FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Optilynx – stavový kontejner
Student:	Matúš Varholík
Vedoucí:	Ing. Filip Glazar
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce je přidat stavovost do klientské části systému pro oční optiky. Původní aplikace vznikla v rámci diplomové práce Bc. Jaroslava Hracha. Základem práce je analyzovat stávající stav aplikace a na základě analýzy zvolit a integrovat vhodný framework pro správu a centralizaci stavů v kompletní aplikaci. Zároveň by během implementace měl být kladen důraz na potřebné opravy, či refaktorizace části softwaru, které budou během analýzy shledány jako nefunkční, či nevhodné.

Postupujte dle následujících kroků:

1) Proveďte důkladnou analýzu stávajících zdrojových kódů aplikace
2) Identifikujte nedostatky aplikaci
3) Zvolte vhodný framework pro správu stavu
4) Integrujte framework
5) Otestujte vaše řešení a zhodnoťte vliv vašich úprav na systém
6) Připravte řešení pro nasazení do produkce a nasaďte

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 7. ledna 2022 v Praze.

Bakalárska práca

OPTILYNX – STAVOVÝ KONTEJNER

Matúš Varholík

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedúci: Ing. Filip Glazar
11. mája 2023

Odkaz na túto prácu: Varholík Matúš. *Optilynx – stavový kontejner*. Bakalárska práca. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Zoznam obrázkov

Zoznam výpisov kódu

Chcel by som sa poďakovať predovšetkým Ing. Filipovi Glazarovi za vedenie tejto práce, jeho užitočné rady a ochotný prístup. Moja vďaka tiež patrí kolegom, ktorí prispeli k jej tvorbe svojimi radami, spätnou väzbou či testovaním. Nakoniec by som chcel vyjadriť vďaku svojej rodine a priateľom za ich bezhraničnú podporu počas celého štúdia.

Vyhlásenie

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dňa 11. mája 2023 .....

Abstrakt

Táto bakalárska práca sa zaoberá integráciou vhodného riešenia pre správu stavu klientskej časti webovej aplikácie. Analyzuje súčasný stav zdrojového kódu, približuje jeho nedostatky a popisuje populárne riešenia tohto problému. Na základe analýzy je zvolená vhodná technológia pre správu stavu a vypracovaný podrobný návrh konkrétneho riešenia, ktoré je následne implementované. Na záver je uskutočnené užívateľské testovanie s cieľom zhodnotiť vplyv daných úprav na systém a vykonaná príprava na nasadenie.

Klíčová slova stavový kontajner, klientska časť, správa stavu, webová aplikácia, Angular, Redux, NgRx

Abstract

This bachelor's thesis deals with the integration of a suitable solution for managing the state of the client part of a web application. It analyzes the current state of the source code, discusses its shortcomings and describes popular solutions to this problem. Based on the analysis, a suitable technology for state management is selected and a detailed design of a specific solution is developed, which is subsequently implemented. At the end, user testing is carried out in order to evaluate the impact of the given modifications on the system and preparation for deployment is carried out.

Keywords state container, client side, state management, web application, Angular, Redux, NgRx

Kapitola 1
Úvod

OptiLynx je aplikácia, ktorá slúži primárne ako pokladničný systém pre optiky. Pomerne krátko po jej vzniku bola nasadená do reálnej prevádzky a dnes ju na svojich pobočkách využívajú zamestnanci a majitelia viacerých optík.

Aktuálne za jej správu zodpovedá firma Jagu s.r.o. a je možné podieľať sa na vývoji tejto aplikácie v rámci povinne-voliteľného predmetu softvérový tímový projekt. Počas dvoch semestrov sme na nej spoločne s piatimi kolegami pracovali, čo predstavuje moju prvú skúsenosť s danou aplikáciou. Mojou ulohou bola údržba a vývoj jej klientskej časti, ktorá vznikla v rámci záverečnej práce Bc. Jaroslava Hracha.

Aj napriek početným nedostatkom OptiLynxu vo mne táto skúsenosť vzbudila záujem výrazne rozšíriť svoje znalosti vo vývoji klientských častí webových aplikácií. Pri ponuke pracovať na nej v rámci záverečnej práce som preto dlho neváhal.

Cieľom tejto práce je pridať stavový kontajner do klientskej časti systému. Medzi čiastočné ciele práce patrí dôkladná analýza aktuálnej podoby zdrojového kódu, identifikácia nedostatkov aplikácie a analýza existujúcich riešení pre správu stavu. Na základne analýzy bude uskutočnená voľba, návrh a integrácia vhodného riešenia, následne jeho otestovanie a zhodnotenie vplyvu úprav na systém. Posledným čiastočym cieľom je príprava riešenia na nasadenie do produkcie a jeho samotné nasadenie.

Kapitola 2
Analýza

V úvode kapitoly bude stručne priblížená technológia, v ktorej bola vyvíjaná aplikácia, ktorá je predmetom záverečnej práce, následne budú zhrnuté výsledky analýzy jej nedostatkov a nakoniec sa budem zaoberať populárnymi riešeniami pre správu stavu v klientskej časti webových aplikácií.

Angular

OptiLynx je napísaný vo frameworku Angular. Ide o populárnu vývojovú platformu postavenú na jazyku TypeScript, ktorá slúži na tvorbu výkonných, sofistikovaných a škálovateľných jednostránkových aplikácií. Na ich budovanie Angular poskytuje rozhranie založené na komponentoch, kolekciu dobre integrovaných knižníc, ktoré poskytujú širokú škálu funkcionalít a sadu vývojárskych nástrojov uľahčujúcich ich vývoj, údržbu a testovanie [1].

Komponenty predstavujú základné stavebné bloky všetkých aplikácií vyvinutúch v Angulare. Každý z nich obsahuje okrem triedy v TypeScripte, ktorá definuje ich správanie, tiež HTML šablónu a CSS selektor umožňujúci ich použitie v iných komponentoch. Môže ešte obsahovať kaskádové štýly, ktoré budú na jeho šablónu aplikované, čím upravia jej výzor [2].

2.1 Analýza nedostatkov aplikácie

Dôležitou súčasťou mojej práce je analýza existujúceho stavu aplikácie a identifikácia jej nedostatkov. Jej výsledky výrazne ovplyvnia ďalšie časti práce, a to voľbu frameworku, vymedzenie rozsahu a v neposlednom rade samotný návrh a implementáciu zvoleného riešenia.

Po dôkladnej analýze súčasného stavu aplikácie som jednotlivé nedostatky rozdelil do dvoch skupín podľa ich dopadu. Do prvej budú patriť problémy, ktoré majú nepriaznivý účinok na užívateľskú skúsenosť. Druhá skupina bude obsahovať nedostatky negatívne vplývajúce na samotný vývoj aplikácie.

2.1.1 Nedostatky s dopadom na užívateľskú skúsenosť

Užívateľská skúsenosť (anglicky user experience, skrátene UX) zahŕňa všetky pocity a dojmy užívateľa vznikajúce z akejkoľvek interakcie s produktom, systémom či službou [3]. Jedná sa o jeden z najdôležitejších prvkov každého produktu. Dokazuje to aj fakt, že práve oreintácia na užívateľskú skúsenosť poskytuje spoločnosti Apple významnú konkurenčnú vyhodu a výrazne prispieva k jej úspechu, a to aj napriek vyšším cenám podobných produktov na trhu [4].

Okrem toho však dobrá užívateľská skúsenosť tiež pozitívne ovplyvňuje zážitok koncových užívateľov z používania danej aplikácie a zvyšuje ich efektivitu. Vďaka tomu ju budú viac pre-

ferovať v porovnaní s ostatnými aplikáciami podobného typu, čo bude viesť k zlepšeniu mena spoločnosti [5], ktorá ju vytvorila a potenciálne k väčšiemu množstvu zákazníkov.

Túto skupinu nedostatkov preto považujem za najkritickejšiu z hľadiska úspechu produktu a je potrebné pracovať na ich odstránení, resp. redukcii ich dopadu na zážitok a dojmy užívateľov z interakcie s danou aplikáciou.

2.1.1.1 Nízka reaktivita aplikácie

Pod pojmom reaktivita aplikácie mám na mysli čas medzi užívateľskou interakciou a chvíľou, kedy sa aplikácia dostane do stavu, v ktorom je užívateľ schopný vykonať ďalší úkon nutný na dokončenie jeho cieľa. V tomto prípade si je pod interakciou možné predstaviť kliknutie na zoznam zákaznikov a požadovaným stavom jeho načítanie.

Stránky, ktoré sa dlho načítajú alebo majú vysoký čas odozvy vytvárajú zlú užívateľskú skúsenosť a používanie takýchto aplikácii je pre jednotlivých užívateľov frustrujúce a motivuje ich k tomu, aby s tým prestali [6]. Frustrácia je o to väčšia, keď je užívateľ zamestnancom a je nútený danú aplikáciu využívať k výkonu svojej práce nezávisle od jeho názoru a bez možnosti s tým čokoľvek urobiť.

Kolegovia, s ktorými som na Optilynxe pracoval, mi iste dajú za pravdu, že reaktivita tejto aplikácie nie je dostačujúca pre poskytnutie dobrej užívateľskej skúsenosti. Hlavný problém vidím v posielaní vysokého počtu dotazov na backend bez akejkoľvek snahy o ukladanie odpovedí do vyrovnávacej pamäte. Pri navštívení adresy sú opätovne zaslané všetky dotazy pre získanie potrebných dát, a to aj v prípade, že od posledného dotazovania v nich nenastala žiadna zmena. To vo výsledku blokuje užívateľa, ktorý je nútený pred ďalšou interakciou čakať na jednotlivé odpovede.

Je zjavné, že nie je možné zaručiť aktuálnosť dát uložených vo vyrovnávacej pamäti, to však vo väčšine prípadov nespôsobuje problém. Cieľom je v užívateľovi vyvolať pocit, že nemusí na dáta potrebné pre svoju prácu čakať, čím sa zvýši jeho efektivita a zlepší skúsenosť s používania aplikácie [6]. Nakoniec stačí vopred poskytnuté dáta nahradiť výsledkom poslaného dotazu, čím sa opäť zaručí ich aktuálnosť.

2.1.1.2 Problémy UX dizajnu

Aj napriek tomu, že klientská časť OptiLynxu, nazývaná tiež FrontLynx, obsahuje viaceré prvky snažiace sa o vylepšenie užívateľskej skúsenosti, vidím tam početné miesta pre zlepšenie. Aplikáciu však nevyužívam denne na rozdiel od zamestancov viacerích optík, kvôli čomu si s veľkou pravdepodobnosťou neuvedomujem mnohé jej problémy vyplývajúce z nedostatočnej orientácie na užívateľa počas návrhu. V pribehu analýzy som bol schopný identifikovať nasledujúce nedostatky, ktorých odstránenie podľa môjho názoru prispeje v najväčšej miere k zlepšeniu užívateľskej skúsenosti. Dopad týchto nedostatkov je však v porovnaní s dopadom nízkej reaktivity aplikácie zanedbateľný.

Jednou z hlavných vecí, ktorá mi v klientskej časti aplikácie ako užívateľovi chýba, je možnosť rýchlo a jednoducho reagovať na jednotlivé udalosti. Vždy keď je užívateľ informovaný o určitej chybe či zmene spôsobenej jeho interakciou, mal by mať možnosť na ňu patrične reagovať stlačením tlačidla. Konkrétne reakcie budú závisieť od špecifickej udalosti.

Napríklad v prípade, že nastala chyba pri pokuse o priadnie zákazníka by mala byť užívateľovi poskytnutá možnosť opätovne dotaz odoslať. Pri oznámení úspešného zmazania kategórie by zas adekvátnou reakciou mohla byť možnosť danú kategóriu znovu pridať pre prípad, že by išlo o chybu užívateľa.

Momentálne sa reakcie tohto typu v aplikácii vyskytujú len na jedinom mieste, čo je škoda, pretože ide o veľmi praktickú funkčnosť, ktorá užívateľom uľahčí a spríjemní prácu.

Ďalšie miesto pre zlepšenie vidím v informovaní užívateľa o aktuálnom stave aplikácie a jednotlivých obmedzení, ktoré z neho plynú. Tento problém sa vyskytuje na viacerých miestach a je evidentný už po samotnom prihlásení do aplikácie. Užívateľ by mal počkať pár sekúnd na možnosť výberu pobočky a terminálu, no nie je tam prítomný žiaden indikátor, ktorý by ho informoval o načítaní tejto časti. Počas toho má možnosť normálne klikať na tlačidlá a keď tak urobí, v dôsledku chyby sa komponent umožňujúci výber pobočky nenačíta a užívateľ sa zablokuje. Hneď po tomto výbere by mal opäť čakať niekoľko sekúnd na ponúknutie možnosti otvoriť pokladňu, znovu bez akejkoľvek indikácie, no to mu nebráni vo vytvorení zakázky a jej následovnom zaplatení.

