Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Vývoj webových aplikací v jazyce Elm

Diplomová práce

Autor: Bc. Lukáš Richtrmoc
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové květen 2020

Prohlášení

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

V Praze dne 29. 4. 2020 Bc. Lukáš Richtrmoc

Poděkování

Děkuji společnosti ABRA Software a.s. a Univerzitě Hradec Králové za podporu distančního vzdělávání a formu, v jaké je možné studium absolvovat.

Anotace

Práce má za cíl popsat jazyk Elm a metodiky vývoje na webu v kontextu funkcionálního programování a vytvořit ukázkovou aplikaci pomocí tohoto jazyka. Teoretická část rozebírá různé typy a paradigma webového vývoje, následně seznamuje s funkcionálním programováním na pozadí lambda kalkulu. V praktické části se implementuje webová aplikace typu SPA, na které se popisují vlastnosti a charakteristiky jazyka Elm s technikami pro webový vývoj.

Annotation

Web Application Development Using Elm Language:

The thesis goal is to describe Elm language and web development methodics in context of functional programming and create presentable application in Elm. The theory part describes types and paradigms used in web development. Afterwards, it shows principles of functional programming with lambda calculus background. The practical part contains implementation of SPA web application and its description of Elm language characteristics with technical aspects for web development.

Obsah

1. Úvod ... 1

1.1. Důvod výběru tématu diplomové práce ... 1

1.2. Cíl diplomové práce ... 3

2. Webové prostředí ... 4

2.1. Historie ... 4

2.1.1. Web 1.0 ... 4

2.1.2. Web 2.0 ... 5

2.1.3. Web 3.0 ... 5

2.1.4. Web 4.0 a další ... 5

2.2. Principy ... 5

2.2.1. Navigace ... 6

2.2.2. Zobrazení ... 6

2.3. Technologie ... 6

2.3.1. Server ... 7

2.3.2. Client ... 7

2.4. Vývoj ... 9

2.4.1. Celostránkový ... 9

2.4.2. Komponentový ... 9

2.4.3. SPA ... 9

2.4.4. PWA ... 10

2.4.5. TWA ... 10

2.5. Omezení ... 10

2.5.1. Hardware ... 11

2.5.2. Operační systém ... 11

2.6. Paradigma ... 11

2.6.1. Imperativní ... 11

2.6.2. Deklarativní ... 12

2.6.1. Řízené událostmi ... 12

2.6.2. Multiparadigmatické ... 13

2.7. Současný stav ... 14

3. Koncept funkcionálního programování ... 15

3.1. Lambda ... 15

3.1.1. Kalkul ... 15

3.1.2. Výraz ... 15

3.1.3. Funkce ... 17

3.2. Funkce ... 18

3.2.1. Čisté ... 19

3.2.2. Anonymní ... 19

3.2.3. Pojmenované ... 19
3.2.4. Identitní ... 19
3.2.5. Vyššího řádu ... 19
3.2.6. Closure ... 20
3.2.7. Generické ... 20
3.3. Aplikace ... 20
3.3.1. Částečná aplikace ... 20
3.3.2. Kompozice funkcí ... 20
3.3.3. Currying ... 21
3.4. Vyhodnocení ... 21
3.4.1. Nestriktní ... 21
3.4.2. Striktní ... 22
3.5. Techniky ... 23
3.5.1. Typovost ... 23
3.5.2. Srovnání se vzorem ... 24
3.5.3. Funktor ... 24
3.5.4. Monáda ... 25
3.5.5. Polymorfismus ... 26
3.5.6. Rekurze ... 27
3.6. Vlastnosti ... 28
3.6.1. Funkce první třídy ... 28
3.6.2. Referenční transparentnost ... 28
3.6.3. Neměnnost dat ... 28
3.7. Dopady ... 29
4. Programovací jazyk Elm ... 30
4.1. Historie ... 30
4.2. Komunita ... 30
4.3. Použití ... 31
4.4. Distribuce ... 31
4.5. Instalace ... 31
4.6. Kompilace ... 32
4.7. Modularita ... 34
4.8. Prostředí ... 34
4.9. Struktura ... 34
4.10. Syntaxe ... 35
4.10.1. Anotace ... 35
4.10.2. Definice ... 36
4.10.3. Operátory ... 36
4.11. Typy ... 38
4.11.1. Primitivní ... 39

4.11.2. Strukturované ... 40
4.11.3. Vlastní ... 42
4.11.1. Nepojmenované ... 43
4.11.2. Opaque types ... 43
4.12. Práce s proměnnými ... 44
4.12.1. Globální ... 44
4.12.2. Model ... 44
4.12.3. Lokální ... 45
4.13. Řídící struktury ... 45
4.13.1. If … then … else … ... 46
4.13.2. Case … of ... 46
4.13.3. Funktory ... 46
4.14. Vyhodnocení ... 47
4.15. Architektura ... 48
4.15.1. Virtuální DOM ... 48
4.15.2. MVU ... 48
4.15.3. Delegování zpráv ... 51
4.15.4. URL ... 53
4.15.5. HTTP ... 54
4.15.6. Interakce s JavaScriptem ... 56
4.16. Balíčky ... 58
4.17. Implementace ... 58
4.18. Ukázky ... 59
5. Webová aplikace napsaná v jazyku Elm ... 61
5.1. Analýza ... 61
5.2. Backend ... 63
5.2.1. Server ... 64
5.2.2. Databáze ... 64
5.3. Frontend ... 65
5.4. Programování ... 65
5.4.1. Tvorba API ... 66
5.4.2. Business logika ... 70
5.4.3. Prezentační vrstva ... 74
5.4.4. Web Push Notifications ... 75
5.5. Výsledek ... 76
5.6. Testování ... 79
6. Diskuze ... 81
6.1. Dotazník ... 81
6.2. Výhody ... 81
6.3. Problémy ... 83

6.4. Doporučení ... *87*
6.5. Zamyšlení ... *88*
7. Závěr ... **89**
8. Seznam zdrojů ... **90**
8.1. Tištěné zdroje ... *90*
8.2. Internetové zdroje ... *91*
9. Přílohy ... **94**
9.1. Příloha č. 1 – Ukázkový případ Higher-Order Delivery ... *94*
9.2. Příloha č. 2 – Dotazník: Uzavřené otázky ... *95*
9.3. Příloha č. 3 – Dotazník: Otevřené otázky ... *96*

Seznam obrázků

Obrázek 1 - Debugger ukazuje vlevo zprávy, vpravo model. Zdroj: Vlastní..................... 33
Obrázek 2 - Stromové rozdělení modulu a publikování funkcí. Zdroj: Vlastní................. 34
Obrázek 3 - MVU architektura Elmu. Zdroj: Tensor (2018). 49
Obrázek 4 - Vyvolání a odchycení HTTP dotazů. Zdroj: Poudel (2018)........................... 55
Obrázek 5 - Mobilní fintechová aplikace. Zdroj: savr.com... 59
Obrázek 6 - SPA jako task management. Zdroj: www.abra.eu.. 60
Obrázek 7 - Use case diagram projektovače. Zdroj: Vlastní... 62
Obrázek 8 - Diagram tříd projektovače. Zdroj: Vlastní... 63
Obrázek 9 - Úvodní obrazovka aplikace. Zdroj: Vlastní... 77
Obrázek 10 - Výpis projektů. Zdroj: Vlastní... 77
Obrázek 11 - Darování částky. Zdroj: Vlastní... 77
Obrázek 12 - Odebírání zpráv. Zdroj: Vlastní... 77
Obrázek 13 - Mobilní zobrazení daru. Zdroj: Vlastní. 78
Obrázek 14 - Mobilní zobrazení formuláře. Zdroj: Vlastní. 78
Obrázek 15 - Zobrazení notifikace na mobilu. Zdroj: Vlastní. 78
Obrázek 16 - Formulář nového projektu: Zdroj: Vlastní... 78
Obrázek 17 - Výsledek testu z Pingdom Tools. Zdroj: Vlastní.. 79
Obrázek 18 - Výsledek testu z Google LightHouse. Zdroj: Vlastní................................... 79
Obrázek 19 - Protokol úrovně implementace PWA. Zdroj: Vlastní................................... 80

Seznam tabulek

Tabulka 1 - Charakteristiky strukturovaných datových typů. Zdroj: Vlastní..................... 40

Seznam ukázek

Ukázka 1 - Struktura modulu. Zdroj: Vlastní. 35

Ukázka 2 - Anotace funkce s dvěma aritami. Zdroj: Vlastní. 35

Ukázka 3 - Rozdíl mezi pojmenovanou a anonymní funkcí. Zdroj: Vlastní. 36

Ukázka 4 - Skládání funkcí. Zdroj: Vlastní. 37

Ukázka 5 - Operátory. Zdroj: Vlastní. 38

Ukázka 6 - Typová inference. Zdroj: Vlastní. 39

Ukázka 7 - Strukturované datové typy. Zdroj: Vlastní. 41

Ukázka 8 - Vlastní algebraické datové typy. Zdroj: Vlastní. 43

Ukázka 9 - Práce s proměnnými a řídící struktury. Zdroj: Vlastní. 45

Ukázka 10 - Funktory a monády. Zdroj: Vlastní. 47

Ukázka 11 - Nejprimitivnější rozložení MVU aplikace. Zdroj: Vlastní. 51

Ukázka 12 - Parsování URL. Zdroj: Vlastní. 54

Ukázka 13 - Zpracování HTTP dotazu. Zdroj: Vlastní. 56

Ukázka 14 - Spuštění zkompilovaného Elmu. Zdroj: Vlastní. 57

Ukázka 15 - Použití portů. Zdroj: Vlastní. 58

Ukázka 16 - Generování schéma z frontendu. Zdroj: Vlastní. 67

Ukázka 17 - Basic OAuth na klientu a serveru. Zdroj: Vlastní. 69

Ukázka 18 - Model aplikace. Zdroj: Vlastní. 71

Ukázka 19 - Zjednodušená část High-Order Delivery. Zdroj: Vlastní. 72

Ukázka 20 - Napojení formuláře na event onSubmit. Zdroj: Vlastní. 73

Ukázka 21 - Porty pro odesílání zpráv napříč aplikací. Zdroj: Vlastní. 74

Ukázka 22 - Komponenta Theme pro prezentační vrstvu. Zdroj: Vlastní. 75

Ukázka 23 - Odesílání Web Push Notificatons. Zdroj: Vlastní. 76

Seznam použitých zkratek

ADT	Algebraic Data Type
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
FRP	Functional Reactive Programming
GUI	Graphical User Interface
JS	JavaScript
JSON	JavaScript Object Notation
HTML	Hypertext Markup Language
HTTP	Hypertext transfer protocol
MVC	Model-View-Controller
MVU	Model-View-Update
NPS	Net Promoter Score
PHP	PHP: Hypertext Preprocessor
PWA	Progressive Web Application
REPL	Read-Eval-Print-Loop
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SPA	Single-Page Application
TWA	Trusted Web Activities
UI	User Interface
URI	Uniform Resource Indentifier
URL	Uniform Resource Locator
UX	User Experience Design
WWW	World Wide Web
W3C	World Wide Web Consortium

1. Úvod

1.1. Důvod výběru tématu diplomové práce

Svět webu je pulsující organismus, který se mění s ohromnou rychlostí. Pryč jsou doby statických skriptovacích jazyků, dnešní doba žádá dynamičnost a efektivitu ve vysokém tempu – to vše za předpokladu minimální chybovosti.

Aby tyto nároky mohly být uspokojeny, je třeba se s nimi vyvíjet. Vývojářská komunita nespí a do webových trendů se dostávají nové, či dříve okrajové, techniky a nástroje. Jedněmi z nich je i funkcionální programování a tlustý klient.

Ukázalo se totiž, že deklarativní přístup na klientské části prohlížeče pomáhá „zkrotit" některé problémy spojené s držením stavu a vedlejšími efekty, jako jsou vstupy od uživatele, synchronizace volání apod., které historicky byly bolestí u imperativních jazyků.

Jazyk Elm není první, ani poslední, funkcionální jazyk pro tvorbu webového obsahu. Avšak jeho uvedení v roce 2012 způsobilo oživení zájmu a ovlivnilo další průkopníky, jako technologii Redux (Kears, 2019). Od té doby uplynul dostatek času, aby se dalo konstatovat, že si Elm našel své místo uplatnění a dostatek příznivců. A to navzdory vstupní náročnosti, kterou klade na frontend vývojáře z řad webových imperativních jazyků.

Ačkoliv jsou k dispozici dokumentace, výukové kurzy a určité množství literatury, stále jde o jazyk, jehož popsání (nejen) v kontextu tvorby webu je nedostatečné. Weboví vývojáři, stejně jako designéři, jsou dost dobře obeznámeni s množstvím technik týkajících se tvorby obsahu, vzhledu a dynamiky v rámci znovupoužitelnosti, avšak při použití Elmu začínají na zelené louce a nemohou si poradit, jak takové znalosti překlopit do rutinní práce ve funkcionálním jazyku. Příkladem může být jak prosté vytvoření zalamujícího se textu v paragrafu, tak vytvoření továrny na tvorbu šablonovitých stránek. V rámci frameworků typu *Nette*, *Django* či *Wicket* jde o známé vzory, které se však zdají být v Elmu neproveditelné – což není pravda.

Současná nabídka informací tak silně pokulhává nad poptávkou o těchto technikách. V rámci práce byly rešeršovány vydané knihy o Elmu. Imsirovic (2018) a Fairbank (2019) přichází vesměs s podobnou strukturou obsahu, ve kterém mluví spíše o syntaxi jazyka a jednom modelovém případu vytrženém z kontextu. Loder (2018) popisuje praktičtěji tvorbu jednostránkových aplikací v Elmu včetně zabalení do bundlovacích nástrojů a různých druhů komunikace. V době psaní této práce Feldman v březnu (2020) „konečně" dostatečně rozčeřil vody, ve kterých se na více jak 350 stranách věnuje podrobněji širším aspektům

jazyka, od efektivity datových struktur přes routování či dekódování grafových modelů až po fuzzy testy. Funkcionálnímu a matematickému vysvětlení se přesto nedostává.

Na poli vypracovaných absolventských prací byla situace před Feldmanem lepší. Holappa (2018) a Jiroudek (2018) se věnovali základům jazyka a implementaci jednoduchých jednostránkových neperzistentních aplikací pohledem klasického webového developera. Bůna (2018) rozšířil přehled o kompletnější výčet funkcionálních prvků, jež doplňují Luxemburkovo (2017) širší pojetí, včetně nakousnutých matematických aspektů. O *lambda kalkulu*, základu funkcionálních jazyků, zevrubně informuje Kadlecaj (2018), ovšem bez pojítka k samotnému jazyku. Slivka (2018) nastínil architekturu pro zpracování složitějších (vícestránkových) aplikací. Mladý (2017) pečlivě porovnává klady a zápory mezi vývojem v Elmu a v JavaScriptu (konkrétně při použití Flow a Reactu), Eriksson a Ärleryd (2016) se věnovali přímo implementaci funkcionálních prvků z Elmu do JavaScriptu.

Důvodem výběru práce je tak zpracování odpovědi na otázku, která by se lidsky dala shrnout následovně: *„Vím, že musím udržitelně generovat HTML, CSS a JavaScript, ale jak se k tomu mám dostat pouhým skládáním funkcí?“* Na pohled jednoduchá, avšak obsahově vyčerpávající otázka, jež v sobě skrývá podbody:

- Reprezentaci webového prostředí a vlastností prohlížečů
- Datovou strukturu
- Mapování objektů
- Modularizaci kódu
- Ukládání citlivých dat
- Oboustrannou komunikaci
- Cachování

Kromě problematiky webu se práce zabývá i funkcionálním programováním a jeho matematickým pozadím v *lambda kalkulu*, jelikož základní pochopení těchto oblastí dokresluje některé černé skříňky, které je jinak nutné z pohledu vývojáře imperativních jazyků pouze slepě akceptovat. Též je pro práci důležité sjednocení terminologie, která se napříč jazyky syntakticky liší, ale sémanticky překrývá. Pokud je to možné, upřednostňovány jsou české názvy před cizími.

Závěrem byla pohnutkou pro výběr práce i otázka, kolik takových aplikací může být vyvíjeno a v reálném provozu nasazeno, či jak si vývojáři poradili s výše zmíněnými problémy.

1.2. Cíl diplomové práce

Cílem práce je zdokumentovat tvorbu webového obsahu funkcionálním programováním a vytvořit webovou aplikaci, konkrétně v jazyku Elm.

V první části se jedná o definici webu, jakožto nástroje pro výměnu informací. Následuje seznámení s různými paradigmy a samotným konceptem funkcionálního programování, ve kterém je text rozdělen na části věnující se *lambda kalkulu* a vlastnostem funkcionálních jazyků.

Druhá část aplikuje teoretické poznatky do prostředí jazyka Elm, kde je nejprve představena a probrána koncepce ekosystému. Následuje implementace aplikace, na které jsou ukázány typické úkony při tvorbě SPA aplikace s PWA prvky. V diskuzi figurují výsledky z dotazníku o reálném použití Elmu mezi vývojáři a dále se rozebírají výhody, problémy a doporučení vzešlé z implementace.

2. Webové prostředí

Prostředí webu je zde téměř přes třicet let. Za tu dobu se výrazně proměnilo, jak po informativní, tak funkční stránce. Počet uživatelů vzrostl na téměř 4 a půl miliardy (Clement, 2019) a je tak bezpochyby největší komunikační sítí mezi lidmi.

2.1. Historie

Počátky však nebyly tak masivní, jako jsou nyní. Síťová komunikace před zavedením standardu World Wide Web (WWW) byla nejednotná a neposkytovala takovou univerzálnost, jakou by potřebovala. Především byl problém, jak nejlépe prolinkovat jednotlivé dokumenty a přidat informacím v dokumentu sémantický význam (Frost, 2016). Proto Tim Berners-Lee navrhl, a poprvé v roce 1990 použil (Wältera, 2019), značkovací jazyk zvaný Hypertext Markup Language (HTML), jež nese doplňující informace na odesílaných značkách.

Ze shluku prostých textových značek do takzvaného Document Object Modelu (DOM) se dá sestavit grafický obraz, který odpovídá standardům popsaných konsorciem (World Wide Web Consortium, též jako W3C), založeným též Leeem. S pomocí kaskádových stylů (Cascading Style Sheets, též jako CSS) a skriptovacího jazyka JavaScript (JS) je možné přidat vykreslované stránce na vizuální bohatosti a dynamičnosti (Wältera, 2019).

Jako milník v produkci webových stránek lze považovat přidání asynchronního volání z JavaScriptu (Asynchronous JavaScript and XML, též jako Ajax) v roce 2005 (Wältera, 2019). Od té doby se mohly začít přenášet po síti pouze části informací, nikoliv nutně celé dokumenty. Tato změna zapříčinila zaměření vývojářů na JavaScript jako na plnohodnotný jazyk pro tvorbou aplikací (Bůna, 2018).

Od prvního použití na začátku devadesátých let se postupem času měnily pohledy, k čemu by měl webový prostor sloužit. Rozlišují se hlavně tři období, takzvané weby 1.0, 2.0 a 3.0, o dalších panuje debata (Francesconi, 2018).

2.1.1. Web 1.0

První etapa představovala provázané statické dokumenty, většinou jako prezentace firemních a osobních stránek (Francesconi, 2018). Interaktivita s uživatelem byla omezená (etapa se též označuje jako *read-only* web), převážně šlo o vyplňování formulářů a odesílání emailů (Prabhu, 2016). Vyhledatelnost dokumentů začaly řešit první firmy (Google, Bing, Yahoo, …) pomocí rozpoznávání klíčových slov z obsahu značek (Francesconi, 2018).

2.1.2. Web 2.0

Po začátku tisíciletí se web začal více rozvíjet mezi technologicky obyčejné uživatele. Důraz byl kladen na skupiny a komunity (Prabhu, 2016), což vedlo ke vzniku řady sociálních platforem (MySpace, Facebook, Linkedin, …) či blogovacích, komunikačních a sdílecích služeb. Interaktivita se zvýšila, z dnešního pohledu se mluví o dynamické etapě či *read-write* webu. Do webového prostoru také pronikli první mobilní uživatelé, jež následně výrazně přispěli k dalšímu růstu (Prabhu, 2016).

2.1.3. Web 3.0

S příchodem třetí generace webu dochází k zaměření na jednotlivce (Francesconi, 2018). Personalizace a porozumění uživatelskému chování jsou hlavními aspekty současného webového prostoru, například pro nabízené služby (Prabhu, 2016). Pokrytí těchto služeb má široký záběr a pomalu dochází k odstranění rozdílů mezi desktopovou a webovou aplikací. K porozumění uživateli se využívá, kromě již dříve zmíněného rozpoznávání klíčových slov, zavedení sémantičnosti (například zavedením ontologie do HTML značek); též se mluví o webu třetí generace jako o inteligentním (Prabhu, 2016), sémantickém či *machine-read* webu (Domingue a kol., 2011).

2.1.4. Web 4.0 a další

Dle Francesconiho (2018) je ovládnutní sémantičnosti důležité pro další stupeň webové generace, která bude využívat inteligentní autonomní agenty.

Tito softwaroví agenti na zpracovaných *smart datech* budou provádět interakci mezi sebou a uživatelem, čímž vznikne symbiosa; odtud pragmatický či *symbiotický* web. Na základě vybudované symbiosy budou agenti moci předvídat chování uživatele, informace vyhledat a předložit řešení. Příkladem může být osobní kalendář a vyhledávání spojů při každodenní rutině, koupě ubytování či letenek v obvyklé době dovolené a jiné.

Autor dále rozpracovává myšlenku ohledně webu páté generace, ve kterém se budou reflektovat emoce uživatele; půjde tedy o *emoční web*. Na základě emocí bude web reagovat určitou akcí či nabídkou k akci, podobně jako v symbiotickém webu. Příkladem může být nabídka lístků do kina/na koncert při špatné náladě.

2.2. Principy

Dle Domingue a kol. (2011) jde na webu o uspokojení dvou základních věcí:

- Zpřístupnění informací.
- Delegaci výkonu.

Zajistit je to možné díky principům, na kterých stojí současný web:

- Otevřenost a vzdálený přístup.
- Interoperabilita (hardwarových či softwarových prostředků).
- Decentralizace.
- Automatizace (předávání požadavků a odpovědí).
- Zajištění vztahů M:N (anonymita).

2.2.1. Navigace

Přeskakování z jednoho dokumentu do druhého je možné díky unikátnímu jménu (Uniform Resource Indentifier, URI) a unikátnímu odkazu (Uniform Resource Locator, URL; Lee a kol., 2005). Na této adrese se může v dokumentu nacházet více prvků, každý je možný unikátně identifikovat fragmentem (Domingue a spol., 2011). Koncept **URI** a **URL** je univerzálně použitelný. Na webu se využívá pro navigaci prokládáním URL adres uvnitř dokumentu, kde jsou vykreseleny jako klikatelné odkazy. JavaScript dává možnost navigovat mezi dokumenty i bez jasně viditelné instance odkazu, v takovém případě se však ztrácí informace o jiném dokumentu.

2.2.2. Zobrazení

Informace uložené značkovacím jazykem HTML je možné zobrazovat vícero způsoby. Asi většina webových vývojářů má ještě v paměti rozkol v interpretaci pravidel mezi různými webovými prohlížeči. Ovšem vedle notoricky známého grafického vykreslení je možné zobrazit data pouze jako podbarvený text (Lynx) či v audiu (Google metalmouth).

W3C konsorcium založilo iniciativu Web Accessibility Initiative, která vydává pokyny (Web Accessibility Initiative Document) k implementaci webového standardu (Burgstahler, 2015). V dokumentu se uvádí, co vše musí implementace splňovat, aby byla přístupná nejen pro běžné uživatele, ale i pro uživatele s hendikepy jako je barvoslepost, hluchota, limitace pohybu a jiné.

2.3. Technologie

Architektura webu je založena na klientské a serverové části. Klientská část vysílá pomocí protokolů požadavky na server identifikovatelný URL adresou a následně zpracovává odpovědi (Domingue a kol., 2011). Je možné, aby si server po prvotním požadavku zapamatoval umístění klienta a poté samovolně odesílal odpovědi bez žádosti klienta (Loder,

2018). Klient může poskytovat služby a být tak v jednom okamžiku též „serverem“. Mechanismy pro skládání odpovědí a vzájemnou komunikaci zajišťují různé technologie.

2.3.1. Server

Server, též nazývaný jako *backend*, je část, která se stará o datovou vrstvu a obsluhu odpovědí na poskytované služby (Wälter, 2019). Hlavní náplní práce serveru je detekce dotazu, zjištění stavu klientské části, manipulace s perzistentními daty a vygenerování odpovědi. Server sedí většinou na statickém umístění v síti a spuštěným programem na pozadí (službou ve Windows, daemonem v Linuxu) naslouchá dotazům na konkrétním portu.

V začátcích šlo o odpovědi, ve kterých se kontrolovala pouze existence dotazovaného souboru. Určitou dynamičnost přinesly odpovědi obsluhované přes CGI, kdy se spustil skript v konkrétním jazyce (C, PHP…), jež dynamicky vytvářel obsah pomocí tisku na standardní výstup. Později byly využívány komplexnější webové servery jako Apache, Nginx či IIS, které nabízely i další funkcionality. Konečně, samotné jazyky obsahují vlastní řešení generování odpovědí, například *Servlety* v Javě, Python a jeho *http.server* či v JavaScriptu napsaný *Node.js*. Kapitolou dneška jsou pak pokročilé webové frameworky (*Wicket*, *Django*) starající se o nejrutinnější části tvorby odpovědí, včetně perzistence dat.

2.3.2. Client

V kontextu webu též *frontend*, je klientská část, ve které dochází k vyvolávání požadavků a vykreslování odpovědí (Wälter, 2019). Mezi různými typy zobrazení odpovědí (probraných v kapitole 2.2.2) má výsadní postavení vykreslování v prohlížeči, jež využívá technologie kaskádových stylů a skriptování. Veškeré změny jsou neperzistentního rázu, respektive přestanou být dostupné při stejném požadavku z jiného klienta.

Vývoj klientské části aplikace prochází bouřlivými změnami, především od již zmíněného roku 2005, kdy aplikace dostaly do vínku možnost částečných odpovědí, což podnítilo rozvoj nových přístupů vývoje a tedy i nároků na samotný jazyk. Kromě mnoha jiných rozšíření to bylo například přidání gridovacích technologií FlexBox v CSS či implementace deklarativních technik do JavaScriptu. A též využití výpočetního výkonu klientských stanic.

Canvas, WebGL a WebGPU

Kontrolu nad vykreslením HTML prvků má pouze prohlížeč. Dodržet pevný vzhled je možné absolutním pozicováním, nebo pomocí obrázků. Do konceptu značkovacích jazyků

zapadá především vektorový formát SVG, jehož značky definují rozložení obrazu a zároveň mohou být přidány do struktury HTML. Pak je obrázek pevnou součástí dokumentu a je možné na jednotlivé tagy navázat posluchače událostí. Cenou je však opět podrobení vykreslení prohlížeči, což má za následek například automatické překreslování při manipulaci s tagem.

V rámci HTML 5 byla představena (ale již dříve implementována a využívána) značka *<canvas>*. Jde o prostor, do kterého je možné pomocí JavaScriptu vykreslovat 2D grafické elementy a obarvovat jednotlivé pixely (Nyman, 2013). Jedná se o čistě klientskou technologii na nízké úrovni, jež nenese žádnou sémantickou hodnotu a narozdíl od vektorů je překreslována pouze na příkaz v kódu.

Pro pokročilejší práci bylo implementováno rozhraní WebGL přímo do JavaScriptu. WebGL je webovou kopií OpenGL včetně možností používat shadery (vertex a fragment) s univerzálním jazykem GLSL. Pro výpočty negrafických primitiv (compute shader) či přímějšímu přístupu k možnostem (kompatibilní) grafické karty se dá využít novější rozhraní WebGPU, které však není přímo propojené s WebGL (Beaufort, 2019).

WebAssembly

Konsorcium W3C vydalo 5. 12. 2019 do světa zprávu o čtvrtém jazyku (vedle HTML, CSS a JS), který je možné spustit ve webovém prohlížeči. Jde o WebAssembly (též Wasm), jehož virtuální stroj v prohlížeči spouští kód jako nativní překlad (*assembler-like*).

Kód může být vygenerován z většiny mainstreamových jazyků, jako je C, C++, Java, Rust či Go; z Elmu prozatím – mimo neoficiální implementace – nikoliv. Po vygenerování je nutné kód zanést do prohlížeče, načež mohou být zpřístupněny v JavaScriptu metody z původního kódu. Metody běží “mimo” prohlížeč a mají přímější přístup k procesoru klienta (Jylänki, 2017).

V důsledku zavedení WebAssembly je možné spouštět v prohlížečí náročné úlohy, jež by bez komponenty třetí strany (dříve Java Applety či Flash) byly nemyslitelné. Díky kompilaci dosahuje úloha několikanásobně menší velikosti než původní zdrojový kód.

WebAssembly nalezne využití u dedikovaných rutin (např. regulární výrazy), matematických úloh, strojového učení či zpracování grafiky. V kombinaci s WebGL je možné zatížit jak GPU, tak CPU, což otevírá prostor pro plnohodnotné herní zážitky (Jylänki, 2017). Zranitelností zůstává zapojení klientské stanice do nekalých praktik, jako je nevědomé těžení kryptoměn (Cimpanu, 2020).

2.4. Vývoj

Vývoj webového softwaru je poplatný době, avšak vždy je bezpodmínečně nutný server[1]. Na serveru jsou umístěny zdroje pro tvorbu obsahu a perzistentní data. Samotnou tvorbu je možné uskutečnit jak na serverové, tak na klientské části. Dle Wältera (2019) rozlišujeme různé druhy vývoje podle obsahu a umístění business logiky.

2.4.1. Celostránkový

Též *server side web app*. Jde o vývoj čistě na straně serveru, kde je tvořena veškerá logika a prezentační vrstva (Wälter, 2019). Celostránkový vývoj je prapředkem všech webových vývojů. Jeho použití způsobuje zbytečné přenášení opakujících se neměnných částí, například hlavičky a patičky, což vede k delším prodlevám. Na druhou stranu není nutné řešit synchronizaci změn mezi různými komponentami, přenáší se celá HTML strana.

2.4.2. Komponentový

Též *widget web app*. Mezi klientem a serverem se přenáší pouze úsek HTML kódu, který se na klientské části dynamicky vymění (Wälter, 2019). Tento přístup byl umožněn zpřístupněním ajaxového volání v roce 2005 (viz kapitola 2.1). Jeho použití šetří přenášený objem dat, na druhou stranu je většinou stejně nutné konstruovat jaderné části aplikace, které by při *stateless* režimu mohly být ušetřeny. Při tvorbě zůstává většina business logiky na serveru, zároveň je částečně přenositelná na klientskou část. Použití metody orientované na komponenty je i dnes populární, příkladem mohou být frameworky Wicket (Java), Nette (PHP) či Vaadin (Java).

