. Styl je použit na atribut class pomocí makra **css!**, které v běhu aplikace vygeneruje soubor se styly, adekvátními názvy tříd a které dále přiřadí právě danému div elementu.

[4] SCSS (Sassy CSS) je nadstavba stylování CSS s podporou pro komplexnější chování a využití proměnných

```
#[styled_component(Forecast)]
pub fn forecast() -> Html {
   let query_handle = use_state(|| String::from(""));
   let query = query_handle.deref().clone();

   let query_fetch = {
       let query = query.clone();
       use_async(async move { query_by_city(query).await })
   };
   let on_submit = {
       let state = query_fetch.clone();
       Callback::from(move |_| state.run())
   };

   let handle_input = { Callback::from(move |val: String|
query_handle.set(val)) };

   let loading = query_fetch.loading;
   let error = query_fetch.error.clone();
   let placeholder = String::from("Enter city ...");
   let can_submit = query.len() > 3;
   html! {
       <LoadingBoundary is_loading={loading} error={error}>
           // Input
           <div class={css!(r#"
               width: 100%;
               margin-top: 2vh;
               display:flex;
               justify-content: center;
           "#)}>
               <Input value={query.clone()} on_input={handle_input}
placeholder={placeholder} />
               <Button on_click={on_submit} disabled={!can_submit}>{ "Load"
}</Button>
           </div>
           // Loaded
           {
               if let Some(forecast) = &query_fetch.data {
                   html! { <WeatherCard weather={forecast.clone()}/> }
               } else {
                   html! {}
               }
           }
       </LoadingBoundary>
   }
}
```