Tieto problémy robia prácu s aplikáciou menej intuitívnou, vyžadujú špecifické skúsenosti užívateľov a zhoršujú ich názor na systém ako taký.

2.1.2 Nedostatky s dopadom na vývoj

Druhú skupinu tvoria nedostatky, ktoré nepriaznivo pôsobia na samotný vývoj aplikácie. Môže sa jednať o technický dlh spôsobený zlými praktikami, nadbytočnú komplexitu kódu zhoršujúcu jeho prehľadnosť a udržateľnosť alebo tzv. pachy v kóde (anglicky code smells), ktoré môžu negatívne vplývať na efektivitu programátora a jeho celkovú snahu pri ďalšom vývoji.

Všetky vyššie spomínané problémy spomaľujú vývoj nových funckionalít či opravu chýb, čo v dôsledku stojí ako zákazníka, tak spoločnosť viac peňazí a navyše ešte poškodzuje jej meno.

Nejedná sa o kritické chyby a aj s nimi môže aplikácia fungovať, ba dokonca pôsobiť profesionálne z pohľadu užívateľov, no ide o nedostatok, ktorý je predurčený ku katastrofe. Podľa teórie rozbitého okna každý problém, ktorému nebola venovaná pozornosť prispieva k formovaniu názoru ľudí na prostredie, v ktorom sa daný problém vyskytuje [7]. To síce neznamená, že v prípade neudržiavaného kódu sa jednotliví vývojari automaticky prestanú snažiť, no z vlastnej skúsenosti viem povedať, že do kódu, kde neviem identifikovať jediný nedostatok, neprispejem ničím iným, než najlepšou verziou toho, čo som schopný napísať.

2.1.2.1 Chýbajúca dokumentácia REST API

Aktuálne za najväčší nedostatok aplikácie s negatívnym dopadom na rýchlosť vývoja považujem chýbajúcu dokumentáciu programovacieho rozhrania, ktoré slúži na komunikáciu so servérovou časťou. Jediným spôsobom, ako si byť istý typom návratových hodnôt (vzhľadom na nedostatký opísané v nasledujúcich dvoch sekciách) je ich vypisovať do konzoly, čo vývojára zbytočne zdržiava a kazí jeho dojem z aplikácie ako takej.

2.1.2.2 Nesprávny typ návratových hodnôt

Tento problém sa našťastie v aplikácii často nevyskytuje, ale keď už áno, tak na naozaj kritických miestach. Osobne som naň narazil práve v službe, ktorej úlohou bola komunikácia so servérovou časťou aplikácie. Kvôli chýbajúcej dokumentácii som sa naivne nádejal, že bol kladený dôraz na správnosť typov návratových hodnôt, no márne. Na základe nedostatočných skúseností s vybraným riešením pre správu stavu som predpokladal, že chyba bude v nepochopení, resp. nesprávnom využívaní poskytnutých nástrojov, čo viedlo k dlhému a frustrujúcemu hľadaniu danej chyby na miestach, kde vôbec nebola.

2.1.2.3 Chýbajúce typy

Jednou z kľúčových výhod, ktoré TypeScript oproti JavaScriptu poskytuje je silné typovanie. Je preto dobrou praktikou uvádzať typy premenných, argumentov funkcií a ich návratových hodnôt všade, kde to je rozumné. Samotným jazykom to síce vyžadované nie je, no zlepšuje to

prehľadnosť kódu, čo prispieva k vyššej efektivite vývoja. Vo väčšine aplikácie je táto praktika realizovaná, no nie konzistentne a v niektorých jej častiach to vyslovene chýba [8].

2.1.2.4 Nesprávna granularita komponentov

Mnohé komponenty sú aktuálne zložené z tisícov riadkov bez dobrého dôvodu, ktorým by sa dal tento rozsah obhájiť. Rozdelením týchto komponentov na viacero menších sa zvýši nielen prehľadnosť, ale potenciálne aj znovupoužiteľnosť vzniknutých komponentov pri správnej abstrakcii [8].

2.1.2.5 Početné inkonzistencie

Každý vývojár softvéru má svoj vlastný štýl programovania. To sa vzťahuje na širokú škálu vecí od praktík, ktoré považuje za dobré a dôležité, cez menné konvencie až po maximálny počet znakov na jednom riadku, ktorý ešte pokladá za prijateľný. Avšak v projekte, na ktorom pracuje viacero ľudí je jednou z najdôležitejších vecí konzistencia. Tá by mala byť osobnému štýlu jednoznačne nadradená, pretože v opačnom prípade aplikácia pôsobí chaoticky a neprofesionálne, čo môže vývojara ľahko demotivovať od snahy písať kvalitný kód [7]. Frontendová časť Optilynxu obsahuje inkonzistencie v názvoch premenných, funkcií a ich argumentov, v názvoch a štruktúre súborov a tiež v jednotlivých riešeniach toho istého problému v rôznych častiach aplikácie.

2.1.2.6 Komponenty s obchodnou logikou

Komponenty by mali správne obsahovať iba logiku súvisiacu s prezentáciou dodaných dát a interakciou s užívateľmi. Na istých miestach v aplikácii však obsahujú aj logiku obchodného charakteru, ktorá by mala byť na základe dobrých praktík v Angulare vyňatá do samostatných služieb. Tieto služby by mali obsahovať všetkú obchodnú logiku, ktorá môže byť v istých prípadoch veľmi komplikovaná a znižovať tak prehľadnosť a znovupoužiteľnosť daného komponentu [9].

2.1.2.7 Málo deskriptívne názvy

Posledným nedostatkom sú názvy s nízkou výpovednou hodnotou, ktoré sa našťastie v aplikácii vyskytujú pomerne málo. Ide najmä o názvy funkcií resp. metód a spôsobujú spomalenie vývoja mrhaním času programátora, ktorý sa o ich účelu zväčša dozvie až po nahliadnutí do kódu a to aj v prípade, že ho vôbec nemuseli zaujímať. Túto chybu však vzhľadom na jej dopad a kvantitu považujem za najmenej problematickú.

2.2 Analýza riešenia pre správu stavu

2.2.1 Správa stavu aplikácie

Stav aplikácie je pojem, ktorý vo svete softwérového inžinierstva nemá jednotnú definíciu. V klientskej časti je ho však možné vnímať ako výsledok všetkých akcií, ktoré boli vykonané užívateľom alebo nastali po načítaní stránky. Jedná sa teda o reprezentáciu systému v danom časovom okamihu.

So zväčšujúcim sa rozsahom aplikácie tiež narastá komplexnosť správy jej stavu [10]. U veľkých aplikácií často nestačia nástroje, ktoré poskytuje daná knižnica resp. framework, a preto je nutné prísť so sofistikovanejším riešením.

V prípade Angularu, v ktorom je Optilynx vyvinutý, sa naskytuje viacero spôsobov, ako stav spravovať. Medzi tie lepšie patrí koncept, podľa ktorého sa komponenty rozdelia na prezentačné a funkčné (alebo tiež hlúpe a chytré). Je tiež možnosť vyvinúť služby predstavujúce jediný zdroj pravdy pre stav zdieľaný viacerými komponentmi s potrebnou obchodnou logikou na jeho úpravu.

V istom bode je však pre aplikáciu najvýhodnejšie používať jednu z knižníc špecificky určených pre správu vybraných častí stavu [11].

Všetky vyššie spomínané riešenia implementujú v istej podobne stavový kontajner.

2.2.2 Správa stavu pomocou funkčných komponentov

Angular je postavený na architektúre založenej na komponentoch, ktoré slúžia ako základné stavebné bloky tohto frameworku [12]. V praxi sa často stáva, že komponenty obsahujú okrem prezentačnej logiky aj logiku obchodnú súvisiacu s ich funkcionalitou či získavaním dát, čo pri početnom výskyte môže spôsobovať problémy s udržateľnosťou, testovateľnosťou a škálovateľnosťou komplexnej aplikácie.

Jeden zo spôsobov, ako sa tomuto problému vyhnúť je rozdeliť daný komponent na jeden či viacero prezentačných komponentov, do ktorých extrahujeme dáta a logiku týkajúcu sa zobrazovania a interakcie s užívateľom. Tieto prezentačné komponenty budú obsiahnuté v tzv. funkčnom či kontajnerovom komponente, ktorý im potrebné dáta dodá a bude obsahovať prípadnú obchodnú logiku [13].

Aj napriek dojmu, že sa takéto rozdelenie môže javiť ako nepatrná zmena, prináša so sebou množstvo výhod. Prezentačné komponenty budú prehľadnejšie a bude ich možné opätovne využívať v rôznych situáciách, nakoľko budú súčasťou jedine prezentačnej vrstvy. Zavedenie funkčných komponentov sprehľadní správu stavu aplikácie, jednoznačne oddelí zodpovednosti [14] a poskytne pre prezentačné komponenty jediný zdroj pravdy pre obsiahnuté dáta, čo prispeje k oprave inkonzistencií tohto stavu.

2.2.3 Správa stavu pomocou stavových služieb

Prepracovanejším riešením pre správu stavu je využívať službu, ktorá bude zohrávať úlohu stavového kontajnera. Dáta nachádzajúce sa v službe tohto typu je možné bez veľkých problémov zdieľať aj naprieč časťami aplikácie, ktoré sú od seba v strome komponentov vzdialené.

Je však nutné zvoliť vhodnú reprezentáciu týchto dát. Jednoduché dátové typy, rovnako ako tzv. sľuby (anglicky promises) na to nestačia. Ich hlavnou nevýhodou je, že komponenty vidia len aktuálnu hodnotu premennej, no nie sú schopné patrične reagovať na jej zmeny. Je tu tiež riziko, že niektorý z týchto komponentov danú hodnotu zmení, čo môže viesť k nepredvídateľným zmenám [15] a nekonzistentnému stavu v aplikácii. Dobrým riešením tohto problému je využitie knižnice RxJS.

2.2.3.1 RxJS

RxJS (celým menom Reactive Extensions for JavaScript, v preklade reaktívne rozšírenia pre JavaScript) je knižnica umožňujúca využívanie paradigmy reaktívneho programovania na uľahčenie písania asynchrónneho kódu a kódu založenom na spätnom volaní. Poskytuje implementáciu návrhového vzoru pozorovateľa pomocou tzv. pozorovateľných objektov (anglicky observables) spolu s veľkým množstvom pomocných funkcií [16] [17].

Tieto pozorovateľné objekty uľahčujú posielanie hodnôt medzi jednotlivými časťami aplikácie vo forme dátových tokov, vďaka čomu budú oboznámené o ich zmenách, čo im umožní reagovať na ne. Rozhranie pre programovanie aplikácií je rovnaké bez ohľadu na to, či sú dáta vysielané synchrónne alebo asynchrónne a programátor sa väčšinou nemusí starať o logiku vzniku či zániku pozorovateľných objektov, kvôli čomu sú v rámci Angularu vo vieľkej miere využívané [18].

2.2.4 Správa stavu pomocou knižníc

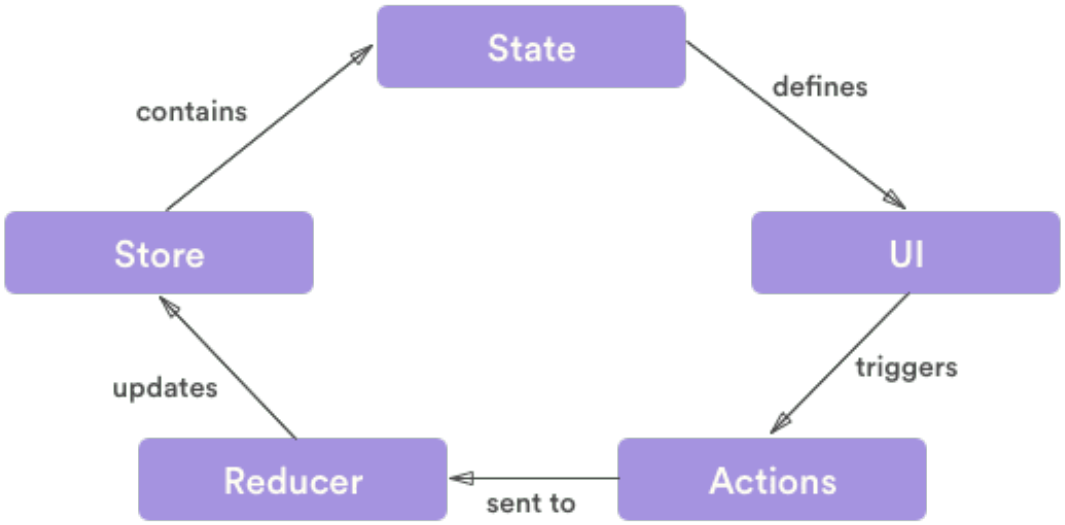
Stav zdieľaný naprieč veľkou časťou aplikácie je vhodné spravovať sofistikovanejším spôsobom, a to s využitím knižníc, ktoré sú na to špecificky určené.