2.4.3. SPA

SPA je zkratkou pro *Single-page Application*. Jedná se o konfiguraci, ve které klient obsahuje téměř veškerou business logiku – jde o takzvaného tlustého klienta (Wälter, 2019). Persistentní data jsou stále na serveru, předávána a upravována přes vzdálené volání API (*application programming interface*). Druhy API mohou být orientovány na služby (SOAP, RPC), nebo na zdroje (REST). Přístup k API je pak regulován autentizací (například OAuth).

Tvorba SPA aplikací pouze pomocí klienta má své klady a zápory. Předně je minimalizováno generování značek na serveru, tvorba HTML tak závisí pouze na síle klienta. Počet přenesených dat je diskutabilní a záleží na složitosti aplikace a implementaci API; velice pravděpodobně se bude přenášet méně dat, ale zato dojde k více volání. Navíc

[1] I pro Service Workers, které po prvotní instalaci mohou fungovat offline.

vzniká režie na synchronizaci volání a celého modelu, který je nutné držet na klientské části. Zároveň často dochází k duplikaci business logiky, kdy se na frontendu „imituje“ chování backendu. Pro ušetření několika dotazů dovolují některá API sdružená (*batch*) volání.

SPA aplikace je možné vytvořit pouze na klientské úrovni, existují však řešení, která spojují i serverovou část. Google vytvořil Framework GWT, kde je co nejvíce logiky předáno přes JavaScript na klientskou část, zbytek se provede v Javě na serveru. Pozoruhodnou implementaci provedl Microsoft. V jazyku C# s nástrojem Blazor je možné celou aplikaci provozovat buď na straně serveru, nebo komponentami, nebo jako SPA ve WebAssembly.

2.4.4. PWA

Je zkratkou pro *Progressive Web Application*. Termín byl zaveden v roce 2015 a sdružuje nejlepší praktiky z webového vývoje pro mobilní zařízení (Russell, 2015). Mezi tyto praktiky patří responzivita, webový manifest (např. určení ikony k uložení na plochu, velikost otevřené aplikace a další), využití *Service Workerů* (systémové notifikace, offline chování) a přístup k hardwarovým zdrojům. Obecně jde o umožnění pocitu, že se web chová (nejen) na mobilním zařízení jako nativní aplikace. Zavedení manifestu je základem, největší výzvou pak kompletní offline provoz s odloženou synchronizací. Zobrazení i výpočet stále zůstává na prohlížeči v takzvaném režimu *WebView*; PWA tedy nedosahují výkonu ani privilegií nativních aplikací.

2.4.5. TWA

Google v roce 2019 přinesl vzpruhu pro PWA přes takzvané *Trusted Web Activities*. Jde o možnost zobrazení webu v rámci instalované nativní aplikace z obchodu Google Play (McLachlan, 2019). TWA je na rozdíl od PWA spouštěna přímo Chromem, nikoliv pouze jako *WebView*. Má tak přístup k funkcím objevujících se v prohlížeči Google Chrome. Bohužel je TWA pouze proprietárním řešením Androidu, stejně tak je omezena k pevně svázané doméně.

2.5. Omezení

Při webové tvorbě je nutné počítat s různými omezeními. Ve většině případů jde o bublinu, kterou kolem sebe prohlížeč striktně vytváří a nepovoluje ji překročit. V této alegorii zůstávají mimo bublinu některé systémové služby a hardwarové zdroje.

2.5.1. Hardware

Přístup k hardwaru byl dříve velmi omezený. Obejít ho šlo povolením doplňků třetích služeb (například již zmíněnými Applety a Flashem). Od jejich zakázání byly prohlížeče postupně obohacovány o JavaScriptová rozhraní k prostředkům jako je kamera, poloha, stav baterie, BlueTooth zařízení a jiné. Naopak umírá přístup k zapojeným USB zařízením, jehož jedinou implementací ponechal funkční Google Chrome (Yacoub, 2020).

Alternativním řešením je tak pouze vytvoření nativní mezivrstvy (např. služby naslouchající na portu) schopné dostat se ke zdroji a přes ajaxová volání z webu příkaz provést.

2.5.2. Operační systém

Ve standardním rozhraní je možné získat informace o vybraném souboru či lokální paměti. Pak jsou zde *Service Workers*, jež mohou vytvářet systémové notifikace (Gaunt, 2019). Zároveň jsou umístěny mezi komunikací prohlížeče a serveru, takže mohou odpovídat za server vlastním obsahem (např. daty uloženými v cache lze simulovat odpovědi při ztrátě připojení). *Web Workers* slouží k paralelizaci procesů, například vzdáleného volání (MacDonald, 2019). Zapovězen je stále přístup ke kontaktům či souborovému systému.

2.6. Paradigma

Programovat lze různými způsoby. Od prvopočátků se vyvinula různá paradigma, která více či méně sedí na konkrétní problematiky. V kontextu webu dle Wältera (2019) existují dva hlavní proudy – imperativní a deklarativní.

2.6.1. Imperativní

Imperativní paradigma nechává v implementaci problému volnější ruku programátorovi. Hlavními prvky jsou přiřazení, cykly a větvení, pomocí nichž mění aplikace svůj stav proměnných (Wälter, 2019). Pro vývojáře je tok programu přirozenější, ale klade větší nároky na správu běžných operací, což může nepřímo vést ke zhoršené konzistence a predikovatelnosti kódu (Harrison, 1997).

Jedním z nejčastějších představitelů imperativního způsobu programování je paradigma objektově orientované, doplňující procedurální. Jako imperativní jazyky se označují například **Java**, **Ruby** nebo **C#**.

2.6.2. Deklarativní

Deklarativní paradigma nechává konkrétní implementaci na interpretu jazyka. Jde o kontrastní rozdíl oproti imperativním jazykům. Abstrahuje tak programátora od rutinních operací, což může mít jak pozitivní, tak negativní důsledky (Wälter, 2019). Hlavními prvky jsou návratové hodnoty z výrazů a rekurze.

Funkcionální programování je představitelem deklarativního paradigmatu. V něm jsou hodnoty zjišťovány vyhodnocováním výrazů matematických funkcí (Harrison, 1997). Stav aplikace, jakožto proměnných, by se v jazyku objevovat neměl; pokud tomu tak je, nejedná se o čistý funkcionální jazyk (Harrison, 1997).

Programovací jazyk **Lisp** byl vyvinut v roce 1956 jako teprve druhý programovací jazyk (Luxemburk, 2017) a dá se považovat za první implementaci funkcionálního paradigma, které představil Alonzo Church v roce 1936. Jazyk jako první přidal koncept spojových seznamů. Na jeho základy navázal (v rámci dalšího vývoje) důležitý funkcionální jazyk **Meta Language**

Po zrodu několika dalších jazyků byla v roce 1987 ustanovena standardizace. Z ní vzešel v roce 1990 **Haskell**, jenž představuje renesanci funkcionálního programování. Jeho výhodnost tkví (nejen) v kompilátoru, který dokáže generovat spustitelné soubory pro různá prostředí, včetně klientské (Miso) i serverové (Servant, Spock) části webu. Haskell je též jedním z mála hojně používaných čistých funkcionálních jazyků (Alexander, 2017).

Ačkoliv má Haskell i webové frameworky, tak jeho použití může být pro běžné webové vývojáře příliš náročné. Výhody funkcionálního programování a snížené vstupní požadavky na znalosti splňuje **Elm**. Elm byl představen v roce 2012 a slibuje „žádné runtime chyby“ (Czaplicki, 2012), které jinak často provází frontendový vývoj. Kombinuje přístup čistých funkcionálních jazyků a udržitelné správy tzv. side-efektů při komunikaci s okolním světem.

2.6.1. Řízené událostmi

Anglicky jako *event-driven*. Jde o případ, ve kterém je aplikace řízena pomocí pozorovatelných událostí (Čížek, 2019). Paradigma může být implementováno jak v imperativním, tak deklarativním přístupu. Velice často je spojováno s ovládáním grafického prostředí (*Graphical User Interface*, GUI).

Reaktivní programování

Konkrétním případem řízení událostmi je reaktivní programování, ve kterém systém reaguje asynchronně na změny z okolního světa (Czaplicki, 2012). Vyznačuje se rozdělením na samostatné komponenty emitor-konzument a paralelním zpracováním (Čížek, 2019). V reaktivním programování se výsledná hodnota automaticky aktualizuje vždy, když se změní hodnoty z ní skládající.

Funkcionálně reaktivní programování

Anglicky též jako *Functional Reactive Programming* (FRP) je speciálním případem použití reaktivního programování, konkrétně pomocí deklarativních technik. Nejde však o pouhé spojení těchto dvou paradigmat (Čížek, 2019), ale o rozlišení proměnných závislých na čase spojitém (chování – *behaviors*) a diskrétním (události – *events*). Tato implementace FRP se nazývá *Classical*, avšak je neefektivní. Řešení nabízí *Real-time* a *Event-driven*, jež mají pro chování i událost zástupný název **signál**, který nabývá existenciální hodnoty něco-nic[2]. Konečné výkonnostní doladění přineslo zavedení *signálních funkcí* v implementaci *Arrowized* (též AFRP), které jsou konceptuálně rovné běžným funkcím a lze s nimi provádět efektivní vyhodnocování (Czaplicki, 2012).

2.6.2. Multiparadigmatické

V dnešní době většina nejrozšířenějších jazyků abstrahovala funkcionality napříč paradigmaty, přesto jsou vnímány dle jejich dominantního použití. Například valná většina programů reaguje na události, ale nelze je považovat pouze za *event-driven*. Také objektovou Javu lze psát funkcionálním přístupem (immutabilitou proměnných typem final, využitím anonynmních funkcí atd.). Takovéto jazyky lze nazvat nečisté (*impure*).

Některé jazyky však vědomě kombinují nejrůznější přístupy a jsou tak v tomto směru nevyhraněné, takzvaně multiparadigmatické. Jedná se například o **PHP**, dříve čistě skriptovací jazyk, dnes často využívající objektové funkcionality jako třídy a rozhraní. Nepříliš známo jest, že je možné využívat také funkcionální techniky, jako anonymní funkce a *lambda výrazy*, a to již od roku 2009.

Podobně se vyvíjel **JavaScript**, který si však prošel kolečkem od funkcionálního k objektovému paradigma a zase zpět. Dalším příkladem může být jazyk **Go** vyvinutý

[2] Jde o takzvaný algebraický datový typ viz kapitola 3.5.1; konkrétně se jedná o typ *Maybe* s možnostmi *Just x* a *Nothing*, kde *x* je jakákoliv možná hodnota signálu.

Googlem v roce 2009 jako odpověď na nedostatky C++. Program je možné kompilovat na různé platformy, také jako WebAssembly. V podobném duchu je od roku 2005 kostruován **F#** z dílny Microsoftu. Kombinováním různých přístupů je možná kompilace na rozdílná prostředí, do JavaScriptu (Fable) či programů grafických jednotek.

2.7. Současný stav

V současnosti dle Wältera (2019) je moderní vyvíjet webové aplikace technologií SPA, ve kterých je „okleštěný" **tenký server** a „bohatý" **tlustý klient**. Server může být řešen pouze jako API poskytující perzistentní data, veškerá business logika a vykreslování jsou pak umístěny na klientovi. Dle autora je trendem vybírání nejlepších prvků z různých paradigmat a jejich vzájemné promíchání. Nicméně se jedná spíše o obohacení aktuálně dominantního imperativního paradigma o deklarativní prvky, především funkcionální.

3. Koncept funkcionálního programování

Jak již bylo zmíněno, funkcionální programování stojí na postupném vyhodnocování výrazů. Na rozdíl od imperativního přístupu je deklarativní přístup, potažmo funkcionální, více spjat s matematickými objekty, což mu zaručuje lepší transparentnost a korektnost. (Harrison, 1997). Zároveň mu dává dostatečnou abstrakčnost nad jazykem a strojem (Michaelson, 2011), takže explicitně neinstruuje, ale pouze deklaruje vlastnosti (Harrison, 1997).

3.1. Lambda

Historie funkcionálního programování se datuje do první poloviny minulého století, ve které položili teoretické základy Church, Rosser, Curry a další. Church přišel s řeckým symbolem λ (lambda) jakožto konceptu funkce (Alexander, 2017). Toto označení bylo spíše náhodné a odkazovalo na dřívější značení ^, publikované v *Principia Mathematica* z roku 1910 (Harrison, 1997). Označení lambdy je v určitém kontextu možné chápat i jako synonymum k anonymní funkci (například syntaxe *lambda: f* v Pythonu).

Z matematického základu vyvstal velký počet vlastností, které dělají funkcionální programování unikátní a také vcelku striktní.

3.1.1. Kalkul

Pokud *lambda* označuje funkci, tak *lambda kalkul* celou oblast týkající se funkcí (Alexander, 2017). Jde o formální popis funkcí a jejich vztahů pomocí *lambda notací* (Harrison, 1997). Základním prvkem je *lambda výraz*.

3.1.2. Výraz

Lamda výraz označuje matematický výraz, který je spojen z několika částí, jež jsou z pohledu *lambda kalkulu* validní. Dle Michaelsona (2011) a Harrisona (1997) jde o **proměnnou** reprezentující hodnotu:

$$x$$

Jež sama o sobě reprezentuje lambda výraz. V použití s jiným *lambda výrazem* (*M*) dochází k **abstrakci**:

$$\lambda x. M$$

Kde je vztah vázaného neznámého (*bound*) parametru *x* do těla (*body*) jiného výrazu *M*. Při analogii z klasického matematického zápisu funkce libovolného těla *t[x]*:

$$f(x) = t[x]$$
$$e.g.: \ f(x) = x^2$$

Jde v Churchově zápisu o úpravu do *lambda* zápisu:

$$\lambda x.t[x]$$
$$e.g.: \lambda x.x^2$$

Kde symbol *lambdy* označuje vázanou proměnou. Takto upravený *lambda výraz* je vyjádřením jiné notace, a sice se šipkami:

$$x \ \rightarrow t[x]$$
$$e.g.: x \rightarrow \ x^2$$

Ve kterém je vztah vázaného neznámého (*bound*) parametru *x* do těla (*body*) výrazu *t[x].*

Díky tomuto pojetí je možné do argumentu předávat i jiné funkce, stejně tak je mít v těle výrazu (Michaelson, 2011). Tečkový zápis zavedený Churchem se používá převážně v matematice, šipkový (s určitými obměnami) v programování pro anotaci funkcí (Haskell, Elm).

K úplnosti chybí poslední část *lambda výrazu*, a sice **aplikace** argumentů jiných *lambda výrazů* (N):

$$(M \ N) \ \ zapsané \ též \ jako \ (M)(N)$$

Což v praktickém významu může znamenat zápis následujících výrazů, které se vyhodnotí *částečnou aplikací* jako hodnota 3:

$$(\lambda x.x \ 3) \ či \ (\lambda y.y + 1 \ 2) \ či \ (\lambda z.z + 1 \ \lambda y.y + 1 \ 1)$$
$$také \ zapsané \ jako$$
$$(\lambda x.x) \ (3) \ či \ (\lambda y.y + 1) \ (2) \ či \ (\lambda z.z + 1)(\lambda y.y + 1)(1)$$

Poslední výraz je možné přepsat na výraz, který přijímá dva parametry:

$$(\lambda z.\lambda y.(z + y + 1)) \ (1) \ (1)$$

Fakticky však v *lambda kalkulu* nic jako „dva parametry“ není (Kadlecaj, 2018). Výrazy mohou pracovat pouze s jednou aritou (unární operace), nikoliv více aritními (binární operace a spol.).

Řešením je tak úprava výrazu na unární přes takzvané *curryování*. Podstatou je, že se při takovém zápisu vytvoří na sebe navazující výrazy, jež se postupně vyhodnotí stejně jako v předchozím případu (více v kapitole 3.3):

$$\begin{aligned}
1:&\quad (\lambda z. z + (\lambda y. y + 1)(1))\ (1)\\
2:&\quad (\lambda z. z + 2)\ (1)\\
3:&\quad (1 + 2)
\end{aligned}$$

Původní výraz lze také řešit jinou metodou, která se nazývá *β-redukce*. Jde o základní operaci *lambda kalkulu*, která redukuje výraz aplikováním jiného výrazu:

$$\begin{aligned}
1:&\quad ((\lambda z. z + 1)(\lambda y. y + 1))\ (1)\\
2:&\quad (\lambda y. y + 1 + \ 1)\ (1)\\
3:&\quad (1 + 1 + 1)
\end{aligned}$$

Kromě *β-redukce* jsou dalšími operacemi *α-substituce* a *η-konverze* (Michaleson, 2011). Postup vyhodnocování výrazů záleží na zvolených strategiích, jejichž výčet je rozebrán v kapitole 3.4.

3.1.3. Funkce

Lambda funkce je zapouzdřením *lambda výrazu* (těla) a argumentu, které je použito v jejich zástupnosti (Michaelson, 2011). Volně přeloženo, jde o jmenné určení, jež se dá obecně vyjádřit následovně:

$$<function> ::= \lambda<arg>.<body>$$

Pomocí *lambda funkcí* se vyjadřují základní výrazy nutné pro rutinní práci, jako booleovské hodnoty, logické operátory, datové struktury či standardní sady utilit. Příkladem může být postup pro vyjádření booleovské hodnoty pravda:

$$true ::= \lambda x. \lambda y. x$$

Který akceptuje dva parametry, ale vrátí pouze první. Nepravda se definuje analogicky:

$$false ::= \lambda x. \lambda y. y$$

Při vytvoření logické operace *and* je pak možné vytvořit výraz, který bude využívat již vytvořených funkcí:

$$and ::= \lambda a.\lambda b.((a)(b)(false))$$

Pro ověření lze aplikovat na funkci *and* argumenty *true* a *false*, které po dosazení a *β-redukci* se zobrazí jako *false*:

$$\begin{aligned}
&1: && \lambda a.\lambda b.((a)(b)(false))(true)(false) \\
&2: && \lambda b.((true)(b)(false))(false) \\
&3: && (true)(false)(false) \\
&4: && (\lambda x.\lambda y.x)(\lambda x.\lambda y.y)(\lambda x.\lambda y.y) \\
&5: && \big(\lambda y.(\lambda x.\lambda y.y)\big)(\lambda x.\lambda y.y) \\
&6: && (\lambda x.\lambda y.y)
\end{aligned}$$

Pro *true* a *true* je výsledek *true*:

$$\begin{aligned}
&1: && \lambda a.\lambda b.((a)(b)(false))(true)(true) \\
&2: && \lambda b.((true)(b)(false))(true) \\
&3: && (true)(true)(false) \\
&4: && (\lambda x.\lambda y.x)(\lambda x.\lambda y.x)(\lambda x.\lambda y.y) \\
&5: && \big(\lambda y.(\lambda x.\lambda y.x)\big)(\lambda x.\lambda y.y) \\
&6: && (\lambda x.\lambda y.x)
\end{aligned}$$

Podobně jsou definovány další operace (Michaelson, 2011), jako *not*, *xor*, *identity* (vrácení sebe sama), predikáty, hodnoty a také rekurze, která je kruciálním prvkem funkcionálního programování (Alexander, 2017).

Z výše uvedeného plyne, že jde pouze o zřetězení velkého množství funkcí a v čistém funkcionálním přístupu se stav nedrží. Aby byl *lambda kalkul* spojitě použitelný, zavádí se vnitřní stav pomocí konstruktu *let* (Harrison, 1997), který se drží v celém výrazu jako pojmenovaná funkce, předávaná neustále dokola v rekurzivní funkci.

3.2. Funkce

Pojem funkce – o kterém je ve funkcionálním programování řeč – tedy vychází z definice *lambda funkce*. Funkce může být popsána určitými vlastnostmi, jež popisují chování uvnitř funkce nebo dopad funkce na okolí. Níže následují vybrané druhy vlastností, které se mohou nacházet v disjunkci.

3.2.1. Čisté

Čisté funkce (anglicky *pure functions*) splňují výše vypsaná pravidla *lambda kalkulu*. Nemají vliv na vnější okolí, jejich výstup závisí pouze na vstupu a za žádných vnějších okolností se nemění (Alexander, 2017). V kontextu funkcionálních jazyků jsou čisté funkce všudypřítomné, ovšem v imperativních jazycích má smysl takovéto označení použít.

3.2.2. Anonymní

Function literal je bezejmenná funkce. Původem je to jakákoliv funkce objevující se v *lambda kalkulu* (Harrison, 1997). Proto se v některých jazycích označuje také jako *lambda*. Hlavní využití (u imperativních jazyků, ale i u deklarativních se stavem) nachází anonymní funkce u krátkodobých výpočtů, většinou na jedno použití (Alexander, 2017). Může se jednat o mapování, redukci, zpřehlednění mezivýpočtu apod.

3.2.3. Pojmenované

Kromě výše zmíněného pojmenování *let,* lze funkci pojmenovat i postfixově pomocí *where* (Harrison, 1997). Pojmenované funkce se mohou předávat a tím udržovat stav, který je immutabilní (viz kapitola 3.6.3). Pomáhají s optimalizací nároků na rekurzi, kdy tzv. koncovým voláním šetří zásobník volání (Harrison, 1997; Alexander, 2017).

3.2.4. Identitní

Jedná se o typ funkce, která vrací sama sebe (Harrison, 1997). V *lambda kalkulu* jde o výraz:

$$\lambda x. x$$

Užitečnost takové funkce se projevuje především při ukončování podmínek rekurzí (viz kapitola 3.5.1), či zachování statusu quo při kompozici funkcí.

3.2.5. Vyššího řádu

V angličtině též *Higher-Order*, je funkce, která je schopna jako argument přijmout funkci, stejně tak ji vrátit jako výsledek (Harrison, 1997). Označení má opět smysl především u imperativních jazyků, jelikož drtivá většina deklarativních jazyků je na předávání funkcí založena.

U některých imperativních jazyků je tato možnost již součástí mechanismů práce s daty (například procedurální ukazatele v C), jinde dochází k postupnému rozšíření vlastností týkající se *lambda kalkulu* (Java, Python, JavaScript a další), nebo je možné vytvořit vlastní implementaci (Alexander, 2017).

3.2.6. Closure

Počeštěna jako kložura, je funkce, která si pamatuje svůj stav, potažmo stav, ve kterém byla vytvořena. Taková možnost dává flexibilitu při opakovaném volání, například inkrementaci nebo aplikaci funkce (Khot a Mishra, 2017). Vytvoření closure může způsobit, že funkce přestanou být čisté, to však záleží na implementaci (Alexander, 2017).

3.2.7. Generické

Též polymorfické, jsou funkce, které využívají parametrizaci pomocí typů (Loder, 2018), viz polymorfické vlastnosti (kapitola 3.5.5).

3.3. Aplikace

Funkce jsou na sebe běžně aplikovány v argumentu, jak bylo naznačeno ve vyhodnocování *lambda výrazu*. Kromě přístupu částečnou aplikací, se rozlišují další dva případy – kompozice funkcí a curryování (Michaelson, 2011).

3.3.1. Částečná aplikace

Anglicky též *partial application*, je základním případem vyhodnocování výrazů. Přijímá výraz a vrací o jeho vyhodnocení, byť jen částečné. Při vícenásobném vyhodnocení funkcí X, Y, Z:

$$X \rightarrow Y \rightarrow Z$$
$$je\ výsledkem$$
$$Y \rightarrow Z$$

Není tak nutné, aby částečná aplikace prošla všechny výrazy; další kroky záleží čistě na operacích následujících po částečném vyhodnocení, čehož se využívá u tzv. řetězení (*chainování*) funkcí.

3.3.2. Kompozice funkcí

V kompozici funkcí dochází k postupné aplikaci dvou (a více) funkcí (Alexander, 2017), kdy výsledek první funkce je předán do druhé. Základ pochází z kompozice funkcí v algebře:

$$v\ klasickém\ zápisu: \quad (f \circ g)(x) = f\big(g(x)\big)$$
$$v\ lambda\ výrazu: \quad \lambda f.\lambda g.\lambda x.f(g(x))\ \text{či}\ \lambda f.f(\lambda g.g(\lambda x.x))$$

Ve volném zápisu na příkladu funkcí A, B, C:

$$X := A \rightarrow B$$
$$Y := B \rightarrow C$$
$$plyne$$
$$X \rightarrow Y \quad či \quad (A \rightarrow B)\,(B \rightarrow C)$$
$$tedy$$
$$A \rightarrow C$$

3.3.3. Currying

Oproti částečné aplikaci přebírá *currying* všechny argumenty a vnitřně z nich vytváří unární funkce:

$$W \rightarrow X \rightarrow Y \rightarrow Z$$
$$na$$
$$W \rightarrow (X \rightarrow (Y \rightarrow Z))$$

Ve výsledku tak dochází k vyhodnocení celého výrazu částečnou aplikací, a tedy k postupné „akceptaci" všech položek iniciační funkcí.

3.4. Vyhodnocení

Aplikaci funkcí doplňuje strategie pro prioritizaci vyhodnocování, též nazývaná jako *reduction strategy* (Harrison, 1997). Strategie určuje, který výraz se vyhodnotí z mnoha možných jako první pomocí určení směru (zleva, zprava) a zanoření (vnitřní, vnější). Hlavními proudy jsou **striktní** a **nestriktní** vyhodnocení. Cílem je dosáhnout **normálního tvaru** (*normal form* či *β-normal form*), ve kterém nepůjde již nic dalšího redukovat (Michaelson, 2011).

3.4.1. Nestriktní

Anglicky *non-strict* či *non-eager*, je vyhodnocení, při kterém se vnitřní funkce vyhodnotí až v poslední možný okamžik (Khot a Mishra, 2017). Tedy jde o odložené vyhodnocování (také *on purpose*) vnějších funkcí zleva doprava, shora dolů – též nazývaném normální pořadí či anglicky **normal order**.

Díky tomu je možné šetřit vyhodnocení až do okamžiku, než je funkce opravdu použita. To však přináší problém, kdy stejné funkce mohou být vyhodnoceny vícekrát (Michaelson, 2011), například:

$$
\begin{aligned}
1&: \quad (\lambda x. x + x)(2 + 2)\\
2&: \quad ((2 + 2) + (2 + 2))\\
3&: \quad (4 + 4)
\end{aligned}
$$

Mezi nestriktní vyhodnocení se řadí modifikace **call by name**, ve které je pojmenovaná funkce vyhodnocena teprve zavoláním jména. A též jde o variantu **call by need** (synonum pro líné vyhodnocení, anglicky *lazy evaluation*), ve které dojde k uložení výsledku funkce do paměti a k následnému opětovnému použití bez nového vyhodnocení (Harrison, 1997).

Nestriktní techniky používají i striktní strategie – například se při vyhodnocení booleovské funkce OR nevyhodnocuje druhý parametr, pokud se první rovnal *true*.

3.4.2. Striktní

Ve striktním vyhodnocení (též nazývaném *strict* či *eager*) jsou všechny argumenty funkcí vyhodnoceny ještě předtím, než jsou do funkce předány (Khot a Mishra, 2017). Toto způsobuje dvě věci (Harrison, 1997):

- Funkce mohou být vyhodnoceny, ačkoliv nebudou potřeba.
- Rekurzivní funkce nemusí skončit.

Ukázka zbytečného vyhodnocení z předchozího příkladu nestriktního vyhodnocení:

$$
\begin{aligned}
1&: (\lambda x. y)(2 + 2)\\
2&: \quad (\lambda x. y)(4)\\
3&: \quad y
\end{aligned}
$$

Na druhou stranu lze vyhodnotit vnitřní výraz jen jednou, pokud je ve vnějším použit vícekrát. Například:

$$
\begin{aligned}
1&: \quad (\lambda x. x + x)(2 + 2)\\
2&: \quad (\lambda x. x + x)(4)\\
3&: \quad (4 + 4)
\end{aligned}
$$

Při vyhodnocování se postupuje zleva doprava, z nejvíce vnitřního k vnějšímu. Pravidlo se nazývá jako aplikativní pořadí, anglicky **applicative order** (Michaelson, 2011).

Mezi striktní vyhodnocení se řadí **call by value** (česky *volání hodnotou*, někdy také zaměnitelně *pass by value*) a **call by reference** (*volání referencí*). Jde o trochu odlišné

implementace výše popsaného principu, například v jazycích C, C++, Scala a jiné (Harrison, 1997; Alexander, 2017).

Speciálním případem je **parciální vyhodnocení** (*partial evaluation*), které však nemá co dočinění s parciální aplikací popsanou v kapitole 3.3.13.3. Parciální vyhodnocení trochu porušuje striktní strategii a pokračuje do těla nevyhodnoceného výrazu, kde vyhodnotí či redukuje některé výrazy dříve, než to striktní přístup povoluje (Harrison, 1997).

Použití

Obě vyhodnocení mají svá pro i proti. Mezi překladači převažuje **striktní** vyhodnocení, především kvůli náročnosti implementace. V reálném prostředí se musí při nestriktním vyhodnocení držet kopie všech výrazů (Michaelson, 2011), což představuje problém.

Proto je lepší, když se program vyhodnocuje striktně a odložená volání se nechají na programátorovi (technikami *call by name* a *call by need*) – tím dochází k prolnutí obou strategií a nejefektivnějšímu použití (Khot a Mishra, 2017). Díky striktnosti je také možné lépe ladit program, jelikož lze přesně určit hodnoty aktuálního vyhodnocení (Harrsion, 1997).

3.5. Techniky

Pro praktickou práci s funkcemi se implementují různé techniky. Některé jsou podpůrné (typovost, srovnání se vzorem, funktory), jiné nezbytné (rekurze).

3.5.1. Typovost

Zavedení typovosti není povinné (vše může být anonymní funkcí), avšak je důležitým prvkem pro rychlejší vyhodnocování výrazů, kompilaci a efektivnější správu paměti; obecně lze říci, že statická typovost (kdy po přiřazení již není možné změnit typ proměnné) zlepšuje výkon (Harrison, 1997). Nehledě na to, že funguje jako doplňková dokumentace kódu a chybových hlášek. I z těchto důvodu je většina funkcionálních jazyků silně typově orientovaná.

Je dobré si uvědomit, že ve funkcionálních jazycích je téměř vše nějaký typ a pokud není uživatelem definován, je mu při vyhodnocování zástupně genericky vytvořen.

Obyčejná

Mezi obyčejné (též primitivní) typy se počítají typy přiřazující literály (numerické, znakové) či funkce, ať už vlastní, nebo jazykem připravené (Harrison, 1997).

Algebraická

Algebraický datový typ (ADT) shlukuje další typy pod jeden zástupný typ. Je velmi důležitý pro polymorfismus a srovnávání se vzorem (viz dále). V algebraických datových typech se rozlišují dvě varianty: *product type* a *sum type*.

