Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Programovací jazyk Elm a jeho využití pro vývoj webových aplikací

Bakalářská práce

Autor: Martin Bureš
Studijní obor: Informační management

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Odborný konzultant: Ing. Jan Černý

Hradec Králové Duben 2023

Prohlášení:

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

V Hradci Králové dne 18.4.2023 Martin Bureš

Poděkování:

Děkuji vedoucímu bakalářské práce doc. Ing. Filipovi Malému, Ph.D. za metodické vedení práce a Ing. Janu Černému za dohled, podporu a rady při psaní práce.

Anotace

Název: Programovací jazyk Elm a jeho využití pro vývoj webových aplikací.

Bakalářská práce se v teoretické části zabývá primárně vznikem programovacího jazyka Elm, jeho přístupem k vývoji front-endové části webové aplikace a jeho vývojem v čase. Mezi dílčí cíle práce patří posouzení, zda je současná verze tohoto programovacího jazyka připravena pro využití v reálném světě a je konkurenceschopná v různých aspektech programování s jinými technologiemi a na trhu práce. Dále také přibližuje funkcionální přístup k programování aplikací a okrajově jej porovnává s objektově orientovaným přístupem. Výstupem praktické části je demonstrace teoreticky popsaných programovacích vzorů a postupů na jednoduché webové aplikaci. V závěru práce následuje shrnutí poznatků, které jsou postupně představeny v průběhu práce a zhodnocení dosažení všech stanovených cílů.

Annotation

Title: Programming language Elm and its utilization for web development.

The theoretical section of this bachelor's thesis deals with the origin of the Elm programming language, its approach to developing front-end part of web application and its evolution over time. As partial goals of this thesis it is to evaluate, whether the current version of this programming language is ready to be used in production, is able to compete with other technologies used on the market at the moment in terms of ease of use, safety of programming and demand on job market. Also, this work contains a slight comparison between functional and object-oriented approach to programming front-end applications. Practical part of this thesis is demonstration of previously described patterns and approaches in action on a simple web application. In conclusion. The conclusion of includes summarization of knowledge described throughout the thesis and evaluation of all established goals.

Obsah

Seznam obrázků

Seznam výpisů kódu

1 Úvod

Bakalářská práce se zabývá technologií s názvem Elm, za kterou nestojí korporátní firma, ale jeden programátor, který ji vdechl život. Tato práce čtenáře postupně provede od historie, přes současnou situaci vzhledem ke konkurenci a k trhu práce, až po demonstraci nabytých poznatků na reálné front-endové aplikaci.

Motivací autora k vypracování této práce byla především pozitivní pracovní zkušenost s touto technologií. Autor měl v minulosti možnost si také vyzkoušet i jiné front-endové technologie a svou přímou zkušenost a porovnání také popisuje v teoretické části. Dalším faktorem, který přispěl k tvorbě této práce, je zajímavý příběh, který tuto technologii doprovází.

V praktické části následuje převedení teoretických poznatků do praxe. Výstupem praktické části je jednoduchá aplikace vytvořena v jazyce Elm a k ní z hlediska funkčnosti analogická aplikace vytvořena za pomocí jazyka JavaScript a frameworku React. Účelem této aplikace je představení hlavních principů funkcionálního programování, syntaxe jazyka a zhodnocení postavení a konkurenceschopnosti na trhu.

1.1 Cíl práce

Hlavním cílem této práce je analyzovat jazyk Elm z hlediska postavení na trhu a schopnosti konkurovat ostatním mainstreamovým front-endovým technologiím.

Mezi dílčí cíle patří:

1. představit jazyk Elm,
2. posoudit příznivost aktuální situace na trhu pro využití této technologie na vývoj webových aplikací,
3. porovnat technologii s konkurenčními technologiemi a přístupy k front-endovému vývoji aplikací,
4. zhodnotit minulou a momentální situaci na trhu práce,
5. vytvořit jednoduchou aplikaci sloužící pro demonstraci technologie.

1.2 Metodika práce

Bakalářská práce čerpá především z internetových zdrojů, jako je oficiální dokumentace jazyka Elm nebo Elm fórum a knihy Elm in action od autora jménem Richard Feldman, který je jedním z členů Elm core týmu a mimo jiné se stará o správu technologie. Bohužel počet zdrojů v českém jazyce je mizivý, tudíž volba zahraničních zdrojů byla nevyhnutelná. Tyto zdroje nicméně disponují dostatkem informací k tomu, aby tato problematika byla pochopena v celém rozsahu. Kromě těchto zdrojů autor také čerpá ze své dvouleté zkušenosti s vývojem v technologii Elm na komerčním projektu.

1.3 Hledání zdrojů

Jedním ze způsobů vyhledávání zdrojů je využití webového vyhledávače Google ve snaze najít konkrétní knihy, které se věnují tomuto tématu. Kromě knih byly důležité i oficiální dokumentace.
Dalším způsobem je využití webových stránek Scopus, Google Scholar pro odborné články a Thesis pro bakalářské a diplomové práce zabývající se obdobným tématem.

2 Teoretická část

Teoretická část bakalářské práce se nejdříve věnuje tématu funkcionálního programování, kde je popsána historie funkcionálního přístupu, specifika tohoto přístupu a nakonec se zabývá programovacími jazyky relevantními pro tuto práci, které využívají tento přístup.

Následně se pozornost přesouvá k technologii Elm, což je stěžejní sekce teoretické části.

V poslední sekci se nachází teoretické porovnání technologie Elm s konkurenčními technologiemi na trhu. Tato část tvoří základ pro praktickou část, kde budou tyto poznatky blíže zkoumány.

2.1 Funkcionální programování

Funkcionální jazyky vycházejí z matematiky. Matematici Hamilton, De Morgan a Boole formalizovali logické systémy a z jejich práce vzešla výroková a predikátová logika.

Výroková logika se zabývá pravdivostními hodnotami určitých výroků a také používá operátory pro spojování různých výroků. V programovacím jazyce jsou pravdivostní hodnoty reprezentovány jako `true, false` a operátory jako `and, or, not, ...`

Za pomocí výrokové logiky je možné skládat i složitější systémy. Například elektrické obvody mohou být definovány skrze logické výroky.

Predikátová logika rozšiřuje výrokovou logiku a díky ní je pro vyjádření možné využít širší spektrum hodnot jako například čísla nebo řetězce. Toto se děje za pomocí zobecnění výrokové logiky, aby bylo možné využít rozšířené hodnoty a aplikování funkcí. Predikátová logika také přináší kvantifikátory pro reprezentaci sekvencí hodnot. Kvantifikátory se dělí na univerzální – v sekvenci má každý element požadované vlastnosti – a existenční – v sekvenci existuje alespoň jeden element s požadovanými vlastnostmi.

Na přelomu tisíciletí se matematici Russell a Whitehead se snažili o kompletní logický popis matematiky ve svém díle Principia Mathematica. Matematik Hilbert se poté snažil demonstrovat fakt, že toto dílo opravdu obsahovalo úplný popis matematiky. Toto snažení se nazývá Hilbertův program. Hilbertův program nastartoval zkoumání teorie vyčíslitelnosti, která se zabývá otázkami algoritmického řešení problémů.

Ve třicátých letech minulého století byly navrhnuty tři různé přístupy k teorii vyčíslitelnosti: Turingův stroj, Kleenovy rekurzivní theorémy a lamba kalkul. Rekurzivní theorémy a lambda kalkul vytvořily základ pro funkcionální programování. (Michaelson, 1989)

2.1.1 Individuality funkcionálního programování

Funkcionální programování je jedno z mnoha programovacích paradigmat. Programovací paradigma je možné si představit jako soubor pravidel, které jistým způsobem omezují programátora a nutí jej psát kód určitým způsobem, což má za cíl zamezit chybám, kterých by se mohl programátor dopustit. Tento přístup je postaven na programování aplikací výhradně z funkcí, které dostávají vstupy a produkují výstupy. Program napsaný ve funkcionálním jazyce je ve skutečnosti strukturovaný sled funkcí. Jedna funkce volá druhou a tento proces se neustále opakuje. Tento proces opakovaného volání se nazývá kompozice nebo vnořování funkcí a vypadá následovně: `funkce1(funkce2(funkce3(...)))`. (Michaelson, 1989)

Funkcionální přístup má řadu výhod. Například za předpokladu, že se proměnné jednou přiřadí hodnota, tato hodnota se již nemění a neprobíhá přiřazování jiné hodnoty této proměnné. Funkce tedy netvoří žádné „vedlejší efekty“ (side-effects). Tento fakt eliminuje obrovské množství errorů v aplikacích a je mnohem snazší takové aplikace debugovat. V tradičních imperativních jazycích je možné deklarovat proměnnou, které je přiřazena hodnota a poté zavolat příkazy, které této proměnné přiřadí jinou hodnotu. Tímto je možné za běhu aplikace měnit stav proměnné, což vede k tomu, že jedno pojmenování může být ve skutečnosti spojeno s více hodnotami. Ve funkcionálních jazycích jsou proměnné pouze formální pojmenování hodnot, které jsou posílány funkcím jako parametry. Jakmile je tato vazba stanovena, není možné k tomuto pojmenování přiřadit jinou hodnotu. (Michaelson, 1989) Funkcionální programovací jazyky jsou charakterizovány tím, že mají implicitně uložený stav, který je upravován příkazy, což zaručuje možnost tyto příkazy sekvencovat a udržet plnou kontrolu nad stavem aplikace. (Hudak, 1989)

2.1.2 Funkcionální programovací jazyky

Na poli softwarového inženýrství společně existují programovací jazyky, které implementují určité funkcionální prvky a jazyky, které využívají plný potenciál funkcionálního přístupu, tzv. čistě funkcionální programovací jazyky. Kromě čistě funkcionálního přístupu je možné se setkat také s hybridním, který v sobě kombinuje prvky funkcionálního a objektově orientovaného paradigmatu.

Jazyky s funkcionálním přístupem jsou často využívány pro back-endové aplikace, protože vývojářům nabízí funkce, jako je imutabilita, možnost vysoké míry abstrakce, silně typový systém nebo efektivní paralelní výpočty. Tyto funkce přispívají na výkonu, což je pro back-endové systémy zásadní.

Funkcionální přístup však není exkluzivní pro vývoj back-endových systémů. Je možné jej využít i při vývoji front-endových aplikací. Ideálním příkladem je Elm, který se využívá především pro vývoj front-endových webových aplikací. Dalším zástupcem je JavaScript, který také mimo jiné umožňuje funkcionální přístup při práci s ním.

2.1.3 Historické funkcionální jazyky

LISP

Jeden z raných programovacích jazyků, který vytvořil John McCarthy mezi lety 1956-1962. (McCarthy, 1979) Ačkoliv je řadou odpůrců označován za nefunkcionální, je jeden z prvních jazyků, který implementuje procedury, což vedlo k funkcionálnímu označení.

ML

Vznikl v roce 1973 na půdě University of Cambridge (Hosch, 2023) a dal základ pro vývoj dalších programovacích jazyků jako Standard ML nebo Caml.

2.1.4 Soudobé funkcionální jazyky

F#

Programovací jazyk F# byl vyvinutý v roce 2005 firmou Microsoft. Jedná se o univerzální jazyk sloužící ke tvoření výkonného softwaru. Své využití nalézá při vývoji front-endových

i back-endových aplikací. V řadě aspektů se podobá jazyku Elm a mezi jeho funkce patří například imutabilita, statické typování nebo porovnání vzorů.
F# komunita se soustředí primárně okolo F# Software Foundation. Komunikace v komunitě původně probíhala skrze fóra. V současné době se trend přesunul na platformu Twitter. (Syme, 2020)

Clojure

Cojure je dynamický a univerzální programovací jazyk. Tento jazyk vychází ze zmíněného LISPu. Umožnuje mutaci dat, nicméně přináší sofistikovaný způsob řešení tohoto přístupu. Obdobně jako F# i Clojure komunita se soustředí na fórech a sociální síti Twitter. (Hickey, 2022a)
Na své oficiální stránce Clojure tým uvádí 513 firem, které využívají tuto technologii. Mezi hlavní případy užití této technologie patří podnikový software, vývoj aplikací pro komerční služby nebo aplikace sloužící k administraci systémů. (Hickey, 2022b)

Scala

Scala podporuje zároveň funkcionální i objektově orientovaný přístup, tudíž jde o univerzální, staticky typovaný, vyšší programovací jazyk. (Baalman, et al. 2023a) Za jeho tvorbou stojí prof. Martin Odersky. Komunita se běžně soustředí na fórech a sociálních sítích jako Twitter nebo Reddit. Také na webové stránce www.stackoverflow.com jsou aktivně probírána témata tykající se této technologie.
Využití nalézá zejména při vývoji mikroslužeb, komunikačních platforem, projektů pracujících s velkými daty nebo při procesovaní dat. (Baalman et al., 2023b)

Haskell

Přestože se vznik tohoto jazyka datuje do roku 1990 (Hudak et al., 2007), lze ho nazvat soudobým, protože i v dnešní době se okolo něho soustředí aktivní komunita. Převážná část komunity využívá tento programovací jazyk pro své osobní projekty. Dále své uplatnění nalézá nejen na trhu práce, ale i na akademické půdě.
Vývojáři tuto technologii využívají především na tvorbu konzolových aplikací, rozhraní, knihoven a frameworků nebo procesování dat. (Fausak, 2022)

JavaScript

JavaScript je jazyk hojně využívaný pro webový vývoj. Ačkoliv se nejedná o čistě funkcionální programovací jazyk, v posledních letech v něm přibývá implementace funkcionálních prvků. Například určité funkce nad poli těží z funkcionálního přístupu a nemutují původní pole, ale vrací nové pole. Kromě nativních částí jazyka je možné využít knihoven, které rozšiřují možnosti funkcionálního programování v JavaScriptu. Jednou z takových knihoven je například Ramda. (Kunasaikaran & Iqbal, 2016)

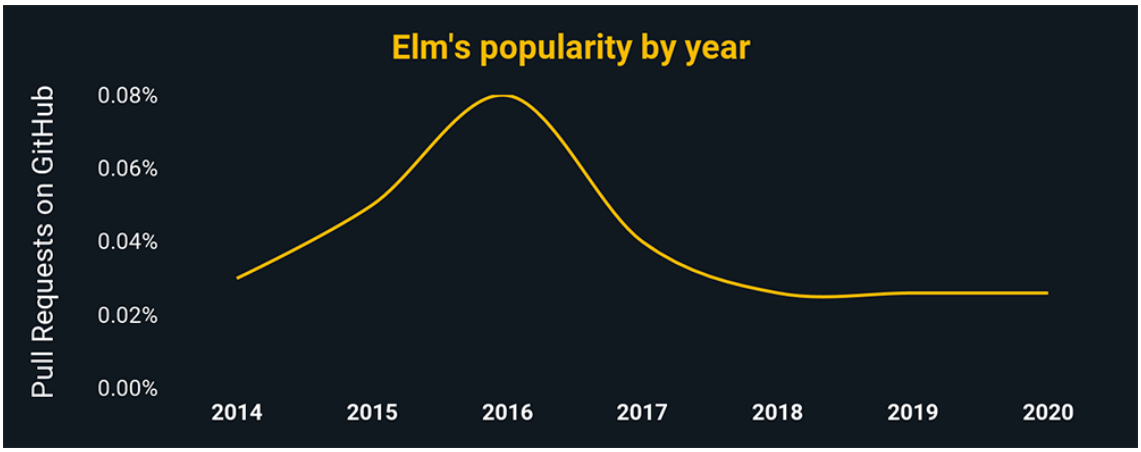
Elm

Programovací jazyk Elm se také řadí mezi soudobé funkcionální jazyky. Jedná se o ryze funkcionální programovací jazyk, který se využívá primárně pro vývoj webových aplikací. (Czaplicki, 2021a) Jeho vzniku, funkcím a dalším detailům se věnuje následující kapitola.

2.2 Představení programovacího jazyka Elm

Elm je ryze funkcionální jazyk kompilovaný do JavaScriptu a využívá se hlavně k vývoji webových aplikací. (Czaplicki, 2021a) Nejčastější případ užití je pro front-endovou část webové aplikace. Kromě toho však existuje například Lamdera, což je platforma, která slouží pro vývoj full-stack (back-end i front-end části) aplikací pomocí Elmu. (Rogic, 2023) Vznik programovacího jazyka Elm se datuje k roku 2012, kdy Evan Czaplicki přišel s touto novou technologií v rámci své diplomové práce. (Czaplicki, 2012) Již tou dobou měl ve svém okolí určité množství příznivců, kteří se zajímali o tuto práci a její výstup. Tento krok umožnil vzniku jedinečné technologii, která byla dále vyvíjena díky zájmu okolí a v budoucnu si našla své uplatnění a příznivce. Elm rostl velmi stabilně od počátku a největší popularitu zažil v roce 2016 (viz Obrázek 1). Nicméně tento rok byl pro technologii zlomový, protože s dalšími lety tento trend nepokračoval a k dnešnímu datu popularita drasticky upadla. Naštěstí Elm ani jádro komunity, které se okolo něj soustředí, toto výrazně negativně nepoznamenalo. V dnešní době má Elm svůj core team, který jej spravuje, a kromě toho také pevnou komunitu uživatelů, která je velmi nápomocna při řešení problémů či tvoření nových funkcionalit či balíčků. Celá tato komunita je přinejmenším specifická tím, že využívá jiné kanály určené pro komunikaci, než na které je běžně programátor zvyklý u jiných technologií. Například na webové stránce

www.stackoverflow.com je o tuto technologii minimální zájem, oproti jiným (např. TypeScript). Jeden z klíčových rozdílů je tedy ten, že tento kanál není hlavním komunikačním médiem, což implicitně vede k představě, že tuto technologii nikdo nepoužívá. Ačkoliv poměr uživatelů této technologie se ani zdaleka neblíží k nejvyšším v tomto oboru, nelze tvrdit, že není dostatečně využívána.



Obrázek 1: Poměr pull requestů na platformě GitHub mezi lety 2014 a 2020 (Gallinelli, 2021)

Jedním z hlavních komunikačních médií je Elm Slack, kde denně probíhají diskuse, řeší se různé problémy a mimo jiné tam jsou také nabízeny pracovní příležitosti. Kromě toho se uživatelé či zájemci o technologii mohou dozvědět více například v týdeníku Elm Weekly. Zde zájemci o novinky zadají svou e-mailovou adresu a každý týden mohou očekávat novinky ze světa Elm technologie. V každém takovém e-mailu je čeká nejen řada novinek, ale především také nabídky pracovních pozic. Ve velké většině jsou pracovní pozice v dálkovém režimu, takže nejsou omezeny místem pobytu, nicméně vyžadují alespoň základní znalost angličtiny. Tyto týdeníky nejsou specifické pro Elm, avšak je to důkaz, že se uživatelé a autoři snaží držet krok s konkurencí.

2.2.1 Výhody technologie Elm

Elm si své příznivce získal primárně skrze své technické funkce, které nabízí. Některé výhody pochází z funkcionálního přístupu, na kterém je tato technologie postavena. Důležitý je fakt, že Elm je zcela odlišný programovací jazyk, který své funkce přinesl na pole front-endového vývoje. Tyto praktiky nebyly v době vzniku této technologie běžné, což potenciálně mohlo pomoci přilákat zájemce.

Stabilita a spolehlivost

Největší výhodou Elmu je bezesporu stabilita a spolehlivost. Elm je staticky typovaný jazyk a dává k dispozici překladač, který tyto typy kontroluje. Díky tomuto zachytí typové chyby ještě před tím, než se dostanou do běhového prostředí. (Fairbank, 2019) To znamená, že nemůže nastat stav, kdy by uživatel narazil na error při interakci s aplikací, který byl vyvolaný špatnými typy v argumentu některé z funkcí. Dále v tomto jazyce neexistuje typ `null`, ale dostáváme k dispozici řadu různých typů, které slouží pro reprezentaci těchto stavů a řeší tyto případy. Zároveň překladač kontroluje, aby programátor ošetřil všechny možné stavy, které mohou v aplikaci nastat – k tomuto slouží například typ `Maybe`.

Jednoduchost

Elm si zakládá také na jednoduchosti, a proto není potřeba instalovat žádné knihovny třetí strany například pro správu stavů nebo přidávat závislosti do aplikace, aby uživatel mohl začít programovat svou aplikaci. Toto vše je v jazyce nativně podporováno. Elm komunita je ovšem velice proaktivní a tvoří balíčky s pokročilými funkcemi a šetří tak práci jiným programátorům, kteří by danou funkcionalitu museli implementovat samostatně, obdobně jako to funguje u jiných jazyků. Všechny balíčky jsou dostupné přímo a stačí je nainstalovat pomocí příkazu `elm-install ‚jmeno-autora/nazev balicku‘` a importovat do daného modulu, kde je programátor potřebuje využít. Tyto balíčky jsou tvořeny celou komunitou a přispět tak může kdokoliv. U těchto balíčků je kladen maximální důraz na kvalitu rozhraní a dokumentace.

Imutabilita

Další klíčovou výhodou je imutabilita neboli neměnnost vnitřního stavu, což znamená, že hodnota se nemůže změnit, jakmile je jednou vytvořena. (Fairbank, 2019, s. 13). Tato funkce přichází vhod zejména při škálování aplikací a zaručuje, že napříč aplikací nebude probíhat neočekávaná změna stavů, které vyústí v nepředvídatelné chování aplikace. V praxi tento přístup vývojáři stále umožní například provádět operace nad daty nebo měnit hodnoty uložené v recordu, nicméně výstupem těchto akcí není původní změněná hodnota, ale nová instance s upravenými daty. Tento přístup dělá z těchto akcí explicitnější operaci.

Interoperabilita

Díky možnosti interoperability s JavaScriptem je možné, aby tyto dvě vrstvy společně komunikovaly, což s sebou přináší například možnost Elm postupně implementovat do svého projektu modul po modulu. Výhodou je tedy možnost napsat pouze část zdrojového kódu v jazyce Elm a získat představu, zda je toto vhodná technologie pro daný případ užití. Zároveň tato funkce nabízí možnost suplovat absenci funkcí, které nejsou dostupné v jazyce Elm, pomocí využití JavaScriptu. (Czaplicki, 2021a)

2.2.2 Nevýhody technologie Elm

Ačkoliv příznivci technologie často mluví pouze o výhodách této technologie, je třeba přiznat, že i přes vynikající vizi existují nevýhody při jejím využívání. Mezi hlavní nedostatky patří velikost komunity, relativně nízké množství pracovních pozic a pracovní síly na trhu a podpora ze strany jiných technologií – například vývojová prostředí.

Malá komunita

Oproti ostatním technologiím, které se ve světě webového vývoje dají považovat za „mainstream“ (hlavní proud), má Elm malou komunitu, což se dá považovat za výhodu, ale i nevýhodu. Pro programátora může tento fakt být užitečný, protože menší velikost komunity zprostředkovává bližší vztahy napříč celou komunitou a problémy se dají řešit poměrně konkrétně a rychle na platformě Slack. Například problém v určitém balíčku, který je spravovaný komunitou může programátor nahlásit přímo a komunita jej spraví

rychleji, než by se stalo u technologie, za kterou stojí velký korporátní systém. Problém však nastává, když se například autor balíčku rozhodne přejít na jinou technologii a balíček tak postupem času zanikne. Další nevýhodou menší komunity je menší kapacita pro tvorbu nových balíčků, funkcionalit, návodů, článků a dalších podpůrných materiálů.

Nízký počet pracovních nabídek

Z hlediska pracovních pozic je opravdu těžké v České republice najít pracovní místa na pozici webového vývojáře v programovacím jazyce Elm. Důvodem je velmi pravděpodobně nedostatek produkčních projektů. V České republice je velice složité najít firmu, která by pro vývoj využívala Elm. Z dohledatelných firem je možné zmínit CN Group a.s. nebo Newired Technology s.r.o., které na svých webových stránkách uvádějí informace o tom, že využívají tuto technologii. Nabídka pracovních pozic je nesrovnatelná s jinými mainstreamovými technologiemi.

Závislost na této technologii může zároveň být do budoucna nevýhodná pro firmy, které pracují na dlouhodobých klientských projektech, protože v České republice je složité najít programátory, kteří by v této technologii byli seniorní. V případě, že pracovník opustí projekt, není snadné najít náhradu jako u jiných mainstreamových technologií. V případě fluktuace pracovníku na projektu může pro firmu být výhodnější využít takový programovací jazyk, který na trhu má vyšší zastoupení a umožní tak úměrně škálovat tým, popřípadě najít náhradu za pracovníky v případě odchodu z projektu.

2.3 Vývoj za použití technologie Elm

Obecným standardem pro vývoj front-endových webových aplikací je JavaScript. Programování pomocí technologie Elm je velmi odlišné od JavaScriptu. Programátor je schopen dosáhnout obdobných výsledků za použití obou technologií, avšak cesta, kterou toho docílí, se liší.

2.3.1 Rozdíly oproti JavaScriptu

Elm i JavaScript sdílí společný cíl – vývoj webových aplikací. Ačkoliv JavaScript je možné použít pro vývoj na mnoha různých platformách, pro účel této práce je důležitý JavaScript jakožto front-endový programovací jazyk. Tyto dva jazyky jsou si blízké svým cílem, ale

liší se svými funkcemi a charakteristikami, proto si zaslouží porovnat alespoň v několika následujících aspektech.

Syntaxe

První nestandardní jev, se kterým se může vývojář setkat při prvním kontaktu s technologií Elm, je velice rozdílná syntaxe.

Například v JavaScriptu je možné se setkat se třemi způsoby deklarování proměnných a funkcí pomocí různých klíčových slov: `var`, `let`, `const`. Každé klíčové slovo má v programu svůj význam. Primárně určují rozsah viditelnosti a možnost mutability proměnné. Elm nemá analogii k těmto zápisům, ale argumenty a funkce se deklarují přímo a bez klíčových slov.

Klíčovým rozdílem mezi těmito technologiemi je však přístup k programování. JavaScript je multiparadigmatický dynamicky typovaný programovací jazyk (MDN Contributors, 2023a), zatímco Elm je čistě funkcionální silně typovaný programovací jazyk (Fairbank, 2019, s. 12). Tento rozdíl pro uživatele znamená, že při volbě JavaScriptu bude mít větší volnost při výběru přístupu a psaní kódu. Na druhé straně Elm stanovuje relativně úzké restrikce pro uživatele, což ale ústí ve větší jistotu a předvídatelnost při tvorbě aplikací.

Dostupné nástroje

JavaScript má mnohem otevřenější ekosystém, pro který je dostupných vysoký počet různých nástrojů. Důležité jsou zejména knihovny a balíčky, které tvoří komunita a poté jsou veřejně dostupné ostatním vývojářům. Pro jazyk JavaScript je dostupná hned řada správců těchto knihoven a balíčků.

Na druhou stranu jazyk Elm má jednoho oficiálního správce balíčků, který tyto balíčky umožňuje sdílet napříč komunitou. Při tvorbě tohoto správce byl kladen důraz na jednoduchost použití, kvalitu rozhraní balíčků a kvalitu dokumentace balíčků. (Czaplicki, 2014)

Typový systém

Jak již bylo zmíněno, typový systém těchto dvou technologií se výrazně liší. Zatímco v JavaScriptu jsou typy dynamicky určovány a také je možné přetypovat proměnnou, v Elmu je vyžadované typy specifikovat předem. Toto je výhodné pro přehlednost kódu

a také pro překladač, který tyto typy kontroluje. V případě, že se v aplikaci nachází nesoulad v typech, se aplikace nezkompiluje, tudíž nedojde k chybě v běhovém prostředí a uživatel je upozorněn na chybu.

2.3.2 Variabilita zápisu funkcí

Zápis funkcí v Elmu je také poměrně variabilní, takže programátor si může vybrat takový způsob, který mu vyhovuje. K tomuto poslouží funkční operátory, které napomáhají při zápisu funkcí.

```
addNumbers : Int -> Int -> Int
addNumbers a b =
    a + b
addNumbers : Int -> Int -> Int
addNumbers a =
    (+) a
addNumbers : Int -> Int -> Int
addNumbers a =
    a |> (+)
```

Výpis kódu 1: Variabilita zápisu jedné funkce v Elmu (vlastní zdroj)

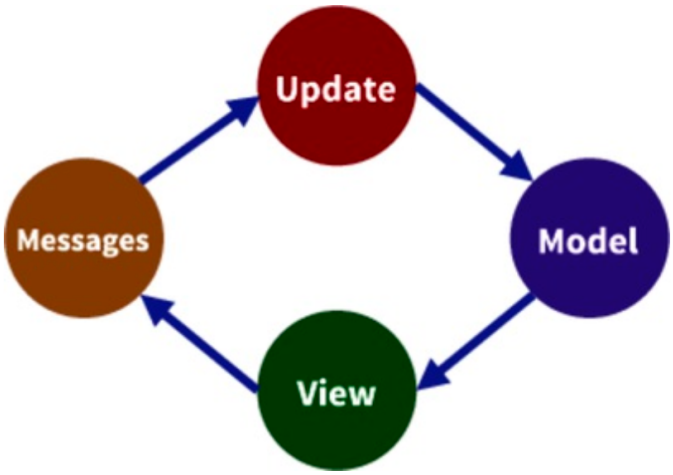
V obou jazycích se využívá rozdílný typový systém. JavaScript je interpretovaný programovací jazyk, který typuje proměnné dynamicky a nevyžaduje striktně definovaný typ pro konkrétní proměnnou. Za běhu programu tedy může změnit typ proměnné podle potřeby. (Simpson, 2020)

V Elmu se využívá staticky typový system, tudíž každá proměnná nebo funkce má svůj typ, který se za chodu programu nemění. Toto zabraňuje přetypování, které může v aplikaci vyvolat nestabilitu. (Fairbank, 2019, s. 13)

Elm mimo jiné využívá TEA (Elm architektura), což je přístup, pomocí kterého se separují různé části logiky od sebe a kód je tím přehlednější a lépe laditelný. (Fairbank, 2019, s. 65) Ačkoliv se v JavaScriptu tento přístup dá také implementovat, tento jazyk k tomu není nativně přizpůsobený.

2.3.3 Elm architektura (TEA)

Elm architektura je návrhový vzor určený pro funkcionální programování. Využívá se konkrétně při tvorbě webových aplikací s technologií Elm. Tento vzor dělí aplikaci na čtyři části: model, view, update a message. Svou charakteristikou připomíná MVC nebo MVVM návrhové vzory, které se hojně využívají při vývoji back-endových i front-endových aplikací. Jejich společným cílem je rozdělit aplikaci na spolu související funkční vrstvy. Tyto přístupy jsou přesto rozdílné, protože MVC a MVVM vzory nabádají k rozdělení stavů do více modelů a tvorbě lokálních stavů. Elm architektura se na druhou stranu snaží o sjednocení stavu na jednom místě. (Fairbank, 2019, s. 66)



Obrázek 2: Elm architektura (Fairbank, 2019, s. 65)

Model

Jedná se o reprezentaci vnitřního stavu aplikace. (Czaplicki, 2021a) Aplikace může disponovat více modely. Typicky každá stránka aplikace má svůj vlastní model, ve kterém se drží lokální stav této konkrétní stránky.

V případě rozšíření aplikace je velmi pravděpodobné, že bude nezbytné reprezentovat globální stav. Tohoto je možné docílit pomocí vytvoření modelu na samotném vrcholu struktury aplikace a posláním tohoto modelu jako parametr do funkcí, kde je potřeba číst nebo upravit globální stav. Při správném nastavení se změna globálního stavu promítne

do všech komponent, které s ním pracují. Tento přístup k řešení globálního stavu nevyžaduje instalaci žádné knihovny či balíčku třetí strany.

Message

Vlastní typ, který sdružuje všechny možnosti zpráv, které mohou v dané sekci nastat. Tyto zprávy jsou volány ve `view` funkci a reakce na zavolání dané funkce je provedena v update funkci. (Czaplicki, 2021a)

```
type Msg
    = NoUpdate
    | SetAccessToken String
    | UpdatedLanguage Translations.Language
    | UpdatedApi Api.Api
    | UpdatedShowingTranslationsButton Bool
    | UpdatedTime Time.Posix
    | UpdatedZone Time.Zone
    | UpdatedAppWidth Int Int
```

Výpis kódu 2: Vlastní typ Msg (vlastní zdroj)

Update

Přímou změnu stavu v aplikaci zajišťuje update funkce. Do této funkce vždy vstupuje typ zprávy, na základě které se provede porovnání vzorů (pattern matching) a pro danou zprávu se provede odpovídající akce. Kromě typu zprávy také do update funkce vstupuje stávající stav, který je úkolem adekvátně změnit. (Czaplicki, 2021a)

```
update : Msg -> Model -> ( Model, Cmd Msg )
```

Výpis kódu 3: Nejzákladnější typová anotace funkce update (vlastní zdroj)

View

Tato část slouží pro vizuální reprezentaci stavu aplikace. Vstupem pro tuto funkci je stav aplikace a výstupem je HTML obsah. Při změně stavu automaticky reaguje, funkce se

provede znovu s aktuální hodnotou na vstupu a výstupem je HTML obsah, který reflektuje aktuální stav aplikace. (Czaplicki, 2021a)

2.3.4 Globální stav aplikace

V předchozí části byl zmíněn model, který drží informace o stavu aplikace. Tento koncept funguje na lokální i globální bázi, přičemž není nutné do aplikace přidávat další závislost na balíčku či knihovně třetí strany.

Při rozšíření aplikace o více stránek či modulů je velmi pravděpodobné, že tyto komponenty budou potřebovat přistupovat ke stejným a hlavně aktuálním informacím. Tohoto je možné docílit pomocí vytvoření globálního modelu, který v sobě drží stav aplikace. Aby měly všechny komponenty napříč aplikací přístup ke globálnímu stavu, stačí jej inicializovat na samotném vrcholu struktury aplikace a poslat jej jako parametry do příslušných komponent. Tyto komponenty poté mohou číst tento stav a reagovat na jeho změny.

Pro změnu stavu je nutné vytvořit update funkci, do které vstupuje typ zprávy a model – v tomto případě globální stav aplikace.

Při výrazném škálování aplikace však tento přístup může narůst na komplexnosti a přinést značné nevýhody. V případě, že aplikace nabývá na velikosti, nabývají i informace, o kterých je potřeba mít přehled. Pakliže bychom tyto informace chtěli držet v jednom globálním stavu a pouze jej rozšiřovali, je potřeba si uvědomit, že při změně jakékoliv z hodnot v tomto modelu se jedná o změnu stavu celého modelu. Toto zapříčiní reakci patřičné `view` funkce, do které tento model vstupuje, která vrátí nový výstup. V případě, že by v tomto modelu byla uložena rozsáhlá data a stav aplikace by se měnil příliš často, může tento přístup vyústit v problémy s výkonem aplikace. (Hanhinen, Klemola, 2019)

```
type SharedState
    = SharedState SharedStatePayload

type alias SharedStatePayload =
    { translations : Translations.Model
    , api : Api.Api
    , key : Nav.Key
    , showingLanguageButtons : Bool
    , currentTime : Time.Posix
    , timeZone : Time.Zone
    , appWidth : Int
    }
```

Výpis kódu 4: Model globálního stavu (vlastní zdroj)

2.3.5 Typový systém

Elm je staticky typovaný, tudíž každá hodnota má explicitně přiřazený datový typ, který se nemění. Mezi základní typy se řadí například `Int`, `String`, `Bool`. Kromě základních typů je možné definovat vlastní typy pomocí typových aliasů.

Typový alias

Pakliže jsou funkce opatřeny typovou anotací, kompilátor hlídá, zda vstupní a výstupní parametry funkce odpovídají své typové anotaci. V případě, že některý z typů se liší od své očekávané typové anotace, kompilátor tento jev zachytí, oznámí chybovou hlášku, přeruší kompilování kódu a tím zabrání, aby se tyto chybné stavy dostaly do běhového prostředí. (Feldman, 2020, s. 57)
Vytvoření vlastního typového aliasu může být vhodné pro:

- sdružení dat, která spolu souvisí,
- opakované použití stejných dat a zamezení duplicitám,

- snížení počtu vstupních argumentů do funkce – například namísto počtu jednotlivých argumentů můžeme poslat jeden typový alias, který zahrnuje všechny tyto hodnoty,
- pro pojmenování hodnot – dva stejné datové typy v roli argumentů stojících za sebou mohou být libovolně zaměněny při volání funkce, tudíž je bezpečnější vytvořit typový alias a tyto hodnoty pojmenovat.

```
-- ARGUMENTY LZE ZAMENIT
pictureView : Int -> Int -› Html msg

pictureView width height =
-- NEBO
pictureView height width =

-- ARGUMENTY NELZE ZAMENIT
pictureView : Dimensions -> Html msg
pictureView { width, height } =
```

Výpis kódu 5: Zamezení záměny argumentů pomocí vlastního typu (vlastní zdroj)

2.3.6 Porovnání vzorů

Tato funkce na základě vstupního výrazu provede specifickou funkci pro každou větev z možností, které mohou nastat. Shoda musí být přesná, aby se daná funkce provedla. Jedním z výrazů, který často vstupuje do této funkce, je vlastní typ. V těle této funkce se pro každou jednu možnost tohoto typu provede specifická funkce. Tyto větve je možné dále rozvíjet, případně je možné využít výchozí funkci. Tato funkce se automaticky aplikuje na všechny větvě, pro které není ve funkci explicitně stanovená návratová funkce.

```elm
pageView : Page -> ReadyModel -> HtmlStyled.Html Msg
pageView page { sharedState } =
    case page of
        ToDoItemPage pageModel ->
            ToDoItemPage.view pageModel sharedState
                |> HtmlStyled.map ToDoItemPageMsg

        Dashboard _ ->
            Dashboard.view sharedState
                |> HtmlStyled.map DashboardMsg

        NotFoundPage ->
            NotFoundPage.view sharedState
```

Výpis kódu 6: Porovnání vzorů s vlastním typem jako parametrem (vlastní zdroj)

2.3.7 Kompilátor

Jednou z velkých konkurenčních výhod jazyka Elm je jeho kompilátor. Jeho úkolem je zkompilovat soubory s příponou `.elm` do typických souborů s příponou `.js`, které nativně běží v prohlížeči. Zároveň dohlíží na dodržování typové správnosti. Například v případě, že se v aplikaci objeví instance, která neodpovídá typové anotaci, kompilátor to při kompilování zachytí dříve, než se tento chybný stav dostane do běhového prostředí. Při tomto zachycení informuje uživatele o chybném stavu. Chybové hlášky, které kompilátor poskytuje jsou jednou z oblíbených funkcí při práci s touto technologií, protože tyto hlášky jsou často velice nápomocné a srozumitelné (viz Obrázek 3 a Obrázek 4). (Feldman, 2020, s.5)

```
Compilation failed
Compiling ...-- MISSING PATTERNS ---------------------------------------------
src/Header.elm

This `case` does not have branches for all possibilities:

23|>    case msg of
24|>        ChangedLanguage language ->
25|>            ( Cmd.none, SharedState.UpdatedLanguage language )
26|>
27|>        ClickedShowLanguageButtons bool ->
28|>            ( Cmd.none, SharedState.UpdatedShowingTranslationsButton bool )

Missing possibilities include:

    ClickedButton

I would have to crash if I saw one of those. Add branches for them!

Hint: If you want to write the code for each branch later, use `Debug.todo` as a
placeholder. Read <https://elm-lang.org/0.19.1/missing-patterns> for more
guidance on this workflow.

Detected problems in 1 module.
```

Obrázek 3: Chybová hláška po zachycení chybného stavu kompilátorem (vlastní zdroj)

```
Type mismatch error.
Expected: `Int`
Found: `Float` Elm
```

```
idCount : Int
```

```
Field on the type alias ModelInternalPayload
```

Obrázek 4: Typový error (vlastní zdroj)

2.3.8 Chybové hlášky

Kromě kompilátoru nabízí Elm i další prostředky pro řešení nežádoucích stavů. Tyto prostředky jsou obzvláště užitečné při komunikaci se serverem. Při pokusu o získání dat ze serveru mohou nastat situace, kdy data ze serveru přijdou v jiné struktuře, než kterou

dotazovatel očekává, anebo tato data nedorazí vůbec. V takovéto situaci má uživatel k dispozici hned několik nástrojů. Prvním je balíček Http, díky kterému se může uživatel mimo jiné dotazovat na externí data z určité adresy. Uživatel se například pomocí metody GET pokusí získat data ze serveru, k čemuž bude potřebovat specifickou `Msg`. Tato `Msg` má své jméno a datový obsah `Result error value`, kdy `error` a `value` jsou typové proměnné, tudíž za ně mohou být dosazené konkrétní typy, které odpovídají konkrétní situaci. `Error` tradičně nese typovou anotaci `Http.Error` a za `value` se dosazuje typ, do jehož podoby jsou namapována obdržená data. V případě, že bude dotaz neúspěšný, odpověď ze serveru je obsažena v části `error`. V případě, že je dotaz úspěšný, data jsou obsažena v části `value`. Obě tyto situace je potřeba zpracovat, takže i když je dotaz neúspěšný a selže z jakéhokoliv důvodu, uživatel má nad stavem plnou kontrolu a nenastane neočekávaná situace, kdy by se mohl pokusit provést operace nad daty, které nemá k dispozici. Jakmile je dotaz úspěšný a dotazovatel získá data v odpovědi ze serveru, je nezbytné tato data dekódovat, čímž je zajištěna konzistence a naprostá kontrola nad strukturou dat napříč aplikací.

Dekódování dat probíhá pomocí dekodérů. Dekodér má k dispozici typ, do kterého tato data namapuje. Dekódování daného pole může být provedeno více způsoby. Uživatel může například rozhodnout, že pole bude povinné, což by při nesplnění podmínek znamenalo, že dekódování selže a tento výsledek bude obsažen v `error` větvi. V případě, že by uživatel rozhodl, že dané pole je volitelné, tak musí poskytnout záložní data, která zastoupí očekávaná data v případě, že podmínky nebudou splněny. S tímto přístupem dekódování neselže.

```elm
decodeFlags : Decode.Decoder Flags
decodeFlags =
    Decode.succeed Flags
        |> Pipeline.required "baseApiUrl" Decode.string
        |> Pipeline.optional "toDoItems" decodeToDoItems
            initialToDoItems
        |> Pipeline.required "accessToken" Decode.string
        |> Pipeline.required "translations"
            decodeTranslations
        |> Pipeline.required "appWidth" Decode.int
```

Výpis kódu 7: Dekodér dotazovaných dat ze serveru (vlastní zdroj)

2.3.9 Komunikace s JavaScriptem

Aby Elm aplikace mohla existovat v prohlížeči, je nutné ji inicializovat ve skriptové části aplikace. Prohlížeče momentálně podporují JavaScript jako jediný skriptovací jazyk, tudíž ve skriptu, který je vložený do stránky, je potřeba inicializovat Elm aplikaci za pomoci JavaScriptu. Od tohoto momentu je teoreticky možné pracovat pouze s Elm aplikací, nicméně uživatel má stále přístup k JavaScriptové vrstvě.

První možností komunikace s vnější vrstvou JavaScriptu, kterou je možné využít, jsou takzvané `flags`. Tato komunikace probíhá hned při inicializaci aplikace. Inicializační funkce Elm aplikace vyžaduje jako parametr element `root`, společně se kterým je možné poslat i `flags`, což je objekt, ve kterém si uživatel může poslat různá data, ke kterým bude mít přístup v Elm aplikaci. Typicky se může jednat o šířku okna prohlížeče, obsah úložiště prohlížeče nebo například momentální čas a podobná data.

Druhou možností, kterou může uživatel zvolit, jsou `ports`. Tento způsob je možné využít kdykoliv za běhu aplikace přímo z Elm vrstvy. Port může sloužit k odesílání či přijímání zpráv z JavaScript vrstvy. Komunikace tímto způsobem probíhá většinou za účelem přistoupit k úložišti prohlížeče či využití protokolu WebSocket. Důležité je navázat na port v JavaScriptové vrstvě a napsat patřičné funkce.

```
const node = document.getElementById("root");

const flags = {
    baseApiUrl: "https://jsonplaceholder.typicode.com/",
    toDoItems: JSON.parse(localStorage.getItem("to-do-
        items")),
    accessToken: "",
    translations: { en, ru },
    appWidth: window.innerWidth
  };

  const app = Elm.Main.init({
    node,
    flags,
  });
```

Výpis kódu 8: Inicializace Elm aplikace v JavaScriptové vrstvě (vlastní zdroj)

2.3.10 Spolehlivost

Autoři tohoto silně typovaného programovacího jazyka v dokumentaci garantují, že uživatel v praxi neuvidí runtime error neboli chybu v běhovém prostředí, který by byl způsoben chybami v tomto jazyce. Toto je velice silné tvrzení, nicméně má své odůvodnění. Částečně je toto způsobeno tím, že s errory je nakládáno jako s daty, takže namísto chyby v běhovém prostředí je uživatel donucen typovým systémem explicitně ošetřit všechny možné stavy, což vede k eliminaci chyb v běhovém prostředí. (Czaplicki, 2021a)

Výborným příkladem je knihovna RemoteData, která je určena pro usnadnění získávání dat z externích zdrojů. Pří získávání dat z externího zdroje je velice časté, že nastane chyba, tudíž je nutné s tímto počítat a ošetřit tyto chybové stavy. Knihovna RemoteData nabízí typ, který má čtyři varianty pro stavy, které mohou nastat při pokusu o získání dat a to následující:

- **NotAsked** – data v této chvílí nejsou k dispozici, nebylo o ně požádáno,
- **Loading** – data v této chvílí nejsou k dispozici, probíhá pokus o získání dat,
- **Failure** *error* – získávání dat selhalo a je k dispozici error, který je možné prezentovat koncovému uživateli,
- **Success** *value* – získávání dat bylo úspěšné a je možné nakládat se získanými daty.

```
type RemoteData error value
    = NotAsked
    | Loading
    | Failure error
    | Success value
```

Výpis kódu 9: Typová anotace pro typ RemoteData (vlastní zdroj)

3 Praktická část

Úkolem praktické části je srovnat jazyk Elm s konkurencí v několika úrovních. V první části se jedná o porovnání z hlediska popularity. Později je předmětem porovnání výkon aplikací vytvořených za pomocí různých technologií. A v závěru praktické části se nachází demonstrace a zhodnocení různých funkcí a charakteristik technologií Elm a React.
Nakonec je zhodnoceno, zda jazyk Elm je schopen konkurovat mainstreamovým technologiím a zda by mohl najít své uplatnění v praxi i do budoucna.

3.1 Současný stav technologie

Ve světě front-endových technologií každý rok vzniká řada nových populárních frameworků, knihoven či jazyků a zároveň převážná většina rychle zaniká. Dlouhodobě se daří udržet pouze nízkému procentu ze všech vytvořených technologií.
V roce 2023 už má za sebou jazyk Elm přes deset let své existence, tudíž je natolik dospělý, aby bylo možné zhodnotit jeho současný stav a na základě toho poté i zhodnotit, jaké má tento programovací jazyk možnosti do budoucna.

3.1.1 Situace na trhu práce

Aktivní člen komunity Luca Mugnaini v roce 2021 a 2022 vytvořil celoroční souhrn novinek a událostí, které se týkají Elmu. V přehledu z roku 2022 autor uvádí částečný výčet firem, které používají technologii Elm. Tento list čítá 143 firem, z čehož pouze dvě firmy působí v České republice, nicméně řada firem nabízí i možnost práce na dálku. (Mugnaini, 2023)
Ačkoliv počet firem, které využívají tuto technologii, se nezdá být příliš nízký, samotné otevřené pozice, na kterých by s ní vývojář přišel do styku, jsou i v těchto firmách rarita. Téměř žádná z uvedených firem v momentálním okamžiku nenabízí volnou pozici pro Elm vývojáře.

Preference komunity

Na konci roku 2022 proběhl doposud nejaktuálnější průzkum State of Elm 2022. Tento průzkum je zaměřen primárně na Elm komunitu a pokládá respondentům otázky ohledně stavu komunity a preferencích jejich členů.

Převážná většina respondentů se řadí do věkové skupiny mezi 30-49 let, pochází ze Spojených států amerických, pracují ve firmě se sto a více zaměstnanci a uvádí o sobě, že jsou alespoň středně zkušení vývojáři s alespoň dvouletou zkušeností s touto technologií. Dle odpovědí v dotazníku je primárním zdrojem informací pro vývojáře Elm dokumentace, knihy na toto téma a Elm Slack. Platforma StackOverflow, která je na poli vývoje webových aplikací významným zdrojem informací, se řadí mezi méně oblíbené v této komunitě.

Závěr tohoto dotazníku se soustředí na kladné a záporné stránky využívání technologie Elm. Mezi hlavní nevýhody, které respondenti uvádí, se řadí pomalý posun technologie vpřed, mechanismus rozhraní cizí funkce, absence podpory některých rozhraní prohlížeče nebo množství duplicitního kódu, který je potřeba napsat pro správný chod aplikace. Naopak jako výhody uvádějí respondenti typový systém, přívětivé chybové hlášky, spolehlivost, jednoduchost a udržitelnost. (Stewart, 2023)

Z tohoto výzkumu vychází, že Elm je technologie, kterou vyhledávají především zkušení vývojáři, kteří mohou mít silné preference a hodí se pro prostředí, v němž je hlavním cílem zajistit bezproblémový chod aplikace s nejvyšší možnou spolehlivostí. Mimo jiné lze usoudit, že podle respondentů tato technologie není úplně samostatná a je alespoň z části závislá na nativním prostředí prohlížeče.

Hledání volných pracovních pozic

Hledání pracovních míst na pozici Elm vývojáře se může zdát neobvykle složité v porovnání s ostatními mainstream technologiemi. Při využití tradičních webových serverů sloužících pro hledání pracovních míst, se nepodařilo najít jediné zveřejněné pracovní místo na pozici Elm vývojáře. K tomuto průzkumu byly využity služby LinkedIn, Indeed, Glassdoor, Jobs.cz a Prace.cz. Oproti tomu jen na webu LinkedIn je v dobu výzkumu vypsáno přes 700 poptávek po webovém vývojáři ovládajícím technologii React v lokalitě Česko.

I přes tento fakt je však možné nalézt volná pracovní místa pro tuto pozici. Jedním ze zdrojů je Elm Slack, což je jeden z komunikačních kanálů využívaných Elm komunitou. Zde se nachází kanál dedikovaný poptávkám a nabídkám prací souvisejících s technologií Elm. Počet zde zveřejněných pracovních pozic ovšem ani zdaleka nedosahuje počtu pracovních pozic volných pro konkurenční technologie. V tomto kanále jsou pro běžné uživatele

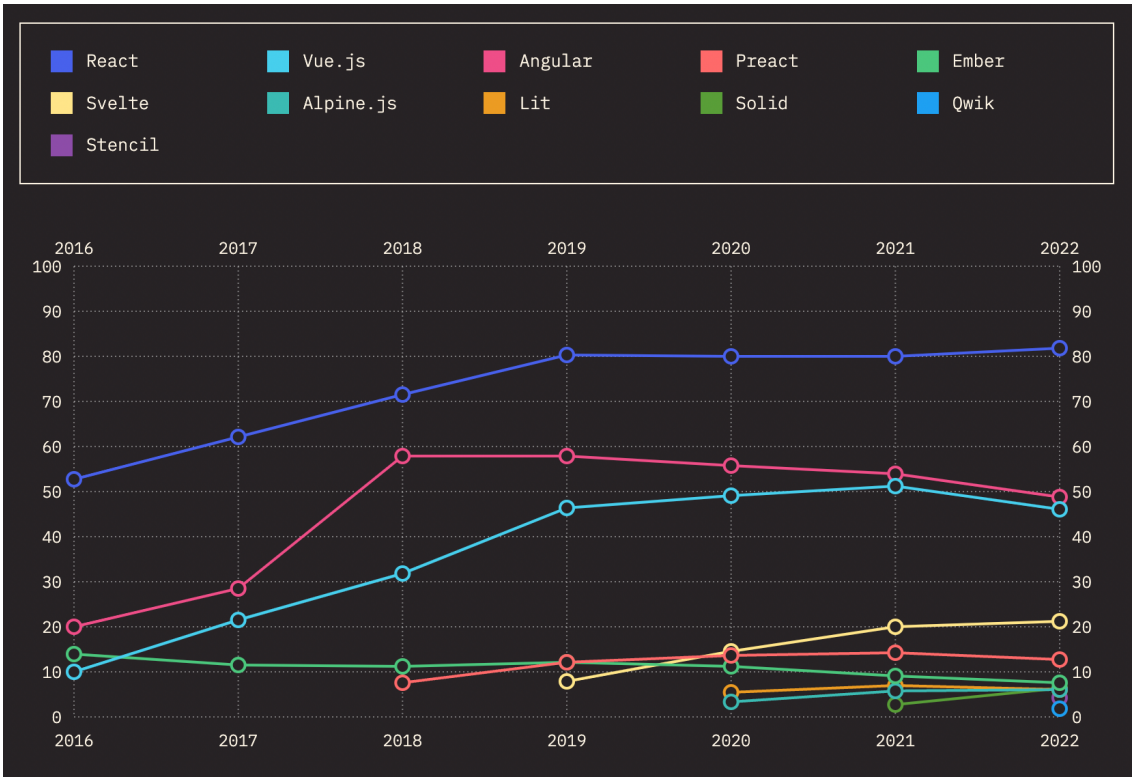
Slacku dostupné pouze zprávy mladší 90 dnů a za toto období čítá 18 nabídek i poptávek po pracovní pozici. Dalším zdrojem volných pracovních pozic je týdeník Elm Weekly od autora Wolfganga Schustera, obsahuje sekci vyhrazenou pracovním nabídkám a v momentě výzkumu čítá 259 čísel.
Hledání volných pracovních pozic se dá označit za složitější, než je tomu u konkurence, ale ne za nereálné. Díky nízkému počtu Elm vývojářů i takto nízký počet pracovních nabídek se nezdá být nedostatečný pro vývojáře. Na druhou stranu právě tento nízký počet vývojářů komplikuje situaci firmám, které poptávají tuto pracovní sílu.

3.2 Porovnání s konkurencí

Rychlé tempo vývoje technologií se promítá i do prostředí softwarového vývoje a konkrétně v oblasti vývoje front-endových aplikací se v průběhu času výrazně mění preference k technologiím, které jsou využívány. Důsledkem rychlého vývoje a měnících se preferencí uživatelů řada frameworků ztrácí na popularitě v průběhu času.
Tato část práce má za cíl srovnat technologii Elm s dalšími konkurenčními mainstreamovými technologiemi. Mezi tyto dlouhodobě úspěšné frameworky bychom mohli zařadit například React, Vue.js nebo Angular.

3.2.1 Oblíbenost mezi vývojáři

Podle dat z let 2016 až 2022 z průzkumu State of JavaScript (viz Obrázek 13) jsou tyto front-endové frameworky nejpoužívanější. Pro výpočet míry užití daného frameworku byla využita následující metoda: *(počet respondentů, kteří by framework použili znovu + počet respondentů, kteří by jej nepoužili znovu) / celkový počet respondentů.*
V tomto období byla pro respondenty jednoznačně nejoblíbenější technologie React, která si od roku 2016 polepšila o téměř 30 %. React se roku 2022 mezi respondenty těší 80% oblíbenosti. Druhou nejoblíbenější technologií k roku 2022 je Angular a třetí je Vue.js. Oba tyto frameworky dosahují přibližně 50% oblíbenosti mezi respondenty.
Technologie Elm není součástí tohoto výzkumu, jelikož pro srovnání byly využity frameworky, o kterých mělo povědomí alespoň 10 % respondentů. Z tohoto výzkumu vychází, že Elm se neřadí mezi mainstreamové technologie.



Obrázek 5: Průzkum State of JS (Greif a Burel, 2022)

3.2.2 Aktivní projekty

Další validní metrikou pro určení úspěchu programovacího jazyka může být počet projektů, které jsou v něm vyvíjené. Obecně je velice složité získat přesná čísla, jelikož řada projektů je například privátní, tudíž je nelze jednoduše dohledat. Zároveň ne všechny platformy nabízí veřejné API, pomocí kterého je možné dotazovat různá data.

Hledání projektů

Metodika dokazování byla omezena na platformu GitHub, která mimo jiné funkce také nabízí hostování projektů a zároveň nabízí uživatelsky přívětivé a jednoduché API, pomocí kterého se lze dotazovat na konkrétní data.

Při interpretaci této části práce je potřeba mít na paměti, že tyto výsledky jsou pouze orientační, protože se jedná o výsledky z pouze jedné z mnoha platforem, které je možné využít pro hostování projektů a zároveň jediný validní ukazatel, který můžeme použít pro dotazování dat pro konkrétní technologie, je jazyk. Jak již v této práci bylo objasněno, Elm je programovací jazyk, ale převážná většina konkurentů na poli front-endového

webového vývoje, které jsou konkurencí pro technologii Elm, jsou frameworky či knihovny, tudíž se jedná o derivace jazyka JavaScript a později TypeScript. Ačkoliv se jedná o orientační výsledky, i přesto nám tato data pomohou získat lepší náhled na situaci, ve které se jazyk Elm nachází.

Cílem této části je získat a porovnat počet repozitářů, které jsou hostovány na platformě GitHub v technologiích Elm a JavaScript. Hodnoty parametrů dotazů byly příslušný jazyk a datum vytvoření po 1.1.2022.

Srovnání výsledků

Výsledky hovoří jednoznačně ve prospěch jazyka JavaScript. Odpověď na dotaz směřovaný na GitHub API, který obsahoval zmíněné parametry, obsahuje tři klíče: total_count, incomplete_results a items. Total_count je pro nás stěžejní hodnota, která udává počet repozitářů odpovídající zadanému dotazu. Incomplete_results značí, zda výsledky v odpovědi jsou kompletní. V případě, že dotaz je příliš obecný, je možné, že hodnota pod tímto klíčem bude false, protože GitHub API v tomto případě preferuje rychlost odpovědi na dotaz, takže pokud obsah odpovědi je příliš obsáhlý, tento obsah rozdělí do vice stránek, na které je poté také možné se dotazovat. Pod klíčem items se v odpovědi nachází pole repozitářů, které odpovídají specifikovaným parametrům. URL s parametry pro dotaz je následující: https://api.github.com/search/repositories?q=language:LANGUAGE+created:>2022-01-01. Pod hodnotou LANGUAGE je dosazen konkrétní jazyk.

Odpověď pro jazyk JavaScript pod klíčem total_count obsahuje hodnotu 6320369 a pro jazyk Elm obsahuje hodnotu rovných 1800.

Tato čísla bez kontextu mohou budit dojem, že Elm není ani zdaleka schopen konkurovat JavaScriptu. Důvodů, proč mezi těmito dvěma technologiemi je takový rozdíl v počtu aktivních repozitářů na platformě GitHub, může být několik.

Prvním z důvodů může být stáří technologií. JavaScript byl vyvinut v roce 1995, zatímco Elm až v roce 2012. V průběhu své existence se stal JavaScript standardem zodpovědným za interaktivní část ve webových prohlížečích (MDN Contributors, 2023b) a tím zaujal velice dominantí pozici v tomto oboru. Postupem času se na poli front-endového vývoje webových aplikací začaly vyvíjet nové technologie, u kterých se z velké části jedná o deriváty JavaScriptu, například React, Angular nebo Vue.js. Všechny tyto technologie,

které jsou založeny na technologii JavaScript jsou tedy zahrnuty i v konečném výstupu z předešlého dotazu.
Dalším důvodem, který by mohl pomoci vysvětlit takto velký rozdíl ve výsledcích, je trend v popularitě jazyka Elm. Elm se těšil největší popularitě okolo roku 2016 (viz Obrázek 1), jenže od této doby popularita nenarůstá konstantně a spíše stagnuje.
V konečném závěru tento průzkum přinesl zajímavé výsledky. Na základě těchto výsledků je možné usoudit, že Elm je potenciálně vhodnější variantou pro vývojáře, kteří nemají v oblibě mainstreamové technologie. Nicméně firma či vývojář, který se rozhodne tuto technologii využívat, si musí být vědomi rizik spojených s nižším zájmem a v průběhu času sledovat, zda je technologie stále aktivní.

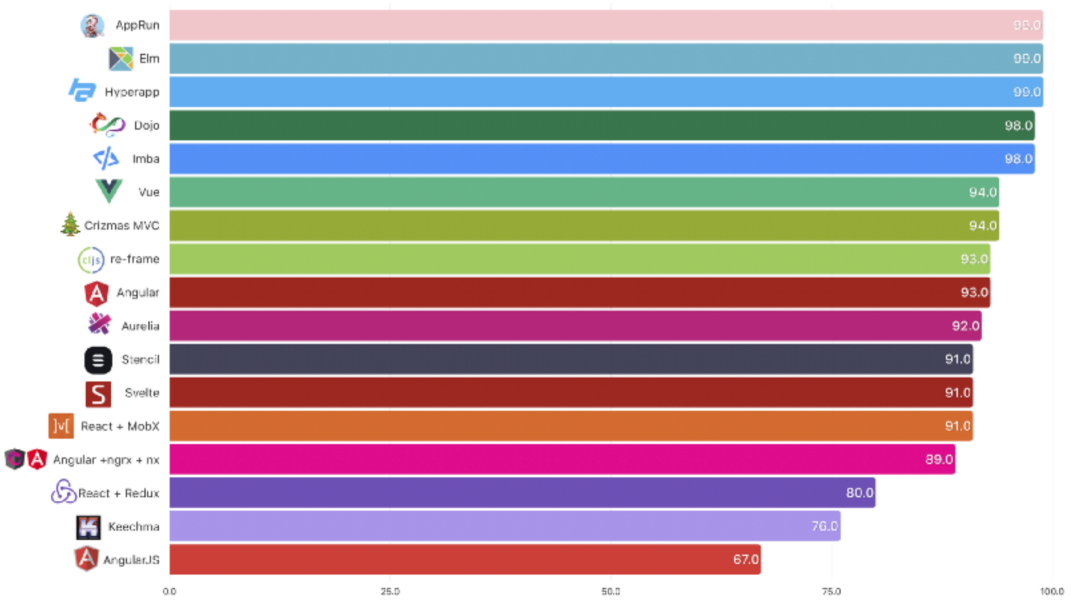
3.3 Benchmark technologií

Pro možnost porovnání různých frameworků a programovacích jazyků byl vytvořen veřejně dostupný repozitář s názvem realworld, ve kterém se nachází přes sto možností implementací aplikace Conduit, která je inspirována blogem Medium. (Eames, 2023)
Jacek Schae vypracoval porovnání řady front-endových frameworků založené na aplikacích z realworld repozitáře. Jako metriky si zvolil výkon, velikost aplikace a počet řádků kódu. Z výčtu technologií, které byly v tomto testu srovnávány, byly pro účel této práce vybrány Elm, Vue.js, React v kombinaci s technologií Redux a Angular v kombinaci s technologií TypeScript.

3.3.1 Lighthouse skóre

Pro tento test byl využit nástroj Lighthouse Audit, který je součástí prohlížeče Chrome. Rozmezí možných výsledků se může pohybovat mezi 0 až 100 body, kdy 100 bodů je maximální dosažitelné skóre.
Nastavení bylo pro všechny testy jednotné. Testy probíhaly na mobilním zařízení, testoval se výkon a internetové připojení bylo zpomalené na 3G při čtyřnásobném zpomalení procesoru.
Z výsledků je jednoznačné, že Elm byl z tří výše porovnávaných front-endových technologií nejrychlejší s 99 body. Na druhém místě se umístil Vue.js s 94 body, na třetím

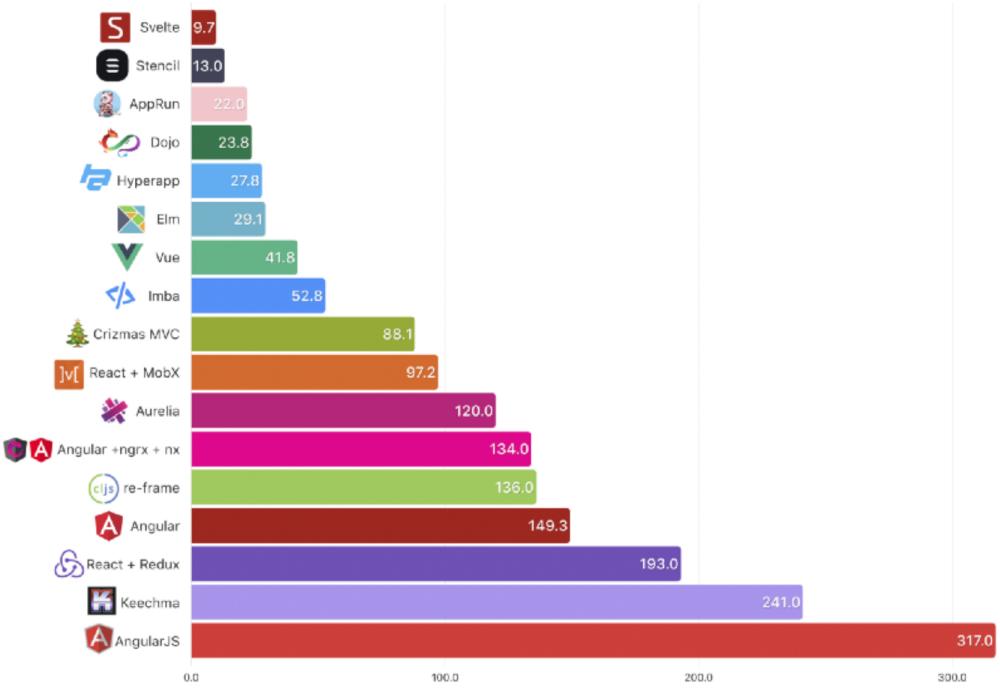
místě Angular za použití TypeScriptu s 93 body a v pozadí na čtvrtém React v kombinaci s technologií Redux s 80 body.



Obrázek 6: Benchmark front-endových frameworků (Schae, 2019)

3.3.2 Velikost aplikace

Tento test měl za úkol srovnat velikost dat přenesených skrze internetovou síť. Toto závisí nejen na velikosti dané technologie, ale také na závislostech, které jsou potřeba pro chod aplikace. Určitou roli může hrát také schopnost technologie odstranit nevyužitý kód. V tomto testu opět uspěl Elm s výslednou velikostí 29,1 KB, druhý byl Vue.js se 41,8 KB, třetí se umístil Angluar za použití TypeScriptu se 149,3 KB a na čtvrtém místě skončil React v kombinaci s Redux s 193 KB, což je více než šestinásobek velikosti aplikace vytvořené za pomocí technologie Elm.



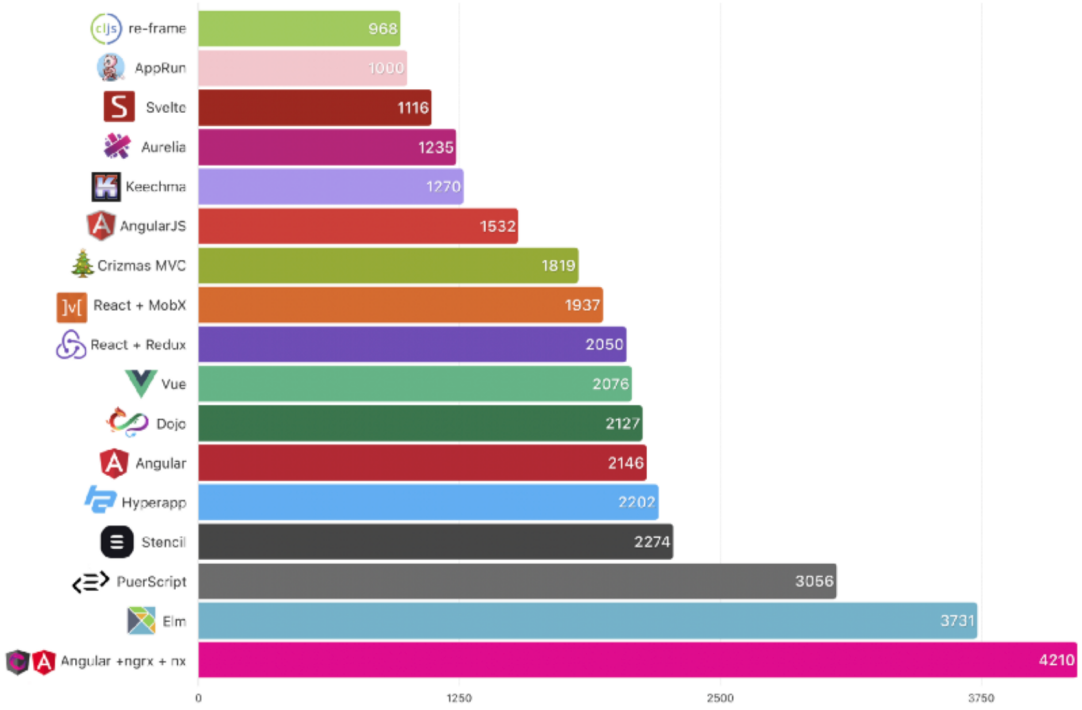
Obrázek 7: Benchmark velikostí aplikací (Schae, 2019)

3.3.3 Počet řádků kódu

Tento test je velice přímočarý – jedná se o srovnání počtů řádků kódu, které jsou potřeba pro vytvoření aplikace v dané technologii. Čím je číslo nižší, tím je práce efektivnější.

Tento test paradoxně vyšel v neprospěch technologie Elm. Po předchozím testu by se zdálo, že když je aplikace nejrychlejší a nejmenší ze vzorku porovnávaných konkurentů, musí obsahovat jednoznačně i nejméně řádků kódu, ale opak je pravdou. Ačkoliv v tomto testu může hrát určitou roli formátování kódu, kdy v Elm nástroje preferují zarovnání spíše pod sebe na více řádků, tak i přesto je rozdíl velicí významný.

Na prvním místě se umístil React v kombinaci s Reduxem s 2050 řádky kódu, na druhém Vue.js se 2076 řádky kódu, na třetím Angular s TypeScriptem s 2140 řádky kódu a konečně na čtvrtém místě Elm se 3731 řádky kódu.



Obrázek 8: Benchmark počtu řádků kódu v aplikaci (Schae, 2019)

3.3.4 Rychlost

K následujícím testům byly využity vzorové aplikace, které jsou veřejně dostupné v repozitáři js-framework-benchmark od užiatele krausest dostupné na webové adrese https://github.com/krausest/js-framework-benchmark.

Pro testování rychlosti frameworků v konkrétních scénářích byl využit přístroj MacBook Pro 14 s následující specifikací:

- 32 GB RAM,
- 8 jádrový CPU,
- 14 jádrový GPU,
- operační systém macOS 13.2.

Jako prostředí, ve kterém testy běžely, byl využit webový prohlížeč Chrome ve verzi 110.0.5481.77 za použití knihovny Puppeteer.

Pro porovnání byly vybrány následující čtyři technologie:

- Elm (verze 0.19.1),
- React (verze 18.2.0),
- Vue.js (verze 3.2.37),
- Angular (verze 15.0.1).

První série testových operací byla prováděna nad rozsáhlou tabulkou a jedná se o následující:

- vytvoření 1000 řádků,
- aktualizace 1000 řádků,
- aktualizace každého 10. z 1000 řádků,
- smazání tabulky s 1000 řádky.

Z výsledků (viz Tabulka 1 a Tabulka 2) je patrné, že technologie Elm dosahovala úctyhodných výsledků a geometrický průměr těchto testů byl shodný s Vue.js, zatímco Angular a React dosahovaly lehce delších časů.

Tabulka 1: Výsledky testů operací nad rozsáhlou tabulkou (zdroj: https://t.ly/vyOM)

Technologie	**elm-v0.19.1-3**	**vue-v3.2.37**	**angular-v15.0.1**	**react-hooks-v18.2.0**
Vytvoření 1000 řádků	48.4	46.3	47,9	48,2
Aktualizace 1000 řádků	50.2	48	52,6	53.8
Aktualizace každého 10. z 1000 řádků	119.5	124.3	107. 9	127.2
Smazání tabulky s 1000 řádky	38,6	40,4	71,5	60,6

Druhá série testů byla prováděna v prohlížeči za pomocí nástroje Lighthouse, což je integrovaný nástroj v prohlížeči Chrome, který analyzuje výkon webových aplikací. Pomocí tohoto nástroje byla zkoumána kompletní velikost aplikace společně se všemi zdroji nutnými pro načtení stránky a metrika Time to Interactive, což je stav, kdy hlavní

obsah stránky je načtený, hlavní vlákno není zaneprázdněné načítáním dat a je k dispozici uživateli pro interakci. (MDN Contributors, 2023c)
V obou z těchto testů byla aplikace Elm nejúspěšnější podle geometrického průměru.

Tabulka 2: Výsledky testů výkonu v prohlížeči (zdroj: https://t.ly/vyOM)

Technologie	**elm-v0.19.1-3**	**vue-v3.2.37**	**react-hooks-v18.2.0**	**angular-v15.0.1**
Metrika TTI	1876,5	2026,9	2551,8	2779,7
Velikost v KB	260,9	296,5	287,0	282,8

3.4 Porovnání vzorových aplikací

Pro praktickou část této práce byly vytvořeny dvě aplikace na zapisování a sledování úkolů se shodnými funkcemi. Tyto aplikace každá umožňují:

1. získání seznamu úkolů z externího zdroje,
2. hledání úkolů pomocí psaného vstupu,
3. přidávání úkolů,
4. smazání jednoho úkolu,
5. smazání všech úkolů,
6. označení úkolů jako splněný/nesplněný,
7. filtrování úkolů podle stavu úkolu,
8. smazání všech úkolů,
9. změnu jazyka,
10. navigaci na interní i externí stránky,
11. zobrazení momentálního času,
12. zobrazení šířky okna, ve kterém běží aplikace.

Pro první aplikaci byl zvolen jazyk Elm a pro druhou byla zvolena knihovna React potažmo jazyk JavaScript. Cílem je vyzkoušet dva různé přístupy ke tvorbě webových aplikací a poté porovnat zkušenost s každou z těchto technologií.

Práce s daty z externích zdrojů

Jednou z esenciálních funkcí, které jsou hojně využívány při vývoji webových aplikací, je práce s daty z externích zdrojů – například získávání a posílání pomocí dotazovacích metod HTTP protokolu. Samotný průběh těchto metod v obou technologiích není příliš rozdílný, nicméně zpracování dat se liší.

Pro konkrétní příklad lze porovnat dvě funkce sloužící k získání seznamu úkolů z externí adresy. V obou případech je možné využít různých přístupů, nicméně princip zůstává stejný. Cílem obou aplikací je na stisk tlačítka požádat o tato data a v mezičase zobrazit informaci o načítání. Jakmile aplikace data získá, vytvoří HTML obsah a zobrazí jej. V případě React aplikace je na stisk tlačítka zavolána funkce `sendRequest`, která nejprve ve stavu aplikace nastaví hodnotu `loading` na `true`. Na základě této informace aplikace zobrazí uživateli informaci o načítání. Díle tato funkce má dvě větve pro úspěšný a neúspěšný scénář. Při úspěšném získání dat nejprve parsuje data do JSON formátu a nastaví je do stavu aplikace do proměnné `todos` a zároveň nastaví hodnotu `loading` na `false`, aby aplikace věděla, že načítání je dokončeno a může zobrazit data. V případě, že získávání dat selže, provede se větev `catch`, která zavolá funkci, která se stará o zobrazení chybové hlášky uživateli a také nastaví hodnotu `loading` na `false`, protože načítání je dokončeno.

Tento přístup má výhodu v krátkém zápisu a své jednoduchosti, díky čemuž je možné ji poměrně rychle implementovat do aplikace. Při vývoji vzorové aplikace tato část byla opravdu rychlá a přímočará, nicméně v porovnání s přístupem, který byl aplikován v případě aplikace Elm, je poněkud křehká, protože může nastat jednoduchá změna ve struktuře dat, což by zapříčinilo selhání v běhovém prostředí. Zároveň tato data jsou získána naslepo a vývojář například přímo z kódu neví, jaké atributy tato data mají.

```
const sendRequest = async () => {
  setLoading(true);
  try {
    await fetch("https://dummyjson.com/todos")
      .then((response) => response.json())
      .then(({ todos }) => {
        setTodos(todos);
        setLoading(false);
      });
  } catch (error) {
    handleError("Error loading source...");
    setLoading(false);
  }
};
```

Výpis kódu 10: JavaScript funkce pro získání dat (vlastní zdroj)

Aby bylo možné stejná data získat ve vzorové Elm aplikaci, je zapotřebí napsat hned několik funkcí, což ústí ve větší počet řádků, ale tento přístup má svůj důvod a své benefity. Nejdříve je potřeba vytvořit samotnou funkci pro odeslání dotazu. V tomto konkrétním případě je nutné specifikovat, že se jedná o `GET` požadavek, dále uvést adresu, kam bude dotaz zaslán a poté definovat funkci `expect`. Cílem funkce `expect` je definovat, jaký výsledek je očekáván a jak zareagovat na odpověď. Uvedená funkce deklaruje, že očekává výsledek v podobě JSON objektu a získaná data nejprve převede to vlastního typu a poté je pošle dále do funkce, která patřičně zareaguje. A poslední důležitý argument je dekodér, který má za úkol zpracovat data a převést je na vlastní typ a tím tak zajistit konzistenci a přehled v aplikaci. V případě, že dekodér úspěšně zpracuje data, získáme naprostou jistotu, že data v aplikaci jsou přesně v tomto tvaru, do kterého byly úspěšně zdekódovány. V případě, že dekódování dat selže, tato data se nedostanou do aplikace. Tímto je zajištěno, že nenastane nečekaná změna struktury vedoucí k nekonzistentnímu stavu aplikace.

V této funkci neprobíhá změna stavu hodnoty načítání, protože tento stav je zahrnutý ve vlastním typu, na který jsou data převedeny v `expect` funkci a tím je možné kdekoliv v okolí těchto dat získat informaci o stavu tohoto požadavku (viz Výpis kódu 9).
Tento přístup má výhodu ve své spolehlivosti a bezpečnosti. Změna struktury dat neznamená chybu v běhovém prostředí a je možné poměrně snadno ošetřit tento případ. Zároveň převedení dat do vlastního typu usnadní vývojářům práci s těmito daty, protože díky tomuto typu je jasné, jakou mají strukturu.
Nevýhodou tohoto přístupu je rozhodně velký rozsah, který je potřeba pokrýt, aby bylo možné získat data. Zároveň tato funkce může být objektivně složitější na pochopení pro začínající vývojáře.

```
sendRequest : Decode.Decoder a -> (RemoteData.WebData a ->
msg) -> String -> Api -> Cmd msg
sendRequest decoder msg endpointSuffix (Api { baseApiUrl,
accessToken }) =
    Http.request
      { method = "GET"
      , headers = [ Http.header "x-customToken" accessToken ]
      , url = baseApiUrl ++ endpointSuffix
      , body = Http.emptyBody
      , expect =
        Http.expectJson
              (\response ->
                  response
                      |> RemoteData.fromResult
                      |> msg
              )
              decoder
      , timeout = Nothing
      , tracker = Nothing
      }
```

Výpis kódu 11: Elm funkce pro získání dat (vlastní zdroj)

Při vývoji vzorových aplikací bylo řešení, které nabízí React rychlejší a jednodušší na použití. Kvůli absenci typů byla však práce s daty mírně složitější, protože struktura dat nebyla natolik jednoznačná, jako tomu je v případě Elm přístupu. Tvorba funkce zodpovídající za získání dat se ukázala být objektivně složitější v případě Elm řešení. Ačkoliv bylo toto řešení složitější, má výraznou výhodu ve své konzistenci a spolehlivosti.

Architektura

Rozdíly v architektuře vzorových aplikací nejsou příliš velké zejména proto, že se jedná o malé aplikace. Tyto rozdíly by byly patrnější u větších projektů.

Hlavní rozdíl, který je možné zaznamenat, je počet řádků v souborech. Samotná dokumentace jazyka Elm uvádí, že není potřeba limitovat počet řádků kódu v jednom souboru za účelem většího přehledu nad aplikací. I přesto, že některé soubory v Elm aplikaci přesahovaly 500 řádků, neměl tento fakt negativní vliv na čitelnost či předvídatelnost aplikace. V případě druhé aplikace s sebou nesly velké soubory potíže. Ve velkých souborech bylo mnohem snazší udělat chybu, kterou je později složité odhalit. Z toho důvodu bylo příjemnější tyto soubory rozdělit.

Dalším rozdílem v architektuře těchto dvou aplikací byla tvorba komponent. Elm aplikace pro tvorbu komponent využívá řadu `view` funkcí, které žijí v souboru společně. Tento přístup měl pozitvní vliv na rychlost vývoje aplikace. V React aplikaci by toto vedlo k nepřehledným souborům, a tudíž bylo příjemnější vytvářet separátní soubory dedikované pro komponenty a poté je imporovat tam, kde je jich potřeba.

Debugování a refaktorování

Při tvorbě obou aplikací nastal stav, kdy momentální stav kódu nebyl funkční. Proces debugování (ladění) a refaktorování byl však u obou rozdílný.

Při vývoji Elm aplikace bylo debugování příjemnější a rychlejší hlavně z toho důvodu, že Elm kompilátor je při tomto procesu velice nápomocný. Chybové hlášky Elm kompilátoru jsou stručné a čitelné, takže se s nimi pohodlně pracuje. Zato debugování React aplikace bylo o poznání složitější právě kvůli absenci čitelných chybových hlášek. Také je vhodné zmínit, že při vývoji Elm aplikace nedošlo k žádné chybě v běhovém prostředí, kdežto v případě druhého řešení byl tento stav běžný.

Každá z aplikací byla v průběhu vývoje v určitých částech svého zdrojového kódu alespoň lehce refaktorována. Tento proces byl mnohem plynulejší a spolehlivější v Elm prostředí. Nápomocný byl zde zase kompilátor, protože dokáže naprosto přesně navádět na místa, kde jsou vyžadované změny a celý proces tak výrazně zrychluje a dodává při něm pocit jistoty. To bohužel nelze říci o druhém přístupu. Rozsah zdrojového kódu naštěstí není natolik velký, aby proces refaktorování naprosto zásadně zasáhl do funkčnosti aplikace, nicméně po zkušenosti s debugováním React aplikace je možné říci, že při tomto přístupu je snazší udělat kritickou chybu, kterou není snadné dohledat později.
Při tvorbě vzorových aplikací bylo potvrzeno, že technologie Elm je velice spolehlivá a nápomocná pro vývojáře.

Správa stavů

Obě aplikace určitým způsobem spravují svůj vnitřní stav.
React aplikace ke správě stavů využívá primárně `useState` hook, což je funkce, která bere na vstupu výchozí stav a vrací dva parametry – stav a setter funkci pro tento stav. Pro konkrétní případ poslouží správa stavu úkolu získaných z externího zdroje. Nejdříve je potřeba specifikovat název proměnné, setter a výchozí hodnotu. Později je setter využit při získání dat tedy konkrétně při načtení úkolů z paměti prohlížeče. V tomto okamžiku jsou již data dostupná v proměnné a je možné k nim přistoupit v další funkci `viewTodo`, která se stará o jejich zobrazení.

```
const [todos, setTodos] = useState([]);

useEffect(() => {
    setTodos(loadItemsFromStorage());
  }, []);

const viewTodo = (todo) => (
        <ToDoItem
          title={todo.todo}
          clickedTodo={() => handleClickedTodo(todo)}
          clickedDeleteTodo={() => handleDeleteTodo(todo)}
          completed={todo.completed}
          id={todo.id}
          key={todo.id}
        />
      );
```

Výpis kódu 12: Správa lokálního stav v React aplikaci (vlastní zdroj)

Elm aplikace naopak využívá zmíněnou TEA. Pro držení stavu slouží `Model` a pro jeho upravování slouží funkce `update`. Tento přístup je možné využít jednak na lokální úrovni a jednak i na globální úrovni.

Aby bylo možné například získat data z externího zdroje, je potřeba v Elm aplikaci inicializovat stav pomocí funkce `init`. Díky této funkci jsou k dispozici výchozí data. Změna těchto konkrétních dat nastane v souvislosti se získáním úkolů z externího zdroje. Při dokončení získávání je zavolána funkce `FetchedTodos` nesoucí datový obsah, který je zabalen do vlastního typu a má varianty určené pro stavy, které můžou při této akci nastat (viz Výpis kódu 9). Konkrétní úprava stavu nastává ve funkci `update`, ve které je provedena náležitá větev funkce odpovídající vstupujícímu typu. V této funkci probíhá nastavení těchto dat do modelu. Na změny modelu poté reaguje `viewToDos` funkce, do níž nyní vstupuje upravený model, ve kterém jsou dostupná data obalená vlastním typem.

V této funkci probíhá porovnání typů a je provedena patřičná renderovací funkce. V případě, že typ obalující data je `Loading`, značí to, že probíhá snaha o získání dat a uživateli je zobrazen vysvětlující text. Pakliže je tento typ `Failure`, pokus o získání dat byl neúspěšný a uživateli je zobrazena chybová hláška. A jakmile je typ `Success`, značí to, že pokus byl úspěšný, data jsou k dispozici a jsou zobrazena uživateli.
Objektivně je možné konstatovat, že Elm využívá techniku práce s daty, která je časově náročnější a vyžaduje větší množství napsaného kódu. Toto může prodloužit čas vývoje a také může přidat aplikaci na komplexitě. Na druhou stranu tento přístup vede ke zvýšené spolehlivosti aplikace a kontrolou nad ní. V průběhu tvorby vzorových aplikací byl příjemnější Elm přístup, právě díky kontrole nad stavem a daty. V případě, že by rychlost tvorby aplikace byla důležitou metrikou, React přístup se jeví jako lepší volba.

Výkon

Vzorové aplikace byly podrobeny stejným testům, které byly popsány v kapitole 3.3.
Hned při testování pomocí nástroje Lighthouse došlo k zajímavému výstupu. Elm aplikace dosáhla horšího výsledku než konkurenční React aplikace, přičemž výsledky v kapitole 3.3 tvrdí naopak. Důvodem může být přehnaná komplexita vzhledem k velikosti Elm aplikace, která vznikla při implementaci funkcí za účelem testování technologie.
Velikost aplikace byla také překvapivá v obou případech, protože Elm aplikace byla téměř dvakrát větší než React aplikace. Zde může být stejný důvod jako v předchozím testu.
Poslední test vyšel podle očekávání. Předmětem tohoto testu byl počet řádků kódu, který byl v případě Elm aplikace třikrát vyšší. (Příloha A: Výsledky testů aplikací)

3.5 Shrnutí výsledků

Elm v obecném srovnání s konkurencí zaostával, nicméně si vedl velice dobře v technologických aspektech.
Při dokazování testů z kapitoly 3.3 na menších vzorových aplikacích vytvořených v rámci praktické části této práce nebylo dosaženo ideálních výsledků. V těchto testech vzorová React aplikace dosahovala lepších výsledků. Důvodem pro tento výstup může být přehnaná komplexita na straně Elm aplikace spojená s implementováním funkcí v rámci testování schopností tohoto jazyka. Dále k tomuto výstupu mohl vést fakt, že tyto aplikace

jsou příliš malý vzorek pro testování tohoto rázu a technické přednosti jazyka Elm by se projevily až ve větším měřítku. Tyto výstupy potvrzují tvrzení, že jazyk Elm není vhodný pro tvorbu malých aplikací a hodí se spíše pro aplikace větší velikosti.

4 Závěr

Hlavním cílem této práce bylo analyzovat jazyk Elm z hlediska postavení na trhu a schopnosti konkurovat ostatním mainstreamovým front-endovým technologiím.

Dle výsledků této analýzy lze konstatovat, že programovací jazyk Elm momentálně není schopen konkurovat mainstreamovým technologiím z hlediska počtu uživatelů či množství pracovních míst a okolnosti nenaznačují jinak. I přesto má tato technologie své místo na trhu díky svým charakteristickým vlastnostem. Tyto vlastnosti, mezi kterými vyniká hlavně spolehlivost, jsou zejména užitečné při tvorbě rozsáhlých aplikací, které vyžadují vysokou míru spolehlivosti a bezpečnosti. Autor v době tvorby práce pracoval souběžně na fintechovém projektu, kde se tato technologie využívá a celému týmu velice napomáhá právě svými vlastnostmi, což také potvrzuje výstup výzkumu.

Představení jazyka, první z dílčích cílů, se podařilo naplnit v průběhu teoretické části. Tato část se zabývala primárně základy jazyka a měla za úkol ho čtenáři co nejvíce přiblížit.

Na úvodu praktické části byla technologie Elm porovnána s různými soudobými mainstreamovými technologiemi, které se vyskytují na trhu. Předmětem porovnání byl primárně výkon a v tomto srovnání byl Elm překvapivě úspěšný. Z této části lze odvodit, že technologické aspekty technologie Elm nejsou zodpovědné za to, že tato technologie není na poli vývoje webových aplikací příliš využívána, ale naopak jsou její silnou stránkou.

Posouzení příznivosti situace na trhu práce proběhlo na základě analýzy pracovních příležitostí na trhu práce a zveřejněných průzkumů mezi komunitou. Momentální situace pro technologii na trhu práce se v širším měřítku ukázala jako spíše nepříznivá kvůli složitosti hledání pracovních míst i pracovních sil. To může pro firmu dlouhodobě znamenat nutnost vkládání většího úsilí do udržení či získání pracovní síly, protože bylo dokázáno, že v porovnání s mainstreamovými technologiemi jsou tyto procesy náročnější.

I pro vývojáře není momentálně příliš příznivá situace na trhu práce kvůli poměrně nízkému počtu volných pracovních míst v porovnání s mainstreamovými technologiemi. Za uvedení však stojí fakt, že tato technologie je podle průzkumu atraktivní pro seniornější skupinu vývojářů, takže firmám může pomoci zasáhnout tuto cílovou skupinu. Z průzkumů také vyplývá, že z hlediska usnadnění hledání pracovní síly je pro firmu výhodné poskytovat práci na dálku.

Závěr praktické části byl věnován porovnání dvou vzorových aplikací, které byly vytvořené v rámci této práce. Ačkoliv porovnání dvou přístupů k programování může být velice subjektivní záležitost, autor se snažil pojmout tuto sekci co možná nejobjektivněji. V této části bylo potvrzeno, že Elm aplikace dosahovala výrazně lepších výsledků v oblasti spolehlivosti, což potvrzuje tvrzení autora tohoto jazyka a i díky tomu si zaslouží své místo mezi ostatními technologiemi sloužícími pro vývoj webových aplikací. Přestože React aplikace v tomto ohledu zaostávala, celková doba tvorby této aplikace byla znatelně kratší a v určitých částech i lépe čitelná a rychlejší na vývoj.

Doporučení

Na základě výstupu této práce lze doporučit využití této technologie pro projekty, které vyžadují vysokou míru spolehlivosti a jeví potenciál pro budoucí škálování. Naopak v malých či nenáročných projektech silné stránky technologie Elm nevynikají a tím ztrácí na své konkurenceschopnosti vůči mainstreamovým technologiím.

Pro firmy, které zvažují využití této technologie, je obzvláště důležité znát situaci na lokálním i globálním trhu práce, pokud je cílem zajistit dlouhodobý rozrůstající se projekt, protože nedostatek pracovní síly může znamenat potenciální překážku. Většině vývojářů lze tento programovací jazyk doporučit, protože nabádá k používání osvědčených přístupů. Tyto přístupy lze aplikovat i v jiných technologiích a těžit i v jiných oblastech. Nicméně pro juniorní programátory bez seniorního dohledu mohou být začátky objektivně náročnější než u tradičních JavaScriptových frameworků.

Seznam zdrojů

1. FAIRBANK, Jeremy. *Programming Elm: Build Safe and Maintainable Front-End Applications*. 1. Pragmatic Bookshelf, 2019. ISBN 978-1680502855.
2. SIMPSON, Kyle. *You Don't Know JS Yet: Get Started*. 1. Seattle: Amazon Publishing, 2020. ISBN 9798602477429.
3. CZAPLICKI, Evan · An Introduction to Elm. [online].2021a [cit. 2.10.2022]. Dostupné z: https://guide.elm-lang.org
4. CZAPLICKI, Evan. Elm: A delightful language for reliable web applications. [online]. 2021b [cit. 2023-04-09]. Dostupné z: https://elm-lang.org/
5. CZAPLICKI, Evan. *Elm Package Manager: Making it easy to share code* [online]. 2014 [cit. 2023-04-09]. Dostupné z: https://elm-lang.org/news/package-manager
6. CZAPLICKI, Evan. *My Thesis is Finally Complete!: Elm: Concurrent FRP for functional GUIs* [online]. 2012 [cit. 2023-04-09]. Dostupné z: https://www.reddit.com/r/haskell/comments/rkyoa/my_thesis_is_finally_complete_elm_concurrent_frp/
7. MICHAELSON, Gregory. Origins of functional languages. In: *An introduction to functional programming through Lambda Calculus*. B.m.: Dover publ, 1989, ISBN 978-0-201-17812-8
8. KUNASAIKARAN, Jagatheesan a Azlan IQBAL. A Brief Overview of Functional Programming Languages. *Journal of Computer Science and Information Technology* [online]. 2016, **6**(1), 32-34 [cit. 2023-01-18]. Dostupné z: http://103.227.140.18/index.php/ejcsit/article/view/97
9. FELDMAN, Richard. *Elm in Action*. Shelter Island: Manning Publications, 2020. ISBN 9781617294044.
10. BAALMAN, Philippus et al. Tour of Scala: Introduction [online]. École Polytechnique Fédérale, Lausanne, 2023a [cit. 2023-04-09]. Dostupné z: https://docs.scala-lang.org/tour/tour-of-scala.html
11. BAALMAN, Philippus. Tour of Scala: Community [online]. École Polytechnique Fédérale, Lausanne, 2023b [cit. 2023-04-09]. Dostupné z: https://docs.scala-lang.org/tour/tour-of-scala.html

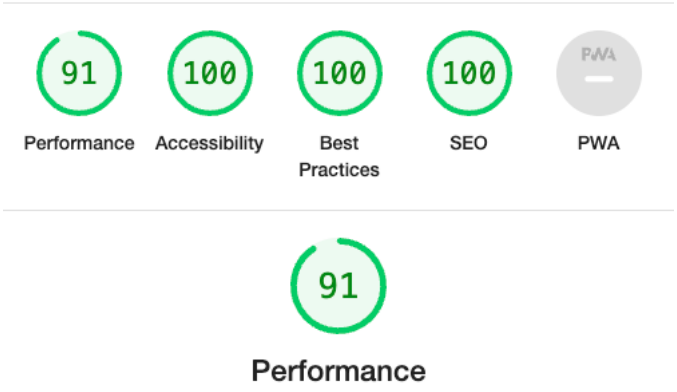
12. HUDAK, Paul, John HUGHES, Simon P. JONES a Philip WADLER. A History of Haskell: Being Lazy With Class [online]. San Diego, 2007 [cit. 2023-04-09]. Dostupné z: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf
13. FAUSAK, Taylor. *2022 State of Haskell Survey Results* [online]. 2022 [cit. 2023-04-09]. Dostupné z: https://taylor.fausak.me/2022/11/18/haskell-survey-results/
14. HANHINEN, Ossi a KLEMOLA, Matias. *Example app on managing shared state in large Elm SPAs. (ex elm-taco).* GitHub: Let's build from here · GitHub [online]. Copyright © 2023 GitHub, Inc. [cit. 26.03.2023]. Dostupné z: https://github.com/ohanhi/elm-shared-state
15. MCCARTHY, John. History of Lisp [online]. 1979, 1 [cit. 2023-04-09]. Dostupné z: http://jmc.stanford.edu/articles/lisp/lisp.pdf
16. Hosch, William L. "Robin Milner". Encyclopedia Britannica, 2023 [cit. 2023-04-09]. Dostupné z: https://www.britannica.com/biography/Robin-Milner
17. SYME, Don. The early history of F# [online]. Edinburgh, 2020, 4-11 [cit. 2023-04-09]. Dostupné z: doi: https://doi.org/10.1145/3386325
18. HICKEY, Rich. The Clojure Programming Language. *Clojure* [online]. 2022a [cit. 2023-04-09]. Dostupné z: https://clojure.org/
19. HICKEY, Rich. State of Clojure 2022 Results. *Clojure* [online]. 2022b [cit. 2023-04-09]. Dostupné z: https://clojure.org/news/2022/06/02/state-of-clojure-2022
20. MUGNAINI, Luca. *Elm 2022, a year in review* [online]. 2023 [cit. 2023-04-09]. Dostupné z: https://dev.to/lucamug/elm-2022-a-year-in-review-33pp
21. STEWART, Martin. *State of Elm 2022* [online]. 2022 [cit. 2023-04-09]. Dostupné z: https://state-of-elm.lamdera.app
22. MDN Contributors, *MDN Web Docs: JavaScript* [online]. 2023a [cit. 2023-04-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript
23. MDN Contributors, *MDN Web Docs: Web and web standards* [online]. 2023b [cit. 2023-04-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/The_web_and_web_standards
24. MDN Contributors, *MDN Web Docs: Time to interactive* [online]. 2023c [cit. 2023-04-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Time_to_interactive

25. EAMES, Joe. *RealWorld Example Apps* [online]. 2023 [cit. 2023-04-09]. Dostupné z: https://github.com/gothinkster/realworld
26. ROGIC, Mario. *Lamdera Quick Overview* [online]. 2023 [cit. 2023-04-11]. Dostupné z: https://dashboard.lamdera.app/docs
27. GALLINELLI, Nicholas. What Programming Language Should I Learn? *Flatiron School* [online]. 2021 [cit. 2023-04-15]. Dostupné z: https://flatironschool.com/blog/what-programming-language-should-i-learn/
28. GREIF, Sacha a Eric BUREL. Front-end Frameworks. State Of JS 2022 [online]. 2022 [cit. 2023-04-15]. Dostupné z: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/
29. SCHAE, Jacek. A RealWorld Comparison of Front-End Frameworks with Benchmarks. Free Code Camp [online]. 2019 [cit. 2023-04-15]. Dostupné z: https://www.freecodecamp.org/news/a-realworld-comparison-of-front-end-frameworks-with-benchmarks-2019-update-4be0d3c78075/

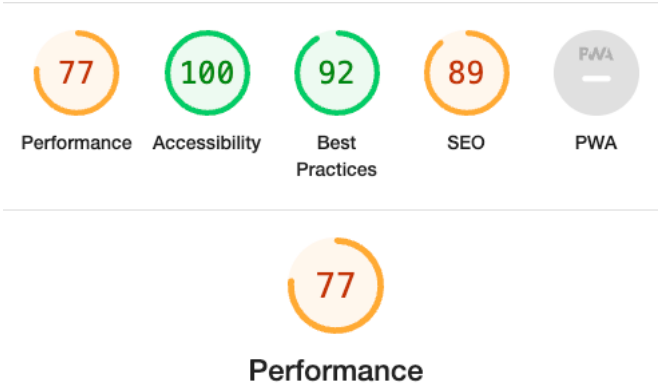
Přílohy

Příloha A: Výsledky testů aplikací

Lighthouse skóre – React

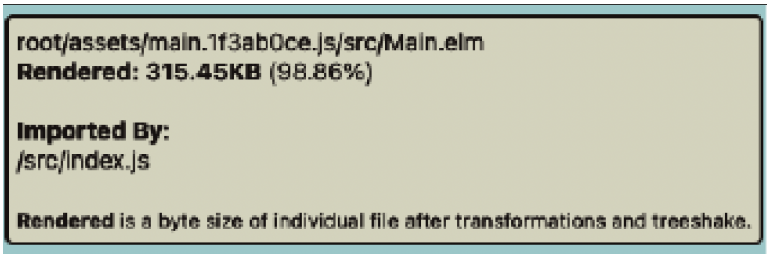


Lighthouse skóre – Elm



Velikost aplikace – React

Velikost aplikace – Elm



Počet řádků kódu – React

Language	files	blank	comment	code
JSX	10	28	0	369
CSS	2	7	0	159
JSON	2	0	0	60
JavaScript	5	5	7	49
SVG	1	0	0	1
SUM:	20	40	7	638

Počet řádků kódu – Elm

Language	files	blank	comment	code
Elm	12	501	38	1736
JavaScript	2	19	35	116
JSON	2	0	0	58
CSS	1	12	6	46
SUM:	17	532	79	1956

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Martin Bureš**
Studium:	I2000463
Studijní program:	B0688A140001 Informační management
Studijní obor:	Informační management
Název bakalářské práce:	**Programovací jazyk Elm a jeho využití pro vývoj webových aplikací**
Název bakalářské práce AJ:	Programming language Elm and its utilization for web development

Cíl, metody, literatura, předpoklady:

CÍL:

Hlavním cílem této práce je prozkoumat programovací jazyk Elm a představit jeho hlavní funkce a charakteristiky. Mezi další cíle patří:

1. Posoudit smysl využití této technologie pro vývoj webových aplikací,
2. Porovnat ji s konkurenčními technologiemi a přístupy,
3. Zhodnotit minulou a momentální situaci na trhu,
4. Vytvořit jednoduchou aplikaci sloužící pro demonstraci technologie.

OSNOVA:

1. Úvod
2. Představení programovacího jazyka Elm
3. Funkcionální programování
4. Programovaní pomocí Elmu
5. Porovnání s konkurencí
6. Závěr
7. Literatura

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Ing. Filip Malý, Ph.D.
Datum zadání závěrečné práce:	26.1.2021