2.2.4.1 Redux

Redux je knižnica a návrhový vzor založený na architektúre Flux, vyvinutej spoločnosťou Facebook v presvedčení, že návrhový vzor MVC má problémy so škálovateľnosťou pre veľké aplikácie [19]. Flux kladie dôraz na jednosmerný dátový tok, centralizovaný spôsob správy stavu a nemennosť dát [20], ktorú ale nevyžaduje [21].

Redux túto architektúru rozširuje o centralizované dátové úložisko vo forme jedinej štruktúry, zjednodušenie dátového toku a dožadovania sa nemmenosti stavu [21]. Dátové úložisko (store) predstavuje jediný zdroj pravdy pre celú aplikáciu. Jednotlivé komponenty alebo služby si pomocou selektorov môžu konkrétne časti stavu extrahovať, čím budú informované o ich zmenách. Tieto zmeny väčšinou nastanú po interakcií užívateľa, ktorá bude mať za následok vyslanie špecifických akcií, na ktoré načúvajú reduktory (reducers). Jedná sa o čisté funkcie (bez vedľajších účinkov), ktoré obdržia aktuálny stav a vyvolanú akciu, na základe čoho vytvoria nový, patrične upravený stav [22].

Použitie tohto návrhového vzoru je však potrebné dôkladne zvážiť, pretože do aplikácie prinesie ďalšiu komplexnosť spoločne s kódom, ktorý slúži iba pre vytvorenie vyžadovaných štruktúr [23].



Obr. 2.1 Redux - životný cyklus [24]

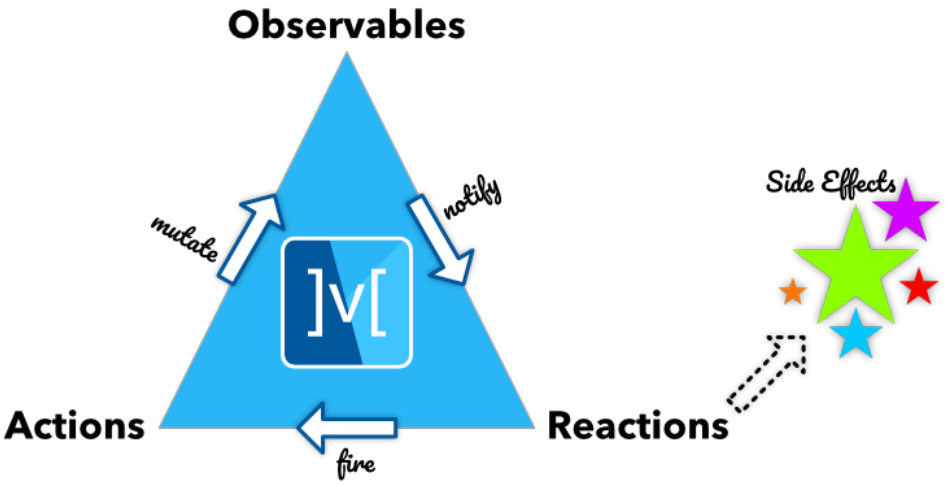
2.2.4.2 MobX

Ďalšou populárnou možnosťou je MobX. Je postavený na návrhovom vzore pozorovateľa a prináša prvky reaktívneho programovania ako ucelená knižnica pre správu stavu [25].

Využíva jednosmerný dátový tok a je založená na troch základných konceptoch. Prvým je samotný stav, ktorý môže byť uložený v ľubovoľnej štruktúre či viacerích štruktúrach, musí však byť označený ako pozorovateľný objekt. Ďalším dôležitým konceptom sú derivácie, ktoré sa delia

na vypočítané hodnoty a reakcie. Vypočítané hodnoty je vždy možné derivovať z akutálneho stavu pozorovateľného objektu pomocou čistých funkcií, zatiaľ čo reakcie, označované ako most medzi reaktívnym a imperatívnym programovaním, slúžia na vykonávanie vedľajších účinkov pri akejkoľvek zmene pozorovaného stavu. Tieto zmeny je možné vykonávať pomocou akcií a sú automaticky propagované do častí aplikácie, ktoré využívajú daný stav alebo jeho derivácie [26].

MobX, narozdiel od iných populárnych knižníc, nevyžaduje písanie veľkého množstva zbytočného kódu a s Angularom funguje veľmi dobre vďaka svojej optimalizovanej verzii MobX-Angular [27].



Obr. 2.2 MobX - životný cyklus [28]

2.2.4.3 Akita

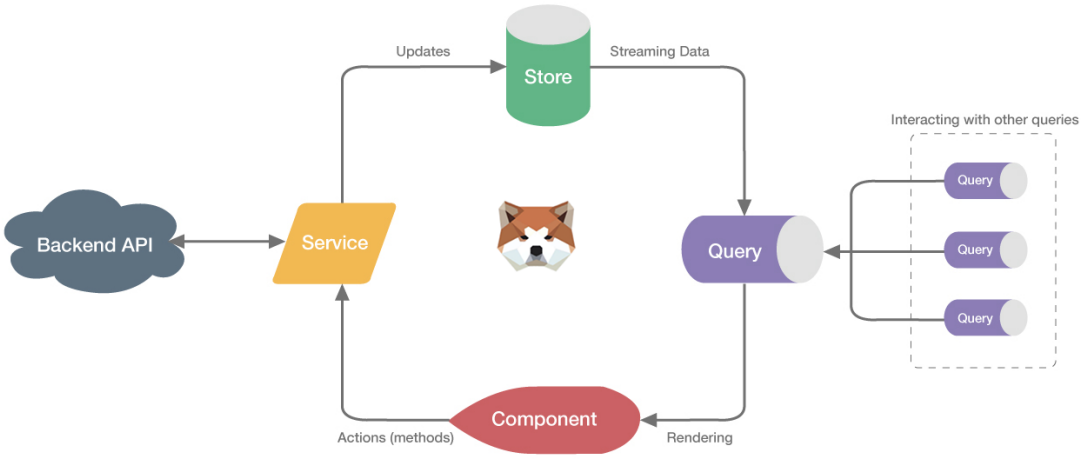
Pomerne nedávno vyvinutou knižnicou pre správu stavu je Akita vydaná ako open-source projekt v máji roku 2018. Je postavená na knižnici RxJS a zakladá na princípoch objektovo orientovaného programovania, čím sa odlišuje od väčšiny. Kombinuje myšlienky viacerích datových úložísk z Fluxu, nemennosti dát z Reduxu s konceptom dátových tokov z RxJS pre vytvorenie modelu pozorovateľných dátových úložísk [29].

Stojí na štyroch základných komponentoch. Dátovom úložisku, ktoré slúži ako jediný zdroj pravdy pre konkrétnu časť aplikácie, akciách reprezentujúcich jednotlivé udalosti, ktoré ovplyvňujú správanie úložiska, dotazov (anglicky queries) slúžiacich pre získanie jeho časti a efektov pre správu vedľajších účinkov [30].

2.2.4.4 NgRx

Od svojho vzniku v roku 2015 začal Redux rýchlo naberať na popularite vďaka svojej jednoduchosti, malej veľkosti (len 2 KB vrátane závislostí) a dobrej dokumentácii. Najčastejšie býva asociovaný s Reactom, ide však o samostatnú knižnicu, ktorú je možné použiť vo väčšine moderných frontendových frameworkov [31]. To však vytvára motiváciu pre vznik riešení, ktoré na ňom zakladajú, no poskytujú lepšiu integráciu a optimalizáciu pre špecifické frameworky.

V prípade Angularu je najznámejším riešením NgRx. Ide o skupinu knižníc, ktorá tento návrhový vzor implementuje prostredníctvom pozorovateľných objektov z RxJS. Má podobný životný cyklus ako Redux, ale rozširuje ho o efekty, ktoré programátorovi umožnia vykonávať vedľajšie účinky spojené so špecifickými akciami s možnosťou ďalšie akcie vysielať [32]. Narozdiel



Obr. 2.3 Akita - životný cyklus [29]

od klasického Reduxu tiež poskytuje viaceré pokročilejšie koncepty a optimalizácie, ktoré však v tejto fáze analýzy nemá zmysel podrobne rozoberať.

2.2.4.5 NGXS

NGXS je knižnica a návrhový vzor pre Angular modelovaný podľa vzoru CQRS (Command and Query Responsibility Segregation, v preklade segregácia zodpovedností príkazov a dotazov), ktorý tiež implementuje Redux či NgRx.

NGXS je však v porovnaní s nimi jednoduchšie používať a redukuje množstvo potrebného kódu využívaním prvkov moderného TypeScriptu, ako sú triedy a dekorátory [33], ale aj vďaka spojeniu zmien stavu, vedľajších účinkov a selektorov do jediného súboru a to dátového úložiska, čo tiež speje k nutnosti vytvoriť menej rôznych akcií, ale tiež k sťaženiu testovania. Nejedná sa však o veľký skok od Reduxu, takže v prípade, že je s ním programátor zoznámený, ľahko pochytí aj NGXS [34].

2.2.5 Zhrnutie

Pri výbere konkrétneho riešenia pre správu stavu je potrebné zvážiť mnoho faktorov.

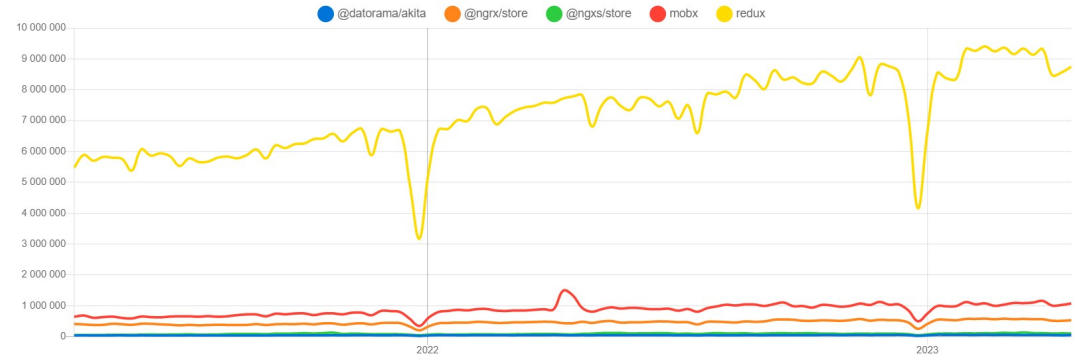
Hneď na začiatku je však možné konštatovať, že spravovanie stavu rozdelením komponentov na prezentačné a funkčné nebude vzhľadom na veľkosť aplikácie dostačovať. V každom prípade sa ale jedná o dobrú praktiku, ktorá môže zvolené riešenie dopĺňať pre zvýšenie znovupoužiteľnosti a prehľadnosti kódu.

Praktickejším riešením je myšlienka kontajnerovej služby, ktorá by obsahovala nejaký stav, čím by bol počas behu aplikácie perzistentný. Zároveň by preň bola jediným zdrojom pravdy, vďaka čomu by sa v rôznych komponentoch dosiahla jeho konzistentnosť. Z môjho pohľadu je najlepšie reprezentovať daný stav formou behaviorálnych subjektov z knižnice RxJS, ktoré by sa navonok (pre jednotlivé komponenty) vystavovali ako pozorovateľné objekty. Nezanedbateľnou výhodou je relatívna jednoduchosť tohto konceptu, keďže práca s pozorovateľnými objektami je v Angulare pomerne bežná.

Z osobnej skúsenosti je však spomínané riešenie vhodné pre aplikácie malého až stredného rozsahu a od istého bodu začne prinášať viac problémov, než výhod. Tu by som sa chcel znovu odvolať na rozsah Optilynxu, ktorý už tento bod prekročil pred dlhou dobou.

Porovnanie knižníc bude kvôli ich počtu komplikovanejšie. Žiadnu z nich navyše nie je možné označiť za objektívne najlepšiu a každá prináša určité výhody oproti ostatným. Voľba konkrétnej knižnice preto závisí najmä na osobných preferenciách programátora a probléme, ktorý chce jej integráciou vyriešiť, prípadne zredukovať.

Najprv zhodnotím popularitu jednotlivých knižníc, ktorá predstavuje pomerne dôležitý porovnávací faktor. Popularita nie je dobrým indikátorom všeobecnej kvality danej knižnice, no dá sa predpokladať, že je priamo úmerná s množstvom materiálov, ktoré je možné o nej na internete nájsť. Z tohto hľadiska, ako je vidieť na grafe 2.4, Redux ostatné knižnice v počte stiahnutí priam dominuje. O niečo menej populárnou knižnicou je MobX a ďalej nasleduje NgRx. Tá je však na Reduxe založená, kvôli čomu sa na ňu dajú mnohé materiály aplikovať.