Product type (jinak též *tuple*) je založen na výčtu typů, které musí typ nutně obsahovat (Alexander, 2017), například ve formě *(a, b, c, ...)*. Matematicky jde o kartézský součin. Často je *product type* zastoupen strukturou, která obsahuje typy jako povinné položky, například datová struktura záznam *{ f1: a, f2: b, f2: c,...}*.

Sum type (jinak také jako *tagged/discriminate union*, *union/variant type*) obsahuje výčet typů jako své „potomky", respektive nabývá jedné hodnoty z jím deklarovaných typů. Z matematického hlediska se jedná o disjunktní sjednocení/kompozici (Harrison, 1997). Zápis je většinou proveden ve formě oddělení typů rourou (*a* | *b* | *c* | ...).

Příkladem ADT může být hodnota boolean, který nabývá typů pravdy a nepravdy.

Inference

Typová inference je vlastností, která říká, zda je jazyk schopný odvodit si jednotlivé typy u nedefinovaných výrazů. Tato vlastnost zapříčiňuje – mimo jiné – statickou typovou kontrolu (Michaelson, 2011).

3.5.2. Srovnání se vzorem

Neboli *pattern-matching*, je technika pro větvení algebraických datových typů, respektive spárování hodnoty s určitým vzorem. Má tři vlastnosti (Harrison, 1997):

- Zachycuje všechny možnosti (je takzvaně *exhaustive*).
- Pouze jeden výsledek pro jeden vzor (*injective*).
- Není možná duplikace (*distinct*).

Za vzorem následuje funkce, která se má vykonat při úspěšném spárování typu. Jako vzor může posloužit i výraz využívající vlastnosti datového typu (například hlavičku v listu). Používáním techniky srovnání se vzorem je možné docílit programování zvaného jako *pattern-matching programming* (Alexander, 2017).

3.5.3. Funktor

Jedná se o návrhový vzor pro speciální druh funkce, která odstraňuje nadbytečnou režii při manipulaci s typy-typů, tedy algebraickými datovými typy.

Ve funktoru se ADT modifikuje prostou funkcí, pokud je možné onen algebraický typ „rozbalit“ (Mugnaini, 2018). Výsledkem je opět algebraický datový typ, buď modifikovaný (ale stále ve stejném typu), nebo původní.

Příkladem je aplikace funkce *f* do algebraického datového typu *T* nesoucí hodnotu typu *X*, v pseudokódu (poslední část oddělená šipkou je výsledek výrazu, zbytek arity) zleva doprava (implementace závisí na konkrétním jazyku):

$$f ::= (X \rightarrow X)$$
$$funktor ::= T\ X \rightarrow f \rightarrow T\ X$$

Obyčejná funkce by nemohla změnit hodnotu, která je zabalena v algebraickém datovém typu. Funktor naproti tomu dokáže nejdříve zjistit podtyp algebraického typu, pak aplikovat změny a následně podtyp zase zabalit (Yallop a White, 2014).

Technika se využívá při nejistém výsledku předchozí funkce, ale i při aplikování funkce, pokud je „obálka“ validní. Místo funktorů (včetně aplikativního funktoru a monád) je možné využít srovnání se vzorem a vlastní algoritmus (což interně dělají i funktory).

Aplikativní funktor

V případě aplikativního funktoru je nutné, aby algebraický datový typ aplikované funkce odpovídal výslednému algebraickému datovému typu (Mugnaini, 2018). Pokud tomu tak je, funkce se aplikuje na hodnotu a znovu zabalí. V pseudokódu:

$$f ::= T\ (X \rightarrow X)$$
$$funktorA ::= T\ X \rightarrow f \rightarrow T\ X$$

Tato technika se využívá například při postupném vícenásobném vyhodnocování parametrů funkce, která nese algebraický typ. Při nerovnosti typu parametru s typem funkce dojde k přerušení vykonávání funkce (respektive je vrácen alternativní typ).

3.5.4. Monáda

Speciální druh aplikativního funktoru (Czaplicki, 2012). Rozbalí algebraický datový typ a aplikuje funkci, zároveň však tato funkce musí sama vracet požadovaný algebraický typ, tzv. nedojde k samozabalení (Mugnaini, 2018). V pseudokódu:

$$f ::= (X \rightarrow T\ X)$$
$$monada ::= T\ X \rightarrow f \rightarrow T\ X$$

Využití nalezne monáda v případech, kdy by se ADT nechtěně zabalil do více vrstev, respektive se chce jeho poslední vrstva.

Díky mechanismům funktorů a monád je možné zavést transparentnost do netransparentních funkcí, nebo alespoň na ni poukázat (Alexander, 2017). Typicky jde o funkce pro uživatelské vstupy a výstupy, API volání a podobně.

Někdy jsou určité ADT (například *Maybe/Optional*) označovány jako monády; to je však mylné. V takových případech se myslí, že pro daný ADT existuje použitelná monáda.

3.5.5. Polymorfismus

Díky zavedení typovosti je možné do jazyka implementovat polymorfismy, které zanáší vztahy mezi funkcemi a lépe strukturují kód.

Parametrický

V parametrickém polymorfismu se funkce anotují přes neznámé parametry, které představují určitý – ne však konkrétní – typ (Harrison, 1997). Tento typ může být v jazyku zástupný (například parametr *number*), nebo nemusí být pro výpočet vůbec důležitý (například parametr *a*, ve funkci pro spojení dvou listů; funkce nepotřebuje znát, co list obsahuje, musí být akorát stejného typu *a*). Schopnost je umožněna díky referenční transparentnosti (viz kapitola 3.6.2), avšak zvolňuje statické typování (Bůna, 2018). Funkce využívající parametrický polymorfizmus se nazývají polymorfické či generické (Loder, 2018).

Typově vyšší

Rozšířením parametrického polymorfismu na jakýkoliv typ vzniká polymorfismus typově vyšší neboli *higher-kinded* (Yallop a White, 2014). V těchto případech nezáleží, jaký typ přichází do funkce (*a, b,…*), respektive může jít o typ zástupný (*number* a jiné). Například datové typy *List* a *Set* mohou mít společnou funkci *append* pro přidávání prvků nějakého typu *a*.

Řádkový

Speciálním případem je polymorfismus řádkový, který je založen na definování pouze části typu, přičemž vychází z libovolného typu (např. *a*); tedy jde o jakousi „povinnou“ položku, jež *a* musí obsahovat. Vzhledem k povinnosti neměnných dat (viz kapitola 3.6.3) je tato vlastnost velmi ceněná při práci s datovou strukturou *Record* (viz kapitola 4.11.2), kdy se změní pouze vybraná část záznamu a zbytek struktury je zkopírován (Bůna, 2018).

3.5.6. Rekurze

Jak již bylo zmíněno, rekurze je jedním z nejdůležitějších prvků funkcionálního programování. Bez rekurze by nebylo možné pracovat například s listy, ani mít program spuštěný v nekonečné smyčce (Harrison, 1997).

Rekurze může být **primitivní** s určitým počtem opakování, nebo **obecná**, kde je počet opakování neznámý (Michaelson, 2011). Aby se rekurze s neznámým počtem opakování nezacyklila, je nutné ji ukončit pomocí podmínky. Podmínka pro ukončení může ovlivnit návratovou hodnotu, či nikoliv (pak se použije funkce *identitní*, například při multiplikaci jde o návratovou hodnotu 1).

Vytvoření rekurze je možné pomocí volání sebe sama (*self-reference*), nebo aplikací přes takzvaný bod rekurze (také jako *paradoxical combinator* nebo *fixed point finder/combinator*). V *lambda výrazu* bod rekurze implementoval Haskell Curry:

$$Y ::= \lambda f.\big(\lambda x.f(x\ x)\big)\ (\lambda x.f(x\ x))$$

Kde je výraz označen jako funkce *Y*. Rekurze kolem funkce *X* je následovná:

$$X \rightarrow (Y \rightarrow X)$$

Při rekurzi dochází k „natáčení“ (*winding*) volání na zásobník (Alexander, 2017). Jakmile se volání završí, tok programu se „odtáčí“ (*unwinding*) zpět k poslednímu volání a pokračuje ve vyhodnocování. Celou dobu si drží lokální kontext, který se postupně s každým dalším voláním inkrementuje. To může způsobit, že při vysokém počtu volání dojde k „přetečení“ zásobníku (*stack overflow*). Řešení nabízí metoda zvaná *tail call optimization* která využívá funkci *tail recursive* (Harrison, 1997). Funkce je založena na sběru návratových hodnot do akumulátoru, takže nedochází k držení celých kontextů všech volání, ale pouze aktuálního (Alexander, 2017). Tento přístup má imperativní prvky, ale výrazně snižuje paměťovou náročnost programu.

Jiné imperativní přístupy, například pomocí smyček *for* a *while*, nejsou povoleny. Jejich částečná implementace je možná přes jmennou rekurzi pomocí takzvaných **redukcí**, kdy se iterativně redukuje list hodnot zleva (*foldl*) či zprava (*foldr*) a postupně se na nich aplikuje funkce do akumulátoru, výsledkem je pak pouze jedna hodnota.

I přes deklarativnost se řada funkcionálních jazyků (mezi nimi i Elm) kompilují do imperativního zápisu cyklů (Wälter, 2019).

3.6. Vlastnosti

Z konceptů předchozích podkapitol lze vyvodit ustálená slovní spojení, jež se (nejen) ve funkcionálním světě používají pro popis konkrétních vlastností.

3.6.1. Funkce první třídy

Také jako *first-class citizen/function*, značí jazyk, který podporuje funkce vyššího řádu (Wälter, 2019). Kromě toho přistupuje k funkcím jako k jiným datovým typům; lze je tak přiřazovat do proměnných.

3.6.2. Referenční transparentnost

Též jako referenční průhlednost, označuje část kódu, která nemá žádné vedlejší efekty (*side-effects*) a vrací stejné výstupy při stejných vstupech (Khot a Mishra, 2017). Pojem je úzce spojen s čistou funkcí. Pokud všechny výrazy v programu mají referenční transparentnost, jsou takové programy deterministické (Alexander, 2017). Důsledkem referenční transparentnosti je urychlení vyhodnocování funkcí s nezměněnými vstupy.

3.6.3. Neměnnost dat

Neměnnost dat neboli *immutabilita* znamená, že nelze přiřadit proměnné novou hodnotu výrazu, pokud byla proměnná již jednou přiřazena (Wälter, 2019). Tento koncept vyplývá ze samotné definice *lambda kalkulu* s rekurzí. Bez neměnnosti dat by nebylo možné provést zjednodušení kódu *α-substitucí* (viz kapitola 3.1.2 o *lambda výrazu*). Neměnnost dat je pravidlem ve funkcionálních jazycích, v imperativních je možné zavést immutabilitu tvořením konstant (*const*, *final*, …).

Neměnnost dat odstiňuje programátora od myšlenek, zda se mu v jiné části kódu nezměnila na pozadí hodnota proměnné (Alexander, 2017). Při úpravě dat je nutné vzít původní objekt, zkopírovat ho a nahradit upravené části (Wälter, 2019). Což by se mohlo zdát jako režie navíc (jak softwarová, tak hardwarová), ale většina jazyků implementuje techniky pro šetrnou změnu s určitou supervizí nad pamětí.

Práce s neměnnými daty je *thread-safe*, čili podporuje bezpečný přístup paralelního zpracování a zabraňuje souběhu (Alexander, 2017). Na druhou stranu některé funkce jsou měnné – pozice myši, stisknutá klávesa… Je tak nutné měnná data přenést do neměnných dat; jedním z řešení je zabalení do monád zpracovávajících příchozí signály.

3.7. Dopady

Využití všech výše zmíněných technik zavádí do programování jistý řád, který zajišťuje dle Alexandra (2017) následující pozitivní vlastnosti:

- *Lepší srozumitelnost kódu* – kód nemá skrytá vedlejší volání, je psán na vyšší úrovni a díky parametrizaci je okamžitě uchopitelný (např. narozdíl od *void*).
- *Jednodušší testování a debuggování* – funkce mají jasné výstupy.
- *Bezpečnější programy* – díky zavedení immutability a typovosti.
- *Podpora determiničnosti* – za použití prostých funkcí.
- *Uřiditelný paralelismus* – zabraňuje souběhům bezestavovostí.

Na druhou stranu vyvstávají i negativa:

- *Prosté funkce se hůře skládají do sebe* – míra abstrakce je vyšší než při imperativním programování.
- *Uživatelské vstupy a výstupy nejsou prosté* – mohou vracet jiná data při stejném vyhodnocení.
- *Vnořená data se špatně mění* – díky immutabilitě je nutné každý podprvek znovu vytvořit.
- *Potenciálně způsobují výkonnostní problémy* – špatnou metodou hodnocení či implementací immutability apod.

4. Programovací jazyk Elm

Elm je zástupcem **deklarativních** jazyků, konkrétně funkcionálního a reaktivního paradigma (ne však FRP, viz kapitola 2.6). V produkčním prostředí je bez interpretu, tedy pouze **kompilovaný**. V kombinaci se **silnou typovou orientací** a **typovou inferencí** slibuje běh programu bez chyb (tzv. *no runtime errors*). Využívá široké palety vlastností funkcionálních jazyků založených na *lambda kalkulu* (viz kapitola 3.1), například funkce první třídy, referenční transparentnost, parametrický polymorfismus, algebraické datové typy a další (viz kapitola 3.5). Vyhodnocovaní výrazů je **striktní**, nestriktnost lze programátorsky doimplementovat.

Elm je také zkratkou pro pojmy *Extreme Learning Machine* ze strojového učení a *Edge Localised Modes* z jaderné fyziky.

4.1. Historie

V roce 2012 Evan Czaplicki představil Elm jako jazyk implementující FRP (konkrétněji AFRP, viz kapitola 2.6.1) pro práci s GUI. Jeho myšlenka byla obecná a implementace do webového prostředí jen jednou z možných. Czaplicki deklaroval dvě hlavní vlastnosti: čistý funkcionální přístup ke grafickému rozhraní a konkurenčnost. Tyto vlastnosti měly napomoci řešit „zamrzání", jež provází interaktivní systémy při přepočítávání hodnot vzniklých změnou vstupu. Díky používání čistých funkcí (a tedy žádných vedlejších efektů) je zaručeno, že u funkcí s nezměněnými vstupy není nutné přepočítat nový výstup. K tomu pomáhá technika konkurenčního zpracování, jejíž implementace využívá *memoizování* vstupů a výstupů u funkcí spojených s událostmi v čase (Czaplicki, 2012).

Autor jazyka má nad vývojem velkou kontrolu a postupně se věnoval odstraňování složitějších konstrukcí, převzatých jak z konceptu FRP, tak Haskellu, jenž byl při vzniku Elmu inspirací (Loder, 2018). Týká se to datových struktur či operátorů. Od verze 0.17 se Elm odprošťuje od FRP odebráním signálů, jež byly nahrazeny jinými mechanismy, především subskripcemi (Czaplicki, 2016) a příkazy. Velká pozornost se věnuje zlepšování výkonnosti (běh, kompilace), rozšiřování knihoven (například o rozhraní WebGL) či vylepšování chybových hlášek. Oproti živelným změnám prvních čtyř let byly za poslední tři roky vydány pouze dvě nové verze (0.19 a 0.19.1) – a to s kosmetickými úpravami.

4.2. Komunita

Kromě oficiálního fóra (https://discourse.elm-lang.org) se vývojáři sdružují na Redditu (https://www.reddit.com/r/elm/) a především Slacku (https://elmlang.herokuapp.com).

K únoru 2020 jde o 9 tisíc přispěvatelů na Redditu a téměř 19 tisíc na Slacku, kteří si navzájem radí v kanálech pro začátečníky i pokročilé.

4.3. Použití

Nejčastějším použitím Elmu je webová aplikace typu SPA (viz kapitola 6.1). Příkladem může být task management provázaný s informačním systémem v české společnosti ABRA Software, nebo správa akciového portfolia z webové i mobilní aplikace ve švédském SAVR. Aplikace nemusí ovládat celý webový prostor, může být pouze začleněna do části HTML, více v kapitole 4.15.6.

Kromě frontendu je možné Elm experimentálně využít na backendu v rámci Node.js či speciálně upraveném frameworku Elmstatic, který generuje statické HTML a CSS. Zdrojový kód je možné úplně odstínit od webové implementace a po kompilaci ho využívat jako univerzální JS skript. Mimo to lze s Elmem interaktivně pracovat v terminálovém režimu REPL (read-eval-print-loop).

4.4. Distribuce

Oficiální distribuce probíhá přes elm-lang.org, na kterém lze nalézt rozšiřující balíčky (https://package.elm-lang.org, 2020) s odpovídající dokumentací a zdrojovými kódy. Popis projektu a závislostí se definují ve formátu JSON (JavaScript Object Notation) pojmenovaném *elm.json*, podobně jako v JavaScriptovém distribučním systému NPM. Nad balíčkovacím systémem mají přímou kontrolu správci Elmu a publikování podléhá kontrole, což napomáhá udržení myšlenkám a „čistotě".

Na druhou stranu vyvstává problém s možností vlastních balíčků, udržovaných soukromě. Pro takové případy je možné rozdělit projekt na GIT podrepozitáře a následně je spravovat přes neoficiální nástroj *elm-git-install*. Nicméně takto spravované balíčky přichází o podporu elm.json. Zdrojové kódy lze také získat přímo ze stránek GitHubu (https://github.com/elm, 2020).

4.5. Instalace

Elm je v širším slova smyslu celá platforma (Imsirovic, 2018), která je šířena jako instalační soubor vytvářející spustitelný příkaz *elm*. Platforma obsahuje několik nástrojů pro práci se zdrojovými kódy, dříve spustitelné pod vlastními příkazy (např. *elm-test*, *elm-package*); nyní jsou začleněny do výše zmíněného příkazu pod argumenty (Feldman, 2020):

- *init* – vytvoří projekt s elm.json
- *make* – kompiluje projekt či soubor

- *repl* – spustí interaktivní konzoly
- *install* – z oficiální distribuce stáhne rozšiřující balíček a přída ho do elm.json
- *publish* – publikuje balíček do package.elm-lang.org

4.6. Kompilace

Elm je jazyk, jenž se oficiálně kompiluje (prozatím) pouze do JavaScriptu. Vzhledem k silné typovosti (viz kapitola 4.11) obsahuje Elm mocný kompilátor napsaný v Haskellu (Fairbank, 2019), který dokáže mít až třínásobnou úsporu vygenerovaného kódu oproti konkurenčním frameworkům React a Angular (Czaplicki, 2018). Úspora je možná díky eliminaci nadbytečného kódu, který je jinak zanášen při importu souborů; v Elmu jsou do kompilace přidány pouze použité funkce, nikoliv celý modul (viz kapitola 4.7). Kompilátor má několik přepínačů, které se přidávají k argumentu *make*, jedná se o:

- *debug* – zpřístupní ladící nástroje
- *optimize* – vytvoří produkční verzi v co nejmenší velikosti
- *output* – definuje umístění a způsob výstupu (HTML s JS, nebo pouze JS)
- *docs* – generuje JSON dokumentaci z anotací

Kromě toho je kompilátor díky stromové struktuře velmi rychlý, ve vývojářském módu překládá pouze části, které se změnily (Czaplicki, 2018). Do zkompilovaného kódu je přeci jen možné zavléci nadbytečnosti, a to přiložením JavaScriptu, jelikož ten nepodléhá typové kontrole. To se však nemůže stát při používání oficiálních balíčků z distribuce, které jsou napsané pouze v Elmu (a tedy otypované).

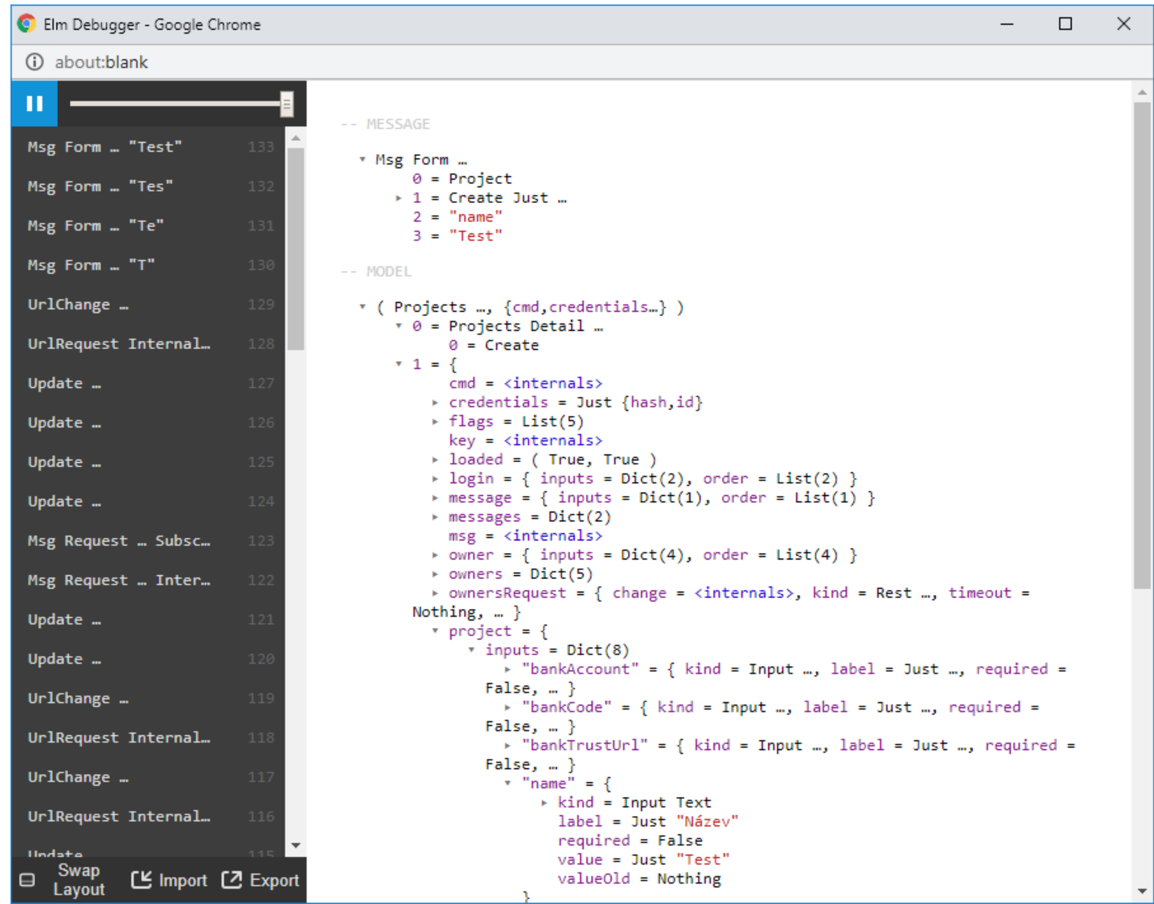
Chybové hlášky

Elm se chlubí bezchybným produkčním během (Loder, 2018). Není to tak úplně pravda. Vyskytují se chyby, které jsou spojeny buď se špatnou implementací některých standardů, nebo specifik prohlížečů. Například funkce *link* s parametrem typu řetězec provádí JavaScriptová volání *pushUrl* i přesto, že jde o odkaz z jiné domény – v takové situaci se zobrazí hláška ve vývojářské konzoli a prohlížeč neprovede přesměrování. V Safari je při stahování bytů nutné počkat, než je možné objekt *Blob* zlikvidovat pomocí *revokeObject*; Elm provádí destrukci okamžitě, čímž uživateli objekt „zmizí“ pod rukou.

Jinak ale platí, že implementace je velmi dobře odstíněna od JavaScriptu a funguje efektivně ve většině prohlížečů, jak by měla. Pokud se objeví chyby, tak vychází z programátorovy mysli. V syntaktických šotcích pomohou podrobné kompilační hlášky, které doznaly výrazných změn v poslední verzi 0.19.1 (Czaplicki, 2019).

Debugging

Ladění aplikace plně využívá možností, které poskytuje *lamda kalkul*. Díky prostým funkcím je možné relativně přesně zjistit stav, ve kterém se aplikace nachází. Tento stav se zobrazuje v debuggovacím nástroji umístěném přímo v prohlížeči vpravo dole pod modrou ikonou Elmu. Nachází se v něm výpis událostí (zpráv neboli *messages*), které aplikací prochází; v každém okamžiku je tak možné stav zrekonstruovat a podívat se na ovlivněné hodnoty. Dalšího rozšíření se dočkal nástroj ve verzi 0.18, kdy přibyla možnost importovat a exportovat kompletní historii (Czaplicki, 2016) – je tak možné resuscitovat sezení cizího stroje a zjistit pohyb napříč aplikaci.



Obrázek 1 - Debugger ukazuje vlevo zprávy, vpravo model. Zdroj: Vlastní.

Kromě tohoto nástroje je možné vysílat obligátní zprávu na vývojářskou konzoli přes funkci *log* z modulu *Debug*, která vyšle příkaz do JavaScriptu k zavolání funkce *console.log* zobrazující typ *a*. Obě výše zmíněné metody ladění je možné spustit pouze v režim *debug*.

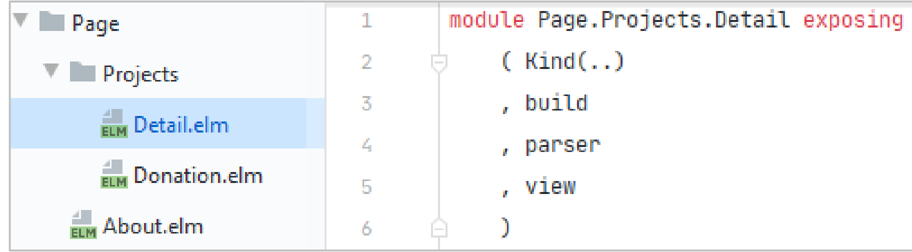
Build

Rozšířit možnosti při kompilaci je možné přes zabalení do buildovacích nástrojů typu Webpack či Parcel, využívající další funkce, jako je minifikace a timestampy assetů.

4.7. Modularita

Skládání výsledného programu se v Elmu provádí pomocí modulů. Název modulu je tvořen velkými počátečními písmeny a tečkovou notací, jež odpovídá umístění ve složce vzhledem ke kořenu projektu – slouží tak i k vytvoření *namespace* či struktuře nadmodulů a podmodulů, podobně jako v jiných jazycích (Mladý, 2017).

Všechny vytvořené funkce (včetně typů) jsou nastaveny k privátnímu použití uvnitř modulu, jejich publikování okolnímu světu je možné přes zařazení do výpisu po *exposing*. Pomocí dvou teček se dá zkráceně zpřístupnit vše, co je v modulu (či *Sum type*) přítomné.



Obrázek 2 - Stromové rozdělení modulu a publikování funkcí. Zdroj: Vlastní.

Moduly mohou fungovat samostatně či se do sebe vnořovat pomocí *import*. Není možné vytvářet cyklické importy, jejich použití je stromové. Takové omezení má jak výkonnostní, tak designové důvody. Vesměs nutí programátora vytvářet komponentní strukturu, která se dle autora ideálně rozdělí na **nezávislé** (většinou obecné komponenty – práce se session, entity a různé utility časové, textové aj.) a **závislé** komponenty (většinou vizuální komponenty – stránka včleňuje hlavičku, formuláře, tabulku aj.).

4.8. Prostředí

Speciální vývojový nástroj pro Elm není, nicméně po instalaci pluginů zvýrazňující syntax lze využívat editory jako Atom, Visual Studio Code, Sublime nebo WebStorm. Nejlépe se autorovi práce jeví WebStorm, který urazil dlouho cestu, avšak některé funkce (například debuggování přímo v **.elm* souborech) mu chybí. Vhodným doplněním je formátovací nástroj třetí strany nazvaný *elm-format*, který lze zakomponovat do výše zmíněných editorů. Pro rychlou úpravu či test je možné využít online editor na adrese https://elm-lang.org/try (2020).

4.9. Struktura

Soubor obsahuje na prvním řádku název modulu (včetně struktury oddělené tečkou) s exportovanými funkcemi a typy. Na dalších řádcích se nachází importované moduly, následují definice funkcí a typů. Všechny prvky jsou odděleny prázdnými řádky.

```
module Page.NotFound exposing (view)

import Component.Session.Model exposing (PageResponse)
import Html exposing (Html)


title : String
title =
  "404"
```

Ukázka 1 - Struktura modulu. Zdroj: Vlastní.

4.10. Syntaxe

Elm je syntakticky chudší než většina běžných imperativních jazyků. Opačně by se však dalo říci, že je v kontrastu s deklarativními jazyky bohatší, především oproti logickým a funkcionálním. Je tak na jistém pomezí, což může usnadnit pozdější přesun k ještě puritánštějším jazykům, jako je například Haskell (Loder, 2018). Jazyk lpí především na **odsazeních** (blok kódu) a **mezerách** (příkazy a operátory), čímž omezuje používání obyčejných závorek jako oddělovačů (lze je však též použít). Složené závorky využívá pro datovou strukturu typu záznam (viz kapitola 4.11.2). V kódu se rozlišuje mezi **velkými** (typ) a **malými** písmeny (funkce a proměnné).

4.10.1. Anotace

Funkce a proměnné jsou v obecném vnímání imperativních programátorů rozdílné, ale v *lambda kalkulu* mezi nimi rozdíl fakticky není (viz kapitola 3.1.3). Podobně se k problematice staví funkcionální paradigma, potažmo Elm, kde **proměnná** je definována jako bezaritní funkce[3] (viz *title* v ukázce 1), **konstanta** je proměnná na první úrovni modulu.

Anotace funkcí je díky *typové inferenci* nepovinná. Doporučují se však anotovat funkce na první úrovni modulu, případně funkce složitější na dalších úrovních (Loder, 2018). Anotace se provádí o řádek výše, než je samotná deklarace. Následuje dvojtečka a **seznam typů** oddělených šipkou, které funkce „přijímá" (viz *currying* v kapitole 3.3.3) s tím, že poslední je typ, který funkce „vrací" (viz ukázka 2). Imperativní označení těchto typů jako *parametrů* a *návratové hodnoty* je nepřesné, avšak pro běžnou komunikaci dostatečné.

```
ellipsis : Int -> String -> String
ellipsis max s =
  if 3 + String.length s > max then
    String.slice 0 max s ++ "..."
  else
    s
```

Ukázka 2 - Anotace funkce s dvěma aritami. Zdroj: Vlastní.

[3] Přirozeně, kompilace vytvoří v JavaScriptu proměnnou, nikoliv funkci.

4.10.2. Definice

Na dalším řádku následuje definice funkce, ve které se nejprve opisuje název funkce a deklarují názvy proměnných, které budou ve funkci přístupné. Je možné, aby zde nebyly vypsány všechny „parametry", které se nachází v anotaci, lze je totiž včlenit do funkce *ex post* pomocí *kompozice funkcí* (viz kapitoly 3.3.2 a 4.10.3). Pro zástupný parametr lze použít univerzální znak **podtržítka** (viz kompozice funkcí v ukázce 4), jež nedovoluje pozdější použití v kódu; jednoduše se tato hodnota ignoruje.