Výpis kódu 5.14: Kontejner-komponenta Forecast. Zdroj: Autor práce

5.4 Obecné srovnání, využití

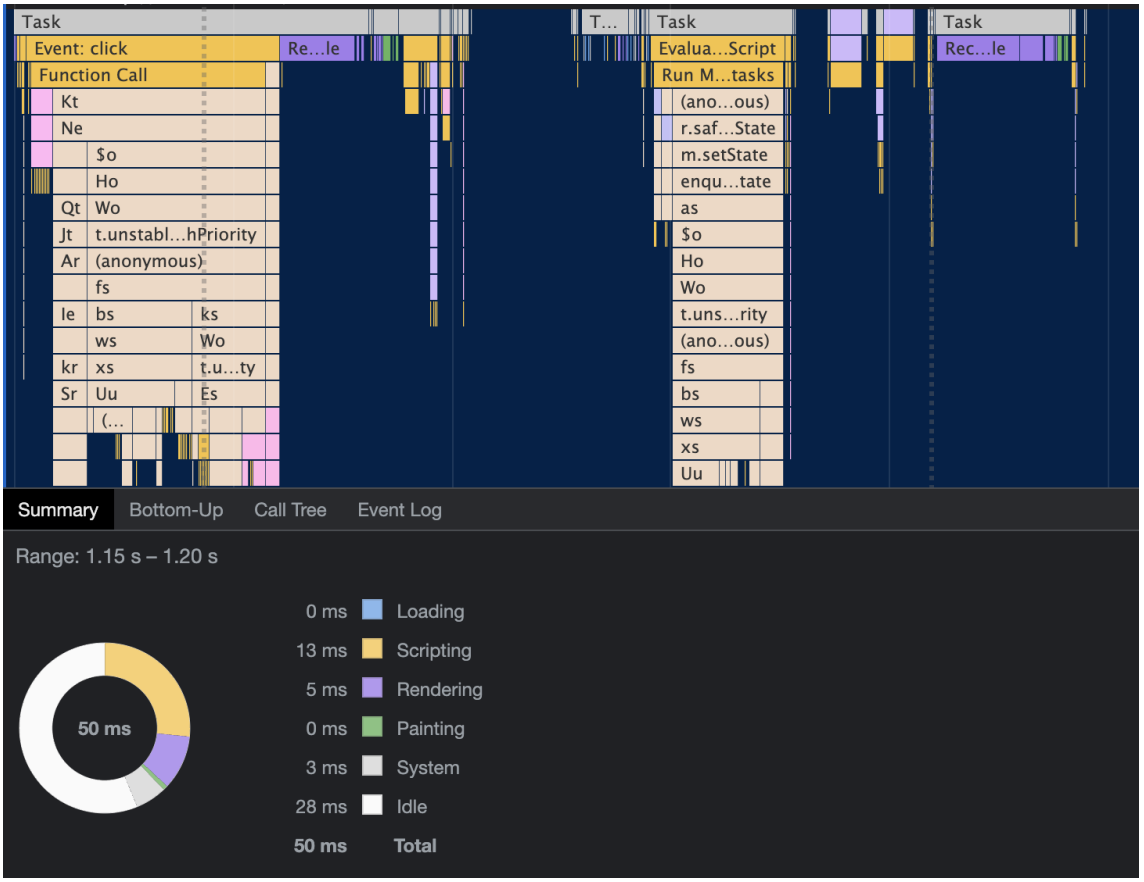
5.4.1 Výkon

Z hlediska výkonu (performance) lze uvažovat nad dvěma přístupy, pomocí kterých je možné měřit výkon SPA aplikací.

UI Změny

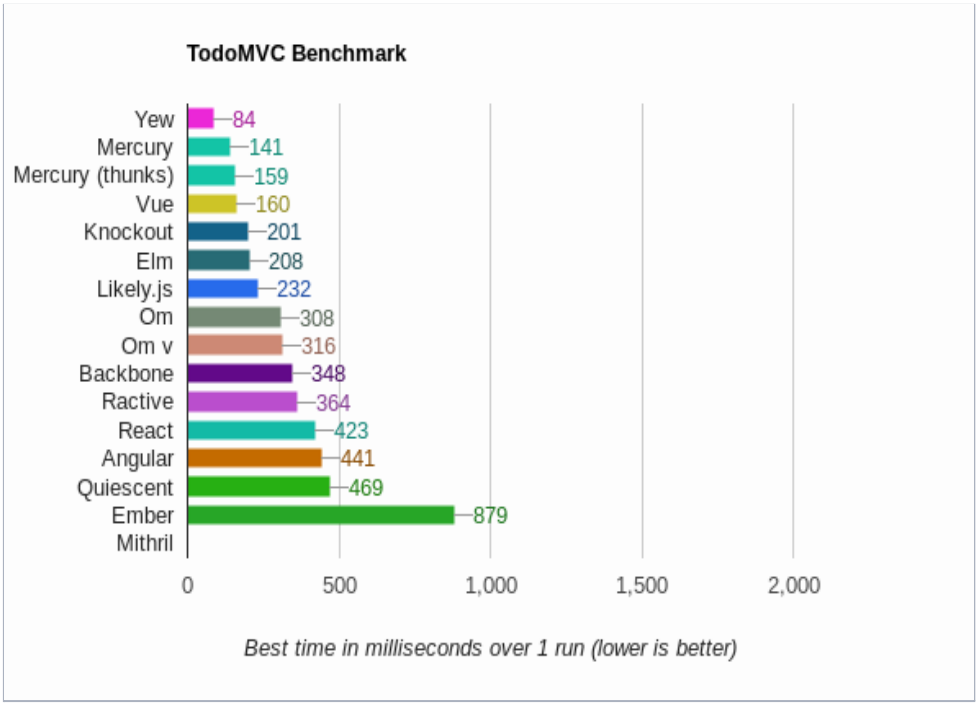
Prvním významným způsobem měření výkonu SPA je měření doby změn UI v závisloti na změnách interního stavu aplikace. Jedná se tedy o situaci, kdy už jsou typicky načtena veškerá data z externího serveru a je tedy veškerá zodpovědnost na klientovi. Z tohoto hlediska lze využít různé benchmarky, které často simulují nereálné podmínky, kdy se na obrazovku dostane obrovské množství html elementů a je dále měřeno, jak dlouho trvá tuto změnu zpracovat.