Obr. 2.4 Graf vyjadrujúci počet stiahnutí uvedených knižníc

Medzi hlavné výhody, ktorými sa Akita odlišuje od ostatných knižníc, patrí nízke množstvo tzv. štandardného (anglicky boilerplate) kódu, ktorý programátor musí opakovane písať pri implementácii každej funkcionality s minimálnymi, resp. žiadnymi zmenami [35]. Keď je navyše vývojár dobre zoznámený s princípmi objektovo orientovaného programovania, nebude pre neho problém naučiť sa pracovať s touto knižnicou [29]. Veľkou nevýhodou je však jej nízka rýchlosť pri vykonávaní veľkého počtu operácií alebo práci s väčším počtom entít [36], kvôli čomu ide o vhodné riešenie najmä pre menšie aplikácie s pomerne jednoduchým stavom.

MobX zas poskytuje okrem svojej jednoduchosti a pomerne malého množstva štandardného kódu tiež dobrý výkon [37]. Jeho hlavná nevýhoda spočíva v stave, ktorý nie je nemenný a je ho tak možné jednoducho upravovať, čo speje k horšej testovateľnosti a udržateľnosti. Ide však o dobré riešenie, keď je pri výbere dôležitou metrikou rýchlosť vývoja [38].

Existuje dobrý dôvod, prečo patria knižnice Redux a NgRx medzi najpopulárnejšie riešenia pre správu stavu. Okrem dobrého výkonu [37] [36] a kvalitnej dokumentácie prinášajú centralizovaný spôsob správy stavu, ktorý prispieva k jednoduchému testovaniu a opravovaniu chýb, dobrej udržateľosti a škálovateľnosti aplikácie. Bývajú však často kritizované pre strmú krivku učenia a nutnosti písania veľkého množstva štandardného kódu, čo má za následok pomalší vývoj v porovnaní s inými riešeniami. Aj kvôli týmto nevýhodám sa ich oplatí využívať takmer výlučne pre aplikácie veľkého rozsahu, kde je potrebné pristupovať k správe stavu čo najviac konzistentne [38] [39] [40].

Poslednou z uvažovaných knižníc je NGXS, ktorá má mnoho podobností s NgRx. Narozdiel od nej je však jednoduchšia, vyžaduje písanie menšieho množstva štandardného kódu a poskytuje viac rôznych funkcionalít [40]. Taktiež vo väčšej miere využíva prvky Angularu ako sú triedy, dekorátory a injekcia závislostí [41].

Kapitola 3
Návrh

V prvej časti tejto kapitoly bude zvolený vhodný spôsob pre správu stavu aplikácie, jeho detailný opis a poskytnutie argumentov pre danú voľbu. Následne bude vymedzený rozsah implementácie a popísané detaily existujúceho návrhu.

3.1 Zvolený spôsob riešenia stavu

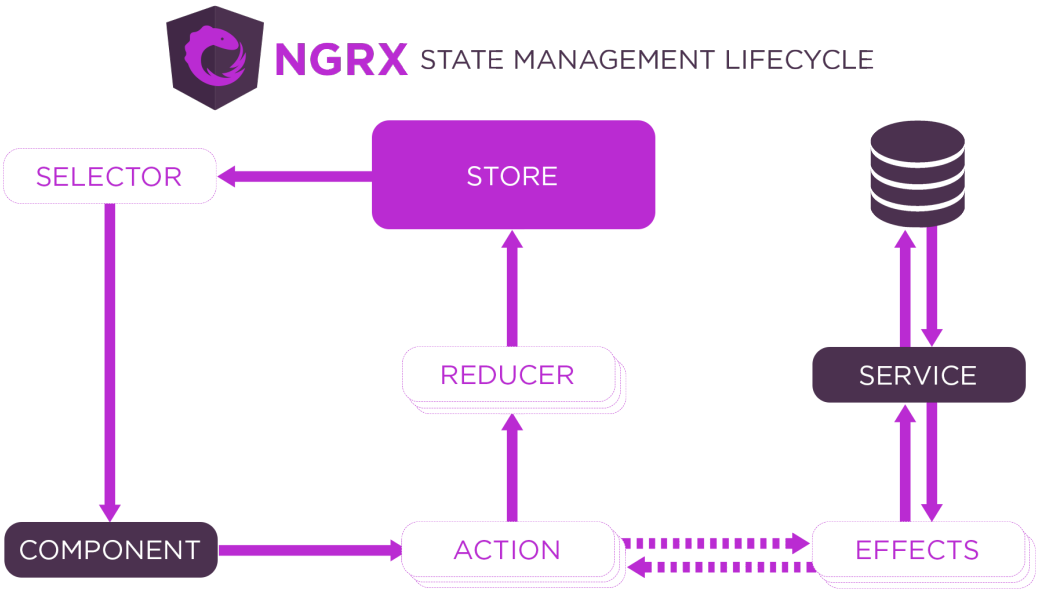
Po dôslednom zvážení všetkých možností pre správu stavu aplikácie som sa nakoniec rozhodol pre knižnicu NgRx založenej na návrhovom vzore Redux. S prihliadnutím na rozsah aplikácie som presvedčený, že ide o vhodné riešenie aj napriek nepopierateľným nevýhodám spojeným s integráciou tejto knižnice. Medzi hlavné výhody, ktoré NgRx poskytuje, patrí možnosť izolovať od užívateľského rozhrania akúkoľvek obchodnú logiku, rovnako ako interakciu s vedľajšími účinkami externých zdrojov, čo vedie k zjednodušeniu jednotlivých komponentov a podporuje princíp jedinej zodpovednosti. Spôsob ukladania a vykonávania zmien spravovaného stavu je predikovateľný, čo uľahčuje testovanie a prispieva k lepšej prehľadnosti a udržateľnosti. Vďaka jednotnej a nemennej štruktúre dátového úložiska je naviac možné pomerne jednoducho detekovať zmeny pomocou onPush stratégie [42], čo má za následok vyšší výkon aplikácie [43]. Poslednou výhodou je veľká popularita ako samotnej knižnice NgRx, tak návrhového vzoru Redux, na ktorom je založená 2.4, čo zaručuje veľké množstvo materiálov, ktoré je možné na internete dohľadať. To je dôležité najmä kvôli vysokej pravdepodobnosti, že aplikáciu budú ďalej vyvíjať predovšetkým študenti, ktorým by som sa rád vopred ospravedlnil.

3.1.1 Bližší opis riešenia

Na obrázku 3.1 popisujúcom životný cyklus NgRx je znázornených päť dôležitých konceptov, na ktorých je daná knižnica postavená. V tejto časti sa budem usilovať o ich stručné priblíženie.

3.1.1.0.1 Dátové úložisko predstavuje stavový kontajner aplikácie a uchováva všetok spravovaný stav vo forme jedinej nemennej dátovej štruktúry, ktorá ho komponentom vystavuje ako pozorovateľné objekty [42]. Tvorí jadro tejto knižnice a programátor prostredníctvom triedy, ktorá ho reprezentuje, vysiela jednotlivé akcie a vyberá si konkrétne časti stavu, s ktorými chce pracovať [44].

3.1.1.0.2 Akcie vyjadrujú jedinečné udalosti, ktoré môžu nastať počas behu aplikácie. To zahŕňa všetko od užívateľskej interakcie so stránkou cez externú komunikáciou prostredníctvom sieťových požiadaviek až po interakciu s programovacími rozhraniami zariadenia, ale tiež mnohé



Obr. 3.1 NgRx - životný cyklus [44]

ďalšie udalosti, ktoré môžu nastať. Pomáhajú porozumieť, ako sú jednotlivé udalosti spracúvané a sú reprezentované typom, ktorý danú akciu jednoznačne opisuje a prípadnými ďalšími vlastnosťami pre poskytnutie dodatočného kontextu alebo potrebných metadát vyžadovaných konkrétnou akciou [45].

3.1.1.0.3 Reduktory alebo tiež redukčné funkcie sú zodpovedné za zmenu stavu dátového úložiska, ktorú vykonávajú synchrónnym spôsobom. Ide o čisté funkcie (vrátia vždy rovnaký výstup pri prijatí tých istých vstupov) bez vedľajších efektov. Každá redukčná funkcia obdrží pôvodný stav a vyslanú akciu, pre ktorú je určená a na základe jej typu vráti nový, vhodne upravený stav [46].

3.1.1.0.4 Selektory sú čisté funkcie používané na získavanie konkrétnych častí stavu dátového úložiska. Oproti klasickým funkciám poskytujú viacero výhod, ako je napríklad prenosnosť, kompozícia, jednoduchá testovateľnosť či typová bezpečnosť, no za najpodstatnejšiu výhodu považujem schopnosť memoizácie. Po vyvolaní selektora sa do pamäte uloží jeho výsledok a pri nasledujúcich volaniach s rovnakými argumentmi sa tento výsledok automaticky vráti bez nutnosti vykonať všetky operácie, ktoré môžu byť potenciálne náročné, čo prispeje k lepšiemu výkonu [47].

3.1.1.0.5 Efekty poskytujú spôsob izolácie vedľajších účinkov spojených s vyslanými akciami preč z komponentov. Medzi tieto účinky môžu patriť rôzne asynchrónne, ale aj synchrónne operácie založené na interakcii s externými zdrojmi či vysielanie ďalších akcií. Komponenty sa vďaka tomu musia starať len o získanie potrebného stavu z dátového úložiska a vysielanie akcií na jeho zmenu, čím je dosiahnutá ich čistejšia štruktúra [48].

3.2 Vymedzenie rozsahu

Implementácia stavového kontajnera prostredníctvom akejkoľvek knižnice pre správu stavu nie je strieborná guľka a prináša so sebou určité nevýhody. Konkrétne pri implementácii knižnice NgRx do aplikácie zavediem vyššiu komplexnosť a nutnosť písania pomerne veľkého množstva štandardného kódu pri rozširovaní funkcionalít v jej častiach, kde bude daná knižnica využívaná [49]. Určite sa preto nejedná o vhodné riešenie pre správu celého stavu aplikácie, kvôli čomu je nutné jasne vymedziť rozsah implementácie.

Po patričnom zvážení výsledkov analýzy som sa rozhodol spravovať vybraným riešením stav, ktorý je zdieľaný naprieč aplikáciou a počas jej behu musí byť perzistentný. Túto kategóriu najlepšie reprezentuje stav, ktorý sa týka terminálu a autorizácie či autentifikácie užívateľa. Rovnako som sa rozhodol spravovať stav súvisiaci s filtrovaním entít v tabuľkách. Ten je tvorený len malým množstvom premenných a za normálnych okolností by sa ho neoplatilo spravovať knižnicou, nakoľko by to prinášalo neúmerne veľa nevýhod oproti získaným výhodám, ide však o stav úzko súvisiaci s druhou skupinou spravovaného stavu a jeho uchovávanie je možné dosiahnuť s minimálnym počtom zmien. Okrem neho do tejto kategórie ešte patrí stav užívateľského rozhrania o načítaní a režime úprav poskytujúci jednotlivým komponentom dodatočné informácie súvisiace so zobrazovaním.

Druhú spomínanú skupinu bude tvoriť stav, s ktorým sa často pracuje v rámci bežných prípadoch užitia. Jeho ukladanie bude slúžiť pre zvýšenie reaktivity aplikácie a zlepšenie celkovej užívateľskej skúsenosti. Konkrétne prípady užitia boli zvolené na základe [50]. Ide o proces pri obsluhe zákazníka, ktorý si chce kúpiť tovar, proces, pri ktorom si zákazník kupuje okuliare či kontaktné šošovky a proces naskladňovania nového tovaru. Stav, ktorý je v týchto procesoch využívaný však pre ne exkluzívny nie je a pomôže k lepšej reaktivite aj pri iných úkonoch. Do tejto skupiny patria zoznamy poisťovní, zákaznických kategórií, dodávateľov a poskytovaných služieb či úprav zákaziek na základe ich relatívne malého množstva. Časť s najväčším dopadom na využitie pamäte aplikáciou tvoria zoznamy zákazníkov a produktov, ktoré sú však využívané vo všetkých uvedených procesoch, kvôli čomu sa domnievam, že práve ukladanie tohto stavu prispeje k zlepšeniu užívateľskej skúsenosti v najväčšej miere. Podobne bude v dátovom úložiskú uchovávaný stav vybraného prvku v prípade všetkých entít, u ktorých je môžnosť zobraziť ich detail.