Po té zápisem rovnítka končí deklarace a na dalším řádku následuje tělo funkce. Opět ve funkci nemusí platit, že explicitně vrací pouze poslední anotovaný typ; může to být identita, částečná aplikace či jiná funkce. Jedinou podmínkou je plné uspokojení anotace při vyhodnocování. Termínem „vrací" je myšlen výsledek funkce, který vyplývá z toku program ovlivněného řídícími strukturami (viz kapitola 4.13). Není možné, aby byl výsledek jednoho větvení odlišný od druhého.

Alternativní cestou pro definici funkce je její **anonymní** varianta (také jako bezejmenná či lambda), která postrádá anotaci. Takzvaná anonymní funkce se definuje pomocí zpětného lomítka, výčtem parametrů a šipkou, za kterou následuje výraz funkce (viz ukázka 3). Přednost v použití dostává anonymní funkce před jmennou při iteraci strukturovaných typů (tzv. *mapping*).

```
-- Pojmenovaná funkce
add: Int -> Int
add x =
   x + 1

-- Anonymní funkce
\x -> x + 1
```

Ukázka 3 - Rozdíl mezi pojmenovanou a anonymní funkcí. Zdroj: Vlastní.

4.10.3. Operátory

K vymezení aplikace funkcí a ovlivnění směru toku programu se užívají operátory. Některé se používají **infixově**, jiné **prefixově**. Infixový způsob značí použití mezi dvěma argumenty, prefixový (se závorkami okolo operátoru) před nimi (čímž napodobuje aplikaci funkce).

(a)

Obyčejné (jednoduché) závorky jsou speciálním případem. Slouží, podobně jako v *lambda kalkulu*, k obalení výrazu. To znamená, že určují hranice působnosti a pořadí, ve kterém se budou výrazy vyhodnocovat, od vnitřního k vnějšímu (viz kapitola 4.14). Časté použití je s anonymní funkcí, kterou obalením oddělí (vytvoří výraz) od zbytku kódu. Kromě výše

zmíněného slouží pro datovou strukturu tuple, kterou si však lze představit jako funkce oddělené čárkou určené k vyhodnocení. Některé infixové operátory mají variantu, ve které mohou být při obalení jednoduchými závorkami použity prefixově či jako jmenné funkce.

<| a |>

Nazývané jako *funkcionální operátory* nebo též *pipe operators*. Jde o **částečnou aplikaci funkcí** (viz kapitola 3.3.1) v infixovém (též prefixovém a jmenném) tvaru. Pořadí aplikace (zleva doprava, zprava doleva) určuje (v infixu) zobáček směřující do funkce, která bude provádět aplikaci (viz ukázka 4). Hlavním použitím je zástupnost za jednoduché závorky, tedy určení pořadí pro vyhodnocení výrazů. „Sémanticky" tak *pajpy* nepřinášejí žádné vylepšení a je možné se bez nich obejít. Jejich použití však nabírá na významu v *řetězení funkcí* (neboli *chainování*), které znatelně zlepšuje čitelnost kódu. Ze zvyklosti se řetězí od největšího po nejmenší, a to na jednom řádku nebo víceřádkově.

<< a >>

Druhým infixovým operátorem jsou dvojité špičaté závorky neboli **kompozice funkcí** (viz kapitola 3.3.2). Kompozice je též zbytná, avšak přináší urychlení a úsporu místa při vkládání předchozího výsledku do další funkce, kde by se musela proměnná deklarovat, či by musela být vytvořena anonymní funkce. Směr je určen, podobně jako u částečné aplikace, šipkami od vnitřní funkce do vnější (viz ukázka 4).

```
hello : String -> String
hello s =
   -- nějaké tělo funkce vracející řetězec, například (s), (s ++ “, jak se máte?”), (always “Přepis”` s) aj.

-- Uzávorkování výrazu
hello ("Vítejte, " ++ " v aplikaci")

-- Prefixový tvar
hello ((++) "Vítejte, " " v aplikaci")

-- Jmenný tvar
hello (append "Vítejte, " " v aplikaci")

-- Částečná aplikace (v řádku)
hello <| "Vítejte, " ++ " v aplikaci"

-- Kompozice funkcí (simulace anonymní třídou)
hello << (++) "Vítejte" << (\_ -> " v aplikaci")
```

Ukázka 4 - Skládání funkcí. Zdroj: Vlastní.

Aritmetické, porovnávací a logické

Plejáda operátorů je vcelku klasická a podobá se nabídce ostatních jazyků. Jde o infixní varianty, většinou se jmennou i prefixní alternativou. Mezi základní operátory patří:

- (+, -, /, //, %, ^) – aritmetické, akceptují pouze tzv. *number* (viz kapitola 4.11.1)
- (==, /=, <, >, <=, >=) – porovnávací, akceptují pouze tzv. *comparable*
- (&&, ||) – logické, nejsou jmenné, akceptují pouze hodnoty *Bool*

Z výše uvedeného je vidět, že použití operátorů je typově striktní, což musí být dodrženo i v komunikaci s okolním světem (viz kapitola 4.15.6). Pro převody existují základní funkce volatelné odkudkoliv, například *toFloat*. Každý modul pak může nabízet další převody, například modul *String* obsahuje funkci *fromInt* (viz ukázka 6).

Speciální a vlastní

Speciálními případy jsou operátory pro spojování (++) typů *appendables* (viz ukázka 4 a ukázka 5) a přidávání nového prvku na první pozici (*::*), viz ukázka 4. Mezi méně využívané patří bitové, ve kterých se nalézá standardní nabídka, avšak pouze ve jmenné formě (*and*, *or*, *xor*, *shift…*). Speciálními případy jsou (|=, |.) v modulu *Parser*, které převádí řetězec na typ pomocí parsování. Výjimkou jsou též operátory (</>, <?>) v *Url.Parser*, jež dekódují URL řetězec na typ. Dříve bylo možné vytvářet vlastní operátory, od verze 0.19 již není kvůli jednoznačnosti povoleno (Czaplicki, 2018).

```
-- Vytvoření listu spojováním O(n)
(["To"] ++ ["je"] ++ ["list"])

-- Vytvoření list přidáváním hlaviček O(1)
("To"::("je"::["list"]))

-- Parsování adresy ve tvaru [0-9]+/edit
Url.Parser.int </> Url.Parser.s "edit"
```

Ukázka 5 - Operátory. Zdroj: Vlastní.

4.11. Typy

Jádro Elmu tvoří typy, jež jsou **statické** (neměnné v běhu kódu). Díky nim je možná takzvaná **typová inference** (viz ukázka 6), která dokáže odvodit, jaký typ je proměnná, aniž by byla předtím anotována. To je výhodné při našeptávání editoru a také v kompilaci – mechanismy přesně ví, jaká funkce či typ se kde může a nemůže objevit.

Ačkoliv se typy dají parafrázovat jako datové typy či v objektovém světě jako třídy[4], ve skutečnosti jde plošně o **algebraické datové typy**[5] (ADT, viz kapitola 3.5.1). K jejich manipulaci se využívá funkcionálních technik, typicky *srovnání se vzorem* (viz kapitola 4.13.2) či použití speciálně navržených funkcí, tzv. *funktorů* (viz kapitola 3.5.3 a 464.13.3), jež urychlují práci s ADT (interně však jde opět o srovnání se vzorem).

V Elmu je též implementován **parametrický** a **řádkový polymorfismus** (viz ukázka 7), nikoliv však vyšší (viz kapitola 3.5.5). V parametrickém polymorfismu je možné vytvářet funkce, které nespecifikují konkrétní typy, avšak zástupné. Řádkový polymorfismus zrychluje práci s podmnožinou ADT, konkrétně kartézským součinem typů (*product type* neboli struktura *Record*, viz dále kapitola 4.11.2).

```
-- Funkce bez anotace
x = "retezec"

-- Anotace je zbytečná, inference zjistí typ z funkce x
y : String
y = x

-- V pořádku
y ++ "404"

-- Chyba v kompilaci
y ++ 404

-- V pořádku
y ++ String.fromInt 404
```

Ukázka 6 - Typová inference. Zdroj: Vlastní.

Je dobré si uvědomit, že vše, co se objevuje jako proměnná či datová struktura, je zároveň typem. Což platí i o funkci, jelikož vyhodnocením nabývá nějakého typu. Nakonec i samotná definice typu může být funkcí jako **datový** či **typový konstruktor** (Luxemburk, 2017). Taková funkce pak přijímá parametry v pořadí, v jakém byl typ definován.

4.11.1. Primitivní

Přes výše uvedené se dle zvyklosti i v Elmu rozlišují datové typy podle složitosti. Jako obyčejné se označují takzvané primitivní datové typy – *Int*, *Float*, *Bool*, *String* a *Char*, jež přiřazují **literály**. Některé mají zástupný parametr, například *Float* a *Int* používají v anotacích proměnnou *number* či *comparable*[6]. Převod mezi typy se provádí buď

[4] Či generika. Záleží, zda jde o *product type* či *sum type*.

[5] Minimálně jde o *Opaque Types*. Pokud není explicitně uvedeno, tak se v práci označuje algebraický datový typ pouze zástupně jako typ.

[6] Takzvané omezené typové proměnné (*Constrained Type Variables*). Jde o speciální případy, které přidávají jazyku na flexibilitě.

globálními funkcemi (např. *toFloat* pro převod celočíselné hodnoty), nebo funkcí z daného modulu (např. *String* se vytvoří z funkce *fromFloat* či *fromInt*).

4.11.2. Strukturované

Strukturované datové typy jsou parafrází na **kolekce**. Jde tedy o seznam jiných typů, které strukturovaný typ udržuje. Všudypřítomným základem je *List*, *Record* a *Tuple*. Importovat za standardní knihovny je možné *Set*, *Array* a *Dict*. Počet strukturovaných typů je v jazyce pevně dán, přičemž každý typ má **unikátní vlastnosti** a jinou **asymptotickou složitost** přístupu k prvku (viz tabulka 1).

Název	Implementace	Konstrukce	Různé typy	Složité typy	Duplicity	Řazení	Přístup
List	Spojový seznam	[] singleton range	Ne	Ano	Ano	Ano	$O(n)$
Array	Relaxovaný vyhle. strom	fromList initialize	Ne	Ano	Ano	Ano	$O(\log n)$
Set	Relaxovaný vyhle. strom	singleton fromList	Ne	Ne	Ne	Auto	$O(\log n)$
Tuple	JS Object (Hash)	() pair	Ano	Ano	Ano	Ano	$O(1)$
Record	JS Object (Hash)	{ } pair	Ano	Ano	Ne (klíč) Ano (h.)	Auto (klíč)	$O(1)$
Dict	Relaxovaný vyhle. strom	fromList	Ne	Ne (kl.) Ano (hodn.)	Ne (klíč) Ano (h.)	Auto (klíč)	$O(\log n)$

Tabulka 1 - Charakteristiky strukturovaných datových typů. Zdroj: Vlastní.

Tvorba struktury je možná vždy přes **funkce**, někdy přes **speciální znaky**. Většina strukturovaných typů implementuje standardní funkce *isEmpty*, *member*, *size*/*length*. Zároveň mohou být k dispozici funkce nabízející iteraci nad prvky, v různých variacích obsahují slovo *map* (viz ukázka 7). Další iterační funkcí je *filter*, jež akceptuje funkci s výsledkem *Bool*, při negativní hodnotě pak automaticky kolekci redukuje. Díky immutabilitě je navrácená kolekce nezávislá na původní a pro efekt je nutné dále pracovat s novou, nikoliv původní kolekcí. Redukci na jednu hodnotu je možné iterativně provést pomocí funkcí *foldl* a *foldr*.

Výhodnost konkrétní struktury tkví v její implementaci. ***List*** je vhodný například pro iterování homogenních prvků s případnou aplikací či filtrací (řádky v tabulce aj.), avšak kvůli absenci indexace[7] nevhodný pro uchovávání číselníků. Též je nutné brát ohled na přidávání prvků, protože interně jde o spojový seznam – tedy při používání spojovat raději velký seznam k menšímu či osamocený prvek pomocí operátoru (*::*) na začátek, nikoliv konec (++), seznamu (viz ukázka 5). Operátor v infixovém tvaru je využitelný při porovnání se vzorem, kdy odděluje hlavičku (*head*) od zbytku těla (*tail*).

Základní indexaci homogenních prvků poskytuje ***Array*** (pole). Struktura je vhodná pro řazené seznamy s celočíselným ukazatelem na aktuální položku (dropdown aj.). Pole je implementováno ve speciálním režimu (relaxované vyhledávání) a zároveň slouží jako interní mechanismus pro další struktury, jako *Set* a *Dict*. Pro unikátní seznam primitivních položek (checkboxy aj.) je vhodný ***Set***, který dokáže provádět množinové operace (*join*, *diff*, *intersection*) na dvou různých datových setech.

```
-- Pozn.: za středníkem následuje výsledek výrazu

-- Inicializace listu; [False, True]
list : List Bool
list = [False, True]

-- Filtrace anonymní třídou; [True]
List.filter (\x -> x /= False) list

-- Mapování funkcí not: Bool -> Bool; [True, False]
List.map not list

-- Převod na pole; 0 - False, 1 - True
Array.fromList list

-- Inicializace záznamu
r : { i_0 : Bool, i_1 : Bool }
r = { i_0 = False, i_1 = True }

-- Extensible record; { i_0 = True, i_1 = True }
{ r | i_0 = True}

-- Inicializace tuplu; (Bool, Bool)
(False, True)

-- Mapování prvku funkcí not: Bool -> Bool; [True, True]
Tuple.mapFirst not

-- Slovník se chová jako záznam, kde je atribut klíč; Dict Int Bool
Dict.fromList <| List.indexedMap (\i k -> (i, k)) list
```

Ukázka 7 - Strukturované datové typy. Zdroj: Vlastní.

[7] Indexaci lze použít alespoň při procházení ve funkci indexedMap.

Tuple je vhodný pro držení dvou, maximálně tří, heterogenních typů. Přístup k položkám je rychlý, realizován přes číselné indexy reprezentované funkcemi *first* a *second*.

Pro více položek se doporučuje ***Record*** (záznam), jenž využívá jmennou indexaci. *Tuple* a *Record* jsou jediní zástupci heterogenních kolekcí[8] a jejich použití se hodí v modelu aplikace či jako funkcionální reprezentanti tříd z objektového světa. Zároveň lze na tuple narazit jako na strukturu, která je výsledkem základní funkce *update* (viz kapitola 4.15.2). *Extensible record* je název pro parametrický zápis záznamu (též *řádkový polymorfismus*), který zanáší minimální potřebnou strukturu (viz ukázka 7). Obě struktury jsou často použitelné s takzvanou *destrukcí hodnot*, která vyplní do předem připravených proměnných odpovídající hodnoty (viz kapitola 4.12).

Pro držení číselníků či modelování vztahů M:N se nejvíce hodí struktura ***Dict*** (slovník), kde je jádrem primitivní klíč a k němu určená (i složitá) hodnota. V takových případech hodnota zastupuje konkrétní záznam entity a klíč představuje identifikátor ve formě řetězce či celého čísla. Též lze o slovníku mluvit jako o struktuře *HashMap*.

4.11.3. Vlastní

Nedílnou součástí vývoje v Elmu je vytváření vlastních typů (*custom types*). Bez vlastních typů by se dalo obejít, ale přehlednost takového kódu by byla problematická. Při vytváření nového typu je nutné rozmyslet (viz ukázka 8), zdali jde o algebraický datový typ *sum*, nebo *product* (viz kapitola 3.5.1).

Sum type (v dokumentaci Elmu též jako *union types*) je výčet typů, kterých zástupný typ **může** nabýt (a alespoň jednoho typu nabyde). Definice začíná slovem *type*, názvem typu a rovnítkem, za kterým následuje disjunktní množina typů (často parametrických) rozdělená znaménkem roury. Typy v množině mohou být parametrické, pak je nutné parametr za názvem typu uvést. *Sum type* nalézá použití při rozlišování druhů, například zpráv, entit či rolí. Je tak zástupcem určité příslušnosti k celku, s kterým lze zacházet jako s generikou.

Product type je výčet typů, které zástupný typ **musí** nabýt. Definice začíná slovem *type alias*, následuje jméno typu, rovnítko a poté struktura záznam, nebo tuple. Zároveň se v případě datového typu záznam vytvoří již výše zmíněný *typový/datový konstruktor*, jenž přijímá argumenty v pořadí jmenných indexů (viz ukázka 8), což je výhodné například při dekódování. *Product type* se dále používá především v okamžicích, kdy je dána pevná struktura, jež se musí naplnit. Svým způsobem tak připomíná třídu s atributy.

[8] Jde o *ADT sum type*, viz kapitola 3.5.123.

```
-- Sum type (dobré řešení), k = Kind.Success
type Kind
   = Error
   | Success

-- Product type, záznam, k = { error = False, success = True } či typový konstruktor (Kind False True)
type alias Kind =
      { error : Bool
      , success : Bool
      }

-- Product type, tuple, k = (False, True)
type alias Kind =
   (Bool, Bool)

-- Sum type parametrický, k = Kind.Success “Povedlo se“
type Kind
   = Error Int
   | Success String

-- Opaque type, e = Error 500
type Error =
   Error Int
```

Ukázka 8 - Vlastní algebraické datové typy. Zdroj: Vlastní.

Zkombinování technik *sum type* a *product type* přináší do funkcionálního světa jisté vztahy mezi typy, které se podobají vlastnostem z objektového světa.

4.11.1. Nepojmenované

Typy mohou být v určitých chvílích nepojmenované. Z faktického hlediska však nejde o anonymní typy, protože je lze typovou inferencí vždy určit z jiného typu dle kontextu. Většinou jde o typy objevující se volně v toku program jako pomocná struktura (tuple, záznam, list) či jako proměnná v destrukcích a řádkovém polymorfismu, kde je v zájmu pouze část struktury. Pomocí *type alias* lze vytvořit jméno typu, které pak lze zástupně používat v anotacích. Typový/datový konstruktor se vytváří, pouze pokud následuje typ *Rec*.

4.11.2. Opaque types

Jak již bylo zmíněno v kapitole 4.9, je možné publikovat do okolního světa vybrané typy a funkce, jež se nacházející uvnitř modulu. Co není publikované, je přístupné pouze v modulu. Jak však udělat, aby bylo možné publikovat typ, ale zároveň znemožnit jeho přímou úpravu? K tomu je určen takzvaný *opaque type*, který obaluje typ nesoucí cílová data (viz ukázka 8). Jedná se spíše o techniku zapouzdření než další druh typovosti.

Fakticky jde o speciální případ *sum type*, kde je k dispozici pouze jedna možnost. Zároveň platí, že se při tomto zabalení vždy vytvoří *typový/datový konstruktor* a typ je tak

volatelný jako obyčejná funkce s parametry. Kromě zapouzdření má vytvořený opaque type výhodu v tom, že nemůže docházet k sémantickým chybám při stejných typech.

Rozbalení se provádí přes srovnání se vzorem, nebo destrukcí argumentu. Což přináší zbytečnou režii při přímém použití mimo modul, avšak takové použití znamená vědomé porušení ochrany.

4.12. Práce s proměnnými

Lambda kalkul, z kterého Elm vychází, klasické imperativní proměnné nezná[9]; vše je funkce. Ke klasickému chování proměnné se dá dopracovat tak, že se funkce pojmenují a jednorázově vyhodnotí. Hodnota se pak pod daným jménem nachází k dispozici po celý svůj životní cyklus. Proměnné jsou **neměnné** (imutabilní), z čehož vyplývají dvě věci:

- Úpravy na proměnné nejsou zpětně reflektovány.
- Proměnnou lze inicializovat pouze jednou.

Běžnou praxí je tedy řetězení jednotlivých operací (*chaining*) na počáteční hodnotě, která se po vyhodnocení všech operací přiřadí k proměnné.

Tok programu v Elmu běží od takzvané inicializační funkce a rozbíhá se pomocí řídících struktur do dalších funkcí včetně jiných modulů. Při tom je důležité mít na mysli, na jaké úrovni kontextu (*scope*) se proměnná právě nachází, protože tato úroveň určuje životní cyklus proměnné (viz ukázka 9).

4.12.1. Globální

Mezi globální proměnné se zařazují proměnné na první úrovni modulu. Tyto proměnné jsou k dispozici napříč modulem po celou dobu a není je možné zpětně upravit. Jde svým způsobem o **konstanty** (viz ukázka 9).

4.12.2. Model

V programu může dokola obíhat jedna proměnná modelu neboli **stav**. Tento stav se inicializuje a poté upravuje v unikátní funkci *update*[10] (viz kapitola 4.15.2). Ke stavu není automatický přístup, je nutné ho do funkcí odesílat a při modifikaci zpětně odebírat pomocí zpráv (viz kapitola 4.15.3). Volbou typu proměnné modelu bývá zpravidla záznam (viz ukázka 18).

[9] O proměnné mluví *Lambda kalkul* v rámci výrazu, což spíše odpovídá pojmenování parametr.
[10] Je to jediná funkce, ve které je možné data trvale změnit.

4.12.3. Lokální

Lokální proměnná je nejčastějším typem použití v programu a má pouze **omezenou oblast** platnosti; po vyhodnocení oblasti přestane existovat. V době platnosti je možné využít jejích hodnot ze zanořených úrovní. Konstrukce se provádí přes klíčové slovo *let*, po kterém následuje oblastně-unikátní název proměnné a přiřazení funkce k vyhodnocení (což může být i literál). Takto je možné definovat více proměnných oddělených prázdnou řádkou, dokud není použito slovo *in*, jež vyznačuje začátek oblasti použití, která též určí návratovou hodnotou. V přiřazení se mohou objevovat nové deklarace a řídící struktury (viz ukázka 9).

Destrukce

Speciálním případem pro práci s výsledkem vyhodnocené funkce je takzvaná **destrukce hodnot** či *argument destructuring* (Fairbank, 2019). Využívá srovnání se vzorem (*pattern matching*) a rozloží strukturu do nových proměnných (Imsirovic, 2018). Časté použití je se strukturou tuple nebo list. Používání desktrukce hodnot odstraňuje část režie a zpřehledňuje kód.

```
listGlobal = [3, 4, 5]          -- Globální proměnná

let                             -- První úroveň
   list = [0, 1, 2]
   (_, t) =                     -- Desktruce hodnot
      let                       -- Druhá úroveň
        list_ = [8, 7, 6]
      in
      case List.reverse list_ of    -- Řídící struktura, všechny možnosti
        h::_ -> (True, h)
        [] -> (False, 0)
in
(list ++ listGlobal ++ [t])        -          -- Spojení a řetězení
|> List.filter (\x -> x > 0)                  -- Filtrace hodnot
|> List.map (String.fromInt << (*) 2)         -- Kompozice funkcí
|> String.join "-"                            -- Spojení listu do jednoho řetězce
-- 2-4-6-8-10-12
```

Ukázka 9 - Práce s proměnnými a řídící struktury. Zdroj: Vlastní.

4.13. Řídící struktury

Též kontrolní struktury. Jde o speciální konstrukty jazyka, pomocí nichž je možné určovat tok programu – takzvaný podmíněný příkaz. Kromě rozhodování na základě logického vyhodnocení (*if-else*) je možné využít ještě srovnání se vzorem, které je denním chlebem ve funkcionálním přístupu (Loder, 2018). Speciálními případy ovlivnění běhu programu jsou funktory.

Ve všech případech musí být návratová hodnota stejného typu a takzvaně vyčerpávající/úplná (*exhaustive*); tedy že pokryje všechny možnosti, které podmíněný příkaz povoluje. Pokud se tak nestane, kompilátor hlásí chybu překladu. Na rozdíl od běžných imperativních jazyků je použití (všech typů) řídících struktur možné využít u přiřazování k proměnným; přeci jen jde pouze o výrazy určené k vyhodnocení.

4.13.1. If … then … else …

Při použití logických podmínek je nutné vzít v potaz, že algebraický datový typ *Bool* nabývá hodnot *True* či *False*. Aby tak splnil podmínku úplnosti, je při každém použití *if* nutné dodat i *else*. V případě nezájmu o obě možnosti se v jedné větvi vrací původní, či prázdná hodnota.

Samotná konstrukce je tedy *if*, následuje výraz vracející *Bool* a po slůvku *then* libovolný výraz k vykonání, jenž vrací stejný typ, který by mohl být vyhodnocen výrazem za *else*. Zkratky či operátory pro *if-else* nejsou povoleny, místo ternárního operátoru lze využít zápis na jednom řádku, jinak se jednotlivé výrazy odsazují do bloku.

4.13.2. Case … of

V případě použití srovnání se vzorem jde o implementaci *pattern matchingu* z *lambda kalkulu* (viz kapitola 3.5.2), tedy kompletní větvení všech možností ADT (viz ukázka 9). Každá možnost musí mít pouze jeden výraz, zástupným znakem pro zbytek je podtržítko. Srovnání se vzorem se využívá jak u složitějších typů, tak primitivních a strukturovaných.

Za klíčovým slovem *case* začíná výraz, jenž vrací ADT k porovnání. Ukončen je slovem *of*, odsazením začínají jednotlivé typy se šipkou, která označuje začátek výrazu k celkovému návratu hodnoty řídící struktury. Úplnost lze naplnit použitím všech typů, částí typů s podtržítkem, nebo jen pouhým podtržítkem. Při vyhodnocování se postupuje od shora dolů.

4.13.3. Funktory

Speciálními případy využití srovnání se vzorem jsou funkce, jenž zrychlují práci s ADT. Nejde tedy tak ani o řídící struktury, jako implementaci funktorů a monád z *lambda kalkulu* (viz kapitoly 3.5.1 a 3.5.4).

Jelikož v Elmu není zaveden typově vyšší polymorfismus, musí se každý ADT řešit individuálně. V systémové nabídce jsou již připraveny ADT *Maybe* (viz ukázka 10) a *Result*, které nabízí funkce s konvenčními názvy *map*, *andMap* (v *Maybe.Extra*), *andThen*.

```
jedna: Maybe Int
jedna = Just 1

nic: Maybe Int
nic = Nothing

-- Funktor
jedna |> Maybe.map ((+) 1) -- Just 2
nic |> Maybe.map ((+) 1) -- Nothing

 -- Aplikativní funktor
jedna |> Maybe.map2 ((+) (Just 1)) -- Just 2
jedna |> Maybe.map2 ((+) (Nothing))  -- Nothing

 -- Monáda
jedna |> Maybe.andThen ((+) 1 >> Just) -- Just 2
jedna |> Maybe.andThen ((+) 1 >> \_ -> Nothing) -- Nothing

 -- Rozbaleni
jedna |> Maybe.withDefault 0 -- 1
nic |> Maybe.withDefault 0 -- 0
```

Ukázka 10 - Funktory a monády. Zdroj: Vlastní.

Tyto funkce nalézají využití především při řetězení nad konkrétním ADT, kdy se provádí určité operace, dokud platí daný typ, jinak se na konci provede alternativní operace. Ovládnutí funktorů je jedním z klíčových prvků pro efektivní a přehlednou práci s ADT, potažmo celým Elmem.

4.14. Vyhodnocení

Vyhodnocení funkcí probíhá **striktně** v **aplikativním** pořadí. Nestriktní vyhodnocení lze provést pomocí techniky *call by value* (též líné), tedy vložením argumentu, například prázdného tuplu.

Při hodnocení se používají běžné funkcionální techniky jako curryování, částečná aplikace a kompozice funkcí. Pro efektivní řetězení (*chaining*) – od obecného ke specifickému, ze shora dolů – se na začátek funkce doporučuje anotovat neměnné složky, jako překlady, atributy a jiné. Na předposlední a poslední místo anotace je vhodné umístit zřetězující prvek, který se do funkce dostane částečnou aplikací (či kompozicí) z předchozí části řetězu s tím, že bude též navrácen jako výsledek.

4.15. Architektura

Funkcionální a reaktivní paradigma silně ovlivňují architekturu Elmu, která je připodobňována k Reactu s Reduxem (Imsirovic, 2018). Obecně je architektura založená na globálně drženém imutabilním modelu, jenž se po přijatých signálech komparuje se striktně vyhodnocenými funkcemi virtuálního DOMu.

4.15.1. Virtuální DOM

Jelikož jsou čtení i zápis DOMu prohlížeče režijně náročnými operacemi (kvůli množství atributů na jednotlivých uzlech), frontendové frameworky často zavádí svůj vlastní – virtuální – DOM. Ten je odrazem skutečného, avšak jen s reálně použitými atributy, tudíž alokuje mnohem méně místa. V něm se pak provádějí veškeré operace, načež se pomocí vnitřních mechanismů zesynchronizují (Imsirovic, 2018). Synchronizaci lze ovlivnit pomocí sady funkcí *lazy*, která obalí vykreslovací funkci a zapamatují si konkrétní vstupy i výstupy; pokud se při příštím průchodu neliší, funkce vrací zapamatovaný výstup, jehož pravost je zaručena díky referenční transparentnosti (viz kapitola 3.6.2). S *lazy* je možné využít funkci *keyed*, jež pomocí řetězce identifikuje danou „línou“ funkci a pak ji využívá pro hromadné operace, jako přičítání, odebírání a řazení. Tyto funkce není radno podceňovat, mají velký dopad na rychlost aplikace, především na mobilních zařízeních s operačním systémem Android.

4.15.2. MVU

Jádrem jednoho cyklu programu je model-view-update (MVU) design. V tomto designu jsou tři složky, které vzájemně komunikují přes **zprávy** (anotovány zástupně jako parametr *msg*) a **příkazy**, potažmo **subskripce**. Prvotní impuls celé aplikace probíhá v pevně dané funkci s názvem *main*, která musí vracet ADT *Program* (viz ukázka 11). Většinou je typ vytvořen funkcí *application* z modulu *Browser*, ve které se předají funkce v datové struktuře záznam[11] pro *init*, *view*, *update* a *subscriptions* vytvářející kruh signálů, jak znázorňuje obrázek 3.