Jelikož je ale v reálném světe relativně neefektivní zobrazovat velké množství dat v jednom seznamu, je v takových případech využíváno takzvané stránkování a k renderu extrémně dlouhých seznamů nedochází. Podobných principů bylo využito i na ukázkových aplikacích výše a z toho důvodu jsou výsledky v načítání obsahu na jednotlivých stránkách velice podobné u obou typů aplikací - viz Obr. 5.3 a Obr. 5.4. Obě ilustrace demonstrují situaci, kdy dochází k přechodu mezi stránkami pomocí routeru a je dále zobrazen seznam obsahující informace o projektu. Byť aplikace napsaná pomocí WebAssembly (Obr. 5.4) byla přibližně o 30ms dříve responzivní, jedná se o hodnotu, kterou v reálném použití uživatel téměř nepozná. Nicméně naměřená data odpovídají jiným nezávislým benchmarkům provedeným komunitou - viz Obr. 5.5: Benchmark TodoMVC. Benchmark pracoval právě s již zmíněným případem dlouhého seznamu a bylo měřeno jak rychle dokáže daná technologie zpracovat a propsat změny. Framework Yew byl ze všech dostupných technologií nejrychlejší.



Obr. 5.3: React přechod mezi stránkami. Zdroj: Autor práce

Obr. 5.4: Yew přechod mezi stránkami. Zdroj: Autor práce

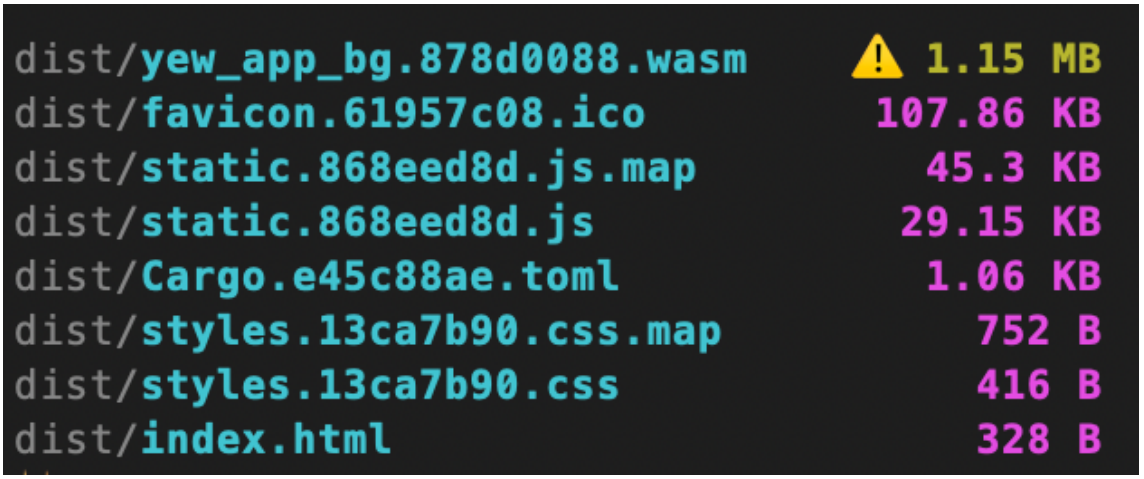


Obr. 5.5: Benchmark TodoMVC. Zdroj: TodoMVC Performance comparison [online]. Dostupné z: https://github.com/DenisKolodin/todomvc-perf-comparison

Load time

Druhým významným způsobem měření výkonu SPAs je měření doby, jak dlouho aplikaci trvá, než se stává responzivní při úvodním načtení. Této době se říká load time. Load time aplikace je vlastnost, kterou lze ovlivnit širokou řadou proměnných a okolností, začínaje například rychlostí připojení klienta, samotného počítače, ale také architekturou projektu. Jelikož ale při vývoji aplikace nelze rychlost připojení uživatelů ovlivnit, je pozornost věnována primárně zmíněné architektuře. V případě WebAssembly aplikací je totiž typicky k dispozici jeden velký soubor, který obsahuje stěžejní chování aplikace a aby bylo možné s aplikací interagovat, je nutné tento soubor načíst. Na Obr. 5.6 můžeme pozorovat strukturu již zkompilovaného projektu v produkční podobě a jedná se tedy o strukturu, která bude vyžádána klientem ze serveru. Hned prvním souborem v ilustraci je zmíněný entry point wasm aplikace, který má samotný 1.15 MB. Byť se nejedná o největší soubor, jsou to stále data, které je nutné přenést až ke klientovi, než se aplikace stane responzivní. V porovnání s aplikacemi, které spoléhají jen na JavaScript a u kterých

existují automatické code splitting řešení, je ve světě Rustu a WASM relativně komplexní a komplikovaná záležitost takový problém řešit.



Obr. 5.6: Yew struktura statických souborů. Zdroj: Autor práce

5.4.2 Výhody, nevýhody WA, JS

Výhody JS

JavaScript a jeho použití pro vývoj webových aplikací je dominantní technologií už téměř dvě desetiletí a díky tomu je k dispozici široké spektrum nástrojů, které vývoj takových aplikací usnadňují. Mezi jednu z hlavních výhod lze tedy zařadit zmíněný rozsáhlý výběr dostupných technologií a podporu komunity. Pokud by se dále porovnal vývoj SPA v JavaScriptu k vývoji SPA pomocí WA, je možné díky způsobu, kterým JavaScript funguje, očekávat, že vývoj v JavaScriptu nebude tak časově náročný. JavaScript má tedy jinými slovy vyšší write-time efektivitu. S rychlostí psaní je přímo spojená další výhoda - jednoduchost jazyku. Oproti jiným jazykům, které jsou typicky kompilovány, je JavaScript odstíněn od nízkoúrovňového programování a není tedy nutné pro jeho pochopení a vývoj řešit úroveň abstrakce navíc.

Výhody WA

Mezi primární výhody vývoje aplikací pomocí nízkoúrovňových jazyků a následné kompilace do WebAssembly binárního formátu patří bezesporu jejich statické

typování. Při psaní takových aplikací musí být vše důkladně ošetřeno a je tedy za běhu aplikace zajištěna vyšší bezpečnost. Dalším hlavním bonusem vývoje pomocí WA je samotná rychlost běhu aplikace - viz předchozí kapitoly. V neposlední řadě lze mezi výhody WA zařadit jeho interoperabilitu s JavaScriptovým světem. Jelikož je možné importovat souboru mimo ale i do kontextu WA aplikací, je naprosto validní cesta využívat pro některé části aplikace JavaScript řešení a pro jiné části aplikace spoléhat naopak na jiný jazyk. Dále lze mezi výhody WA zařadit samotnou možnost volby. Někdo je zvyklý využívat na většinu věcí jeden jazyk a díky WebAssembly má možnost psát i webové aplikace ve svém oblíbeném jazyku.

Nevýhody WA

Pozorovat nevýhody je obvykle u informačních technologií individuální záležitost, ale mezi ty objektivní lze u vývoje WA SPA jednoznačně zařadit nevyspělý ekosystém podpůrných knihoven a nástrojů. Jako příklad lze uvést framework Yew a jeho omezené možnosti, co se stylování týče. V porovnání s JavaScriptovými řešeními je k dispozici jen hrstka dostupných způsobů, pomocí kterých je možné využívat CSS. Dále lze uvést makro !html. Byť je zvýraznění syntaxu a automatické formátování kódu bráno v moderních IDE[5] jako samozřejmost, podpora pro toto makro zatím není k dispozici a je tedy nutné veškerý kód spravovat a udržovat čitelný manuálně. Mezi další nevýhodu, kterou lze považovat za kontroverzní, patří takzvaný write-time-speed. Jedná se o slovní spojení, které značí jak rychle je možné danou aplikaci vytvořit (napsat). Díky své typové bezpečnosti jsou WA aplikace “upovídané” a obsahují hodně kódu, který by se v kontrastních JS aplikacích nacházet nemusel. Byť tedy tento kód přidává na bezpečnosti, zároveň ubírá na rychlosti, kterou je aplikaci možné napsat.

[5] IDE (Integrated Development Environment) je nástroj určený pro vývoj softwaru

6 Shrnutí výsledků

Základním cílem této práce bylo prozkoumat technologii WebAssembly a dále její využití na reálných projektech, konkrétně pro tvorbu singe page aplikací. V rámci výzkumu byly nejprve představeny a prozkoumány stávající JavaScriptové řešení pro tvorbu singe page aplikací, které jsou v porovnání s WebAssembly dominantní volbou. V průběhu výzkumu bylo WebAssembly představeno jakožto binární formát, který definuje, jakým způsobem je WA reprezentováno. Mimo základní informace o WA a jeho využití pro tvorbu SPA bylo zmíněno obecné využití této technologie a jakými způsoby je ji možné z hlediska vývojářů dělit. V praktické části byly vyvinuty dvě identické aplikace. První aplikace byla napsána v JavaScriptu s nadstavbou TypeScript a frameworkem React, druhá aplikace byla kontrastně napsána v jazyku Rust s využitím frameworku Yew. Po obecném představení každé aplikace zvlášť byly implementace porovnány z hlediska výkonu a přístupu vývoje. V neposlední řadě došlo ke stručnému shrnutí výhod a nevýhod využití WebAssembly pro tvorbu SPA oproti klasickým JavaScriptovým technologiím.

7 Závěry a doporučení

Některé nejnovější technologické trendy poukazují na WebAssembly jako na plnohodnotného nástupce JavaScriptu v prohlížečích. Argumenty pro toto tvrzení jsou primárně založeny na rychlé exekuci kódu, ale rychlost není jediný faktor, který je nutné zvážit při výběru webových technologií. Byť má WebAssembly své výhody, má i své nevýhody a z toho důvodu je velice pravděpodobné, že se v následujících letech WebAssembly nestane dominantním směrem ve vývoji single page aplikací. Nicméně díky své interoperabilitě s JavaScriptem a všemi řešeními, které svět JS nabízí, má WebAssembly potenciál pro využití v komplexních situacích, kde je důležité UI, ale zároveň i výkon v některých částech aplikací. Dále je vhodným doporučením oblast vývoje WebAssembly a Single Page Aplikací nadále sledovat a podobný průzkum této práce za několik let zopakovat a pozorovat, zda se ekosystém nástrojů pro WebAssembly rozšířil.

8 Seznam použité literatury

[1] ROURKE, Mike. Learn WebAssembly: Build web applications with native performance using Wasm and C/C++. Birmingham: Packt Publishing, 2018. ISBN 1838821090.

[2] SCHERER, Justin. Hands-On JavaScript High Performance: Build faster web apps using Node.js, Svelte.js, and WebAssembly. Birmingham: Packt Publishing, 2020. ISBN 1838821090.

[3] GALLANT, Gerard. WebAssembly in Action. New York: Manning Publications, 2019. ISBN 1617295744.

[4] CRAIG, William. Future of web development. Webfx [online]. [cit. 2022-04-29]. Dostupné z: https://www.webfx.com/blog/web-design/what-is-the-future-of-web-development/

[5] NIELSEN, Emily. Static websites. GraphCMS [online]. [cit. 2022-04-29]. Dostupné z: https://graphcms.com/blog/what-is-a-static-website

[6] JS where to. W3Schools [online]. [cit. 2021-09-24]. Dostupné z: https://www.w3schools.com/js/js_whereto.asp

[7] Routing in Javascript. Medium [online]. [cit. 2021-09-27]. Dostupné z: https://medium.com/@fro_g/routing-in-javascript-d552ff4d2921

[8] Hosting static websites. PPLS [online]. [cit. 2021-09-25]. Dostupné z: http://learningtechnology.ppls.ed.ac.uk/learn/hosting-static-websites-in-learn/

[9] Client-side rendering vs. server-side rendering. Rockcontent [online]. [cit. 2022-02-14]. Dostupné z: https://rockcontent.com/blog/client-side-rendering-vs-server-side-rendering/

[10] DAITYARI, Shaumik. Angular vs React vs Vue. Codeinwp [online]. [cit. 2022-02-14]. Dostupné z: https://www.codeinwp.com/blog/angular-vs-vue-vs-react/

[11] WebAssembly. Wikipedia [online]. [cit. 2022-01-12]. Dostupné z: https://cs.wikipedia.org/wiki/WebAssembly

[12] Strojová instrukce. Wikipedia [online]. [cit. 2022-01-24]. Dostupné z: https://cs.wikipedia.org/wiki/Strojov%C3%A1_instrukce

[13] Strojový kód. Wikipedia [online]. [cit. 2022-01-15]. Dostupné z: https://cs.wikipedia.org/wiki/Strojov%C3%BD_k%C3%B3d

[14] KUČERA, Jan. Od strojového kódu k programovacím jazykům. MUNI

[online]. [cit. 2022-04-24]. Dostupné z: https://www.fi.muni.cz/usr/jkucera/pv109/sl5.htm

[15] TRIGGS, Robert. Arm vs x86. Androidauthority [online]. [cit. 2022-01-24]. Dostupné z: https://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/

[16] Binární soubor. Wikipedia [online]. [cit. 2022-01-24]. Dostupné z: https://cs.wikipedia.org/wiki/Bin%C3%A1rn%C3%AD_soubor

[17] ZEMEK, Petr. Textové vs binární soubory [online]. [cit. 2022-01-24]. Dostupné z: https://cs-blog.petrzemek.net/2015-08-26-textove-vs-binarni-soubory

[18] Made with WebAssembly [online]. [cit. 2022-02-16]. Dostupné z: https://madewithwebassembly.com/

[19] TULKA, Tomas. Running Wasm in the Browser [online]. [cit. 2022-02-15]. Dostupné z: https://blog.ttulka.com/learning-webassembly-5-running-wasm-in-the-browser

[20] Wasm-bindgen [online]. [cit. 2022-04-24]. Dostupné z: https://yew.rs/docs/next/concepts/wasm-bindgen/introduction

[21] Projects using Seed [online]. [cit. 2022-04-24]. Dostupné z: https://github.com/seed-rs/awesome-seed-rs#projects-using-seed

[22] VON MAXIMILAN, Lang. WebAssembly [online]. [cit. 2022-04-24]. Dostupné z: https://www.adesso.de/de/news/blog/why-you-should-not-use-webassembly-to-build-your-spa.jsp

[23] ASP.NET Core Blazor. Docs.microsoft [online]. [cit. 2022-04-24]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/blazor/?view=aspnetcore-6.0

[24] Webpack [online]. [cit. 2022-04-24]. Dostupné z: https://webpack.js.org/

[25] Parcel docs. Parcel [online]. [cit. 2022-04-24]. Dostupné z: https://parceljs.org/docs/

[26] Trunk [online]. [cit. 2022-04-24]. Dostupné z: https://trunkrs.dev/

9 Přílohy

- Zdrojový kód React aplikace - https://github.com/thomas4t/weather-rust-app
- Zdrojový kód Yew aplikace - https://github.com/thomas4t/weather-js-

Oskenované zadání práce

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2020/2021

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

Jméno a příjmení:	**Tomáš Trávníček**
Osobní číslo:	**I1900266**
Adresa:	Komenského 494, Sezemice, 53304 Sezemice, Česká republika
Téma práce:	Využití technologie WebAssembly pro tvorbu SPA
Téma práce anglicky:	Using WebAssembly to build SPA
Vedoucí práce:	Ing. Jakub Beneš Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Cílem bakalářské práce je prozkoumat, seznámit a rozšířit znalosti čtenáře v oblasti vývoje Single Page Aplikací (SPA). Práce si dále klade za cíl porovnat tradiční technologie pro tvorbu SPA a nově vznikající technologie WebAssembly na základě předem definovaných kritérií. Následně v praktické části práce porovnat vybrané technologie na praktických příkladech.

Osnova:

1. Úvod
2. Single Page Aplikace (SPA)
3. WebAssembly
4. Porovnání vybraných technologií pro tvorbu SPA
5. Shrnutí, výsledky, závěr Literatura

Seznam doporučené literatury:

- SCHERER, Justin. Hands-On JavaScript High Performance: Build faster web apps using Node.js, Svelte.js, and WebAssembly. Birmingham: Packt Publishing, 2020. ISBN 1838821090.
- ROURKE, Mike. Learn WebAssembly: Build web applications with native performance using Wasm and C/C++. Birmingham: Packt Publishing, 2018. ISBN 1838821090.
- GALLANT, Gerard. WebAssembly in Action. New York: Manning Publications, 2019. ISBN 1617295744.

Podpis studenta: Datum: 2.3.2021

Podpis vedoucího práce: Datum: 2.3.2021