Pôvodne som chcel do druhej skupiny zahrnúť aj zoznam jednotlivých zákaziek. Po dôkladnejšej analýze som však došiel k záveru, že na základe spôsobu ich načítania je očakávané množstvo príliš veľké na to, aby ho bolo vhodné v dátovom úložisku uchovávať.

3.3 Detaily návrhu

Po vymedzení rozsahu je nutné popísať zámer a implementačné detaily daného riešenia. Hlavným zámerom návrhu bolo zlepšenie užívateľskej, ale aj vývojárskej skúsenosti.

3.3.1 Užívateľská skúsenosť

Zlepšenie užívateľskej skúsenosti je dosiahnuté najmä zvýšením reaktivity aplikácie. To je docielené ukladaním dát, ktoré sú často využívané pri bežných procesoch do vyrovnávacej pamäte. Tieto dáta je možné užívateľom ihneď poskytnúť spolu s vyslaním dotazu a následne ich nahradiť dátami prijatými v odpovedi, aby sa zaručila ich aktuálnosť. Výsledkom tejto zmeny bude vyvolanie ilúzie, že sa stránka načíta rýchlejšie a lepšie reaguje na interakcie používateľa, čo by malo viesť k vyššej efektivite pri jeho práci s aplikáciou [6]. Je však potrebné zdôrazniť, že cieľom nie je znížiť počet vyslaných dotazov, pretože aplikácia môže bežať na viacerých zariadeniach súčasne, kvôli čomu je nutné zaručiť konzistenciu dát, a teda zachovať procesy ich synchronizácie v aktuálnej podobe.

3.3.2 Vývojárska skúsenosť

Omnoho väčší problém však predstavuje zlepšenie vývojárskej skúsenosti. Kvôli nevýhodám spomenutým v sekcii 3.2, ktoré integrácia knižnice NgRx prináša je možné namietať, že zlepšenie vývojárskej skúsenosti v tomto prípade ani nemôže nastať. Za predpokladu, že budú Optilynx vyvíjať hlavne študenti v rámci záverečných prác či predmetu softvérový tímový projekt mi neostáva nič iné, než s daným tvrdením súhlasiť. Budem sa však usilovať o čo najväčšie zmiernenie negatívnych dopadov integrácie tejto knižnice.

V prvom rade bude kladený dôraz na úplnú izoláciu správy stavu do služieb kvôli skrytiu konkrétnej implementácie jednotlivých procesov. Taktiež bude snaha v čo najväčšej miere zachovať pôvodnú štruktúru týchto služieb, aby sa spôsob, akým sú v komponentoch využívané vôbec nelíšil, resp. sa líšil iba minimálne.

Používaním výhradne základných nástrojov, ktoré zvolená knižnica poskytuje, chcem prispieť k zníženiu negatívneho dopadu na rýchlosť vývoja spôsobeného kladením ďalších nárokov na programátora. Dúfam v tvorbu riešenia, ktoré bude objektívne prehľadné, ľahko rozšíriteľné, jednoducho testovateľné a na pochopenie ktorého bude potrebná len základná znalosť konceptov NgRx.

3.3.3 Implementačné detaily

Po integrácii riešenia musí byť systém schopný vykonávať všetky doteraz implementované funkcionality spôsobom, aby z pohľadu užívateľa neboli viditeľné žiadne rozdiely. Programátor si určitých zmien byť vedomý môže, no bolo by vhodné obmedziť ich na minimum kvôli konzistencii metód jednotlivých služieb.

Pre entity, ktorých zoznamy budú uchovávané v dátovom úložisku, je potrebné implementovať minimálne akcie pre získavanie aktuálnych a nastavovanie nových hodnôt, ale bolo by tiež príhodné implementovať akcie slúžiace na pridávanie, úpravu či mazanie jednotlivých entít.

Pre stav, ktorý má byť počas behu aplikácie perzistentný, je nutné vytvoriť akcie, ktoré budú pri práci s dátovým úložiskom imitovať pôvodnú funkcionalitu metód služieb určených pre správu tohto stavu.

Spočiatku som sa chybne domnieval, že bude možné zachovať pôvodnú štruktúru všetkých služieb v plnej miere. To však nešlo uskutočniť kvôli metódam súvisiacim s pridávaním, aktualizovaním a mazaním entít. Výsledkom komunikácie so servérovou časťou aplikácie za použitia programovacieho rozhrania poskytnutého Angularom je pozorovateľný objekt, s ktorým sa v konkrétnom komponente pracuje. Replikovať toto správanie po vyslaní akcie, ktorá nemá žiadnu návratovú hodnotu by síce možné bolo, ale išlo by o naozaj kuriózne riešenie, ktoré by som nevedel naimplementovať s čistým svedomím. Rozhodol som sa preto presunúť logiku z komponentov do vedľajších účinkov akcií, čím som dosiahol zjednodušenie daných komponentov, ale tiež porušenie konzistencie spôsobu, akým sú jednotlivé služby využívané. Na druhú stranu sa mi však podarilo zachovať štruktúru služieb pre správu stavu reprezentujúceho prvú skupinu podľa sekcie 3.2 a metód pre získavanie aktuálnych hodnôt stavu druhej skupiny, ktoré sú používané najčastejšie.

Kapitola 4
Implementácia

Táto kaptiola sa bude zaoberať podrobnejším opisom realizovaného riešenia. Jeho implementácia však neprebiehala ľahko kvôli početným chybám spomaľujúcim či inak znepríjemňujúcim vývoj. Najväčší negatívny dopad naň podľa môjho názoru mala chýbajúca dokumentácia programovacieho rozhrania určeného na komunikáciu so servérovou časťou aplikácie a chýbajúce či nesprávne návratové hodnoty jednotlivých metód, no neprehľadné komponentny v dôsledku veľkého rozsahu tomu tiež nepomáhali. Aj z tohto dôvodu bol počas implementácie kladený dôraz na opravu podobných nedostatkov, ktoré som si v jej priebehu všimol a nezabudol na ne.

Ešte pred začiatkom realizácie návrhu som však považoval za dôležité izolovať komunikáciu so servérovou časťou do samostatných služieb, aby bola jasne oddelená od obchodnej logiky a manipulácie s dátovým úložiskom.

4.1 Štruktúra projektu

Dátové úložisko je zložené zo štyroch celkov, pričom pomenovanie, umiestnenie a obsiahnutý stav každého z nich je zvolený na základe služieb, ktoré s ním pracujú. Aj napriek tomu, že by sa tento stav dal ešte ďalej rozdeliť na menšie časti, daná granularita bola zvolená kvôli úzkemu súvisu jednotlivých dát a lepšej orientácii v štruktúre projektu.

Štruktúra projektových súborov, ktoré boli počas realizácie riešenia pridané je znázornená na 4.1. Ide o priečinky a súbory, ktoré sa v projekte pred začiatkom implementácie nenachádzali, a teda boli v jej priebehu vytvorené, obrázok 4.2 obsahuje súbory, v ktorých bolo potrebné vykonať zmeny spojené s integráciou riešenia. Ako je vidieť, aj napriek jeho neveľkému rozsahu bolo nutné uskutočniť početné úpravy naprieč mnohými časťami aplikácie.

Logika jednotlivých celkov dátového úložiska je rozdelená do piatich súborov. Prvý z nich obsahuje model daného celku, ktorý mu jednoznačne určuje štruktúru a konkrétnym premenným poskytuje možné typy, od ktorých sa nemôžu odkloniť. V druhom súbore sú obsiahnuté akcie súvisiace s manipuláciou danej časti dátového úložiska. Môže ísť o jednoduché akcie určené na pridávanie či aktualizovanie konkrétneho prvku, ale tiež komplexné zmeny stavu odvodené od procesov spojených s funkciou danej aplikácie. V ďalšom súbore sa nachádzajú selektory, vďaka ktorým je možné vybrať si časť stavu dátového úložiska a využívať ju v konkrétnych komponentoch. Štvrtý súbor obsahuje počiatočný stav daného celku a reduktor slúžiaci na vykonávanie synchrónnych zmien na základe vyvolaných akcií a v poslednom súbore sa nachádzajú efekty spravujúce vedľajšie účinky a asynchrónne operácie.

Od tejto štruktúry sa odlišuje iba priečinok umiestnený v adresári zdrojového kódu aplikácie, ktorý slúži na prepojenie jednotlivých celkov do jediného dátového úložiska a reduktoru.

4.2 Podrobný opis kódu riešenia

Kvôli veľkému množstvu štandardného kódu, ktorý je pomerne repetitívny nie je vhodné ani zaujímavé detailne ho opísať v celom jeho rozsahu. Preto budú popísané len vybrané časti, ktoré prinášajú zaujímavé riešenie nejakého problemu alebo niečím špecifickú funkcionalitu.

Všetky akcie, selektory, reduktory a efekty súvisiace so správou jednotlivých zoznamov obsiahnutých v častiach dátového úložiska, ktoré zohrávajú úlohu vyrovnávacej pamäte sú takmer totožné. Pre detailný opis som si zvolil zoznam obsahujúci kategórie zákazníkov, pretože narozdiel od väčšiny týchto kolekcií je možné jednotlivé kategórie mazať.

Ako je možné vidieť v ukážke kódu 4.1, dátové úložisko obsahuje v súvislosti so zákaznickými kategóriami okrem ich kolekcie a konkrétnej kategórie tiež premenné poskytujúce informáciu o načítaní a móde úprav, ktoré sú využívané jednotlivými komponentmi. Na prvý pohľad by sa to mohlo zdať zbytočné, no existuje na to dobrý dôvod. Komponenty môžu akcie vysielať, no nie sú oboznámené o jej dokončení. V prípade, že si pred jej vyslaním nastavia stav na načítanie, aby zabránili užívateľovi vykonávať určité operácie, nebude možné s istotou určiť, kedy tento stav zmeniť späť. Je preto nutné uchovávať ho v dátovom úložisku spoločne so všetkými premennými, ktoré jednotlivé komponenty menia v reakcii na dokončenie určitej operácie resp. akcie.

```
loading: boolean;
editMode: boolean;

categories: CustomerCategory[];
category: CustomerCategory;
```

Výpis kódu 4.1 Časť dátového úložiska - zákaznícke kategórie

Tieto komponenty pristupujú k vybraným častiam stavu dátového úložiska pomocou selektorov uvedených v ukážke 4.2. Prvý selektor je určený na poskytnutie celého výrezu resp. celku dátového úložiska, zatiaľ čo ostatné umožňujú pozorovať konkrétne premenné. Ako som už naznačil, návratovou hodnotou selektorov je vždy pozorovateľný objekt. Je možné a často tiež vhodné vytvoriť aj komplikovanejšie selektory, ktoré danú hodnotu pred navrátením upravia do ľubovoľnej podoby podľa konkrétnych potrieb komponentov, v tomto prípade to však nebolo nutné.

```
export const selectCustomersState =
    createFeatureSelector<CustomersState>('customers');

export const selectLoading = createSelector(
    selectCustomersState,
    (state: CustomersState) => state.loading
);

export const selectEditMode = createSelector(
    selectCustomersState,
    (state: CustomersState) => state.editMode
);

export const selectCustomerCategories = createSelector(
    selectCustomersState,
    (state: CustomersState) => state.categories
);

export const selectCustomerCategory = createSelector(
    selectCustomersState,
    (state: CustomersState) => state.category
);
```

Výpis kódu 4.2 Selektory zákazníckych kategórií

Ukážky 4.3 a 4.4 obsahujú všetky akcie, ktoré slúžia na manipuláciu časti stavu dátového úložiska uvedenej v 4.1. Ako je na prvý pohľad zrejmé, ide o veľké množstvo opakujúceho sa kódu, kvôli čomu som sa pri detailnejšom opise obmedzil na jeho pomerne malú časť. Za povšimnutie tiež stojí, že ku každej akcii vykonávajúcej jednu či viac asynchrónnych operácií bolo nutné implementovať ďalšie dve akcie reprezentujúce jej pozitívny a negatívny výsledok.

```
export const setCategories = createAction(
    '[Customer] Set Categories',
    props<{ categories: CustomerCategory[] }>()
);
export const getCategories = createAction(
    '[Customer] Fetch Categories'
);
export const getCategoriesSuccess = createAction(
    '[Customer] Fetch Categories Success',
    props<{ categories: CustomerCategory[] }>()
);
export const getCategoriesError = createAction(
    '[Customer] Fetch Categories Error'
);

export const setCategory = createAction(
    '[Customer] Set Category',
    props<{ category: CustomerCategory }>()
);
export const getCategory = createAction(
    '[Customer] Fetch Category',
    props<{ idCategory: number }>()
);
export const getCategorySuccess = createAction(
    '[Customer] Fetch Category Success',
    props<{ category: CustomerCategory }>()
);
export const getCategoryError = createAction(
    '[Customer] Fetch Category Error'
);
```