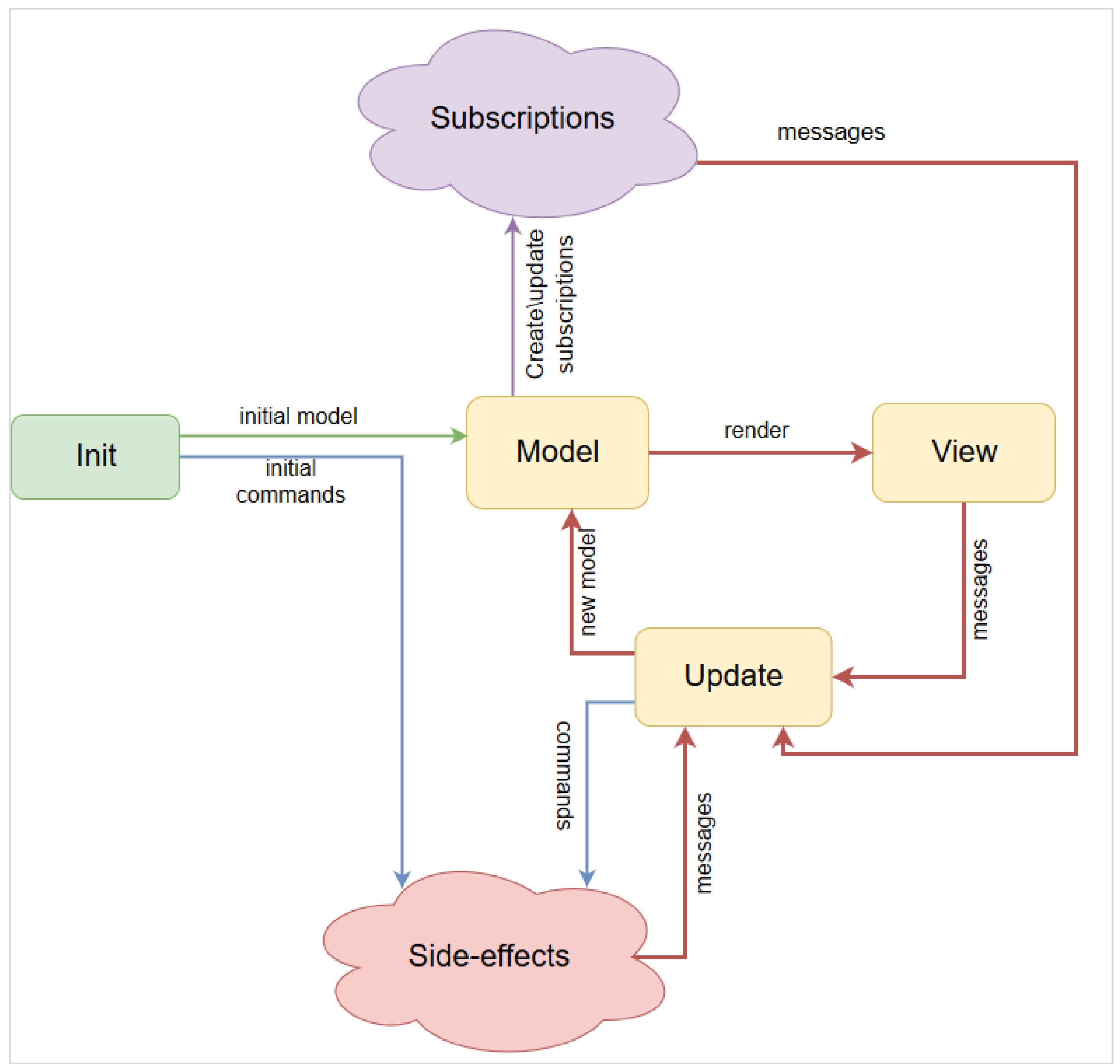
Update

Update je nejdůležitější funkcí celé aplikace. Je automaticky volána s aktuálním modelem při obdržení jakékoliv zprávy, jež se musí vrátit zpět spolu s příkazy k vykonání v datové struktuře tuple.

Zprávou může být jakýkoliv typ (konvenčně označován **Msg**), jako nejvhodnější se – pro heterogenitu dat, rychlý přístup a pohodlné vnořování – používá ADT *Sum type*. Zprávy mohou být iniciovány ze subskripcí (například kliknutí na obrazovce, stlačení klávesy), z prostředí vytvořeného funkcí *view* (například po kliknutí na tlačítko, změny hodnoty tagu *input*), nebo z příkazu (systémový parametrický ADT **Cmd msg**), který dokončil nějakou činnost (například odeslání dat na server, zjištění aktuálního času).

[11] Běžnou konvencí je pojmenování funkce podle indexu, tedy update = update a podobně.

Veškeré změny v aplikaci se provádí pouze ve funkci *update*, která se zároveň po vykonání dané zprávy loguje do debugeru se stavem modelu.



Obrázek 3 - MVU architektura Elmu. Zdroj: Tensor (2018).

Model

Funkcí *init* se v tuplu vytváří model a prvotní příkazy; strukturou odpovídá funkci *update*. Model může nabýt libovolného typu, (konvenčně označován jako **Model**, zástupně se v anotacích používá *model*), většinou však jde o strukturovaný datový typ záznam. Model je – jako všechny hodnoty – immutabilní, při použití je tak dobré dávat pozor, který model je do funkcí předáván[12]. Na konci cyklu aplikace je model opět předán do nového cyklu, takže hodnoty jsou zachovány. Propadávání celého modelu do podmodulů se nazývá *Shared State* (dříve *elm-taco*; Hanhinen, 2017), nebo *Session* (Feldman, 2020).

[12] Jedná se o jednu z nejčastějších chyb, kdy se zprávy přenesou, příkazy vykonájí, ale změny se neprojeví. Změna modelu je možná pouze ve funkci update.

View

Funkce view je určena k interakci s uživatelem. Je zabalena do ADT *Document*, jenž v záznamu přijímá *title* a *body*, kde se již funkcemi do sebe vnořují jednotlivé tagy v podobě parametrického ADT **Html msg**. Paleta funkcí pro tvorbu virtuálního DOMu je pestrá, vesměs jde o všechny běžné značky (*p*, *div*, *a*, *table*, …), navíc je možné vytvořit vlastní tag přes funkci *node*. Kromě typů značek je nutné definovat list atributů (**Attribute msg**) a též list potomků. Pro komunikaci s *update* jsou k dispozici buď atributy přímo s cílovou zprávou (*onClick*, *onChange*), nebo atributy způsobující odchytnutí změny URL (*href*, viz další kapitola).

Standardní ADT *Html msg* z modulu *elm/html* je postačující, ale v praxi se objevuje velmi rozšířený ADT *Element msg* z balíčků *elm-ui* od *mdgriffith*. Výhodu skýtá především v implementaci CSS pravidel pomocí jmenných funkcí, nikoliv přes řetězce, jako je to v *Html*.

```
module Main

import Browser exposing (Document)
import Browser.Navigation
import Html
import Url

main =
    Browser.application
        { init = init
        , update = update
        , view = view
        , subscriptions = subscriptions
        , onUrlRequest = \_ -> ()
        , onUrlChange = \_ -> ()
        }

init : () -> Url.Url -> Browser.Navigation.Key -> ( (), Cmd model )
init flags url key =
    ( (), Cmd.none )

update msg model =
    ( (), Cmd.none )

view model =
    { title = "Hello World"
    , body = [ Html.text "Hello World" ]
    }

subscriptions model =
    Sub.none
```

Ukázka 11 - Nejprimitivnější rozložení MVU aplikace. Zdroj: Vlastní.

4.15.3. Delegování zpráv

Při rozšiřování aplikace do podmodulů se vývojář nevyhne dilematu, jak MVU architekturu škálovat. Počáteční definice jednoho typu zpráv a jednoho modelu ve funkci main má totiž s bobtnajícím kódem omezenou srozumitelnost a koncepci. Běžnou praktikou tak je, že se v aplikaci objevuje více modelů a zpráv, přičemž se různě předávají mezi sebou tak, aby nakonec byla změna zaznamenána v kořenovém modelu. Přístupů, jak delegovat zprávy, je více. Čtyři vzory (Chaves, 2017) jsou v komunitě dobře známé (*Mapování*, *NoMap*, *OutMsg*, *Translator*), jeden (Lentzner, 2019) méně (*Service*) a další dva jsou autorské modifikace stávajících (*Request*, *Higher-Order Delivery*). Konečně, vzory je možné kombinovat.

Mapování

Základní technikou je takzvané mapování. V nadmodulu je *Nadmodul.Msg*, v podmodulu *Podmodul.Msg*. Při nutnosti vyřízení *Podmodul.Msg* se zpráva zabalí do nadmodulní zprávy *(Nadmodul.Msg Podmodul.Msg* – v rovině ADT jde díky typovému konstruktoru o funkci *Msg -> Podmodul.Msg*) a následně přemapuje pomocí funkce *map* (odpovídající *Html*, *Cmd* či *Sub*). Tento typ potom bude zavolán v nadmodulní funkci *update* (se „zabalenou“ zprávou podmodulu) a zpracování zprávy je čistě na nadmodulu; většinou ho pouze provolá do podmodulní funkce typu *update*. Ostatně takto „propustně“ fungují všechny vzory, kdy podmodul neví, co se stane s jeho akcí a jen čeká, zda bude zavolán.

NoMap

Přílišné mapování dalo vzniknout vzoru, při kterém není nutné mapovat v nadmodulu podmodul, jelikož zprávy nadmodulu jsou již v podmodulu přítomny. Problémem je, že podmodul musí vědět o zprávách nadmodulu a naopak, což vytváří smyčku importů. Řešením je odeslání funkce, která provede zabalení *(Nadmodul.Msg Podmodul.Msg)* v podmodulu. Pak není nutné, aby podmodul o onom typu *(Msg -> msg)* věděl.

OutMsg

Nutnost komunikovat z podmodulu do nadmodulu zavedla vzor, ve kterém se vrací tuple se třemi složkami, kde první člen je model, druhý příkazy a třetí člen je návratová hodnota, většinou ve formě zprávy volající po akci nadmodulem (například ShowPopup), může však jít i o datový typ. Návratový typ je definován v podmodulu a je jen na nadmodulu, jak ho zpracuje.

Translator

Opačný přístup, kdy pevně definované možnosti nadmodulu se předají do podmodulu pomocí takzvaného translátoru, což je ve většině případů datový typ záznam. Podmodul tak má k dispozici veškeré zprávy nadmodulu (včetně mapovacího typu na podmodul) a k tomu může přidat vlastní. Nadmodul následně odchycené zprávy buď zpracuje (pokud jde o jeho vlastní), nebo je deleguje do podmodulu.

Service

Vzor, ve kterém se provázaně zpracovávají malé službičky pomocí příkazů a žádostí. V ústředí je jeden sjednotitel, který koordinuje jejich komunikaci a předává vyžádané části modelu k výpočtu. Použití je zatím minoritní.

Request

Kompetetivní přístup ke vzoru *OutMsg*. Podmodul má vlastní zprávy, které si vykoná běžnou cestou. Avšak v návratové hodnotě předává kromě modelu i list žádostí, jež musí implementovat nadmodul. Jeho užití je dobré pro komponenty, které nechtějí mít žádné vedlejší efekty a udělají jen to nejnutnější na vlastním modelu.

Higher-Order Delivery

Vzor vychází od autora práce, přičemž byl testován v praktické části. V kořeni aplikace se nalézá typ zprávy, který očekává funkci ve tvaru *(Model -> (Model, Cmd Msg))*, tedy funkci, do které se aplikuje model a vrací výsledek funkce *update* (tedy tuple modelu a příkazy). Tento typ se též ukládá do modelu a spolu s ním prochází celou aplikací. V případě použití se vytvoří anonymní funkce a přilepí se k ní typ z modelu. Aplikace se pak provede automaticky bez nutnosti dalších zpráv (viz ukázka 19).

4.15.4. URL

Základním kamenem práce s HTML dokumenty je jejich vzájemné prolinkování pomocí URI, potažmo URL (viz kapitola 2.2.1). Princip linkování URL je v Elmu implementován modulem *Document*, tedy v případě SPA ho „vyrábí“ funkce *application* z modulu *Browser*.

Zde je nutné předat strukturu záznam s funkcemi, které řeší příchozí a odchozí komunikaci:

- *onUrlRequest* – před vyžádanou změnou, rozlišuje mezi interní a externí URL, použití nalézá například pro odeslání dalšího požadavku, uložení stavu před odchodem a jiné.
- *onUrlChange* – po změně URL, vyzývá k úpravě modelu dle aktuální adresy. Reaguje například na příkaz *Nav.pushUrl* nebo po kliknutí na štítek *<a>* s atributem *href*.

Tedy je pouze v gesci uživatele, jak bude na změnu URL reagovat. V *onUrlChange* je jako první parametr předán ADT *Url*, jenž obsahuje rozčleněnou URL na strukturu záznam s položkami *protocol*, *host*, *port*, *path*, *query* a *fragment*.

Pomocí modulu *Url.Parser* je možné rozparsovat *Url* přes operátory *</>*, *<?>* a *<#>* na předem určené ADTs hodnoty a uložit je do modelu. Díky tomu je pak možné určit, co bude vráceno funkcí *view* (viz ukázka 12).

```
type Kind
    = Details
    | Registration


url : String
url =
    "uzivatel"


parse : Url.Url -> Kind
parse url =
    let
        pathBasic =
            Url.Parser.s url
    in
    Url.Parser.map Kind
     (Url.Parser.oneOf
        [ Url.Parser.map Details pathBasic
        , Url.Parser.map Registration (pathBasic </> Url.Parser.s "registrace")
        ]
     )
     url
```

Ukázka 12 - Parsování URL. Zdroj: Vlastní.

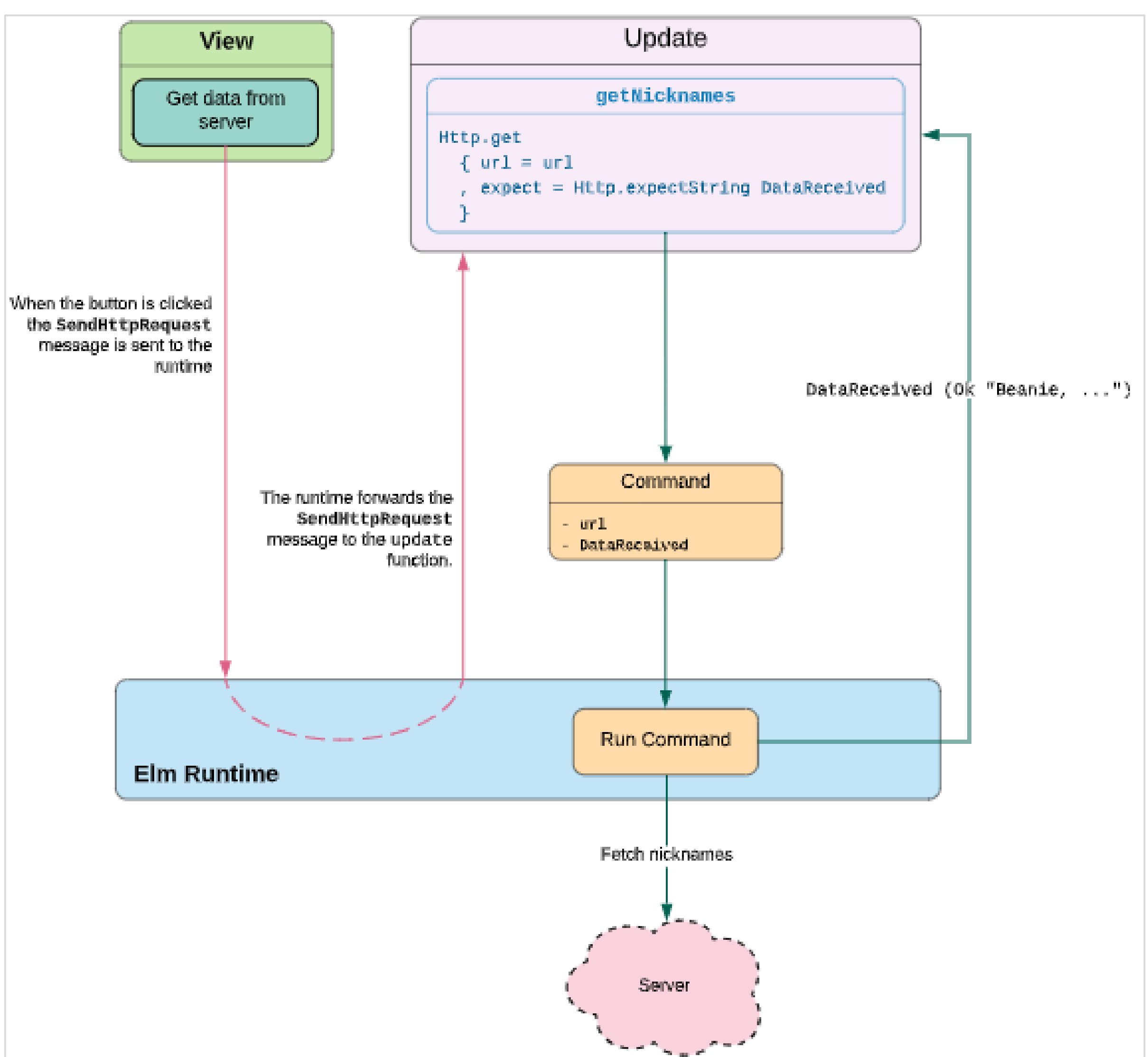
4.15.5. HTTP

Pro webové aplikace je jedním z nejdůležitějších faktorů schopnost komunikovat s okolím. V Elmu je samozřejmě možné vytvářet pouze lokální aplikace, avšak většinové použití počítá s nějakou formou vzdálených požadavků. K tomuto účelu funguje modul *Http*, starající se asynchronně o JavaScriptová volání *XMLHttpRequest*. Funkce *get* a *post* z modulu *Http* vytvoří požadavky odpovídající zvolené metodě, přičemž přijímají

enkódovanou hodnotu jako tělo. Obě funkce využívají obecnější funkci *request*, jež umožňuje nastavení jiných metod, hlaviček či hodnotu timeoutu aj.

Všechny výše zmíněné funkce vrací příkaz *Cmd msg*, který provede samotný požadavek. Odchycení návratové hodnoty pak probíhá pomocí zprávy v *update*, jež je definována v ADT *Expect* jako typový parametr *Result*. Jedná se o trochu komplikovanější konstrukt (viz obrázek 4 s prostým text, nebo ukázka 13 ve formátu JSON), který má za cíl lépe určit obsah odpovědi, případně reagovat na chyby.

ADT *Expect* je vytvářen funkcemi *expectString*, *expectJson*, *expectBytes* a *expectWhatever*. Nejzajímavější je pravděpodobně *expectJson* přijímající kromě zprávy též dekódér z modulu *Json*. Je tak možné rovnou namapovat přijatý JSON na konkrétní typy v Elmu, případně pracovat s chybou přenosu nebo mapování.



Obrázek 4 - Vyvolání a odchycení HTTP dotazů. Zdroj: Poudel (2018).

Kromě příkazů je možné vytvořit subskripci funkcí *track*, která naslouchá a propaguje změny během přenosu unikátně identifikovaného typem *String*. Jde o ADT *Progress* (opět z modulu *Http*), který nabývá dvou hodnot: *Sending* a *Receiving*. Využití nálezá funkce při vizualizaci postupu odesílání a přijímání dat.

```
expect : (Result String a -> msg) -> Json.Decode.Decoder a -> Http.Expect msg
expect msg d =
    Http.expectStringResponse msg
        (\r ->
            case r of
                Http.BadStatus_ m body ->
                    error body

                Http.GoodStatus_ m body ->
                    decode d body

                _ ->
                    error "{msg : \"Něco se pokazilo\"}"
        )

decode : Json.Decode.Decoder a -> String -> Result String a
decode d b =
    Json.Decode.decodeString d b
        |> Result.mapError Json.Decode.errorToString


error : String -> Result String a
error b =
    Json.Decode.decodeString
        (Json.Decode.field "msg" Json.Decode.string)
        b
        |> Result.mapError Json.Decode.errorToString
        |> (\r ->
                case r of
                    Ok e ->
                        Err e

                    Err e ->
                        Err e
           )
```

Ukázka 13 - Zpracování HTTP dotazu. Zdroj: Vlastní.

Websocket

Nabídka oboustranné komunikace přes websockety byla do verze 0.18 doménou oficiálních balíčků *elm-lang*. Do další verze však balíček nebyl převeden, nyní je tak nutné spoléhat na uživatelské implementace přes porty.

4.15.6. Interakce s JavaScriptem

Vzhledem k provázanosti s JavaScriptem je přirozené očekávat komunikaci spojenou s tímto skriptovacím jazykem směrem dovnitř i ven. Pro oba případy platí povinné

přetypování pomocí modulů *Json.Encode* a *Json.Decode*, vztahující se pouze na primitivní a strukturované typy. Jedinou ADT výjimkou je *Maybe*.

JavaScript volá Elm

Kromě známého včlenění do stránek pomocí HTML značky *iframe* je možné integrovat výslednou aplikaci kompilací s přepínačem *output* (viz kapitola 4.6) jako samostatný JavaScriptový soubor (viz ukázka 14). Po načtení do stránek je nutné kód aktivovat zavoláním funkce *Elm.{module}.init* (kde *{module}* je název kompilovaného modulu), která je globálně dostupná. Funkce očekává jako parametr JSON objekt s povinným atributem *node* (ukazující na HTML značku, ve které proběhne vykreslení) a nepovinnými *flags* (předávající data z JS do kódu Elmu).

Vedle použití v prohlížeči modulem *Browser* lze Elm spustit i v terminálovém režimu přes rozhraní *Platform*. JavaScript též může iniciovat výpočet funkce v Elmu přes portem zpřístupněnou subskripční funkci, avšak odpověď již nemusí dostat (viz dále).

```
<script src="/main.js"></script>
<script type="application/javascript">
   const key = "interests";
   let subscription;

   const app = Elm.Main.init({
      node: document.getElementById("app"),
      flags: JSON.parse(window.localStorage.getItem(key)) || []
   });
</script>
```

Ukázka 14 - Spuštění zkompilovaného Elmu. Zdroj: Vlastní.

Elm volá JavaScript

Výše zmíněná technika portů se definuje klíčovým slovem *port* před funkcí určenou k zpřístupnění, zároveň tak musí být označen i modul, jenž tuto funkci obsahuje. Funkce pro naslouchání z JavaScriptu je subskripcí *Sub msg* s parametrem funkce, do které přichází JS hodnota. Funkce pro přenos do JavaScriptového světa je příkaz *Cmd msg*, přijímající přenášené parametry.

K úplnému zprovoznění je nutné na úrovni spuštěné aplikace *Elm.{module}.init* zaregistrovat subskripce na instanci aplikace přes atribut *ports.{name}.subscribe*, případně volat příkaz *ports.{name}.send*, kde *name* je název funkce. Pozor je třeba dávat na to, že co je zamýšleno jako subskripce v Elmu, je nutné vykonat příkazem v JavaScriptu a naopak.

```
-- Port v Elmu
port module Utils.Base64 exposing (put, receive)
port put : ( Int, String ) -> Cmd msg
port receive : (( Int, String ) -> msg) -> Sub msg
subscriptions : Model Msg -> Sub Msg
subscriptions _ =
    Utils.Base64.receive Base64Received      -- type alias Msg = Base64Received (Int, String)


/* Port v JS */
const app = Elm.Main.init({….});

app.ports.put &&
app.ports.put.subscribe(function ([id, s]) {
  app.ports.receive.send([id, btoa(s)]);
});
```

Ukázka 15 - Použití portů. Zdroj: Vlastní.

4.16. Balíčky

Mezi zajímavé balíčky se řadí *elm-ui* pro základní práci s HTML, *elm-mdl* implementující Material Design, *svg* pro vektorovou grafiku, *webgl* pro práci (nejen) se shadery, *elm-physics* s již hotovými primitivy a detekcí hranic, vizualizační nástroje *line-charts*, *elm-visualization*, *elm-graphql* a též *websocket*.

4.17. Implementace

Při implementaci je nutné prvotně rozhodnout, zda půjde o použití vizuální, či výpočetní. Následně o nasazení na backendu, nebo frontendu. Jak již bylo zmíněno, backendová řešení nejsou primární doménou Elmu, avšak ne nemožnou. Frontendové možnosti jsou využívanější, především jednostránková aplikace (SPA). Nakonec není vyloučeno zakomponování SPA do mobilních zařízení pomocí webu (PWA), hybridních technologií (Apache Cordova) či TWA.

Databáze

Pro běh aplikace bude velice pravděpodobně potřeba ukládat data externě mimo model. Buď je možné využít nabídku portů a provést uložení do **lokální** paměti prohlížeče nebo na **vzdálený** server.

V případě lokálních úložišť se nabízí možnosti takzvaných Web Storage, tedy Cookies, Local Storage, Session Storage, IndexedDB, Web SQL nebo Cache API, přičemž každá technologie podléhá jistému omezení (Cohen, 2019). Doporučením pro SPA je IndexedDB. Bohužel, jak již bylo zmíněno, žádné oficiální řešení pro Elm není. K dispozici jsou však komunitní balíčky využívající porty.

Pro práci se vzdálenými daty lze využít přístupových rozhraní, zkráceně API. Druhy API se orientují buď na služby (SOAP, RPC), nebo na zdroje (REST). Přístup je nutné zabezpečit, minimálně šifrovaným přenosem (např. SSL), dále autentizací (např. OAuth) a autorizací. Volání se provádí přes probíraný *Http* modul, data se musí dekódovat a enkódovat.

V praxi se kombinují obě varianty. Vzdálené uložiště slouží pro perzistenci dat, lokální pak pro offline provoz a kopii stavu modelu.

Webový server

Vygenerované JavaScriptové soubory je nutné spolu s dalšími zdroji umístit na webový server, odkud budou poskytované ostatním uživatelům. K tomu lze využít různé technologie, například Apache HTTP Server, nginx, IIS či řešení v rámci Node.js. Hostovat webový server lze na vlastních strojích nebo zprostředkovaně, případně využít předinstalovaných služeb Google Cloud, AWS, Azure atd.

4.18. Ukázky

V rámci diplomové práce byly prozkoumány i možné druhy implementací. S dvěma přišel autor do přímého styku (další v kapitole 6.1).

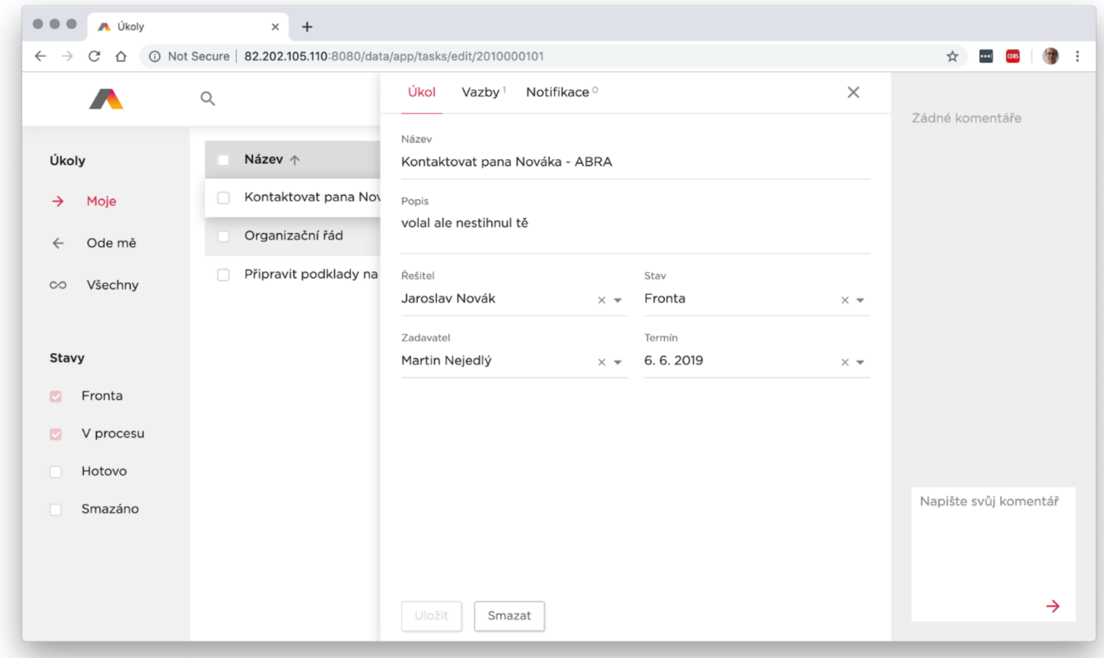
SAVR AB

Fintechová švédská společnost, zabývající se nákupem a prodejem akcií. Funguje především na webu, odkud je portována (ve stylu TWA) do Androidu a iOS. Pracuje s grafy ve formátu SVG (savr.com, 2020).

Obrázek 5 - Mobilní fintechová aplikace. Zdroj: savr.com.

ABRA Software a. s.

U výrobce informačních systému je SPA předávána rovnou v API na unikátní adrese. Jedná se o částečně implementovanou PWA aplikaci typu task management, ve které se přidělují úkoly provázané k určitému business objektu (www.abra.eu, 2020).



Obrázek 6 - SPA jako task management. Zdroj: www.abra.eu.

5. Webová aplikace napsaná v jazyku Elm

Pro demonstraci použití Elmu byl vybrán projekt menší webové aplikace, kombinující SPA s prvky PWA a Web API. Z frontendového hlediska jde o běžný případ, avšak atypicky provázaný s backendem, kdy se zpřístupnění API a tvorba perzistentních entit generuje z frontendové definice mapování. Celý projekt využívá platformu Node.js a je schopný obsluhovat klientskou (Elm) i serverovou (JavaScript) část.

5.1. Analýza

Impulsem pro tvorbu aplikace veřejného „projektovače“ byla současná situace kolem pandemie. Myšlenka minimalistického komunikačního kanálu pro drobné příspěvky však je obecně platná a mohla by najít využití i u jiných příležitostí (charitativní akce, skautské vybavení a podobně). Přes prvotní brainstormingovou techniku lotosového květu (Eklund, 2013) byl nápad rozveden do scénáře, funkčních a nefunkčních požadavků, use case diagramu a diagramu tříd.

Scénář

Jsem ředitel neziskové organizace. Kvůli vládnímu nařízení byla pozastavena činnost některých podniků a služeb. Někteří participanti těchto služeb jsou nyní v hmotné nouzi. Rád bych veřejně prezentoval projekt, který by vysvětlil situaci některých klientů a zpřístupnil údaje k bankovnímu účtu, který by sloužil jako alokátor pomoci. Též by bylo dobré, kdybych mohl přidat k projektu aktuality. Lidé, zajímající se o projekt, by dostali o publikované aktualitě notifikaci. Projekt by měl mít přehledné rozhraní a neměl by omezovat pohodlí přispěvatelů. Dalo by se tedy uvažovat o mobilním zobrazení, či nasazení QR kódů. Zároveň bych z pozice ředitele ocenil, kdybych mohl publikovat více projektů najednou a věděl, jaký je o ně zájem. Pro IT dobrovolníky by mohlo být připraveno veřejné rozhraní. Nakonec by stránky mohly pomoci nejenom neziskovému sektoru, ale i dalším podnikatelům v nouzi.

Požadavky

Vypracované funkční požadavky odráží myšlenky ze scénáře:

1. Publikování projektu pod IČ.
2. Aktualizace stavu projektu.
3. Přidání aktuality k projektu.
4. Zjednodušená příspěvková akce.
5. Notifikace odběratelů.

