FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Analýza a optimalizace UX nástroje pro podporu tvorby odhadů pracnosti
Student:	Bc. Petr Panský
Vedoucí:	Ing. Michal Petřík
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

1. Seznamte se se současným procesem a nástroji pro odhad SW metrik zadavatele, zaměřte se na použitelnost a využití současných nástrojů. Seznamte se s předchozími pracemi adresujícími shodné téma.
2. Proveďte analýzu a zhodnocení nedostatků současných nástrojů z pohledu UX, identifikujte klíčové nedostatky z pohledu použitelnosti a využitelnosti nástrojů.
3. Definujte potřebné kroky, které povedou k eliminaci identifikovaných nedostatků (předpokládá se testování s reálnými uživateli).
4. Na základě předchozího kroku navrhněte novou architekturu a výběr technologií, které zajistí naplnění definovaných kroků.
5. Implementujte navržená vylepšení ve zvolených technologiích včetně testů (minimálně unit).
6. Nově vytvářené části zdokumentujte. Na vzorku reálných uživatelů zhodnoťte přínosy realizovaných vylepšení.
7. Zhodnoťte výhody a nevýhody vašeho řešení vzhledem k řešení původnímu.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 16. září 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Analýza a optimalizace UX nástroje pro podporu tvorby odhadů pracnosti

Bc. Petr Panský

Katedra softwarového inženýrství
Vedoucí práce: Ing. Michal Petřík

7. ledna 2021

Poděkování

Rád bych poděkoval vedoucímu této práce Ing. Michalu Petříkovi za jeho pomoc, hodnotné rady a ochotu, se kterou diplomovou práci vedl. Rád bych také poděkoval společnosti Profinit EU, s.r.o. za možnost realizace této práce. V neposlední řadě bych rád poděkoval mé rodině a přátelům, kteří mě během studia podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. ledna 2021 .....

Odkaz na tuto práci

Panský, Petr. *Analýza a optimalizace UX nástroje pro podporu tvorby odhadů pracnosti*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá zhodnocením současných řešení pro tvorbu odhadů softwarových projektů ve společnosti Profinit EU, s.r.o. Cílem zhodnocení je odhalení nedostatků těchto řešení, návrh a realizace nového nástroje pro tvorbu odhadů. Nejdříve je v práci představena obecná metodika tvorby odhadů spolu s metodikou používanou uvnitř společnosti. Následně se práce věnuje analýze současných řešení pro tvorbu odhadů a identifikaci jejich nedostatků. Na základě provedené analýzy následuje návrh a realizace nového nástroje Estimate, v rámci čehož jsou eliminovány chyby ze současných řešení a jsou řešeny i nové požadavky na aplikaci. Vývoj nástroje je rozdělen do několika iterací obsahujících všechny disciplíny softwarového inženýrství, včetně přímého testování koncovými uživateli pro zajištění maximální zpětné vazby. V závěru se práce zaměřuje na zhodnocení vytvořeného nástroje a představuje plán budoucího rozvoje, který bude pokračovat i po ukončení této diplomové práce.

Klíčová slova odhad pracnosti softwarového projektu, tvorba odhadu, softwarový projekt, kužel nejistoty, GitLab, Continuous Integration, UX, Kotlin, Spring Boot, Gradle, React, Typescript, MongoDB, Material-UI, Excel

Abstract

This thesis focuses on the assessment of current solutions for creating software project estimates in the company Profinit EU, s.r.o. The aim of the assessment is to identify the weaknesses of these solutions, design, and realization of a new tool for creating estimates. At first, the thesis presents a general methodology of estimates creation together with the methodology used within the company. Then, the thesis continues with the analysis of the current solutions for making estimates and identification of their weaknesses. The performed analysis serves as the basis for the design and realization of the new Estimate tool. The realization phase of the tool also deals with bugs in the current solutions and implements new application requirements. The development of the tool is divided into several iterations that cover all disciplines of software engineering. To get maximal feedback each iteration also includes real end-user testing. The last chapters of the thesis focus on the assessment of the created tool and introduce the plan of future development that will continue after the completion of this master's thesis.

Keywords software project effort estimation, estimate creation, software project, cone of uncertainty, GitLab, Continuous Integration, UX, Kotlin, Spring Boot, Gradle, React, Typescript, MongoDB, Material-UI, Excel

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešním světě je velké množství softwarových projektů vytvářeno podle konkrétních představ zákazníků. Ať už se jedná o zakázkový vývoj nebo o úpravu existujícího řešení, na projekt jsou zákazníkem kladeny specifické požadavky. Jednou z činností softwarového procesu, která je typicky klíčová pro celkový úspěch projektu, je schopnost odhadování pracnosti daného projektu. Jak ve své knize říká Steve McConnell, „Dobrý odhad je odhad, který poskytuje dostatečně jasný pohled na realitu projektu, aby umožnil vedení projektu dělat dobrá rozhodnutí ohledně toho, jak řídit projekt tak, aby dosáhl svých cílů.“[1, s. 14, překlad vlastní]

Tato práce je tvořena ve spolupráci se softwarovou firmou Profinit EU, s.r.o., která v rámci svých projektů a odpovídajících aktivit softwarového procesu taktéž vychází ze schopnosti tvorby kvalitních odhadů pracnosti. V současné době se pro jejich tvorbu využívá program Excel s předvyplněnou šablonou a metodika, která říká, jak odhady vytvářet. Pomocí této metodiky jsou ve firmě vytvářeny všechny odhady, přičemž každý takto vytvořený odhad je ukládán ve formě souboru v interním verzovacím systému, aby byl kdykoliv dohledatelný. Tento proces odhadování funguje ve firmě již několik let. Poslední dobou se však toto řešení ukazuje jako ne příliš efektivní. Odhady nejsou centrálně ve verzovacím systému ukládány na jednom místě a existuje již několik odlišných verzí Excel šablony odhadu. Odhady proto nemají vždy jednotnou strukturu.

Tato problematika se již v minulosti adresovala, nicméně se nepodařilo dosáhnout zdárného řešení, které by uživateli bylo používáno. Výsledný nástroj pro tvorbu odhadů byl kvůli nedostatkům v uživatelském rozhraní pro zaměstnance hůře použitelný. Práce v nástroji byla ve výsledku náročnější než řešení pomocí programu Excel a nástroj se po čase přestal úplně používat. Z tohoto důvodu padlo rozhodnutí restartovat celý projekt tvorby nástroje pro odhady, začít jeho vývoj od úplného začátku a při realizaci se zaměřit více na koncové uživatele. Myšlenka projektu se mi líbila a působila zajímavě, proto jsem se rozhodl zvolit toto téma.

Cíl práce

Tato práce se zabývá analýzou současného procesu a nástrojů pro tvorbu odhadů ve firmě Profinit EU, s.r.o. Dále také návrhem, implementací a ověřením nově navrženého řešení tvorby odhadů s reálnými uživateli.

Nejdříve se zaměřím na analýzu současného procesu tvorby odhadu ve firmě. Poté zanalyzuji řešení, která jsou pro tvorbu odhadů v současné chvíli ve firmě používána. V rámci jejich analýzy se zaměřím i na jejich UX vlastnosti.

Po této analýze přejdu k návrhu vlastního nástroje, který tvorbu odhadů uživatelům zpříjemní. Nejdříve si zvolím, jaký přístup k vývoji aplikace zaujmu. Protože se pravděpodobně bude jednat o jistou formu iterativního vývoje, bude mít zvolený přístup vliv na členění výsledné práce. Poté na základě zvoleného přístupu zdokumentuji celý proces od analýzy až po návrh, realizaci a samotné testování tohoto nástroje v reálném provozu. V závěru se zaměřím na plán budoucího rozvoje.

Kapitola 1

Seznámení se s doménou odhadů

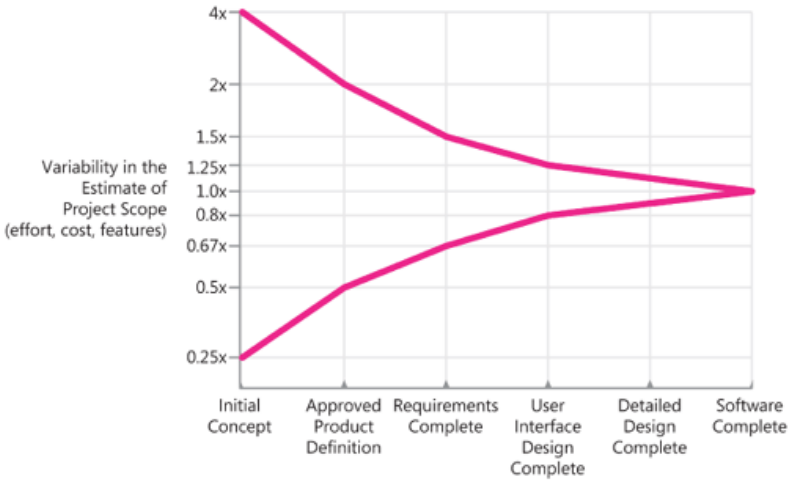
1.1 Odhad a jeho význam

Co tedy přesně samotný odhad znamená? Podle slovníkové definice, kterou uvádí Steve McConnell ve své knize, „je odhad předpověď, jak dlouho bude projekt trvat nebo kolik bude stát.“[1, s. 3, překlad vlastní] Dále Steve McConnell ve své knize píše [1, s. 3], že pod pojmem odhad si čtenář může představit i následující tři body:

- orientační hrubá kalkulace
- předběžná kalkulace nákladů projektu
- posouzení založené na dojmech či názoru člověka

Jak by se mohlo na první pohled podle definice zdát, odhad neslouží pouze k určení, jak dlouho projekt potrvá nebo kolik bude stát. Slouží také dodavateli k tomu, aby si o celém projektu udělal představu, tedy usoudil, jak je projekt velký a jakého je rozsahu. Dále odhad souvisí i se samotným projektovým plánováním. Projektové plánování do velké míry vychází právě ze samotného odhadu. V čem se ale liší, je, že „odhad by měl být považován za nezaujatý, analytický proces, kdežto plánování by mělo být považováno za zaujatý proces usilující o dosažení cíle.“[1, s. 4, překlad vlastní] V rámci plánování totiž ovlivňujeme cíleně plány, aby bylo dosaženo konkrétních cílů.

V rámci projektového plánování se stává, že projektoví manažeři často nacházejí mezeru mezi obchodními cíli projektu, jeho odhadovaným harmonogramem a náklady. Pokud jsou mezery dostatečně malé, jsou schopni je vyřešit v rámci projektového řízení. Pokud jsou mezery velké, je nutné cíle projektu přehodnotit.



Obrázek 1.1: Kužel nejistoty [1, s. 36]

„Primárním účelem odhadu softwaru tedy není předpovědět výsledek projektu. Jeho cílem je zjistit, zda jsou cíle projektu dostatečně realistické, aby umožnily na projektu kontrolu jejich splnění.“[1, s. 13, překlad vlastní]

1.2 Kužel nejistoty

Protože na softwarový projekt během celého jeho procesu působí různé aspekty z několika stran, jsme nuceni během jeho vývoje provádět velké množství rozhodnutí týkajících se výsledného produktu. Díky těmto rozhodnutím se projekt neustále mění a vyvíjí. Tyto změny proto mají kromě dopadu na samotný projekt i dopad na odhad pracnosti projektu. Tato problematika byla zpracována ve vědecké studii. Ta zjistila, že odhad pracnosti projektu má v různých fázích předvídatelné množství nejistoty. Tento jev se nazývá kužel nejistoty. [1, s. 36]

Kužel nejistoty je vyobrazen na Obrázku 1.1. Na horizontální ose se vyskytují jednotlivé milníky projektu a vertikální osa ukazuje míru chyby. Jak je vidět, odhady, které byly vytvořeny v počáteční fázi projektu, mohou vykazovat větší chybovost. V počátcích se tak může odhadovaná pracnost projektu lišit až čtyřnásobně od reálné pracnosti. Odhad tak může být 4x větší (přeceněný) oproti realitě, ale i 4x menší než realita (podceněný).

1.3 Termíny v odhadování

Při tvorbě odhadů se nevyhneme termínům používaným v této doméně. Mezi nejznámější termíny, které pravděpodobně budeme znát i z jiných disciplín softwarového inženýrství (např. projektové plánování, ...), patří jednotky

pracnosti, ve kterých jsou odhady vytvářeny. Aby bylo možné jednoduše odhadovat jak malé projekty v řádu hodin, tak i velké projekty v řádu měsíců, používají se ve firmě pro tyto potřeby primárně 2 jednotky pracnosti. Pro úplnost zmíním i třetí jednotku MM.

- **MH** – Man-hour nebo také person-hour, v češtině člověkohodina. Představuje 1 hodinu práce 1 člověka.
- **MD** – Man-day nebo také person-day, v češtině člověkoden. Jednotka symbolizuje práci, kterou 1 člověk odvede za 1 pracovní den.
- **MM** – Man-month nebo také person-month, v češtině člověkoměsíc. Vyjadřuje 1 měsíc práce jednoho člověka.

Vztah mezi zmíněnými jednotkami je následující:

$$8 \cdot 20 \text{ MH} = 20 \text{ MD} = 1 \text{ MM}$$

Jak jsou jednotky pracnosti chápány, demonstruji na následujícím příkladu. Projekt má pracnost 20 MD. Pokud na projektu bude pracovat pouze jeden člověk, bude mu práce trvat 20 dní, tedy jeden MM. Pokud budu chtít dokončení projektu uspíšit a jeho povaha to bude umožňovat, zaměstnám na projektu 20 lidí a za den bude hotový. Mohu samozřejmě na projekt alokovat i méně lidí. V čtyřčlenném týmů zabere dokončení projektu 5 dní.

1.4 Metodiky

1.4.1 Vypočítat a posoudit

Na softwarových projektech se vyskytuje velké množství činností, které mohou být počítány. Může se jednat o činnosti před vývojem jako jsou diagramy případu užití, funkční a nefunkční požadavky a další. V průběhu vývoje můžeme naopak počítat obrazovky, stránky, dialogy, databázové tabulky, počty polí ve formulářích atd. Po dokončení projektu můžeme počítat reportované chyby, počty tříd, řádků, ... [1, s. 85] Pro takto provedený odhad ale potřebujeme kromě samotného počítání ještě statistiky, nejlépe z podobných projektů. Na jejich základě poté můžeme projekt rozpadnout na menší částí, jejichž pracnost lze na základě počítání odhadnout. Pokud ovšem nemáme statistiky pro výpočet, je tato metodika pro odhadování nevhodná a je třeba použít jinou.

Výhodou a jedním z důležitých bodů této metriky je, že tvůrce odhadu při odhadování nepoužívá vlastní úsudek, ale čistá data. Odhad je tak vytvořený pouze na základě počítání a statistiky dat z historie.

1.4.2 Historická data

Pokud neděláme náš první odhad, a v minulosti jsme již provedli několik odhadů, můžeme se rozhodnout pro tuto metodiku. Při ní využíváme existence dat z realizovaných projektů. V této metodice je tedy důležité, abychom v průběhu projektů sbírali data. Po jejich dokončení budeme mít přesná data pro naše metriky. Můžeme tak na základě minulých projektů odhadovat pracnost nových. Pokud se navíc jedná o podobné projekty, je výrazně vyšší pravděpodobnost, že s pomocí této metodiky odhadneme nový projekt nebo alespoň zpřesníme a revidujeme již vytvořený. Data, která budeme sbírat, jsou čistě v naší režii. Může se jednat o počty řádků, počty tabulek nebo například počty požadavků. [1, s. 95-98].

1.4.3 Expertní odhad

Poslední metodiku, kterou je třeba představit, je strukturovaný expertní odhad. Jak píše McConnell, jedná se o nejběžnější způsob odhadování projektů v praxi. Princip metodiky je založený na odhadnutí pracnosti pomocí intervalu a nikoliv pouze jednoho čísla. Při odhadu se tedy pracuje s intervalem minimální (nejlepší případ) a maximální pracnosti (nejhorší případ). Pro získání finálního čísla odhadu by mohl být použit střed intervalu. Ten se ale může v mnoha případech lišit od toho očekávaného. Protože odhady by při použití pouze těchto hodnot byly pravděpodobně nepřesné, je v této metodice zavedena ještě jedna hodnota. Jedná se o vyjádření nejpravděpodobnějšího případu, který je určen na základě úsudku experta provádějícího odhad. Díky těmto třem hodnotám může být dle McConnella aplikována *PERT formule*. Jejím výsledkem je hodnota očekávaného případu, která bere v potaz všechny tři odhadnuté hodnoty. Definice formule vypadá následovně:

$$Očekávaný Případ = \frac{nejlepší + 4 \cdot nejpravděpodobnější + nejhorší}{6}$$

Jak lze vidět, výpočet je založen na váženém průměru. Protože nejpravděpodobnější případ je odhadnut expertem, očekává se, že se blíží nejvíce realitě. Z toho důvodu je mu ve váženém průměru přiřazena vyšší váha, která má na výsledný odhad dopad. Očekávaný případ odhadu se tak pohybuje na ose mezi nejlepším a nejhorším případem.

Aby se v expertním odhadu eliminovalo nebezpečí, že se při odhadu na nějakou oblast či celek zapomene, jsou součástí expertních odhadů i kontrolní seznamy neboli takzvané checklisty. Jedná se o seznam aktivit, na které se při odhadování nesmí zapomenout a jejichž nezahrnutí do odhadu by mohlo mít nemalý dopad na jeho celkovou pracnost. Tento seznam je třeba si nadefinovat ještě před tvorbou odhadu a poctivě aktualizovat pro budoucí odhady. [1, s. 106-110]

1.5 Používané metodiky ve firmě

Ve firmě Profinit je sepsán dokument s metodikou, jak při tvorbě odhadů postupovat. Metodika čerpá ze znalostí v publikacích [1] a [2]. Při standardní tvorbě odhadů se postupuje podle metodiky Expertního odhadu1.4.3. V duchu této metodiky je také připravena Excel šablona, ve které se odhad tvoří.

Pokud se vytváří odhad u zákazníka, pro kterého byly v minulosti již realizovány jiné podobné projekty nebo se jedná o změnový požadavek pro existující projekt, můžeme využít znalosti z předešlých projektů. V případě, že máme z minulých projektů u zákazníka sesbírané statistiky ohledně různých činností na projektech, můžeme firemní metodiku zkombinovat s metodikou Vypočítat a posoudit1.4.1. Na základě sesbíraných údajů můžeme odhadnout, kolik času zabere například vytvoření webové stránky nebo formuláře. Díky těmto údajům můžeme odhadnou celý projekt nebo alespoň jeho část a tím odhad zpřesnit na základě sesbíraných statistik.

Pokud se jedná rozsahem a funkčností o podobný projekt, můžeme k metodice použít i metodiku Historická data1.4.2 a na základě porovnávání projektů odhadnout pracnost nového projektu.

Kapitola 2

Analýza současných řešení

Po provedení analýzy současných řešení tvorby odhadu pracnosti vzešly 2 nástroje, které v dalších kapitolách popisuji. Kapitoly jsou řazeny historicky tak, jak byly zaváděny pro standardní tvorbu odhadů uvnitř firmy.

2.1 Excel šablona

Prvním ze dvou nástrojů pro tvorbu odhadů je dokument (dále jen šablona) pro tabulkový procesor Microsoft Excel. Toto řešení existuje ve firmě poměrně již dlouho a momentálně se jedná o nejvíce používané řešení pro tvorbu odhadů.

Excel šablona vychází z metodiky odhadů zmíněné v části 1.5. Šablona umožňuje rozpad odhadu na jednotlivé kategorie softwarového inženýrství. V šabloně jsou definovány následující kategorie:

- Analýza
- Design
- Implementace
- Testování
- Project Management (dále jen PM)
- Dodávka
- Ostatní

Každá výše zmíněná kategorie je v šabloně reprezentována Listem. List obsahuje celkem 2 tabulky – checklist a jednotlivé položky rozpadu dané kategorie odhadu. Checklist je reprezentovaný čistě textovými řádky a připomíná uživateli, čeho se držet a na co nezapomenout při tvorbě odhadu. Druhou tabulkou, která je pro samotný odhad nejdůležitější, je tabulka s položkami

rozpadu konkrétní kategorie. Tabulka umožňuje rozpad kategorie na jednotlivé položky, kde je každá položka reprezentována jejím názvem, její odhadovanou pracností a volitelnou příslušností do varianty odhadu. Ta nám umožňuje při tvorbě odhadu počítat s více možnými scénáři. Dobu pracnosti lze odhadnout celkem 3 hodnotami vyjádřenými v číslech – minimální, maximální a nejvíce pravděpodobná. Na základě těchto údajů jsou vypočítány průměrná a očekávaná pracnost. Např. odhad jedné položky kategorie Implementace by mohl obsahovat název „Vytvoření REST API“ a odhadované časy dokončení.

Kromě listů s jednotlivými kategoriemi softwarového inženýrství obsahuje šablona také list Předpoklady. Tento list slouží k identifikaci všech předpokladů a omezujících podmínek, které jsou kladeny na odhadovaný projekt. Lze tak částečně eliminovat riziko, že odhadce na něco zapomene a výsledný odhad tak bude nepřesný.

Posledním listem nacházejícím se v šabloně je list s Přehledem odhadu. Na listu se nachází tabulka s jednotlivými kategoriemi a jejich pracnostmi. Pracnosti jsou vyjádřené jak v MDs, tak procentuálními podíly vůči celkové pracnosti. Data zobrazená v těchto tabulkách pomáhají uživateli při tvorbě odhadu kontrolovat, zda jednotlivé kategorie mají rovnoměrně rozdělenou pracnost. Např. zda má testování alespoň 60 % pracnosti implementace. Na tomto listu se také nachází konfigurace výše zmíněných variant, zde je možné jednotlivé varianty aktivovat (zahrnout) či deaktivovat (nezahrnout do projektu) a tím ovlivnit výsledek odhadu.

Excel šablona umožňuje několik variant konfigurace odhadu. První z konfigurací je volba výše záruky. Ta se vyjadřuje procentuálně (X %) a její hodnota v MDs je rovna X % celkového součtu samotné realizace. Ta obsahuje pouze některé kategorie odhadu, konkrétně Design, Implementaci, Testování a PM.

Dále lze nastavit cenu za 1 MD, která může být pro každý projekt individuální. Aby Excel šablona umožňovala vytvářet odhady jak pro menší projekty v rozsahu několika MDs, tak pro velké projekty v řádu stovek MDs, je konfigurovatelná i časová jednotka, ve které se odhad tvoří. Na výběr je ze dvou jednotek – MD a MH.

Poslední funkcionalitou, kterou šablona odhadu umožňuje, je export odhadu, který má formát textové reprezentace. Během vytváření exportu lze vybrat, aby se vyexportované údaje o odhadu naimportovaly rovnou do issue tracking systému Bugzilla, který je interně používaný.

2.2 Aplikace Estimate

Dalším nástrojem, který je možné využít při tvorbě odhadů, je nástroj Estimate. Ten vznikl v rámci bakalářské práce studentů Milana Vancla[3] a Juraje Polačoka[4]. Nástroj měl sloužit jako nástupce hojně používaného Excelu. Byl realizován jako webová aplikace, přičemž backend část byla implementována v jazyce Java a využívala framework Spring. Data byla ukládána do NoSQL

grafové databáze Neo4j. Frontend část aplikace pak byla také v jazyce Java v kombinaci s frameworkem Vaadin.

Toto řešení se bohužel ukázalo jako ne příliš udržitelné a provozuschopné. Realizace výsledného nástroje byla zaměřena převážně na jeho dokončení a finální uživatelé, kteří ho měli používat, viděli až vyhotovený nástroj. Výsledkem byla sice funkční aplikace, ale pro uživatele byla hůře použitelná. Mezi hlavní důvody, proč byli uživatelé nespokojení, patřilo uživatelské rozhraní, v němž vytvoření odhadu bylo pro uživatele pracnější a zabralo více času než v původní Excel šabloně.

Použití grafové databáze pro tento typ nástroje bylo navíc zbytečně předimenzováno. Proto byl nástroj Estimate v rámci diplomové práce Miroslava Štaffy[5], která se zabývala zlepšením softwarového procesu ve firmě Profinit, částečně upraven. Ke změnám došlo převážně na straně backend části, kde byla provedena migrace z grafová databáze Neo4j na databázi relační. Na frontend části se jednalo jen o drobné změny. Z nich mohu zmínit například povýšení verze frameworku Vaadin z verze 7 na novější verzi 8.

Kapitola 3

Zvolení přístupu realizace

Před samotným začátkem procesu vývoje je třeba nejprve rozhodnout, jaký proces pro vývoj nástroje zvolit. Pro Estimate přicházejí v úvahu celkem 2 možné přístupy – klasický vodopádový a iterativní přístup.

3.1 Vodopádový přístup

Model vodopádu, známější pod svým anglickým názvem Waterfall, je jeden z nejstarších přístupů a je z rodiny sekvenčních modelů. Softwarový proces se podle klasického modelu zmíněného v [6] dělí na fáze v následujícím pořadí:

1. Specifikace a analýza požadavků
2. Návrh
3. Implementace
4. Testování
5. Údržba a provoz

Během vývoje postupně procházíme jednotlivé fáze. Pro započetí další fáze je potřeba, aby předchozí fáze byla již dokončená. Jakmile je tedy jednou fáze dokončena, nemělo by se k ní zpětně ve smyčce vracet.

Tento přístup je jeden z jednodušších z pohledu projektového řízení. Jsme schopni provést přesnější odhad náročnosti před samotnou realizací, lze stanovit termín dokončení nebo např. v jaké fázi se projekt v danou chvíli nachází.

Využití vodopádového přístupu s sebou nese i jistá rizika. Zákazník musí vědět již na začátku, co od projektu očekává a jaké má na něj požadavky. „Spustitelnou“ ukázkovou verzi softwaru má zákazník šanci vidět až v pozdní fázi projektu. Na základě toho pak mohou přijít změnové požadavky, které způsobí dramatické dopady např. do architektury softwaru, které mohou ovlivnit cenu a datum dokončení.

3.2 Iterativní přístup

Jak už název napovídá, iterativní přístup je založen na provádění jednotlivých iterací, kde se v každé z nich postupně vystřídají všechny fáze tak, jak byly popsány ve vodopádovém modelu. Díky tomu jsme schopni mít na konci každé iterace její reálný výsledek, ať už se jedná o část softwaru nebo částečný prototyp.

Stejně jako u vodopádového přístupu jsme schopni odhadnout cenu a termín dokončení. Zákazník díky jednotlivým kontrolním bodům mezi iteracemi vidí průběh vývoje softwaru. To mu umožňuje odhalit např. chyby ve specifikaci v prvotní fázi projektu.

Mezi nevýhody patří projektové řízení, které vyžaduje vyšší nároky oproti vodopádu. Zákazník musí vědět již na začátku, co od softwaru očekává. Díky iteracím je zde vyšší pravděpodobnost odhalení neporozumění si se specifikací v prvotní fázi. Nicméně větší chyby ve specifikaci mohou mít vliv na termín dokončení i cenu realizace.

3.3 Vybraný přístup

V realizacích obou bakalářských prací pro aplikaci Estimate byl pro vývoj použit přístup pomocí vodopádového modelu, při kterém se vracelo do již proběhlých fází. Vývoj si tak prošel jednotlivými fázemi a na jejich konci vznikl finální nástroj pro tvorbu odhadů. V testovací fázi uvnitř firmy se ale ukázalo, že uživatelé, kteří denně vytvářejí odhady, se v aplikaci nedokáží efektivně pohybovat. Bohužel některé zvolené UI komponenty frameworku Vaadin neumožňovaly dostatečné přizpůsobení. Proto se uživatelé vrátili k metodě odhadů pomocí nástroje Excel. Ač byl použit vodopád, nedodržovala se jeho pravidla tak striktně a v rámci jednotlivých kategorií se bylo možné vracet i k předešlým fázím a částečně je měnit.

Na základě předchozích realizací a výhod, resp. nevýhod obou popisovaných přístupů, jsem se rozhodl v rámci této práce aplikovat iterativní přístup s lehkými prvky agilního přístupu. Tento přístup mi umožní vždy na konci každé iterace provést testování s uživateli, kteří denně vytvářejí odhady. Dostanu tak okamžitou zpětnou vazbu, co se uživatelům líbí a co by naopak na nástroji řešili jinak. Všechny tyto získané poznatky pak budu moci zpracovat v rámci další iterace. V rámci vývoje budou plánovány schůzky jednou týdně. Na těchto schůzkách bude probráno, jaké úkoly jsou již hotové, na kterých se momentálně pracuje a na kterých se v průběhu týdne pracovat začne.

3.4 Požadavky před 1. iterací

Před samotnou 1. iterací je potřeba sesbírat alespoň základní funkční a nefunkční požadavky. Bez těchto požadavků by se nedala navrhnout architek-

tura aplikace a zvolit vhodné technologie. Právě z dobře navržené architektury a správně vybraných technologiích může aplikace v budoucnu těžit. Pokud by se totiž na začátku zvolila nesprávná cesta, mohlo by se v pozdějších fázích projektu ukázat, že architektura nebyla zvolena nejlépe a začala by mě při vývoji omezovat. Výsledkem by poté byl technologický dluh a nutnost předělání celé architektury, což by bylo časově velice náročné.

Z analýzy požadavků vzešly následující výstupy:

3.4.1 Funkční požadavky

1. Jednotlivé položky odhadu nabízejí stejné metriky jako používaný odhadovací Excel. Tzn. každá položka obsahuje název a 5 časových údajů, z toho 3 vyplňuje uživatel a 2 jsou dopočítávány na základě uživatelem zadaných hodnot.
2. Odhad se definuje pomocí JSON šablony. Tato šablona je konfigurovatelná uživatelem a obsahuje veškerá data odhadu.
3. Při úpravě šablony není nutné měnit kód samotné aplikace.
4. Nový odhad se reprezentuje jako nová instance šablony s nevyplněnými daty.
5. Položky v šabloně odhadu je možné členit ploše do tabulky, ale i pomocí stromové struktury.
6. Základními položkami šablony pro prvotní plochý odhad jsou:
   - Analýza
   - Design
   - Implementace
   - Testování
   - PM
   - Dodávka
   - Ostatní
7. Každá položka odhadu má v šabloně definovaný checklist. Ten udává, co je třeba splnit.
8. Odhad nabízí uživateli vyplnění předpokladů.
9. Aplikace podporuje tvorbu odhadu ve dvou časových jednotkách – man-day a man-hour.
10. Je možné spočítat pro odhad základní statistiky jako je tomu v Excel šabloně.

11. Uživatel může exportovat odhad do současně používané Excel šablony.
12. Uživatelské rozhraní nabízí pohyb pomocí klávesnice s minimální nutností využití myši.
13. Aplikace umožňuje přihlášení uživatelů a práce s odhady na základě jejich rolí.
14. Uživatel je schopný si v aplikaci zobrazit seznam odhadů.

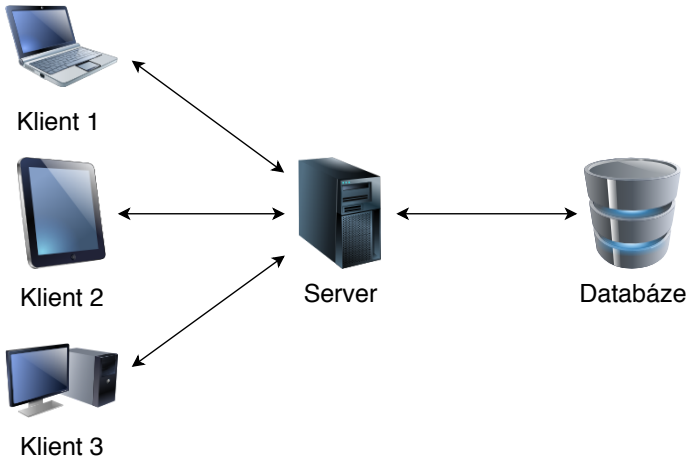
3.4.2 Nefunkční požadavky

1. Klient podporuje současnou práci více uživatelů.
2. Aplikace vyžaduje přihlášení uživatele a uživatelské role.
3. UX aplikace dle současných standardů (SPA).
4. Případná backend část je postavena na frameworku Spring (dáno požadavky zadavatele).
5. Klient reaguje na srovnatelné požadavky uživatele maximálně o 25 % pomaleji než v rámci současného řešení (Excel šablona).
6. Primární použití aplikace je pro laptop a desktop zařízení (rozlišení FullHD).
7. Aplikace podporuje nejnovější verze prohlížečů Google Chrome a Mozilla Firefox.
8. Aplikace je připravena na podporu více jazyků.

3.5 Architektura

Před začátkem realizace je nutné vybrat architekturu, na které budu celou aplikaci stavět. Už na začátku jsem měl v plánu aplikovat návrh, ve kterém bude aplikace rozdělena na 2 části – server a klient, viz Obrázek 3.1. Server bude řešit správu a persistenci dat a pomocí klienta bude moci uživatel pracovat s daty přes uživatelské rozhraní. Na základě těchto znalostí jsem se proto rozhodl vybrat architekturu klient-server.

Server (dále jen backend část) má několik odpovědností. Jak už z názvu architektury plyne, musí mezi klientem a serverem existovat propojení, aby klient mohl získat data od backend části. Tato část proto vystavuje pro klienta služby pomocí API vrstvy. V tomto případě se jedná o REST API, které je realizováno pomocí protokolu HTTP a umožňuje klientovi snadný přístup ke zdrojům (resources), kterými jsou v tomto případě data odhadu. Díky tomuto API je backend část zcela nezávislá na implementaci klienta. Aby ale backend



Obrázek 3.1: Architektura klient-server

část mohla přes API rozhraní data poskytovat, musí mít v sobě aplikační logiku pro práci s daty a tato data musí někde ukládat. K tomu mu slouží databáze, ke které je backend část připojena a v které všechna data uchovává.

Když už známe architekturu backend části, je třeba se ještě podívat na detaily klienta (dále jen jako frontend část). Frontend část uživateli poskytne uživatelské rozhraní pro práci s odhady a veškerá data budou získávána a ukládána přes API rozhraní backend části. Samotný frontend klient bude realizován jako webová single page aplikace.

Single Page Aplikace (SPA) je webová aplikace, která pro získání a ukládání dat využívá napojený server. Od klasické webové aplikace se liší v tom, že si na začátku stáhne všechna potřebná data (JavaScript soubory, HTML, CSS, ...) a při reakci na uživatelské akce jen dynamicky mění svůj obsah bez obnovy celé stránky. Jedinými okamžiky, kdy poté komunikuje přes síť, jsou požadavky na backend část a další akce jako např. stažení obrázku z webů 3. stran. Pro využití SPA aplikace jsem se rozhodl kvůli jejím výhodám, mezi které patří hlavně rychlá odezva bez nutnosti přenačtení stránky.

3.6 Technologické řešení

Po rozmyšlení a návrhu architektury přichází na řadu výběr technologií, ve kterých bude samotná aplikace realizována. Jelikož se aplikace skládá ze dvou částí, které se od sebe významně liší, je třeba pro každou z nich vybrat správné řešení.

3.6.1 Backend

Pro serverovou část, na které bude umístěna business logika, přístupy do databáze a poskytování dat pomocí REST API, je nutné vybrat framework, který by všechny tyto požadavky splňoval. Rozhodl jsem se tedy pro aplikační framework Spring Boot. K tomuto kroku mě vedly i zkušenosti s jinými aplikačními frameworky. Nemalou roli hrál i fakt, že je tento framework napříč projekty ve firmě hojně používán. Po dokončení diplomové práce tak nebude problém s údržbou a dalším rozšiřováním vzniklé aplikace.