Výpis kódu 4.3 Akcie na získavanie a nastavovanie zákazníckych kategórií

U pridávania, mazania či aktualizácie jednotlivých entít som si musel zvoliť medzi optimistickým a pesimistickým prístupom aktualizácie ich zoznamu. Optimistický prístup spočíva v predpoklade, že vyslaný dotaz vráti nechybovú odpoveď. Spoločne s vyslaním dotazu je daná entita pridaná, aktualizovaná či zmazaná z kolekcie a v prípade, že nastane chyba, upravený zoznam sa vráti do pôvodnej podoby. Kvôli momentálnej architektúre aplikácie tento prístup však nie je možné využívať pri pridávaní entít a užívateľ je schopný mazať jedine zákaznícke kategórie. Zvolil som preto pesimistický prístup pre zachovanie konzistencie aj napriek tomu, že v porovnaní s optimistickým prístupom poskytuje o trochu horšiu užívateľskú skúsenosť.

```
export const addCategory = createAction(
    '[Customer] Add Category',
    props<{ category: CustomerCategory }>()
);
export const addCategorySuccess = createAction(
    '[Customer] Add Category Success',
```

```
    props<{ category: CustomerCategory }>()
);
export const addCategoryError = createAction(
    '[Customer] Add Category Error'
);

export const updateCategory = createAction(
    '[Customer] Update Category',
    props<{ category: CustomerCategory }>()
);
export const updateCategorySuccess = createAction(
    '[Customer] Update Category Success',
    props<{ category: CustomerCategory }>()
);
export const updateCategoryError = createAction(
    '[Customer] Update Category Error'
);

export const deleteCategory = createAction(
    '[Customer] Delete Category',
    props<{ idCategory: number }>()
);
export const deleteCategorySuccess = createAction(
    '[Customer] Delete Category Success',
    props<{ idCategory: number }>()
);
export const deleteCategoryError = createAction(
    '[Customer] Delete Category Error'
);
```

Výpis kódu 4.4 Akcie na pridávanie, aktualizáciu a mazanie zákazníckych kategórií

Namiesto ukážky časti reduktoru a všetkých efektov, ktoré reagujú na akcie súvisiace so zákaznickými kategóriami popíšem praktické využitie a životný cyklus akcií na pridanie novej a získanie konkrétnej kategórie. Ukážka 4.5 obsahuje všetky metódy a premenné v zákazníckej službe, ktoré sú v komponentoch pri snahe vyslať tieto akcie využívané.

Jednotlivé selektory boli priradené do samostatných premenných pre prípad, že by bolo nutné vykonať určité úpravy získaných častí stavu. V prípade kategórie musí byť vytvorená jej hlboká kópia, aby ju konkrétne komponenty mohli podľa svojich špecifických potrieb patrične upraviť.

Zaujímavá je ešte metóda na získanie konkrétnej kategórie, ktorá pred navrátením selektoru vyšle akciu pre tento účel. To spôsobí, že komponenty môžu na metódu reagovať rovnako ako pri jednoduchých volaniach metód klienta určeného na komunikáciu so servérovou časťou aplikácie, čo má za následok úplnú izoláciu logiky spojenej s manipuláciou dátového úložiska. Komponent si tak nie je vôbec vedomý, že logika získavania konkrétnej kategórie sa výrazne odlišuje od získavania špecifickej zákazky. To aspoň v malej miere prispieva k zníženiu inkonzistencií spojených s využívaním služieb, ktoré boli implementáciou zasiahnuté.

Za povšimnutie tiež stojí nutnosť naklonovať kategóriu pred vyslaním akcie na jej pridanie. Dôvodom je, že NgRx sprístupní každý objekt vyslaný do akcie iba na čítanie, aby sa zabránilo jeho zmenám pred dokončením akcie. Takto vyslaný objekt vo výsledku nie je možné meniť ani po jej dokončení, kvôli čomu je nutné vytvoriť jeho hlbokú kópiu.

```
private loading$ = this.store.pipe(
    select(selectLoading)
);
private category$ = this.store.pipe(
    select(selectCustomerCategory),
```

```
    map((category) => clone(category)),
)
getCategory(idCategory: number): Observable<CustomerCategory> {
    this.store.dispatch(
        CustomersActions.getCategory({ idCategory })
    );
    return this.category$;
}
addCategory(originalCategory: CustomerCategory) {
    const category = clone(originalCategory);
    this.store.dispatch(
        CustomersActions.addCategory({ category })
    );
}
```

Výpis kódu 4.5 Vybrané metódy zákazníckej služby

Po vyvolaní metódy zákazníckej služby pre získanie konkrétnej kategórie sa najprv vyšle akcia rovnakej funkcionality a následne sa vráti selektor, ktorý komponentu danú kategóriu dodá z dátového úložiska a bude ho informovať o jej zmenách. Je potrebné myslieť na fakt, že táto kategória nemusí byť inicializovaná, resp. jej hodnota nemusí byť aktuálna, o čo sa postará daná akcia. Ako je možné vidieť v ukážke 4.6, po vyslaní akcie sa do premennej reprezentujúcej kategóriu v dátovom úložisku zapíše hodnota, ktorá bola nájdená v kolekcii všetkých kategórií alebo nedefinovaná hodnota v prípade, že ju daná kolekcia neobsahuje. Následne je akcia spracovaná efektom, ktorý je pre ňu určený. V ukážke 4.7 je obsiahnutá konkrétna logika, ktorá spočíva vo vyslaní dotazu na servérovú časť aplikácie a na základe jej výsledku vyvolá akciu reprezentujúcu úspech či chybu. Pri úspechu sa do dátového úložiska zapíše hodnota vrátená servérovou časťou a pri chybe nedefinovaná hodnota spolu so zobrazením chybovej hlášky. Zároveň dôležitým vedľajším efektom je zvýšenie či zníženie hodnoty, ktorá slúži sa grafické znázornenie prebiehajúceho načítavania užívateľom, ktoré však nie je blokujúce.

```
on(getCategory, (state, action) => {
    return {
        ...state,
        category: state.categories.find(
            (category) => category.id === action.idCategory
        ),
    };
}),
on(getCategorySuccess, (state, action) => {
    return {
        ...state,
        category: action.category,
    };
}),
on(getCategoryError, (state, action) => {
    return {
        ...state,
        category: undefined,
    };
}),
```

Výpis kódu 4.6 Časť reduktoru súvisiaca so získaním kategórie

```
getCategory$ = createEffect(() => this.actions$.pipe(
    ofType(getCategory),
    tap(() => {
```

```
        this.authService.progressBarIncrement()
    }),
    mergeMap(({ idCategory }) =>
        this.customersAPI.getCategory(idCategory).pipe(
            map((category) => getCategorySuccess({ category })),
            catchError(() => of(getCategoryError())),
        )
    ),
));
getCategorySuccess$ = createEffect(() => this.actions$.pipe(
    ofType(getCategorySuccess),
    mergeMap(({ category }) => of(setCategory({ category }))),
    tap(() => {
        this.authService.progressBarDecrement();
    }),
));
getCategoryError$ = createEffect(() => this.actions$.pipe(
    ofType(getCategoryError),
    tap(() => {
        this.snackBar.open(
            'Chyba pri nacitani kategorie',
            'Zavrit',
            { duration: 10000 }
        );
        this.authService.progressBarDecrement();
    }),
), { dispatch: false });
```

Výpis kódu 4.7 Efekty súvisiace so získaním kategórie

O niečo zaujímavejšia je akcia slúžiaca na pridanie novej kategórie. Po jej vyslaní sa v dátovom úložisku nastaví na pravdivú hodnotu premenná reprezentujúca načítavanie, odošle sa dotaz na servérovú časť a pri odpovedi sa vyvolajú patričné akcie. V prípade neúspechu sa nebude diať nič zvláštne. Zobrazí sa chybová hláška, zruší sa načítavanie a hodnota kategórie sa zmení na nedefinovanú. Pri úspechu sa v dátovom úložisku zmení hodnota načítavania, módu úprav a kategória obdržaná zo servérovej časti sa pridá do kolekcie všetkých kategórií a zapíše do premennej predstavujúcej vybranú z nich. Ako vedľajší efekt sa zobrazí hláška potvrdzujúca úspech operácie, zruší sa vizuálna časť načítavania a presmeruje užívateľa na adresu tejto kategórie.

```
on(addCategory, (state, action) => {
    return {
        ...state,
        loading: true,
    };
}),
on(addCategorySuccess, (state, action) => {
    return {
        ...state,
        loading: false,
        editMode: false,
        categories: [...state.categories, action.category],
        category: action.category
    };
}),
on(addCategoryError, (state, action) => {
    return {
        ...state,
```

```
        loading: false,
        category: undefined,
    };
}),
```

Výpis kódu 4.8 Časť reduktoru súvisiaca so pridávaním kategórie

```
addCategory$ = createEffect(() => this.actions$.pipe(
    ofType(addCategory),
    tap(() => {
        this.authService.progressBarIncrement()
    }),
    mergeMap(({ category }) =>
        this.customersAPI.addCategory(category).pipe(
            map((category) => {
                category.count = 0;
                return addCategorySuccess({ category });
            }),
            catchError(() => of(addCategoryError())),
        )
    ),
));
addCategorySuccess$ = createEffect(() => this.actions$.pipe(
    ofType(addCategorySuccess),
    tap(({ category }) => {
        this.router.navigateByUrl('/kategorie/' + category.id);
        this.snackBar.open(
            'Kategorie ulozena',
            'Zavrit',
            { duration: 5000 }
        );
        this.authService.progressBarDecrement();
    }),
), { dispatch: false });
addCategoryError$ = createEffect(() => this.actions$.pipe(
    ofType(addCategoryError),
    tap(() => {
        this.snackBar.open(
            'Nastala chyba pri pridavani kategorie',
            'Zavrit',
            { duration: 10000 }
        );
        this.authService.progressBarDecrement();
    })
), { dispatch: false });
```

Výpis kódu 4.9 Efekty súvisiace s pridaním kategórie

Zaujímavú funkcionalitu tiež prináša celok dátového úložiska reprezentujúci stav pokladničného terminálu. Objekt reprezentujúci všetky produkty v termináli spoločne s celkovou sumou týchto produktov musia byť pri každej ich zmene uložené do lokálneho úložiska, aby bol ich stav perzistentný aj pri opätovnom načítaní aplikácie. Každá akcia schopná meniť tento stav musí ako vedľajší efekt vyvolať akcie na jeho uloženie a pri zavolaní metód zo služby určených na získanie častí tohto stavu musí byť vyvolaná akcia na ich načítanie. Spomínané akcie nijak nemenia stav dátového úložiska v reduktore, ale ako vedľajší efekt si z neho opatria aktuálnu hodnotu, ktorú následne uložia, resp. získajú hodnotu z lokálneho úložiska, ktorú nastavia, ako znázorňuje ukážka 4.10.

```
saveTerminalList$ = createEffect(() => this.actions$.pipe(
    ofType(saveTerminalList),
    concatLatestFrom(() => this.store.pipe(
        select(selectTerminalList)
    )),
    tap(([action, terminalList]) => {
        localStorage.setItem(
            'terminalList',
            JSON.stringify(terminalList)
        );
    }),
), { dispatch: false });
loadTerminalList$ = createEffect(() => this.actions$.pipe(
    ofType(loadTerminalList),
    mergeMap(() => {
        const data = localStorage.getItem('terminalList');
        return of(setTerminalList({
            terminalList: data  ? JSON.parse(data)
                                : initialState.terminalList
        }));
    }),
));

saveTerminalSum$ = createEffect(() => this.actions$.pipe(
    ofType(saveTerminalSum),
    concatLatestFrom(() => this.store.pipe(
        select(selectTerminalSum)
    )),
    tap(([action, terminalSum]) => {
        const data = localStorage.setItem(
            'terminalSum',
            JSON.stringify(terminalSum)
        );
    })
), { dispatch: false });
loadTerminalSum$ = createEffect(() => this.actions$.pipe(
    ofType(loadTerminalSum),
    mergeMap(() => {
        let data = localStorage.getItem('terminalSum');
        if(data) {
            const terminalSum = parseFloat(data);
            return of(
                                setTerminalSum({
                    terminalSum: isNaN(terminalSum) ? 0
                                                    : terminalSum
                })
            );
        }
        return EMPTY;
    }),
));
```