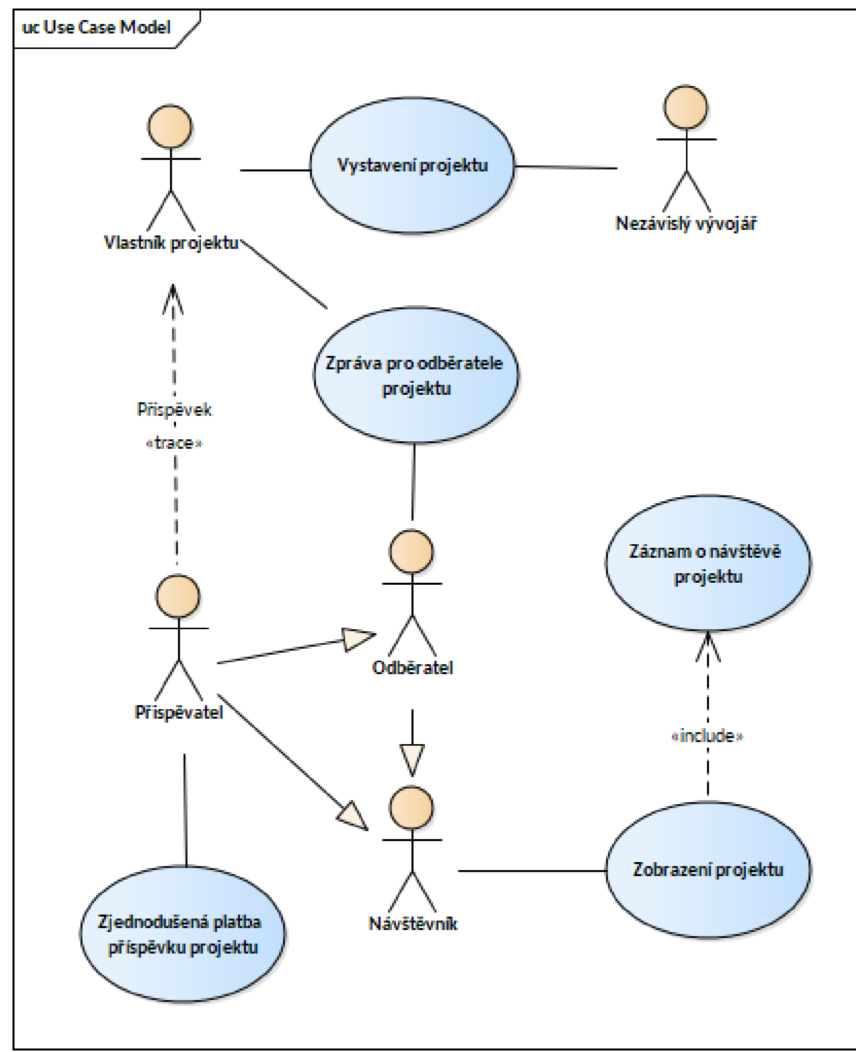
6. Aktivita návštěvníků.
7. Doprovodné informace o stránkách.

Do nefunkčních patří především rychlost a tvorba rozhraní:

1. Rychlost odezvy (250ms).
2. Zpřístupnění dat vývojářům.

Use case diagram

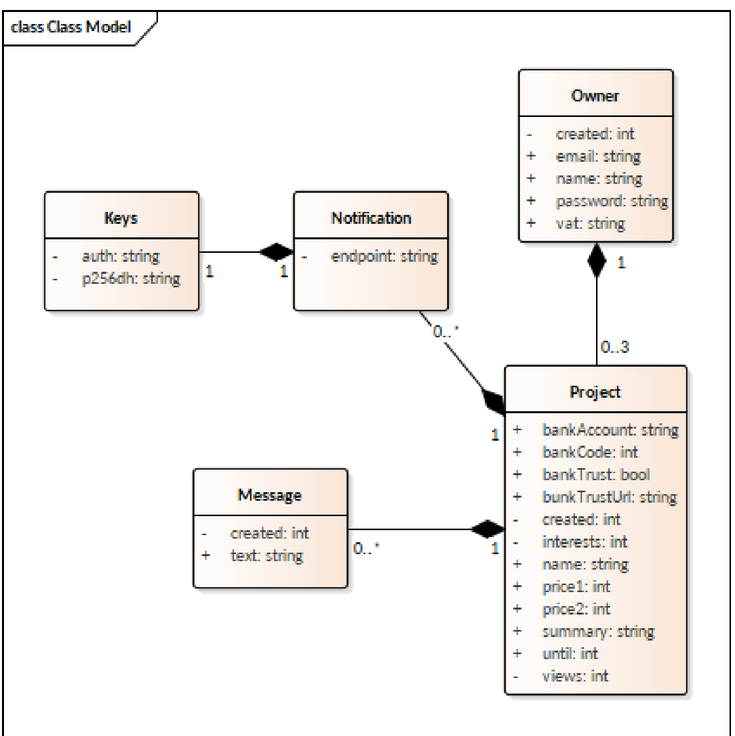
V diagramu užití (viz obrázek 7) jsou zaznamenáni dva základní aktéři, bez kterých by aplikace nedávala smysl. Je to vlastník projektu a návštěvník. Z návštěvníka se odběrem notifikací stává odběratel, při odeslání příspěvku pak přispěvatel (stále je ale i odběratelem a návštěvníkem). Posledním aktérem je nezávislý vývojář, který může využít veřejné rozhraní pro vlastní tvorbu.



Obrázek 7 - Use case diagram projektovače. Zdroj: Vlastní.

Diagram tříd

Ze scénáře a diagramu užití vzešly v diagramu tříd (viz obrázek 8) 3 základní třídy doplněné o 2 technické třídy zajišťující webové notifikace, respektive identifikaci prohlížeče žádajícího subskripci. Všechny třídy jsou vzájemně propojené přes vztah kompozice, tedy smazání jednoho článku zapříčiní kaskádovitou změnu. Tento přístup zaručuje nižší paměťovou náročnost, stejně jako omezení na maximálně 3 projekty pro jednoho vlastníka.



Obrázek 8 - Diagram tříd projektovače. Zdroj: Vlastní.

5.2. Backend

Neduh vytváření aplikací skrze frontendové frameworky přináší backendové dluhy v podobě perzistence dat. Pokud nejde o „fullstack“ framework s rozšířeným frontendem (například Blazor), je nutné rozdělit frontend od backendu a provádět logiku na každé části, což vede k možným duplikacím či naopak odlišnostem.

Manuální provázání obou vrstev bez nutnosti použití frameworku je však oproti očekávání relativně přístupné. Buď backend obsahuje definici svých možností a frontend si ji před spuštěním aplikace stáhne a intrepretuje, nebo se server obsluhuje jazykovou dispozicí (např. Java servlety) a JavaScriptový frontend se z něj, včetně definic, kompiluje.

V případě implementované aplikace této práce se využije opačný postup, kdy backendový ekosystém Node.js v souboru *server.js* je „injektován“ z Elmu pomocí portů a kompilačních skriptů. Elm uvnitř obsahuje definice entit a konfiguraci API, které sám

využívá, například pro tvorbu formulářů. V tomto případě je stav backendu a frontendu konzistentní, přičemž není vyloučeno poskytování vrstev přes různé stanice.

5.2.1. Server

Jak již bylo zmíněno, server využívá Node.js s balíčkovacím systémem NPM. Balíčky obsažené v práci jsou následující:

- *json-server* – modifikace Express.js, poslouchá na portu 8080 (*deploy*) nebo 8094 (*debug*), zpřístupňuje API a statický server pod složkou */be/public/* (*deploy*) nebo */fe/* (*debug*), kde se nachází vygenerované soubory Elmu
- *crypto* – balíček pro tvorbu hesel šifrovaných *sha256*, ukládaných v *base64*
- *web-push* – zpřístupnění skriptů pro tvorbu Web Push Notifications
- *uglify-j*s – zmenšení JS souborů při *deploy*
- *cpx* – univerzální kopírování souborů využívané skripty

K dispozici jsou skripty ve dvou režimech (*deploy* a *debug*):

- *start* – spustí server v režimu *debug*
- *start-deploy* – spustí server v režimu *deploy*
- *debug* – kompiluje Elm s debuggerem
- *deploy* – kompiluje Elm do produkční verze (optimalizace, uglifikace, kopírování do složky */be/public*)
- *generate* – kompiluje databázová pravidla (*scheme.json*, *config.json*) z Elmu pro server

5.2.2. Databáze

Uvnitř závislosti *json-server* je použita JSON databáze *lowdb*. Jde o takzvanou *flat-file* databázi, která se – díky implementaci základních operací (CRUD, stránkování, filtrace,…) a objektovému modelu – dá připodobnit k typu NoSQL. Pro *lowdb* neplatí žádná schémata, jakoukoliv strukturu je možné přidat, vyplní se pouze unikátní *id* jako celé číslo. Pak je možné na základě tohoto *id* editovat a odebírat záznam, opět kýmkoliv.

Tato volnost má velkou výhodu – schéma je možné libovolně vymyslet a implementovat na frontendu, na backendu se ověří před samotnou prací s databází. Na druhou stranu je nutné naprogramovat veškerou režii. Více v kapitole 5.4.1 o tvorbě API.

5.3. Frontend

Při práci na frontendu byl kladen důraz na co nejmenší počet závislostí a využití pouze balíčků od tvůrců Elmu, což se nakonec povedlo. Balíčky jsou uspořádány v *elm.json* ve tvaru *majitel/název*, kde majitel je pouze „jaderný“ *elm*. Jde o následující balíčky:

- *elm/browser* – spuštění aplikace, debugger, virtual DOM
- *elm/core* – datové struktury, Cmd, Process, Task
- *elm/html* – značkovací funkce, atributy a události
- *elm/http* – komunikace, vytváření dotazů
- *elm/json* – dekodér a enkodér
- *elm/parser* – parsování řetězců na typy
- *elm/time* – práce s formátem posix
- *elm/url* – transformace URL do řetězce a naopak

V průběhu programování vyvstaly potřeby nových funkcionalit, které by se daly importovat jako závislosti. Avšak šlo pouze o malé funkce, které byly implementovány vlastními silami. Nachází se ve složce *fe/src/utils*.

Jiné externí CSS či JS použity nebyly, vše je generováno Elmem, jehož výsledný soubour *main.js* je synchronně vložen do stránek. Dalšími soubory jsou:

- *webmanifest* – k dodefinování vlastností PWA (ikona, typ zobrazení, …)
- *vapid.js* – veřejný klíč pro Web Push Notifications, který se generuje při prvotním zapnutí aplikace
- *favicon.ico* – ikona webu
- *sw.js* – service worker řešící subskripci Web Push Notifications
- *index.html* – nezbytné značky, spuštění *Elm*, předání flags a inicializace portů

5.4. Programování

Během programování bylo použito vývojové prostředí WebStorm s *elm-format*. Ke kompilaci sloužila verze Elmu 0.19. Aplikace se dělí na backendovou (*/be/*) a frontendovou část (*/fe/*). K verzování byl využit systém GIT se vzdáleným umístěním u společnosti Bitbucket na adrese *https://bitbucket.org/bohemak2/dont-shut-the-shop-down*. V repozitáři je pět větví, ve kterých lze sledovat postupný vývoj a rozdíly mezi přístupy s anotacemi a bez nich:

- *pure* – původní větev, která byla pro co nejrychlejší vývoj minimálně anotována

- *annotation* – větev zavádějící všechny anotace na *pure*, v důsledku čehož bylo nutné přemístit některé typy na okraje stromového uspořádání
- *theme* – navazuje na annotation zavedením vzhledu
- *master* – nejnovější změny od *theme*, které již nejsou v *pure* ani *annotation*
- *deploy-test* – zrcadlo *masteru*, které má upravený gitignore tak, aby v něm byly sledovány změny v adresářích (*/be/public/*, */be/db/*, */be/key/*), které jsou nutné k okamžitému spuštění s testovacími daty bez nutnosti kompilace

Pro lepší názornost se používají v celé aplikaci plné cesty k funkcím, které nebyly součástí vlastní práce a pocházejí z balíčků tvůrců *elm*; tedy místo *Html exposing (..)* a použití funkcí *p*, *div* aj., jde o *Html.p*, *Html.div* aj.

5.4.1. Tvorba API

Jak již bylo zmíněno, k JSON databázi se přistupuje přes *json-server*, který pomůže i se základními *status code* odpověďmi (404, 201, 200, …). Přístup do tzv. WEB API je modifikován architekturami REST a RPC:

- *REST* – CRUD operace, povolené metody GET, POST, PATCH, DELETE
- *RPC* – služby a příkazy (login, subskripce, inkrementace), povolené metody jsou GET a POST

Přičemž některé operace či příkazy jsou chráněny (viz dále Bezpečnost). Před cestou k architektuře ještě stojí prefix *api* a verze API, nyní jde o řetězec *v0*. Konfigurace je sdílena z Elmu pomocí generování konfiguračních JSON souborů ze skriptů umístěných v */be/utils/* do pomocných souborů v adresáři */be/db/* :

- *config.json* – cesty k REST a RPC, název entity uživatelů (ověření), velikost řetězců, které entity se kaskádově mažou a které jsou chráněny před GET
- *scheme.json* – entity a jejich atributy (*jméno, typ$podtyp)* z Elm funkce *db*, přičemž typ je *int*, *string*, *bool, ignore* a *, podtypy jsou rozděleny dle typu, například *string$bankCode*, *string$vat*, *int$posix* aj.

Schéma

Typ s podtypem udávají sérii pravidel, které se postupně aplikují na backendu pomocí objektů *Promise*, kde návratová hodnota předává při úspěchu (*Promise.resolve*) hodnotu, při neúspěchu (*Promise.reject*) řetězec s chybou (viz ukázka 16). Například při tvorbě emailu se nejdříve kontroluje, zda jde o řetězec, následně zda má požadovanou délku a až nakonec

regulárním výrazem odpovídající formát. V typu *emailUnique* je pak kontrola rozšířena o unikátnost v databází. Podobně funguje prověřování IČ (8 čísel) nebo url (regulární výraz).

```
-- Elm v Entity.Message
type alias Model =
    { id : Int
    , projectId : Int
    , created : Time.Posix
    , text : String
    }

db : Rules
db =
    [ ( "projectId", "int$fk" )
    , ( "created", "int$posixOnCreate" )
    , ( "text", "string" )
    ]

/* Vygenerovaný scheme.json */
{"message":[["projectId","int$fk"],["created","int$posixOnCreate"],["text","string"]], … }

/* Backend - server.js, config.delimiter = $, router = JSON server */
…
case "int" + config.delimiter + "fk":
    return (t, p, e) => isInt(t, p).then(p => isFk(t, p, e));
case "int" + config.delimiter + "posixOnCreate":
    return (t, p) => createPosix();
case "string":
    return (t, p) => isString(t, p).then(p => isShort(t, p));
…
function isString(t, p) {
    return typeof p == "string"? Promise.resolve(p) : Promise.reject(Error(t + " není string"));
}

function isShort(t, p) {
    const np = p.trim();
    return np.length === 0 ? Promise.reject(Error(t + " je prázdný")) : np.length < config.string.short ?
Promise.resolve(np) : Promise.reject(Error(t + " je příliš dlouhý"));
}

function isInt(t, p) {
    return typeof p === "number" ? Promise.resolve(p) : Promise.reject(Error(t + " není int"));
}

function isFk(t, p, e) {
    return router.db.get(t.replace("Id", "")).getById(p).value() !== undefined ? Promise.resolve(p) :
Promise.reject(Error(t + " není fk"));
}

function createPosix() {
    return Promise.resolve(Math.floor(new Date()));
}
…
/* type alias Error = { text : Maybe String, code: Maybe Int } */
function Error(t, c) {
    return {t: t, c: c,}
}
```

Ukázka 16 - Generování schéma z frontendu. Zdroj: Vlastní.

Po úspěšném průchodu je v *lowdb* uložena konečná návratová hodnota, takže je modifikace umožněna v kterékoliv části. Následně se upravený záznam vrátí v odpovědi jako výsledek dotazu – čímž frontend dostane informaci o reálně uložených datech a může to reflektovat. Technika se používá u všech atributů typu *string* (trim řetězce), vytvoření aktuálního času typem *int$posixOnCreate* (vytvoření položky) či zašifrování typu *string$hash* (heslo).

Struktura

Ačkoliv téma práce a samotná knihovna *lowdb* vybízí ke grafové struktuře, nakonec byl zvolen relační model. Důvodem byla především ukázka práce se světově dominantním modelem, zástupně pak nejasné efektivní rutiny ohledně strukturování, volání a mapování takového grafu (což ve své knize nově osvětluje Feldman, 2020). V práci jsou tak v kořenu JSON souboru v listu uspořádány všechny entity, přičemž se klíč drží vždy na straně 1:M, respektive v jednom případě 1:1 u notifikace.

Bezpečnost

V RPC jsou všechny GET a POST metody přístupné bez autentizace, s výjimkou procedury *login*. Jde o inkrementaci či dekrementaci zobrazení projektů, stejně tak jako o subskripci a odhlášení notifikací. Teoreticky je tak možné vzdáleně upravit počty přes automatické HTTP dotazy, v reálném nasazení by se toto dalo ošetřit přístupem z konkrétní domény jako *referer* či zavedením tokenů pro uživatele.

V REST je GET povolen pro všechny entity, kromě zmíněných v listu *protected* generovaných do *config.json* (což se nyní rovná entitě *notification* s daty o prohlížečích vyžadujících notifikace projektu). V ostatních případech je nutné se řídit Basic OAuth, neboli *Basic access authentication*.

Pokud tedy přijde žádost s metodou POST, PATCH, nebo DELETE, je nutné vyplnit hlavičku *Authorization* s hodnotou *Basic* a enkódovaným base64 řetězcem ve tvaru *email:heslo*. Následuje kolečko *Promise*. Jde o již zmíněnou autentizaci, dále autorizaci (zda objekt patří autorizovanému uživateli), kontrolu pravidel pro publikování (3 projekty, 1 zpráva za 2 dny na projekt), kontrolu dat oproti *scheme.json* a nakonec notifikaci.

Při průchodu může kterákoliv *Promise* vrátit chybovou hlášku s konkrétním status kódem, většinou jde o 401 (přihlášovací údaje nesedí), 404 (nenalezeno), 405 (není vlastník), 406 (nesplňuje podmínky).

```
-- Vytvořené Http.Header v modulu Api, k vytvoření hash slouží porty (viz ukázka 15)
headers : Maybe Credentials -> List Http.Header
headers c =
   case c of
      Just c_ ->
         [ Http.header "Authorization" ("Basic " ++ c_.hash) ]
      Nothing ->
         []

/* Backend - server.js – vypořádání s REST požadavkem */
if (config.protect.includes(entity)){
   setResponse(res, 406, entity + " je chráněn");
} else {
   if (req.method === "POST" || req.method === "PUT" || req.method === "PATCH" || req.method ===
"DELETE") {
      if (type === config.rest) {
         isAuthorized(req, entity)
            .then(user => itsObject(req, entity, id, user))
            .then(user => hasLimits(req, entity, id, user))
            .then(user => completeBody(req, entity, id, user))
            .then(user => notificate(req, entity, id, user).then(() => { next(); }));
            .catch(e => setResponse(res, e.c !== undefined ? e.c : 406, e.t !== undefined ? e.t : "Neznámá
chyba"));
      }
   }

/* OAuth - Authorization: Basic base64(email:pwd) */
function isAuthorized(r, e) {
   const auth = r.headers["authorization"];

   if (r.method === "POST" && e === config.users) {
      return Promise.resolve();
   } else if (auth !== undefined) {
      const [type, base64] = auth.split(" ");
      const [email, password] = Buffer.from(base64, "base64").toString().split(":");

      return createHash(type, password).then((p => {
            const u = router.db.get(config.users).filter({
               "email": email,
               "password": p
            }).value();
            return u.length === 1 ? Promise.resolve(u[0]) : Promise.reject(Error("Přihlašovací údaje nesedí",
401));
         }
      ));
   } else {
      return Promise.reject(Error("Nejsou vyplněny přihlašovací údaje", 401))
   }
}
…
function createHash(t, p) {
   try {
      return Promise.resolve(crypto.createHmac('sha256', '94_salt').update(p).digest("base64"));
   } catch (e) {
      return Promise.reject(Error(t + e.toString()))
   }
}
```

Ukázka 17 - Basic OAuth na klientu a serveru. Zdroj: Vlastní.

5.4.2. Business logika

Frontendová část (*/fe/*) je v tlustém klientu silně spjata s business logikou. Je pouze na klientu, co uživateli vykreslí[13] a jak na to bude aplikace reagovat. Aplikace řídí svůj stav podle aktuální URL[14], pro předávání zpráv byl využit kombinovaně návrhový vzor *Mapování* a *High-Order Delivery* (viz kapitola 4.15.3).

V adresáři (*/fe/*) je místo pro assety (*assets*), Vapid klíč (*key*) a zdrojové kódy (*src*). Zde jsou umístěny všechny Elmovské soubory nutné pro kompilaci klienta. Logika je rozdělena dle adresářové struktury:

- *Api* – pro komunikaci s Web API modulem *Http*
- *Component* – nezávislé funkce, které se importují napříč aplikací
- *Entity* – definice cesty, modelu, dekodéru, enkodéru a formuláře entit
- *Page* – různé druhy stránek, každá vrací cestu, parsovací funkcí a zobrazení
- *Utils* – moduly, které se za normálních okolností dají stáhnout jako balíček

Při konkrétním případu pak dojde k volání přes tečkovou notaci, např. *Page.Projects.view*. Tím vzniká přehledná stromová struktura, kam se další „potomci" vnořují, např. *Page.Projects.Detail.view*. V kořeni jsou umístěny rozcestníkové moduly *Api*, *Entity* a *Page*, které určí další větvení podle modelu umístěného v *Main*.

Model

Základní model má strukturu *(Page.Kind, Session msg)*, přičemž označení *Kind* je ze zvyklosti disjunktní sjednocení typů (*ADT Sum type*) s možným výčtem typů daného modulu, v tomto případě jde o určení, na jaké stránce/modulu (*Home* | *About* | ...) se zavolá funkce *view*.

Session msg je pak typickým představitelem modelu jakožto kartézského součinu typů (*ADT Product type*) v datové struktuře záznam, která se předává do podstránek jakožto „nositel pravdy" – co zde není, neexistuje (viz ukázka 18).

[13] Nakonec je nutné neustále pamatovat, že frontend je tak bohatý, jak mu backend dovolí a při práci s business logikou musí oba vycházet ze stejných pravidel.

[14] Některé aplikace však ignorují, že by měl stav modelu odpovídat hodnotě URL, čímž ani nedokáží reprodukovat stav z URL.

Session

Komponenta *Session* stojí mimo stromovou strukturu a má vlastní funkce *init* a *update*, které se aplikují na model (*Session.Model*) pomocí zpráv (*Session.Messages*). Jako zprávy jsou předávány několika aritní ADT typy *Form* a *Request*, jež přijímají *Entity.Kind* a *Type*, kde typem je určitá žádost (např. *Read*) nad konkrétní entitou (*Entity.Message*). Skládáním vznikají různé akce, například vytvoření nového uživatele se provede přes odeslání zprávy *(Request Entity.Owner Type.Create)*.

```
-- Component.Model
type alias Model msg =
    ( Page.Kind, Session msg )


-- Page.Kind
type Kind
    = About
    | Contact
    | Home
    | Instructions
    | NotFound
    | Projects Page.Projects.Kind
    | User Page.User.Kind


-- Session.Model
type alias Session msg =
    { credentials : Maybe Credentials
    , flags : Flags
    , key : Browser.Navigation.Key
    , loaded : ( Bool, Bool )
    , login : Form.Model
    , message : Form.Model
    , messages : Dict Int (List Entity.Message.Model)
    , msg : Session.Msg -> msg
    , cmd : Cmd msg -> msg
    , owner : Form.Model
    , owners : Dict Int Entity.Owner.Model
    , ownersRequest : Api.Request (List Entity.Owner.Model) msg
    , project : Form.Model
    , projects : Dict Int Entity.Project.Model
    , projectsRequest : Api.Request (List Entity.Project.Model) msg
    , toasts : Dict String Toast.Model
    , url : Url.Url
    , zone : Time.Zone
    }
```

Ukázka 18 - Model aplikace. Zdroj: Vlastní.

Higher-Order Delivery

ADT *Session msg* je parametrický, a to kvůli umístění zprávy *Msg* z *Main*, která očekává funkci *(Session.Msg -> msg).* Díky tomuto přístupu je možné volat zprávy definované v modulu *Session* kdekoliv v aplikaci. O samotné provedení se opět postará modul *Session* svojí funkcí *update*, kam propadne z *Main.update* spolu se zprávou *Update*.

Zpráva *Update* očekává funkci ve tvaru *(Model Msg -> (Model Msg, Cmd Msg))*, nejčastěji anonymní, například *(\model -> ({ model | loaded = True }, Cmd.none)).* Tento *Higher-Order Delivery* princip zaručí, že je předložen do *Main.update* pouze „předpis" (ukazatel) funkce a samotná aplikace se provede až s čerstvým modelem.

Výhodou je nejen čerstvost dat, ale i možnost odeslat další požadavek (viz příloha 9.1). Vše je navíc znásobeno užitím *updateEntity* a *updateEntityAll* v *Session*, kdy se funkce aplikuje pouze v případě, že proběhla úspěšně. Při neúspěchu se místo toho neprovede ani změna, ani příkaz; naopak se odešle žádost o zobrazení chyby (viz ukázka 19).

```
-- Update, který se má vykonat v updateEntity, pokud bude úspěšná odpověď a dekoding
f = \( page, session ) p -> ( ( page, { session | projects = Dict.insert p.id p session.projects } ), Cmd.none )

-- Parametr r je ADT Result, v příkladu z prvního řádku by šlo o (Result String Entity.Project)
updateEntity f r =
    \( p, s ) ->
        case r of
            Ok e ->
                f ( p, s ) e

            Err e ->
                ( ( p, s ), Toast.error "err" e )
```

Ukázka 19 - Zjednodušená část High-Order Delivery. Zdroj: Vlastní.

Modul *Main* je tak odstíněný od okolního ruchu a stará se pouze o výměnu typu stránky na základě odchycené zprávy o změně v URL a o subskripce komponent. Zbytek práce okolo *Session msg* deleguje na modul *Session* pomocí funkce *update* a *load*.

Formuláře

Vstupy uživatele jsou (nejen) v Elmu považovány za narušitele referenční transparentnosti, tudíž je k nim omezený přístup. Je tak zakázáno zjišťovat aktuální hodnotu značek, kromě grafických rozměrů [15]. Změny na značkách musí být odchytávány a propagovány do virtuálního DOMu přímo od uživatele, jinak se o ně při překreslení přijde. Pro odchyt lze

[15] Samozřejmě jde omezení obejít přes porty.

využít zprávu *Form* ze *Session.Msg*, kdy je předána do *Form.view* a poté přemapována na *Main.Msg*.

```
-- Page.view: Zavolání vykreslení formuláře owner ze Session, předání zprávy pro změny
Form.view (Session.Form Entity.Owner (Session.Create Nothing)) s.owner

-- Form.view: Napojení zprávy na událost onSubmit
changeAttr : (FieldId -> String -> msg) -> FieldId -> Html.Attribute msg
changeAttr msgChange id =
    Html.Events.onInput (msgChange id)

-- Session.update: Odchycení zprávy a probublání do Form.update
case msg of
  …
  Session.Form k t id v ->
    case k of
      Entity.Owner ->
        case t of
          Session.Login Nothing ->
            ( { s | login = Form.update id v s.login }, Cmd.none )

          _ ->
            ( { s | owner = Form.update id v s.owner }, Cmd.none )

-- Form.update: Změna starého modelu, vrácení nového
update : FieldId -> String -> Model -> Model
update id v m =
  case Dict.get id m.inputs of
    Just i ->
      let
        newInput =
          { i | value = Just v }
      in
      { m | inputs = Dict.insert id newInput m.inputs }

    _ ->
      m
```

Ukázka 20 - Napojení formuláře na event onSubmit. Zdroj: Vlastní.

Porty

Alternativním typem komunikace jsou porty. Nejen, že se dají využít pro zjištění hodnoty ze světa JavaScriptu, ale fungují jako takové „červí díry“ napříč celou aplikací.

Toho je využito při implementaci dialogu *Toast* – kdekoliv v aplikaci je možné odeslat příkaz k vytvoření toastu (*success*, *error*, *info*), načež se ihned vynoří v modulu *Main* pod subskripcí *Toast.add*, aby se následně, po časovém úseku definovaném JS *setTimeout*, zlikvidoval pod subskripcí *Toast.remove* (viz ukázka 21). Použití může být i bez zpoždění, dojde pouze k jakémusi „proxy“ předání. Omezením v takové komunikaci je nutnost enkódovat a zase dekódovat hodnoty, což je například pro složitější tvary či opaque types nemožné.

Kromě tohoto speciálního typu se porty v aplikaci používají pro převod řetězce do *base64* a především tradičního přístupu k *localStorage*, kde se nachází informace o odběrech.

```
port module Component.Toast exposing (Model, add, … , remove)
…
type alias Model =
   { id : String
   , kind : Kind
   , message : String
   , sleep : Int
   }
…
add : (Maybe Model -> msg) -> Sub msg
add msg =
   addToast (msg << Result.toMaybe << Json.Decode.decodeValue decoder)

remove : (String -> msg) -> Sub msg
remove msg =
   removeToast msg

decoder : Json.Decode.Decoder Model
decoder =
   Json.Decode.succeed Model
      |> Json.Decode.Extra.andMap (Json.Decode.field "id" Json.Decode.string)
      |> Json.Decode.Extra.andMap (Json.Decode.field "kind" <| Json.Decode.map fromString
Json.Decode.string)
      |> Json.Decode.Extra.andMap (Json.Decode.field "message" Json.Decode.string)
      |> Json.Decode.Extra.andMap (Json.Decode.field "sleep" Json.Decode.int)

port putToast : Json.Encode.Value -> Cmd msg
port addToast : (Json.Encode.Value -> msg) -> Sub msg
port removeToast : (String -> msg) -> Sub msg

/* JS: Toast port */
app.ports.putToast &&
app.ports.putToast.subscribe(function (t) {
   app.ports.addToast.send(t);
   if (t.sleep > 0) {
      setTimeout(function () {
         app.ports.removeToast.send(t.id)
      }, t.sleep);
   }
});
```

Ukázka 21 - Porty pro odesílání zpráv napříč aplikací. Zdroj: Vlastní.

5.4.3. Prezentační vrstva

U téměř každé funkce vytvářející ADT *Html msg* je nutné předat list (*Html.Attribute msg*), který modifikuje atributy HTML značek. Jde o téměř kompletní výčet ze standardu W3C, včetně událostí, jež se vytvářejí z modulu *Html.Events*, například funkce typu *Html.Events.onClick*, *Html.Attributes.href*, *Html.Attributes.id* aj.

Nabízí se tak aplikování CSS pravidel funkcí *Html.Attributes.style*, což zajistí též dynamickou výměnu při změně hodnot. Za tímto účelem byla vytvořena vlastní modulová komponenta s názvem *Theme* založená na technologii *FlexBox*. V modulu jsou k dispozici

listy CSS pravidel, jež se dají skládat za sebou, například *(Theme.centerSelf ++ Theme.spacing)* vytvoří atribut *<…style="padding:.5em;align-self:center">*. Kromě samostatných pravidel jsou připraveny i shluklé listy do obecnějších vzorů, například *Theme.button*. Filosofie spojování listů zaručí, že poslední změny přepíší již předešlé, tedy je možné vzor přizpůsobit.

Samozřejmě není ideální takto označit všechny značky, lepší je pro určité skupiny vytvořit společné pravidlo. Možností je využít *Html.Attributes.class* a externího CSS souboru. Nebo založit vlastní stylopis pomocí funkce *Html.node*, která vytvoří libovolnou značku – proč ne tedy *<style>* s potomkem *Html.text* držícím daná pravidla (v práci implementováno jako *Theme.render*). Podobně by se dalo pracovat i s assety.

```
-- Component.Theme
style : String -> String -> Attribute msg
style =
    Html.Attributes.style       -- alias, funkce má stejnou anotaci

color : String
color =
    "#131313"

button : Attributes msg
button =
    [style "padding" ".33em .5em", style "color" color, … ]

render : String
render =
    "html,body{height: 100%;margin: 0…"

-- Vytvoření <style>
viewStyle : Html msg
viewStyle  Html.node "style" [] [ Html.text Theme.render ]

-- Vytvoření <article style="…">
viewArticle : Html msg
viewArticle = Html.article (Theme.column ++ Theme.spacing ++ [ Theme.style "min-width" "160px"]) …
```

Ukázka 22 - Komponenta Theme pro prezentační vrstvu. Zdroj: Vlastní.

V prezentační vrstvě nebyly vytvořeny žádné jednotící UI komponenty, jako *Button*, *Link*, *Card* a podobně, jelikož k sjednocení vzhledu prozatím postačily vzory z modulu *Theme*. Na pár místech se využívá stejného view, například *viewAwareness* pro upozornění nad pravostí projektů.

5.4.4. Web Push Notifications

Odeslání a příjem webové notifikace probíhá čistě v režii JavaScriptu (viz ukázka 23), avšak akce k odběru, její výsledek a následný zápis do databáze je řešen z Elmu. Zprávy na sebe navazují tak, že pokud některý článek vypadne, nedojde k odeslání dat do databáze.

Přesto je implementace náchylná na chyby a je ukázkou, jak tříštění logiky mezi Elmem a JavaScriptem není správným přístupem. Bohužel, oficiální rozhraní pro získávání informací o udělených privilegiích prohlížeče, či možnostech kamery, mikrofonu apod. zůstavají stále hudbou budoucnosti.

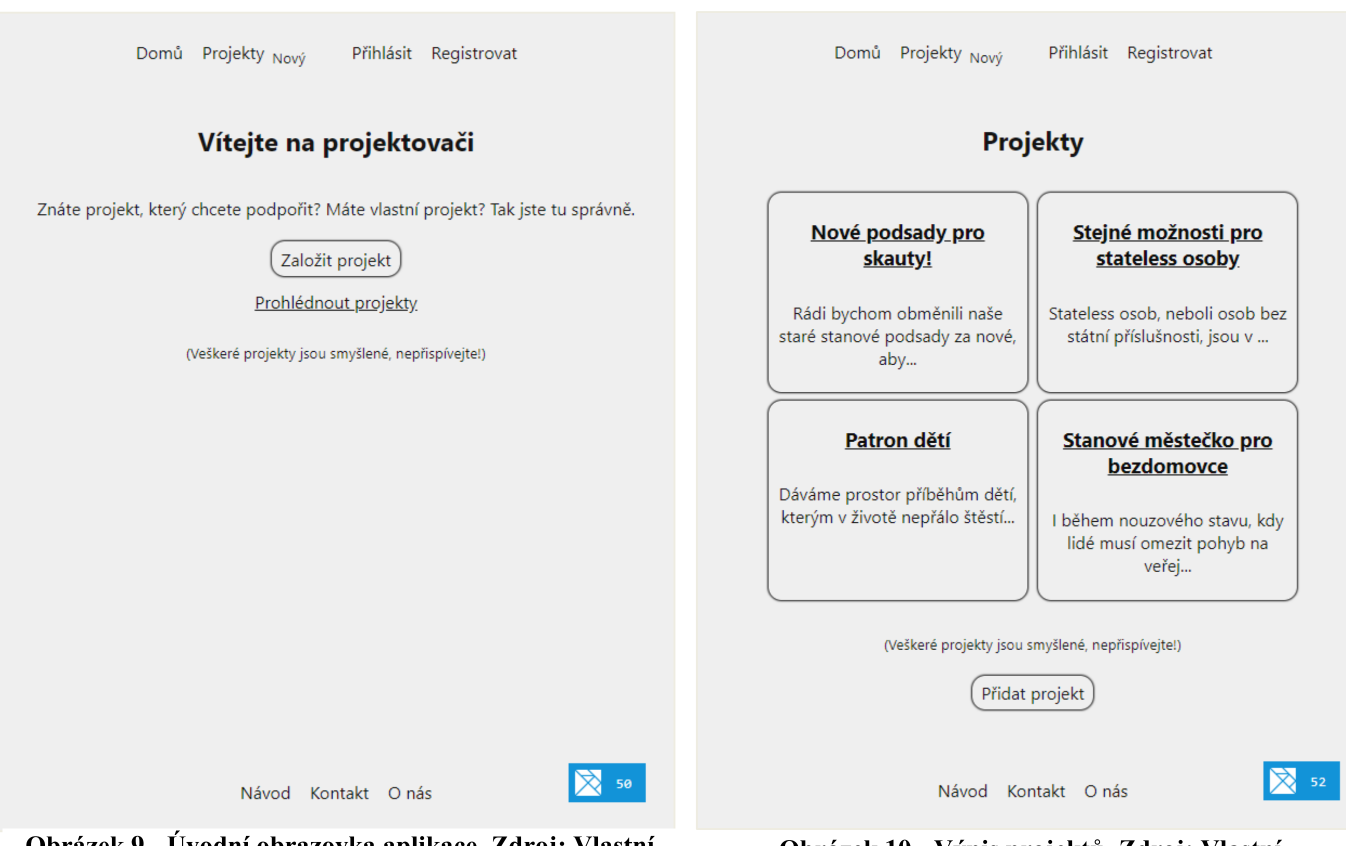
```
/* Předpoklad wp – web-push, router = JSON server, r = Request , e = Entita notifikace */
const ns = router.db.get(config.notifications).filter({"projectId": pId}).value();
for (const n of ns) {
   const payload = JSON.stringify({
      title: p.name,
      body:  r.body["text"].substr(0, r.body["text"].length < 100 ? r.body["text"].length : 100) || "Nová
zpráva",
      entity: e,
      id: p.id
   });

   wp.sendNotification(n, payload)
}
```

Ukázka 23 - Odesílání Web Push Notificatons. Zdroj: Vlastní.

5.5. Výsledek

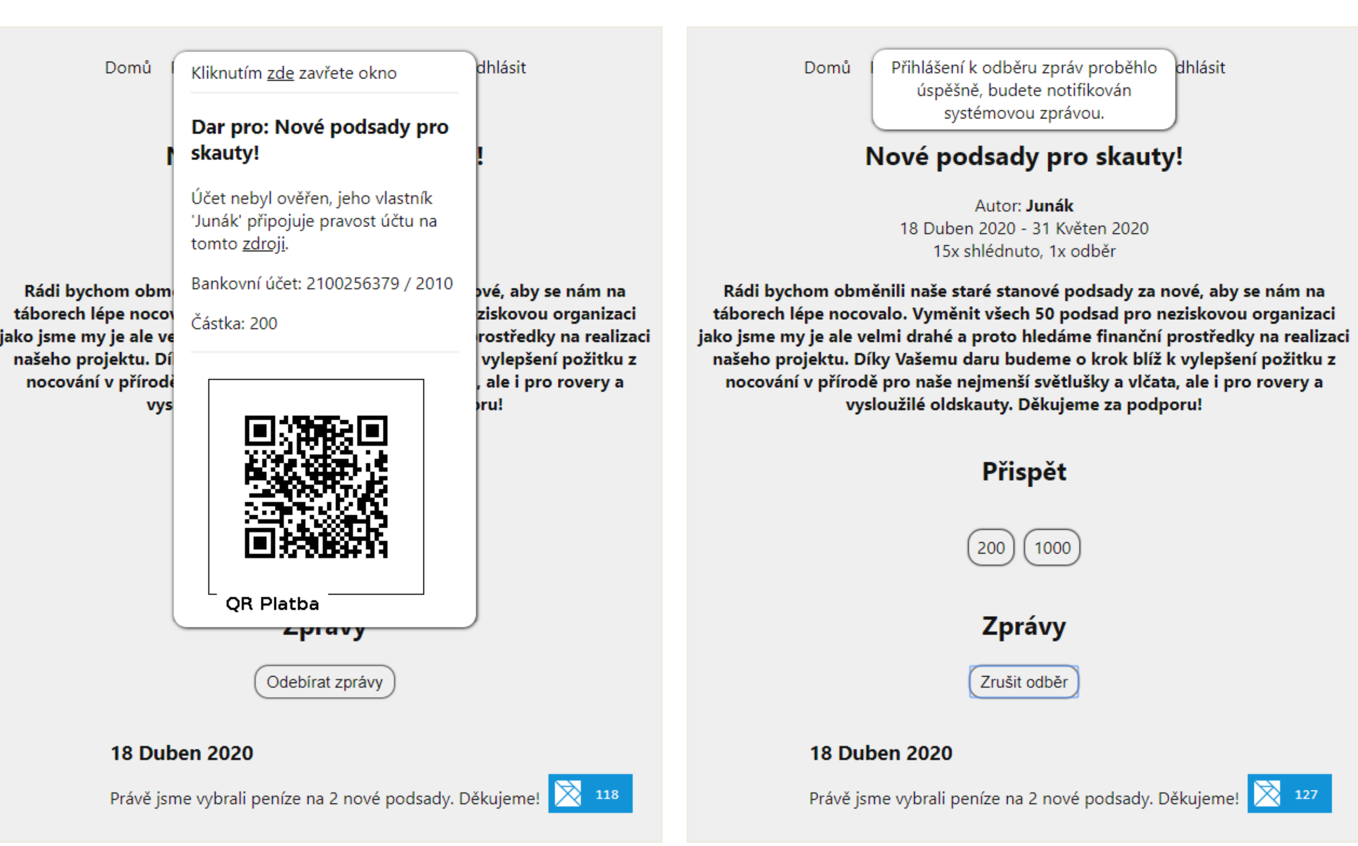
Výsledná webová aplikace obsahuje 9 stránek, 4 entity, 5 komponent a 4 rozšíření. Zdrojový kód čítá cca 700 řádků JavaScriptu, cca 3700 řádků Elmu a cca 100 řádků portů, přičemž po kompilaci zabírá deploy verze 72 kB a debug verze 435 kB, včetně řetězců obsahu stránek, které by se daly přiložit v samostatném souboru (balíček *elm-i18next*). Z grafického hlediska nebyly použity žádné assety, web je pouze prototypem a čeká na zásah designéra.

V práci již nebyly dotaženy některé funkcionality, buď z časových, nebo nad rámec práce přesahujících důvodů. Na frontendu jde o postupné načítání projektů stránkováním či nekonečným posuvníkem, využití definic entit v evaluaci formuláře ještě před odesláním na server a v neposlední řadě dynamické zjišťování odběru zpráv u projektu. Na backendu pak rozšíření identifikátoru entity o „hezkou“ URL podle jména (projekt, uživatel).



Obrázek 9 - Úvodní obrazovka aplikace. Zdroj: Vlastní

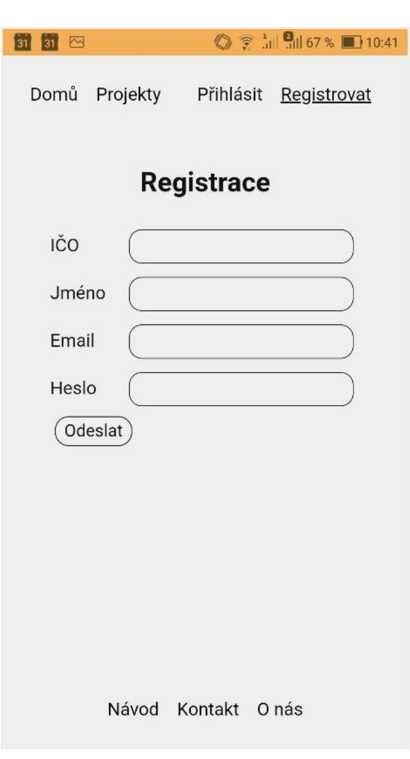
Obrázek 10 - Výpis projektů. Zdroj: Vlastní.



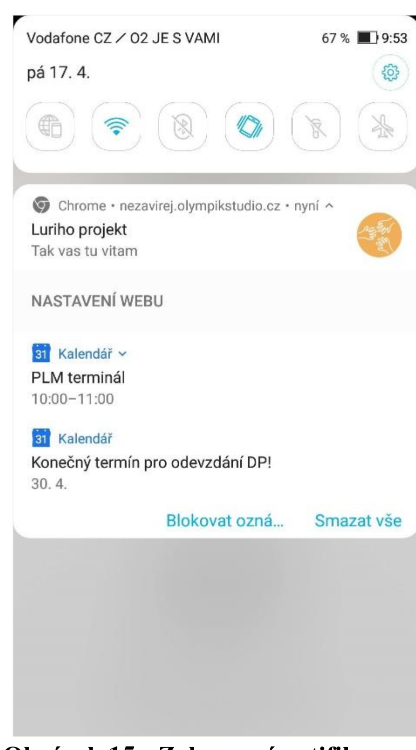
Obrázek 11 - Darování částky. Zdroj: Vlastní.

Obrázek 12 - Odebírání zpráv. Zdroj: Vlastní.

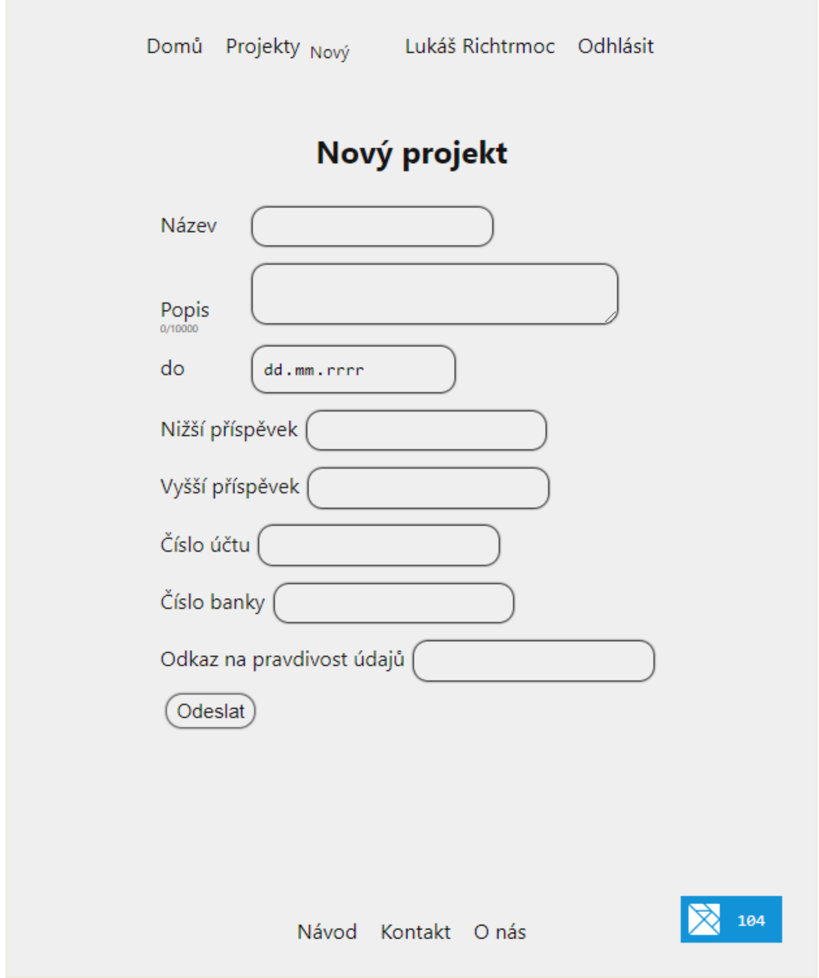
Obrázek 13 - Mobilní zobrazení daru. Zdroj: Vlastní.



Obrázek 14 - Mobilní zobrazení formuláře. Zdroj: Vlastní.



Obrázek 15 - Zobrazení notifikace na mobilu. Zdroj: Vlastní.



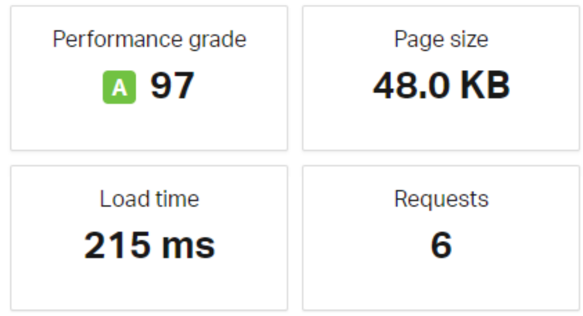
Obrázek 16 - Formulář nového projektu: Zdroj: Vlastní.

5.6. Testování

K ověření vlastností webové aplikace byly použity veřejně dostupné testovací nástroje Pingdom Tools a Google LightHouse. Aplikace byla vystavena na hosting umístěný v Praze za veřejnou doménou s SSL šifrováním.

Pingdom Tools

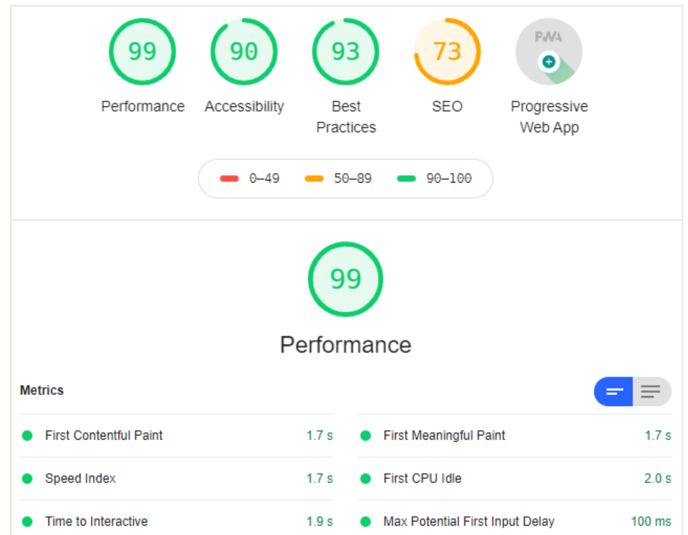
Test rychlosti splnil nefunkční požadavek na načtení stránky do 250 ms. Velikost přenesených dat se při zakódování do formátu gzip ještě zmenšila. Bylo provedeno 10 testů z Frankfurtu při průměrné době 237 ms se směrodatnou odchylkou 27 ms, s minimem 215 ms a maximem 288 ms.



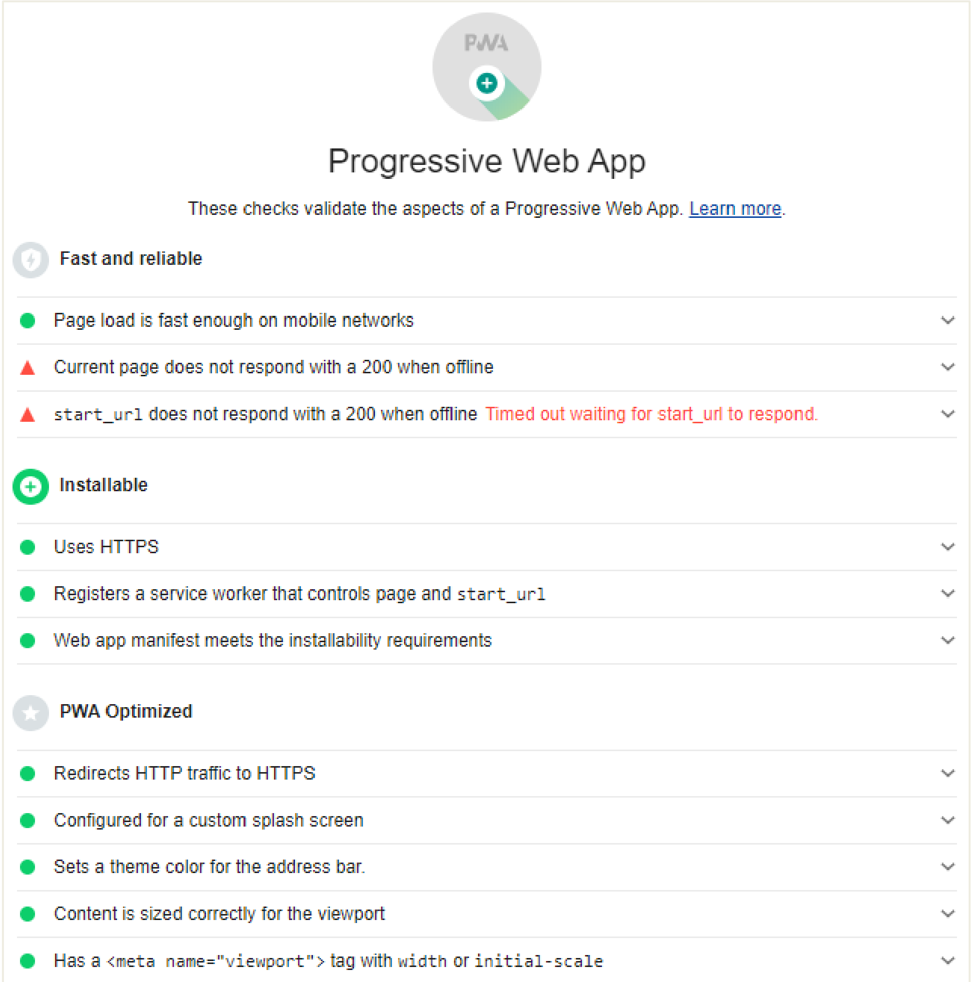
Obrázek 17 - Výsledek testu z Pingdom Tools. Zdroj: Vlastní.

Google LightHouse

Kromě rychlosti byla testována i celková výkonnost aplikace pomocí auditního systému Google Lighthouse. Ve většině testů aplikace uspěla, sražena měla SEO skóre kvůli atributům *noindex, nofollow*, jež zamítají indexaci vyhledávácím robotům. V hodnocení PWA způsobilosti pak chyběla především možnost offline práce.



Obrázek 18 - Výsledek testu z Google LightHouse. Zdroj: Vlastní.



Obrázek 19 - Protokol úrovně implementace PWA. Zdroj: Vlastní.

Akceptační test

Aplikace byla zpřístupněna dvěma subjektům a testována metodou *On-Site Observation*. Kromě odsouhlasení dostatečné implementace požadavků bylo na základě zpětné vazby rozšířeno prostředí o UX prvky, jako počet zapsaných znaků v popisku projektu, nebo zvětšování značky *textarea*.

Automatizované testy

V rámci práce bohužel již nebyly provedeny unit, fuzz či penetrační testy. Elm podporuje unit testování v balíčku *elm-explorations/test* a pro testování HTML značek nabízí modul *Test.Html.Query*.

6. Diskuze

Při prvotním seznámení s jazykem Elm dochází často k reakci, že je moc „akademický“. Proto byl vytvořen dotazník mající za cíl zjistit poznatky z komerčního vývoje a postavit toto tvrzení do kontrastu se zkušenostmi z reálného prostředí.

Kromě dotazníku kapitola shrnuje obecné výhody jazyka, možné problémy při vývoji a doporučení pro začínající programátory. Na závěr autor práce předkládá zamyšlení, které by rád v budoucnu kolem platformy Elm prozkoumal.

6.1. Dotazník

Dotazník byl uvolněn na Slacku (viz kapitola 4.2) ve dnech 9. 1. a 18. 4. 2020. Zároveň byl odeslán vytipovaným firmám hledající Elm vývojáře. Dorazilo celkem 13 odpovědí od zaměstnanců společností sídlících v Dánsku, Francii, Itálii, Velké Británii, Spojených státech amerických a Švédsku. Šlo o volitelné, uzavřené (viz příloha 9.2) a otevřené (viz příloha 9.3) otázky. Z důvodu malého počtu respondentů jsou výsledky spíše úvahou, nežli statistickým hodnocením.

Z dat je patrné, že v Elmu pracují převážně jednotlivci, velké týmy (10 a více vývojářů) jsou spíše výjimkou (2x). Na stupnici od 1 (nespokojen) do 10 (spokojen) hodnotili respondenti Elm pr ůměrnou známkou 9,04 s odchylkou 1, při použití NPS (Net Promoter Score) by to znamenalo 70% promotérů.

V délce programování i nasazení na produkci se Elmovské programy mezi sebou nijak významně neliší a jejich rozložení je normální. Pozice ve firmě je většinou stabilní. Některé projekty již byly ukončeny či převedeny do Reactu, mezi další obchodní problémy patří odchod expertů a odmítavý postoj kolegů. Třicet procent respondentů uvedlo jako obtížné adaptovat se na nový jazyk a MVU architekturu.

Vývojářům v jazyce chybí řešení pro server-side a též větší genericita v podobě zavedení Type classes nebo reflexí. V syntaxi naopak vyzdvihují výhody funkcionálního programování: ADT, pattern matching nebo že vše je výraz, Jako nejoblíbenější balíček dominuje *elm-ui* (7x). Oblíbeným bundlerem je Webpack (4x).

6.2. Výhody

K první praktické práci v Elmu se autor dostal o rok dříve, než začal psát diplomovou práci. V této podkapitole se nachází shrnutí výhod, na které při vývoji narazil a jež ze svého pohledu považuje za významné.

Syntaxe

Syntaxe jazyka je díky *mezerování* velmi úsporná, což pomáhá při rychlosti vývoje. Není nutné vše uzávorkovat, pro změny směru toku aplikace slouží částečná aplikace (*pipe*) a kompozice funkcí (*arrow*). Jejich použití je velmi příjemné.

Anotace

Limitně znemožňuje psát špatný kód, neošetřovat výjimky, vpouštět špatné typy do funkcí, přetypovávat. Funguje jako popis funkce, kterou tak není nutné složitě pojmenovávat. Napomáhá IDE hledat chyby ještě dříve, než probublají do funkce *main*, kde mají menší výpovědní hodnotu.

Referenční transparentnost

Transparentnost zaručuje jasný vstup a výstup z funkce, aniž by mohlo dojít k nějaké „zvlášnosti" na pozadí. Též zanáší jasný kontext na těch pár řádků kódu působnosti funkce, kde je tak možné úsporně pojmenovávat funkce *go* a proměnné *x*.

Funktory

Maybe a *Result* velmi pomáhají při asynchronním zpracování. Není nutné bádat, v jakém ifu je běh programu, protože se ví, jaké funkce se aplikují, pokud je hodnota odpovídajícím algebraickým datovým typem. Zároveň uživatele nutí k ošetření výjimek, které by ho nenapadly či by je přešel. Samozřejmě je lze též ignorovat, ale (*withDefault ""*) je do očí bijící.

Vícenásobný pattern matching

Využití pattern matchingu jako *ad hoc* kartézského součinu více typů je skvělým urychlením, pokud je zájem pouze o konkrétní variantu. Například při *(Bool, Maybe Int)* je možné aplikovat funkci pouze v případě *(True, Just 1)*, ve zbytku (_) provést cokoliv jiného.

Silná statická typovost a ADT

Přetypování, které se provádí v JavaScriptu, není možné. Též není možná změna typu „pod rukou", vždy se ví, s čím se pracuje (opět referenční transparentnost). Při využívání vlastních algebraických datových typů je hodnota „chráněna" proti sémantické chybě.

Parser a Decoder

Přeměna stringů na typy a strukturovaná data. Velmi pokročilé, efektivně odstíní kontext od chaosu. Krása spočívá v tom, že se převezmou do funkce pouze hodnoty uvedené dekóderem – není tak možné zanést model něčím, co by „vyjedlo“ paměť.

Funkce první třídy a kompozice funkcí

Další konstrukt, který výrazně zpřehledňuje program. Předávání funkce jako hodnoty do argumentu v kombinaci s kompozicí funkce drží změny tam, kde byly vyvolané – například u kliknutí (viz *Higher-Order Delivery*).

Zřetězené mapování

Využití mapovacích funkcí u datových struktur *Dict*, *List*, *Array*, *Set*, *Tuple* a jejich postupné řetězení operátorem částečné aplikace zrychluje psaní kódu.

Rozumná správa side-efektů

Abstrakce kódu od systému a uživatele pomocí zpráv je velmi přínosná. Zapříčiňuje, aby aplikace pracovala pouze s daty, která jsou v modelu, nikoliv například v HTML značkách.

Výpočetní náročnost

Pevně dané datové struktury spolu s určováním *lazy* funkcí a ovládáním virtuálního DOMu transparentně determinují, jaká bude výpočetní náročnost. Spolu s mocnou kompilací se produkují rychlé aplikace.

6.3. Problémy

Podkapitola úvadí, s jakými nejčastějšími problémy se vývojář Elmu může setkat, od rozšiřování po techniky zrychlení běhu aplikace.

Mohutnost

Pro začínající programátory je smysluplné rozšiřování kódu jedním z největších oříšků. Jde o to, jakou techniku delegace zpráv zvolit. Každá aplikace může mít jedinečnou kombinaci řešení a ne vždy je na první pohled patrné, jakým směrem se vydat.

Refaktorizace

Při dobrém počátečním rozvržení je refaktorizace snadná. Problém nastává, kdy se mění architektura předávání zpráv, či model, který místo ADT používá primitivní datové struktury.

Rozpadávání modulů

Stromově rozpadávající model je povinností. Na jednu stranu nutí vytvářet ze složitého jednoduché, avšak při použití techniky *Shared State* nutí rozdrobit *Modul* i na soubory *Modul.Msg* a *Modul.Model*, jinak by při použití v *Modul* vznikl cyklický import.

Duplicita modelu

Nekonzistence stavu aplikace může vznikat při použití stejného typu dat v nadmodulu a podmodulu, které užívají vlastní modely. Příklad: V podmodulu B je pracováno s entitami, které se načtou pouze jednou. Nadmodul A pracuje s těmito entitami taktéž, přičemž je občerstvuje jednou za čas (jakýsi *refresh*). Zároveň drží ve svém modelu model podmodulu B. Množina entit v A může i nemusí být stejná jako v B; to nelze říci.

Varianty řešení:

a) Přenechat vše v A, který občerstvuje a předává vybrané entity do B. A je velký nabobtnalý číselník s veškerou logikou, kterou musí předávat do B.
b) A vždy mění data v B, avšak B též obsluhuje sám sebe. B však může vytvářet dotazy, na které již má odpověď z A.
c) B se stará pouze o svoji množinu nezávisle na A, modul A odchytává funkce B a případně aktualizuje též svůj model.

Boilerplates

Opakování business logiky na více místech pro trochu jiné podmínky. Například místo *Listu* je k dispozici pouze *Array*, přičemž se píše stejná funkce, akorát pro jinou datovou strukturou. Toto je možné vyřešit buď dobrým rozmyšlením vstupů a výstupů funkcí, nebo přidáním vlastního typu, který zavede něco jako „interface“. Je třeba dbát na to, že zde není polymorfismus, který by to vyřešil za vývojáře. Je možné ho implementovat ve stylu fce (rozšířená) a fce_ (základní), ale to je špatné řešení. Generické typy jsou lepší, ale dají dost práce na vymýšlení. V podobném duchu je také *Maybe*, kdy při špatně volbě ADT je neustále nutné řešit stav, který může nastat minimálně.

Inklinace ke konkretizaci

Velice často se stává, že kód je napsán tak, aby co nejrychleji fungoval na konkrétní řešení v konkrétním modulu. Při následné refaktorizaci se pak „vynáší“ funkce do univerzálnější struktury, je nutné zavést zprávy z komponenty do použitých nadmodulů aj.

Povinné větvení

If nelze zadat jako jednu větev, je vždy nutné napsat i druhou. Tím vzniká trochu zbytečná režie, například vracením prázdných listů, příkazů apod. Ovšem je to nutnost, vzhledem k vyhodnocování výrazů jako funkcí.

Explicitní dědičnost

V Elmu dědičnost v pravém slova smyslu není, ale lze ji namodelovat vkládáním jednoho modulu do druhého. Avšak pokud není závislost zachycena v modelu datovou strukturou, tak se o jejím importování neví. Pak je třeba spoléhat na pojmenování (např. *HomePage*) nebo adresářovou strukturu (*Page.Home*).

Chybějící interface

Zavázání se implementovat funkci. Například všechny moduly v nadmodulu *Page* by měly implementovat funkci *view*, nyní je to pouze nezávazný konsenzus. Při rozšíření o typově vyšší polymorfismus (ve stylu *type classes*) by interfaci smysl dávaly.

Mapování ADT

Při použití slovníku není možné použít jako klíč ADT – to je přirozené, typ může být velmi složitý. Je tedy třeba jako klíč použít primitivní hodnotu, což ve výsledku znamená napsat dekodér a enkodér pro každý ADT.

Primitivní klíče slovníků

Se slovníkem souvisí též problém, kdy je reprezentací číselníku, ve kterém jako klíč musí být primitivní hodnota. Je to Id? Je to URI? Je to cizí klíč? Též se při slovníku přichází o pořadí.

HTML odpovědi

Kvůli obavám z XSS útoků není možné od verze 0.19 přebírat kusy HTML značek z HTTP odpovědí a vkládat je do struktury, což dříve přes funkci *property* a argument *innerHTML* šlo. Nyní jsou dvě možné cesty:

a) Použít porty pro přidání značek (možná kolize s virtuálním DOMem, možné provedení XSS).
b) Použít knihovny pro parsování String -> List Html (https://package.elm-lang.org/packages/hecrj/html-parser/latest/), nefungují však SVG a JS skripty.

Práce s bytes přes HTTP

Nahrávání a stahování souborů je nutné realizovat přes bytes (v JS blob), jež nefungují na všech platformách správně (iOS).

Asymptotická složitost řetězení

Je třeba mít na paměti, že stále jde o kompilaci do imperativního jazyku. Převody mezi strukturovanými datovými typy (*Array.toList*, *Dict.fromList*) mají linerání složitost, stejně jako použití funkcí manipulujících se všemi daty, například *map* či *filter*. Při použití řetězení tedy dochází k tolika cyklům, kolikrát byla funkce zavolána. U malých datových souborů to nevadí, u velkých je řešením provést všechny manipulace v aplikované funkci a projít data pouze jednou, nebo provést postupnou kompozici funkcí, nebo využít předpřipravené funkce *filterMap*, *concatMap* a jiné.

Akumulace

Je trochu problém ze strukturovaného datového typu udělat samostatnou proměnnou. Není možné použít imperativní řešení pomocné proměnné uložené někde stranou, do které by se při průchodu „kolekcí" zapisovalo, protože by šlo o ovlivňování struktury mimo vlastní kontext (*side-effect*) a byla by porušena referenční transparentnost. Řešením je rekurze s akumulátorem, do kterého se postupně mapují odebírané položky ze strukturovaného datového typu. Není nutné vždy psát rekurzi odspodu, většina modulů implementuje funkce se správnou ukončovací podmínkou, nazývané *foldl* (akumulace zleva) a *foldr* (akumulace zprava).

Monitoring a sběh asynchronních požadavků

Nutné zavedené mechanismů, které budou hlídat stav provádění vedlejších efekt (např. loading na pozadí) a provádět opětovné pokusy (napří. znovu odeslání dotazu) Velmi pomáhá použití *Maybe*, které v modelu rozliší, zda jde o prázdnou strukturu nebo nenačtenou položku. Sběhem je myšleno čekání na dokončení více vedlejších efektů, které je nutné mít k dispozici najednou. Řešením je po každém úspěšném dokončení vedlejšího efektu zavolat funkci, která se má teoreticky vykonat po sběhu – v ní pak teprve zjišťovat (například opět pomocí *Maybe*, ale může to být i *Bool*), zda již všechna volání doběhla. Teprve poslední doběh zaručí, že kód se za touto podmínkou vykoná.

Zanesení starými daty

Je velmi důležité, aby se přes vedlejší efekty nikdy nepřenášela data z modelu, pouze zprávy a identifikátory. Při odesílání strukturovaných dat (či dokonce celého modelu) do vedlejšího efektu se může stát, že se zprávy předají, vedlejší efekt proběhne, ale model je stále stejný. Pokud je nutné pracovat s daty z modelu, tak je správné přenést pouze identifikátor a vylovit entitu z aktuálního modelu.

Vzdálená a lokální databáze

Tento neduh vyvstane téměř při každém projektu. Je škoda, že není k dispozici modul, který by automaticky řešil přístup k datům jak vzdáleným, tak lokálním.

Aplikace je pomalá

Stává se, že stránka reaguje pomalu, ač je asymptotická složitost nízká. Pravděpodobně se jedná o velké množství zpráv, které prochází aplikací (například *scroll*) a pokaždé mění model. V takovém případě je nutné odchytit opravdovou změnu v modelu, která je nutná k překreslení HTML a zabalit ji do funkce *lazy* z modulu *Html.Lazy*. Pak se vykreslující funkce zavolá pouze v případě změny.

6.4. Doporučení

Autor práce doporučuje vytvářet co nejvíce vlastních datových typů, pomůže to jednoznačnosti a režii nutné k refaktorizaci. Jako model použít záznam, jelikož může obsahovat heterogenní data a má rychlý přístup přes jmenné indexy. Adresářovou strukturu vytvářet jako vnořující se specializace stromové struktury – od složitějšího kořene

k jednodušším listům. V začátcích využívat typovou inferenci a referenční transparentnost na místech, která jsou bez anotace jasná. Předem si vytyčit, jak v projektu přezdívat modulům místo celé notace. Držet velikost funkcí kolem 5-20 řádků. Vytvářet zástupné typy (jako *EntityId*) i přes to, že jde o pouhé aliasy primitivních hodnot. Vyhnout se použití onClick a raději měnit URL adresu. Funkce k vykreslení (*Html msg*) by měly vracet vždy list hodnot, buď úspěšně naplněných, nebo prázdných. Entity držet jako slovník, kde id je primárním klíčem. Vytvořit si modulové rutiny, jako *Kind*, *Model*, *Msg* a podobně. Používat krátké názvy proměnných a výstižná jména funkcí s postfixy jako *old* a *new*. Zároveň mít stejné prefixy pro funkce, které rozšiřují původní funkci (*view*, *viewDetail*, …). Šetřit v modelu s *Maybe*, pokud je hodnota téměř jistá (například zjištění datumu z prohlížeče). Vytvořit si funktory a monády pro *Model*. Jako poslední položku funkce přijímat data, která budou pravděpodobně měnná. V modulu vytvořit kromě modelu ještě parametrický ADT (*Props a*), který definuje veškeré nutné závislosti pro vykreslení řádkovým polymorfismem (*{ a | time: Date, ... }*). Pak se nemusí všude předávat celá *Session*, ale pouze část, která je nezbytně nutné – tím se modul odstřihuje od *Session* a je možné do něj vkládat zprávy či typy z modulu, aniž by docházelo k cyklickým importům. Nepřesouvat logiku aplikace na stranu portů (funkce port by měla být definována svým názvem, například *getHash*), v komunikaci preferovat pouze primitivní identifikátory.

6.5. Zamyšlení

Při práci s Elmem vyvstávají jistá dilemata. Jedním z nich je otázka, zda se držet typové inference a neanotovat, či být v anotaci striktní. Díky anotaci se generují lepší chybové hlášky, avšak je nutné všude importovat konkrétní typ anotace, čímž se zanesou cyklické závislosti do importů. Pro dodržení stromové struktury je pak nutné buď rozdrobit ADT do vlastních souborů, nebo opustit návrhový vzor *Shared State*, ve kterém se *Session* importuje v nezměněné struktuře do všech modulů.

V pokročilejších projektech chybí větší genericita jazyka, například při dekódování HTTP odpovědí či volání funkcí podle hodnoty ADT. Toto by se mohlo změnit, pokud by se Elm rozšířil o typově vyšší polymorfismus. Též by mohl slovník akceptovat neparametrický ADT jako klíč a kompilátor snížit asymptotickou složitost při práci se strukturovanými datovými typy, pokud by je převedl do jedné velké smyčky.

Závěrem je nutné podotknout, že Elmu schází nějaký „zvučný“ framework nebo profesionálnější startovací balík, který by vývojáře „hodil“ do vody s předpřipravenými typy, funkcemi a vzory, kde by mu ukázal, jak lze zpříjemnit (nejen) frontendový život.

7. Závěr

Teoretická část začala úvodem do historie webového prostředí a seznámením s principy navigace a zobrazení. Byly prozkoumány různé druhy vývoje a technologií používaných na webu. Následný výčet programovacích paradigmat přenesl pozornost ke konceptu funkcionálního programování, ve kterém byla zobrazena provázanost s lambda kalkulem. Výčet vlastností a technik funkcionálního programování uzavřel první část.

Na začátku praktické části byl proveden vhled do Elmovského ekosystému. Poté proběhlo seznámení se základními dovednostmi jazyka a představení technik pro správu stavu aplikace. V samotné implementaci byl nejdříve představen scénář s požadavky, poté diagram tříd a návrh řešení provázání backendu s frontendem. K delegování zpráv byla vybrána experimentální metoda Higher-Order Delivery, která se ukázala v praxi jako účinná. Výsledná SPA byla podrobena úspěšnému testování.

Nad celkovým výsledkem diplomové práce proběhla diskuze, ve které byly prvotně prezentovány výsledky dotazníku, jež zjišťovaly preference a zkušenosti vývojářů Elmu z celého světa. Dále byly shrnuty výhody a nejčastější problémy vývoje v Elmu s doporučením, jak se s nimi vypořádat.

Práce je dostupná ve vzdáleném repozitáři (https://bitbucket.org/bohemak2/dont-shut-the-shop-down/).

8. Seznam zdrojů

8.1. Tištěné zdroje

ALEXANDER, A. Functional Programming, Simplified. USA: CreateSpace Independent Publishing Platform, 2017. 780s. ISBN: 978-1-9797-8878-6.

BŮNA, M. Technologie Elm a její použití pro front-end webové aplikace. Praha: Vysoká škola ekonomická v Praze, 2018. Diplomová práce. Vedoucí práce Jarmila Pavlíčková.

BURGSTAHLER, S. Opening doors or slamming them shut? Online learning practices and students with disabilities. V: *Social Inclusion*. USA: Cogitatio Press, 2015. ISSN: 2183-2803.

ČÍŽEK, K. Reaktivní programování. Hradec Králové: Univerzita Hradec Králové, 2019. Bakalářská práce. Vedoucí práce Tomáš Kozel.

DOMINGUE, J. FENSEL, D. HENDLER, J. Introduction to the Semantic Web Technologies. V: *Handbook of Semantic Web Technologies*. Německo: Springer, 2011. ISBN: 978-3-540-92912-3.

ERIKSSON, N., ÄRLERYD, CH. Elmulating JavaScript. Linköping: Linköping University 2016. Diplomová práce. Vedoucí práce Anders Fröberg.

FAIRBANK, J. Programming Elm. USA: The Pragmatic Programmers, 2019. 308s. ISBN: 978-1-68050-285-5.

FELDMAN, R. Elm in Action. USA: Manning Publications, 2020. 375s. ISBN 978-1-6172-9404-4.

FRANCESCONI, E. On the Future of Legal Publishing Services in the Semantic Web. V *Future Internet*: Švýcarsko, 2018. 38 (10-48). ISSN 1999-5903.

FROST, B. Atomic Design. USA: Brad Frost Web, 2016. 193s. ISBN: 978-0-9982-9660-9.

HARRISON, J. Introduction to functional programming. Anglie: Cambridge University, 1997. 160s.

HOLAPPA, A. Web Frontend Development with Elm. Oulu: Oulu University of Applied Sciences, 2018. Bakalářská práce. Vedoucí práce Veikko Tapaninen.

IMSIROVIC, A. Elm Web Development: An introductory guide to building functional web apps using Elm. Anglie: Packt Publishing, 2018. 369s. ISBN: 978-1-7882-9905-3.

JIROUDEK, V. Tvorba webových aplikací jazykem ELM. České Budějovice: Jihočeská univerzita v Českých Budějovicích, 2018. Bakalářská práce. Vedoucí práce Petr Pexa.

KADLECAJ, J. Online Lambda Calculus Evaluator. Brno: Masarykova univerzita, 2018. Bakalářská práce. Vedoucí práce Jan Obdržálek.

KHOT, A. MISHRA, R. Learning Functional Data Structures and Algorithms. USA: The Pragmatic Programmers, 2017. 514s. ISBN: 978-1-7858-8588-4.

LODER, W. Web Applications with Elm: Functional Programming for the Web. USA: Apress, 2018. 208s. ISBN 978-1-4842-2609-4.

LUXEMBURK, J. Functional Programming for Web Frontend. Praha: České vysoké učení technické v Praze, 2017. Bakalářská práce. Vedoucí práce Robert Pergl.

MICHAELSON, G. An Introduction to Functional Programming Through Lambda Calculus. USA: Courier Corporation, 2011. 320s. ISBN: 978-0-4864-7883-8.

MLADÝ, L. Funkcionální jazyky kompilované do JavaScriptu v praxi. Praha: Vysoká škola ekonomická v Praze, 2017. Diplomová práce. Vedoucí práce Tomáš Bruckner.

PRAHBU, D. Application of Web 2.0 and Web 3.0: An overview. V *International journal of research in library science*: Indie, 2016. 8 (54-62). ISSN 2455-104X.

SLIFKA, J. Data Stewardship Portal: Client-side Web Frontend. Praha: České vysoké učení technické v Praze, 2018. Diplomová práce. Vedoucí práce Robert Pergl.

WÄLTER, J. Functional Programming for Web and Mobile–A Review of the Current State of the Art. Švýcarsko: University of Applied Sciences Rapperswil, 2019. Vedoucí práce Farhad Mechta.

YALLOP J., WHITE L. Lightweight Higher-Kinded Polymorphism. V *Functional and Logic Programming*. Švýcarsko: Springer International Publishing, 2014. 367s. ISBN 978-3-319-07150-3.

8.2. Internetové zdroje

BEAUFORT, F. Get started with GPU Compute on the Web [online]. Srpen 2019 [cit. 2020-02-05]. Dostupné z <https://developers.google.com/web/updates/2019/08/get-started-with-gpu-compute-on-the-web>.

CHAVES, R. Child-Parent Communication in Elm: OutMsg vs Translator vs NoMap Patterns [online]. Červen 2017 [cit. 2020-04-18]. Dostupné z <https://medium.com/@_rchaves_/child-parent-communication-in-elm-outmsg-vs-translator-vs-nomap-patterns-f51b2a25ecb1>.

CIMPANU, C. Half of the websites using WebAssembly use it for malicious purposes [online]. Leden 2020 [cit. 2020-02-05]. Dostupné z <https://www.zdnet.com/article/half-of-the-websites-using-webassembly-use-it-for-malicious-purposes/>.

CLEMENT, J. Global digital population as of October 2019 [online]. Listopad 2019 [cit. 2020-01-30]. Dostupné z <https://www.statista.com/statistics/617136/digital-population-worldwide/>.

COHEN, M. Web Storage Overview [online]. Únor 2019 [cit. 2020-03-22]. Dostupné z <https://developers.google.com/web/fundamentals/instant-and-offline/web-storage>.

CZAPLICKI, E. Elm: Concurrent FRP for Functional GUIs [online]. Březen 2012 [cit. 2020-02-05]. Dostupné z <https://elm-lang.org/assets/papers/concurrent-frp.pdf>.

CZAPLICKI, E. Elm: A Farewell to FRP [online]. Květen 2016 [cit. 2020-02-26]. Dostupné z < https://elm-lang.org/news/farewell-to-frp>.

CZAPLICKI, E. Small Assets without the Headache [online]. Srpen 2018 [cit. 2020-02-28]. Dostupné z <https://elm-lang.org/news/small-assets-without-the-headache>.

CZAPLICKI, E. The reasoning behind removing operators [online]. Srpen 2018 [cit. 2020-03-05]. Dostupné z <https://gist.github.com/evancz/769bba8abb9ddc3bf81d69fa80cc76b1>.

CZAPLICKI, E. The Syntax Cliff [online]. Říjen 2019 [cit. 2020-02-28]. Dostupné z <https://elm-lang.org/news/the-syntax-cliff>.

CZAPLICKI, E. The Perfect Bug Report [online]. Listopad 2016 [cit. 2020-02-29]. Dostupné z <https://elm-lang.org/news/the-perfect-bug-report>.

EKLUND, A. Brainstorm Technique: Lotus Blossom [online]. Červenec 2013 [cit. 2020-04-26]. Dostupné z <https://andyeklund.com/brainstorm-technique-lotus-blossom/>.

GAUNT, M. Service Workers: an Introduction [online]. Srpen 2019 [cit. 2020-02-09]. Dostupné z <https://developers.google.com/web/fundamentals/primers/service-workers>.

HANHINEN, O. Elm Shared State example [online]. Srpen 2017 [cit. 2020-02-09]. Dostupné z <https://github.com/ohanhi/elm-shared-state/>.

JYLÄNKI, J. WebAssembly for Native Games on the Web [online]. Červen 2017 [cit. 2020-04-20]. Dostupné z <https://hacks.mozilla.org/2017/07/webassembly-for-native-games-on-the-web/>.

KEARS, D. The Redux Pattern As a First-Class Citizen [online]. Červen 2019 [cit. 2020-01-29]. Dostupné z <https://medium.com/elm-for-redux-devs/the-redux-pattern-as-a-first-class-citizen-b1d7fa1e4438>.

LEE, T. MASINTER, L. FIELDING, R. Uniform Resource Identifier (URI): Generic Syntax [online]. Leden 2005 [cit. 2020-03-19]. Dostupné z <https://tools.ietf.org/html/rfc3986>.

LENTZNER, M. A Service Pattern [online]. Září 2019 [cit. 2020-04-18]. Dostupné z <https://github.com/mzero/elm-service-pattern>.

MCLACHLAN, P. Introducing a Trusted Web Activity for Android [online]. Únor 2019 [cit. 2020-02-09]. Dostupné z <https://blog.chromium.org/2019/02/introducing-trusted-web-activity-for.html>.

MACDONALD, M. A Simple Introduction to Web Workers in JavaScript [online]. Červenec 2019 [cit. 2020-02-09]. Dostupné z <https://medium.com/young-coder/a-simple-introduction-to-web-workers-in-javascript-b3504f9d9d1c>.

MUGNAINI, L. Functors, Applicatives, And Monads In Pictures (In Elm) [online]. Červenec 2018 [cit. 2020-01-25]. Dostupné z <https://medium.com/@l.mugnaini/functors-applicatives-and-monads-in-pictures-784c2b5786f7>.

NYMAN, R. The concepts of WebGL [online]. Duben 2013 [cit. 2020-02-05]. Dostupné z <https://hacks.mozilla.org/2013/04/the-concepts-of-webgl/>.

POUDEL, P. Fetching Data Using GET [online]. 2018 [cit. 2020-04-22]. Dostupné z <https://elmprogramming.com/fetching-data-using-get.html >.

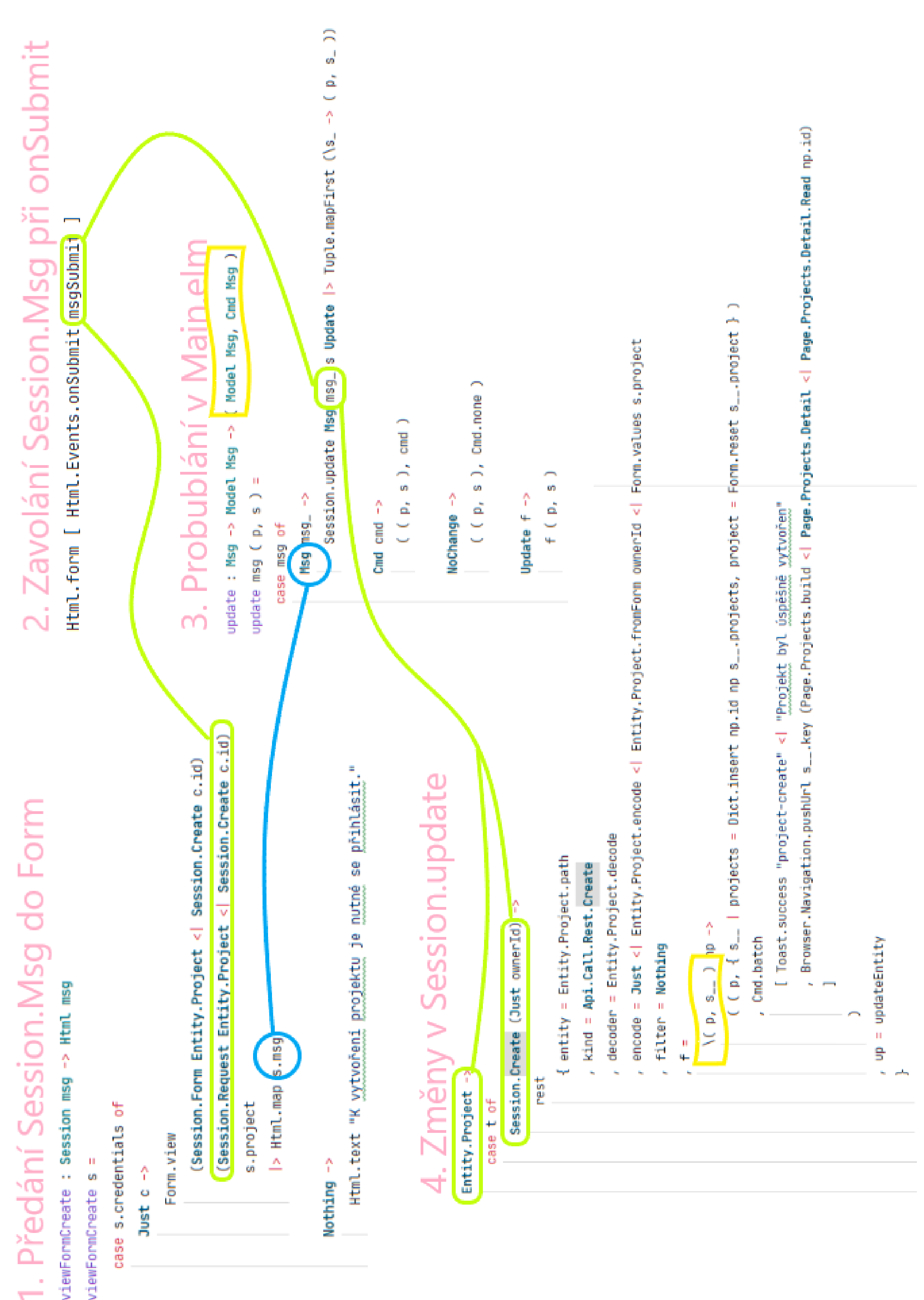
RUSSELL, A. Progressive Web Apps: Escaping Tabs Without Losing Our Soul [online]. Červen, 2015 [cit. 2020-02-09]. Dostupné z <https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/>.

TENSOR. Using the Elm Architecture or the MVU pattern with Dartea inside of Dart's Flutter Framework [online]. Květen 2018 [cit. 2020-04-22]. Dostupné z <https://steemit.com/utopian-io/@tensor/using-the-elm-architecture-or-the-mvu-pattern-with-dartea-inside-of-dart-s-flutter-framework>.

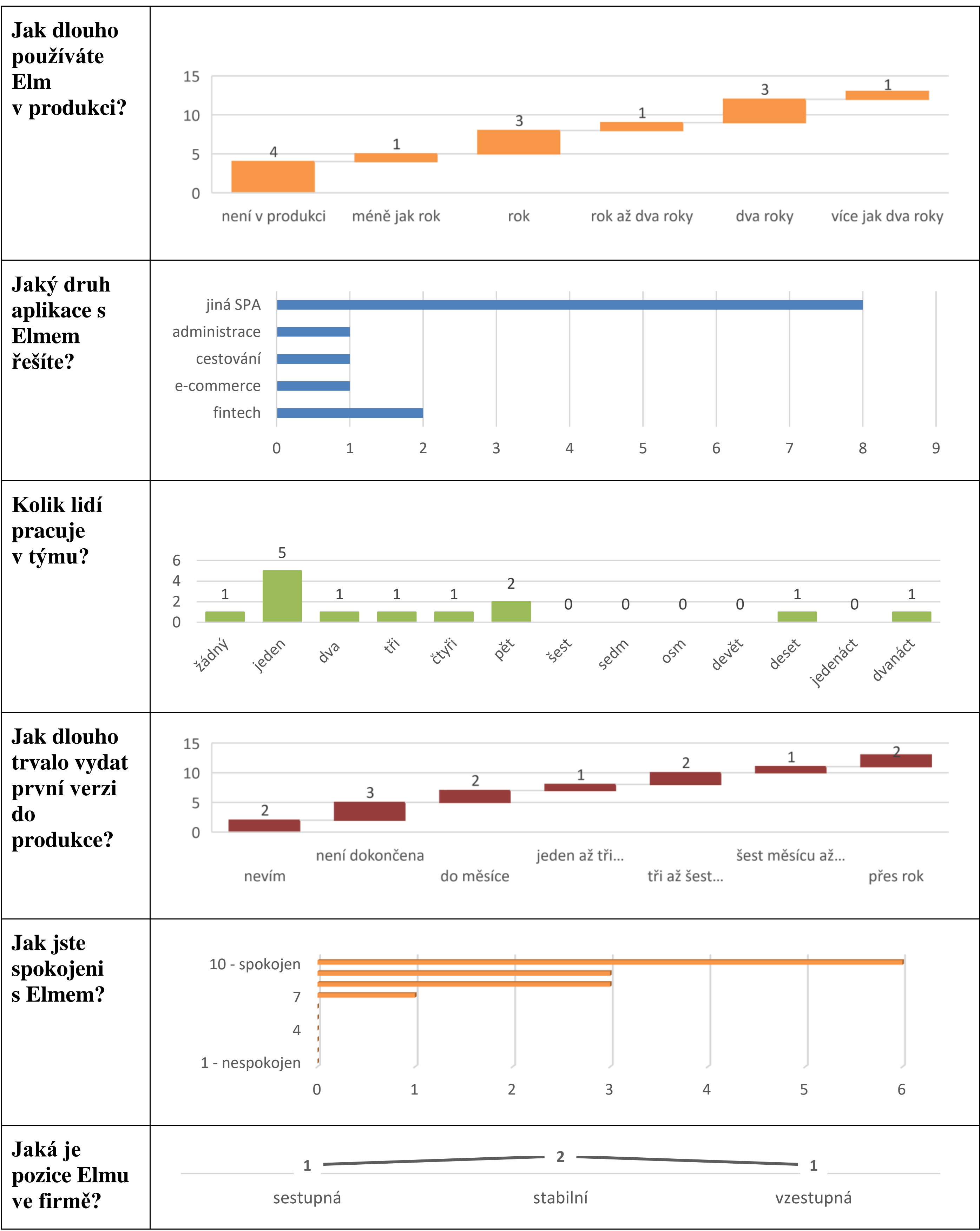
YACOUB, H. WebUSB is dead [online]. Únor, 2020 [cit. 2020-02-09]. Dostupné z <http://www.hbyconsultancy.com/blog/webusb-is-dead.html>.

9. Přílohy

9.1. Příloha č. 1 – Ukázkový případ Higher-Order Delivery



9.2. Příloha č. 2 – Dotazník: Uzavřené otázky



9.3. Příloha č. 3 – Dotazník: Otevřené otázky

Jaká byla největší obtížnost, které jste museli čelit s Elmem?	• Adaptace jazyka (3x) • Adaptace na centrální model • Programování server-side • Transakce • Omezený debugger (ukazující <internal>) • Žádné privátní balíčky • Škálování velkých sobourů • Zpracování více eventů najednou (click, drag, double click…)
Jaký je váš oblíbený balíček?	• elm-ui (7x) • elm-css • elm-geometry
Jaká je vaše oblíbená věc v syntaxi Elmu?	• Vše je výraz • Pipes • Čistá syntaxe • ADT a Pattern matching • Anotace
Co postrádáte v Elmu?	• Type Classes (2x) • Dobrou knihovnu pro práci s datumy, časem a zónami • Nástroje pro IDE • Programování server-side • Reflexe pro enkódování a dekódování
Jaký používáte bundler?	• Webpack (4x) • Žádný (2x) • Parcel
Čelíte nějakým ‘business‘ potížím?	• Elm byl nahrazen Reactem • Vývoj trval příliš dlouho • Evangelisti odešli z firmy • Ostatní se brání výhodám Elmu

Zadání diplomové práce

Autor: **Bc. Lukáš Richtrmoc**

Studium: I1800767

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Vývoj webových aplikací v jazyce Elm**

Název diplomové práce AJ: Web Application Development Using Elm Language

Cíl, metody, literatura, předpoklady:

Cíl: Popsat jazyk Elm a metodiky vývoje na webu v kontextu funkcionálního prográmování a vytvořit ukázkovou aplikaci pomocí tohoto jazyka.

Osnova:

1. Úvod
2. Webové prostředí
3. Koncept funkcionálního programování
4. Programovací jazyk Elm
5. Webová aplikace napsaná v jazyku Elm
6. Závěr

--

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Datum zadání závěrečné práce: 14.1.2018