3.6.1.1 Spring Boot

Nejdříve je nutné si říci něco o frameworku Spring. Jedná se o open-source aplikační framework, který je postavený na jazyce Java a umožňuje vývojářům vytvářet moderní enterprise aplikace. Pomáhá jim s realizací široké škály oblastí, jako například s připojením do databáze, vystavováním webových služeb, bezpečností nebo správou služeb v cloudu. Zároveň umožňuje uživatelům používat dependency injection a aspektově orientované programování.

Právě na tomto frameworku staví mnou zvolený framework Spring Boot, který Spring rozšiřuje a snaží se uživatelům co možná nejvíce usnadnit prvotní konfiguraci. Jak sami autoři uvádí: „Spring Boot usnadňuje vytváření samostatných, stabilních, spolehlivých aplikací založených na frameworku Spring, které „jen spustíte“. Máme jasný názor, jak použít platformu Spring a knihovny třetích stran, takže můžete začít s minimálním zdržením. Většina Spring Boot aplikací potřebuje velmi málo Spring konfigurace.“ [7, překlad vlastní].

Pro sestavení aplikace a správu všech závislostí jsem měl na výběr mezi nástroji Gradle a Maven. Vybral jsem si Gradle, který je na rozdíl od Maven mladší a používá doménový specifický jazyk vycházející z Groovy místo XML.

3.6.1.2 Java nebo Kotlin

Po výběru Spring Bootu přichází na řadu volba programovacího jazyka. Doposud jsem používal Spring Boot vždy jen v kombinaci s jazykem Java. Naskytla se ale možnost realizovat backend část v poměrně mladém jazyku Kotlin. Jedná se o staticky typovaný jazyk, který je open-source a původně byl navržen jen pro běh pod JVM a na mobilních platformách Android. Nyní je už ale možné ho zkompilovat i například do JavaScriptu nebo nativního kódu. Jazyk je poměrně nový, byl představen teprve v roce 2011. Autorem je vývojářská firma JetBrains, která je tvůrcem několika známých vývojových prostředí, např. IntelliJ IDEA.

Pro realizaci jsem zvolil jazyk Kotlin a to především kvůli několika následujícím výhodám oproti jazyku Java:

- Interoperabilita – Jazyk je plně interoperabilní s Javou. Je možné použít kus kódu Kotlinu i kódu z Javy. Lze tak používat knihovny, které jsou

implementovány v Java.

- Null Safety – Typování v Kotlinu se snaží vyvarovat používání referencí s null hodnotou a s tím spojenými pády programů kvůli typické Java chybě `NullPointerException`, kdy se volá metoda na null referenci. Kotlin toto řeší explicitní definicí, zda datový typ může nabývat i hodnoty null či nikoliv. Pokud při kompilaci zjistí chybné použití typu, který může být null, nedovolí programátorovi kód zkompilovat. Jedná se například o přiřazení hodnoty typu String, která může být null do proměnné, která je také typu String, ale nesmí obsahovat null hodnotu.

- Extension funkce – Tyto funkce nám umožňují přidat k již existující třídě novou metodu bez nutnosti vytvářet její podtřídu. Zároveň nám umožňují se vyhnout klasickým Java utils třídám. Můžeme tak například třídu `List<String>` rozšířit o metodu, která nám vrátí počet výskytů `String` hodnot na základě předaného parametru.

- Datové třídy (Data Classes) – Datové třídy jsou v Kotlinu poměrně silným nástrojem, který nám na rozdíl od klasických tříd usnadní implementaci a zredukuje množství psaného kódu. Datové třídě totiž stačí předat v konstruktoru pouze atributy s jejich typy, o vše ostatní už se poté postará Kotlin. Při kompilaci se pak vytvoří pro jednotlivé atributy třídy odpovídající settery a gettery a metody `toString`, `equals`, `hashCode` a `copy`.

- Native Immutability – Kotlin v základu podporuje práci s immutable objekty [8], které nemohou po inicializaci měnit své hodnoty. Pokud je potřeba v objektu změnit vnitřní hodnotu, musí být celý objekt znovu inicializován nebo duplikován s příslušnou hodnotou.

3.6.1.3 Databáze

Po výběru technologií pro backend část přichází na řadu volba databáze. V předešlých bakalářských pracích se používala jak relační, tak i grafová databáze pro persistenci dat v databázi. Já jsem se rozhodl jít v počáteční fázi projektu tou nejjednodušší cestou a vybrat databázi, ve které bude jednoduché vzít celý model odhadu a persistovat ho v databázi bez dalších náročných konfiguračních věcí kolem. Jelikož je model odhadu reprezentován pomocí JSON šablony a umožňuje stromovou strukturu, nejevila se volba relační databáze jako zcela vhodná. Místo toho jsem se rozhodl zvolit databázi z rodiny NoSQL. Tyto databáze nejdou cestou klasických relačních databází. Jejich princip tak nevychází z běžně používaných tabulek, sloupců a řádků a SQL dotazů nad nimi. NoSQL databáze tak nemají jasně definované databázové schéma a data jsou ukládána např. v podobě grafu, JSON souborů nebo třeba na základě klíče a hodnoty. NoSQL databáze jsou hojně využívané např. v oboru Data

science nebo tam, kde stačí data ukládat strukturovaně, jako např. v mém případě. V neposlední řadě na rozdíl od relačních databází nabízejí NoSQL databáze kromě vertikální také horizontální škálovatelnost. Díky tomu jsou vhodnější pro použití například i v distribuovaném prostředí.

Jelikož je odhad navržen tak, aby ho bylo možné reprezentovat JSON objektem, jevilo se pro začátek projektu jako nejvhodnější přímé uložení tohoto JSON objektu do databáze. Proto jsem se rozhodl sáhnout právě po NoSQL databázi, která umí pracovat s objekty JSON – konkrétně po multiplatformní open-source databázi MongoDB[9]. Jedná se o dokumentovou databázi, do které jsou data neboli dokumenty ukládány do kolekcí (obdoba tabulek v relačních databázích). Dokumenty jsou v databázi ukládány ve formátu BSON. BSON je binárně kódovaná reprezentace JSON formátu, která má podobnou strukturu jako JSON. Skládá se tak ze sady klíčů a hodnot.

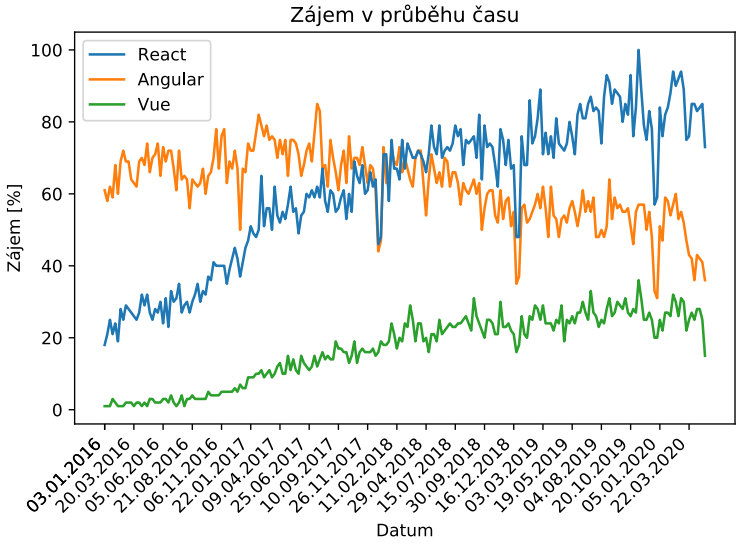
3.6.2 Frontend

Poslední částí, pro kterou bylo nutné vybrat vhodné technologie, byla frontend neboli klientská část aplikace. Ta poskytuje uživateli přístup k funkcionalitám serverové části pomocí grafického uživatelského rozhraní. Při návrhu architektury jsem se rozhodl jít cestou Single Page Aplikace a podle toho se také odvíjel výběr použité technologie. JavaScriptových knihoven a frameworků pro tvorbu SPA existuje velké množství, já jsem nicméně výběr omezil na následující tři nejvíce používané. Uvádím záměrně i verze, ve kterých byly v době výběru technologie dostupné:

- React 16.5.2 – knihovna
- Angular 7.0.0 – framework
- Vue.js 2.5.17 – framework

Všechny tři zmíněné technologie mají své výhody a nevýhody, a proto nelze obecně říci, která z nich je nejlepší. Výběr vždy závisí na konkrétních požadavcích daného projektu. Výběr tedy provedu na základě několika kritérií. Prvním z nich je učící křivka dané technologie. React a Vue.js mají učící křivku relativně plochou. Proto není takový problém si dané technologie v poměrně krátkém čase osvojit. Programátor je tak už po několika hodinách schopný vytvořit jednodušší aplikaci. Naopak Angular má učící křivku strmější, proto pochopení a schopnost vytvoření jednodušší aplikace zabere pravděpodobně více času. Je to způsobeno převážně tím, že v základu už obsahuje poměrně velké množství komponent a návrhových vzorů, které je nutné znát, a vývojář tak nemusí hledat potřebné knihovny jako např. pro práci s formuláři.

Dalším důležitým prvkem výběru technologie je odezva na uživatelské akce. V tomto bodě opět vychází lépe React a Vue.js. Obě zmíněné technologie používají virtuální DOM, který umožňuje překreslit vždy jen ty fragmenty

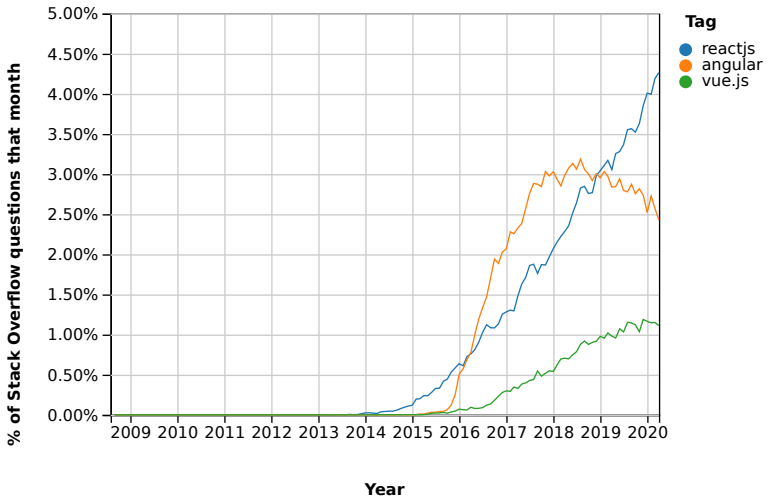


Obrázek 3.2: Vyhledávanost frontend technologií ve vyhledávači Google. Zdroj dat: [10]

HTML kódu, které byly opravdu změněny, zbytek nechává v původní podobě. Překreslení tedy zabere méně času a to umožňuje rychlejší odezvu aplikace.

Posledním prvkem pro rozhodování, kterou technologii vybrat, je vyhledávanost jednotlivých technologií vývojáři. Data o vyhledávání jsem se rozhodl získat ze dvou nezávislých zdrojů kvůli lepší objektivitě. Data jsem čerpal ze serverů Stackoverflow[11] (vývojářská komunita typu Q&A) a Google vyhledávače[10]. Obě služby poskytují webové rozhraní pro zobrazení vyhledávaností. Z Obrázku 3.2 a Obrázku 3.3 lze vyčíst, že Vue je stále mladá technologie, a proto je její vyhledávanost oproti dvěma zbývajícím nižší. V souboji React a Angular vedl do roku 2018 Angular. Během roku 2018 se jejich vyhledávanost vyrovnala. V následujícím roce 2019 už byl React nejvíce vyhledávanou technologií z výše zmíněných a tento náskok si drží i v začátku roku 2020. Z hlediska popularity tak vychází nejlépe React, případně hned za ním držící se Angular.

Na základě výše popsaných srovnání jsem se rozhodl využít knihovnu React. S touto knihovnou jsem již v minulosti pracoval a plně se mi osvědčila. Nevybráním Vue.js a Angular tak ubude problém s učením se nové technologie. Navíc si mohu oproti Angularu plně vybrat knihovny, které v aplikaci využiji a které nejsou doporučeny frameworkem.



Obrázek 3.3: Vyhledávanost frontend technologií na Stackoverflow. Zdroj dat: [11]

3.6.2.1 React

Jedná se o JavaScript open-source knihovnu pro tvorbu uživatelského rozhraní. Knihovna byla vytvořena v roce 2013 společností Facebook. Ta ji v současné době také stále spravuje a rozšiřuje o nové funkcionality.

Základním stavebním kamenem jsou komponenty, které reprezentují určité fragmenty stránky jako jsou např. formuláře, seznamy, obrázky či menu. Vývojář tyto komponenty může libovolně kombinovat a skládat dohromady. Tak tomu může být například u komponenty listu, která se dále skládá ještě z komponent jednotlivých řádků. Výsledkem je poté struktura komponent, která odpovídá stromu.

V React jsou data mezi komponentami předávána jednosměrně. To znamená, že data mohou být předána vždy jen z nadřazené komponenty směrem dolů do podřazených komponent neboli potomků, ze kterých se nadřazená komponenta skládá. Data předávaná komponentám mohou být dvojího typu. První z nich jsou proměnné, ty mohou nabývat jak základních hodnot jako je číslo, řetězec, tak i složitějších objektů. Druhým typem dat, který může být potomkům předán, jsou funkce. Ty slouží v React jako volání, která z potomků mohou ovlivnit stav rodiče a vynutit si tak jeho překreslení. Jiný způsob, jak změnit hodnotu rodiče než pomocí předané funkce, není.

3.7 Plán iterací

Poté, co jsou zanalyzované požadavky na aplikaci, je navržená architektura a vybrány technologie, ve kterých bude aplikace realizována, je na čase si rozvrhnout plán, jak bude vývoj aplikace probíhat. Protože jsem zvolil iterativní vývoj s agilními prvky, bude tvorba aplikace rozdělena do několika iterací, které na sebe budou navazovat. V rámci každé iterace proběhne na začátku detailní popis, čeho v ní budu chtít dosáhnout, neboli si stanovím její cíle. Zároveň si omezím rozsah věcí, aby se veškerá funkcionalita aplikace neřešila hned v počáteční fázi. Definuji si tak, čím se v dané iteraci budu zabývat. Zároveň si ale určím, co v ní řešit nebudu a danou problematiku si například nechám k řešení do jedné z dalších iterací. Po vymezení funkcionalit v dané iteraci budou následovat jednotlivé disciplíny softwarového inženýrství. Zanalyzuji funkcionality, které má aplikace v rámci dané fáze dostat. Poté proběhne návrh, ze kterého bude následně vycházet samotná realizace funkcionalit aplikace. V rámci každé iterace se samozřejmě počítá i s řešením bugů z předešlých iterací, na které se během realizace a testování přijde.

Abych si samotný vývoj lépe rozvrhl do jednotlivých iterací, rozhodl jsem se vytvořit jejich hrubý plán, jak popisuji výše. Po detailnější úvaze nad složitostí a rozsahem aplikace jsem se rozhodl vývoj rozdělit do celkem čtyř iterací. Každá z těchto iterací bude trvat okolo měsíce a půl. Během této doby bude nutné dokončit všechny předepsané úkoly této iterace. Na konci poslední iterace poté dojde ke zhodnocení vytvořené aplikace a jejímu nasazení do produkce.

3.7.1 První iterace

První neboli iniciální iterace – jedná se o iteraci, na jejímž začátku se projekt nachází ještě v plenkách a doposud neexistuje ani řádek kódu. V této fázi se zaměřím na vytvoření projektu a poskytnu uživatelům možnost vytvořit jednoduchý odhad, který bude možné uložit do databáze pomocí REST API a následně si ho opět otevřít a editovat.

Bude tedy nutné pro projekt vytvořit GitLab repozitář, ve kterém budou zdrojové kódy verzovány. Až bude repozitář vytvořený, přejdu k návrhu a struktuře členění zdrojových kódů. Na jejím základě poté mohu zinicializovat frontend a backend část aplikace. Po inicializaci se bude nutné zaměřit na doménový model odhadu, konkrétně na jeho analýzu a návrh. Právě tento krok je jedním z důležitých kroků, protože na doménovém modelu budou poté stavět i další iterace, které jej již budou pouze doplňovat o další informace. Dále bude nutné navrhnout a implementovat backend část, která bude data ukládat do databáze a poskytovat k nim přístup přes REST API.

Aby uživatel mohl odhad jednoduchou cestou vytvářet, proběhne návrh a implementace frontend části s jednoduchým uživatelským rozhraním. Pomocí tohoto UI bude uživatel schopný odhad vytvářet, modifikovat či pří-

padně archivovat. V rámci tvorby UI bude nutné zmapovat současné knihovny poskytující tabulkové komponenty a na základě rychlého PoC vybrat tu nejvhodnější. Jedná se o jednu v budoucnu z nejvíce používaných komponent aplikace, musí tedy umožňovat požadované funkcionality. Její výměna by byla v pozdější fázi z hlediska pracnosti komplikovanější.

Vytvořený odhad by nám ale byl k ničemu, pokud by byl uložený pouze na straně frontend části. Proto poslední částí bude zajištění komunikace frontend s backend částí přes REST API. Frontend část tak bude moci získávat seznam odhadů či posílat modifikovaný odhad k uložení.

Protože se v první fázi zaměřím na funkční aplikaci umožňující vytvoření odhadu v uživatelském rozhraní a následné uložení přes REST API, nebude v této iterace vůbec řešen uživatel, tedy jeho autentizace ani autorizace k příslušnému odhadu. Dalším krokem, který mě vedl k vypuštění uživatele v této fázi je fakt, že aplikace nepoběží v testovacím prostředí. Nebude tak vadit, pokud uživatel uvidí a bude moci modifikovat odhady i ostatních uživatelů.

3.7.2 Druhá iterace

Až budou splněny všechny body z předešlé iterace, je čas přejít k druhé iteraci. Z předešlého kroku bude mít aplikace implementovanou základní funkcionalitu v omezené míře. Proto se i v této iteraci zaměřím především na rozvoj funkcionality aplikace. Zároveň ale počítám s tím, že se během vývoje objeví chyby, které byly do aplikace zaneseny (ať už v první nebo v této iteraci) a bude je proto třeba zanalyzovat a opravit.

Ještě předtím, než začnu s implementací dalších funkcionalit, by bylo dobré po každé nově přidané funkcionalitě nebo opraveném testu vědět, zda testy prošly. Proto je v plánu nastavit pro vývojový proces Continuous Integration (CI) prostředí, které po každém odeslání změn v commitu do GitLab repositáře ověří, zda je možné sestavit aplikaci a zdali všechny implementované testy doběhnou v pořádku. Když už budeme mít tento proces přidaný, byla by škoda nevyužít jeho potenciál. Proto v případě, že se bude jednat o hlavní vývojovou větev a sestavení s otestováním neskončí chybou, bude se aplikace nasazovat do testovacího prostředí, které bude nutné též zařídit. V tomto testovacím prostředí bude aplikace dostupná k otestování jak pro samotné vývojáře projektu, tak i pro potencionální koncové uživatele.

Vedle nastavení CI, bude dále probíhat vývoj funkcionalit. Mezi ty nejdůležitější bude patřit:

- Tagování odhadu – Aplikace bude umožňovat uživateli přidat několik tagů k jednotlivým odhadům.

- Export odhadu do staré Excel šablony – Uživatel si po dokončení odhadování může celý odhad vyexportovat ve formátu Excel šablony. Navíc pokud bude pro odhad potřebovat nějakou klíčovou funkcionalitu, která

zatím není implementovaná, může si odhad vyexportovat a dokončit ho v šabloně.

- Validace vstupu pracností

- Internacionalizace – Aplikace bude v základu podporovat češtinu a angličtinu.

- Dialog pro detailnější editaci odhadu – Dialog, ve kterém bude možné kromě názvu odhadu přidat krátký popis a objeví se tu například i právě Tagování odhadů. Do budoucna se zde objeví další nastavení odhadu.

- Konfigurace okna pro editaci odhadu – Okno bude rozděleno na několik separátních částí, které si uživatel bude moci uzpůsobit dle své preference.

3.7.3 Třetí iterace

Ve třetí iteraci již bude aplikace schopná vytvářet první odhady a bude nasazená na testovacím prostředí. Zároveň od druhé iterace aplikace poběží na testovacím serveru, kde bude dostupná prvním uživatelům. Oslovení kolegové budou při tvorbě odhadů používat Estimate v testovacím prostředí místo současně používaných Excel šablon.

Od testování aplikace uživateli se očekává, že vznikne množina úkolů s chybami nebo připomínkami ke zlepšení, na kterých se převážně bude v této iteraci pracovat. Bude se pravděpodobně jednat hlavně o přehlédnuté chyby ve funkcionalitě a ladění UI na všech typech zařízení (různorodé prohlížeče, rozdílná rozlišení monitorů, lokalizace, ...).

Kromě připomínek a chyb je v této iteraci plánováno přidání dalších klíčových funkcionalit, mezi které patří hlavně:

- Varianty odhadu – Jedna z nejvíce chtěných funkcionalit, která je i v současném Excel řešení. Uživatel si může jednotlivé položky odhadu zařadit do příslušných variant(čísla 1-6). Varianty je možné si pojmenovat. S takto vytvořeným odhadem si poté uživatel může jednotlivé varianty zapínat nebo vypínat a vidět tak celkovou pracnost s danou variantou nebo bez ní.

- Helper funkce – Pomocné funkce definované přímo v šabloně odhadu, které mohou být nadefinovány pro jednotlivé kategorie odhadu (Analýza, Implementace, ...). Funkce provede nad stromem odhadu výpočet a v příslušné kategorii se v UI zobrazí dole pod tabulkou s odhadem dané kategorie.

3.7.4 Čtvrtá iterace

V poslední části realizace projektu by měla být již většina funkcionalit implementovaných. V této iteraci předpokládám, že se bude pokračovat v ladění UI připomínek od uživatelů, které vzešly z předešlé nebo současné iterace. Předtím, než bude vydána první verze aplikace Estimate, bude třeba ještě přidat pár klíčových funkcionalit, bez kterých by se uživatelé neobešli.

Aby uživatelé mohli aplikaci rozumně používat, bude třeba v této fázi přidat v první řadě přihlašování uživatelů. K přihlášení je v plánu využít současných služeb uvnitř firmy. Odpadne tím starost řešit registraci uživatele a umožní to uživatelům použít přihlašovací údaje, které používají již v současnosti při přihlašování do ostatních interních aplikací. Poté, co aplikace autentizuje uživatele, bude možné na základě informací o přihlášeném uživateli řídit přístup k odhadům podle příslušných uživatelských práv.

Poté, co budou vyřešeni uživatelé v aplikaci, bude třeba přidat jednu z posledních funkcionalit, o kterou bude aplikace obohacena. Konkrétně se jedná o vylepšení stránky se seznamem odhadů neboli Dashboard. Na této stránce byly do současné chvíle zobrazeny veškeré odhady, které se v aplikaci vytvořily. Na základě nově přidaných uživatelských práv se zde nyní budou nacházet jen některé vybrané. Například běžnému uživateli se zobrazí pouze odhady, které tento uživatel sám vytvořil nebo které s ním sdílí ostatní uživatelé. Zároveň zde bude uživateli umožněno tyto odhady ještě dále filtrovat podle několika atributů (např. název, autor, tagy, ...). Na základě těchto atributů bude možné odhady i řadit a vyhledávat v nich přes fulltextové vyhledávání.

Mezi další funkcionality, které nejsou takto majoritní, ale i tak mohou uživateli s prací na odhadu pomoci, budou dále patřit:

- Možnost odškrtnout si položky checklistu. Uživatel díky tomu bude mít přehled, které položky odhad splňuje a které ještě ne.
- Podpora referencí, rovnic a výrazů v rámci jedné tabulky.

Poté až budou všechny výše zmíněné funkcionality implementované, bude vytvořena první dodávka aplikace, která bude výsledkem této poslední fáze.

Kapitola 4

1. iterace

V této kapitole se zaměřím na detailní popis činností, které je nutné provést v rámci první iterace projektu. Ten se v tuto dobu nachází zcela v zárodku. Před samotným začátkem iterace jsou již známy funkční a nefunkční požadavky. Dále je vybrána a navrhnuta architektura, na jejímž základě byly poté vybrány technologie, ve kterých se bude aplikace realizovat. Všechny tyto kroky byly detailně popsány v předchozí kapitole 3, kde je jim věnován nemalý prostor.

4.1 Verzování kódu

Ještě předtím, než začnu popisovat vývoj Estimate v první iteraci, je třeba se zmínit o verzování. Všechny zdrojové kódy, které jsou během implementace vytvářeny, jsou samozřejmě verzovány, aby během vývoje nedošlo k jejich ztrátě. Na výběr je ze dvou verzovacích systémů – SVN a GIT. Použití prvně zmíněného systému jsem zavrhl a vybral jsem druhý jmenovaný. GIT je v současné době jeden z nejvíce používaných verzovacích systémů, který používá většina známých projektů. Oproti SVN nabízí větší množství užitečných funkcionalit, mimo to se výběr váže i k volbě nástroje pro podporu DevOps na projektu.

Pro Estimate jsem vybral DevOps nástroj GitLab. Nejpřesněji vystihuje tento nástroj popis v GitLab repozitáři od samotných tvůrců. „GitLab je open-source end-to-end platforma pro vývoj softwaru se zabudovanou správou verzí (verzování kódu), issue tracking systémem, revizemi, CI/CD (Continuous Integration/Continuous Delivery) a dalšími funkcionalitami.“ [12, překlad vlastní]. Systém běží na firemních serverech jako samostatná instance a jsou v něm standardně verzovány i některé zákaznické projekty. GitLab tak běží na definované adrese a přístup k němu je možný přes webové rozhraní. Jak už bylo zmíněno, systém kromě klasického repozitáře obsahuje i další nástroje, které podporují tvorbu softwaru. Jedná se především o nástroj pro správu tiketů (issue tracking), tvorbu merge requestů s podporou procesu revize, možnost nastavit si CI, CD a stránky s dokumentací, nazývané Wiki, kam je možné vy-

tvářet dokumentaci celého projektu v Markdown[13] syntaxi a udržovat jejich historii pomocí verzování.

Po založení GitLab repozitáře je třeba si definovat, jak se bude v repozitáři pracovat se zdrojovými kódy. Rozhodnutí padlo pro „klasický“ model, který je často k vidění na většině projektů. V repozitáři existuje hlavní větev, nazvaná master. V této větvi se nachází zdrojové kódy aplikace, která je spustitelná. Jedná se tedy o stabilní větev. Dále mohou v repozitáři existovat další dva typy větví. První z nich je feature větev, ve které se vyvíjí nové funkcionality. Druhým typem je větev bug, ve které jsou opravovány chyby v implementaci z master větve. Pro obě tyto větve platí, že jsou později zamergovány do master větve.

Protože GitLab umožňuje spravovat issues (tikety) ve svém vlastním issue tracking systému, rozhodl jsem se ho použít. Vyhnu se tak zbytečnému bobtnání potřebných systémů na projektu a integracím mezi nimi, jako je například integrace hojně používaného issue tracking systému YouTrack s GitLab repozitářem. Zároveň je třeba si nadefinovat pravidla a konvence, jak issue tracking systém používat společně v kombinaci s názvy větví, merge requesty a jejich revizemi.

4.1.1 GitLab workflow

4.1.1.1 Role

Než se dále zmíním o issues, labelech a merge requestech, je potřeba si nejprve říci něco o rolích v používaném workflow. Aby bylo možné rozlišit vývojáře (osoby), které figurují v životním cyklu issue, vznikly celkem tři role, které vývojář může nabývat v rámci práce s issue:

- Issuer – Zadavatel issue.
- Vývojář – Řešitel zadané issue.
- Reviewer – Ten, kdo provádí revizi.

4.1.1.2 Konvence pro pojmenování větví a commit zpráv

Aby byly všechny názvy větví v repozitáři jednotné, bylo třeba zavést konvenci pro jejich pojmenování. Větve pro tento projekt jsou pojmenovány podle následujícího vzoru – `branchType/login/issueId-summary`

- `branchType` – Může v současnosti nabývat dvou hodnot v závislosti na typu issue:
  - *bug* – Zadané issue je report chyby. Tzn. obsahuje label *bug*.
  - *feature* – Zadané issue je nová funkcionalita. Tzn. obsahuje label *enhancement*.

- `login` – Login neboli username uživatele v doméně firmy Profinit.
- `issueId` – Číslo issue v GitLab issue tracking systému.
- `summary` – Název issue bez diakritiky (mezery jsou nahrazeny pomlčkami).

4.1.1.3 Životní cyklus issue

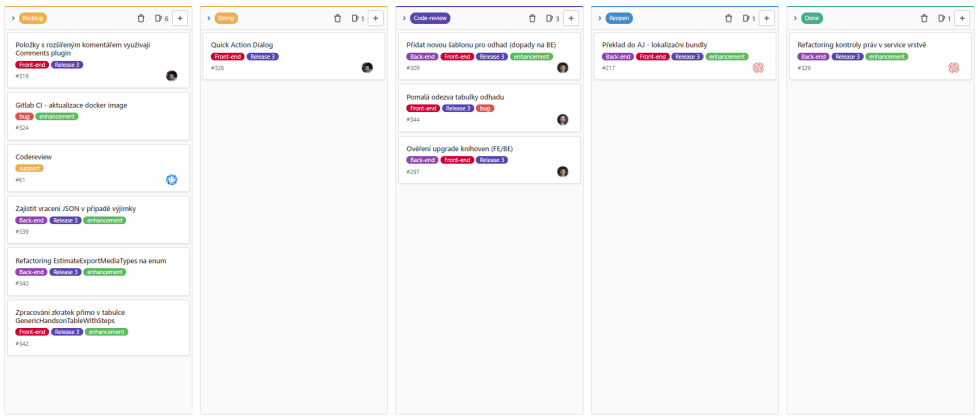
Předtím než popíšu samotný životní cyklus issue a merge requestu, je třeba si představit jednotlivé labely, které slouží k reprezentaci stavu issue. Možné labely jsou následující:

- stav Open – Issue je vytvořené, ale není ve frontě na práci.
- Backlog – Issue, na kterém může vývojář začít pracovat.
- Doing – Vývojář na issue pracuje.
- Code-review – Požadavek z issue je již implementovaný a probíhá jeho revize.
- Reopen – Kód neprošel revizí a je třeba ho předělat.
- Done – Issue je dokončené a připravené na zamergování.
- stav Closed – Issue je zamergováno do master větve.

V systému GitLab existuje nástroj Board, na kterém je možné vidět všechny úkoly ve sloupcích. Sloupce reprezentují jednotlivé labely. Proto se v nich úkoly zobrazují v závislosti na tom, zda obsahují label daného sloupce. Pro lepší představu je možné se podívat na Obrázek 4.1, kde je Dashboard zobrazený.

S labely merge requestů (MR) se pracuje obdobně a měly by být vždy ve stejném stavu jako je jeho issue. Nyní si již pojďme představit používaný model životního cyklu úkolů:

1. Vytvoření nového issue. Issue je ve stavu Open.
2. Na issue se může začít pracovat. Issue je přesunuto v Boardu do Backlogu. Automaticky se tak přidá label Backlog.
3. Vývojář, který bude na issue pracovat, jej přiřadí sobě nebo je mu přiřazeno.
4. Předtím, než vývojář začne na issue pracovat, musí jej v Boardu přesunout do sekce Doing. Automaticky je přidán label Doing a odstraněn Backlog.



Obrázek 4.1: GitLab Issue Board

5. Dané issue je třeba řešit v rámci jedné větve. Je třeba si vytvořit odpovídající MR a větev. To je možné provést přímo v detailu issue v GitLabu. Před vytvořením je třeba upravit název větve podle výše zmíněných konvencí pro pojmenování větví.

6. Vývojář pracuje na issue a commituje změny do vytvořené větve. Zároveň by si měl udržovat svoji větev aktuální vůči větvi master. Jednou z možností je dělat si každý den rebase větve vůči master větvi.

7. Když je issue hotové, vývojář si vybere člena z týmu na code review. Issue přesune v Boardu do Code-review. MR a issue přiřadí reviewerovi. Zároveň u MR klikne na tlačítko „Mark as ready“, přidá label Code-review a odstraní Doing.

8. Reviewer dokončí code review. Preferují se komentáře v záložce Changes, která podporuje revizi kódu.

    - Pokud reviewer nemá žádné připomínky:

        a) Reviewer přidá komentář, že vše je připraveno na merge do master větve, odebere label Code-review, klikne na tlačítko „Approve“ a MR přiřadí vývojáři. Issue přiřadí issuerovi.

        b) Issuer zkontroluje, zda výsledek odpovídá jeho představám. Pokud nemá žádné výtky, přiřadí issue vývojáři a přesune ho v Boardu do sekce Done. Pokud něco není podle jeho představ, popíše problém v issue. Nastaví issue a MR jako Reopen a přiřadí ho zpět vývojáři. Dále se pokračuje bodem číslo 2 jako při opravě MR.

c) Po souhlasu issuera vývojář zamerguje MR do master větve. Při kliknutí na tlačítko „Merge“ je třeba mít zaškrtnuté „Delete source branch“.

d) Merge automaticky zavře MR a issue s ním spojené. Odstraní větev a MR je označen jako MERGED.

- Pokud je potřeba v rámci MR provést nějakou změnu:

a) Reviewer napíše do revize, co je třeba upravit.

b) U issue a MR udělá následující. Přiřadí je zpět vývojáři, přidá label Reopen a odstraní Code-review.

c) Vývojář na issue začne pracovat. U issue a MR přidá místo labelu Reopen nový label Doing.

d) Po dokončení změn přidá issue a MR místo labelu Doing label Code-review a přiřadí je zpět na reviewera. Ten dále pokračuje stejně jako v bodu 8.

4.2 Analýza

Po dokončení této iterace by aplikace měla být ve stavu, kdy uživateli umožňuje založit nový odhad a editovat existující. Tzn. že frontend i backend části musí správně fungovat a musí mít zajištěnou vzájemnou komunikaci. Jak již bylo zmíněno v hrubém plánu, v této fázi nebude řešen uživatel. Jedná se sice o důležitou součást samotné finální aplikace, avšak v iniciální fázi projektu má tato funkcionalita nízkou prioritu. Vyšší prioritu má v tomto případě implementace ostatních funkčních požadavků (požadovaných funkcionalit) a vyladění UX.

Protože se celá aplikace točí kolem tvorby odhadů, zaměřím se v analýze nejprve na to, jak bude vypadat samotný doménový model odhadu. Požadavky na model byly definovány už v sesbíraných funkčních požadavcích 3.4.1. Konkrétně se jedná o funkční požadavky 1-11, které se týkají jenom modelu.

Protože doménový model je popsaný již v samotných funkční požadavcích, můžu se přesunout k další důležité části aplikace a to, jak na frontend části zadávat odhady. Jak již bylo zmíněno 2.1, v Excel šabloně jsou reprezentovány odhady jednotlivých kategorií softwarového inženýrství pomocí Listů. Každý List obsahuje tabulku položek odhadu, kde jsou jednotlivé řádky tvořeny textovým popiskem, třemi uživatelem zadanými a dvěma dopočítávanými číselnými údaji. Taková forma zadávání ale uživatele limituje v tvorbě odhadů se stromovou strukturou. Uživatel je schopný rozlišit pouze kategorie odhadu a jejich položky čili pracuje s „plochým“ odhadem. Sám si sice může uvnitř tabulky rozpadnout odhad ještě na další podčásti, musí k tomu ale použít vlastní značení pro nadřazené a podřazené řádky (např. pomlčky, křížky, ...). Při tomto způsobu stejně ale nelze zobrazit součet pro takto agregované řádky. Jak tedy vymyslet vyplňování odhadu v aplikaci? Přemýšlel jsem, jak

by bylo vyplňování pro uživatele nejjednodušší a efektivní. Nakonec jsem dospěl k tomu, že ve finále je pro uživatele nejrychlejší pohybovat se a vyplňovat data přímo v tabulce, tak, jako tomu je u Excel verze. V tabulce sice nebude uživatel schopný vytvořit stromový odhad, ten ale nebyl ani v první fázi projektu plánovaný.

Už je tedy rozhodnuto, jak se data budou do odhadu zadávat. Uživatel bude v aplikaci většinu času trávit vyplňováním metrik odhadu, tudíž se bude pohybovat uvnitř tabulky. Je proto z hlediska výběru knihovny s tabulkovou komponentou důležité, aby takto zvolená knihovna poskytovala požadovanou funkcionalitu a byla dobře zdokumentována. Proto je nutné správně zvolit knihovnu hned na začátku. Bylo by totiž nemilé, kdyby se během vývoje narazilo na problém, který není ve vybrané knihovně řešitelný. To by znamenalo vybrat podobnou knihovnu, která by měla stejnou funkcionalitu, a nevyskytoval by se v ní tento problém. Pokud by to navíc nastalo v pozdější fázi projektu, mohl by se projekt zbrzdit a jeho pracnost by výrazně narostla. Navíc vyměňovat jednu z nejvíce používaných komponent v již zaběhlém projektu není ve většině případů triviální.

Abych se tomuto černému scénáři vyhnul, rozhodl jsem se před začátkem projektu zmapovat JavaScriptové knihovny, které nabízejí tabulkové komponenty přímo pro knihovnu React. Z vytipovaných knihoven jsem si nakonec vybral jednoho kandidáta, knihovnu Handsontable [14]. Ta se mi ze všech nabízených knihoven zamlouvá nejvíce, má výbornou dokumentaci a navíc je v komunitní verzi zdarma. Ještě předtím, než se knihovnu rozhodnu použít, chci ji pořádně vyzkoušet. Zvolil jsem proto formu Proof of Concept (PoC). V PoC jsem si ověřil, že knihovna poskytuje všechny potřebné funkcionality a funguje tak, jak bych očekával. Nic již tedy nebrání tomu ji použít.

V rámci analýzy je třeba také zanalyzovat, jak se uživatel bude v aplikaci pohybovat a co v ní bude chtít realizovat. Nalezeny byly celkem tři možné případy, kdy chce uživatel aplikaci použít.

- Založení nového odhadu – Uživatel si chce v aplikaci založit nový odhad. V tomto případě klikne na tlačítko se založením nového odhadu. Po jeho kliknutí se otevře obrazovka pro editaci nového neuloženého odhadu. Uživatel může prázdný odhad začít editovat nebo rovnou uložit a dále editovat.

- Zobrazení existujícího odhadu – Uživatel si v seznamu najde odhad, který si chce zobrazit a na příslušném řádku klikne na tlačítko *Zobrazit*. Poté se zobrazí obrazovka s detailem odhadu, kde si ho uživatel může detailně prohlédnout.

- Editace existujícího odhadu – Uživatel postupuje stejně jako v případě Zobrazení existující odhadu. Po zobrazení detailu začne odhad editovat. Po dokončení editace může odhad uložit kliknutím na tlačítko *Uložit*.

4.3 Návrh

4.3.1 Model odhadu

Při návrhu modelu odhadu vycházím z funkčních požadavků 3.4.1, kde jsou definovány některé požadavky, které model musí splňovat. Protože implementace veškerých zdrojových kódů bude využívat anglických názvů, rozhodl jsem se popis modelu popsat již teď pomocí anglických výrazů. Při návrhu odhadu jsem došel k tomuto výsledku.

Protože všechny odhady budou persistovány v databázi, v prvé řadě je potřeba mít pro každý odhad unikátní identifikátor `id`. Podle něho budu schopný odhad z databáze získat. Samotný identifikátor by ale uživatelům nic o odhadu neříkal a špatně by se pamatoval. Z tohoto důvodu musí být možné odhad pojmenovat. Na základě pojmenování bude uživatel schopný říci, o co se zhruba v odhadu jedná, a odlišit ho od ostatních. K popisu odhadu nebude sloužit jen samotný atribut `description`. Ke zlepšení jeho popisu bude možné přidat i vlastní množinu tagů neboli klíčových slov, která odhad vystihují. Ta budou v odhadu pod názvem `tags`. Tagy lze použít například pro zmínění jednotlivých technologií, ve kterých bude odhadovaný projekt implementován, pro kontext zákazníka, oblast odhadu (půjčky, pojištění, ...) atd. Kromě výše zmíněných atributů bude také nutné vědět, kdo odhad vytvořil. K tomu mi poslouží atribut s identifikací autora odhadu `author`, kde bude uloženo `username` a `name`, neboli uživatelské a celé jméno autora. Pro lepší odhad projektu sloužily v původní Excel šabloně odhadu předpoklady vstupních podmínek. Proto v modelu nesmí chybět atribut `preconditions`, ve kterém je uložena množina těchto předpokladů.

Z modelu už zbývá popsat jen položky a konfiguraci pro samotný odhad. Nejdříve představím konfiguraci skrývající se pod atributem `configuration`. Díky němu je možné uchovat konfiguraci odhadu uvnitř modelu. Každý odhad musí být odhadován v nějaké časové jednotce – `timeUnit`. Ve společnosti Profinit se používají dvě – man day a man hour. Proto jedním z atributů konfigurace je právě tato časová jednotka. Dále je zde uchována číselná hodnota záruky odhadu pod názvem `warranty`. Ta udává, o kolik procent navíc bude z celkové pracnosti projektu plánovaná záruka. Poslední součástí konfigurace je množina vyhodnocovacích funkcí `valueFunctions`. Tyto funkce definují výpočet, kterým se vypočítají některé hodnoty položek z ostatních hodnot na základě typu koeficientu. Může se jednat například o hodnotu průměr, která se vypočítá z koeficientů minimální a maximální hranice. Každá takováto funkce má tedy `coefficient`, jehož hodnotu tato funkce vypočítává, a atribut `definition`, ve kterém je definováno tělo samotné funkce.

Poslední důležitou částí odhadu, která doposud nebyla představena, jsou jeho položky. Odhad obsahuje atribut `items`, který je tvořen množinou jednotlivých položek neboli itemů. Při návrhu členění itemů v odhadu bylo třeba vycházet z požadavku „Položky v šabloně odhadu je možné členit ploše do ta-

bulky, ale i pomocí stromové struktury.“. V Excel šabloně byl odhad rozdělen na jednotlivé kategorie (analýza, design, ...). Tyto kategorie poté obsahovaly tabulku, ve které se nacházely položky dané kategorie. Toto rozdělení splňovalo první podmínku o plochém odhadu, nikoliv však druhou část o stromové struktuře. Tu splňuje jen částečně. Starou strukturu lze chápat jako strom. Kořenovým uzlem je samotný odhad, ze kterého vedou hrany do kategorií. Z každého uzlu kategorie poté vedou hrany do příslušných itemů tabulek kategorií. Jak lze vidět, jedná se sice o stromovou struktura, nicméně hloubka tohoto stromu je limitována. V tomto případě má strom hloubku právě dva. Nelze tak koncovým uzlům přidat další uzly a udělat z nich uzly vnitřní.

Strukturu z Excel šablony je proto třeba částečně modifikovat. Odhad bude chápán jako strom, stejně tomu bylo v Excelu. Rozdíl bude jen v uzlech jednotlivých položek v kategoriích. Tyto uzly budou buď koncové, nebo vnitřní. Pokud bude uzel koncový, nebude obsahovat další položky a oproti Excel verzi zde nebude žádná změna. Druhou možností je, že bude uzlem vnitřním. V tomto případě bude uzel obsahovat další položky. Ve stromu to znamená, že z uzlu povedou další hrany do podpoložek. U nich nastane stejná situace jako u položek. Buď bude koncovým uzlem, a nebo vnitřním, který se bude dále větvit. Takovéto členění již splňuje definici stromové struktury.

Členění jednotlivých položek odhadu je již popsáno, je třeba ještě zmínit jejich atributy. Data, která jednotlivé itemy uchovávají, jsou hodně podobná těm z Excel šablony. Proto bude položka obsahovat její popis pod slovem `description`. V něm je ve zkratce shrnuto, co daná položka reprezentuje. Může zde být uvedeno například „Sběr funkčních požadavků“. Jak už bylo zmíněno výše, item má dále atribut `items`. Zde může být uložena množina dalších položek, díky kterým jsem schopný v odhadu tvořit stromovou strukturu. Aby mohly být jednotlivé položky odhadnuty, musí kromě popisu umožňovat přidat i samotný odhad pracnosti itemu, který se skládá z 5 hodnot. Tyto hodnoty jsou uloženy jako množina pod atributem `values`. Každá takováto hodnota je tvořena číslem, tedy samotným odhadem pracnosti, a koeficientem, který popisuje typ odhadované hodnoty. Typy jsou stejné jako v Excel šabloně, může se jednat například o minimální, maximální hranici, o expertní odhad aj. V modelu hodnot jsou tyto informace uchovány pod atributy `value` a `coefficient`.

Návrh modelu odhadu je již celý představený. Pro konkrétnější představu návrhu jsem využil vlastního popisu pomocí pseudoobjektů. Tento popis lze vidět ve Zdrojovém kódu 4.1.

4.3.2 Struktura projektu

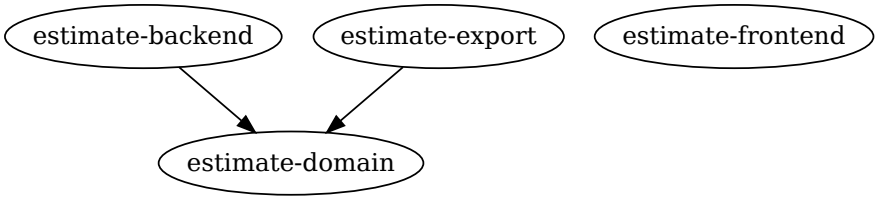
Při návrhu struktury je třeba vyřešit, jak nejlépe rozdělit jednotlivé části aplikace, aby byly některé její části nezávislé na implementačních detailech. Protože jsem se rozhodl použít pro správu závislostí nástroj Gradle, jeví se mi nejlepším řešením rozdělit aplikaci do několika modulů. Oproti monolitické

```
Tag := {id: string, name: string}
Coefficient := 'MIN' | 'MAX' | 'MOST_PROBABLE' | 'EXP' | 'AVG'
EstimateValue := {coefficient: Coefficient, value: number }

EstimateItem := {
    description: string
    tags: Set<Tag>
    items: List<EstimateItem>
    values: Set<EstimateValue>
}

Estimate := {
    id: string
    description: string
    items: List<EstimateItem>
    tags: Set<Tag>
    preconditions: List<{description: string}>
    author: {
        username: string
        name: string
    },
    configuration: {
        timeUnit: 'MD' | 'MH'
        warranty: number
        valueFunctions: Set<{
            coefficient: Coefficient
            definition: string
        }
    }
}
```

Zdrojový kód 4.1: Doménový model odhadu



Obrázek 4.2: Závislost mezi moduly projektu

aplikaci je rozdělena do logických funkčních celků, které jsou reprezentovány moduly. Mezi těmito moduly je možné vytvářet acyklické závislosti. Navíc jsou také mnohem lépe testovatelné než jeden velký monolit. Pokud se například někde v aplikaci vyskytne chyba, stačí ji lokalizovat a opravit jen příslušný modul, který ani nemusí mít dopad na ostatní moduly, které jsou na něm závislé.

V mém případě jsem se aplikaci rozhodl rozdělit do 4 modulů. Závislosti mezi nimi si lze prohlédnout na Obrázku 4.2. Jedná se následující moduly:

- `estimate-domain` – Jedná se o jeden z hlavních modulů, na kterém je většina závislá. Modul totiž obsahuje deklaraci celého modelu odhadu.
- `estimate-export` – Modul je zodpovědný za export modelu odhadu do podporovaných formátů.
- `estimate-backend` – Modul, který je zodpovědný za komunikaci s databází a poskytnutí REST služeb, které umožňují práci s odhadem.
- `estimate-frontend` – Modul poskytující uživatelské rozhraní, ve kterém uživatel může s odhadem pracovat. Jako jediný modul není napsaný v Kotlinu a není závislý ani na jednom modulu.

4.3.3 Obrazovky

V této části se budu zabývat jednotlivými obrazovkami aplikace, jak budou vypadat a co musí obsahovat. Z požadavků na aplikaci 3.4 mi vzešly dvě následující obrazovky. Pro lepší představu o jednotlivých prvcích v uživatelském rozhraní jsem se rozhodl pro obě stránky vytvořit wireframy návrhy.

4.3.3.1 Dashboard

Jedná se o obrazovku, na které uživatel vidí seznam odhadů. V této a několika dalších iteracích budou na této obrazovce zobrazeny všechny odhady, které jsou uložené v databázi. Všechny odhady je nezbytné zobrazit kvůli tomu, že aplikace není momentálně schopna zjistit identitu uživatele. Nelze tak podle něj vyfiltrovat ty relevantní. Odhady jsou na stránce prezentovány v tabulce,

kde jsou v každém řádku zobrazeny informace o odhadu. Tabulka obsahuje několik sloupců, prvním z nich je identifikátor odhadu, dále jeho název a autor. V posledním sloupci je vždy tlačítko Zobrazit. Kliknutím na něj uživatel přejde na obrazovku Detail odhadu 4.3.3.2. Jak stránka bude vypadat, lze vidět na wireframu Obrázku 4.3.

4.3.3.2 Detail odhadu

Druhá je obrazovka, na kterou se uživatel dostane, pokud chce vytvořit nový odhad nebo si zobrazit a případně editovat již existující. Protože se na této obrazovce tvoří odhad, bude v ní uživatel trávit většinu času při používání aplikace Estimate. Obrazovka proto musí být maximálně vyladěná a mít propracované UI. Jak lze vidět na wireframu 4.4, v horní části je zobrazen název odhadu, který je možné editovat, a tlačítko pro uložení odhadu. Nalevo se nachází menu s jednotlivými kategoriemi odhadu (analýza, design, ...), které je možné vybrat. Pod menu je oblast pro konfiguraci odhadu. Zde může uživatel vybrat časovou jednotku a procentuální hodnotu záruky. O zbývající prostor obrazovky se dělí tabulka s položkami aktuálně vybrané kategorie z menu a dolní oblast obrazovky. V dolní oblasti jsou tři záložky, mezi kterými je možné přepínat. Jedná se o celkový přehled pracnosti, jako tomu bylo v Excel šabloně, seznam checklist prvků a editovatelný seznam předpokladů.

4.4 Implementace

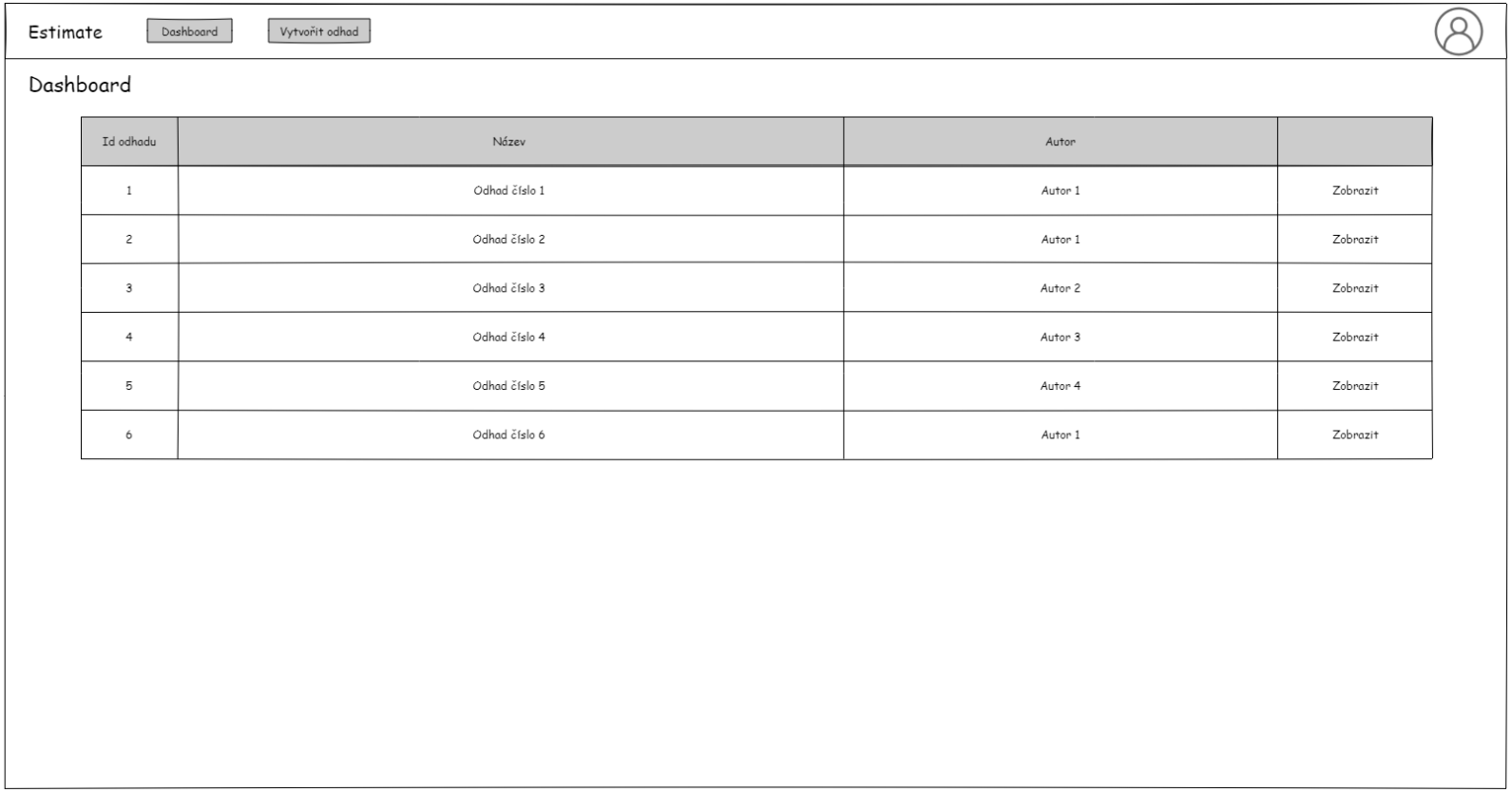
V této části zmíním realizaci samotné aplikace v první iteraci. Část je rozdělená podle logických celků na backend a frontend komponenty.

4.4.1 Backend

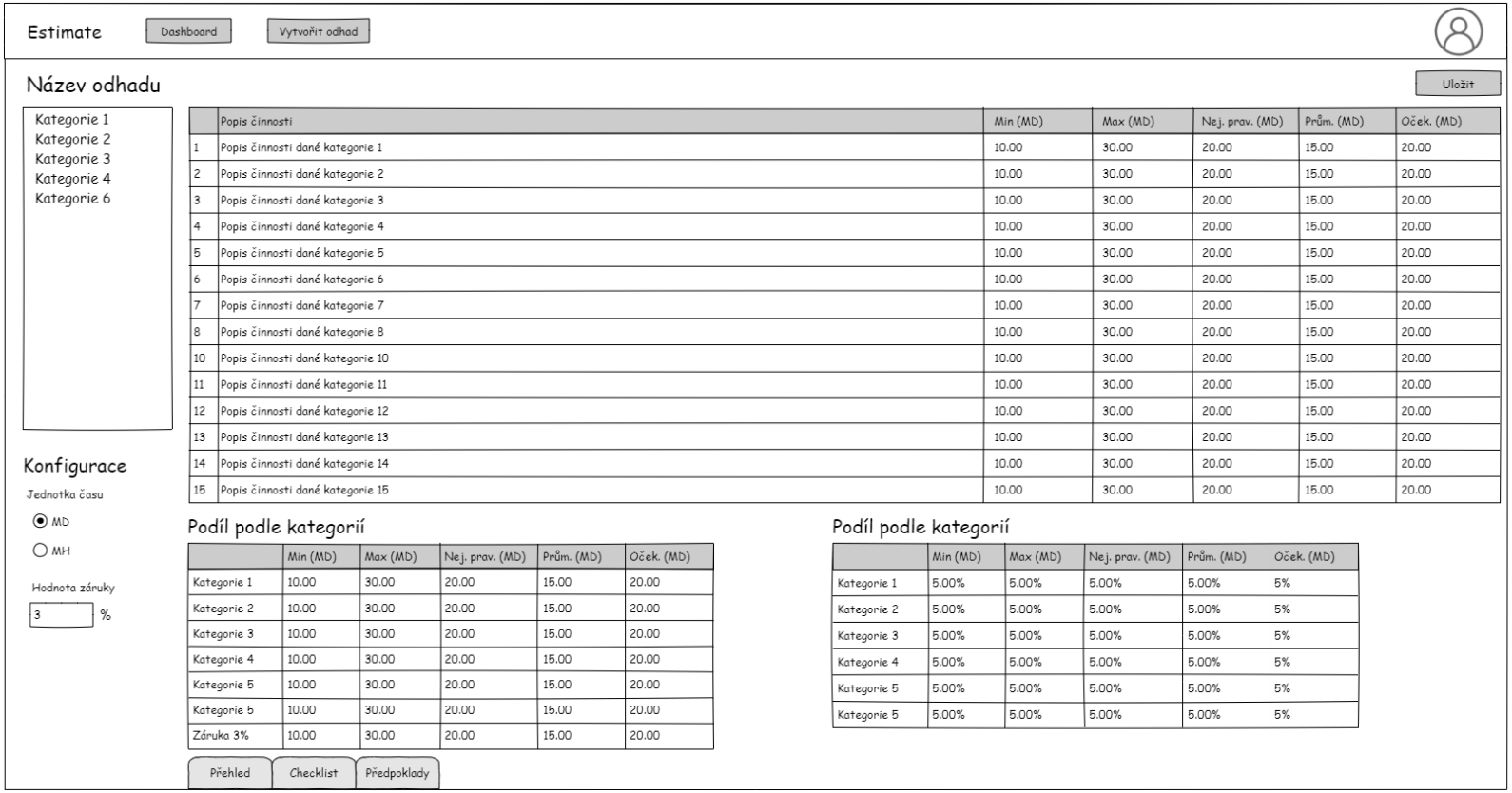
Samotná implementace je zahájena nejprve backend částí. Na začátku je vytvořený Spring Boot projekt s příslušnými moduly (kromě `estimate-frontend`), které jsou popsány v podkapitole Návrh. Celá backend část není implementačně nijak zvlášť náročná, dalo by se říci, že je oproti frontend části až triviálně jednoduchá. Backend část bude totiž v této iteraci poskytovat pouze odhady přes REST API a komunikovat s databází. Frontend část je oproti tomuto úkolu složitější a je na ni neustále co zlepšovat ke spokojenosti uživatele. Backend část se skládá z těchto modulů:

4.4.1.1 Modul `estimate-domain`

Jedná se core modul, na kterém jsou závislé všechny ostatní backend moduly. Modul obsahuje několik tříd, které reprezentují datový model odhadu. Protože jednotlivé třídy v sobě uchovávají pouze jednoduchá data, rozhodl jsem se pro deklaraci tříd využít pro Kotlin specifické datové třídy. S použitím datových



Obrázek 4.3: Wireframe obrazovky Dashboard



Obrázek 4.4: Wireframe obrazovky Detail odhadu

tříd odpadá nutnost psaní metod jako jsou gettery, settery, `equals` a dalších, které mohou být známy z klasických tříd. Navíc jsem se rozhodl celý model reprezentovat jako immutable objekt [8]. Nelze tak měnit data odhadu. Pro odhad s upravenými daty je nutné jej naklonovat s příslušnými daty. K tomu pomáhá v datových třídách poskytovaná metoda `copy`, která vývojáři zjednodušuje naklonování instance třídy. V této metodě je možné při kopírování přepsat volitelně libovolný atribut kopírované instance. Jediné, co je pro datovou třídu potřeba, je nadefinovat v primárním konstruktoru její atributy. Kotlin se pak už o poskytnutí zmíněných metod postará.

Z nejdůležitějších tříd, které reprezentují odhad, stojí za zmínku samotná reprezentace odhadu. Ta je modelována třídou `Estimate` a dále třídou `EstimateItem`, která reprezentuje v odhadu jak jednotlivé kategorie, tak i jejich položky, případně položky položek atd.

Aby mohl být vyplněný odhad vyhodnocen, obsahuje modul kromě samotného modelu také třídu `EstimateEvaluator`. Tato třída je odpovědná za vyhodnocení odhadu na základě vyplněných hodnot položek. K tomu slouží metoda `getEvaluatedModel`, které se předá jako parametr odhadu neboli instance `Estimate`. Protože odhad může mít stromovou strukturu a vyplněné hodnoty mají jen koncové položky, které jsou ve stromu identifikovány jako listy, funguje jeho vyhodnocení směrem od listů ke kořenu. Algoritmus navštíví jednotlivé kategorie, jejich položky, pokud mají i podpoložky, tak i je. Takto jde až ke koncovým uzlům. Když narazí na list neboli na položku, která už žádné další položky nemá, dopočítá hodnoty, jejichž výpočet je závislý na ostatních hodnotách položky, a položku prohlásí za vyhodnocenou. Při zpáteční cestě stromem tuto hodnotu přičte k nejbližšímu rodiči. Obdobně výpočet funguje při vracení se z tohoto uzlu k uzlu rodiče, kdy se opět hodnoty uzlu přičtou. Tímto průchodem se všechny hodnoty dostanou až ke svým nadřazeným kategoriím odhadu. V nich jsou poté uloženy aktuální hodnoty z jejich podstromů. Takto vyhodnocený odhad je poté vrácen metodou. Vstupní odhad, který byl předán v parametrech, zůstává nevyhodnocený neboli nemodifikovaný s původními hodnotami v jednotlivých uzlech.

4.4.1.2 Modul `estimate-export`

Tento modul je zodpovědný za exportování odhadu do různých podporovaných formátů. Ze začátku byl tento modul v plánu až v druhé iteraci, nakonec je ale přidán již v této. Důvod je jednoduchý. Když bude uživatel v budoucnu dělat odhad pomocí Estimate v naší šabloně, může se mu stát, že některá z funkcionalit, která byla v původně používané Excel šabloně, ještě nebude implementovaná. Uživatel by tak musel zbytečně překopírovat odhad do Excelu a s ním dále pracovat podle potřeby. Další možností je do budoucna například export v podobě nějakého reportu, který by mohl být předán rovnou zákazníkovi.

Aby bylo exportování odhadu pro všechny typy exportů sjednocené, rozhodl jsem se nadefinovat v modulu rozhraní `Converter` viz Zdrojový kód 4.2.

```
interface Converter {
    fun convert(estimate: Estimate, destinationPath: String)
}
```

Zdrojový kód 4.2: Rozhraní pro export odhadu

Všechny třídy konvertorů, které umožňují export odhadu do různých formátů, toto rozhraní jednoduše implementují. Díky němu není třeba znát vnitřní implementaci tříd ani jejich veřejné metody. Stačí znát pouze definici rozhraní, ve kterém je jediná metoda `convert`, která export provede.

V současné implementaci je v aplikaci realizovaný pouze export do jednoho formátu. Jedná se o Excel formát s koncovkou `.xlsx`. Ten exportuje odhad do používané Excel šablony. Jedná se o třídu `XLSXConverter`. Ještě předtím, než popíšu, jak export funguje, je třeba zmínit, že Excel je binární soubor a tudíž do něj nelze zapisovat standardní formou jako je tomu například u textových souborů. Pro tento případ, kdy chce programátor modifikovat Excel soubory přímo v kódu, naštěstí existuje několik knihoven. Pro Estimate jsem si vybral knihovnu `org.apache.poi:poi-ooxml`. Jedná se o open-source knihovnu, která je implementována v Javě a vyvíjena společností Apache Software Foundation.

Když jsem přemýšlel nad tím, jak celý export odhadu do Excelu zrealizovat, rozhodl jsem se pro následující řešení. Vzal jsem si původní Excel šablonu a předpřipravil si ji pro programový zápis. Všechny tabulky pro zápis položek kategorií obsahují stejné množství prázdných řádků, řádky jsou nastylovány jednotně, ... Takto připravená šablona je uložena ve složce resources v tomto modulu. Ve třídě `XLSXConverter` a pomocných třídách pro zápis do Excel souboru je implementována logika vyplňování šablony. Logika vyplnění vychází z předpokladů předpřipravené šablony. Potřebné řádky jsou na daných indexech, to samé sloupce. Listy v Excelu jsou pojmenované podle jasných pravidel atd. Díky knihovně je tak možné během vyplňování přidávat další řádky do Excelu, kopírovat buňky (i včetně stylů) a spousta dalších funkcionalit. Přitom jsou po vyplnění zachována všechna makra, formátování a vzorce buněk. Odhad tedy vypadá, jako by ho opravdu vyplňoval uživatel místo stroje.

4.4.1.3 Modul `estimate-backend`

Po core modulu je toto další důležitá část aplikace. Modul je zodpovědný za komunikaci s databází, logiku aplikace a poskytování služeb pomocí REST rozhraní.

Dříve, než zmíním způsob komunikace s databází, je potřeba doplnit, jak se databáze při vývoji spouští. Pro usnadnění vývoje a spouštění testů používám knihovnu `de.flapdoodle.embed.mongo`. Její závislost pro tento modul je přidána v Gradle build skriptu. Díky této závislosti odpadne starost s insta-

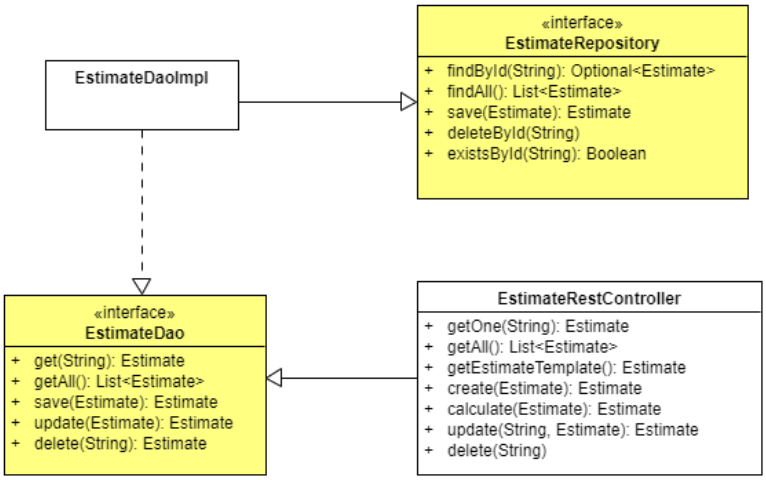
lací databáze MongoDB na lokální vývojový stroj. Knihovna sama při spuštění backend části zajistí stažení implementace MongoDB, její nacachování do lokálních souborů a následně databázi z těchto souborů spustí. Ke stažení implementace databáze dojde pouze při prvním spuštění backend části nebo po smazání nacachovaných souborů. V ostatních případech je tento krok kvůli optimalizaci přenosu dat po síti vynechán. Vývojáři tedy stačí spustit backend část, která se sama postará o spuštění databáze. Po jejím ukončení se obdobně postará o ukončení běhu databáze. Vývojář se tak nemusí o nic starat. Navíc pokud na projekt přijde nová posila, nebude zatížena instalací a zprovozněním databázového serveru. Zároveň vývojáři knihovna šetří čas během implementace, kdy se nemusí starat o mazání a vytváření databáze a jejích kolekcí při změně datového modelu nebo při resetování dat. Pro budoucí produkční prostředí není v plánu používat takto spuštěnou databázi, místo ní bude běžet reálná databázová instance formou služby.

Pro samotnou komunikaci s databází používám Springem přímo nabízená Spring Data. Jedná se o projekt, který vývojářům říká, jak s databází v projektu pracovat, a nabízí unifikovaný pohled a postupy, jak při komunikaci s databází postupovat. Protože jsem si zvolil databázi MongoDB, použiji jenom modul z tohoto projektu, konkrétně knihovnu `spring-boot-starter-data-mongodb`, která ja určená právě pro použití s MongoDB databází. Když mám takto přidanou závislost v projektu, mohu přejít k samotné implementaci přístupu k databázi. Ke komunikaci je použito knihovnou nabízené rozhraní `MongoRepository<T, ID>`, kde `T` je třída dokumentu a `ID` třída jeho identifikátoru. V mém případě bude mít rozhraní generické parametry `Estimate` a `String`, protože do databáze chci ukládat objekty třídy `Estimate`, které mají identifikátor typu `String`. Protože rozhraní `MongoRepository<Estimate, String>` poskytuje větší množství metod, než které budu reálně potřebovat, nadefinuji si pouze podmnožinu metod z tohoto rozhraní do mého vlastního `EstimateRepository`. Toto rozhraní poté budu používat pro komunikaci s databází. Metody rozhraní lze vidět na Obrázku 4.5.

Aby high-level části aplikace nekomunikovaly přímo s repozitářem umožňujícím přímý přístup do databáze, používám vzor Data Access Object. Ten mi umožňuje oddělit nízkoúrovňový přístup k datům v databázi od logiky aplikace. Podle vzoru je tedy vytvořeno rozhraní `EstimateDao` a jeho implementace je `EstimateDaoImpl`. Ta obaluje a volá metody z repozitáře. V logické části aplikace tak není nutné znát přímo metody repozitáře, ale pouze rozhraní `EstimateDao`. Všechny jeho metody jsou znázorněny na Obrázku 4.5.

Komunikace s databází a obalení jejího volání je vyřešené. Zbývá už tedy jen servírování dat pomocí REST rozhraní. Před popisem jeho realizace zde ještě zmíním jeho jednotlivé endpointy, které potřebuje frontend část:

- `GET /rest/estimates` – vrátí seznam všech odhadů
- `GET /rest/estimates/{id}` – vrátí odhad s konkrétním ID



Obrázek 4.5: Prvotní implementační pohled na `estimate-backend` modul

Akce	Anotace
GET	`@GetMapping("path")`
POST	`@PostMapping("path")`
PUT	`@PutMapping("path")`
DELETE	`@DeleteMapping("path")`

Tabulka 4.1: Mapování mezi akcemi a anotacemi controlleru

- `GET /rest/estimates/new` – vrátí prázdný odhad neboli šablonu
- `POST /rest/estimates` – uloží nový odhad z těla požadavku
- `POST /rest/estimates/calculate` – přepočítá odhad a vrátí ho
- `PUT /rest/estimates/{id}` – aktualizuje již existující odhad
- `DELETE /rest/estimates/{id}` – odstraní odhad s konkrétním ID

Když znám jednotlivé endpointy REST API, můžu přejít k jeho implementaci. K těmto účelům je v projektu třída `EstimateRestController`. Třída je označena Spring anotací `@RestController`, což mi umožňuje metody této třídy vystavit jen jako REST rozhraní. Pokud jsou navíc těmito metodami vraceny klasické Java objekty, jsou ve výchozím stavu převedeny na JSON objekty. Metody, které jsou zde implementovány, odpovídají jedna ku jedné jednotlivým endpointům zmíněným výše. V závislosti na akci má každá metoda jednu ze čtyř anotací z Tabulky 4.1 (kde *path* odpovídá url API). Aby jednotlivé metody mohly přistupovat k datům z databáze a modifikovat je, provádí všechny akce ohledně ukládání a modifikace dat přes rozhraní `EstimateDao`.

4.4.2 Frontend

Po dokončení backend části už v této iteraci zbývá jen implementace frontend části. V této části práce se zaměřím na některé implementační detaily frontend části.

4.4.2.1 Inicializace frontend části

Veškerá frontend část aplikace se nachází ve složce `estimate-frontend`. Tato složka je nyní ale prázdná. Nejdříve začnu samotnou inicializací frontend části. Jak už bylo zmíněno v přehledu iterací, aplikace bude postavené na knihovně React v jazyku JavaScript. Při inicializaci projektu vycházím z oficiální dokumentace umístěné na stránkách pro knihovnu React [15]. Pro aplikaci je použit nabízený nástroj *Create React App*. Jak píší samotní autoři: „*Create React App* je pohodlné prostředí pro učení Reactu a je tou nejlepší cestou, jak začít vytvářet novou single page aplikaci v Reactu.“ [15, překlad vlastní] Nástroj je potřeba si nainstalovat. Následně už vývojář jen těží z jeho výhod. K vygenerování kostry aplikace se základní konfigurací stačí nástroj spustit podle dokumentace. Vygenerovaná kostra s konfigurací bude většině uživatelů stačit a nebude nutné provádět žádné složitější změny. Konfigurace v základu obsahuje například skripty pro spuštění, sestavení produkční aplikace a interní závislosti na knihovny, například server *webpack* nebo kontrola syntaxe *ESLint*. Vždy tedy stačí povýšit v souboru `package.json` verzi samotného nástroje místo jednotlivých knihoven, které jsou nástrojem zastřešovány.

4.4.2.2 Stylování komponent

V rámci inicializace frontend části aplikace je třeba se také zamyslet nad tvorbou React komponent a hlavně jejich stylováním. Možnost nepoužít žádnou knihovnu a veškeré styly si pro základní komponenty, jako jsou například tlačítka nebo taby, napsat sám, jsem hned zavrhl. Z časového hlediska by to bylo náročnější a nedávalo mi smysl realizovat něco, s čím už si někdo jiný dal v rámci vývoje knihovny práci. Proto jsem zvolil řešení s využitím knihovny, spíše řečeno frameworku. Tím je framework Bootstrap. Jak sami autoři uvádějí, „Bootstrap je nejpopulárnější HTML, CSS a JavaScript framework pro vývoj responzivních, mobilních projektů na webu.“ [16, překlad vlastní]. Framework navíc znám a mám s ním pozitivní zkušenosti z jiných projektů. V této práci tedy používám jeho React JavaScript knihovnu `react-bootstrap`. Ta v základu obsahuje naimplementované komponenty z frameworku Bootstrap. Jsou zde například `Button`, `Navbar`, `Modal` a mnoho dalších. Knihovna obsahuje Bootstrap stylování. Komponenty jsou podle něj již nastylované a mají vlastní `props`, které se liší podle potřeb dané komponenty.

4.4.2.3 Směrování URL

V aplikaci jsou plánovány celkem tři stránky. Není proto možné, aby se uživateli při otevření aplikace otevřela vždy jedna vybraná stránka, v tomto případě Dashboard s odhady. Uživatel by musel vždy přejít na detail existujícího odhadu ze seznamu nebo na nové vytvoření o odhadu přes menu. Navíc by nebylo možné sdílet existující odhady pomocí URL. Z tohoto důvodu je potřeba zavést knihovnu pro směrování.

Pro React takováto knihovna existuje, jmenuje se `react-router`. Knihovna umožňuje zobrazovat komponenty jako obrazovky v závislosti na URL v prohlížeči a tím řídit tok v aplikaci. Směrování funguje následovně. Uživatel zadá pomocí komponent knihovny pro jednotlivé adresy komponenty neboli obrazovky, které se mají vykreslit. Následně si při zadání URL React při renderování obsahu vyhodnotí, která URL je momentálně v prohlížeči , a na jejím základě spáruje jednotlivé podmínky. Následně vybere první odpovídající komponentu v pořadí a tu vykreslí.

V aplikaci jsou následující stránky:

- `/` – tato stránka je přesměrována na Dashboard
- `/dashboard` – Dashboard
- `/estimates/new` – Vytvoření nového odhadu
- `/estimates/detail/{id}` – Detail existujícího odhadu

Jak je vidět, většina stránek má statickou adresu. Jenom Detail existujícího odhadu má dynamickou adresu, kde se mění identifikátor odhadu. Toto není pro knihovnu žádný problém, ta umí směrovat i dynamické stránky. Ve Zdrojovém kódu 4.3 je možné vidět část kódu, jak knihovnu používám v mé diplomové práci. Kód obsahuje všechna pravidla směrování pro výše zmíněné stránky.

4.4.2.4 Komunikace s backend částí

Aby mohla frontend část komunikovat s backend částí, je potřeba na straně frontend části zajistit klienta, který se bude starat o posílané požadavky na backend část a adekvátně poté zpracovávat odpovědi. Před samotnou implementací jsem vybíral mezi dvěma nástroji. *Axios* a *Fetch API*.

Prvně zmíněný *Axios* je knihovna, která poskytuje HTTP klienta pro JavaScript a Node aplikace. Klient je v tomto případě asynchronní a umožňuje uživateli posílat na server HTTP požadavky s velkou možností konfigurovatelných hodnot.

Oproti tomu *Fetch API* není knihovna. Toto API je nabízeno základním rozhraním JavaScriptu v prohlížeči, není tak třeba přidávat závislost na další knihovně a zvětšovat velikost aplikace. Stejně jako předchozí knihovna nabízí

```
<Switch>
  <Route path="/estimates/detail/:id" component={EstimateArea} />
  <Route path="/estimates/new" component={EstimateArea} />
  <Route path="/dashboard" component={Dashboard} />
  <Route
    exact
    path="/"
    component={() => <Redirect to="/dashboard" />}
  />
</Switch>
```

Zdrojový kód 4.3: Definice přesměrování pomocí `react-router`

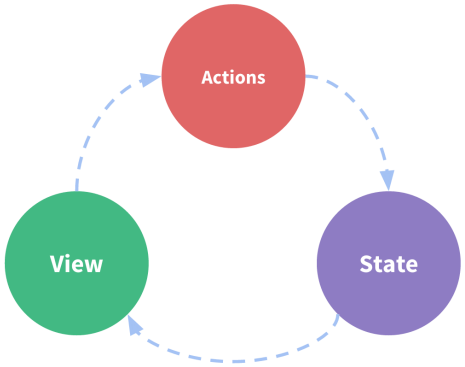
nástroj, přes který je aplikace schopná posílat na server HTTP dotazy s širokou škálou hodnot.

Než jsem vybral jedno z řešení, porovnal jsem si oba nástroje. Z porovnání mi vzešly následující poznatky. *Fetch API* je podporováno v prohlížečích až od jejich určitých verzí. Mohlo by se tak stát, že pokud někdo nemá aktualizovanou verzi prohlížeče na minimální podporovanou verzi, aplikace by vůbec nefungovala. V tomto případě má *Axios* jasnou výhodu, protože se jedná o knihovnu, a proto poběží i na starších verzích prohlížečů, kde by konkurent nefungoval. Další výhodou pro *Axios* je, že JSON data přijatá v odpovědi serveru se automaticky transformují do odpovídajícího JavaScript objektu. Vývojář nemusí jako v případě *Fetch API* pokaždé převádět data z odpovědi na JSON objekt a ušetří si tím práci. Poslední výhodou pro *Axios* je zpracování error handlingu. Protože obě knihovny jsou asynchronní, vracejí `Promise` objekt, se kterým se dále pracuje. V případě *Fetch API* se v metodě `then` zpracují odpovědi serveru s jakýmkoliv stavovým kódem, tedy i chybným. *Axios* má tuto funkcionalitu z mého hlediska lépe vyřešenou. Všechny odpovědi s chybovým stavovým kódem jsou zpracovávány v metodě `catch`, do metody `then` se tak dostanou jen odpovědi s nechybovými stavovými kódy.

Po porovnání obou nástrojů je pro mě jako jasný vítěz knihovna *Axios*, která oproti svému konkurentovi nabízí více výhod. Z toho důvodu je v projektu tato knihovna použita.

4.4.2.5 Kde odhady ukládat

Aby aplikace mohla fungovat správně, je třeba si dočasně někam stažené odhady ze serveru ukládat. V mém případě tento problém řeším pomocí knihovny *Redux*. Tato knihovna umožňuje v aplikaci ukládat data na jednom místě, v takzvaném *storu*. Většina komponent je na tento *store* napojena a všechny informace získávají z něj. Kromě výhody úložiště na jednom místě přináší toto řešení i další funkcionalitu. Tou je, že pokud je daná komponenta napojena na



Obrázek 4.6: Tok dat přes *Redux store* [17]

nějakou část stavu ze *storu* a tato část se změní, komponenta na to zareaguje a na základě této změny se sama překreslí. Pro ukázku uvádím příklad. Uživatel zadá novou položku do tabulky s položkami odhadu, konkrétně upraví číselnou hodnotu. Ta se uloží následně do *storu*. Aby se tato změna projevila v tabulkách přehledu, tabulky se kvůli změně překreslí a obsahují tak opět aktuální hodnoty.

A jak to tedy celé funguje? Aplikace se pomyslně skládá ze tří částí:

- Stav – aktuální stav *storu*
- Pohled – popis UI podle aktuálního stavu
- Akce – akce, které jsou spouštěny na základě uživatelského vstupu

Nejlépe to půjde ukázat na příkladu stránky s detailem odhadu. Uživatel spustí aplikaci na URL odpovídající existujícímu odhadu. Ve *storu* zatím nejsou žádná data o odhadu, tudíž se zobrazí progress bar s informací o načítání odhadu. Odhad se tedy začne na pozadí stahovat ze serveru. Po jeho stažení se vyvolá akce uložení odhadu do *storu*. Akce se zde zpracuje a podle logiky se změní hodnota odhadu ve stavu, a tím se celý stav aktualizuje. Na základě této změny se následně zobrazí stránka s komponentami detailu odhadu. Zde může uživatel měnit různé hodnoty odhadu, přepínat mezi kategoriemi a provádět další možné akce. Při takovéto změně vždy dojde k předchozímu scénáři, do *storu* se pošle informace např. o změněné záruce odhadu a postupuje se jako v předchozím případě. Výsledkem pak je, že hodnota záruky je v zadávacím políčku aktualizována a zároveň jsou přepočítány přehledové tabulky v dolní částí obrazovky. Na tomto principu funguje celá komunikace se *storem*. Data tedy tečou pouze jedním směrem a přes místo, kde jsou data jednotně uložena. Celé flow lze vidět na Obrázku 4.6.

4.4.2.6 Knihovna Handsontable

Komponenta `HotTable` z knihovny Handsontable[14] nepodporuje přímé napojení na stav z *Redux storu*. Proto jsem se rozhodl pro následující postup. Při renderování příslušné komponenty, v níž je `HotTable` použita, se vždy získá aktuální odhad. Z tohoto odhadu se získá požadovaná kategorie. Aktuálně vybraná kategorie v menu je také uložena ve *storu*. Takto získané položky kategorie jsou následně přemapovány do požadováného formátu, se kterým umí tabulka pracovat. Současně je komponentě předán i objekt s konfigurací tabulky. V ní je definováno, kolik sloupců má, jaké jsou hlavičky a jak jsou široké. Zároveň je zde i definováno kontextové menu tabulky, které se zobrazí po kliknutí na pravé tlačítko v komponentě. Tímto je vyřešena situace ohledně renderování dat k zobrazení.

Zobrazení mám již vyřešené. Co je ale rovněž důležité, je uložení změn v tabulce po editaci uživatelem a jejich propsání do *storu*. Komponenta nabízí možnost, jak k ní dostat jednotlivé obslužné metody, které jsou zavolány při různých akcích. V mém případě se jedná o metody:

- `afterChange` - Umožňuje mi reagovat na změnu dat v konkrétním řádků v tabulce. Funkce má několik parametrů. Jedním z nich jsou i konkrétní změny na řádku. Z těchto změn jsem tedy schopný sestavit nová data pro řádek a ty poslat do *storu*. Dojde tak automaticky k přepočítání a překreslení tabulky a dalších komponent, které čtou data z odhadu.

- `afterCreateRow` – Díky této metodě jsem schopný zachytit přidání nového řádku a provést příslušné akce do *storu*. Dojde k překreslení jako v 1. případě.

- `afterRemoveRow` – Tato props mi umožňuje reagovat na smazání existujícího řádku. Následně je provedena odpovídající akce do *storu*. Dojde k překreslení jako v 1. případě.

4.4.2.7 Stránka s editací odhadu

Z první iterace je nejzajímavější stránka s detailem odhadu. Na tuto stránku se uživatel dostane buď při vytváření nového odhadu nebo při editaci existujícího. Oba zmíněné případy se poté liší jen URL v prohlížeči a endpointem, kam se odhad ukládá. Při příchodu na stránku se uživateli nejdříve zobrazí progress bar a na pozadí se stáhne v závislosti na akci šablona odhadu nebo existující odhad podle identifikátoru z URL. Po jeho stažení se zobrazí uživatelské rozhraní, které bylo představeno v sekci 4.3.3. Poté již může uživatel editovat odhad, jak je mu libo.

V současném uživatelském rozhraní může tedy uživatel v levém menu měnit vybranou kategorii. Po její změně se vždy vybere checklist dané kategorie v dolní části a v tabulce napravo zobrazí adekvátní položky kategorie. Ty

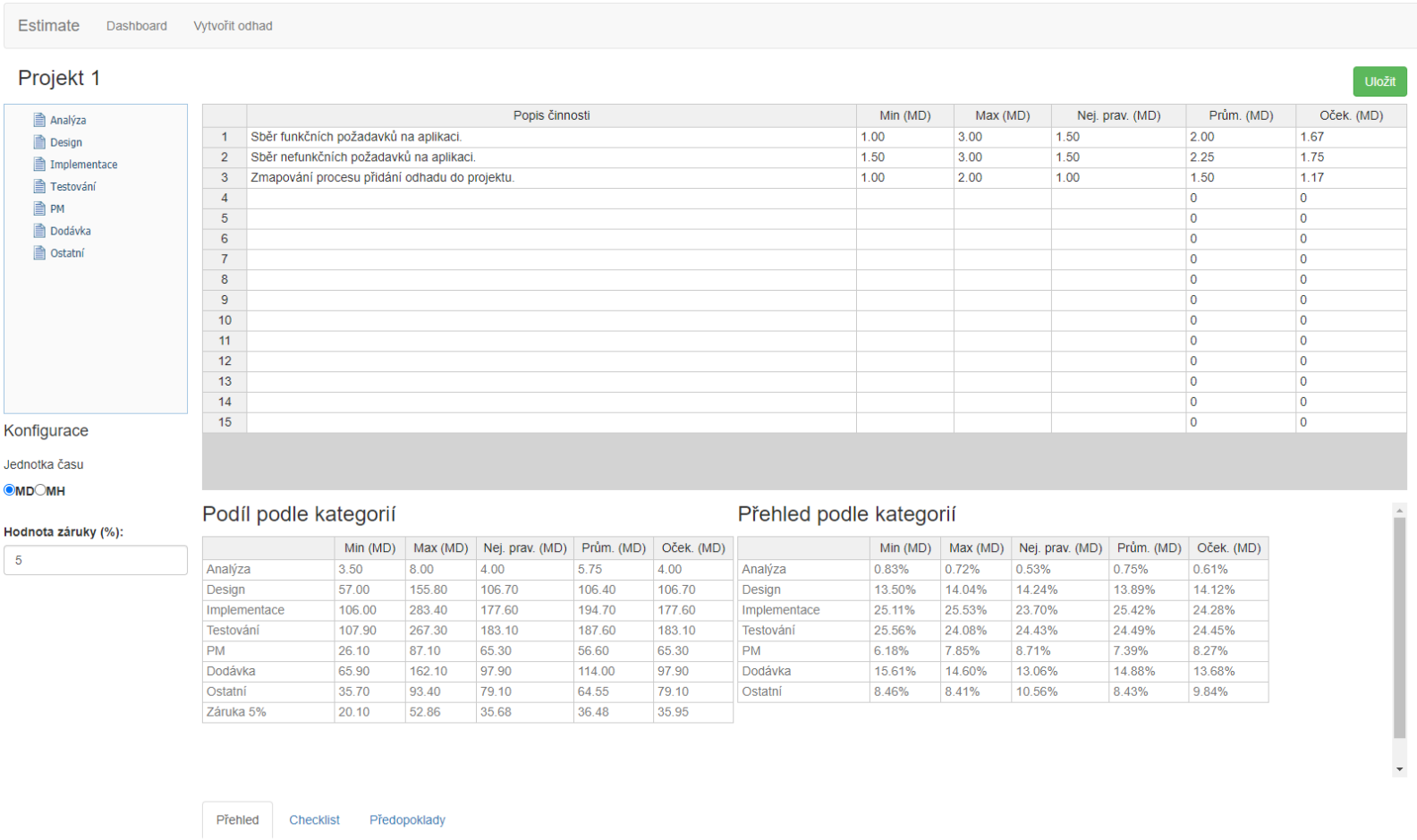
může uživatel modifikovat dle libosti, přidávat či odebírat řádky. V této komponentě tabulky bude uživatel trávit nejvíce času, proto musí být její funkcionalita bezchybná. Například pokud uživatel zadá do buňky, kde se očekává číslo, text, buňka po uložení bude prázdná a daný text zde neuloží. Po každé změně a jakékoliv další změně dojde k poslání akce do *Redux storu*. V něm se daná akce aplikuje na aktuální stav, ve kterém je uložen odhad. Tento odhad bude zároveň před aktualizací stavu přepočítán. Výsledkem pak je, že tabulka s celkovým přehledem v dolní částí obrazovky bude mít aktuální data.

Takto se tomu ale nemělo dít od začátku vývoje. Na jeho počátku byl scénář ohledně provedení výpočtu odlišný. Výpočet všech hodnot odhadu měl probíhat pouze na straně backend části. Frontend část měla pouze umožňovat odhad tvořit, tj. definovat předpoklady, nastavit konfiguraci a zadávat položky jednotlivých kategorií. Frontend část by byla v tomto případě úplně odstíněna od nutnosti vědět, jak odhad vyhodnotit. Zprvu se dané jevilo jako dobrá myšlenka, nicméně během implementace se tento přístup ukázal jako zcela nevhodné řešení. Uživatel sice mohl zadávat položky odhadu. Co ale v průběhu neviděl a co je z hlediska odhadů důležité, je souhrnná pracnost dané kategorie vůči ostatním kategoriím. Tato znalost je při tvorbě odhadu poměrně důležitá. Pomáhá uživateli korigovat správnost odhadu, aby jednotlivé kategorie nebyly co se týče pracnosti odlišné v řádu desítek procent. Pokud chtěl uživatel tento údaj znát, musel zmáčknout tlačítko vyhodnotit. To způsobilo odeslání odhadu k výpočtu na příslušný endpoint backend části. Odpovědí byl vyhodnocený odhad, který se uložil do *storu*. Po této akci již uživatel viděl vyhodnocený odhad v uživatelském rozhraní. Neustálé posílání odhadu na vyhodnocení se mi zdálo zbytečně komplikované a uživatel si vyhodnocení musel vždy vynutit. Znamenalo to tedy kliknutí navíc a nějaký čas na odeslání a přijmutí odpovědi. Z tohoto důvodu jsem od výpočtu na backend části upustil. Místo toho je nyní odhad vyhodnocen na straně frontend části. Jeho vyhodnocení je tak implementováno přímo v jazyce JavaScript ve třídě `utils/EvaluateUtils`. Komunikace s backend částí je tak omezena pouze na získávání šablony odhadu, existující odhad a jeho následné vytvoření či uložení.

V tomto stavu, který je popsán v této podkapitole, se aplikace nachází po první iteraci. To, jak vypadá výsledné UI, je možné si prohlédnout na Obrázku 4.7.

4.5 Testování

V dalších iteracích je primárně plánován rozvoj na frontend části. Proto v této iteraci nejsou vytvořeny žádné testy ze strany React frontend části. Testovat UI a komponenty, které se mohou a pravděpodobně i v budoucnu výrazně měnit budou, mi v této fázi projektu nedává smysl. Prozatím se spokojím s manuálním průchodem UI aplikace, abych otestoval všechny funkcionality.



	Popis činnosti	Min (MD)	Max (MD)	Nej. prav. (MD)	Prům. (MD)	Oček. (MD)
1	Sběr funkčních požadavků na aplikaci.	1.00	3.00	1.50	2.00	1.67
2	Sběr nefunkčních požadavků na aplikaci.	1.50	3.00	1.50	2.25	1.75
3	Zmapování procesu přidání odhadu do projektu.	1.00	2.00	1.00	1.50	1.17
4					0	0
5					0	0
6					0	0
7					0	0
8					0	0
9					0	0
10					0	0
11					0	0
12					0	0
13					0	0
14					0	0
15					0	0

	Min (MD)	Max (MD)	Nej. prav. (MD)	Prům. (MD)	Oček. (MD)
Analýza	3.50	8.00	4.00	5.75	4.00
Design	57.00	155.80	106.70	106.40	106.70
Implementace	106.00	283.40	177.60	194.70	177.60
Testování	107.90	267.30	183.10	187.60	183.10
PM	26.10	87.10	65.30	56.60	65.30
Dodávka	65.90	162.10	97.90	114.00	97.90
Ostatní	35.70	93.40	79.10	64.55	79.10
Záruka 5%	20.10	52.86	35.68	36.48	35.95

	Min (MD)	Max (MD)	Nej. prav. (MD)	Prům. (MD)	Oček. (MD)
Analýza	0.83%	0.72%	0.53%	0.75%	0.61%
Design	13.50%	14.04%	14.24%	13.89%	14.12%
Implementace	25.11%	25.53%	23.70%	25.42%	24.28%
Testování	25.56%	24.08%	24.43%	24.49%	24.45%
PM	6.18%	7.85%	8.71%	7.39%	8.27%
Dodávka	15.61%	14.60%	13.06%	14.88%	13.68%
Ostatní	8.46%	8.41%	10.56%	8.43%	9.84%

Obrázek 4.7: UI na konci 1. iterace

Navíc funkcionalit zatím není mnoho, a proto takovéto otestování není časově náročné a nepotřebuje žádné složité scénáře.

Co už je ale potřeba mít otestováno, je backend část aplikace. Zde se nachází jak export odhadu do Excel formátu, tak model odhadu a jeho vyhodnocení. Nesmím zapomenout ani DAO vrstvu a samotný Controller, starající se o servírování dat přes REST API. Například export odhadu už se v budoucnu nebude moc měnit. Navíc testy během vývoje usnadní jeho vývoj a vývojář si bude jist, že export funguje správně. Obdobně tomu je i u dalších funkcionalit jako je například vyhodnocení odhadu. Proto jsou všechny funkcionality v aplikaci řádně otestovány pomocí unit testů. Pokrytí jednotlivých modulů testy dosahuje hodnot mezi 90-96 %, což mi přijde v pořádku.

4.6 Zhodnocení

V rámci první iterace se podařilo dosáhnout očekávaného výsledku. Aplikace se nyní nachází ve stavu, kdy je v ní možné provést jednoduchý odhad. Během vývoje se objevily některé problémy, jako je například potřeba výpočtu odhadu lokálně. Všechny tyto překážky se ale podařilo vyřešit a nyní neblokují další vývoj. Po funkční stránce se aplikace vyrovná okleštěné verzi Excel šablony. Chybí zde například helper funkce nebo možnost přidání varianty, ty budou nicméně v dalších iteracích přidány. Po designové stránce obsahuje ještě některé nedostatky. Ty bude třeba v dalších iteracích vyladit ke spokojenosti uživatele. V rámci zhodnocení této iterace bude zkušenějším kolegou provedena revize celkového návrhu a kódu aplikace. Pokud by se přišlo na nějaká zásadní pochybení, nebylo by v této fázi vývoje ještě tak časově náročné změnit některá rozhodnutí a změny v aplikaci provést. Výsledky revize budou zmíněny v další iteraci.

Kapitola 5

2. iterace

V předešlé iteraci jsem dokončil základní část aplikace. V té je uživatel v současné chvíli schopný vytvořit základní odhad. V této kapitole se soustředím na to, jak aplikaci dále rozšířit o další potřebné funkcionality. Zároveň se předpokládá oprava chyb, které vznikly během předchozí iterace, ale ještě se na ně nepřišlo. V kapitole se tedy soustředím na činnosti, které bude třeba provést během druhé iterace projektu.

Na začátku této iterace se ve firmě uvolnily kapacity na jednom projektu. Obecně, jako v každé jiné firmě, převládá snaha přiřazovat vývojáře na projekty zákazníků. Žádný takový vhodný v současné době ale není. A protože je Estimate interní firemní projekt, rozhodlo se, že mi bude do týmu přidělen jeden vývojář. Od této chvíle tedy nebudu řešit jen samotnou realizaci aplikace. Protože jsme v týmu dva vývojáři, je třeba řešit i plánování a rozdělení práce mezi členy týmu, za které jsem odpovědný.

5.1 Podpůrné činnosti projektu

V této podkapitole se zaměřím na činnosti, které nejsou součástí finálního výsledku projektu Estimate, nicméně mají na jeho výsledek částečný vliv.

5.1.1 Dokumentace

Jak jsem psal již v úvodu kapitoly, do týmu se ke mně na začátku této iterace připojil další člen vývojového týmu. Díky tomu se ukázala důležitost celkové dokumentace jako například stručný popis projektu, jak si aplikaci spustit a jak ji vyvíjet. Když nový člen nastoupil, bylo nutné s ním projít celý projekt. Vysvětlit architekturu aplikace, popsat, kde se co provádí, proč to tak je a na základě čeho jsem se pro to rozhodl. Dále ukázat, jak si celý projekt sestavit a spustit. Poslední věcí bylo dodržování vývojářských pravidel při úpravách či přidávání nového kódu do aplikace jako je správný code style, pojmenování proměnných atd. Na základě těchto zkušeností a feedbacku po novém nástupu

na projekt jsem se proto rozhodl vytvořit dokumentaci formou Wiki stránek v používaném systému GitLab. Dokumentace obsahuje několik stránek, na kterých jsou jak informace o projektu, jeho struktuře a použité architektuře, tak i návod, jak si aplikaci spustit v lokálním vývojovém prostředí a čeho se držet během vývoje. Pokud se tedy v budoucnu do projektu začlení další vývojář, bude moci postupovat podle návodu na Wiki. Ušetří se tak čas jak pro nového vývojáře, tak i pro současného, který by novému vývojáři musel všechny informace předávat osobně. Takto si nový člen týmu pročte celou Wiki, a v případě problému se zeptá, a pokud informace na Wiki nejsou úplné, budou poté doplněny.

5.1.2 Testovací prostředí

Aplikace je po první iteraci ve stavu, kdy je uživatel schopen vytvořit jednoduchý odhad. Mohou ji tak v omezeném režimu používat běžní uživatelé, pro které je určena. Byla by tedy škoda neposkytnout kolegům tento nástroj. Vývoj ale probíhá na plné obrátky, další klíčové funkcionality budou teprve implementovány a funkčnost aplikace je stále laděna. Nedává proto v této fázi smysl vytvářet již produkční prostředí.

Abych tedy uživatelům nabídl aplikaci k vyzkoušení a zároveň umožnil vývojovému týmu mít aplikaci nasazenou k otestování, rozhodl jsem se, že vytvořím testovací prostředí. V něm budu mít nasazenou aplikaci vždy v nejnovější verzi. Po vytvoření testovacího prostředí navíc budu mít možnost získávat zpětnou vazbu i od samotných uživatelů a na jejím základě aplikace neustále vylepšovat.

Pro potřeby testovacího prostředí jsem se rozhodl využít firmou přidělený virtuální server s Linux distribucí CentOS. Tento stroj je dostupný z interní sítě. Protože k němu není přístup zvenčí, nestane se, že by se do aplikace, ve které ještě není realizováno přihlášení, dostal uživatel, který není zaměstnancem. To, jak aplikace na tomto prostředí poběží a jak bude nasazována nejaktuálnější verze, je zmíněno v následující části se zaváděním Continuous Integration.

5.1.3 Continuous Integration

Další věcí, kterou je nutné zavést, je kontinuální integrace, známější pod názvem Continuous Integration. „Kontinuální integrace je vývojová praxe, která vyžaduje po vývojářích, aby několikrát denně integrovali kód do sdíleného repozitáře. Každý takto integrovaný kód je poté ověřen pomocí automatického sestavení, což umožňuje týmům brzy detekovat problémy. Pravidelnou integrací můžete rychle detekovat chyby snáze je lokalizovat.“ [18, překlad vlastní] Jedná se tedy o sadu nástrojů a postupů k urychlení vývoje softwaru. Od vývoje, přes testování až po nasazení aplikace. Typicky se tato činnost provádí několikrát denně. Doposud jsem žádné testovací prostředí neměl a na projektu

jsem pracoval sám. Zprovoznění CI tedy nebylo velkou prioritou. Veškerý kód v repozitáři byl pouze ode mne. Abych si udržel funkčnost kódu napříč přidáváním nových funkcionalit, vždy jsem před commitem nových změn spustil testy. Ověřil jsem si tak, že přidáním nové funkcionality jsem nechtěně nezanesl do kódu chybu a aplikaci jsem měl otestovanou.

K zavedení CI mě v této fázi projektu vede hned několik důvodů. Prvním z nich je, že v rámci této iterace bylo vytvořeno testovacího prostředí. Nasazovat tak aplikaci po každé změně ručně by bylo otravné a navíc by nebylo zajištěno její otestování, pokud by se aplikace před nasazením zapomněla otestovat. Navíc CI v dnešní době patří již mezi best practices každého projektu. Není tedy důvod, proč CI nemít. Poslední a nemalou motivací je fakt, že na projektu nyní bude pracovat více vývojářů a mohlo by docházet k situacím, kdy některé testy neproběhnou v pořádku. Jednalo by se například o situaci, kdy jeden vývojář přidá novou funkcionalitu, ale spolu s ní zavede do projektu chybu a před commitnutím změn zapomene pustit testy. Nevšimne si tak chyby. Druhý vývojář si stáhne změny z repozitáře a začne implementovat další funkcionalitu. Když je funkcionalita hotová, spustí testy ještě před finálním commitem. Zde ale zjistí, že je v projektu chyba. A nastává její dlouhé hledání, kde je v implementaci chyba. Ta tam ale byla již na začátku kvůli chybě prvního vývojáře. K tomuto problému by ale pravděpodobně nemuselo dojít, kdyby bylo na projektu zavedeno CI.

5.1.3.1 GitLab CI

Z výše zmíněných důvodů jsem se proto rozhodl CI zavést již teď. Zvažoval jsem několik možností, jak CI zavést. Nakonec jsem se rozhodl využít CI nástroje poskytované systémem GitLab. V tomto systému mám nyní jak samotné verzování projektu, tak i stránky s dokumentací. Proto byly GitLab CI nástroje jasnou volbou.

Nyní se zaměřím již na samotné nastavení kontinuální integrace. Protože jsem si vybral nástroj GitLab, pro zavedení na projektu stačí provést následující dva kroky.

1. Je třeba nejprve nainstalovat a posléze zaregistrovat takzvaný *GitLab Runner*. Jedná se o aplikaci, který spouští jednotlivé joby v GitLab CI pipeline, které jsou na ni delegovány z GitLab systému. V mém případě bude *Runner* nainstalován na serveru, na kterém poběží i testovací prostředí.

2. Je potřeba na server, kde poběží Runner, nainstalovat nástroj Docker [19]. Pro ten jsem se rozhodl, protože sestavení aplikace nechci provádět přímo na tomto serveru. Díky nástroji Docker stačí použít správný obraz s vhodně zvolenými technologiemi. Tím se vyhnu instalaci potřebných nástrojů přímo na server. Po změně technologií bude tedy stačit pouze změnit obraz.

3. Vytvořit v kořenové složce GitLab repozitáře soubor `.gitlab-ci.yml`. V tomto souboru budou nadefinovány pomocí GitLab syntaxe jednotlivé kroky kontinuální integrace.

Při konfiguraci jsem se rozhodl pro celkem 3 etapy CI procesu. Tyto etapy jsou spuštěny vždy po odeslání změn do repozitáře. Pokud některá z etap skončí chybou, další se již neprovádí. Jedná se o **build**, **test** a **deploy** etapy, které se spouštějí ve stejném pořadí, v jakém jsou zmíněny.

1. **build** – Fáze, která je spouštěná pomocí Docker image na všech větvích. Na jejím konci je sestavena backend část aplikace. Je tedy otestováno, že v aplikaci nejsou chyby a je možné ji sestavit.

2. **test** – Druhá fáze je také pouštěna v Dockeru ve všech větvích v repozitáři. V rámci jejího běhu se spustí testy nad backend částí.

3. **deploy** – Fáze, ve které se provádí nasazení na testovací prostředí. Protože pouze v master větvi je zaručeno, že se jedná o nejnovější funkční verzi aplikace, spouští se pouze při změnách v této větvi. Při této etapě je spouštěn Shell skript. Ten sestaví pomocí Docker odděleně kontejnery pro backend a frontend část. Po jejich vytvoření je skript přenasadí. Výsledkem je nasazení nejnovějších změn do testovacího prostředí.

Takto kontinuální integrace fungovala na projektu několik týdnů. Poté se bohužel ukázalo, že při větším množství odeslaných změn do repozitáře trvá celý CI proces poměrně dlouhou dobu, protože se v něm provádí zbytečně moc sestavení aplikace. Rozhodl jsem se proto provést několik změn. Za prvé už není odděleno sestavování backend a frontend části. Místo toho je frontend sestavován v rámci backend části. Druhou a nejzásadnější změnou je, že aplikace nasazená v testovacím prostředí již neběží jako Docker služba. V první konfiguraci CI se totiž v rámci nasazení na testovací prostředí prováděl znovu build celé aplikace a nasazení proto trvalo příliš dlouho. Dalším mínusem byl ztížený přístup k log souborům, které byly umístěny uvnitř Docker kontejneru a jejich archivace napříč kontejnery by byla složitá. Nově se v etapě **deploy** pouze získá kompletně sestavená aplikace z **build** etapy. Není tedy zbytečně znovu sestavována, jako se tomu dělo dříve. Následně je aplikace nasazena do testovacího prostředí jako plnohodnotná systémová služba.

Pro představu, jak je v souboru `.gitlab-ci.yml` definováno pořadí etap a samotná etapa **build**, přikládám ukázku ve Zdrojovém kódu 5.1.

5.1.4 Code review

Jak bylo zmíněno na konci první iterace, v rámci jejího ukončení bylo provedeno kolegou code review navržené architektury a dosud implementovaného kódu. Z revize vzešlo několik připomínek, které budou zváženy a přepracovány.

```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - chmod +x ./gradlew
    - ./gradlew --build-cache build -x test
  tags:
    - estimate-docker
```

Zdrojový kód 5.1: Část definice GitLab CI

Na straně frontend části se jednalo například o doporučení snažit se používat co nejvíce proměnné typu `const` před proměnnými `let`.

Na podnět kolegy je naplánováno zavést nástroj `estlint`, který provádí statickou analýzu kódu. Je třeba si proto nastudovat, jak nástroj na projektu zprovoznit. Aby byla statická analýza zároveň podle pravidel, která jsou na projektu definována, je třeba si dále nastudovat možnost konfigurace a nadefinovat pravidla na základě doposud používané syntaxe kódu.

Poslední otázka, která v rámci code review padla, byla, zda jsem nepřemýšlel na frontend části o typové kontrole objektů. Je pravdou, že jak projekt roste, zvětšuje se i rozsah celé aplikace a rozrůstá se počet objektů. Tímto tempem by stejně za pár týdnů byla typová kontrola potřeba, protože pamatovat si veškeré klíče v objektech, typy parametrů atd. prostě nelze. Navíc pokud vývojář udělá chybu v typování, nemusí se na to během vývoje vůbec přijít a zjistit chybu až v produkční aplikaci. Zmapoval jsem situaci ohledně knihoven či lépe řečeno frameworků, které typovou kontrolu do JavaScriptu přidávají, a rozhodl se vybrat si knihovnu *flow*.

5.2 Analýza

Nyní se přesouvám k analýze samotné aplikace. V rámci analýzy jsem se zaměřil převážně na obrazovku s detailními informacemi o odhadu. Na této obrazovce chybí momentálně některé funkcionality, které je třeba doimplementovat, než aplikace půjde do testovacího prostředí, aby ji tam mohli uživatelé otestovat. Jedním ze současných nedostatků aplikace je, že odhady v UI lze rozlišovat jen na základě názvu. To bude třeba změnit. Po zvážení možných dalších doplňujících informací jsem se rozhodl nabídnout uživateli možnost si přidat k odhadu tagy, kratší popis a jméno autora. Všechny údaje kromě

krátkého popisu v modelu již jsou. Získání jména autora bude v tuto chvíli realizováno pouze přidáním textového pole, do kterého uživatel vyplní své jméno. Samotný uživatel bude řešen v dalších iteracích. V tuto chvíli mi nevadí, že uživatel může zadat jakékoliv jméno a zároveň přistupovat k libovolnému odhadu. Aplikace sice poběží na testovacím prostředí, ale budou v ní vytvářeny jen testovací odhady. Zde se bude spoléhat na čestnost uživatelů, že si editují pouze jimi vytvořené odhady. Dočasné opomenutí přihlášeného uživatele mi umožní soustředit se na jiné funkcionality s momentálně vyšší prioritou. Další přidanou funkcionalitou, kterou bude třeba na testovacím prostředí mít, je samotný export. Pokud si uživatel vytvoří odhad, bude si ho chtít na konci pravděpodobně stáhnout k sobě do počítače. Proto nově bude v UI nabídnuto stažení odhadu ve formě Excel souboru.

Když jsem si v aplikace z první iterace zkoušel vytvářet odhad, všiml jsem si také, že aplikace mě neupozorní, pokud zadám špatná čísla k jednotlivým položkám. Toto je nezbytné v rámci této iterace vyřešit. Zmiňovaný případ demonstruji na následujícím příkladu. Vyplňuji položku a zadám její minimální pracnost, v tomto případě 4. Dále vyplním její maximální odhadovanou pracnost hodnotou 2. Jak je vidět, minimální pracnost je vyšší než její maximální. Pokud takovýto případ nastane, je dobré uživatele upozornit, že jedno ze zadaných čísel je pravděpodobně špatně. Minimalizují se tím chyby či překlepy při zadávání pracností.

Doposud je možné v aplikaci pracovat pouze v jednom jazyce, a to v češtině. Žádný další jazyk zatím není podporován. Toto bych chtěl během této iterace změnit a zavést nástroj, který mi umožní mít všechny texty v aplikaci na jednom místě. Pokud to tak bude, nebude problém zavést podporu dalšího jazyka. V této iteraci bude přidán druhý podporovaný jazyk a tím je angličtina. Ta byla vybrána kvůli tomu, že pro zahraniční zákazníky je třeba odhad vytvářet v anglickém jazyce.

V rámci analýzy se také zaměřím na samotné rozložení jednotlivých částí obrazovky s detailem odhadu. Po dokončení první iterace má rozhraní aplikace jasně dané rozložení, kde se co nachází. Část s menu vlevo, dolní oblast ve spodní části a tabulka pro editaci položek kategorie v pravé horní části. Detailní rozložení lze vidět na Obrázku 4.7. Pokud si tedy uživatel otevře odhad na zařízení s menší obrazovkou, může se stát, že menu nebo dolní oblast bude zabírat zbytečně více místa, než je potřeba. Samotná tabulka s položkami kategorie tak bude menší na úkor ostatních částí obrazovky. Na základě této analýzy jsem se rozhodoval, jak celý layout uchopit a umožnit uživateli jeho konfiguraci. Jak to tedy bude? Do aplikace přidám novou funkcionalitu, která uživateli umožní pro jednotlivé části layoutu měnit jejich velikost a pozici. Uživatel si tak bude moci u každé části říci, zda ji schovat či naopak zobrazit a zároveň si tyto části mezi sebou přesouvat a měnit tak jejich uspořádání na obrazovce.

Poslední věcí, kterou se v rámci analýzy musím zabývat a která je na kritické cestě, je změna licence knihovny Handsontable. Na začátku této iterace

došlo ze strany autorů knihovny ke změně licencování. Doposud byla knihovna vydávána ve dvou verzích. Komunitní, která byla zdarma i pro komerční účely, a produkční, pro kterou bylo potřeba zakoupit licenci. Nově pro všechny další vydané verze je knihovna poskytována pod jednotnou licencí, která je pro open-source projekty zdarma. Pro komerční účely je ovšem třeba zakoupit licenci. V praxi to tedy pro můj projekt znamená, že se vydám jednou z následujících cest. Jako první z nich se nabízelo zafixovat verzi používané knihovny na poslední verzi, která byla vydávaná ještě v rámci komunity verze. Bohužel bych se tím tak připravil o jakékoliv opravy a nové funkcionality, které by vývojáři do novějších verzí přidávali, proto jsem tuto cestu zavrhl. Rozhodl jsem se proto ještě jednou zmapovat situaci ohledně tabulkových komponent. Pokud bych našel knihovnu, která by poskytovala stejnou funkcionalitu a její licence by byla výrazně levnější, migroval bych na tuto knihovnu. Pokud bych žádnou jinou nenašel, pravděpodobně by se zakoupila tato licence.

V rámci rychlého průzkumu knihoven a malého PoC jsem našel jednu knihovnu. Byla jí knihovna AgGrid. Ta nabízela podobné funkcionality jako Handsontable. Dokonce obsahovala i některé, které současná knihovna neměla. Její licence byla sice také zpoplatněná, vycházela ale cenově výrazně lépe. Licence byla potřeba jenom jedna pro celý projekt a nebyla jako v případě Handsontable vázaná na počet vývojářů na projektu. Navíc její zakoupení bylo potřeba až v případě, kdy je aplikace nasazená v produkčním prostředí. Rozhodl jsem se proto na tuto tabulku migrovat. Celá migrace bude popsána dále v druhé iteraci v sekci Implementace.

Jak jsem si nyní ověřil, rozdělení realizace aplikace na několik iterací se v tomto případě osvědčilo. Díky analýze v druhé iteraci se přišlo na to, že knihovna, kterou používám, již není zdarma. Protože jsem ale ještě relativně na začátku projektu a bude ještě několik dalších iterací, mohu v celku rychle reagovat na změnu, která nastala, a knihovnu vyměnit. Kdybych nejdříve celý projekt detailně analyzoval včetně knihovny pro tabulky, zřejmě bych přišel na problém s licencí až v momentě implementace. To by znamenalo ztracený čas na detailní analýze a návrhu něčeho, co nakonec nemohu ani použít. Proto si stojím za tím a v praxi se to i ukázalo, že iterativní přístup byl vhodné řešení.

Je možné, že v rámci analýzy nebyly zanalyzovány všechny funkcionality, které budou nakonec implementovány. Je to dáno tím, že cílovým uživatelem jsou zaměstnanci firmy, kteří se podílejí na odhadu realizace softwaru. Možnost pomoci při tvorbě odhadů mají i vývojáři na projektu. Je tedy možné, že během implementace se díky reálné zkušenosti s používáním nástroje zjistí, že je například potřeba naimplementovat věc, na kterou se během analýzy nepřišlo, ale která uživatelům výrazně pomůže při tvorbě odhadů. Reálné používání aplikace samotnými vývojáři beru jako velké plus, protože mají šanci ovlivnit chování aplikace a vylepšit ho na základě práce s aplikací z pohledu běžného uživatele.

Knihovna	Týdenní počet stažení
`react-intl` [20]	539 321
`react-i18next` [21]	384 005

Tabulka 5.1: Týdenní počet stažení knihoven pro internacionalizaci v React

5.3 Návrh

V této části se zaměřím na návrh několika nových funkcionalit a návrh obrazovek.

5.3.1 Podpora více jazyků

Jednou z věcí, kterou je potřeba vyřešit co nejdříve, je způsob, jakým bude realizována internacionalizace aplikace neboli podpora více jazyků. V současné chvíli se aplikace nachází ve stavu, kdy jsou veškeré texty, které může uživatel v aplikaci vidět, pevně vloženy v komponentách. Není tedy možné podporovat v současné chvíli více jazykových variant. Pokud bych se pro podporu více jazyků rozhodl až v pozdější fázi projektu, bylo by to s velkou pravděpodobností pozdě a kvůli větší náročnosti na implementaci i časově více náročné. Protože se v současnosti v aplikaci nenachází ještě mnoho textů, bude zavedení internacionalizace jak z časového hlediska, tak co do náročnosti změny, jednodušší provést nyní.

Je třeba prozkoumat možnosti internacionalizace aplikace React aplikace. Jako nejideálnější řešení se mi jeví mít v aplikaci složku, ve které budou soubory s jednotlivými překlady pro každý jazyk. Každý takto přeložený text bude mít klíč, na jehož základě půjde překlad v souboru nalézt. Následně se v komponentě pro jeho vložení bude muset zavolat funkce nebo knihovnou podporovaná komponenta, které se předá klíč a na základě uživatelem vybraného jazyka se vyrenderuje příslušný překlad.

Provedl jsem proto průzkum napříč knihovnami podporující internacionalizaci v React. Identifikoval jsem celkem dvě kandidátní knihovny. Jsou to `react-intl`[20] a `react-i18next`[21]. Obě pracují na principu, který jsem popsal výše. Po prohlédnutí repozitářů na GitHub a internetových článku ohledně daného tématu jsem se nakonec přiklonil ke knihovně `react-intl`. Ta mi přišla mezi React komunitou oproti druhé zmiňované četněji používaná. Při porovnání četnosti použití jsem vybral jako metriku týdenní počet stažení knihovny na webu `npmjs.com`. Výsledky porovnání lze vidět v Tabulce 5.1.

5.3.2 Úprava modelu odhadu

V současné době model odhadu vypadá stejně jako v předchozí iteraci. Abych ale mohl podporovat několik funkcionalit, které mají být v této iteraci přidány,

```
ChecklistItem := {description: string, tags: Set<Tag>}

EstimateItem := {
    ...
    checklist: List<ChecklistItem>
}

Estimate := {
    ...
    name: string,
    description: string
}
```

Zdrojový kód 5.2: Doménový model odhadu v 2. iteraci

bude potřeba doplnit model o některé atributy. Jelikož aplikace zatím neběží na testovacím prostředí a neexistují data, nebude potřeba migrace dat na novější model. V případě, že by ale data existovala, nejednalo by se o složitý problém. Důvod je ten, že jsem si vybral dokumentově orientovanou databázi MongoDB, která nemá pro kolekce dokumentů pevně daná schémata. Pokud tedy modifikuji model, stačí nad databází pustit jednoduchý skript, který současně uložené dokumenty zmigruje do požadované podoby. Skript může atributy přidávat, měnit či mazat. Navíc může být napsaný v nativním JavaScript kódu. Zde se ukazují výhody výběru typu databáze. Kdybych totiž na začátku projektu vybral relační databázi, vyskytly by se mi v této fázi zbytečně další problémy spojené s migrací schématu, kterým se mi v mém řešení podařilo vyhnout.

Teď již k samotnému modelu. Nově bude mít uživatel možnost k odhadu přidat kromě názvu i krátký popis. Doposud se název odhadu ukládal pod atributem `description`. Tomu tak již do budoucna nebude a pod tímto atributem se bude nově ukládat krátký popis projektu. Aby bylo možné ale uložit název odhadu, bude nově přidán atribut `name`. V něm bude název uložen v textové podobě.

Nově bude u každého itemu v odhadu možnost uložit si checklist s informacemi, co je třeba pro danou kategorii splnit. Proto je nutné rozšířit v modelu jednotlivé itemy o atribut `checklist`. Tento atribut představuje pole, které bude obsahovat jednotlivé checklist objekty. Přičemž checklist objekt se skládá z názvu (nebo jinak řečeno popisu) a množiny tagů, které ho vystihují.

Oba dva atributy, které byly do modelu přidány, je možné vidět i pomocí pseudoobjektů ve Zdrojovém kódu 5.2.

5.3.3 Jak dále tvořit obrazovky

V 1. iteraci jsem v rámci návrhu obrazovek vytvářel wireframy. V té fázi projektu ještě neexistovaly žádné implementované obrazovky. Bylo proto potřeba rozmístění jednotlivých částí rozmyslet a návrh provést. V této fázi již mám k dispozici existující implementace stránek. Mám tedy již přibližný koncept, jak mají jednotlivé obrazovky vypadat. V rámci přidávání nových funkcionalit nebo úprav současných bude docházet k modifikaci původně navržených obrazovek. Nedává mi ale smysl dále udržovat návrh pomocí přepracovaných wireframů. Čas, který bych pro to obětoval, raději využiji efektivněji. Proto budu ladit design obrazovek v rámci implementace jednotlivých funkcionalit. Jeví se mi to jako nejefektivnější řešení, protože například při změně umístění komponenty budu mít okamžitou zpětnou vazbu, jak obrazovka vypadá.

Pokud se jedná o drobnou UI změnu, jako je například padding, barva prvku, typ ohraničení aj., kterou si chci nejdříve vyzkoušet, než provedu implementaci, lze použít ve webovém prohlížeči nástroje pro vývojáře a příslušnou změnu na zkoušku provést. Není tedy třeba zasahovat do kódu aplikace a měnit ho. To je třeba až v případě samotné realizace změny v aplikaci.

5.4 Implementace

Nyní následuje část realizace aplikace v druhé iteraci. Část je rozdělena do několika částí podle řešených funkcionalit aplikace.

5.4.1 Tagování odhadů

Dříve, než se začnu zabývat, kam umístím komponentu pro zadávání tagů k odhadu, je potřeba nejprve mít nějakou vybranou. Co by všechno měla komponenta splňovat? Měla by uživateli umožňovat intuitivně přidávat tagy, mazat je a zároveň je v hezké formě zobrazovat. Ideální by bylo, kdyby komponenta umožňovala i našeptávání existujících tagů. Vlastní psaní komponenty pro přidávání tagů jsem rovnou zavrhl. Nedává mi smysl psát si vlastní komponentu, když si s tím již někdo jiný dal práci a komponentu vyladil. Rozhodl jsem se proto sáhnout po knihovně `react-tag-input`, kterou jsem již na jiném projektu použil a měl jsem s ní dobrou zkušenost. Komponentě z této knihovny stačí předat v props tagy, které se mají zobrazit jako již přidané, a návrhy tagů, které bude uživateli při zadávání našeptávat. Následně ještě v props předat listener metody, které budou reagovat na jednotlivé akce jako je přidání nebo odebrání tagu.

Knihovnu mám tedy vybranou, zbývá vyřešit, kde získat množinu tagů k našeptávání. Množina by měla být v ideálním případě taková, aby v ní bylo co nejvíce tagů. Proto jsem se rozhodl, že množinu navrhovaných tagů získám tak, že ji vytvořím z tagů všech odhadů, které jsou uloženy v databázi. Pokud

uživatel bude dělat odhad a bude ho chtít označit například tagem *Java*, je poměrně velká pravděpodobnost, že tento tag již v minulosti někdo použil.

Jak tyto tagy ale na frontend části získat? Jako nejideálnější řešení se mi jeví přidání nového endpointu do API. Frontend část nemusí znát všechny odhady, bude znát pouze množinu všech tagů. Endpoint bude dostupný přes metodu `GET` a umístěn na `/rest/tags`. Jeho implementace se nebude nacházet v existujícím controlleru, který vrací odhady. Pro tuto funkcionalitu jsem vyčlenil novou třídu `TagRestController`. Ta obsahuje pouze jednu metodu `getAll()`, která je namapována na příslušný endpoint. Získání tagů pro našeptávání probíhá na frontend části vždy po získání detailu odhadu. Je tedy nejprve načten samotný odhad a následně tagy pro našeptávání.

5.4.2 Editace detailu odhadu

Nově má aplikace podporovat kromě zadání názvu odhadu také krátký popis, otagování a autora. Je proto třeba zasáhnout výrazněji do uživatelského rozhraní. Nyní je v aplikaci pouze textové pole, do kterého uživatel zapisuje název odhadu. Toto textové pole je umístěno hned pod horní lištou aplikace. Protože by ale přidání dalších textových polí pod lištu výrazně omezilo velikost hlavní oblasti, ve které uživatel tráví při tvorbě odhadu většinu času, rozhodl jsem se zvolit pro umístění detailů odhadu jiné místo.

Doposud je v aplikaci v horní liště pouze logo aplikace a navigační menu, přes které se uživatel dostane na tvorbu nového odhadu a Dashboard. Zbylých 80 % lišty je nevyužito. Proto jsem se rozhodl přesunout editační pole s názvem odhadu do horní lišty a zvětšit tím tak místo pro tvorbu odhadu pod ní. Zároveň se díky tomu dostal do popředí samotný název odhadu. Při předchozím umístění nebyl tak jasně viditelný a uživateli chvíli trvalo, než se zorientoval.

Aby uživatel mohl editovat i další zmíněné atributy odhadu, je potřeba vymyslet pro uživatele nejjednodušší cestu, kterou atributy zedituje. Protože jsem přesunutím názvu odhadu získal více místa pro hlavní oblast pod lištou, nedává mi smysl ji znovu zmenšovat a umisťovat na původní místo názvu odhadu další textová pole pro vyplnění zmíněných atributů. Umístit atributy do lišty se mi také nelíbí, protože lišta je pro tyto účely příliš malá. Navíc by uživatele zahltila příliš mnoha údaji, které většinu času nepotřebuje znát. Po této úvaze jsem proto zvolil pro editaci údajů modální okno, o kterém budu dále mluvit jen jako o dialogu. Dialog se zobrazí přes UI a nebude oproti současné situaci zabírat místo. Bude v něm tedy jak samotný název odhadu, tak jeho krátký popis. Dále musí mít uživatel možnost zadat autora projektu. Ten bude zatím možné zadat jako libovolný text do textového pole. Poslední položkou budou tagy. Ty jsou realizováno pomocí komponenty ze zmíněné knihovny `react-tag-input`.

Dialog je již hotový, je potřeba ještě vyřešit, jak ho zobrazovat. Rozhodl jsem se ho otevírat kliknutím na tlačítko *Detail*. To bude umístěné v horní

```
// cs.json
{
    "app.name": "Estimate",
    ...
}
// messages.js
const messages = {
    app: {
            name: {id: 'app.name'}
            ...
    },
    ...
}
// použití s metodou
formateMessage{messages.app.name}
// použití s komponentou
<FormattedMessage  ...{messages.app.name}/
```

Zdrojový kód 5.3: Ukázka intl

liště hned vedle textového pole s názvem odhadu. Nezabere proto v UI tolik místa a zároveň bude uživateli jasné, co se pod tlačítkem skrývá.

5.4.3 Internacionalizace textu

Jak jsem zmínil již na začátku, internacionalizace textů se týká frontend části. Knihovnu pro sjednocení a internacionalizaci textu jsem již vybral. Je to knihovna `react-intl`. Začnu tedy nejprve tím, kde jsou všechny texty uloženy. Texty jsou ukládány v JSON souboru, kde jsou reprezentovány v objektu jako klíč-hodnota. Klíč je v tomto případě unikátní identifikátor textu a hodnota je text v příslušném překladu. Tyto JSON soubory se nachází ve složce `src/lang`. Protože je v plánu podpora dvou jazyků, jedná se o český `cs.json` a anglický `en.json`. Soubory jsou zatím prázdné, protože v textu jsou texty přímo v kódu. Dále si ve stejné složce zadefinuji soubor, ve kterém bude definován objekt `messages`. Ten bude později obsahovat jednotlivé identifikátory textů, které budou v lokalizačních souborech. Ukázku s názvem aplikace lze vidět ve Zdrojovém kódu 5.3.

Soubor, který se má použít, se vybere na základě lokalizace prohlížeče. Pokud má tedy uživatel prohlížeč v češtině, zvolí se `cs.json`. Pro všechny jiné jazyky se vybere anglický lokalizační soubor. Tímto je v projektu nastavené vše, aby lokalizace fungovala tak, jak má.

Jediné, co doposud nebylo provedeno, je odstranění hard coded textů z kódu. Jak tedy postupovat, aby aplikace byla ve stavu, kdy je celá loka-

lizována? Nejprve identifikuji ve všech komponentách hard coded texty a pro každý takovýto text postupuji podle následujících kroků:

1. Určím smysluplný identifikátor textu na základě toho, co obsahuje a přidám příslušné překlady do lokalizačních JSON souborů.
2. Přidám do objektu `messages` zvolený identifikátor.
3. Text v komponentě smažu. Místo něj použiji v kódu jedno z řešení:
   - V místě musí být čistý text a ne React element – Použiji metodu `formateMessage()`. Její použití lze vidět ve Zdrojovém kódu 5.3. Tato metoda vrací odpovídající překlad v textovém formátu. Používá se například u atributů, který vyžadují přímo string hodnotu.
   - Může být použit React element – Použiji knihovnou nabízenou komponentu `FormattedMessage`. Pro příklad s názvem aplikace vypadá použití obdobně, lze opět vidět ve Zdrojovém kódu 5.3. V tomto případě je překlad realizován pomocí komponenty, která text obalí do React elementu, který si při inicializace knihovny zvolím. V mém případě se jedná o React Frament, takže text není ve vyrenderovém HTML ničím obalen.

Tímto způsobem jsem převedl všechny texty do příslušných překladů a zároveň sjednotil texty na jedno místo. Při přidání nových textů budu postupovat obdobně jako v případě odstraňování hard coded textů z kódu.

5.4.4 Konfigurace částí obrazovky

Další důležitá funkcionalita, která bude v této části přidána, je konfigurace částí obrazovky v detailu odhadu. Na menších zařízeních se stávalo, že ostatní části mimo samotné tabulky pro zadávání odhadu zabíraly příliš mnoho místa a tabulka tak na jejich úkor byla menší. Tento problém jsem se ze začátku rozhodl řešit pomocí knihovny `react-split`. Tato knihovna poskytuje funkcionalitu, kdy si určím 2 části, mezi kterými chci měnit velikost, a obalím je touto komponentou. Komponenta mi pak umožňuje zvětšit jednu oblast na úkor druhé. Toto řešení se později ukázalo jako ne zcela vyhovující. Neumožňuje mi měnit pozici jednotlivých části v aplikaci a zároveň části nejde zcela schovat. Bylo proto třeba vybrat jinou knihovnu, která by poskytovala lepší funkcionalitu.

Po krátkém průzkumu jsem narazil na knihovnu `react-mosaic-component`, která splňovala veškeré požadavky. Jak sami autoři tvrdí, „jedná se o plně vybaveného dlaždicového správce oken, který uživateli poskytuje kompletní kontrolu nad jeho pracovním prostorem.“ [22, překlad vlastní] Implementaci podle původní knihovny jsem proto nahradil touto knihovnou. Knihovna mi umožňuje definovat výchozí rozložení jednotlivých dlaždic při

startu stránky. Dále nabízí možnost přesunutí dlaždice na jinou pozici a změnit její velikost. Pokud navíc dlaždici s obsahem momentálně nepotřebuji, mohu ji minimalizovat do lišty, která je současní UI. V případě, že minimalizovanou dlaždici budu chtít opět zobrazit, kliknu na jí příslušící ikonku v liště a tím ji opět zobrazím. Dlaždice a lištu dlaždic je možné vidět na Obrázku 5.1 pod hlavní lištou aplikace.

5.4.5 Výměna tabulkové komponenty

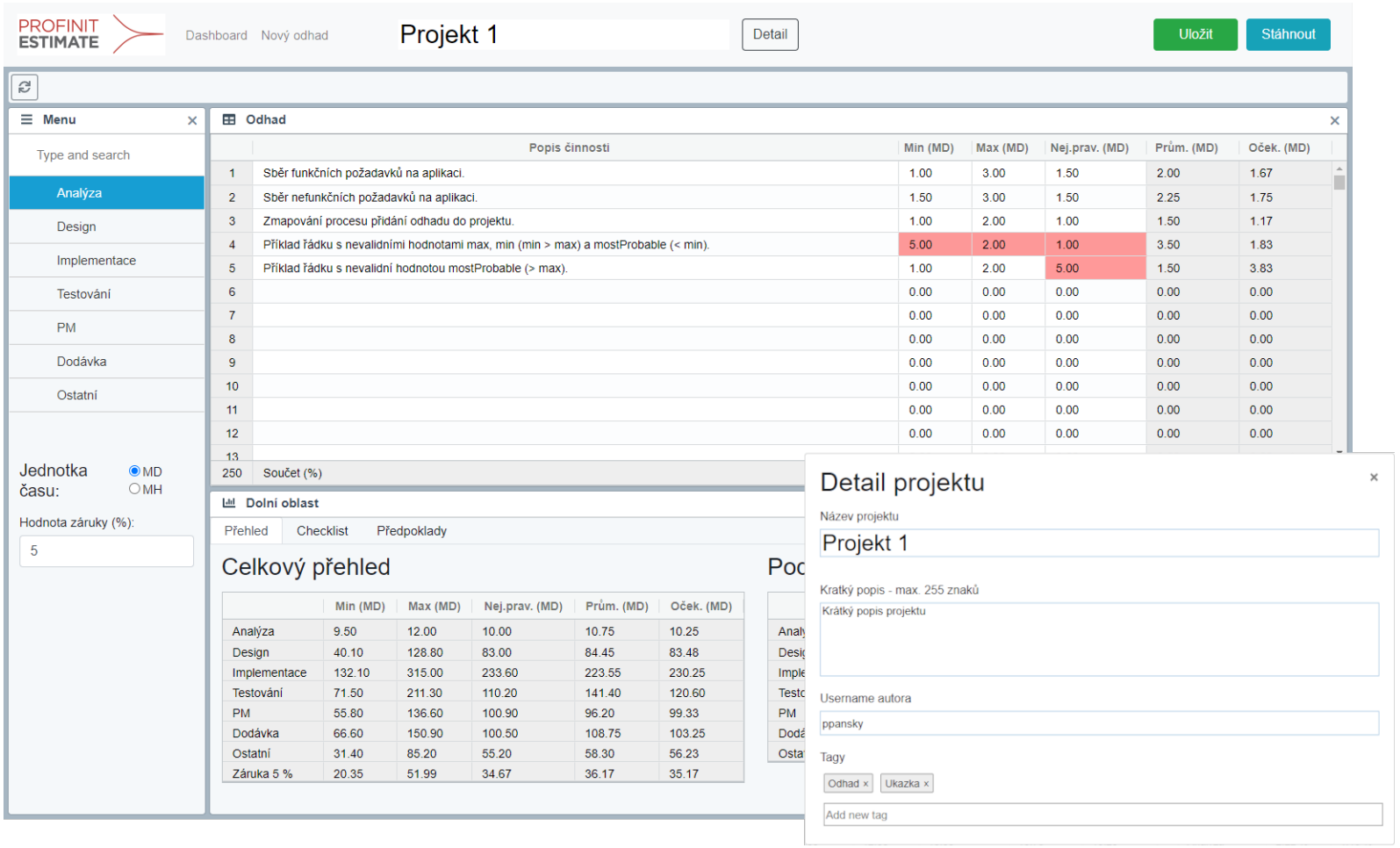
Jako vývoj každé funkcionality i migrace z knihovny Handsontable na AgGrid probíhá v samostatné větvi. Neblokuje tak vývoj dalších funkcionalit, které nesouvisí přímo s funkcionalitou v tabulce. Obě knihovny jsou si přístupem k předávání dat komponentě, reagování na jejich změny a stylováním, částečně podobné. Proto přepis z jedné komponenty na druhou není tak složitý. Zároveň ještě nejsou pomocí Handsontable tabulky implementovány složitější funkcionality, které jsou plánovány v další iteraci.

Nyní mám nad Handsontable tabulkovou komponentu napsaný vlastní wrapper, který mi umožňuje definovat si nad komponentou vlastní metody a formát dat, ve kterém jsou komponentě předávány. Až v těle wrapper komponenty probíhá integrace na funkcionality Handsontable tabulky. Díky wrapper komponentě mě při použití tabulky v aplikaci až tak nezajímá knihovní implementace Handsontable, ale rozhraní, které wrapper poskytuje. Migrace na knihovnu AgGrid proto nebude až tak složitá a na použití wrapper komponenty bude mít jen minimální dopad. Kompletně se změní pouze její vnitřní implementace, kde dojde k výměně tabulkové komponenty.

5.5 Testování

Z pohledu backend části v testování neproběhly žádné výrazné změny. Přibyly pouze testy pro nově přidané funkcionality, které jsou zmíněny v předešlých částech iterace. Jedná se například o testy pro nově přidaný `TagRestController` nebo o testování endpointu poskytujícího odhad ve formě Excelu.

Kde se ale oproti předchozí iteraci změny provedly, je frontend část. Ta v předešlé části neobsahovala žádné testy. Nově byl do projektu přidán testovací JavaScript framework `Jest`. Frontend část tak obsahuje několik testů, které kontrolují základní funkcionalitu. Nejedná se ale o UI testování, protože testovány jsou pouze vybrané utils třídy. V testech je nejvíce pokryta funkcionalita souboru `utils/EvaluateUtils.js`, ve kterém se nacházejí funkce pro vyhodnocení odhadu. Tato část je na rozdíl od ostatních věcí pokryta testy kvůli tomu, že se jedná o klíčovou funkcionalitu. Pokud by ve výpočtu vyhodnocení odhadu byla chyba, byla by napříč celou aplikací. Chyba by se projevila jak v samotné tabulce, ve které uživatel odhad zadává – zde jsou dopočítávány pro každou položku dvě buňky, tak i v celkovém přehledu v dolní



Obrázek 5.1: UI na konci 2. iterace

části aplikace. Navíc takto špatně vypočítaná data by byla po uložení persistována i v databázi backend částí a při exportu do Excel formátu by rovněž data nebyla správná. Jak je tedy vidět, dopad špatně vyhodnoceného odhadu není zrovna malý.

Co je ale z pohledu testování pro projekt momentálně asi nejdůležitější, je fakt, že se v této iteraci podařilo zprovoznit testovací prostředí. Do prostředí má přístup každý kolega, který zná adresu běžící aplikace. V rámci testování jsme proto poprosili některé z kolegů, zda by nové odhady nerealizovali místo v Excel šabloně přímo v aplikaci běžící na testovacím prostředí. Kolegové se toho s chutí ujmuli a za to jim patří velký dík. Zároveň je to pro projekt poměrně důležitý milník. Od této chvíle jsem schopný získávat od kolegů zpětnou vazbu, jak se jim v aplikaci pracuje, co se jim na ní líbí a co by naopak vylepšili. Jejich nápady tak mohu v průběhu zbylých dvou iterací zpracovávat a aplikaci na základě jejich zpětné vazby vylepšit ke spokojenosti uživatelů.

5.6 Zhodnocení

Během druhé iterace se podařilo dosáhnout hned několika dílčích úspěchů aplikace. V první řadě je aplikace nasazená v testovacím prostředí, kde se k ní mohou dostat reální uživatelé mimo řady vývojového týmu a posílat zpětnou vazbu. Povedlo se také vylepšit uživatelské rozhraní přidáním oken pomocí dlaždic, rozšířily se možnosti pro popis odhadu pomocí jeho detailů a aplikace je nyní lépe připravena pro případnou podporu dalších jazyků. Během analýzy nastal problém ohledně změny licenčních podmínek jedné z nejvíce používaných komponent. Díky zvolenému přístupu iterací se nicméně změnu licence podařilo zjistit včas a v krátkém čase na ni adekvátně reagovat. V neposlední řadě přibyly testy i na straně frontend části. V dalších iteracích se tak nabízí možnost otestování více funkcí frontend části.

Z důvodu zavedení testovacího prostředí se v příští iteraci očekávají podněty od kolegů na vylepšení aplikace. Zároveň je v plánu vývoj funkcionalit, které v Excel šabloně jsou. Jedná se o helper funkce a používání variant.

Kapitola 6

3. iterace

Před začátkem třetí iterace se aplikace nachází ve stavu, kdy je uživatel schopný vytvářet odhady a zadávat detailní informace o odhadu, jako je například tagování. Také si může dle svých preferencí uzpůsobit dlaždice na obrazovce s editací odhadu. Z pohledu projektových věcí je zprovozněné CI a v interní firemní síti běží testovací server, na kterém je vždy nasazená aktuální verze vyvíjené aplikace Estimate. V této kapitole se zaměřím na činnosti, které je nutné provést během třetí iterace.

6.1 Podpůrné činnosti projektu

V této části se zaměřím na činnosti, které nesouvisejí s realizací finální aplikace, ale na jejím výsledku nesou také podíl.

6.1.1 Spring profily aplikace

Spring v rámci vývoje aplikací nabízí používání profilů. „Spring profily poskytují způsob, jak oddělit části konfigurace vaší aplikace a zpřístupnit je pouze v některých prostředích.“[23, překlad vlastní] Je proto možné si nadefinovat pro každé prostředí vlastní profil. Pro každý profil lze použít odlišnou konfiguraci. Spring poskytuje nastavení konfigurace pro profil pomocí souboru `application.properties`, který je pojmenovaný ve formátu `application-nazevProfilu.properties`. Zároveň lze konfiguraci pro profil nastavit pomocí anotace `@Profile("nazevProfilu")`, kterou mohu přidat k libovolné `Bean` a `Configuration` třídě.

V aplikaci se nyní používají pouze dva profily. První z nich je tzv. výchozí. S tímto profilem se aplikace spustí vždy, když ji v rámci inicializace není předán žádný profil pomocí parametru. Tento profil se v současné chvíli používá pro lokální vývoj. Pokud je ale aplikace spuštěna na testovacím prostředí, použije se testovací profil `test`. V tomto profilu dochází k přepsání některých hodnot aplikačních properties, které Spring používá. Jedná se například

o port, na kterém běží server, adresu databáze nebo informaci, zda-li se mají v databázi smazat data a vygenerovat se do ní předepsaná ukázková data.

Zmíněná property pro resetování databáze `estimate.db.reset-data` se nastavuje pomocí boolean hodnoty. Následně se v komponentě `DataInitializer` při spuštění aplikace kontroluje, zda se má databáze resetovat. Toto řešení se mi nyní příliš nezamlouvá, navíc ve výchozím profilu je nastaveno resetování na hodnotu `true`. Mohlo by se tak stát, že v případě, kdy se aplikace nasadí na testovacím prostředí bez profilu, přijdu o všechna data resetováním databáze. Rozhodl jsem se proto zavést na projektu vývojový profil `dev`. Protože k resetování a generování dat v databázi dochází pouze v rámci vývoje, může být zmíněná property odstraněna a komponenta `DataInitializer` může být označena anotací `@Profile(Environments.DEVELOPMENT)`. K vytvoření a spuštění této Bean tedy dojde pouze v případě, že je aplikace spuštěna s vývojovým profilem. Nikdy jindy se resetování dat provádět nebude. Zároveň se vyhnu chybě, kdy by se mohlo v jiném profilu stát, že zapomenu property nastavit na false a například v budoucím produkčním prostředí dojde k resetování dat po spuštění aplikace.

6.1.2 Migrace databázového modelu

Zatím v rámci každé iterace došlo i k úpravě samotného modelu odhadu. Aplikace nyní ale běží na testovacím prostředí a obsahuje uživatelská data v podobě odhadů, které uživatelé v rámci testování vytvořili či vytváří. Z tohoto důvodu není od této iterace možné provést pouze úpravu modelu v kódu aplikace. Je nutné proto provést i úpravu dat v samotné MongoDB databázi, která běží na testovacím serveru.

Rozhodl jsem se úpravu dat v databázi řešit následujícím způsobem. Pokud bude v rámci úkolu nutné modifikovat strukturu modelu odhadu, bude nutné v rámci tohoto úkolu také vytvořit migrační skript. Tento skript bude mít za úkol převést odhady na testovacím prostředí do novější struktury odhadu, která v rámci úkolu vznikne. Může se jednat například o doplnění nového atributu, přejmenování a další možné akce nad modelem. Definice skriptu bude uložena u konkrétního úkolu v GitLab issue tracking systému. Po zamergování změn v rámci úkolu do větve master a přenasazení aplikace na testu bude muset vývojář tento migrační skript spustit. Do budoucna se plánuje tuto činnost automatizovat. Skript bude možné spustit z konzole nad MongoDB databází na testovacím serveru. Proto je nutné, aby byl napsaný v jazyce, který databáze dokáže spustit. Jsou proto možné dva způsoby, jak skript bude vypadat.

První způsob je použití pouze příkazů, které databáze nabízí. Protože se jedná o úpravu nad všemi odhady, bude ve většině případů použit nad kolekcí odhadů příkaz `updateMany(filter, update, options)` [24]. Tomu je možné kromě samotné konfigurace, jak se mají data v kolekci upravit, říci

pomocí filtru, nad jakými odhady se má spustit, a také mu předat konfiguraci ovlivňující provedení aktualizace dat.

Druhý způsob, jak napsat migrační skript, využívá možnosti kombinovat příkazy databáze s kódy v jazyce JavaScript. Lze tak pomocí nativního databázového příkazu `find`[24] vyhledat v kolekci potřebné odhady, nad kterými se má změna provést. Tento příkaz vrátí pole odhadů. Mohu proto zkombinovat volání příkazů `find` a nalezeným polem odhadů iterovat pomocí JavaScript funkce `forEach`. K iteraci odhadu využiji lambda funkci. V jejím těle upravím strukturu odhadu tak, jak potřebuji, a poté model odhadu uložím. Díky tomu, že má model identifikátor, dojde k přepsání starého odhadu a v databázi bude upravený odhad s novou strukturou. Až skript proiteruje celým polem odhadů, budou v databázi pouze zmigrovaná data.

6.2 Analýza

V rámci analýzy je třeba se zaměřit na dvě nejzásadnější funkcionality, které budou v rámci této iterace realizovány. Jedná se o práci s variantami a pomocnými „helper“ funkcemi pro tvorbu odhadu. Proč jsem se rozhodl právě pro tyto funkce? Jedná se o poměrně důležité funkcionality, které jsou při tvorbě odhadu napříč celou firmou hojně používány. Pokud chce uživatel použít současnou verzi Estimate z 2. iterace a potřebuje v odhadu použít varianty, může aplikaci využít pouze pro vytvoření základní kostry odhadu. Tou se myslí rozpad jednotlivých kategorií na menší celky a odhadnutí metrik času. Takto vytvořený odhad si uživatel musí stáhnout v Excel šabloně a jednotlivé položky kategorií si překopírovat do verze šablony, která varianty podporuje. Takováto práce je neefektivní a výsledný odhad není centralizovaný na jednom místě, jak je nyní v plánu. Většina menších funkcionalit, které jsou v Excel šabloně, jsou již implementovány. Z tohoto důvodu jsem se v této iteraci rozhodl rozvinout právě tuto oblast.

Nejdříve se zaměřím na funkcionalitu variant odhadu. Varianty slouží v Excel šabloně k odhadnutí projektu, který může mít například scénář, kdy chci odhadnout hlavní část spolu s dalšími částmi, o kterých nevím, zda budou finálně realizovány. Varianty tedy umožňují při tvorbě odhadu do něj zahrnout více možných variant "budoucnosti". V dosud používané Excel šabloně fungují varianty následujícím způsobem. Uživatel má na výběr celkem šest možných variant, při čemž všechny využít nemusí. Po vybrání, které varianty bude chtít použít, se přesune k samotnému odhadu a jeho kategoriím. V nich pracuje uživatel úplně stejně jako doposud. V Excel šabloně je ovšem navíc jeden sloupec, do kterého může zadat číslo varianty. Každý záznam tak má kromě popisu a odhadu pracnosti také příslušnost k určité variantě. Pokud záznam žádnou variantu nemá, jedná se o položku, která patří do všech variant. Záznam může zároveň patřit maximálně k jedné variantě. Kombinace více variant v jedné položce není možná. V takto vytvořeném odhadu v Excel

šabloně může uživatel následně pracovat s odhadovanou pracností celého projektu právě díky variantám. Může tak například získat pracnost bez variant (záznamy bez udaných variant) nebo například pracnost projektu s variantou 1. Toho lze dosáhnout díky možnosti zahrnout či nezahrnout jednotlivé varianty do celkového přehledu pracnosti odhadu. Nejlépe jde tato situace demonstrovat na příkladu. Dostanu za úkol odhadnout náročnost realizace aplikace. Je možné, že zákazník bude chtít realizovat i moduly A a B. Proto si oba moduly namapuji na odpovídající varianty 1 a 2. Realizaci základní části aplikace odhadnu bez variant, jejich použití by zde bylo zbytečné. Ovšem u realizaci obou modulů použiji jejich varianty. Díky variantám jsem po dokončení odhadu schopný říci, kolik zabere realizace základní části bez modulů, s jedním z nich nebo s modulem A i B.

Po analýze, jak varianty fungují v Excel šabloně, je na čase se rozhodnout, jak s nimi pracovat v aplikaci. Zvolil jsem obdobné chování jako je v Excel šabloně. Oproti němu budou ale v aplikaci dvě změny. První z nich je podpora pojmenování variant. Varianta tedy bude mít textový popis. Druhou, a z pohledu funkčnosti větší změnou je, že bude možné kombinovat u jedné položky více variant. Bude tak možné vytvořit záznam, se kterým se bude počítat ve více variantách. Tuto funkcionalitu současná Excel šablona neposkytuje.

Když mám vyřešenu analýzu ohledně variant v odhadu, je čas se přesunout k druhé zmiňované funkcionalitě. Tou jsou takzvané helper funkce. V Excel šabloně se jedná o extra řádky pod tabulkou se záznamy kategorie, kde se vypočítávají data podle předem daného vzorce. Každá takováto helper funkce má dané přesné umístění. Například zda je určena pro implementaci či testování. Funkce pracuje ve většině případů na základě dat z celkového přehledu a doporučuje uživateli, jakou pracnost by daná kategorie měla mít. Pokud tedy vím, že na většině projektů mi projektové řízení zabere 10 %, mohu si nadefinovat helper funkci, která mi zobrazí, kolik je 10 % z celkové pracnosti. Podíly projektového řízení a dalších aktivit použitých v helper funkcích vycházejí z obecných metrik, které popsal Steve McConnell [1, s. 233-237]. Na základě "nápovědy"této funkce budu poté schopný korigovat hodnotu projektového řízení na odhadu. Helper funkce jsou tedy sada funkcí, které pomáhají zpřesnit samotný odhad jednotlivých kategorií a tím i celý odhad.

Obdobně budou helper funkce pracovat i v aplikaci Estimate. Rozhodoval jsem se, jak helper funkce v aplikaci pojmout. Ty nakonec budou nadefinovány již v jednotlivých šablonách na základě nejvíce používaných helper funkcí z Excel šablon ve firmě. Uživatel, který s aplikací bude pracovat, je tedy nebude moci v základu měnit. Místo toho si vybere příslušnou šablonu, ve které už budou funkce nadefinovány.

6.3 Návrh

V této podkapitole se budu zabývat návrhovými věcmi, které je nutné vyřešit, než se přejde k samotné implementační části.

6.3.1 Úprava modelu odhadu

V této iteraci mají být do modelu přidány varianty a helper funkce. Je proto třeba upravit současný model, aby podporoval i nově přidané funkcionality. Zároveň je třeba myslet také na již existující data. Na testovacím prostředí je vytvořeno několik odhadů, bude proto nutné mimo úpravy modelu na tomto prostředí realizovat i migraci dat, aby jejich struktura odpovídala nově upravenému modelu.

Začnu nejprve helper funkcemi. Jejich dopad na model není příliš velký. Helper funkce budou stejně jako ostatní data součástí šablony – budou proto uloženy v modelu. Protože se jedná o konfiguraci helper funkcí, rozhodl jsem se je v modelu umístit do konfigurace, která je pod atributem `configuration`. V ní jsou helper funkce uloženy pod atributem `helperFunctions`. Zde se bude nacházet množina helper funkcí. Samotná helper funkce se bude skládat ze tří atributů. Prvním z nich je `itemDescription`, pod ním se ukrývá název kategorie, ve které má být helper funkce používána. Podle tohoto názvu se tedy pozná, zda jsem ve správné kategorii, a na jeho základě se helper funkce aktivuje. Dalším atributem je `helpText`. Jedná se o text s nápovědou, co daná helper funkce představuje. Tento text se později využije k zobrazení v UI. Posledním atributem a z pohledu funkcí velmi důležitým je `definition`. Jedná se o definice helper funkce. Ta uvádí, jak pro danou helper funkci vypočítat její hodnoty. V definici se zpravidla objevují výpočty typu: "vezmi pracnost několika kategorií a vynásob či vyděl ji určitými hodnotami". Z počátku jsem proto přemýšlel nad vytvořením jakéhosi pseudojazyka, ve kterém by hodnota funkce byla nadefinována. Nakonec jsem se ale rozhodl pro definici přímo v jazyce JavaScript. Tento způsob jsem zvolil proto, že JavaScript umožňuje vyhodnotit funkci, která je zadefinovaná uvnitř textové hodnoty. Navíc oproti návrhu vlastního pseudojazyka je toto řešení časově méně náročné a pro budoucí tvůrce dalších funkcí lépe uchopitelné. Tímto je tedy úprava modelu za helper funkce hotová.

Poslední úpravou, kterou je nutné provést, je podpora variant. Nejdříve je třeba přidat do modelu množinu variant, ze kterých bude moci uživatel vybírat. Pro jejich uložení jsem se rozhodl vytvořit nový atribut `variants`, který se nachází přímo v samotném odhadu, tedy na úrovní například atributu `name`. Tato množina se skládá z jednotlivých variant, které obsahují celkem tři atributy. Prvním z nich je atribut `id`. Díky němu lze variantu jednoznačně identifikovat mezi ostatními, protože každá varianta má unikátní identifikátor. Další atribut `description` umožňuje jednotlivé varianty pojmenovat textovou reprezentací. Aby bylo možné varianty nezávisle na sobě zapínat a vypínat,

```
Variant := {id: number, description: string, active: boolean}

HelperFunction := {
    itemDescription: string,
    helpText: string,
    definition: string
}

EstimateItem := {
    ...
    variants: List<number>
}

Estimate := {
    ...
    variants: List<Variant>,
    configuration: {
        ...
        helperFunctions: Set<HelperFunction>
    }
}
```

Zdrojový kód 6.1: Doménový model odhadu v 3. iteraci

obsahuje varianta také atribut `active`. V něm je uložena hodnota, zda je varianta aktivní či nikoliv.

Množina variant je popsána. Aby bylo možné přidávat varianty do jednotlivých položek, je potřeba ještě upravit model položek (item) a předpokladů, kterým jde jednotlivé varianty přiřazovat. Začnu nejprve předpoklady. U těch lze stejně jako u položek odhadu nadefinovat, kterých předpokladů se týkají které varianty. Proto je model varianty rozšířen o atribut `variants`. Pod ním se skrývá množina, která může obsahovat jednotlivé identifikátory příslušných variant. Podobně je tomu i u položek kategorií odhadu. Ty byly stejně jako předpoklady rozšířeny o atribut `variants`. Takto upravený model odhadu již plně podporuje práci s variantami a helper funkcemi.

Pro lepší představu, jak byl model upraven, se lze podívat ve Zdrojovém kódu 6.1, kde je model nadefinován pomocí pseudoobjektů.

6.4 Implementace

V této podkapitole se zaměřím na samotnou implementační část aplikace Estimate.

```
function getHelpData(estimate, estimateWithWarranty, coefficient)
{
    // výpočet a vrácení celkové pracnosti
    const total = estimate.getValue(coefficient).value;
    const warranty = estimate.configuration.warranty;
    return (1 + warranty/100) * total;
}
```

Zdrojový kód 6.2: Ukázka helper funkce

6.4.1 Helper funkce

Podívám se nejdříve na helper funkce. Jak je vysvětleno v podkapitole Analýza 6.2, helper funkce se zobrazují vždy pro danou kategorii v odhadu. Je proto důležité, aby se v aplikaci funkce zobrazovala vždy ve správné kategorii. Jak bylo zmíněno v předchozích odstavcích, ke správné identifikaci kategorie slouží atribut `itemDescription`. V něm má helper funkce uloženou svou příslušnost ke kategorii na základě jejího názvu. Například funkce pro analýzu zde proto bude mít hodnotu „Analýza“. Dále má funkce její popis. Poslední a nejdůležitější částí je definice. V části 6.3.1 uvádím, že jsem se rozhodl pro definici funkce pomocí JavaScript jazyka. Jedná se tedy o JavaScript kód, který je uložený v textové podobě. Kód definice funkce musí splňovat několik podmínek. Za prvé se musí jednat o validní JavaScript funkci, která neobsahuje syntaktickou chybu. Dále má tato funkce jasně dané parametry a jejich pořadí. Definice helper funkce je sice volná, ale musí dodržovat tato pravidla, aby volání funkcí šlo generalizovat a bylo možné tímto způsobem zavolat jakoukoliv helper funkci bez znalosti její implementace. Funkce má následující parametry:

- `estimate` – vyhodnocený odhad
- `estimateWithWarranty` – vyhodnocený odhad s připočtenou zárukou do pracnosti
- `coefficient` – koeficient konkrétní hodnoty pracnosti (minimum, maximum, nejvíce pravděpodobné)

Ukázku helper funkce lze vidět ve Zdrojovém kódu 6.2.

Jak definovat helper funkci je již tedy jasné. Zbývá vyřešit, jak ji vyhodnotit na straně frontend části nad reálným odhadem. Možností, jak v JavaScript jazyce naparsovat funkci, která je definovaná ve string hodnotě, není mnoho. Pro mé účely jsem našel celkem dvě možnosti. První z nich je objekt `Function`. Tomu lze předat názvy parametrů a samotné tělo metody, následně pak lze

takto vytvořený objekt zavolat s příslušnými parametry. Druhou možností je použití nástroje `eval`[25].

Já jsem se rozhodl použít nástroj `eval`. Na rozdíl od objektu `Function` nemusí být v těle funkce předem jasně pojmenované parametry, musí být pouze dodrženo jejich pořadí. Jedná se o funkci zabudovanou přímo v JavaScript API, která umí vyhodnotit JavaScript kód v textové podobě. V dokumentaci autoři varují před jejím použitím a doporučují ji používat co nejméně kvůli bezpečnosti. V mém případě helper funkce definuje vývojový tým. Jedná se o interní aplikaci, nemusím se proto obávat, že by v textové podobě definice funkcí byl škodlivý kód, který by se pomocí `eval` spustil a provedl na uživatele útok.

Pro vyhodnocení konkrétní helper funkce bude tedy stačit zavolat zabudovanou funkci `eval`, které předám definici funkce. Ta mi následně vrátí korektně zadefinovanou JavaScript funkci, kterou mohu volat s parametry v definovaném pořadí.

Po tom, co jsem ozřejmil, jak se helper funkce načítají a volají, je čas přejít k jejich umístění v uživatelském rozhraní. V Excel šabloně byly helper funkce umístěné pod tabulkou s položkami kategorie pod řádkem se sumou pracností. V Estimate jsem se proto rozhodl zvolit stejné umístění. Helper funkce tak budou po celou dobu tvorby odhadu uživateli na očích. K zobrazení řádků s funkcemi pod tabulkou, aby byly celou dobu uživateli k dispozici, se využívá v komponentě `AgGrid` vlastnost `pinnedRows`. Jedná se o fixní řádky, které jsou připnuty k dolní části tabulky a vizuálně odlišeny od běžných řádků. Jejím použitím je již řešeno například zobrazení sumy pracností. Bude tedy nutné tyto řádky pouze rozšířit o helper funkce na základě aktuálně editované kategorie. Helper funkce se budou zobrazovat tak, že ve sloupci popis se zobrazí název helper funkce a ve sloupcích s hodnotami vypočtené pomocné hodnoty.

6.4.2 Varianty

Kromě úprav na backend části je třeba k podpoře variant provést několik úprav i v UI.

V první řadě musí mít uživatel možnost si na obrazovce zobrazit seznam variant. Tento seznam zároveň musí uživateli nabídnout editaci názvů jednotlivých variant. Je tedy třeba nejprve vybrat jeho umístění. K tomuto seznamu uživatel nebude muset přistupovat příliš často. Předpokládá se, že na začátku projektu si uživatel rozmyslí, kolik variant bude potřebovat. Podle toho si následně pojmenuje v seznamu varianty. K seznamu poté bude uživatel pravděpodobně přistupovat jen velmi málo, například jen v případě editace varianty. Nedává proto smysl umisťovat tento seznam na místo, kde by byl uživateli celou dobu na očích. Zbytečně by zabíral potřebné místo. Z tohoto důvodu jsem se rozhodl nevytvářet pro tento seznam další dlaždicové Mosaic okno. Místo toho ho umístím do dolní části obrazovky, kde bude dostupný pod záložkou *Varianty*. Bude tedy možné si přepnout například z přehledové tabulky klik-

nutím na záložku *Varianty*, upravit varianty a poté opět přepnout na jinou záložkou. Po zvolení umístění je třeba vyřešit, jak bude seznam editovatelný. Pro tento případ jsem se rozhodl využít komponentu tabulky z knihovny AgGrid. Ta je použita i pro editaci samotných položek odhadu a předpokladů. Tabulka tedy obsahuje sloupec s identifikátory variant, který není možné editovat, a popis variant, ke kterému si uživatel může přidat jakýkoliv textový řetězec.

Další nezbytnou úpravou, kterou je nutné pro podporu variant realizovat, je umístění a zvolení přepínání mezi aktivní a neaktivní variantou. Tuto funkcionalitu bude uživatel na rozdíl od seznamu variant využívat výrazně více. Nelze ji tak umístit do záložky se seznamem. Na tomto místě by byla viditelná pouze v případě, kdy by si uživatel přepnul na tuto záložku. Zároveň by nemohl přepínat varianty a dívat se na dopad v přehledech, které jsou rovněž v této části pod jinou záložkou. Rozhodl jsem se proto varianty umístit do okna s konfigurací, které se nachází vlevo pod menu s kategoriemi odhadu. Je to celkem rozumný krok, protože tato část je uživateli neustále na očích. Vypnutí nebo zapnutí varianty navíc souvisí s konfigurací odhadu. Proto mi dává smysl tuto funkcionalitu umístit vedle konfigurace jednotky času a hodnoty záruky. Do tohoto okna je tedy přidána nově implementovaná komponenta, která umožňuje zapnout či vypnout variantu. Komponenty jsou ve dvou sloupcích po třech. Pro reprezentaci jednotlivých variant jsem si vybral komponentu `Switch`. Ta umožňuje uživateli přepínat mezi dvěma stavy. V labelu této komponenty je zobrazen `Badge`, ve kterém se nachází identifikátor příslušné varianty. Navíc pro každou variantu má tento `Badge` rozdílnou barvu pozadí, aby se varianty od sebe lépe rozeznaly.

Aplikace umí varianty editovat a vypínat či zapínat. Je tedy nutné umožnit uživateli zadat varianty k jednotlivým položkám kategorie a předpokladům. Do obou tabulek je proto nutné přidat nový sloupec. Aby byla příslušnost jednotlivých řádků k variantám jasnější, rozhodl jsem se tento sloupec přidat na začátek před všechny ostatní sloupce. Navíc jsem pro tento sloupec použil vlastní renderer. Jedná se o komponentu, která identifikátory variant zobrazí uvnitř komponenty `Badge`, jako je tomu u jejich konfigurace. Pro editace či smazání variant v jednotlivých buňkách sloupce stačí začít editaci buňky a psát identifikátory variant do buňky v textové podobě. Po ukončení editace se text namapuje na identifikátory variant a záznamu se přidají příslušné varianty dle identifikátorů.

Varianty je možné zadávat, je s nimi ale potřeba pracovat i při výpočtech. V celkovém přehledu musí uživatel vidět pouze pracnosti položek, které obsahují zapnuté varianty. Je proto nutné upravit vyhodnocovací funkci odhadu ve třídě `EvaluateUtils`. Nově je tak funkci `evaluate` kromě odhadu a vyhodnocovacích funkcí předávána také množina aktivních variant. Pokud ji nejsou v parametrech předány aktivní varianty, je použit pro varianty výchozí parametr `null`. Vyhodnocení po zapracování variant probíhá následovně:

- **Není použit parametr pro aktivní varianty** – Výpočet probíhá stejně jako doposud a varianty jsou ignorovány. Tento případ slouží pro vyhodnocení celého odhadu např. po změně dat a přepočítání všech hodnot.
- **Je použit parametr pro aktivní varianty** – je předán jejich list
  - **List je prázdný** – Vyhodnotí a vrátí se v odhadu pouze položky, které nemají přiřazenou žádnou variantu.
  - **List obsahuje alespoň jednu variantu** – Vyhodnotí a vrátí se v odhadu položky, které nemají přiřazenou žádnou variantu nebo ty, které mají přiřazenu alespoň jednu variantu z listu aktivních variant.

6.4.3 Zavedení Material-UI

V současné době používám pro tvorbu UI knihovnu `react-bootstrap`. Poslední dobou se při tvorbě UI občas vyskytují problémy, pro které je nutná detailní znalost CSS stylů. Posledním z nich bylo například špatné zarovnání labelu u komponenty `Switch`, která slouží pro přepínání aktivních variant. Aby bylo možné jejich vzájemné vycentrování, musel jsem provést netriviální opravu pomocí CSS stylů se zápornou hodnotou margin. Další problém, který se také řešil, byla špatná výška tabu v dolní oblasti. Největším problémem jsou ale nutné nestandardní změny stylů některých knihoven, aby jejich vzhled byl podle Bootstrap stylování. Knihovna `react-bootstrap` bohužel obsahuje jen základní komponenty pro tvorbu UI. Protože jsem v druhé iteraci potřeboval pro přidání funkcionalit komponenty, které Bootstrap neobsahuje, musel jsem přidat závislosti na dalších knihovnách. Ty mi sice poskytly potřebné funkcionality, díky tomuto kroku ale počet knihoven narostl. Jedná se například o knihovnu pro zadávání tagů nebo knihovnu pro tvorbu menu (kategorie odhadu). Při přidání další nebo aktualizaci současných knihoven je vždy třeba ověřit jejich design. V případě, že neodpovídá Bootstrap pravidlům, je třeba všechny její komponenty přestylovat do Bootstrap designu. Toto je do budoucna problém a občas vzhledem k omezeným možnostem konfigurace knihovny časově náročné. Někdy je potřeba „ohackovat“ stylování nepěkným způsobem a při další aktualizaci knihovny se může stát, že stylování je opět potřeba upravit, protože se například změnily názvy CSS tříd. Jak jsem již zmínil, Bootstrap knihovna obsahuje jen základní UI komponenty. Pro podporu další funkcionality je potřeba přidat další knihovny a přestylovat je. Navíc práce s `react-bootstrap` občas není tak přímočará, jak by se mohlo zdát. I když používám Bootstrap design systém, je velice často potřeba ke komponentám z knihovny dopisovat vlastní CSS stylování a celý proces vývoje je díky tomu pomalejší.

Rozhodl jsem se proto tento problém řešit a do budoucna se těmto krokům vyhnout. Shodou okolností jsem nedávno ve firmě pracoval na projektu, kde

jsme začali používat pro tvorbu UI jinou knihovnu. Ta se nám na projektu velice osvědčila. Protože jsem ve třetí iteraci a do konce projektu bude ještě jedna, mám čas a možnost vyměnit tuto knihovnu. Rozhodl jsem se ji proto vyzkoušet a nahradit současnou Bootstrap knihovnu.

Jaké řešení jsem zvolil? Jedná se o knihovnu pro tvorbu uživatelského rozhraní `material-ui` [26]. Nejlépe ji vystihují sami tvůrci na webu. „React komponenty pro rychlejší a snadnější vývoj webových aplikací. Vytvořte si vlastní designový systém nebo začněte s Material Designem.“[26, překlad vlastní]. Knihovna je tedy podobná Bootstrap knihovně, je postavena na komponentách. Všechny komponenty ctí pravidla a doporučení designového systému Material Design. Zároveň knihovna oproti Bootstrap obsahuje větší množství komponent, které pokrývají různé funkcionality. S velkou pravděpodobností tak při přidání nové funkcionality bude knihovna obsahovat komponentu, kterou pro danou funkcionalitu budu potřebovat. Například komponenta pro přidávání tagů je tu již v základu, není tedy nutné přidávat další knihovnu. Oproti Bootstrap knihovně nemusí vývojář psát takové množství CSS stylování pro komponenty u knihovny. Navíc práce s komponentami mi přijde výraznější efektivnější než při použití knihovny Bootstrap.

Výměna UI knihovny je velká změna, která zasáhne většinu komponent. Rozdělit migraci na novou knihovnu do několik úkolů mi nedává smysl. Aplikace by byla mezi dokončením prvního a posledního úkolu ve stavu, kdy se v aplikaci vyskytují komponenty z obou knihoven a design není vyladěný. Proto je třeba provést změnu v rámci jednoho úkolu, kde se odladí výměna komponent a styl aplikace. Po jeho dokončení tak nevznikne velké množství merge konfliktů a změny budou moci být zamergovány. Začal jsem tedy pracovat na migraci z knihovny `react-bootstrap` do `material-ui`. Postup, jak migrovat, byl vcelku jasný. Postupně jsem procházel každou komponentu a nahrazoval ji odpovídající komponentou v nové knihovně. Ve většině případů byly komponenty nahrazeny v poměru 1:1. Tím myslím, že například komponenta pro zadávání textu v Bootstrap knihovně byla nahrazena adekvátní komponentou v Material-UI. Zároveň se během migrace ukázala síla a jednoduchost nové knihovny. Kde jsem měl pro Bootstrap komponenty složité stylování s velkým množstvím CSS kódu, tam došlo k náhradě komponentou Material-UI, která měla jen pár řádek stylování a navíc pocitově lépe vypadala.

Ve finále tedy migrace proběhla bez větších problémů. Došlo k redukci používaných UI knihoven, zároveň i samotný kód doznal změn. V prvé řadě zmizelo velké množství souborů s definicemi CSS tříd a komponenty se výrazně zjednodušily. Tvorba UI se zlepšila. Material-UI nabízí propracovanější komponenty, které se nesou ve stylu Material Design. Ten se pro aplikaci na rozdíl od Bootstrap více hodí a není třeba do něj tolik zasahovat. Největším přínosem migrace je ale změna uživatelského prostředí k lepšímu a jeho rychlejší tvorba při vývoji. Aplikace nyní mnohem lépe vypadá a na uživatele působí jednotným dojmem. Díky úpravě tématu v Material-UI je aplikace naladěna do firemních barev. Aplikace se tak vzhledově posunula o dimenzi výše. Pro

představu, jak UI po migraci vypadá, znázorňuje Obrázek 6.1.

6.4.4 Migrace na TypeScript

Poslední dobou mě na projektu taktéž trápilo používání typové knihovny Flow. Knihovna značně zpomalovala vývojové prostředí a některá pokročilejší typování v ní nebylo možné provést právě kvůli tomu, že se jedná jen o knihovnu a nikoliv součást JavaScript jazyka. Zároveň typová kontrola byla zavedena jen nad modely a utils třídami. Typová kontrola nad samotnými komponentami zavedena kvůli problémům s typováním nebyla.

Rozhodoval jsem se proto, jak tento problém řešit. Na jiných projektech v Profinitu, na kterých jsem měl možnost pracovat, jsme již tou dobou používali React spolu s jazykem TypeScript. Zvažoval jsem, zda zůstat u současného typování pomocí Flow nebo sáhnout po ozkoušeném TypeScriptu. Když jsem si vedle sebe dal výhody a nevýhody obou zmíněných možností, vyšel mi jako vítěz TypeScript. Zvolil jsem tedy změnu a rozhodl se JavaScript nahradit jazykem TypeScript, který poskytuje vývojářům typování. Ten má oproti Flow několik výhod:

- Třídní a modifikátory přístupu – U třídou definovaných metod lze řídit přístup pomocí modifikátorů `public`, `protected` a `private`
- Větší podpora našeptávání ve vývojovém prostředí
- Podpora typování pro Redux (store i napojení komponent na stav Redux store)

Když jsem se nyní zpětně díval, proč jsem TypeScript nepoužil již na začátku projektu, ukázalo se, že v té době byl TypeScript ve spojení s React teprve ve vývoji. Nebylo tedy ještě možné použít tuto kombinaci a místo toho jsem se rozhodl pro JavaScript a knihovnu Flow.

Nyní je tedy na čase začít s migrací aplikace do TypeScript. Jak ale migraci co nejlépe realizovat? Nabízejí se celkem dvě možnosti. První možností je celkový přepis. To by znamenalo zastavení současného vývoje, rozdělení přepisu komponent do několika úkolů a jejich postupný přepis. Zde je riziko možných konfliktů při mergování do master větve. Další nevýhodou je, že se budu muset zaměřit na samotný přepis. To by znamenalo zastavení veškerého vývoje nových funkcionalit a ladění uživatelského rozhraní a projekt by nabral zpozdění. Z těchto důvodů jsem se tuto možnost rozhodl zamítnout. Místo toho jsem zvolil migraci do TypeScriptu postupně. V praxi to znamená, že v rámci migračního úkolu bude zavedena pouze podpora pro TypeScript včetně nastavení potřebné konfigurace. Následně se bude pokračovat ve vývoji aplikace jako doposud. Pokud v rámci daného úkolu bude nutné provést změny na komponentě, která je napsaná v JavaScriptu, zmigruje se zároveň tato komponenta do TypeScript. Pokud v rámci úkolu má být přidána nová

komponenta, bude napsána již v TypeScript. Tímto postupem by se měla aplikace během řešení několika desítek úkolů dostat do situace, že je již skoro celá napsána v TypeScriptu. V ten moment bude nutné dokončit migraci v rámci nového úkolu, použití JavaScript jazyka zakázat a odstranit Flow typování.

Co bylo v rámci začátku migrace nutné provést:

1. Přidat do projektu závislosti na knihovně `typescript`.

2. Přidat soubor `tsconfig.json`, ve kterém je konfigurace TypeScript projektu. Povolit v něm používání jazyka JavaScript nastavením atributu `allowJs` na `true`.

3. Přidat do `eslintrc.json` pravidla pro syntaxi TypeScript.

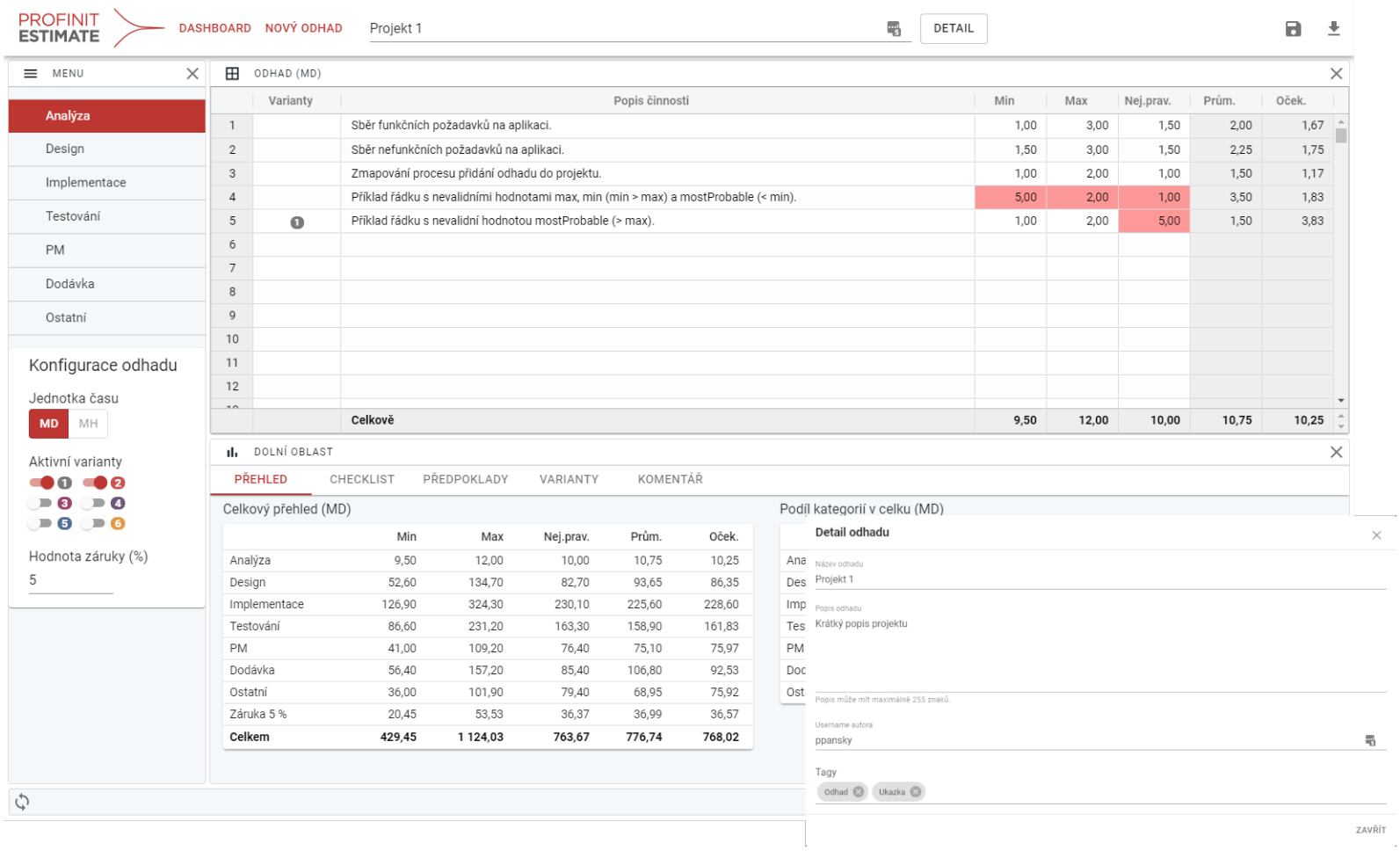
Po těchto změnách je aplikace připravena na postupný přepis do TypeScript.

6.4.5 Další UX změny

Pokud nepočítám migraci na Material-UI, došlo během implementace k některým dalším vylepšením v UX, které stojí za zmínku. Prvním z nich je úprava v horní liště aplikace. Dříve zde byla tlačítka pro uložení a stažení s textem uvnitř. Text je zde zcela zbytečný a zabírá zbytečně příliš mnoho plochy. Místo textu jsou nyní použity odpovídající ikonky pro uložení a stažení odhadu. Pokud by uživatel přeci jen nevěděl, jaké tlačítko použít, při najetí myší na tlačítko se zobrazí text, jakou akci tlačítko provede.

Další výraznější změnou je přesunutí status bar komponenty pro práci s dlaždicovými okny z horní části úplně dospod okna. Z pohledu uživatele se mi tato změna zdá praktičtější. Nyní jsou jak lišta aplikace, tak status bar jasně odděleny. Přesun vyvolalo částečně i přidání funkcionality, kdy se při označení buněk v liště zobrazí průměr a součet číselných hodnot v buňkách. V dolní části si této informace uživatel spíše všimne. Navíc je tato funkcionalita velice podobná s Excel chováním.

Poslední výraznější změnou, kterou zmíním, je odstranění vybrané časové jednotky ze sloupců tabulky pro zadávání položek kategorie. Tento nápad vycházel z Excel šablony, kde v každém sloupci byl název aktuálně používané časové jednotky. V Excel šabloně se jednotka v hlavičce tabulky zobrazovala pravděpodobně kvůli tomu, že nebyla jiná možnost, kde jednotku zobrazit. Kvůli těmto popiskům v hlavičce nebylo možné snížit šířku sloupců s hodnotami pracností. Ty tak byly v některých místech zbytečně široké. Proto byla jednotka přidána přímo do názvu dlaždicového okna s tabulkou. Tím pádem byla časová jednotka z hlavičky odstraněna a sloupcům mohla být zmenšena šířka.



Obrázek 6.1: UI na konci 3. iterace

6.5 Testování

V části testování nedošlo k žádným výrazným změnám. Na backend části nebyly přidány žádné nové funkcionality, proto nebylo třeba přidávat další unit testy. Na frontend části došlo během implementace k úpravě vyhodnocení odhadu. Byly proto upraveny testy, které testují funkčnost vyhodnocení odhadu. Zároveň byly přidány další scénáře, které testují správnost vyhodnocení s několika možnými aktivními variantami.

6.6 Uživatelské testování

Aplikace od druhé iterace běží na testovacím prostředí, proto jsem během této iterace dostával od kolegů, kteří aplikaci používali, zpětnou vazbu. Jednalo se o návrhy na vylepšení v podobě UX změn a žádostí o přidání nové funkcionality. Zároveň jsem se nevyhnul několika chybám v aplikaci, na které se během vývoje a revize kódu nepřišlo. Právě díky uživatelskému testování se tyto chyby podařilo nahlásit, objevit jejich příčinu a všechny je opravit.

Aplikace je v této fázi sice ještě ve vývoji a neobsahuje veškerou funkcionalitu, ale uživatelské testování v této fázi se osvědčilo. Kdybych si na začátku projektu vybral odlišný přístup a uživatelům ukázal aplikaci až ve finální podobě, mohlo by se stát, že by ji nechtěli používat. Takto se mohli oni sami podílet částečně na jejím vývoji. Měli tak možnost odhalit věci, které se jim v aplikaci nelíbí. Myslím si, že díky této spolupráci budou aplikaci rádi a hojně v budoucnu používat. Je tedy velmi malá pravděpodobnost, že by aplikaci po nasazení nechtěli kolegové používat a aplikace by byla odsouzena k zániku.

Následuje výběr několika reportovaných chyb a vylepšení, které byly v rámci testování objeveny a realizovány.

- **Chybí celková pracnost** – Uživatelem bylo nahlášeno, že v přehledové tabulce jsou pouze součty jednotlivých kategorií a hodnoty záruky. Všiml si však, že v ní chybí celková suma všech kategorií a záruky. Uživatel tak nebyl schopný říci, jakou celkovou pracnost odhad má. Z programátorského hlediska se jednalo o jednoduchou opravu, z pohledu uživatele ale o poměrně zásadní chybu. Pravděpodobně došlo k přehlédnutí této chyby tak, že po dokončení odhadu si většina uživatelů stahovala odhad ve formě Excel exportu, kde tato suma byla. Oprava chyby byla rychlá a bezproblémová.

- **Chybějící mezisoučet** – Byl dán námět o doplnění funkcionality, která je v Excelu. Uživatel si chtěl označit několik buněk a vidět jejich součet a průměr. Jednalo se o jednoduchou funkcionalitu. Kdyby ale nebyla implementována, mohlo by se stát, že by uživatelé aplikaci neradi používali nebo by se vrátili k Excel šabloně. Funkcionalita byla proto implementována.

- **Schované prázdné řádky** – Uživateli přišlo nepřehledné, když viděl u prázdných řádků samé nuly. Nebylo tak na první pohled jasné, zda je daný řádek prázdný nebo obsahuje alespoň jedno vyplněné číslo. Jednalo se o praktické vylepšení, které nezabralo příliš času, a proto jsem ho implementoval.

6.7 Zhodnocení

V třetí iteraci se podařilo dosáhnout stanovených cílů. Byla implementována podpora helper funkcí a variant. Došlo k výraznému vylepšení UX a uživatelského rozhraní. Během implementace se narazilo na pár problémů s tvorbou uživatelského rozhraní. Ty byly nakonec vyřešeny změnou hlavní knihovny pro jeho tvorbu. Výsledkem je tedy sjednocení komponent a barev v uživatelském rozhraní. Zároveň došlo k další větší změně ve vývoji aplikace. Tou je použitý jazyk, nyní je aplikace napsána částečně v jazyce JavaScript a TypeScript. V další iteraci je proto naplánováno dokončení migrace na TypeScript. Zároveň během vývoje přišlo několik nápadů na vylepšení z řad samotných uživatelů. Většina jejich připomínek byla v běhu iterace schválena a zpracována.

V příští a zároveň poslední iteraci je naplánována integrace uživatelského účtu do aplikace, dokončení migrace aplikace do TypeScript a výraznější změna stránky Dashboardu, kde se nachází seznam odhadů. Zároveň se stále počítá se zpětnou vazbou od uživatelů z testování. Po zapracování všech zmíněných funkcionalit bude vytvořen release a aplikace bude nasazena do produkčního prostředí.

Kapitola 7

4. iterace

Na začátku této iterace aplikace obsahuje většinu funkcionalit, které byly před začátkem vývoje naplánovány. Tato iterace je zároveň poslední před nasazením první verze aplikace do produkčního prostředí. Je proto nutné nejdříve dokončit všechny plánované funkcionality a opravit nahlášené chyby, než bude aplikace nasazena pro užití v rámci větší skupiny uživatelů.

7.1 Podpůrné činnosti projektu

V této části se zaměřím na podpůrné činnosti na projektu.

7.1.1 Gradle Kotlin DSL

Jak je zmíněno v části 3.6.1.1, pro správu závislosti backend části používám nástroj Gradle. Ten je v současné chvíli používán ve starší verzi 4.2. Od verze 5 ale podporuje build skripty psané v jazyce Kotlin, konkrétně Gradle Kotlin DSL. Z důvodu zastaralosti jsem se proto rozhodl povýšit na novější verzi 6.2. V posledních verzích se změnila některá syntaxe, je třeba skripty předělat podle postupu na oficiálních stránkách. Protože musím zasahovat do samotných skriptů a nestačí jen povýšení verze, zvažuji, zda v rámci povýšení nezměnit i jazyk, ve kterém jsou skripty psané. V současné chvíli se v Gradle používá jazyk Groovy. Nově bych, jak jsem zmínil, mohl začít používat stejný jazyk, kterým je psaný veškerý kód backend části a to Kotlin. Bude sice nutné výrazněji zasáhnout do skriptů, existují ale návody, jak zmigrovat z Groovy na Kotlin. Kotlin má z mého pohledu navíc několik výhod oproti současnému řešení v Groovy. Není třeba znát jazyk Groovy, veškeré části skriptu lze psát v Kotlinu, ve kterém je navíc vyvíjen i backend. Kotlin je oproti Groovy typovaný jazyk, nese s sebou tedy jistá omezení, co se například dynamického generování úkolů týče. Na druhou stranu díky jeho typovosti může vývojové prostředí upozornit vývojáře na chybu ve skriptu a zároveň je schopno mu poskytnout dokonalejší našeptávání při psaní skriptů.

Nově jsou tak build skripty napsány kompletně v jazyce Kotlin a Gradle je používán ve verzi 6.2. Zároveň jsem v rámci povýšení verze Gradle musel upravit i verzi používaného Docker image na testovacím prostředí. Pokud bych verzi zapomněl upgradovat, nebylo by možné aplikaci pomocí Docker sestavit a celá GitLab Pipeline by selhala.

7.2 Analýza

V analýze této iterace se zaměřím na dlouhodobě plánované funkcionality, pro které jsem se rozhodl již při hrubém plánu iterací. Za prvé se jedná o integraci uživatelských účtů do aplikace. V současné době se do aplikace může dostat kterýkoliv zaměstnanec firmy a může upravovat jakýkoliv odhad. S tím souvisí i potřeba řešit práva při zobrazení a editaci odhadů. Druhou funkcionalitou je zpříjemnění práce se seznamem odhadů. V současné době se uživateli zobrazí na Dashboardu seznam všech odhadů, nově na tuto stránku budou přidány filtry pro jednotlivé sloupce a dále podpora pro řazení odhadů. Zároveň bude během této iterace vyřešeno dokončení migrace na TypeScript.

Začnu nejprve integrací uživatelských účtů do aplikace. Tuto funkcionalitu jsem se rozhodl ponechat až na samotný konec. Z pohledu vývoje se nejedná o funkcionalitu s vysokou prioritou. V aplikaci je sice potřeba tuto funkcionalitu mít, na tvorbu odhadu na testovacím prostředí to ale nemá vliv. Proto jsem se doposud zabýval spíše funkcionalitami, které měly dopad do samotné tvorby odhadu. Protože se jedná o firemní aplikaci, nedává mi smysl realizovat registrační proces. Místo toho využiji stávající uživatelské struktury uvnitř firmy a přihlašování postavím nad již existujícími účty. Ty zaměstnanci používají při přihlašování do většiny interních aplikací. Prvotní myšlenkou bylo implementovat si celý proces autentizace sám. V praxi by to znamenalo implementovat přihlašovací stránku na straně frontend části. Ta by volala autentizaci na backend části, která by byla řešena přes API. Na backend části by bylo ověření uživatele realizováno voláním dotazy na firemní LDAP server, kde jsou uloženy všechny uživatelské účty. Tato myšlenka se mi zamlouvala na začátku. Při podrobnější analýze jsem zjistil, že realizace celého autentizačního procesu je implementačně náročnější, než jsem si původně myslel.

Rozhodl jsem se proto podívat po dalším řešení, kterým by autentizace a autorizace byla realizovatelná. Shodou okolností řešil kolega před nedávnem podobný problém. Na základě diskuze s ním jsem se rozhodl zanalyzovat nabízené řešení od firmy Microsoft. Ve firmě je totiž používána služba Office 365 nabízená společností Microsoft. Díky ní má každý uživatel vlastní identitu a na jejím základě používáme například emailové účty. Po důkladné analýze možných řešení jsem narazil na knihovnu *Microsoft Authentication Library* (MSAL). „Knihovna MSAL umožňuje vývojářům získávat tokeny z koncových bodů Microsoft Identity Platform pro účely autentizace uživatelů a přístupům k zabezpečených webovým API.“[27, překlad vlastní]. Lze ji tak použít

i pro autentizaci v rámci vlastního API s pomocí služby *Azure Active Directory*, která je ve firmě používaná pro správu uživatelských skupin a uživatelů, respektive jejich identit. Knihovna navíc funguje podle rozšířeného principu OAuth 2.0 [28], neměl by proto být problém s její integrací do aplikace. V rámci analýzy jsem narazil i na knihovny, které bych v rámci implementace mohl využít na straně frontend i backend části. Z tohoto důvodu mi toto řešení přijde z pohledu pracnosti jednodušší. Dalším důvodem je také stránka bezpečnosti. Jedná se o službu, kterou Microsoft nabízí firmám. Z hlediska použití je tak výrazně menší pravděpodobnost, že by v aplikaci mohlo dojít k bezpečnostní chybě v rámci autentizace. Řešení pomocí LDAP serveru jsem z těchto důvodů proto zamítl a zvolil řešení od Microsoft.

Autentizaci uživatelů mám tedy vyřešenou, mohu se zabývat jejich autorizací vůči jednotlivým odhadům. Doteď všichni uživatelé vidí na Dashboard stránce všechny odhady. Tomu tak již nyní nebude. Uživatel tak uvidí pouze své odhady, které ho zajímají a na které má právo. Zároveň ale musí mít autor odhadu možnost přidat kterémukoliv uživateli právo pro zobrazení odhadu, aby s ním mohl například některé věci v odhadu konzultovat. Z tohoto důvodu bude moci uživatel vidět na Dashboardu své odhady a odhady, které s ním ostatní uživatelé budou sdílet. Posledním požadavkem, který byl v rámci analýzy identifikován, je možnost přidat některým z uživatelů právo tzv. „supervisora“. Skupina těchto uživatelů by měla mít možnost vidět všechny odhady napříč celou firmou. Z tohoto důvodu tak pravděpodobně vzniknou v aplikaci dvě skupiny uživatelů. Tuto problematiku však budu řešit až v rámci návrhu práv pro odhady 7.4.

Předposlední výraznou funkcionalitou, co bude přidána v rámci iterace před nasazením, je úprava Dashboardu. Doposud je na této stránce zobrazován pouze seznam odhadů pomocí tabulky. Uživatel si tak může zobrazit pouze odhady v pořadí, ve kterém jsou vráceny z API. Nově zde bude možné řadit odhady podle jednotlivých sloupců tabulky. Zároveň bude uživateli umožněno filtrování odhadů na základě hodnot jednotlivých sloupců.

Poslední funkcionalitou je podpora referencí a vzorců v aplikaci. Protože Excel šablona je postavená na programu Excel, je v něm samozřejmě možné používat reference na buňky a různé vzorce, na jejichž základě je možné získat hodnotu buňky. Rozhodl jsem se proto zařadit tuto funkcionalitu i do aplikace Estimate, kde bude podporována vždy v rámci jedné tabulky dané kategorie.

7.3 Návrh

V této části iterace se budu věnovat návrhovým věcem, které budou později realizovány v implementační části.

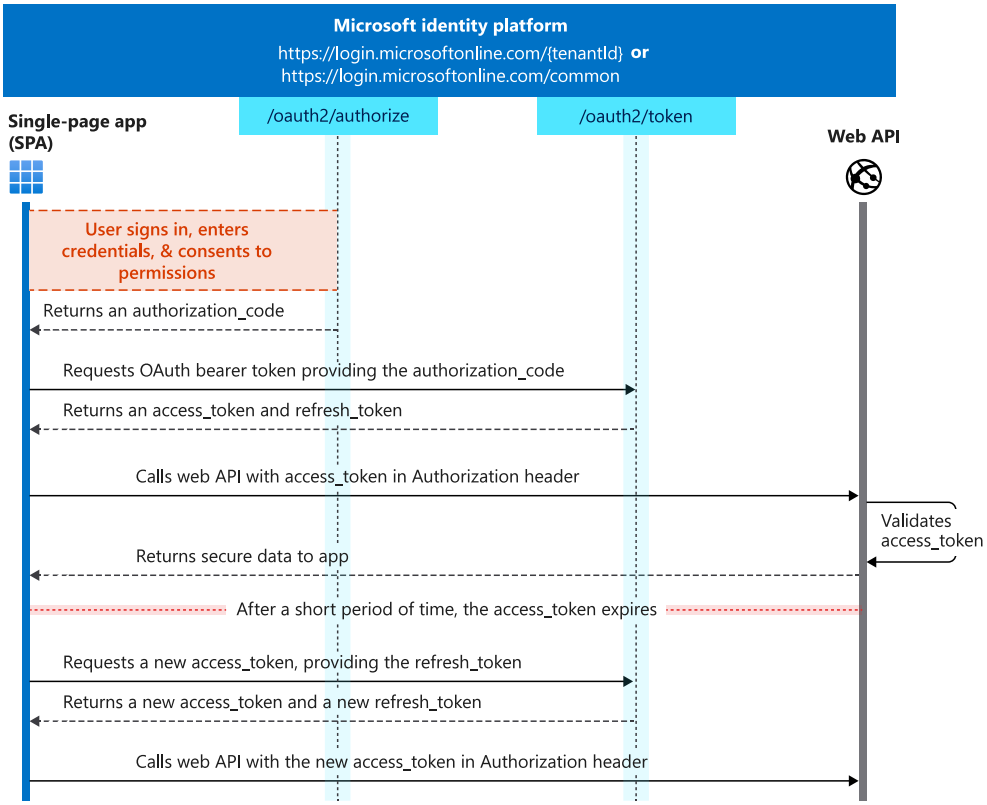
7.3.1 Autentizace uživatelů

Jak je zmíněno v podkapitole Analýza 7.2, rozhodl jsem se pro autentizaci uživatele využít knihovnu *Microsoft Authentication Library*. V rámci analýzy jsem se samozřejmě zabýval i průzkumem knihoven, které mohu použít pro backend a frontend část.

Pro implementaci backend část jsem se rozhodl použít knihovnu `com.microsoft.azure:azure-active-directory-spring-boot-starter`. „Tato knihovna pomůže s rychlým vytvořením autentizačního workflow pro webovou aplikaci, která používá Azure AD a OAuth 2.0 k zabezpečení její backend části.“[29, překlad vlastní] Knihovna pracuje ve frameworku *Spring* v kombinaci s jeho modulem *Spring Security*. Díky němu je možné v aplikaci ladit vlastní autentizaci a autorizaci uživatelů. Při správné konfiguraci Spring a použití *Security Filter Chain* filtru nabízeného knihovnou jsem schopný nastavit autentizaci pomocí MSAL. Knihovna zároveň umožňuje používat k autorizaci i role, které má uživatel přiděleny v rámci *Azure Active Directory*. Lze tedy např. ve firmě vytvořit skupinu, ve které budou uživatelé, kteří mají přístup k aplikaci. Ostatní uživatelé se proto do aplikace nedostanou.

Backend část je z pohledu návrhu hotová. Pro frontend část také existuje knihovna. Jedná se o `react-aad-msal`[30]. Nejedná se o oficiální knihovnu vydanou přímo společností Microsoft. Nicméně knihovna podporuje autentizaci uživatele a umožňuje správnou komunikaci s backend částí, na které je pro autentizace používána knihovna MSAL. Frontend knihovna umožňuje do určité části aplikace pustit pouze autentizovaného uživatele. Pokud se tedy do této části uživatel chce dostat, musí se nejprve přihlásit na stránce, na kterou je z aplikace přesměrován. Po úspěšném přihlášení je opět přesměrován zpět do aplikace.

V rámci návrhu jsem se rozhodl vytvořit jednoduchý PoC. Cílem je ověřit si, že kombinace obou výše zmíněných knihoven mi poskytne funkční řešení pro autentizaci uživatelů a umožní mi komunikaci mezi backend a frontend částí. V rámci PoC jsem proto provedl registraci nové aplikace na portálu Azure, abych získal požadované identifikátory autentizace. Ty je nutné znát, aby bylo možné provést konfiguraci obou knihoven. K vytvoření PoC jsem použil dostupné demo aplikace, které jsou u knihoven přiloženy, a upravil si je pro svou potřebu. Po několika problémech s konfigurací obou knihoven se mi povedlo zprovoznit komunikační kanál mezi frontend a backend částí. Ověřil jsem si, že obě knihovny mi poskytnou funkční řešení autentizace a mohu je bez starostí využít v implementaci Estimate. Princip, na kterém funguje autentizace uživatele, je zobrazen na Obrázku 7.1. SPA na obrázku je v mém případě frontend část a backend část se ukrývá pod pojmem Web API.



Obrázek 7.1: Workflow autentizace [31]

7.4 Řízení práv k odhadům

Jak je zmíněno v podkapitole Analýza 7.2, je potřeba vyřešit konkrétní autorizaci uživatelů a jejich práv vůči samotným odhadům. Z analýzy vyšlo, že uživatel používající aplikaci by neměl mít možnost vidět všechny odhady všech uživatelů. Rozhodl jsem se proto omezit jeho přístup k odhadům. Uživatel proto bude moci vidět a editovat pouze odhady, které jím byly vytvořeny. Zároveň z analýzy vyplývá, že by měl mít uživatel možnost sdílet svůj odhad s někým jiným. Pro tento případ jsem se rozhodl uchovávat v modelu množinu uživatelů, se kterými je odhad sdílen. Do této množiny bude moci autor odhadu přidávat nebo odebírat konkrétní uživatele. Ti budou mít právo si odhad v aplikaci zobrazit. Editovat ho však moci nebudou, toto právo zůstane výhradně v rukou autora.

Na základě předchozí definice pravidel jsem se proto rozhodl v aplikaci použít dva typy akcí. Na jejich základě budu schopný řídit práva jednotlivých uživatelů k odhadům. Jedná se o akce READ a WRITE. Na základě těchto akcí budu schopný řídit práva uživatelů k provedení akce na konkrétním odhadu.

V případě akce READ nad odhadem umožním tuto akci provést autorovi odhadu a uživatelům, se kterými je odhad sdílen. Jedná se o akci, kdy si uživatel bude chtít zobrazit seznam odhadů nebo detail konkrétního odhadu. Druhou akci WRITE může provést pouze samotný autor odhadu, nikdo jiný nebude mít možnost odhad editovat.

Z analýzy také vyplynula potřeba mít takzvanou skupinu supervisorů. Proto jsem se rozhodl rozdělit uživatele v aplikaci do dvou skupin. První skupinou budou běžní uživatelé. Pro ty budou platit běžná práva, která byla představena. Druhou skupinou budou supervisor uživatelé. Mezi ty budou patřit pouze vybraní kolegové. Od běžných uživatelů se budou vyznačovat tím, že budou mít automaticky právo pro akci READ nad všemi odhady. Právo na akci WRITE pro ně ale bude omezeno stejně jako pro běžné uživatele, tedy pouze pokud bude tento uživatel autorem, bude moci odhad editovat. Mezi tyto uživatele budou typicky patřit tzv. Delivery manažeři, projektoví manažeři a další vybraní kolegové a kolegyně. Správa uživatelů ve skupinách není řešena na úrovni projektu. Jejich správu si bude řešit firemní oddělení ITS.

7.4.1 Úprava modelu odhadu

Protože v této iteraci má být přidáno několik nových funkcionalit, je nutné na ně také připravit samotný model odhadu.

První změna odhadu souvisí s uživateli a řízením práv k odhadům. Protože chci mít možnost k jednotlivým odhadům přidat uživatele, se kterými je odhad sdílen, bude nutné tyto uživatele mít uloženy v modelu odhadu. Z tohoto důvodu jsem se rozhodl do modelu přidat atribut `sharedTo`. Pod tímto atributem se nachází množina uživatelů, kterým uživatel umožní náhled na odhad. Dále, protože je aplikace schopna poznat aktuálně přihlášeného uživatele, budou přidány do modelu následující dva auditní atributy.

Prvním z nich je, kdo provedl poslední změnu na odhadu. Jedná se o atribut `lastModifiedBy`. Momentálně sice nemůže změnu provést nikdo jiný než sám autor, tento atribut se ale v budoucnosti plánuje využít. Druhým z nich je atribut `whenLastModified`, pod kterým je uložen čas poslední změny odhadu.

Další úpravu, kterou je nutné provést, je příprava na reference a vzorce. Ke každé hodnotě musí být možné přidat referenci či vzorec, který bude v textové podobě. Proto jsem každému modelu `EstimateValue`, který představuje danou hodnotu položky, přidal třetí atribut `formula`. Ten může buď obsahovat textovou reprezentaci vzorce či být prázdný.

Jak bude model odhadu po těchto změnách vypadat, je možné se podívat ve Zdrojovém kódu 7.1. Zde je model zobrazen pomocí pseudoobjektů.

7.4.2 Dashboard – řazení a filtrování

Na Dashboardu jsou odhady momentálně zobrazovány pomocí tabulkové komponenty z knihovny `material-ui`. Jedná se o jednoduchou komponentu, která

```
EstimateValue := {
    coefficient: Coefficient,
    value: number,
    formula: string | null
}

Estimate := {
    ...
    sharedTo: Set<string>,
    lastModifiedBy: {
        username: string
        name: string
    },
    whenLastModified: Date
}
```

Zdrojový kód 7.1: Doménový model odhadu v 4. iteraci

umožňuje zobrazovat data v řádcích a sloupcích. V plánu je na této stránce filtrování a řazení podle jednotlivých sloupců. Rozhodl jsem se tuto funkcionalitu neimplementovat sám a sáhnout po knihovně třetích stran. Z rychlého průzkumu a PoC mi vzešla jako jasný vítěz knihovna `material-table`[32]. Jak může název napovídat, jedná se o knihovnu, která poskytuje vývojářům propracovanou tabulkovou komponentu se spoustou užitečných funkcionalit. V základu obsahuje požadované filtrování i řazení. Navíc do budoucna podporuje například i vzdálená data a stránkování, která získává přes REST API. Tabulka je implementovaná pomocí knihovny `material-ui`, perfektně tak zapadá do celkového vzhledu aplikace. Není tedy třeba řešit její integraci do Material Design stylu.

7.5 Implementace

Nyní se zaměřím na samotnou část realizace aplikace v poslední iteraci. Podkapitola obsahuje několik částí, které jsou logicky rozdělené podle jednotlivých funkcionalit.

7.5.1 Autentizace

První věcí, kterou je v implementaci třeba řešit, je zprovoznění autentizace v aplikaci. Jak nakonfigurovat a zprovoznit konfiguraci knihoven, jsem si vyzkoušel již v rámci malého PoC v části 7.3.1. Mám tak ověřené, že knihovna je funkční. Mohu proto použít částečně zdrojové kódy z PoC.

```
@EnableGlobalMethodSecurity(securedEnabled = true,
prePostEnabled = true)
class WebSecurityConfig(
val aadAuthFilter: AADAuthenticationFilter
) : WebSecurityConfigurerAdapter() {
    override fun configure(http: HttpSecurity) {
        http.cors().and()
        .csrf().disable().headers().frameOptions().disable()

        http.authorizeRequests()
         .antMatchers("api/**")
         .authenticated()

        http.addFilterBefore(aadAuthFilter,
        UsernamePasswordAuthenticationFilter::class.java)
    }
}
```

Zdrojový kód 7.2: Konfigurace autentizace backend části

Začnu nejprve backend částí. Nejprve je třeba přidat do závislostí knihovnu `azure-active-directory-spring-boot-starter`. Po jejím přidání je třeba ještě nastavit autentizaci v samotné aplikaci. Aby autentizační knihovna mohla ve Spring aplikaci pracovat správně, je třeba přidat závislost na Spring knihovně, která řeší bezpečnost. Tou je knihovna `spring-boot-starter-security`. Po jejím přidání mohu vytvořit konfigurační třídu `WebSecurityConfig`, kterou jsem schopný nakonfigurovat zabezpečení aplikace proti neautentizovaným uživatelům. Vývoj backend a frontend části probíhá na rozdílných doménách, je proto třeba povolit Cross-origin resource sharing (CORS). Ten mi umožní volat API z aplikace na jiné doméně. Požadavky na API tak budou validní. Dále musím nastavit, aby požadavky na API mohli zasílat pouze autentizovaní uživatelé. Poslední úpravou pro správnou autentizaci pomocí MSAL je přidání Spring filtru. Jedná se konkrétně o autentizační filtr `AADAuthenticationFilter`. Tento filtr je z knihovny MSAL. Musí být přidán před výchozí Spring autentizační filtr. Samotný filtr provede autentizaci uživatele a pokud například kvůli špatným údajům nelze uživatele autentizovat, vygeneruje výjimku. Celá konfigurační třída `WebSecurityConfig` je k nahlédnutí ve Zdrojovém kódu 7.2. Aby konfigurace byla kompletní, je potřeba přidat do `application.properties` některé konfiguračních property vygenerované na Azure portálu a dále seznam skupin uživatelských skupin, které se mohou do aplikace přihlásit.

Po nastavení backend části zbývá vyřešit frontend část. Pro ni využiji kni-

```
<AzureAD provider={authProvider} forceLogin>
{({
    login,
    logout,
    authenticationState,
    accountInfo
  }: IAzureADFunctionProps) => {
        ...
  }
</AzureAD>
```

Zdrojový kód 7.3: Autentizace na frontend části

hovnu `react-aad-msal`, kterou jsem použil v rámci PoC. Abych zajistil, že se do aplikace dostane pouze autentizovaný uživatel, obalím obsah komponenty `App` knihovní komponentou `AzureAD`. Této komponentě předám celkem dvě property. V property `provider` předám veškerou konfiguraci pro autentizaci na straně MSAL. Ta obsahuje i typ metody, kterou si uživatel v aplikaci autentizuje. Na výběr je přihlášení formou popup okna nebo pomocí přesměrování na přihlašovací stránku Microsoft. Rozhodl jsem se pro přihlášení formou přesměrování. Zároveň jsem v komponentě vynutil automatické přesměrování na přihlašovací stránku v případě, kdy uživatel není autentizovaný pomocí property `forceLogin`. Uživatel je při spuštění aplikace přesměrován na přihlašovací stránku, zde zadá jméno a heslo firemního účtu. Pokud jsou zadané údaje správné, dojde k přesměrování zpět do aplikace se správnými autentizačními údaji. V aplikaci je poté třeba jen správně nastavit autentizační hlavičky dotazů posílaných na API, aby na backend části došlo k ověření uživatele. Jak využití knihovny vypadá lze vidět ve Zdrojovém kódu 7.3.

7.5.2 Autorizace a refactoring backend části

Po hotové autentizaci zbývá z pohledu zabezpečení přístupu k odhadům implementovat práva k jednotlivým odhadům. Je proto potřeba modifikovat jednotlivé metody pro získání samotného odhadu a zároveň upravit i dotaz do databáze pro získání všech odhadů, na které má uživatel právo. Doposud jsou všechny tyto metody realizovány přes volání rozhraní `EstimateDao`, jak je možné vidět v implementačních pohledu na Obrázku 4.5. V implementaci tohoto rozhraní se pouze převolávají metody nad repozitářem `EstimateRepository`. Protože při získávání seznamu i jednotlivých odhadů je třeba řešit i jejich práva, rozhodl jsem se backend část výrazněji předělat.

V první řadě je třeba nadefinovat rozhraní, přes které se budou kontrolovat práva k samotným odhadům. K tomu slouží rozhraní `AuthorizationService`, které je následně implementováno. To obsahuje celkem dvě metody. První

z nich je `hasUserPrivilege`. Ta kontroluje, zda má uživatel na konkrétní odhad požadované právo, a podle nároku vrací Boolean hodnotu. Druhá metoda se nazývá `hasUserPrivilegeToAllEstimates` a kontroluje, zda má uživatel požadovaná práva nad všemi odhady, čili zda se jedná o supervisora. Aby bylo možné poznat skupinu supervisorů, je v `application.properties` přidána property `estimate.security.supervisorGroups`. V té je uložen seznam skupin, které se považují za supervisorské, a v implementaci autorizační služby se s těmito skupinami pracuje. Uživatel je získáván ze Spring contextu, kam se uloží během procesu autentizace.

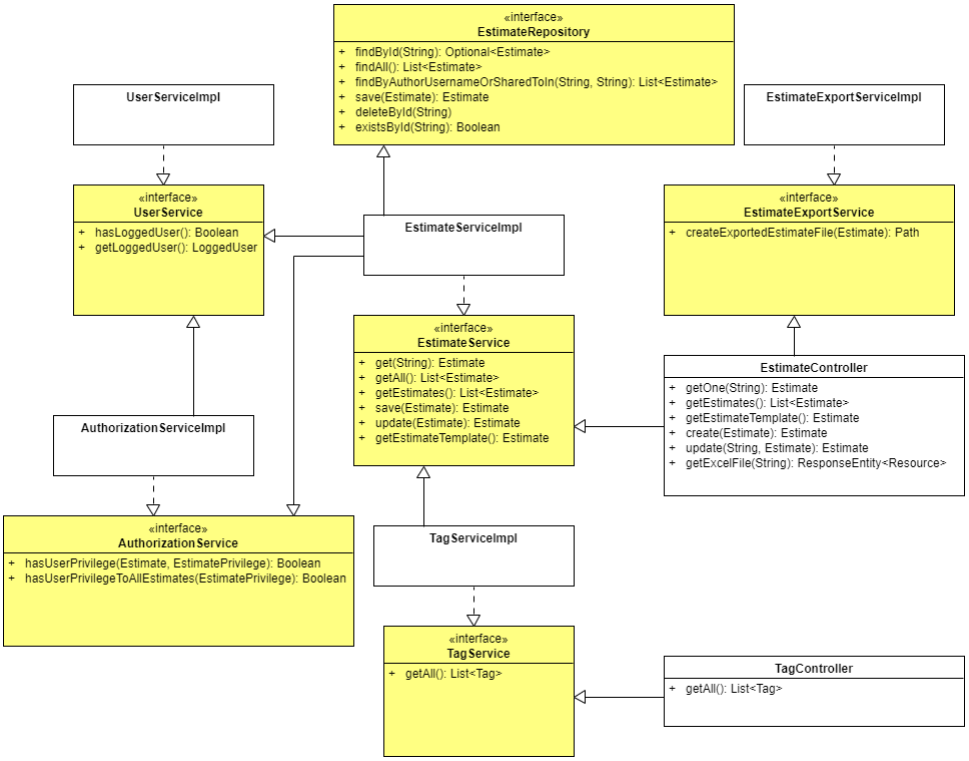
Ověření práv uživatele pro příslušný odhad či odhady jsem schopný provést, je potřeba ho tedy při volání metod použít. Zatím jsou všechny metody pro získání odhadů volány přes `EstimateDaoImpl`. Zde se nachází pouze volání databázových operací. Protože třída nebude volat pouze metody z rozhraní `EstimateRepository`, ale bude obsahovat i aplikační logiku, provedl jsem několik úprav. Aby z názvu třídy bylo na první pohled patrné, že obsahuje i logiku, přejmenoval jsem implementaci rozhraní na `EstimateServiceImpl` a stejně jsem přejmenoval i rozhraní, které implementuje. Během implementace aplikační logiky je použita třída pro správu autorizačních práv k odhadům.

Během refactoringu došlo i k úpravě dalších tříd. Jedná se například o třídu `EstimateRepository`. V té došlo k přidání metody `findByAuthorUsernameOrSharedToIn`, která umožňuje hledat odhady na základě uživatelského jména. Metoda najde odhady, kde je uživatel jako autor nebo osoba, se kterou je odhad sdílen. Jak celá backend části po refactoringu vypadá, je možné vidět v implementačním náhledu na Obrázku 7.2.

7.5.3 Dashboard

Jak je zmíněno v návrhu 7.4.2, pro implementaci tabulky na Dashboardu použiji knihovnu `material-table`. Knihovna poskytuje komponentu `MaterialTable`, ve které je implementováno několik funkcionalit. Tabulka jednoduše nabízí filtrování a řazení nad daty, která jsou tabulce poskytnuta. Vývojář tak nemusí žádnou funkcionalitu implementovat, pouze v komponentě vše správně nastaví. Součástí komponenty jsou dvě property, které bych rád zmínil.

První z nich je konfigurace sloupců, které jsou v tabulce zobrazeny. Ty jsou komponentě předávány přes property `columns` ve formátu pole, kdy každá položka pole musí obsahovat následující věci. Ve sloupci je třeba definovat název hlavičky pomocí klíče `title`. Dále je třeba určit, jaký atribut se ve sloupci z nabízených dat zobrazuje pomocí klíče `field`. Na základě něj poté probíhá řazení a filtrování nad daty. Volitelně může sloupec definovat, zda je možné pomocí něj filtrovat, řadit a zda má být zahrnut do globální hledání při vyhledávání v horní liště tabulky. Jak definice sloupců vypadá lze vidět ve Zdrojovém kódu 7.4.



Obrázek 7.2: Implementační pohled na `estimate-backend` modul

```
[
    {
        title: formatMessage({id: 'dashboard.name'}),
        field: 'name',
    },
    ...
    {
        title: formatMessage({id: 'dashboard.whenLastModified'}),
        field: 'whenLastModified',
        filtering: false,
        render: estimate =>
            moment(estimate.whenLastModified)
                        .format(DateFormats.DD_MM_YYYY_HH_MM)
},
```

Zdrojový kód 7.4: Definice sloupců pro `material-table`

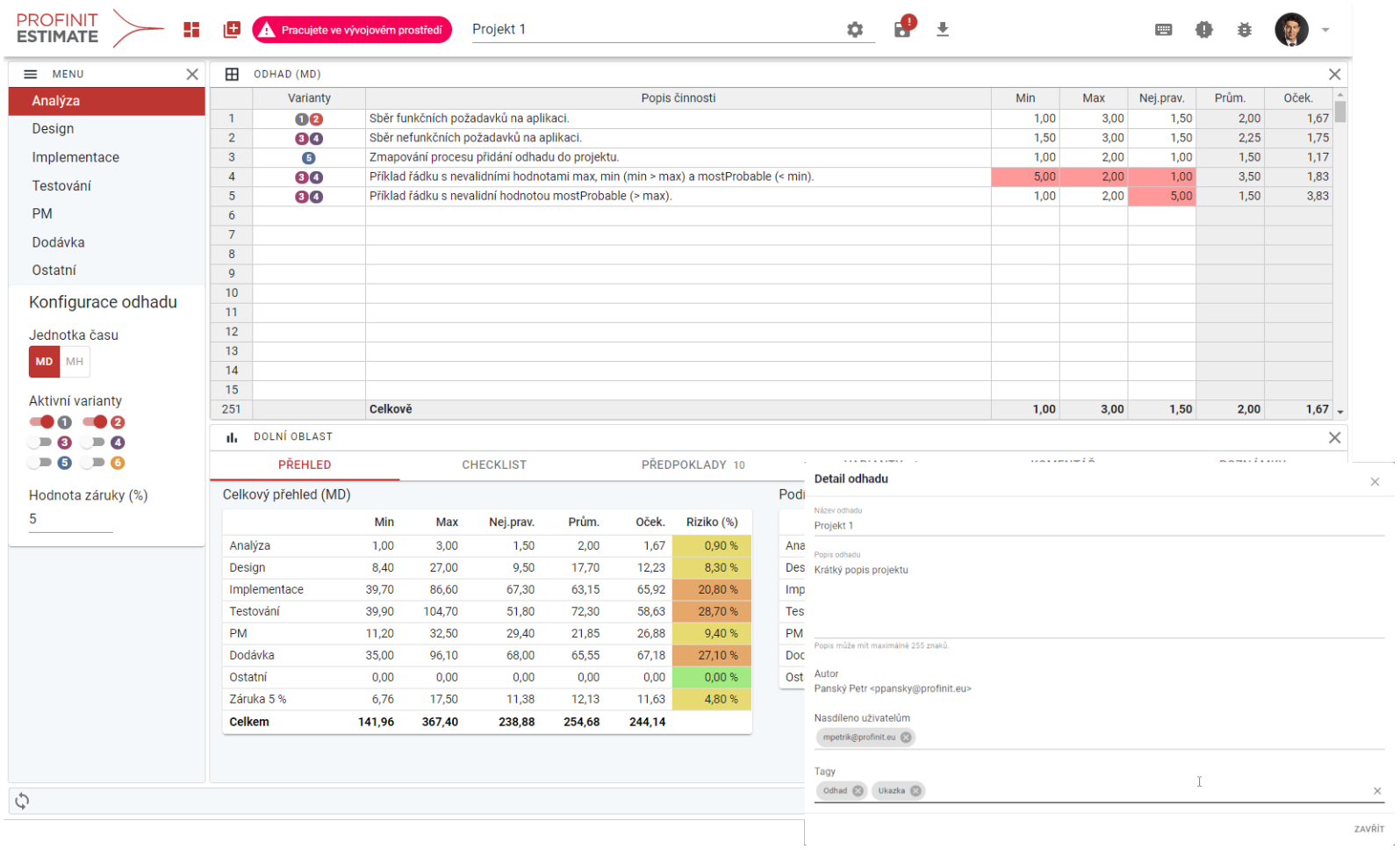
Dále komponenta nabízí celkem dva způsoby, jak je možné tabulce data poskytovat. První z nich je možnost komponentě předat všechny odhady v jednom poli. Druhý způsob nabízí využití stránkování tabulky a data jsou získávána vždy až v momentě, kdy se má uživateli zobrazit daná stránka. V této fázi projektu jsem si vybral z pohledu náročnosti implementace jednodušší variantu. Tou je první zmiňovaná, kdy jsou data tabulce předávána pomocí pole. Při tomto způsobu probíhá řazení, filtrování a globální vyhledávání nad daty v poli a tyto funkcionality jsou v tabulce již připraveny. V případě použití vzdálených dat by API muselo podporovat stránkování, řazení a filtrování. Tyto funkcionality jsou plánovány až později v rámci dalšího rozvoje aplikace.

7.5.4 Problém s licencí AgGrid

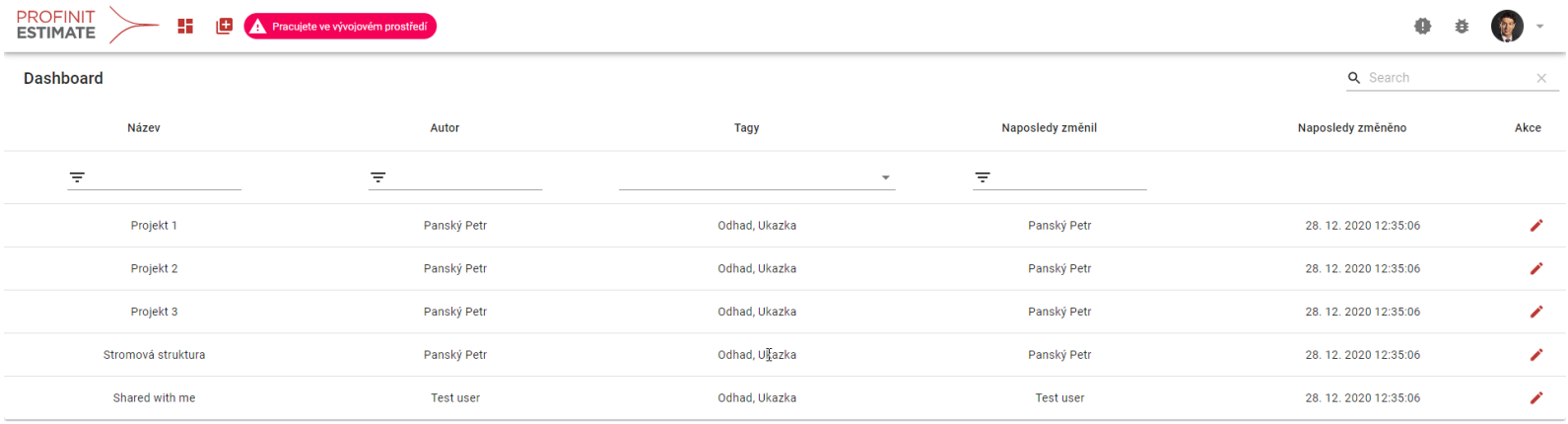
Necelé tři týdny před finálním nasazením začínáme řešit koupi licence knihovny `AgGrid`. Jak je zmíněno v analýze 2. iterace 5.2, knihovna je vázaná na projektu na počet produkčních prostředí. V tomto případě je tedy potřeba pouze jedna licence pro produkci. Při čtení licenčních podmínek jsme ale narazili na problém. Licenční podmínky se od druhé iterace výrazně změnily. Nyní je koupě produkční licenci podmíněna vlastnictvím vývojové licence. Vývojová licence je v tomto případě potřeba pro každého vývojáře, který na projektu pracuje. V praxi to tedy znamená, aby na projektu mohlo pracovat více lidí, musí mít každý zakoupenou licenci. Protože se ale jedná o interní projekt a může se na něm střídat více lidí, bylo by potřeba zakoupit zhruba 4 vývojové licence a jednu produkční. Tím pádem se cena licence vyšplhá na pětinásobek původní ceny. Při těchto podmínkách je knihovna poměrně drahá a navíc se může stát, že zaplacenou licenci nebude určitou dobu ani nikdo využívat, protože velikost vývojového týmu na projektu v budoucnu může neustále měnit.

Rozhodl jsem se proto zjistit více informací o původně používané knihovně Handsontable. Cena za její licence vychází velmi podobně jako samotná produkční licence AgGrid. Na rozdíl od AgGrid ale není třeba žádných dalších licencí. Navíc je licence potřeba pouze pro vývojáře, který pracuje s tabulkovou komponentou. V tomto případě tedy postačí pouze jedna licence. Za těchto podmínek se knihovna Handsontable z ekonomických podmínek jeví výhodněji. Rozhodl jsem se proto vyzkoušet, zda je možné současně používanou komponentovou tabulku zmigrovat zpět do knihovny Handsontable.

V odděleném issue si proto přidám mezi závislosti právě tuto knihovnu a pokusím se pomocí ní nahradit současně používanou AgGrid komponentu. Po několika hodinách práce se mi daří zprovoznit základní funkčnost tabulky v aplikaci na knihovně Handsontable. Z tohoto důvodu jsem se rozhodl vrátit na původně používanou tabulku Handsontable. V rámci dalších několika úkolů jsem proto převedl veškerou funkcionalitu na nově používanou knihovnu. Poté byla zakoupena licence pro jednoho vývojáře. Z licenčního pohledu již tedy nic nebrání tomu, aby aplikace byla nasazena do produkce.



Obrázek 7.3: UI na konci 4. iterace



Obrázek 7.4: UI Dashboardu na konci 4. iterace

Modul	Třídy	Metody	Řádky
`estimate-backend`	87% (29/33)	81% (61/75)	68% (111/161)
`estimate-domain`	100% (28/28)	99% (103/104)	98% (295/301)
`estimate-export`	63% (7/11)	76% (13/17)	87% (27/31)
Celkem	88% (64/72)	90% (177/196)	87% (433/493)

Tabulka 7.1: Pokrytí backend modulů testy (pouze Kotlin)

7.6 Testování

Protože byli v této iteraci do aplikace přidáni uživatelé, došlo k úpravě unit testů aplikace. V prvé řadě přibyl autentizovaný uživatel, proto bylo v rámci testování Controlleru potřeba otestovat i dotazy v případě, kdy uživatel není autentizovaný. Dále bylo třeba plně pokrýt testy nově vzniklou třídu `EstimateServiceImpl`. Ta obsahuje veškerou aplikační logiku. Řeší se v ní, jak práva na jednotlivé odhady jednotlivých uživatelů, tak komunikace s databázovým repositářem, export, ... Velká váha testování je kladena především na oblast odhadů a práv uživatelů pro čtení a zápis. Pokrytí testy všech tří backend modulů se pohybuje kolem 90 %, což je slušný výsledek. Jak jsou na tom s pokrytím jednotlivé moduly a pokrytí celkově je možné vidět v Tabulce 7.1.

Na frontend části nedošlo z pohledu unit testů k žádným větším změnám. Většina komponent se často mění. Proto v tomto případě sázím místo testů především na kombinaci testování od vývojářů a uživatelské testování. Vývojář si po přidání funkcionality aplikaci prokliká a otestuje, zda pracuje správně. Zároveň pokud se některá z chyb dostane do testovacího prostředí, je velká šance, že se na ni přijde včas díky uživatelskému testování. Do budoucna, až se vývoj frontend části ustálí a kapacita vývojového týmu ze zvětší, je plánováno přidání UI testů.

7.7 Uživatelské testování

Díky uživatelskému testování došlo v této iteraci k objevení několika chyb, kterých jsem si doposud nevšiml. Chyby tak byly pohotově opraveny. Zároveň se během testování podařilo získat od několika uživatelů zpětnou vazbu. Na jejím základě došlo v aplikaci k několika úpravám.

7.7.1 Průběžné ukládání odhadu

Během uživatelského testování se jednomu z kolegů stalo, že začal v aplikaci vytvářet odhad. Když už měl odhad z větší části hotový, spadl mu prohlížeč a o celý odhad přišel. Z pohledu funkčnosti se tak jedná o poměrně velký problém, který je potřeba řešit. Z tohoto důvodu je do aplikace přidána funk-

cionalita, která po nějakém čase vyzve uživatele k uložení odhadu. Zároveň byl do modelu odhadu přidán atribut `internalVersion`, v kterém je uložená verze odhadu. Čím vyšší verze je, tím je odhad novější. Pokud tedy uživatel edituje již existující odhad, ukládá se jím změněný odhad každých pět minut do lokálního úložiště *LocalStorage* a interní verze odhadu se po každém uložení zvýší. Pokud mu během editace tedy aplikace havaruje, nepřijde o všechny změny. Při návratu na editační stránku se v aplikaci porovnají verze z *LocalStorage* a načteného odhadu. Pokud je verze odhadu z *LocalStorage* vyšší, nabídne aplikace uživateli pokračovat v editaci přerušeného odhadu. Uživatel tak může pokračovat v editaci verze odhadu, kterou předtím neměl šanci uložit.

7.7.2 Poznámky odhadu

Doposud neměl uživatel možnost si v aplikace ukládat někde bokem poznámky k odhadovému projektu. Na základě zpětné vazby jsem se to rozhodl změnit. Nově má uživatel možnost napsat si k projektu vlastní poznámky, které mu zůstanou zachovány napříč změnami odhadu. Kvůli větší přehlednosti a možnosti strukturovat si poznámky je podporována syntaxe Markdown [13]. Tu je možné znát z například souborů README na serverech jako jsou GitHub, GitLab aj. Poznámky jsou umístěny v dolní oblasti v nově přidaném tabu.

7.8 První release

V rámci diplomové práce je tato iterace poslední a veškeré práce v rámci ní jsou hotové. Zbývá tedy již jen vytvořit release verzi. Protože se jedná o první release, je mu přiřazena verze 1.0.0.

V první řadě je třeba před nasazením domluvit s oddělením ITS uživatelské skupiny, které budou v aplikaci použity. Jedná se o skupiny, na jejich základě aplikace rozezná, zda je jedná o běžného uživatele nebo supervisora. Po několika vyměněných emailech a diskuzi s oddělením ITS jsme se dohodli, že pro aplikaci budou vytvořeny nové uživatelské skupiny. Do běžné skupiny uživatelů proto budou přidáni všichni zaměstnanci, kteří mohou aplikaci použít. Druhá skupina supervisorů je pouze podmnožinou běžných uživatelů a jsou zde pouze vybraní uživatelé. Správa obou skupin je a bude v rukou ITS oddělení.

Aby aplikace mohla být nasazena, je potřeba nejprve vytvořit a nakonfigurovat produkční prostředí. Proto byl po dohodě s ITS vytvořen server se základní konfigurací. Ten je nakonfigurován podobným způsobem jako testovací prostředí 5.1.2. Na serveru tak běží systém CentOS, obsahuje MongoDB databázi a aplikace funguje jako plnohodnotná systémová služba.

S takto vytvořeným produkčním prostředím je nutné ještě upravit Pipeline nástroje GitLab CI. Pipeline nyní podporuje pouze nasazení na testovací server. Nově je proto v konfiguraci GitLab CI přidána fáze `deploy_prod`. Co

se týče samotného skriptu pro nasazení aplikace na server, ten se od toho pro testovací prostředí neliší. Jediné, v čem se liší, je, že nasazení probíhá na produkční stroj. Aby bylo zaručeno, že se nasazení na produkci provede pouze v případě, že je opravdu vyžadováno, je možné stejně jako v případě testovacího prostředí nasadit pouze z master větve. Zároveň je při spuštěné Pipeline od vývojáře vyžadováno, aby po úspěšném dokončení celé Pipeline ručně spustil fázi nasazení na produkci. Pokud tak neučiní, aplikace se na produkční prostředí nenasadí. Jak část konfiguračního souboru vypadá, je možné vidět v ukázce Zdrojového kódu 7.5.

```
deploy_prod:
    stage: deploy
    script:
        ...
tags:
    - estimate-shell-prod
when: manual
only:
    - master
```

Zdrojový kód 7.5: Definice nasazení na produkci pomocí GitLab CI

S takto nastavenými procesy nezbývá, než vytvořit release samotné aplikace pomocí služby GitLab Releases [33]. Jedná se o verzi 1.0.0. V rámci vytvoření release je přidán krátký popis funkcionalit, které byly přidány. Zároveň je v rámci vytvoření release aplikace nasazena na produkční prostředí, které se nachází uvnitř interní sítě firmy. Aplikaci je tak od této chvíle možné použít v produkčním prostředí, kde je dostupná všem kolegům.

7.9 Zhodnocení

V rámci poslední iterace se podařilo dokončit veškerou požadovanou funkcionalitu aplikace. Do aplikace bylo přidáno přihlašování uživatelů na základě firemních účtů. Zároveň s tím byla přidána kontrola práv odhadů. Výrazných změn se dočkala i stránka Dashboard, kde je vylepšen list odhadů. Nyní je tak možné filtrovat a řadit záznamy. Během iterace se vyskytlo i několik komplikací, které se podařilo zdárně vyřešit. Jednalo se hlavně o změnu licencování tabulkové knihovny a řešení požadavku z uživatelského testování s automatickým ukládáním změn odhadu a jeho následnou obnovou při násilném vypnutí prohlížeče. Mohu tedy konstatovat, že v iteraci došlo ke splnění předem stanovených požadavků. Výsledkem celé iterace je první release aplikace Estimate, který je nasazen na produkčním prostředí.

Kapitola 8

Plán budoucího rozvoje

Vývoj aplikace v rámci diplomové práce je již ukončen a řešení je dostupné uživatelům na produkčním prostředí, kde v něm mohou zakládat a pracovat s odhady. K datu publikace této diplomové práce se v produkční databázi aplikaci nachází celkem 77 odhadů od 31 uživatelů.

Po čtyřech proběhlých iteracích je možné vidět zásadní pokrok jak v rozsahu funkcionalit, tak v samotném uživatelském rozhraní aplikace. Přidávané funkcionality bylo možné detailně sledovat v rámci jednotlivých kroků a tak získávat okamžitou zpětnou vazbu od uživatelů. K porovnání uživatelského rozhraní na začátku a na konci práce nejlépe poslouží snímky obrazovek z první a poslední (čtvrté) iterace. Snímek obrazovky po první iteraci je k vidění na Obrázku 4.7. Naopak finální vzhled aplikace po poslední iteraci je na Obrázcích 7.3 a 7.4. Při porovnání snímků obrazovek s uživatelským rozhraním je patrné, že aplikace ušla od začátku vývoje výraznou cestu. Rozhraní se výrazně vylepšilo, optimalizovalo se umístění jednotlivých prvků dle potřeb uživatelů a stylově se sladilo do jednoho tématu.

V rámci diplomové práce jsem vývoj ukončil poslední čtvrtou iterací. Vývoj samotný ale zdaleka nekončí. V plánu je pokračovat ve vývoji dále a implementovat další funkcionality, které v rámci diplomové práce nebyly řešeny. V této kapitole se proto pokusím rozvrhnout realizaci dalších funkcionalit. Protože se vývoj v iteracích osvědčil, hodlám v něm pokračovat i nadále. Proto funkcionality rozdělím do několika iterací. Po každé takovéto iteraci bude následovat vytvoření nové produkční verze aplikace. Ta bude nasazena do produkčního prostředí. Poté se začne pracovat na realizaci následující iterace.

Po zvážení počtů iterací a funkcionalit jsem se rozhodl budoucí vývoj rozdělit do celkem tří nejbližších iterací. Plánovat vývoj funkcionalit na delší časové období mi v současné chvíli nedává smysl. V plánu se tedy nachází pouze funkcionality, na které vznikl požadavek. Neznamená to ale, že v rámci zpětné vazby od uživatelů nedojde k zařazení dalších funkcionalit, na které se zatím nepřišlo, ale které budou v rámci zpětné vazby zjištěny. Iterace jsou číslovány nezávisle na iteracích, které byly zmíněny v této práci. Každá z nich

obsahuje seznam nových funkcionalit a jejich stručný popis. V následujících podkapitolách je tedy možné vidět plán dalšího postupu.

8.1 Iterace 1

- **Podpora Undo/Redo** – Nyní je v aplikaci jakákoliv změna a její uložení nevratná akce. Při přepsání původních hodnot novými a uložení tak tyto hodnoty již nelze získat zpět. Je proto nutné přidat podporu pro akce zpět a znovu (dopředu) jako je tomu například právě v původně používané Excel šabloně.

- **Editace odhadu více uživateli** – V současné verzi aplikace může odhad editovat pouze autor samotný. Ostatní uživatelé si odhad mohou pouze zobrazit, pokud je s nimi sdílen nebo jsou v supervisorské skupině. V rámci této iterace proto bude řešena editace odhadu více uživateli, aby na něm mohlo spolupracovat více osob. Způsob, jakým bude editace řešena, zatím není zcela upřesněn. Nabízí si však možnost využití například algoritmu ve smyslu "výlučný zámek pro editaci", apod.

- **Archivace odhadu** – Odhad po nějakém čase již není aktuální a není potřeba, aby byl se uživateli na Dashboardu zobrazoval. Doposud ale není možné v aplikaci jeho smazání. Po této iteraci by již mělo být možné odhad smazat. Aby ale nedošlo po smazání k jeho fyzické ztrátě v databázi, rozhodl jsem se zavést místo smazání jeho archivaci. Odhad bude pořád v databází, ale bude možné odlišit aktivní a archivované odhady. Aby bylo možné zobrazit si jak aktivní, tak archivované odhady, bude nutné upravit částečně Dashboard. Po této úpravě musí být možné si na něm zobrazit v jednom okamžiku pouze aktivní nebo archivované odhady pravděpodobně na základě filtru. Zároveň by mělo být možné z Dashboardu daný odhad archivovat nebo aktivovat v závislosti na jeho stavu.

- **Kopírování odhadů** – Aplikace by měla umožňovat vytvořit na základě existujícího odhadu jeho kopii a umožnit její uložení. Zároveň by měla být během kopírování zachována reference na původní odhad na základě jeho identifikátoru. V budoucnu by se tato informace mohla spolu s ostatními metadaty využít například pro některé statistiky.

- **Formulář s výběrem šablony** – Aplikace se musí před přidáním dalších šablon připravit na používání s více šablonami. Doposud se používá pouze výchozí šablona, nebylo proto potřeba ji vybírat. Daná šablona se použije vždy. V příští iteraci ale plánuji přidání dalších šablon. Proto by se před samotným vytvořením odhadu měl uživateli nabídnout formulář, ve kterém vybere typ šablony a vyplní základní informace o odhadu.

- **Filtrování vlastních odhadů** – Uživatel by měl mít možnost si na Dashboard stránce vyfiltrovat pouze odhady, u kterých je vedený jako autor. U uživatelů ze supervisorské skupiny, kteří vidí všechny odhady, by se mohlo stát, že odhadů vidí na Dashboardu velké množství. Tento filtr by jim měl usnadnit práci s vlastními odhady.

8.2 Iterace 2

- **Klávesové zkratky** – Do aplikace budou přidány klávesové zkratky. Nyní musí uživatel používat velmi často myš i na běžné operace. Šikovné využití klávesových zkratek může snížit nutnost použití myši a zásadně tak vylepšit UX aplikace. Jedná se například o zkratky pro uložení odhadu, přepnutí kategorie nebo funkci Zpět.

- **HTML export** – Aplikace bude podporovat export odhadu ve formě HTML souboru. Soubor bude obsahovat data jednotlivých kategorií odhadu, celkové přehledy a další informace, které je možné si pomocí uživatelského rozhraní zobrazit.

- **Lokalizace šablony** – V aplikaci je nyní lokalizováno jen čistě uživatelské rozhraní. Data ze šablony jako kategorie a checklist nikoliv. Bude proto nutné upravit model a funkce frontend části tak, aby bylo možné lokalizaci v šabloně podporovat. Zůstane zachována podpora pro češtinu a angličtinu.

- **Implementační šablona** – Bude přidána nová šablona. Ta se od současné bude lišit tím, že bude obsahovat pouze jednu kategorii Implementace a Ostatní, ve které se podle definovaných helper funkcí a jejich hodnot vyplní poměrově k Implementaci ostatní kategorie jako Analýza, Testování, ... Poměry jednotlivých kategorií budou vycházet na základě zkušeností a ve firmě a metrik, které uvádí ve své knize Steve McConnell[1, s. 233-237].

- **Quick Action Dialog** – Do Detailu odhadu bude přidán dialog rychlých akcí. V něm bude možné provést akce jako je například přepnutí do tabulky odhadu, změna kategorie nebo uložení odhadu. Akce, které v něm bude možné provést, budou odpovídat akcím, které budou namapované na klávesové zkratky. Jedná se o další vylepšení UX, přičemž tato funkcionalita je velmi populární ve vývojářských IDE nástrojích.

- **Komentáře přímo k položkám odhadu** – K jednotlivým položkám kategorií odhadu bude možné přidat poznámku stejně jako je tomu v Excel šabloně. Funkcionalita bude pravděpodobně řešena obdobně pomocí komentáře v tabulce, kterou tabulková komponenta podporuje.

- **Full-text search** – Vyhledávání nad daty získaných z API je nyní řešeno jednoduchým vyhledáváním poskytovaným přímo tabulkovou komponentou. Ta má definováno, jak má vyhledávání fungovat. Nově bude možné získat odhady i na základě hledaného slova přímo z API. Vyhledávací řetězec bude součástí dotazu na API. Fulltextového vyhledávání tak bude řešeno na backend části aplikace a frontend jím nebude zatížen.

8.3 Iterace 3

- **Verzování odhadu** – Nyní aplikace umožňuje mít jednu verzi odhadu čili nepodporuje verzování. Nemůžu se tak po nějakém čase vrátit v zpět a podívat se na změny, které jsem provedl, když jsem odhad například editoval. V rámci této iterace by proto mělo být přidáno verzování odhadu. Uživatel bude moci vytvořením verze uložit současnou podobu odhadu pod číslem verze. Po následně provedených změnách bude schopný si zobrazit data před změnou. Množství verzí nebude pravděpodobně limitováno. Před realizací verzování bude ale třeba nejdříve zanalyzovat a rozhodnout, jak se jednotlivé verze budou ukládat. Momentálně se nabízí dvě řešení. První by jako verzi ukládalo vždy celý odhad. Druhá možnost nabízí ukládat verzi pouze jako změny oproti předchozí verzi. Každé z řešení má jak své výhody tak i nevýhody.

- **Buňky podporují Markdown** – Díky teto funkcionalitě by mělo být možné formátovat částečně text v buňkách tabulky. Mělo by se jednat především o stylování typu tučný text, kurzíva či podtržení textu. Formátování bude dořešeno při samotné realizaci. Bude nutné nejdříve zjistit, do jaké míry je tato funkcionalita možná.

- **Další Quick Actions** – Zkratky a rychlé akce jsou definované pouze na stránce s detailem odhadu. Nově by měly být dostupné i na Dashboardu. Přibudou proto v rámci této funkcionality jak klávesové zkratky, tak jim odpovídající rychlé akce.

- **Email notifikace** – Při přidání práv pro zobrazení odhadu nebo práv pro editaci by se měla přidanému uživateli poslat emailová notifikace. Ta ho bude informovat o možnosti zobrazit nebo editovat odhad, na který mu byla přidána práva. Pokud tedy přidám uživateli právo pro zobrazení odhadu a odhad následně uložím. Vzápětí se odešle emailová notifikace autorovi a tomuto uživateli.

- **Možnost podbarvení buněk** – Bude přidána podpora pro podbarvování buněk řádků v tabulce s položkami odhadu. Uživatel si bude moci vybrat přes kontextové menu barvu pozadí příslušného řádku.

Závěr

Cílem práce bylo seznámit se se současným procesem a s nástroji pro tvorbu odhadů softwarových metrik v prostředí zadavatele a zaměřit se na použitelnost a využití současných nástrojů. Dále analyzovat nedostatky aktuálně používaných řešení a v rámci diplomové práce navrhnout a implementovat řešení nové, ve kterém budou eliminovány identifikované nedostatky. Mezi další cíle patřila eliminace slepých uliček při vývoji, díky kterým předešlé práce u uživatelů neuspěly.

V práci se podařilo splnit všechny vytyčené cíle. Seznámil jsem se se současnými řešeními a provedl jejich analýzu. Na základě v rámci analýzy zjištěných chyb jsem navrhl nové řešení. Tím je aplikace Estimate, který slouží pro tvorbu odhadů. Při návrhu a implementaci aplikace jsem vycházel mimo jiné z předem zjištěných UX problémů a ty v aplikaci eliminoval. Pro vývoj aplikace jsem se poučil z postupů v předešlých pracích. Rozhodl jsem se proto využít iterativní přístup s agilními prvky. Na jeho základě vznikla na začátku hrubá architektura aplikace, následně byl vývoj rozdělen do celkem čtyř iterací, které jsou v diplomové práci detailně popsány. Mnou zvolený iterativní postup se při tvorbě aplikace několikrát osvědčil, například při řešení licencí hlavní tabulkové komponenty. Současně během vývoje probíhalo uživatelské testování ve spolupráci s kolegy, kteří vytvářeli reálné odhady pro reálné projekty. Díky tomu mohla práce těžit z intenzivní a detailní zpětné vazby od reálných uživatelů. Implementační chyby a nedokonalosti v aplikaci, na které se během testování přišlo, byly průběžně v rámci iterací odstraněny. Zakončením celé této práce bylo nasazení aplikace do produkčního prostředí v interní sítí firmy Profinit. K datu publikace obsahuje aplikace 77 unikátních odhadů vytvořených 31 uživateli.

Na základě této diplomové práce a úspěchu vytvořené aplikace Estimate byl ve firmě vytvořen interní vývojový tým. Ten má na starosti další rozvoj aplikace, vývoj nástroje je do budoucna zajištěn. Jak je zmíněno v kapitole 8, jsou chystány další nápady na vylepšení, které ale nebylo možné v rozsahu diplomové práce obsáhnout. Mezi ně patří například verzování odhadů nebo

podpora Undo/Redo funkce. Zároveň se v rámci budoucího vývoje i nadále očekávají nápady na vylepšení a reporting chyb ze strany aktivních uživatelů. Protože se zvolený iterativní přístup vývoje nástroje v rámci diplomové práce osvědčil, hodlám v něm spolu s vývojovým týmem pokračovat i do budoucna.

Literatura

[1] McConnell, S.: *Software Estimation: Demystifying the Black Art*. Redmond: Microsoft Press, 2006, ISBN 0735605351.

[2] Brooks, F.: *The Mythical Man-Month: Essays on Software Engineering*. Boston: Addison-Wesley, 1995, ISBN 9780201835953.

[3] Vancl, M.: *Návrh datového modelu aplikace pro podporu tvorby odhadů pracnosti softwarových projektů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2017 [cit. 2020-12-29]. Dostupné z: `http://hdl.handle.net/10467/69580`

[4] Polačok, J.: *Aplikace pro podporu tvorby odhadů pracnosti softwarových projektů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2017 [cit. 2020-12-29]. Dostupné z: `http://hdl.handle.net/10467/69578`

[5] Štaffa, M.: *Implementace nástrojů pro zlepšení procesu SW vývoje*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2018 [cit. 2020-12-29]. Dostupné z: `http://hdl.handle.net/10467/76246`

[6] Shaffi, A.; Al-Obaidy, M.: Analysis and Comparative Study of Traditional and Web Information Systems Development Methodology (WISDM) Towards Web Development Applications. *International Journal of Emerging Technology and Advanced Engineering*, 2013.

[7] Spring Boot - Overview. Dostupné z: `https://spring.io/projects/spring-boot`

[8] Immutability. Dostupné z: `https://kotlinlang.org/docs/reference/native/immutability.html`

[9] What Is MongoDB? Dostupné z: https://www.mongodb.com/what-is-mongodb

[10] Google Trends. Dostupné z: https://trends.google.com/trends/explore?date=2016-01-012020-05-09&q=Reactjavascript,Vuejavascript,Angularjavascript

[11] Stack Overflow Trends. Dostupné z: https://insights.stackoverflow.com/trends?tags=reactjs,angular,vue.js

[12] Gitlab. Dostupné z: https://gitlab.com/gitlab-org/gitlab

[13] What is Markdown? Dostupné z: https://www.markdownguide.org/getting-started/

[14] Handsontable. Dostupné z: https://handsontable.com/

[15] Create a New React App. Dostupné z: https://reactjs.org/docs/create-a-new-react-app.html

[16] Introduction Bootstrap. Dostupné z: https://getbootstrap.com/docs/4.5/getting-started/introduction/

[17] Redux Essentials. Dostupné z: https://redux.js.org/tutorials/essentials/part-1-overview-concepts

[18] Continuous Integration. Dostupné z: https://www.thoughtworks.com/continuous-integration

[19] Why Docker? Dostupné z: https://www.docker.com/why-docker

[20] react-intl - npm. Dostupné z: https://www.npmjs.com/package/react-intl

[21] react-i18next - npm. Dostupné z: https://www.npmjs.com/package/react-i18next

[22] react-mosaic. Dostupné z: https://github.com/nomcopter/react-mosaic

[23] Spring Profiles. Dostupné z: https://docs.spring.io/spring-boot/docs/1.2.0.M1/reference/html/boot-features-profiles.html

[24] Collection methods – MongoDB Manual. Dostupné z: https://docs.mongodb.com/manual/reference/method/js-collection/

[25] eval(). Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/evalc

[26] MATERIAL-UI. Dostupné z: https://material-ui.com

[27] MSAL. Dostupné z: https://docs.microsoft.com/cs-cz/azure/active-directory/develop/msal-overview

[28] OAuth 2.0. Dostupné z: https://oauth.net/2/

[29] Azure AD Spring Boot Starter client library for Java. Dostupné z: https://github.com/Azure/azure-sdk-for-java/tree/master/sdk/spring/azure-spring-boot-starter-active-directory

[30] react-aad. Dostupné z: https://github.com/syncweek-react-aad/react-aad

[31] Application types for Microsoft identity platform. Dostupné z: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-app-types

[32] material-table. Dostupné z: https://material-table.com/

[33] Gitlab Releases. Dostupné z: https://docs.gitlab.com/ee/user/project/releases/index.html

Příloha A

Seznam použitých zkratek

API Application Programming Interface

Azure AD Azure Active Directory

CD Continuous Delivery

CI Continuous Integration

DAO Data Access Object

GUI Graphical user interface

JSON JavaScript Object Notation

MD Man-day (člověkoden)

MH Man-hour (člověkohodina)

MM Man-month (člověkoměsíc)

MR Merge Request

MSAL Microsoft Authentication Library

PM Project Management

PoC Proof of Concept

REST Representational State Transfer

UI User Interface

UX User Experience

XML Extensible Markup Language

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `impl` ..... zdrojové kódy aplikace Estimate
- `text` ..... text práce
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `DP_Pansky_Petr_2021.pdf` ..... text práce ve formátu PDF