Výpis kódu 4.10 Efekty slúžiace na načítanie a uloženie stavu terminálu

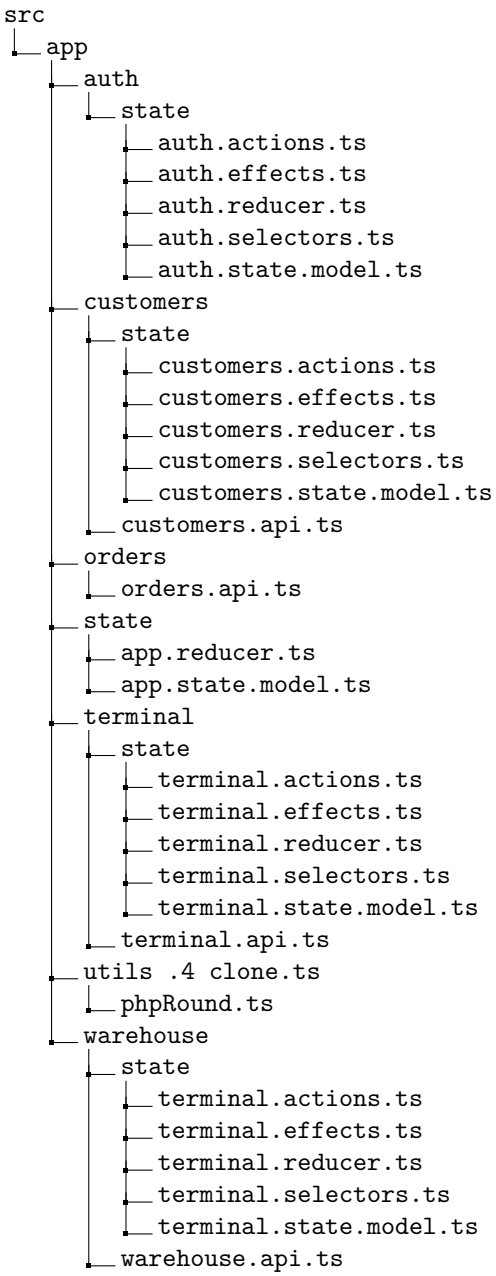
Implementácia navrhnutého riešenia však kvôli časovej náročnosti jeho integrácie neprebehla v plnom rozsahu. Ide konkrétne o celok dátového úložiska spojený s autorizáciou a autentikáciou užívateľov. Vyhodnotil som to ako časť, ktorá zlepší užívateľskú či vývojársku skúsenosť v naj-

menšej miere, a tak sa od jej integrácie upustilo v prospech celkov, ktoré som považoval za dôležitejšie aj napriek už existujúcemu kódu.

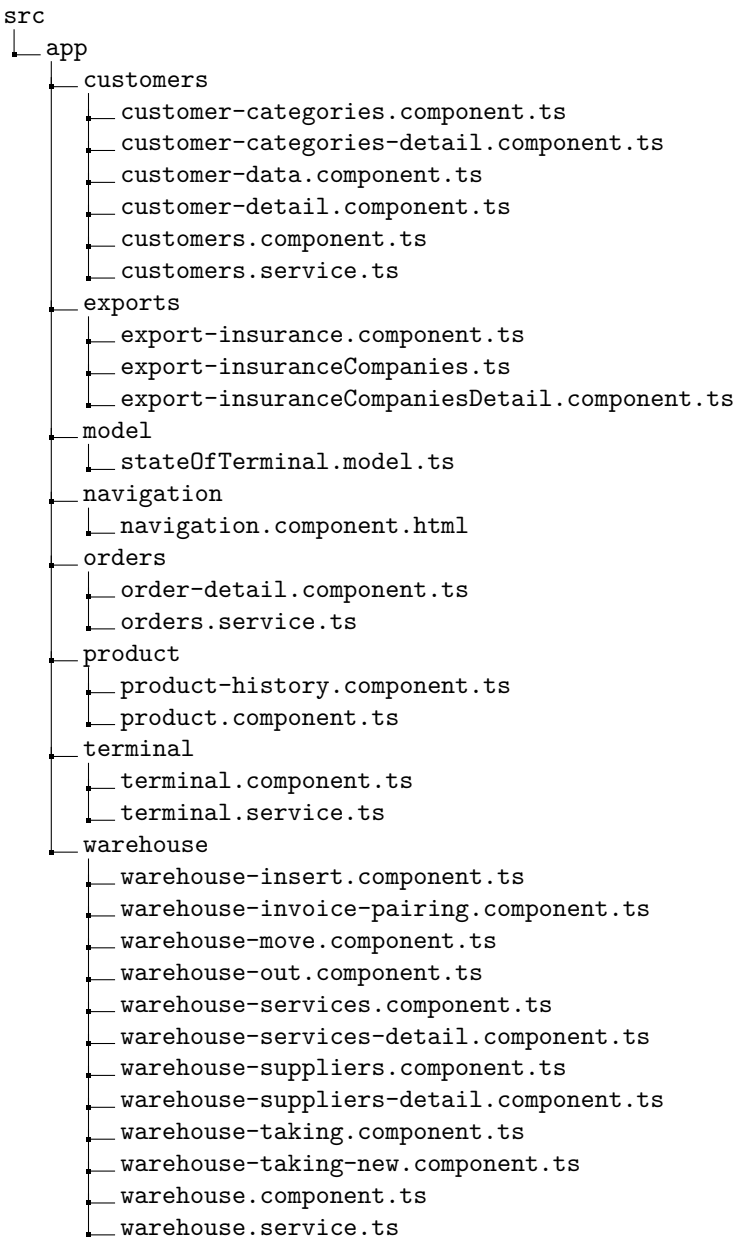
4.3 Nasadenie riešenia

Jedným z cieľov mojej práce je príprava na nasadenie do produkcie a samotné nasadenie. Nakoľko systém už v produkcii beží pomerne dlhú dobu, budem vykonávať len prípravu súvisiacu s integráciou zvolenej knižnice pre správu stavu, ktorá neobnáša nič zvláštne či komplikované. V prvom rade je nutné dôkladne otestovať všetky časti aplikácie, kde nastala zmena mojim pričinením, aby sa zaručila identická funkčnosť systému s jeho pôvodnou verziou. Následne je potrebné podrobne prejsť všetok kód a odstrániť z neho zbytočné komentáre a kód určený výhradne pre ladenie počas implementácie, ktorý tam mohol ostať. Počas toho tiež bude venovaný dôraz na identifikáciu častí riešenia, ktoré by mohli pôsobiť neprehľadne, komplikovane či chaoticky a patrične ich okomentovať, aby sa predišlo zmäteniu a spomaleniu pri budúcom vývoji. Nakoniec už stačí riešenie pridať do samostatnej vetvy repozitára aplikácie, kde bude čakať na samotné nasadenie.

V čase písania tejto práce však realizované riešenie do produkcie ešte nasadené nebolo, pretože podľa informácií, ktoré som obdržal od vedúceho práce, bude nasadenie môjho riešenia spojené s rozsiahlejšou aktualizáciou systému. V dobe obhajoby by však toto riešenie už malo bežať vo všetkých optikách využívajúcich daný systém.



Obr. 4.1 Štruktúra pridaných súborov do projektu



Obr. 4.2 Štruktúra súborov upravených počas integrácie riešenia

Kapitola 5

Testovanie

Náplňou tejto kapitoly je priblíženie spôsobu testovania realizovaného riešenia a zhodnotenie vplyvu daných úprav na systém na základe jeho výsledkov. Je naivné domnievať sa, že pri vývoji nedôjde k žiadnym chybám. Niektoré z nich môžu byť celkom zanedbateľné, bez reálneho vplyvu na používanie systému, zatiaľ čo iné vedia jeho používanie úplne znemožniť, čím spôsobia obrovské škody ako zákazníkovi, tak aj spoločnosti, ktorá daný systém spravuje. Testovanie je preto veľmi dôležitou súčasťou vývoja akéhokoľvek softvéru.

5.1 Funkčné testovanie

Pre overenie, že sa systém po realizovaných úpravách správa rovnako ako jeho predošlá verzia je vhodné vykonať jednotkové a integračné testovanie.

Úlohou jednotkového testovania je overiť správnosť fungovania jeho najmenších izolovaných častí. Zväčša ide o jednu z prvých fáz testovania, ktorá umožní včasné odhalenie chýb, ktorých identifikácia by neskôr mohla byť výrazne náročnejšia [51]. Jednotkové testovanie v tomto prípade zahŕňa overiť správnosť získaného stavu pomocou selektorov, korektnosť reduktorov v spôsobe, akým spracúvajú jednotlivé akcie a či na základe ich typu vhodne upravujú stav a zaistiť, že všetky akcie sú správne definované, obsahujú potrebné údaje a metadáta a vyvolávajú príslušné vedľajšie účinky.

Integračné testy kontrolujú správnosť interakcie jednotlivých častí aplikácie pri vykonávaní zložitejších úkonov. Zaisťujú ich kompatibilitu, overujú bezchybnosť ich komunikácie a zvyšujú celkovú spoľahlivosť a kvalitu systému [52]. Konkrétne by mali preverovať správnosť stavu získaného z dátového úložiska v komponentoch a jeho zmien po uskutočnení možných akcií.

V čase písania tejto práce boli obe fázy funkčného testovania vykonané iba manuálne s využitím vývojárskych nástrojov určených pre ladenie zmien stavu založených na podobnej architektúre, akú prináša návrhový vzor Redux a častým vypisovaním do konzole. Z toho dôvodu bol kladený veľký dôraz na dôkladné otestovanie funkčnosti a identického správania systému oproti jeho minulej verzii počas celej realizácie návrhu a po jej dokončení vo všetkých častiach aplikácie, ktoré boli v dôsledku integrácie riešenia upravené.

Bolo by však vhodné automatizovať aspoň jednotkové testy, pretože vďaka využívaniu čistých funckií, izolácii vedľajších efektov a poskytnutým nástrojom je jednotkové testovanie priamočiare a pomerne jednoduché [42], nehovoriac o početných výhodách, ktoré prináša.

5.2 Testovanie užívateľskej a vývojárskej skúsenosti

Aby bolo možné zhodnotiť vplyv mnou realizovaných úprav na aplikáciu, je nutné vykonať testy, ktoré sa na to špecificky zameriavajú. Ako napovedá názov tejto sekcie, testovaný bude vplyv integrácie zvoleného riešenia na vývojársku a užívateľskú skúsenosť. Testy budú založené na porovnaní daných skúseností subjektmi, ktoré im poskytne pôvodná verzia systému a verzia, v ktorej je riešenie realizované. Výsledky týchto testov však z princípu nemôžu byť objektívne, keďže hodnotenie kvality užívateľskej a vývojárskej skúsenosti je výlučne subjektívna záležitosť.

Vývojársku skúsenosť budú posudzovať štyria kolegovia, s ktorými som OptiLynx vyvíjal v rámci predmetu softvérový tímový projekt, pričom každý z nich mal možnosť pracovať s klientskou časťou systému. Najväčšiu váhu však budem pripisovať názoru tých, ktorí sa zameriavali na jej vývoj alebo sa tomu odvtedy aktívne venujú. Všetci z nich obdržia obe verzie kódu a na základe ich spätnej väzby bude zhodnotený vplyv úprav na vývojársku skúsenosť, prípadne bude riešenie dodatočne upravené.

Vzhľadom na nemožnosť uskutočniť užívateľské testovanie s reálnymi koncovými užívateľmi systému ho budem musieť vykonať taktiež za pomoci kolegov. Bude založené primárne na bežných prípadoch užitia uvedených v sekcii 3.2 rozšírené o viaceré úkony, ktoré budú simulovať chyby spôsobené užívateľmi pri jednotlivých procesoch. Aj v tomto prípade budú dané procesy vykonávané na oboch verziách systému a následne porovnaná užívateľská skúsenosť, ktorú každá z nich poskytuje.

5.3 Zhodnotenie vplyvu úprav na systém

Po dokončení testovania všetkými kolegami je na základe ich spätnej väzby možné o niečo objektívnejšie zhodnotiť vplyv realizácie riešenia na užívateľskú a vývojársku skúsenosť.

V prípade užívateľskej skúsenosti sa každý zhodne vyjadril, že pri vykonaných scenároch, ktoré sa od bežných prípadoch užitia líšili len minimálne, zlepšenie skutočne nastalo, a to vo výraznej miere. Aplikácia je po integrácii stavového kontajnera pri danom používaní omnoho reaktívnejšia a plynulejšia, v dôsledku čoho sa javí oveľa výkonnejšie ako predtým. V skutočnosti je však opak pravdou kvôli nutnosti vytvárať hlboké kópie veľkého počtu entít. To sa ale deje na pozadí, vďaka čomu si to užívateľ častokrát nemusí ani všimnúť. Realizácia riešenia pre správu stavu so sebou neoddeliteľne priniesla vyššie nároky aplikácie na pamäť. Dané zvýšenie je však vzhľadom na dnešnú dobu zanedbateľné a neočakávam, že bude v blízkej dobe predstavovať problém ani pri ďalšom rozšírení dátového úložiska.

Vývojárska skúsenosť obstála v porovnaní s tou užívateľskou výrazne horšie. Pozitívnym prínosom úprav na systém bolo presunutie pomerne veľkého množstva obchodnej logiky preč z komponentov, čím došlo k ich zjednodušeniu, zavedenie predvídateľného spôsobu správy stavu, čo viedlo k uľahčeniu testovania a odstráneniu kurióznych a chaotických riešení relatívne jednoduchých problémov, ktoré zhoršovali dojem z celého zdrojového kódu aplikácie. Integrácia NgRx so sebou však priniesla vyššiu komplexitu, inkonzistencie vo využívaní jednotlivých služieb a v neposlednom rade obrovské množstvo štandardného kódu. Aj napriek mojej úprimnej snahe spríjemniť vývoj aplikácie však musím na základe spätnej väzby kolegov a osobnej skúsenosti skonštatovať, že k zlepšeniu vývojárskej skúsenosti nedošlo, ba naopak je možné namietať, že došlo k jej zhoršeniu.

Ako už bolo spomenuté v návrhu, integrácia stavového kontajnera nie je strieborná guľka, no aj napriek nevýhodám, ktoré priniesla ju považujem za prospešnú zmenu. Je dôležité, aby aplikácie boli orientované na užívateľov, pretože ich úspech sa do veľkej miery odvíja od kvality skúsenosti, ktorú im pri používaní poskytnú.

Kapitola 6
Záver

Cieľom práce bolo pridať stavovosť do klientskej časti aplikácie vo forme stavového kontajnera.

Najprv prebehla dôkladná analýza zdrojového kódu aplikácie, výsledkom ktorej bola identifikácia a priblíženie jej nedostatkov. Práca tiež ukázala a stručne popísala často využívané riešenia pre správu stavu.

Na základe analýzy bolo vzhľadom na konkrétne potreby aplikácie zvolené vhodné riešenie pre správu stavu a vypracovaný podrobný návrh, ktorý kládol dôraz na zlepšenie užívateľskej a vývojárskej skúsenosti. Následne bola uskutočnená implementácia vypracovaného návrhu a bližší opis konkrétnej štruktúry riešenia, ako aj vybraných častí kódu, ktoré som považoval za dôležité či zaujímavé.

Integrované riešenie bolo počas celého vývoja manuálne testované a po jeho dokončení prebehlo testovanie užívateľskej a vývojárskej skúsenosti za pomoci kolegov, ktorí už mali možnosť aplikáciu vyvíjať a pracovať s ňou. Po uskutočnení testovania bol zhodnotený vplyv úprav na systém a opravené zistené nedostatky.

Nakoniec bola vykonaná príprava na nasadenie do reálnej prevádzky, ktoré však v čase písania tejto práce ešte uskutočnené nebolo.

Do budúcna by bolo vhodné zautomatizovať testovanie na úrovni jednotkových a integračných testov a rozdeliť dátové úložisko na menšie celky. Taktiež by stálo za zváženie dodatočné rozšírenie stavového kontajnera, ktorému by však mala predchádzať obsiahla refaktorizácia zdrojového kódu aplikácie. Jeho aktuálna podoba výrazne spomaľuje ďalší vývoj alebo úplne znemožňuje rozšírenie zvoleného riešenia spôsobom, ktorý by bolo možné považovať za prehľadné a ľahko rozšíriteľné.

Bibliografia

1. *What is Angular?* 2022. Dostupné tiež z: https://angular.io/guide/what-is-angular.
2. *Angular components overview.* 2022. Dostupné tiež z: https://angular.io/guide/component-overview.
3. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *Ergonomics of human-system interaction – Part 210: Human-centred design for interactive systems.* 2019. International Standard, ISO 9241-210. ISO. Dostupné tiež z: https://www.iso.org/standard/77520.html.
4. BAJARIN, Ben. *Apple and the User Experience Business Model.* iMore, 2015. Dostupné tiež z: https://www.imore.com/apple-and-user-experience-business-model.
5. DENITSA-KOSTOVA. *Why is UX design important - 5 important arguments.* denitsa-kostova, 2021. Dostupné tiež z: https://www.resolutesoftware.com/news/why-is-ux-design-important/.
6. *Why does site speed matter? — Improve webpage speed.* [B.r.]. Dostupné tiež z: https://www.cloudflare.com/learning/performance/why-site-speed-matters/.
7. WIGMORE, Ivy. *What is broken window theory?: Definition from TechTarget.* TechTarget, 2015. Dostupné tiež z: https://www.techtarget.com/whatis/definition/broken-window-theory.
8. *Angular best practices and security.* 2023. Dostupné tiež z: https://www.tatvasoft.com/blog/angular-optimization-and-best-practices/.
9. FREECODECAMP.ORG. *Best practices for a clean and performant angular application.* freeCodeCamp.org, 2019. Dostupné tiež z: https://www.freecodecamp.org/news/best-practices-for-a-clean-and-performant-angular-application-288e7b39eb6f/.
10. GOHIL, Jitendrasinh. *Understanding state management in front-end paradigm* [online]. 2020. [cit. 2023-02-10]. Dostupné z : https://www.linkedin.com/pulse/understanding-state-management-front-end-paradigm-jitendrasinh-gohil/.
11. GARG, Hitesh. *Understanding state management - angular* [online]. Byteridge, 2020 [cit. 2023-02-10]. Dostupné z : https://ideas.byteridge.com/angular-state-managment/.
12. SINGHAL, Gaurav. *Understanding the Purpose of a Component and Its Basic Parts* [online]. 2019. [cit. 2023-02-12]. Dostupné z : https://www.pluralsight.com/guides/understanding-purpose-component-and-basic-parts.
13. NIELSEN, Lars Gyrup Brink. *Container components with angular* [online]. DEV Community, 2021 [cit. 2023-02-10]. Dostupné z : https://dev.to/this-is-angular/model-view-presenter-with-angular-533h.

14. UNIVERSITY, Angular. *Angular smart components vs Presentational Components*. Angular University, 2023. Dostupné tiež z: https://blog.angular-university.io/angular-2-smart-components-vs-presentation-components-whats-the-difference-when-to-use-each-and-why/.
15. 2MUCHCOFFEE. *Angular State management: Ä must-have"for large scale angular apps!* Medium, 2019. Dostupné tiež z: https://medium.com/@2muchcoffee/angular-state-management-a-must-have-for-large-scale-angular-apps-8b98e5a761c7.
16. *Introduction*. [B.r.]. Dostupné tiež z: https://rxjs.dev/guide/overview.
17. *The RxJS library*. [B.r.]. Dostupné tiež z: https://angular.io/guide/rx-library.
18. *Using observables to pass values*. [B.r.]. Dostupné tiež z: https://angular.io/guide/observables.
19. ARORA, Chandermani; HENNESSY, Kevin; ULUCA, Doguhan; NORING, Christoffer. *Building Large-Scale Web Applications with Angular*. Birmingham: PACKT Publishing Limited, 2018. ISBN 978-1-78995-956-7.
20. SAEED, Shamaz. *A step-by-step guide to flux architecture with react*. Bits a Pieces, 2023. Dostupné tiež z: https://blog.bitsrc.io/a-step-by-step-guide-to-flux-architecture-with-react-8517017f89cf.
21. *Redux vs flux: Find out the10 important differences (with infographics)*. 2023. Dostupné tiež z: https://www.educba.com/redux-vs-flux/.
22. ABRAMOV, Dan et al. *Redux Essentials, part 1: Redux overview and concepts* [online]. 2022. [cit. 2023-02-13]. Dostupné z : https://redux.js.org/tutorials/essentials/part-1-overview-concepts.
23. ABRAMOV, Dan. *You might not need redux* [online]. Medium, 2016 [cit. 2023-02-13]. Dostupné z : https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367.
24. SHARMA, Radium. *ABC of Redux*. DEV Community, 2018. Dostupné tiež z: https://dev.to/radiumsharma06/abc-of-redux-5461.
25. TALIN. *Introduction to reactive programming using MobX*. Machine Words, 2019. Dostupné tiež z: https://medium.com/machine-words/introduction-to-reactive-programming-using-mobx-2c032cac818e.
26. *The gist of MobX · mobx*. [B.r.]. Dostupné tiež z: https://mobx.js.org/the-gist-of-mobx.html.
27. AU-YEUNG, John. *How to use mobx for easy state management in your angular app* [online]. Level Up Coding, 2019 [cit. 2023-02-13]. Dostupné z : https://levelup.gitconnected.com/how-to-use-mobx-for-easy-state-management-in-your-angular-app-78e482407da8.
28. *Core concepts*. 2022. Dostupné tiež z: https://mobx.netlify.app/concepts/.
29. BASAL, Netanel. *introducing akita: A new state management pattern for angular applications*. Netanel Basal, 2021. Dostupné tiež z: https://netbasal.com/introducing-akita-a-new-state-management-pattern-for-angular-applications-f2f0fab5a8.
30. JUAN, William. *State management in angular using akita pt 1*. 2022. Dostupné tiež z: https://auth0.com/blog/state-management-in-angular-with-akita-1/.
31. BACHUK, Alex. *Redux · an introduction*. 2016. Dostupné tiež z: https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/.
32. *Introduction to NgRx and its implementation area in project*. [B.r.]. Dostupné tiež z: https://wearecommunity.io/communities/india-united-ui-community/articles/1476.

33. *Angular State management with NGXS: Loginradius blog.* [B.r.]. Dostupné tiež z: https://www.loginradius.com/blog/engineering/angular-state-management-with-ngxs/.
34. XIONG, Emily. *My developer experience with NGXS.* Medium, 2021. Dostupné tiež z: https://emilyxiong.medium.com/my-developer-experience-with-ngxs-9da7431073f4.
35. *What is a boilerplate code?* 2023. Dostupné tiež z: https://www.educative.io/answers/what-is-a-boilerplate-code.
36. PRANSKUNAS, Vytautas. *Performance comparison of state management solutions in react.* Medium, 2019. Dostupné tiež z: https://medium.com/@vpranskunas/performance-comparison-of-state-management-solutions-in-react-8aee5ae15b3c.
37. *Mobx vs redux – which solution is better?* 2022. Dostupné tiež z: https://codete.com/blog/redux-vs-mobx-everything-you-need-to-know.
38. RAVICHANDRAN, Adhithi. *Redux vs. MobX: Which performs better?* 2021. Dostupné tiež z: https://blog.logrocket.com/redux-vs-mobx/.
39. *React State Management: Redux or Akita?* 2022. Dostupné tiež z: https://initechglobal.com/2021/09/01/react-state-management-redux-or-akita/.
40. BELGIUM, Ordina. *NGRX vs. NGXS vs. akita vs. rxjs: Fight!* 2018. Dostupné tiež z: https://ordina-jworks.github.io/angular/2018/10/08/angular-state-management-comparison.html#fight.
41. KRISTINA DŽENOPOLJAC, Mid Software Developer. *State management in Angular Applications – NGXS vs. NGRX.* Serengeti, 2021. Dostupné tiež z: https://serengetitech.com/tech/state-management-in-angular-applications-ngxs-vs-ngrx/.
42. *NgRx docs: Why use Store?* 2023. Dostupné tiež z: https://ngrx.io/guide/store/why.
43. GHOSH, Tushar. *Angular performance improvement using change detection-default and onpush.* Medium, 2021. Dostupné tiež z: https://tusharghosh09006.medium.com/angular-performance-improvement-using-change-detection-default-and-onpush-943916f7e39a.
44. *NgRx docs: Getting Started.* 2023. Dostupné tiež z: https://ngrx.io/guide/store.
45. *NgRx docs: Actions.* 2023. Dostupné tiež z: https://ngrx.io/guide/store/actions.
46. *NgRx docs: Reducers.* 2023. Dostupné tiež z: https://ngrx.io/guide/store/reducers.
47. *NgRx docs: Selectors.* 2023. Dostupné tiež z: https://ngrx.io/guide/store/selectors.
48. *NgRx docs: Effects overview.* 2023. Dostupné tiež z: https://ngrx.io/guide/effects.
49. OSIFESO, Peter. *NGRX: Is it worth it?* Web Factory LLC, 2018. Dostupné tiež z: https://medium.com/web-factory-llc/ngrx-is-it-worth-it-6ad9585dcbaa.
50. HRACH, Bc. Jaroslav. *Optilynx frontend - Informační systém pro optiky.* 2018. Dipl. pr. České vysoké učení technické v Praze.
51. *Unit testing: A detailed guide.* 2022. Dostupné tiež z: https://www.browserstack.com/guide/unit-testing-a-detailed-guide.
52. *Integration testing: A detailed guide.* 2023. Dostupné tiež z: https://www.browserstack.com/guide/integration-testing.

Obsah přiloženého média

- `src`
  - `FrontLynx` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF