Univerzita Jana Evangelisty Purkyně

v Ústí nad Labem

Přírodovědecká fakulta

Software pro kategorizovanou evidenci osobních výdajů s využitím OCR a AI

BAKALÁŘSKÁ PRÁCE

Vypracovala: BcA. Vlasta Michalcová

Vedoucí práce: Mgr. Jiří Fišer, Ph.D.

Studijní program: Aplikovaná informatika

ÚSTÍ NAD LABEM 2024

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně a použila jen pramenů, které cituji a uvádím v přiloženém seznamu literatury.

Byla jsem seznámena s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona c. 121/2000 Sb., ve znění zákona c. 81/2005 Sb., autorský zákon, zejména se skutečností, že Univerzita Jana Evangelisty Purkyně v Ústí nad Labem má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Jana Evangelisty Purkyně v Ústí nad Labem oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladu, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

V Ústí nad Labem dne 19. dubna 2024 Podpis: . . . . . .

Ráda bych poděkovala panu Mgr. Jiřímu Fišerovi, PhD. za vedení bakalářské práce. Za přečtení a zpětnou vazbu, jakož i za spolupráci při studiu, děkuji Michalu Melicherovi. V neposlední řadě děkuji svému manželovi Pavlovi, který mi pomáhal s Javou a nasazováním projektu, diskutoval se mnou všechny byznysové i implementační nápady, mnohé mě naučil, a především se po dobu mého bakalářského studia vedle mě nezbláznil.

SOFTWARE PRO KATEGORIZOVANOU EVIDENCI OSOBNÍCH VÝDAJŮ S VYUŽITÍM OCR A AI

Abstrakt:

Tato bakalářská práce se věnuje vývoji softwaru pro kategorizovanou evidenci osobních výdajů s využitím technologií OCR (*Optical Character Recognition*) a umělé inteligence. Cílem práce je poskytnout uživatelům jednoduchý a efektivní nástroj, který umožní automatizované zpracování účtenek a faktur pro lepší správu osobních financí. V práci je nejprve provedena analýza současných technologií OCR a AI a přehled existujících řešení na trhu. Následně je představen návrh a implementace softwaru včetně architektury aplikace a databáze. Důraz je kladen na využití moderních technologií a nástrojů, jako jsou React Native, Java Spring Boot a PostgreSQL, a také na analýzu možností využití těchto technologií v kontextu evidování osobních výdajů. Závěrem práce jsou shrnuty dosažené výsledky a možnosti dalšího rozvoje projektu.

Klíčová slova: OCR, AI, osobní finance, React Native, Java Spring Boot, PostgreSQL

SOFTWARE FOR CATEGORIZED RECORDING OF PERSONAL EXPENSES USING OCR AND AI

Abstract:

This bachelor thesis focuses on the development of software for the categorized recording of personal expenses using Optical Character Recognition (OCR) and Artificial Intelligence technologies. The goal of the thesis is to provide users with a simple and efficient tool that enables the automated processing of receipts and invoices for better personal finance management. The work begins with an analysis of current OCR and AI technologies and an overview of existing solutions in the market. Subsequently, the design and implementation of the software are presented, including the application architecture and database. Emphasis is placed on the use of modern technologies and tools, such as React Native, Java Spring Boot, and PostgreSQL, as well as on analyzing the possibilities of using these technologies in the context of recording personal expenses. The thesis concludes with a summary of the results achieved and the possibilities for further development of the project.

Keywords: OCR, AI, personal finance, React Native, Java Spring Boot, PostgreSQL

Obsah

Úvod

Propojení technologií OCR[1] a umělé inteligence[2] ve sféře komerčních aplikací nabízí způsob, jak značně zjednodušit a automatizovat mnoho každodenních procesů, jako je vedení osobní evidence výdajů pro mnoho domácností[3]. OCR umožňuje rychlé a přesné digitalizování tištěného nebo psaného textu z fyzických dokumentů, jako jsou účtenky a faktury, zatímco AI dokáže tato data analyzovat a kategorizovat. Společné využití těchto technologií může výrazně snížit čas strávený správou osobních financí a zároveň zvýšit přesnost a užitečnost získaných informací.

V této bakalářské práci se zaměřím na prozkoumání možností, jak tyto pokročilé technologie mohou být aplikovány na vývoj softwaru pro kategorizovanou evidenci osobních výdajů. Cílem je nejen představit teoretický rámec a technické aspekty implementace, ale také nabídnout praktické řešení, které by mohlo vyplnit existující mezeru nejen na českém trhu. Vyvíjený software by měl poskytnout uživatelům jednoduchý a efektivní nástroj pro správu jejich osobních financí, využívající nejnovější dostupné technologie v oblasti OCR a AI[4]. Aplikace si klade za cíl mít následující funkce:

- Umožnit uživateli pohodlně nahrát účtenku
- Zpracovat tuto účtenku a kategorizovat položky na ní
- Umožnit uživateli upravit zpracovanou účtenku (změnit kategorii, datum atd.)
- Poskytnout uživateli přehled o jeho výdajích na základě kategorií

Práce je strukturována do sedmi kapitol. V první kapitole jsou představeny současné technologie OCR a AI a ve druhé jsou shrnuta existující řešení pro evidenci osobních výdajů. Třetí kapitola slouží k představení použitých technologií a nástrojů pro vývoj software. Následuje analýza možností využití a vyhodnocení požadavků na aplikace ve čtvrté kapitole a návrh architektury aplikace a databáze v páté kapitole. V šesté kapitole popisuji chronologický postup implementace

[1] *Optical Character Recognition* (optické rozpoznávání znaků).
[2] V této práci synonymně používám také běžnou zkratku AI (*Artificial Intelligence*), konkrétněji ale v aplikaci využívám LLM (*Large Language Model*).
[3] V oblasti financí došlo za posledních 50 let k velkému vývoji, nicméně základní principy, jako je kontrola rozpočtu, spoření a evidence příjmů a výdajů, patří stále k nejdůležitějším technikám (Gallo, 2023).
[4] Nejnovější je bráno k říjnu 2023, kdy byla tato bakalářská práce zadána a vývoj aplikace zahájen.

a v poslední kapitole prezentuji výsledný software ke konci první iterace. V závěru práce jsou zhodnoceny dosažené výsledky a diskutovány možnosti dalšího vývoje aplikace.

1. Analýza současných technologií OCR a AI

Technologie OCR a AI se v posledních letech dočkaly především díky zdokonalení LLM mnoha vývojových přelomů, a tak není divu, že i jejich využití v softwarech všeho druhu narůstá.

1.1. OCR (*Optical Character Recognition*)

Optické rozpoznávání znaků se používá k převodu různých typů dokumentů, včetně skenovaných papírových dokumentů, PDF souborů nebo digitálních fotografií, na upravitelná a prohledávatelná (indexovatelná) data. Hlavním cílem OCR je převést obraz textu na strojově čitelný text, aby bylo možné dokumenty elektronicky editovat, vyhledávat a efektivněji uchovávat (Christensson, 2018).

Historie technologie OCR sahá až do 20. let minulého století, kdy Emanuel Goldberg získal patent na tzv. statistický stroj, přístroj schopný vyhledávat v dokumentech s využitím světelných paprsků a fotočlánků (Buckland, 2006). První skutečné OCR stroje se začaly objevovat v 50. letech 20. století, kdy exponenciálně narůstalo množství dat a také investice do technologického sektoru. Tyto první stroje na optické rozpoznání znaků našly své využití převážně v bankách, pojišťovnách nebo dalších velkých společnostech, které se rozhodly investovat do automatizace zpracovávání dat. V 60. letech začala do rozpoznávání vzorů investovat MIT[5], nicméně nejdůležitější pokroky ve výzkumu OCR se odehrály až po spojení této technologie se strojovým učením.[6] Čím pokročilejší byly algoritmy strojového učení, tím se zlepšovala i přesnost OCR. V 90. letech se technologie OCR začala objevovat u velkých komerčních softwarů, např. Adobe začalo umožňovat konvertování naskenovaných dokumentů na upravitelné textové soubory (Tripathi, 2023).

Mezi průlomové projekty patří jednoznačně *Tesseract OCR*, původně vyvíjený jako open source projekt pod záštitou Google (Tesseract OCR, 2024). V roce 2005 díky propojení pokročilých algoritmů strojového učení s počítačovým viděním[7] bylo dosaženo přesnosti rozpoznání znaků až

[5] Massachusetts Institute of Technology
[6] Ať už mluvíme o ICR (*Intelligent Character Recognition*) nebo MICR (*Magnetic Ink Character Recognition*).
[7] Častější je pojem *Computer Vision* (Farley, Mehrotra, & Urban, 2023).

63 % (oproti původním 38 %). Postupný komunitní vývoj dokázal přesnost OCR zlepšit až na necelých 72 %.

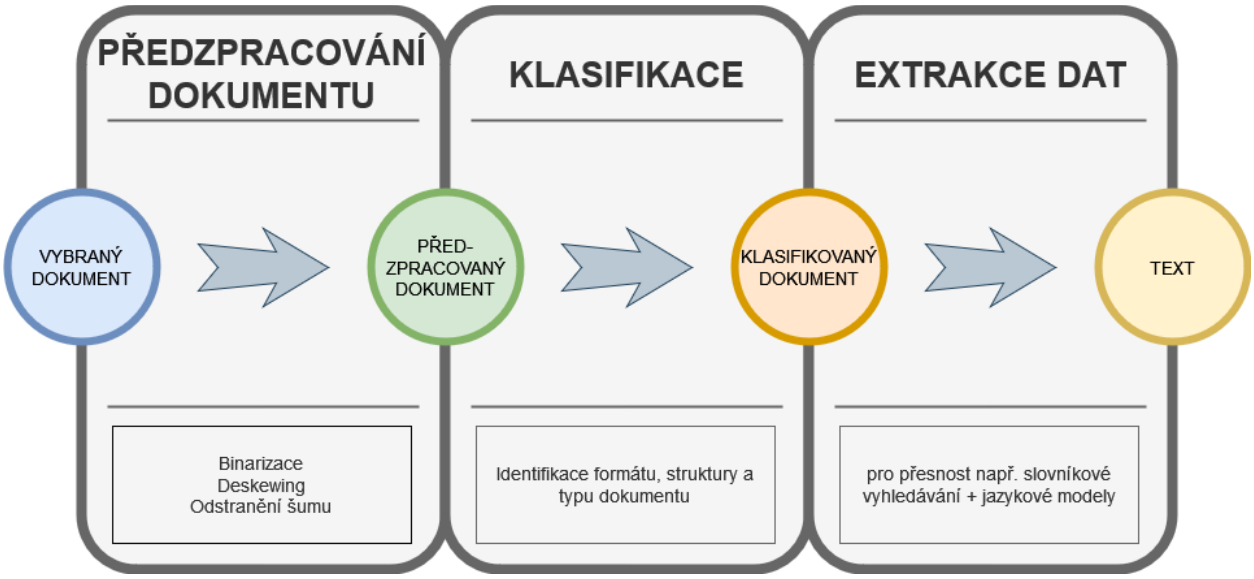
Z dnešního hlediska však mezi největší milníky patří využití neuronových sítí[8] a hlubokého učení (Goodfellow, Bengio, & Courville, 2016), díky čemuž lze dosáhnout až 99% přesnosti rozpoznání znaků. Kromě zdokonalení přesnosti rozpoznání znaků na čitelných textech je dnes možné s vysokou přesností rozpoznávat širokou škálu písem a jazyků, a to i v náročných podmínkách, jako jsou rozmazané nebo zkreslené texty. Díky dostupnosti platforem na inteligentní zpracování dokumentů nemusí firmy investovat velké částky do počátečních nákladů na využití OCR, takže proces digitalizace pokračuje (Farley, Mehrotra, & Urban, 2023).

Právě díky vysoké přesnosti technologie a její dostupnosti jsem se v rámci vývoje software pro kategorizovanou evidenci výdajů rozhodla využít *Document Intelligence* od Azure Cognitive Services (Urban & další, 2024). Postup převedení účtenky z tištěné do digitální podoby je následující:

- **Předzpracování dokumentu**: Zde dochází k extrakci dat pomocí OCR. Dokument projde první fází předzpracování, která zahrnuje techniky jako binarizaci (převod barevného obrazu na černobílé pixely) pro zřetelné oddělení textu od pozadí, deskewing (korekci naklonění skenovaného obrazu) a odstranění šumu pro eliminaci nežádoucích teček či skvrn, které by mohly OCR zaměnit za znaky[9].
- **Klasifikace dokumentů**: Tento krok zahrnuje identifikaci formátu souboru[10], struktury dokumentu (pevně strukturované, polostrukturované nebo nestrukturované) a typu dokumentu (např. v našem případě účtenka), což je základ pro efektivní extrakci dat.
- **Extrakce dat**: Klíčová část, kde dochází k přeměně obrazového záznamu účtenky na strukturovaná data. OCR slouží jako první krok extrakce dat, ale může chybovat při detekci

[8] Převážně konvolučních (CNN) a rekurentních (RNN).
[9] Již v tomto kroku je technologie od Azure velice přesná, a této přesnosti se mi s knihovnou Tesseract během výzkumu dosáhnout nepodařilo.
[10] Model od Azure Document Intelligence podporuje různé druhy účtenek (většinou se liší podle typu výdaje, např. Účtenka z hotelu, parkovacího automatu nebo z lékárny může vypadat jinak) a nahrávat lze v 10 podporovaných formátech, mezi které patří samozřejmě PDF, JPG, PNG nebo soubory Microsoft Office.

slov nebo selhat v rozpoznání znaků. Pro zvýšení přesnosti extrakce dat se využívají pokročilé techniky, jako jsou slovníkové vyhledávání a jazykové modely, které pomáhají opravit chybně rozpoznané termíny nebo doplnit chybějící kontext. Kromě toho, techniky strojového učení a extrakce založené na pravidlech umožňují systému flexibilně identifikovat a extrahovat klíčové informace, jako jsou názvy položek, ceny, data a časové údaje, a to nezávisle na jejich umístění v dokumentu. Tyto metody také umožňují rozpoznání vzorů a struktur v datech, což je klíčové pro kategorizaci a další specifické analýzy výdajů[11].



1.2. AI a strojové učení pro kategorizaci dat

Strojové učení je základem nejen pro technologii OCR, ale také pro umělou inteligenci. *Intelligent Document Processing* technologie totiž využívají stejné algoritmy jako umělá inteligence ke zlepšení míry přesnosti rozpoznání znaků (např. na základě rozpoznání kontextu) nebo při validaci dat.

Síla generativní umělé inteligence byla veřejnosti představena na konci roku 2022 prostřednictvím jazykového modelu ChatGPT od společnosti OpenAI. V rámci vývoje software pro kategorizovanou evidenci osobních výdajů využívám umělou inteligenci[12] právě pro

[11] Všechna data, která Receipt Model extrahuje, lze nalézt v této tabulce. (Urban & další, 2024)
[12] Konkrétně model GPT-3.5 Turbo (OpenAI, 2024)

automatizovanou kategorizaci. Úkolem AI je zařazení textových dat do předem definovaných kategorií na základě jejich obsahu. Klíčové koncepty a techniky, které jsou relevantní pro kategorizaci textů, zahrnují:

- **Předzpracování textu**: Zahrnuje odstranění nežádoucích znaků, normalizaci textu, tokenizaci (rozdělení textu na jednotlivá slova nebo věty), stemming (redukce slov na jejich kořenovou formu), lemmatizaci (převedení slov na jejich základní slovní formu) a POS tagging (přiřazování slovních druhů). Tento krok je zásadní pro snížení složitosti dat a zvýšení účinnosti algoritmů ML.[13] (Aydin, 2023)
- **Vektorizace textu**: Převod textu na numerické vektory pomocí technik jako je *Bag of Words*, TF-IDF (*Term Frequency-Inverse Document Frequency*), nebo embeddings (vložení)[14], což umožňuje algoritmům ML pracovat s textovými daty (Clu & další, 2021).
- **Trénování modelu**: Využití trénovacích dat k naučení modelu rozpoznávat vzory a vztahy mezi textovými daty a jejich kategoriemi. To může zahrnovat učení s dohledem (*supervised*), kde modelu poskytneme příklady s označenými kategoriemi, nebo učení bez dohledu (*unsupervised*), které hledá skryté struktury v datech bez předem definovaných kategorií.[15]
- **Evaluace a optimalizace**: Hodnocení výkonu modelu na testovacích datech a jeho optimalizace pomocí různých technik a hyperparametrů pro dosažení nejlepších možných výsledků. Mezi nejznámější techniky patří matice záměn (*confusion matrix*), *F1 Score* (harmonický průměr *precision*[16] a *recall*[17]) nebo ROC AUC[18].

---

[13] V našem případě předáváme AI dvě složky: CSV soubor obsahující id položky, název položky a prázdný sloupec pro doplnění kategorie položky a další soubor obsahující předem definovaný seznam kategorií/podkategorií. Oba texty jsou redukované na relevantní informace, což činí výstup AI přesnějším.

[14] Je velice pravděpodobné, že Azure Document Intelligence nepoužívá základní techniky jako *Bag of Words* nebo TF-IDF, ale má vlastní implementace hustých vektorových reprezentací, tzv. embeddings, které mohou zachycovat bohatší sémantické informace o slovech nebo frázích v kontextu. Konkrétní informace implementace vektorizace však nejsou veřejně dostupné.

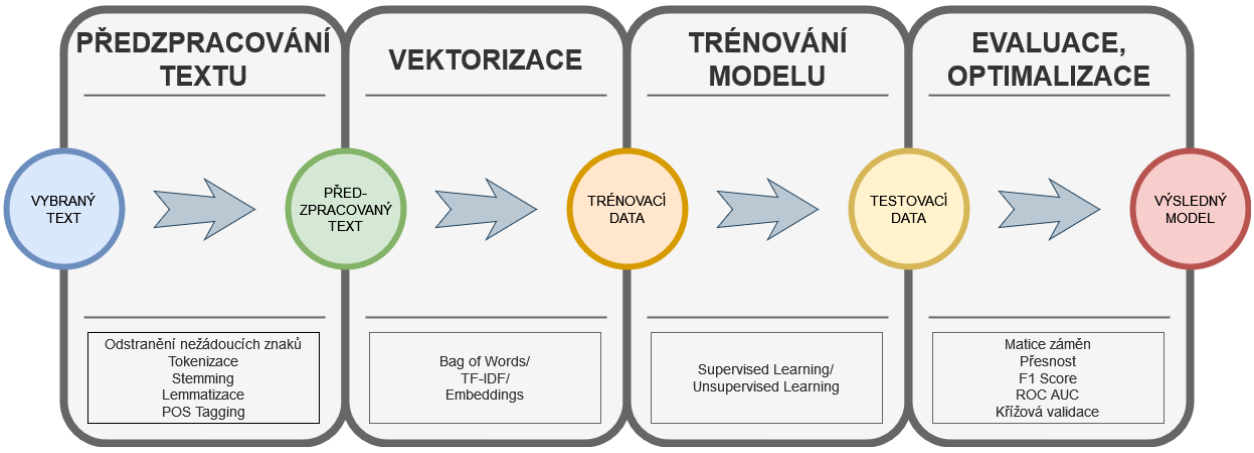
[15] AI je zde dán „návod“, jak položky kategorizovat, přímo v promptu.

[16] Podíl správně identifikovaných pozitivních výsledků z celkového počtu pozitivních výsledků. Nezaměňovat s přesností (*accuracy*), což je podíl správně klasifikovaných případů ze všech případů.

[17] Podíl správně identifikovaných pozitivních výsledků z celkového počtu skutečných pozitivních výsledků, také označován jako senzitivita.

[18] Křivka *Receiver Operating Characteristic* a *Area Under the Curve* hodnotí, jak dobře model rozlišuje mezi třídami. Čím vyšší AUC, tím lepší výkon modelu.

Proces vytváření modelu strojového učení pro zpracování přirozeného jazyka (NLP) je zjednodušeně znázorněn na tomto diagramu:



2. Přehled existujících řešení pro evidenci osobních výdajů

Trh s aplikacemi pro evidenci osobních výdajů je rozmanitý a nabízí řadu řešení, která usnadňují správu osobních financí. Po koronavirové pandemii vzrostla preference uživatelů vyřizovat finanční záležitosti online, což jen posílilo vývoj funkcionalit nabízených aplikací. V současnosti zde hraje velkou roli využití generativní AI a různých chatbotů (Miles, 2020). Většina dostupných aplikací však primárně funguje na principu sledování karetních transakcí, což zahrnuje propojení aplikace s bankovními účty uživatelů[19]. Tato funkcionalita umožňuje automatickou kategorizaci a evidenci výdajů, avšak s určitými omezeními:

- **Nedostatek podpory pro papírové účtenky**: Mnohé aplikace nezahrnují možnost zaznamenávat transakce v hotovosti, které jsou stále běžné u určitých typů výdajů (typicky restaurace, ale např. i parkovací automat na mince). Celkový přehled o výdajích tudíž není úplný.
- **Kategorizace na základě obchodu, nikoliv položek**: Ačkoli je automatická kategorizace výdajů podle obchodu pohodlná, nezohledňuje skutečnost, že mnoho obchodů, jako je např. Hypermarket Globus, nabízí širokou škálu produktů z různých kategorií. Tímto způsobem může být zaznamenávání výdajů nepřesné a méně užitečné pro detailní analýzu spotřebního chování uživatele.
- **Bankovní aplikace a jejich omezení**: Bankovní aplikace obvykle kategorizují transakce pouze podle obchodu a zahrnují jen ty platby, které proběhly přes kartu od této banky, což omezuje uživatelům možnost získat komplexní přehled o všech svých výdajích a svým způsobem tak uživatele uvazuje k setrvání u dané banky, neboť by v případě přechodu k jiné bance o svá data přišli.
- **Nedostatek detailní kategorizace v nových aplikacích**: Některé novější aplikace nabízejí možnost nahrávání účtenek pro evidenci transakcí v hotovosti, avšak často neposkytují funkci automatické kategorizace jednotlivých položek na účtence, případně neumožňují některé položky z účtenky do evidence nezahrnout.

[19] Populárními aplikacemi dle Forbes jsou *Wallet* či *Spendee*, které umožňují evidenci příjmů i výdajů, propojení s bankou a automatické synchronizovaní a třídění do základních kategorií (Basiková, 2022). Tyto aplikace, fungující jako přehledná peněženka, však výdaje kategorizují podle provedených plateb, nikoli podle jednotlivých položek. Mají navíc mnoho funkcí a uživatel se může cítit zahlcen a zmaten. Další aplikace zmiňuje např. článek na webu Fakturoid (Fakturoid, 2024).

Z těchto pozorování vyplývá, že existuje značná mezera na trhu pro aplikaci, která by:

- **Podporovala jak digitální, tak papírové účtenky**: Aplikace, která by umožňovala uživatelům snadno digitalizovat a zahrnout výdaje z papírových účtenek do celkového přehledu o jejich výdajích.
- **Kategorizovala výdaje podle položek, ne obchodů**: Tím by poskytla přesnější a užitečnější informace o spotřebním chování a umožnila detailní sledování výdajů v různých kategoriích, které si uživatel může přizpůsobit na míru svým potřebám.
- **Byla dostupná na českém trhu**: S ohledem na lokální potřeby a specifika, včetně podpory českého jazyka a rozpoznávání lokálních měn a formátů účtenek[20]. Zde je také potenciál pro rozšíření na jiných malých trzích (menší evropské státy).
- **Nabízela možnost porovnání cen**: Jako dlouhodobější vizi by aplikace mohla poskytovat funkcionalitu pro sledování cen konkrétních produktů v různých obchodech, čímž by uživatelům usnadnila najít nejlepší nabídky bez nutnosti procházet mnoho letáků, případně by bylo možné jednotlivé produkty porovnávat mezi sebou ještě před samotným nákupem, a to například načtením čárového kódu v obchodě a automatickým porovnáním s ceníkem jiných obchodů.

Takové řešení by zaplnilo stávající mezeru na trhu tím, že by poskytovalo komplexní a uživatelsky přívětivou platformu pro evidenci a analýzu osobních výdajů, zahrnující všechny typy transakcí a nabízející hlubší vhled do finančního chování uživatelů. Právě na vývoji takové aplikace, kterou pracovně nazývám **CheckChecker** (**Účtenkovač**), jsem v rámci bakalářského studia pracovala.

[20] Tento požadavek například aplikace *Spendee* splňuje, avšak nesplňuje 1 či více dalších zmíněných požadavků.

3. Představení použitých technologií a nástrojů

3.1. React Native

React Native je open-source framework pro vývoj mobilních aplikací, který byl vytvořen a je podporován společností Facebook (nyní Meta Platforms, Inc.). Umožňuje vývojářům psát aplikace pro operační systémy iOS a Android použitím JavaScriptu a Reactu, což je populární javascriptová knihovna pro tvorbu uživatelských rozhraní. (React Native, 2023)

Při rozhodování, zda při vývoji frontendu aplikace použít právě React Native, pro mě byly stěžejní tyto důvody:

1) Technologie mi již byla známá, a tak byl vývoj poměrně plynulý, což by neplatilo při zvolení např. Kotlinu, který je pro vývoj mobilních aplikací pro Android typičtější volbou, nicméně já se s ním doposud nesetkala.
2) React Native umožňuje využívat nativní komponenty jak pro Android, tak iOS[21], takže je možné naráz vyvinout uživatelsky přívětivou aplikaci pro oba operační systémy.
3) Díky React Native je možné sdílet kód mezi mobilní a webovou platformou, tudíž pokud by software někdy vyžadoval webové rozhraní, dala by se část stávající implementace převzít a zjednodušilo by to implementaci webové aplikace.

3.2. Java Spring Boot

Java[22] spolu s frameworkem Spring Boot tvoří dvojici, která je v oblasti vývoje softwaru osvědčená a vysoce ceněná. Díky desetiletím využívání v komerční sféře, ať už v rozsáhlých korporacích nebo malých startupech, představují Java a Spring Boot stabilní základ, který nabízí široké spektrum ověřených knihoven a nástrojů (IBM, 2023). Tyto technologie jsou známé svou robustností, bezpečností a škálovatelností, což je činí ideální volbou pro vývoj backendové části

[21] *„In Android development, you write views in Kotlin or Java; in iOS development, you use Swift or Objective-C. With React Native, you can invoke these views with JavaScript using React components. At runtime, React Native creates the corresponding Android and iOS views for those components. Because React Native components are backed by the same views as Android and iOS, React Native apps look, feel, and perform like any other apps. We call these platform-backed components Native Components.* (React Native, 2023)

[22] Konkrétně používám verzi Java 17, která byla nejnovější LTS v době začátku vývoje.

aplikací, včetně těch, které vyžadují spolehlivou a efektivní správu dat, jako je aplikace pro kategorizovanou evidenci osobních výdajů.

Spring Boot, který je součástí ekosystému Spring, dále zjednodušuje vývoj aplikací tím, že automatizuje konfiguraci a nasazení, umožňuje vývojářům se soustředit na logiku aplikace namísto rutinního nastavování prostředí (Spring, 2024). Integrace s nástroji jako Liquibase pro správu verzí databáze a migrace dat a JOOQ pro vytváření staticky typovaných SQL dotazů v Javě poskytuje vývojářům silné nástroje pro efektivní manipulaci s databází a zajištění konzistence dat při aktualizacích. Tento přístup umožňuje rychlé iterace a snadné adaptace na změny v požadavcích aplikace, což je zásadní pro dynamický vývoj softwaru.

Podpora modelu MVC (*Model-View-Controller*) ve Springu navíc přináší osvědčenou architekturu pro vývoj webových aplikací, která odděluje logiku aplikace od uživatelského rozhraní. Toto rozdělení umožňuje lepší organizaci kódu, usnadňuje testování a podporuje modularitu, což vede k vyšší kvalitě softwaru a zjednodušuje spolupráci v týmu. S takto pevným a flexibilním základem mohou vývojáři efektivně vytvářet a udržovat komplexní aplikace, jako je systém pro kategorizovanou evidenci osobních výdajů, a současně zůstat agilní a připraveni na budoucí rozšíření nebo změny[23].

3.3. PostgreSQL

Při výběru databázového systému pro mou software pro evidenci osobních výdajů jsem se rozhodla využít PostgreSQL, což bylo rozhodnutí založené na několika faktorech, které odpovídají potřebám a cílům projektu a jsou klíčové pro podporu růstu aplikace a zvládání případných nárůstů počtu uživatelů nebo dat bez kompromisů v oblasti výkonu (The PostgreSQL Global Development Group, 2023).

Ve srovnání s MySQL mě také vedly k výběru PostgreSQL mé osobní zkušenosti a znalosti tohoto systému, což znamená, že mohu efektivněji využívat pokročilé funkce, rychleji řešit případné problémy a optimalizovat databázi pro potřeby aplikace. Navíc, PostgreSQL je považován za *Enterprise Grade* technologii, která je přesto k dispozici zdarma. Tento aspekt činí PostgreSQL

[23] V této části se opírám o konzultaci se Senior Java Developerem.

atraktivní volbou pro projekty všech velikostí, včetně startupů a nezávislých vývojářů, kteří hledají robustní, spolehlivou a zároveň cenově dostupnou databázovou platformu.

4. Analýza možností využití

Na počátku byl nápad pramenící z osobní touhy po takové aplikaci, která by uživateli jednoduše a systematicky umožnila evidovat výdaje, bez nutnosti zadávat položku po položce do Excelu nebo se smířit s tím, že uvidí pouze čísla, ale nikoliv za co sumy utratil (grafy v online bankovnictví).

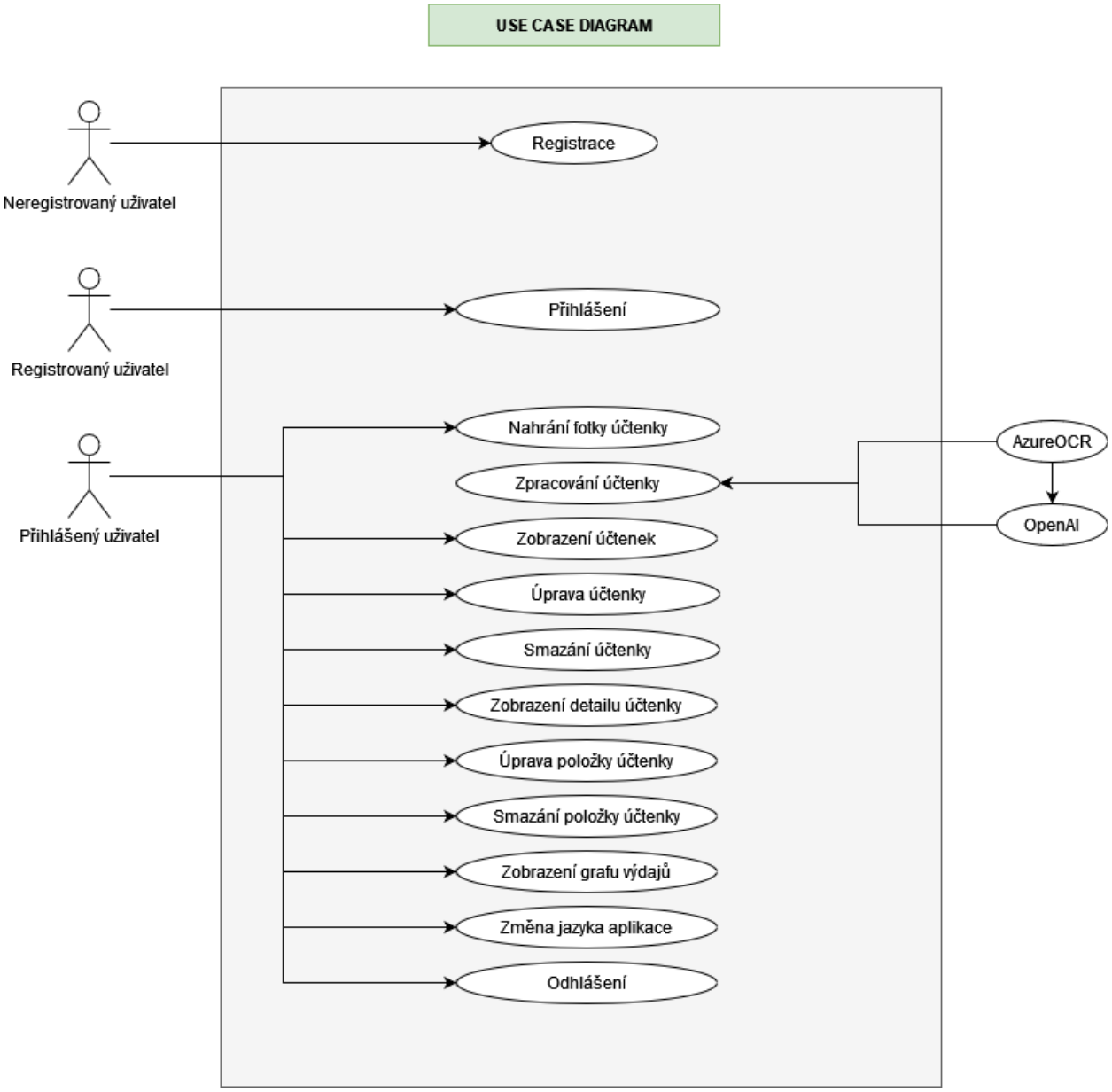
Před samotným návrhem architektury a vývojem aplikace jsem však rozeslala dotazník, který mi pomohl rozšířit a analyzovat potenciální uživatelské požadavky na aplikaci[24].

Respondenti[25], kteří si evidují nějakým způsobem své výdaje, uvedli, že by kategorizovanou evidenci uvítali (75 %). Dotazník dále zkoumal, jaké základní kategorie by uživatelé používali[26], jaký způsob přidávání účtenky by pro ně byl nejpohodlnější[27], jaký formát by zvolili pro export dat[28], jakým způsobem by se do aplikace přihlásili[29] a jaký operační systém využívají[30].

Výsledky dotazníku byly zohledněny při definici funkčních specifikací aplikace.

Následovalo vytvoření *use case* diagramu, který znázorňuje případy užití software. Jsou zde zastoupeny hlavní aktivity uživatelů a ukázka, kdy do aplikace vstupují služby od třetích stran [31].

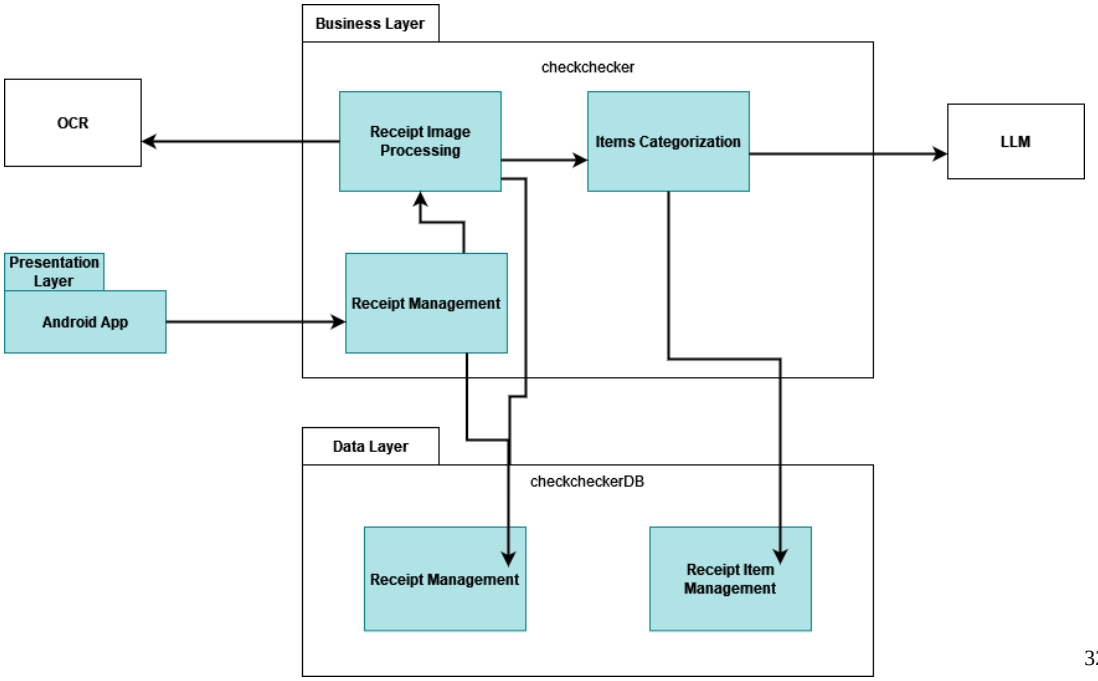
[24] Výsledky dotazníku lze nalézt na Githubu.
[25] 75 % respondentů byly ženy, věk respondentů se pohybuje od 24 do 51 let, většina jich pochází z České republiky a mezi uvedenými zaměstnáními jsou např. živnostník, *IT specialist*, učitelka, projektový manažer nebo *Talent Acquisition Specialist*.
[26] Zde převažují kategorie jídlo, doprava, oblečení drogerie, cestování, alkohol a předplatné.
[27] Vede vyfocení účtenky přímo v aplikaci, ale zájem byl i o možnost nahrání souboru, zaslání přes email nebo ruční zadání.
[28] Výsledné umístění v tomto pořadí: PDF, XLSX, CSV.
[29] 50 % respondentů preferuje možnost přihlášení přes Google účet.
[30] Android 75 %, iOS 25 %.
[31] Dostupné na Githubu.


USE CASE DIAGRAM
Neregistrovaný uživatel
Registrace
Registrovaný uživatel
Přihlášení
Přihlášený uživatel
Nahrání fotky účtenky
Zpracování účtenky
Zobrazení účtenek
Úprava účtenky
Smazání účtenky
Zobrazení detailu účtenky
Úprava položky účtenky
Smazání položky účtenky
Zobrazení grafu výdajů
Změna jazyka aplikace
Odhlášení
AzureOCR
OpenAI

5. Návrh architektury aplikace a databáze

Při návrhu architektury aplikace jsem primárně vycházela ze specifikací vyplývajících z *use case* diagramu. Bylo pro mě však důležité ponechat dostatek prostoru pro budoucí implementace (např. díky implementaci rozhraní lze v budoucnu uvažovat nad využitím jiné služby nebo implementací vlastního řešení OCR a AI kategorizace). Návrh architektury aplikace a struktury databáze se řídí principem *database first*, což znamená, že jsem nejprve navrhla kompletní databázi a až poté přistoupila k vývoji samotné aplikace.

Z následujícího *package* diagramu je zřejmé, že aplikace se skládá ze tří hlavních vrstev: prezentační, datové a byznysové. Prezentační vrstvu tvoří Android aplikace, která slouží jako uživatelské rozhraní pro interakci s aplikací. Datová vrstva obsahuje databázi *checkcheckerDB*, která spravuje (mimo jiné) údaje o účtenkách a položkách v nich. Business vrstva (backend) zahrnuje logiku aplikace, např. zprocesování obrázku účtenky pomocí externí OCR služby, a automatické třídění položek do předdefinovaných kategorií pomocí externího jazykového modelu. Interakce mezi komponentami, jak je ilustrováno v diagramu, zajišťuje plynulý tok informací a efektivní fungování celé aplikace.



[32]

[32] Dostupné na Githubu.

5.1. Databáze

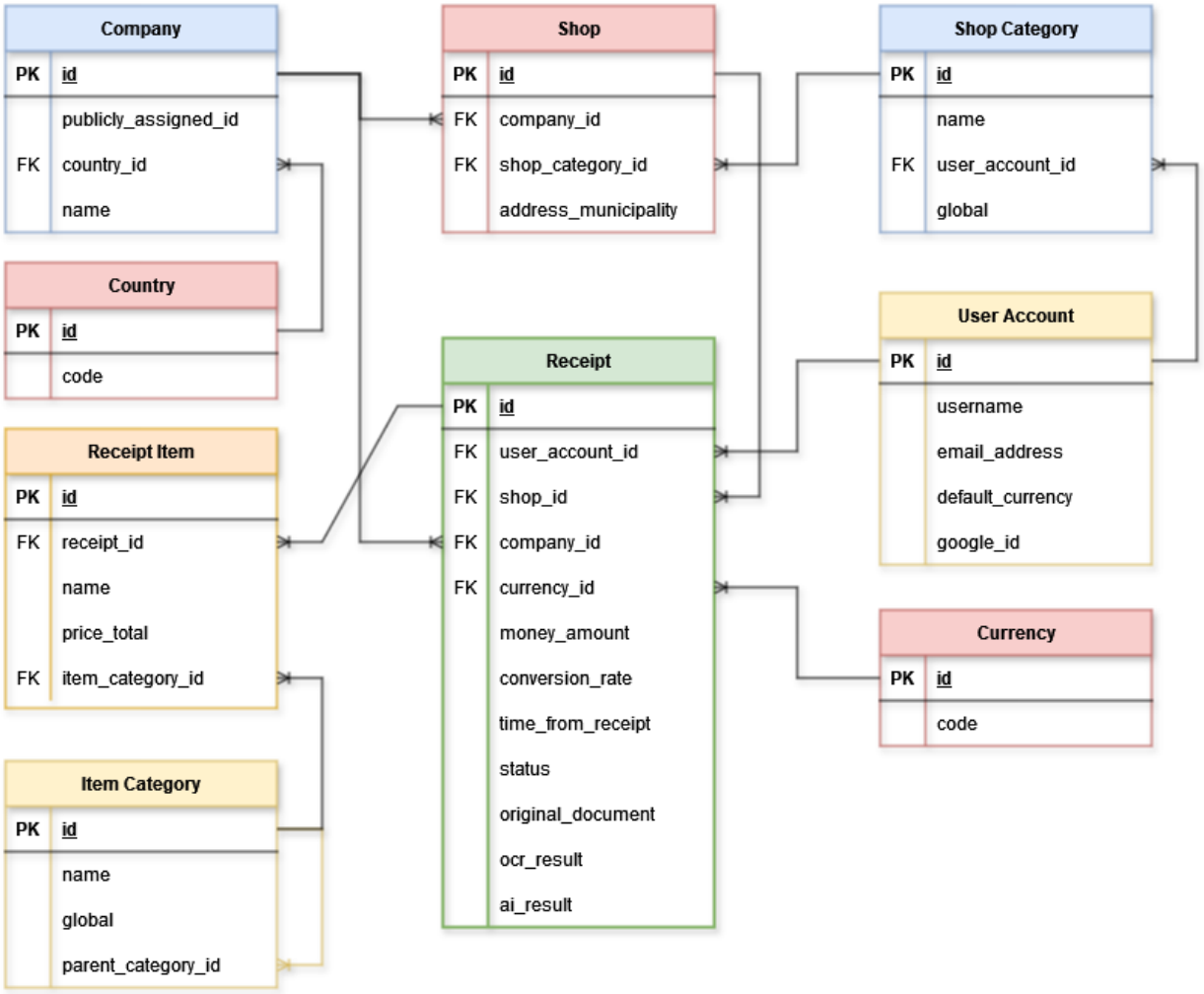
V návrhu databáze pro mou aplikaci jsem se rozhodla zaměřit na přímou manipulaci s databázovými tabulkami místo použití Java Persistence API (JPA). Tento krok nebyl o volbě mezi dvěma vzájemně se vylučujícími metodami, ale spíše o preferenci jednoho přístupu nad druhým v závislosti na specifických potřebách projektu. Zatímco JPA nabízí abstrakci datových modelů ve formě entit, což může usnadnit práci s daty na vysoké úrovni, přímá práce s tabulkami mi umožňuje detailnější kontrolu nad databázovými operacemi a jejich efektivitu. Liquibase, které používám pro správu databázových schémat, mi navíc pomáhá udržet databázovou strukturu jasnou a organizovanou bez nutnosti komplexní konfigurace spojené s JPA.

Dalším důvodem, proč jsem se rozhodla vyhnout JPA, je mé přesvědčení, že přímý přístup k databázovým tabulkám lépe odpovídá mému způsobu myšlení o datech. Tento přístup mi také umožňuje vyvarovat se potenciálních problémů, které mohou vzniknout při mapování objektového modelu aplikace na relační databázový model při operacích souvisejících s načítáním a ukládáním entit. Využitím technologie JOOQ pro tvorbu staticky typovaných dotazů si navíc udržuji vysokou míru flexibility a přesnosti při práci s databází, což by s použitím JPA a jeho abstraktních dotazů mohlo být částečně negováno.

Přestože tato rozhodnutí mohou přinášet určitá omezení, například v menší míře využití automatizace, kterou JPA poskytuje, věřím, že pro potřeby mé aplikace a její aktuální fázi vývoje převažují výhody. Tato volba mi poskytuje jasnější a přímější cestu k efektivnímu návrhu a implementaci databázové vrstvy, zatímco ponechává prostor pro budoucí integraci JPA nebo jiných ORM technologií, pokud se ukáže, že by přinášely značné výhody pro rozvoj projektu.

Následující diagram představuje strukturu databáze v PostgreSQL[33]:

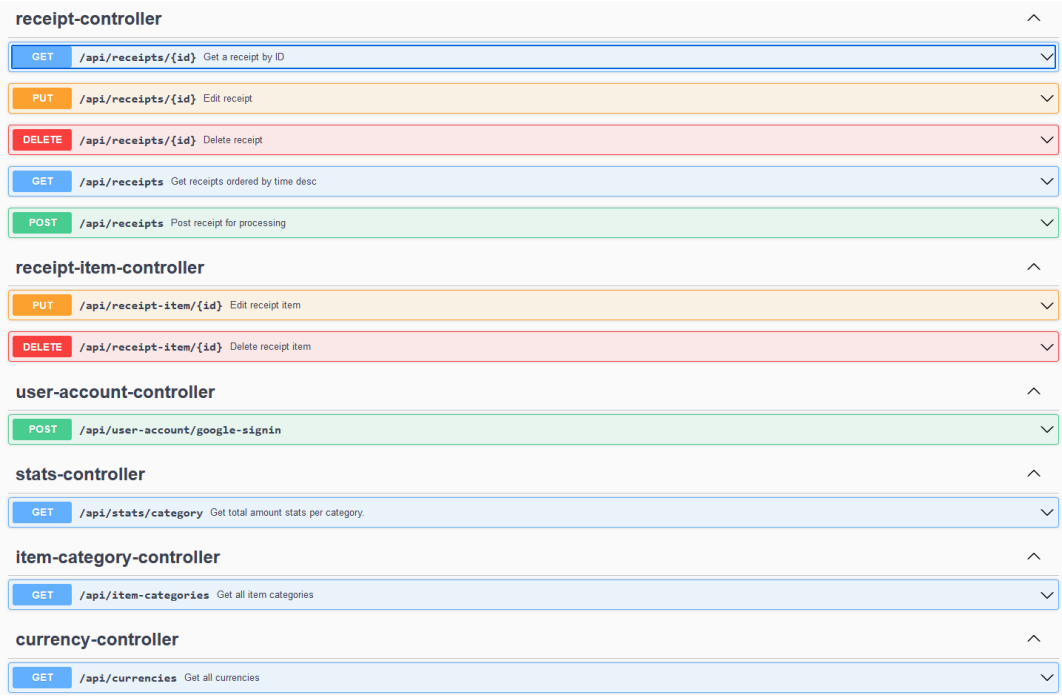
[33] Tento diagram je oproti skutečnému DB návrhu značně zjednodušený pro lepší orientaci. Pro vice detailů a informace např. o datových typech jednotlivých sloupců doporučuji podívat se na diagram vygenerovaný pomocí IDE IntelliJ, obrázek dostupný na Githubu.



5.2. Endpointy

Dokumentace REST rozhraní byla realizována s použitím nástroje Swagger, který je založen na OpenAPI specifikaci (Swagger, 2020). Dokumentace je generována automaticky. Endpointy víceméně kopírují požadavky z use case diagramu.[34]

[34] Kompletní dokumentaci je možno dočasně dohledat na tomto odkaze: https://checkchecker-be-version0-1-test.onrender.com/docs/swagger-ui/, případně obrázek lze ve zvětšené podobě zobrazit na Githubu. Aplikace je v tuto chvíli (10.04.2024) nasazena na testovacím serveru.



Backend aplikace (ve frameworku Spring) využívá standardní architekturu postavenou na vrstvách controllerů, servis a dao[35] objektů. Třídy jsou navrhovány tak, aby splňovaly jedinou zodpovědnost podle principu SRP[36]. Struktura kódu je členěna do tříd s aktivními funkcemi (*services*), které vykonávají byznys operace aplikace, a do tříd typu *Record*, které jsou neměnné (*immutable*). Hodnoty tak stačí tak inicializovat přes konstruktor a není třeba využívat na objektech *setter* metody, což je standard například při použití POJO[37].

Další z dobrých praktik OOP je *composition over inheritance* neboli preference kompozice objektů před dědičností v případě sdílení funkcionalit mezi třídami. Zatímco dědičnost je vhodná pro znázornění vztahů „x je typem y“, může vést k složité a nepřehledné hierarchii a obtížnější údržbě kódu. Oproti tomu kompozice spočívá v tom, že třída obsahuje instance jiných tříd, které rozšiřují její funkcionalitu. Tyto vztahy („x má y“) mohou být flexibilnějším řešením, protože umožňují snadné vyměňování a kombinování komponent. Pro využití tohoto principu jsem se rozhodla z důvodu lepší modulárnosti, snazšího testování a nižší vzájemné závislosti tříd.

[35] *Data Access Object*
[36] *Single Responsibility Principle*, jeden z tzv. *SOLID* principů objektově orientovaného programování (FreeCodeCamp, 2024).
[37] *Plain Old Java Object* (Baeldung, 2024).

6. Implementace

Následuje chronologický popis vývoje aplikace, přičemž zdůrazňuji subjektivně nejdůležitější / nejzajímavější části kódu / procesu vývoje.

6.1 Backend

Základní setup projektu byl konfiguračně výzvou, neboť bylo třeba odchýlit se od výchozí konfigurace některých knihoven, například Liquibase. Ta ve výchozím nastavení migruje databázi při spuštění aplikace, JOOQ však generuje Java kód na základě struktury databáze v jedné z fází Maven buildu[38]. Bylo tedy nutné konfigurovat Maven build tak, aby během něj proběhla i migrace Liquibase, a to ještě před tím, než dojde ke spuštění JOOQ generátoru. Konfigurace v *pom.xml* odpovídá tomuto pořadí – napřed proběhne Liquibase migrace ve fázi inicializace, v následující fázi provede JOOQ generátor na základě DB struktury generování Java kódu.

Poměrně zajímavým důsledkem je závislost buildu na databázi, neboť bez připojení na funkční PostgreSQL není možné build úspěšně dokončit. To samozřejmě zvyšuje složitost procesu vývoje, ale v rámci aplikace přinesl tento fakt možnost jednoduše používat funkce databáze.

Pom.xml[39]:

```
<plugin>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-maven-plugin</artifactId>
    <version>4.2.0</version>
    <configuration>
        <propertyFile>src/main/resources/liquibase.properties</propertyFile>
        <promptOnNonLocalDatabase>false</promptOnNonLocalDatabase>
        <url>${db.url}</url>
    </configuration>
    <executions>
        <execution>
            <phase>initialize</phase>
            <goals>
                <goal>update</goal>
            </goals>
        </execution>
    </executions>
</plugin>
<plugin>
    <groupId>org.jooq</groupId>
    <artifactId>jooq-codegen-maven</artifactId>
```

[38] Přehled a popis fází Maven buildu viz (Apache Maven, 2024)
[39] Snímek obrazovky k nalezení na Githubu.

```
    <version>3.18.6</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <jdbc>
                    <driver>org.postgresql.Driver</driver>
                    <url>${db.url}</url>
                    <user>check_checker_BE</user>
                    <password>xxxx</password>
                </jdbc>
                <generator>
                    <target>
                        <packageName>checkchecker.generated.db</packageName>
                        <directory>${project.build.directory}/generated-
sources/jooq</directory>
                    </target>
                </generator>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Následovala základní definice REST API, nejprve pro získání nejnovějších účtenek, získání konkrétní účtenky přes její ID, získání položek na účtence, a postupně byly implementovány další endpointy dle připraveného schématu a požadavků vyvstanuvších v průběhu vývoje.

Definice Controlleru podle OPEN API specifikace, zde ze souboru *ReceiptController.java*[40]:

```
@Operation(summary = "Get a receipt by ID")
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Receipt found"),
        @ApiResponse(responseCode = "404", description = "Receipt not found", content
= @Content(schema = @Schema(implementation = ClientErrorResponse.class)))
})
@GetMapping("/{id}")
public ResponseEntity<ReceiptResponse> getReceiptById(
        @PathVariable final UUID id,
        final Authentication authentication
) {
    final JwtUserData jwtUserData = createJwtUserData(authentication);
    final ReceiptResponse response = receiptService.getReceiptById(id, jwtUserData);
    if (response == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(response);
}
```

[40] Snímek obrazovky k nalezení na Githubu.

Pro snazší vývoj jsem přistoupila k dockerizaci backendu, a i databázi jsem používala v Docker kontejneru. Kvůli závislosti kódu na stavu databáze (generování JOOQ) bylo nutné jít cestou *multistage buildu* pomocí *docker-compose*. Na testovacích a produkčních prostředích zůstane backend v kontejneru, PostgreSQL však z výkonnostních důvodů poběží mimo kontejner.

docker-compose-DEV.yml[41]*:*

```yaml
version: '3.8'

services:
  maven:
    image: maven:3.9.4-eclipse-temurin-17-alpine
    container_name: check_checker_be-maven-eclipse_temurin-17-alpine
    working_dir: /app
    volumes:
      - .:/app
    networks:
      - mynetwork
    depends_on:
      - db
    command: "mvn package -Pdocker -DskipTests"

  db:
    image: postgres
    container_name: check_checker_DB
    restart: always
    environment:
      POSTGRES_DB: check_checker_DB
      POSTGRES_USER: check_checker_BE
      POSTGRES_PASSWORD: xxxxx
    ports:
      - "5433:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - mynetwork

  app:
    build:  # build context is the current directory
      context: .
      dockerfile: dockerfile
    container_name: check_checker_be
    ports:
      - "8080:8080" # Change if your app uses a different port
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/check_checker_DB
      SPRING_DATASOURCE_USERNAME: check_checker_BE
      SPRING_DATASOURCE_PASSWORD: xxxxxx
    depends_on:
```

[41] Snímek obrazovky k nalezení na Githubu.

```
      - db
    networks:
      - mynetwork

volumes:
  postgres_data:

networks:
  mynetwork:
    driver: bridge
```

V další fázi bylo třeba vytvořit *mock data*[42], aby byl umožněn současný vývoj frontendu (zobrazení účtenky). Jakmile byl frontend schopen odeslat na backend fotografii účtenky, následovala implementace volání *AzureDocumentIntelligenceReceipt* API, která fotografii účtenky převádí do strojově zpracovatelného strukturovaného formátu.

Ukázka ze souboru *CreateReceiptAsyncServiceImpl.java*[43]:

```
@Override
public void processAsync(
        final CreateReceiptParam vo,
        final UUID receiptId
) {
    try {
        final AzureDocumentIntelligenceReceipt ocrResult = callOcr(vo);

        updateReceiptWithOcr(
                ocrResult,
                receiptId
        );
        if (ocrResult.documents() == null || ocrResult.documents().isEmpty()) {
            throw new OcrProcessingException("No documents found in OCR result.");
        }

        final List<AzureDocumentIntelligenceReceiptItem> azureItems = ocrResult
                .documents()
                .get(0)
                .items();
        if (azureItems == null) {
            throw new IllegalStateException("azureItems cannot be null");
        }
        final CategorizedReceiptItems categorizedReceiptItems =
categorizeReceiptItems(azureItems, vo.userAccountId(), vo.userLanguage());
        updateData(categorizedReceiptItems, azureItems, receiptId,
vo.userLanguage());
    } catch (final Exception e) {
        LOGGER.error("Receipt processing error on receipt id " + receiptId, e); }}
```

[42] Simulovaná, uměle vytvořená data, která napodobují strukturu a formát reálných dat. Používají se pro testování.
[43] Snímek obrazovky na Githubu.

Důležitá byla integrace knihovny *MapStruct*, která přemapovává hodnoty mezi objekty. Díky tomu bylo možné automatizovat část vývoje – konkrétně vyhnout se ručnímu přemapovávání hodnot mezi instancemi jednotlivých tříd. Následuje příklad definice jednoho z mapperů, konkrétně *AzureDocumentIntelligenceReceiptItemMapper.java*[44]:

```
@Mapper(componentModel = MappingConstants.ComponentModel.SPRING, uses =
AzureDocumentIntelligenceBaseMapper.class, injectionStrategy =
InjectionStrategy.CONSTRUCTOR)
public interface AzureDocumentIntelligenceReceiptItemMapper {
    @Mapping(source = "Description", target = "description")
    @Mapping(source = "Price", target = "price")
    @Mapping(source = "TotalPrice", target = "totalPrice")
    @Mapping(source = "Quantity", target = "quantity")
    @Mapping(source = "QuantityUnit", target = "quantityUnit")
    AzureDocumentIntelligenceReceiptItem
toAzureDocumentIntelligenceReceiptItem(Map<String, DocumentField> map);
}
```

Generovaná implementace[45]:

```
@Override
public AzureDocumentIntelligenceReceiptItem
toAzureDocumentIntelligenceReceiptItem(Map<String, DocumentField> map) {
    if ( map == null ) {
        return null;
    }

    String description = null;
    Double price = null;
    Double totalPrice = null;
    Double quantity = null;
    String quantityUnit = null;

    if ( map.containsKey( "Description" ) ) {
        description = azureDocumentIntelligenceBaseMapper.getAsString( map.get(
"Description" ) );
    }
    if ( map.containsKey( "Price" ) ) {
        price = azureDocumentIntelligenceBaseMapper.getAsDouble( map.get( "Price" )
);
    }
    if ( map.containsKey( "TotalPrice" ) ) {
        totalPrice = azureDocumentIntelligenceBaseMapper.getAsDouble( map.get(
"TotalPrice" ) );
    }
    if ( map.containsKey( "Quantity" ) ) {
        quantity = azureDocumentIntelligenceBaseMapper.getAsDouble( map.get(
```

[44] Snímek obrazovky dostupný na Githubu.
[45] Snímek obrazovky dostupný na Githubu.

```
"Quantity" ) );
    }
    if ( map.containsKey( "QuantityUnit" ) ) {
        quantityUnit = azureDocumentIntelligenceBaseMapper.getAsString( map.get(
"QuantityUnit" ) );
    }

    AzureDocumentIntelligenceReceiptItem azureDocumentIntelligenceReceiptItem = new
AzureDocumentIntelligenceReceiptItem( description, price, totalPrice, quantity,
quantityUnit );

    return azureDocumentIntelligenceReceiptItem;
}
```

Další výzvu představovala implementace zabezpečení. Díky jednoduchosti implementace jsem zvolila možnost přihlášení prostřednictvím účtu Google. Na backendu následně vytváříme vlastní token pomocí Spring Security. Důvodem je nutnost vkládání vlastních uživatelských dat v rámci JWT[46] řetězce.

JwtServiceImpl.java[47]:

```
@Override
public final String createJwtToken(final String userId, final String languageStr) {
    final Language language = Arrays.stream(Language.values())
            .filter(value -> value.getIso().equals(languageStr))
            .findAny()
            .orElseThrow(() -> new
UnsupportedLanguageException(String.format("Language %s not supported",
languageStr)));

    return Jwts.builder()
            .subject(userId)
            .issuedAt(new Date())
            .claim(Language.PROPERTY_NAME, language)
            .expiration(new Date(System.currentTimeMillis() +
jwtConfig.getJwtExpiration()))
            .signWith(secretKey)
            .compact();
}
```

[46] JSON Web Token
[47] Snímek obrazovky na Githubu.

Až doposud nebyly pevně stanovené všechny kategorie, které budou k dispozici pro automatické třídění. Jakmile jsem sestavila finální seznam kategorií a podkategorií, bylo nutné přidat je do Liquibase a následně i do promptu odesílaného do ChatGPT.

OpenAiServiceImpl.java[48]:

```
private List<ChatMessage> createPromptChatMessages(
        final List<ReceiptItem> items,
        final UUID userId,
        final Language language
) {
    final String itemsCsv = createItemsCsv(items);
    final String actualCategories = getCategoriesAsString(userId, language);
    LOGGER.debug("Actual categories: {}", actualCategories);

    return List.of(
            new ChatMessage(
                    PROMPT_USERNAME,
                    PROMPT_INTRODUCTION + PROMPT_CATEGORIES_EXAMPLE +
PROMPT_INPUT_EXAMPLE_1 + PROMPT_INPUT_EXAMPLE_2 + PROMPT_INPUT_EXAMPLE_3
            ),
            new ChatMessage(
                    PROMPT_AINAME,
                    PROMPT_RESPONSE_EXAMPLE_1 + PROMPT_RESPONSE_EXAMPLE_2 +
PROMPT_RESPONSE_EXAMPLE_3
            ),
            new ChatMessage(
                    PROMPT_USERNAME,
                    PROMPT_INTRODUCTION + actualCategories + itemsCsv
            )
    );
}
```

V průběhu vývoje se ukázalo, že provést všechny kroky (nahrání účtenky, převod na text, kategorizace) najednou je časově poměrně náročné, a že při dlouhém načítání by uživatel mohl nabýt dojmu, že se nic neděje a aplikace nefunguje, případně pokud účtenka některou z fází neprošla, nebylo to dostatečně patrné. Přistoupila jsem tedy k implementaci asynchronního procesování a rozdělila *POST request* na dvě části.

- Na základě požadavku dojde k založení účtenky v DB, uložení souvisejících dat včetně fotografie a vrácení UUID nově založeného záznamu.
- Následně dochází ke zpracování účtenky stejným způsobem jako dosud. Frontend se v kterýkoli moment může dotázat na stav procesování. Do finálního stavu *DONE* se

[48] Snímek obrazovky dostupný na Githubu.

účtenka dostane po kategorizaci jednotlivých položek účtenky. V tento bod frontend prezentuje výsledek procesu uživateli.

Důležitá byla také implementace *PUT* a *DELETE requestů*, aby uživatel mohl libovolné položky na účtence, jakožto i údaje o ní, upravovat a mazat.

ReceiptItemController.java[49]:

```
@Operation(summary = "Edit receipt item")
@ApiResponses(value = {
        @ApiResponse(responseCode = "204", description = "ReceiptItem edited
successfully"),
        @ApiResponse(responseCode = "404", description = "ReceiptItem not found",
content = @Content(schema = @Schema(implementation = ClientErrorResponse.class)))
})

@PutMapping("/{id}")
public ResponseEntity<Void> editReceiptItem(
        @PathVariable final UUID id,
        @RequestBody @Valid final ReceiptItemEditRequest editRequest,
        Authentication authentication
) {
    final JwtUserData jwtUserData =
ControllerUtils.createJwtUserData(authentication);
    try {
        receiptItemService.editReceiptItem(id, editRequest, jwtUserData);
        return ResponseEntity.noContent().build();
    } catch (final IllegalStateException e) {
        return ResponseEntity.notFound().build();
    }
}

@Operation(summary = "Delete receipt item")
@ApiResponses(value = {
        @ApiResponse(responseCode = "204", description = "ReceiptItem deleted
successfully"),
        @ApiResponse(responseCode = "404", description = "ReceiptItem not found",
content = @Content(schema = @Schema(implementation = ClientErrorResponse.class)))
})

@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteReceiptItem(@PathVariable final UUID id) {
    try {
        receiptItemService.deleteReceiptItem(id);
        return ResponseEntity.noContent().build();
    } catch (final IllegalStateException e) {
        return ResponseEntity.notFound().build();
    }
}
```

[49] Snímek obrazovky dostupný na Githubu.

Sestavení statistik pro uživatele byla výzva jak z pohledu implementace, tak z pohledu architektury. Nabízela se možnost sestavovat statistiky na frontendu, problém s tímto řešením spočívá v množství požadavků, které by frontend musel na backend zaslat – každá položka každé účtenky za dobu, za kterou se statistika vytváří. Počítání statistik na backendu navíc přináší do budoucna možnost výsledek snadno cachovat, byť v tuto chvíli toto implementováno není. Po zavolání endpointu pro statistiky dojde vždy k novému spočítání statistik pro jednotlivé kategorie. Zde ukázka *CategoryStatsServiceImpl.java*[50]:

```
@Override
public CategoryStatsResponse loadCategoryStats(
        final OffsetDateTime since,
        final OffsetDateTime until,
        final JwtUserData jwtUserData
) {
    final UUID userId = jwtUserData.userId();
    final Language language = jwtUserData.language();

    // This map contains all existing categories eligible for logged-in user and
corresponding now initialized to ZERO.
    final Map<UUID, CategoryAmountVo> categories = loadCategories(userId, language);

    // This map contains amounts only per category (not including subcategories).
    final Map<UUID, BigDecimal> amounts = loadItemsAndAmountsPerCategory(
            since,
            until,
            categories,
            userId
    );

    for (final Map.Entry<UUID, BigDecimal> entry : amounts.entrySet()) {
        processAmountRow(entry, categories);
    }

    final List<CategoryStatsDto> categoryStatsList = categories.values()
            .stream()
            .map(this::createCategoryStatsDto)
            .collect(Collectors.toList());

    return new CategoryStatsResponse(categoryStatsList);
}
```

Při rozhodování, kam projekt nasadit, zvítězila služba Render (Render, 2024), která v tuto chvíli nabízí nasazení Docker kontejneru a provoz PostgreSQL na tři měsíce zdarma.

[50] Snímek obrazovky dostupný na Githubu.

6.2 Frontend

Zpočátku bylo nutné nastavit prostředí, což zabralo více času a energie, než jsem původně plánovala. Nakonec se ale povedlo propojit React Native s programem Android Studio, nakonfigurovat Babel[51], TypeScript[52], Metro[53] a Eslint[54], a vývoj mobilní aplikace mohl začít. Zde ukázka souboru *app.json*[55], v němž je definována struktura konfigurace Expo[56] projektu:

```
{
   "expo": {
      "name": "CheckChecker-FE",
      "slug": "CheckChecker-FE",
      "jsEngine": "hermes",
      "version": "1.0.0",
      "orientation": "portrait",
      "icon": "./assets/icon.png",
      "userInterfaceStyle": "light",
      "splash": {
         "image": "./assets/splash.png",
         "resizeMode": "contain",
         "backgroundColor": "#ffffff"
      },
      "assetBundlePatterns": ["**/*"],
      "ios": {
         "supportsTablet": true,
         "googleServicesFile": "./GoogleService-Info.plist"
      },
      "android": {
         "googleServicesFile": "./google-services.json",
         "adaptiveIcon": {
            "foregroundImage": "./assets/adaptive-icon.png",
            "backgroundColor": "#ffffff"
         },
         "package": "com.anonymous.CheckCheckerFE"
      },
   "plugins": ["@react-native-google-signin/google-signin"]
   }
}
```

[51] Javascriptový transpiler, který umožňuje psát moderní JavaScript (ES6+), který je následně přeložen do zpětně kompatibilního kódu a funguje i ve starších prohlížečích.
[52] Rozšíření Javascriptu o statické typování.
[53] Bundler pro React Native, který převádí zdrojový kód aplikace a jejích závislostí do jednoho balíčku (apk).
[54] Nástroj pro statickou analýzu kódu a udržování jeho kvality.
[55] Snímek obrazovky dostupný na Githubu.
[56] Expo využívám pro snadný proces vývoje a distribuci aplikace. Má vlastní bundler a sadu nástrojů, kterými doplňuje ty výše zmíněné.

Design aplikace měl být již od začátku velmi minimalistický, protože v první iteraci jde především o funkcionalitu.[57] Již na začátku jsem tedy vybrala základní barevnou paletu[58], vytvořila *HomeScreen*, *Sidebar*, *Upload Modal*, a byla jsem připravena na první zkušební API call (endpoint *GET receipts* v komponentě *Latest Receipts*)[59]:

```
const fetchReceipts = async (pageNumber = 1) => {
    setIsLoading(true);
    setLoadedPages({ ...loadedPages, [pageNumber]: true });
    const token = await getToken();
    if (!token) {
       setError(t("noTokenFound"));
       return;
    }
    try {
       const pageSize = 10;
       const response = await
fetch(`${API_ENDPOINT}/api/receipts?pageNumber=${pageNumber}&pageSize=${pageSize}`, {
            headers: { Authorization: `Bearer ${token}`, },
        });
       const data = await response.json();
       if (data.receipts) {
          function parseReceiptStatus(
             status: string
          ): ReceiptStatus | undefined {
             return Object.values(ReceiptStatus).includes(
                status as ReceiptStatus
             )
                ? (status as ReceiptStatus)
                : undefined;
          }
          const processedReceipts = data.receipts.map(
             (receipt: { status: string }) => ({
                ...receipt, status: parseReceiptStatus(receipt.status),
          }));
          if (pageNumber === 1) {
             setReceipts({ receipts: sortReceipts(processedReceipts) });
          } else {
             setReceipts((prevState) => ({
                receipts: sortReceipts([...prevState.receipts,...processedReceipts]),
             }));
          }
          setHasNextPage(data.receipts.length === pageSize);
       } else {
          setError(
             data.message || "An error occurred while fetching receipts");
          setHasNextPage(false);
```

[57] Pokud bude projekt dál úspěšný, pravděpodobně bude design předělán na základě návrhu grafika či UI designéra.
[58] Pomocí nástroje *Color Palettes* (SheCodes, 2024).

[59] Snímek obrazovky dostupný na Githubu.

```
      }
    } catch (err) {
      setError(
        err instanceof Error ? err.message : "An unknown error occurred");
      setHasNextPage(false);
    }
    setIsLoading(false);
    dispatch({ type: "RESET_UPLOAD_COMPLETED" });
 };
```

Bylo nutné vyřešit přihlašování, a jak jsem již zmínila, *Google Sign In* se jevil jako nejlepší řešení. Následoval tedy průzkum možností propojení s React Native, výsledkem je nezbytný soubor *google-services.json* a komponenta *Login.tsx*. Po implementaci na backendu pak bylo nutné začít přidávat do *headers* autorizační token, a od této chvíle se účtenky zobrazovaly pouze přihlášenému uživateli. *Login.tsx*[60]:

```
const signInWithGoogle = async () => {
    try {
       await GoogleSignin.hasPlayServices();
       const userInfo = await GoogleSignin.signIn();
       if (userInfo && userInfo.idToken) {
          const response = await fetch(
             `${API_ENDPOINT}/api/user-account/google-signin`,
             {
                method: "POST",
                headers: {
                   "Content-Type": "application/json",
                },
                body: JSON.stringify({
                   idToken: userInfo.idToken,
                   language: currentLanguage,
                }),
             }
          );
          const json = await response.json();
          if (response.ok && json.token) {
             await AsyncStorage.setItem("jwt_token", json.token);
             dispatch({
                type: "LOGIN_SUCCESS",
                payload: { userName: userInfo.user.name },
             });
             if (userInfo.user.name) {
                onLoginChange(userInfo.user.name);
             }
             Alert.alert(t("success"), t("signInSuccess"));
          } else {
             Alert.alert(t("error"), t("signInError"));
```

[60] Snímek obrazovky dostupný na Githubu.

```
            }
        } else {
            console.error("User info or idToken is missing.");
        }
    } catch (error) {
        const errorCode = (error as { code?: string }).code;
        if (errorCode === statusCodes.SIGN_IN_CANCELLED) {
        } else {
            console.error("Google Sign-In Error:", error);
            Alert.alert(t("error"), t("loginFailed"));
        }
    }
};
```

Vývoj pak probíhal souběžně s vývojem backendu. Každý přidaný endpoint na backendu jsem zakomponovala do frontendu, otestovala, případně opravila chyby v React Native či v Javě. Byly přidány modály pro editaci účtenky a položek účtenky. Pracným, avšak implementačně nenáročným úkolem bylo přidání jazykových verzí aplikace (lokalizace do českého, anglického a německého jazyka) a otagování všech textových výstupů v aplikaci.

Posledními zajímavými částmi implementace bylo již zmíněné asynchronní zpracování účtenky, ale dále také např. třídění podle data či částky[61], postupné načítání více účtenek[62] a samozřejmě grafická vizualizace. Tam se jako nejproblematičtější ukázal výběr správné React Native knihovny, která umí s grafy pracovat podle potřeb této aplikace. Nakonec jsem zvolila knihovnu Victory Native, protože umožňuje více přizpůsobovat graf a labely. *GraphicalVisualization.tsx*[63]:

```
const fetchData = async (startDate: string, endDate: string) => {
    const token = await getToken();
    const headers = {
        Authorization: `Bearer ${token}`,
    };
    try {
        const response = await fetch(
            `${API_ENDPOINT}/api/stats/category?since=${encodeURIComponent(
                startDate
            )}&until=${encodeURIComponent(endDate)}`,
            { headers }
        );
        const data = await response.json();
        console.log("API RESPONSE", data);
        setFetchedData(data.stats);
        if (data && Array.isArray(data.stats)) {
```

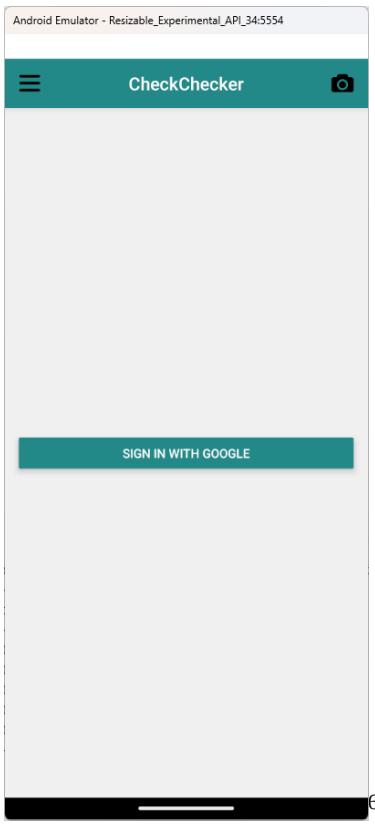
[61] Funkcionalita je však omezena.
[62] Počet účtenek na stránce je omezen na 10.
[63] Snímek obrazovky dostupný na Githubu.

```
            const formattedData = formatChartData(data.stats);
            setCategoryStatsData(formattedData);
        } else {
            console.error("Invalid response structure:", data);
        }
    } catch (error) {
        console.error("Error fetching category stats:", error);
    }
};
```

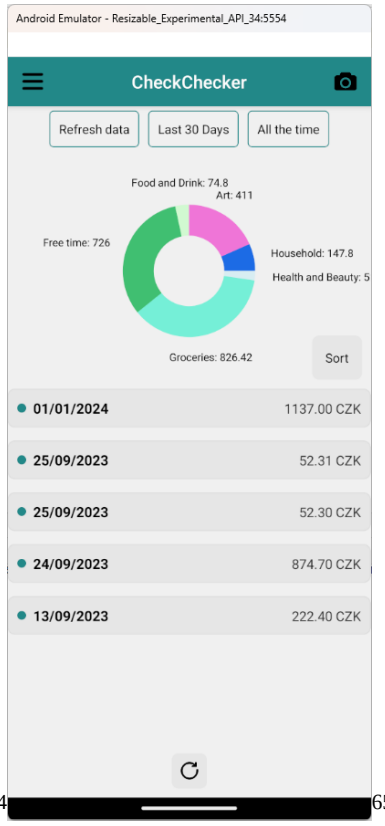
7. Náhled aplikace na konci první iterace

V tento bod je na místě ukázat, jak samotná aplikace nazvaná v češtině **Účtenkovač**, v angličtině **CheckChecker**, vypadá ve své finální podobě v rámci první iterace. Snímky obrazovky aplikace jsou pořízeny v Android Emulatoru.

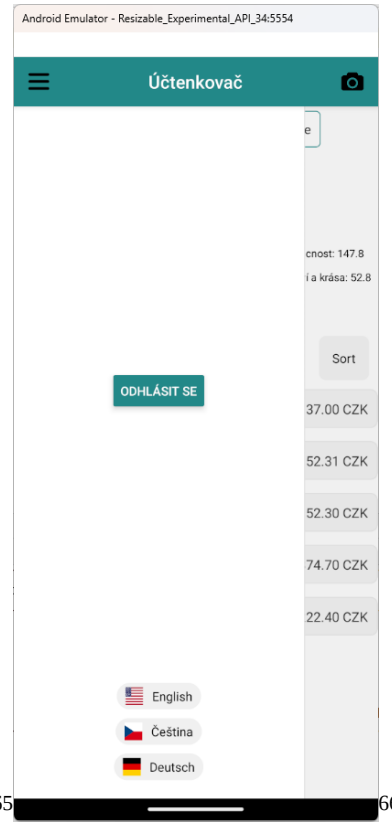
Vlevo na obrázcích níže je vidět úvodní obrazovka aplikace. Dokud se uživatel nepřihlásí, není schopen nahrávat účtenky a ani nic zobrazit. Uprostřed pak náhled domovské obrazovky uživatele po přihlášení. Oba snímky jsou pořízeny v anglické jazykové verzi. Snímek vpravo ukazuje postranní panel, v němž si uživatel může přepnout aplikaci do jiného jazyka nebo se odhlásit. Přepnutí do jiného jazyka uživatele také odhlásí, což je nutné z důvodu změny jazyka v JWT řetězci (kde informace slouží pro načtení správné jazykové verze kategorií, jejichž názvy jsou uloženy v databázi).



[64]



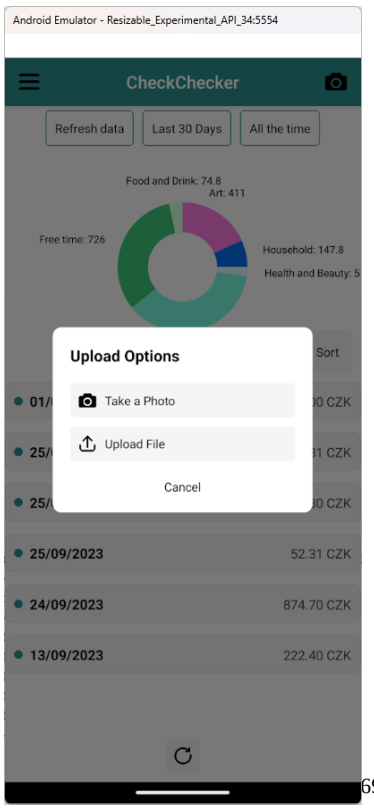
[65]



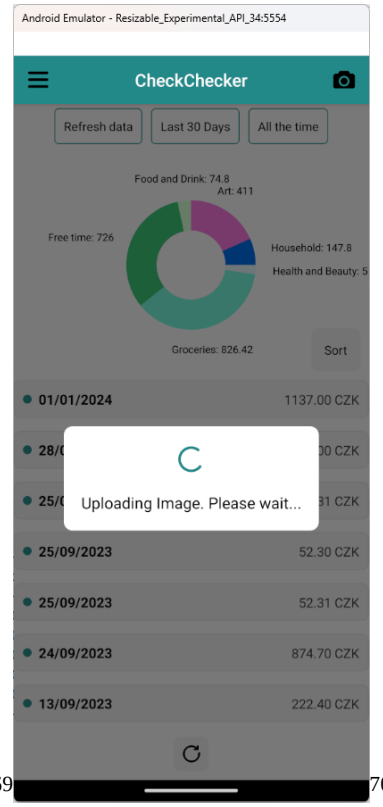
[66]

[64] Snímek obrazovky dostupný na Githubu.
[65] Snímek obrazovky dostupný na Githubu. Rozbalená účtenka je pak k nahlédnutí zde.
[66] Snímek obrazovky dostupný na Githubu.

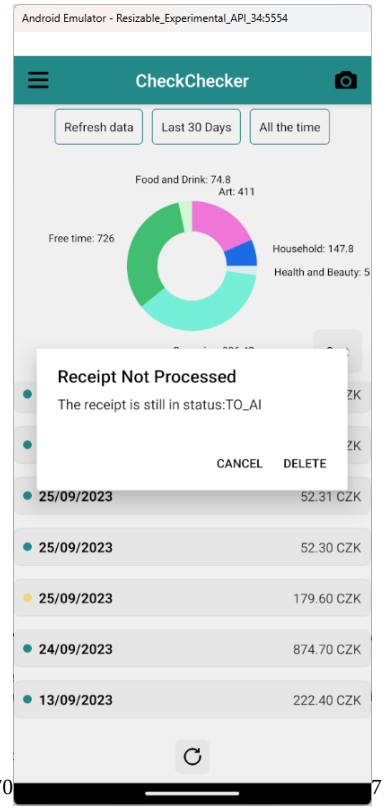
Další ukázky zobrazují nahrávací modál, který se uživateli zobrazí po kliknutí na ikonku fotoaparátu v pravém horním rohu. Uživatel může vyfotit účtenku přímo z aplikace (neukládá se mu do telefonu), nebo nahrát libovolný soubor do velikosti 4 MB a ve formátu *png, jpg* nebo *pdf*[67]. Uprostřed je vidět modál, který se zobrazí při nahrávání účtenky. Po implementaci asynchronního zpracování účtenky dochází k jejímu nahrání velmi rychle, následně uživatel vidí stav účtenky podle barvy vlevo od data účtenky (obrázek vpravo). Pokud je puntík šedý, značí to stav *TO_OCR* (účtenka ještě nebyla převedena do digitální podoby), pokud je žlutý, stav je *TO_AI* a probíhá kategorizace položek na účtence. V těchto dvou stavech není možné účtenku rozkliknout, pokud by se v nich ale zasekla dlouhodobě[68], uživatel ji může smazat a zkusit nahrát znovu. Modrá barva pak značí zpracovanou účtenku, kterou je možné rozkliknout a zobrazit.



69



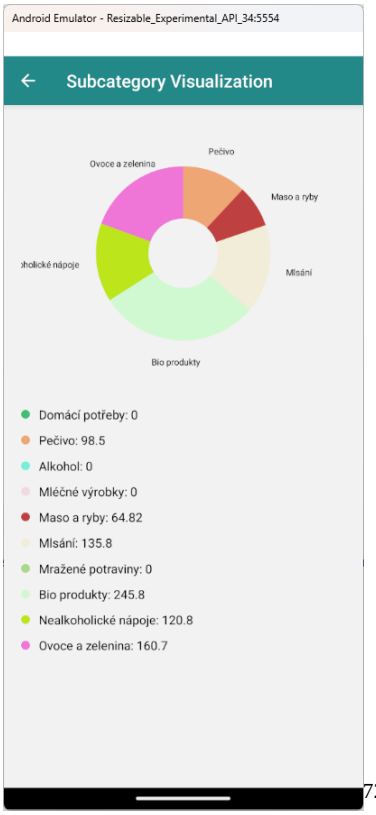
70

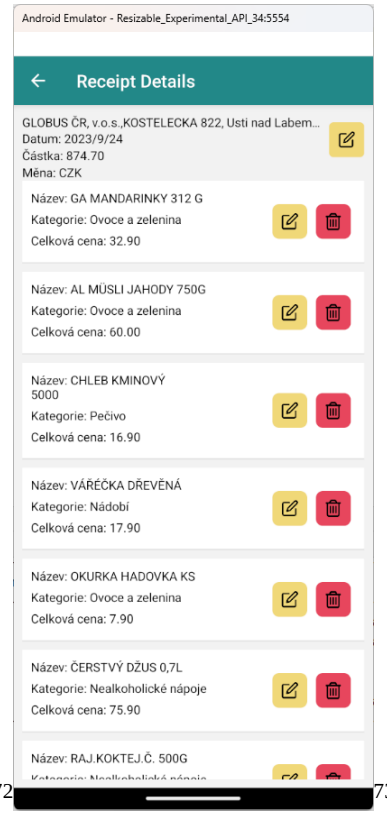


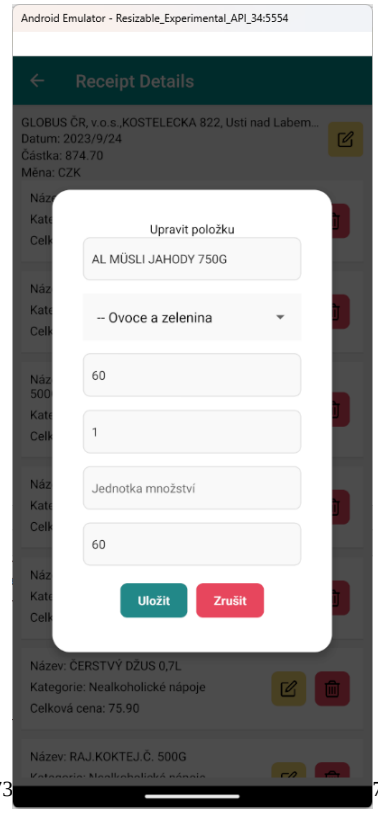
71

[67] Všechny podporované formáty v aplikaci jsou: *png*, *jpg*, *jpeg*, *pdf*, *bmp*, *tiff*, *heif*. Limitace je dána použitou OCR technologií.
[68] Zhruba v jednom z 300 případů se účtenku nepodařilo kategorizovat, protože obsahovala zvláštní znaky. Může ale dojít i k jiným technickým problémům, které znemožní zpracování účtenky.
[69] Snímek obrazovky dostupný na Githubu.
[70] Snímek obrazovky dostupný na Githubu.
[71] Snímek obrazovky dostupný na Githubu.

Zbývá ukázat detail statistik pro vybranou kategorii (obrázek vlevo), který uživatel vidí po kliknutí do grafické vizualizace na hlavní stránce. Obrázek uprostřed zachycuje *Receipt Screen*, který se uživateli zobrazí po rozkliknutí účtenky (tlačítko „Zobrazit celou účtenku“). Zde jsou vidět detaily o účtence a o všech položkách v ní. Uživatel může změnit detaily účtenky (např. měnu či datum) nebo detail každé jednotlivé položky. Nejčastěji editační modál využije (obrázek vpravo), pokud bude chtít změnit kategorii položky. Smazáním položky či úpravou její ceny dojde k přepočítání celkového množství na účtence.







72 73 74

V závěrečné fázi vývoje jsem opět využila nástroj Expo, který byl již během celého procesu vývoje klíčovým nástrojem umožňujícím efektivní iteraci a testování aplikace[75]. Díky integraci s Expo byl proces sestavení finální distribuční verze aplikace ve formátu APK pro platformu

[72] Snímek obrazovky dostupný na Githubu.
[73] Snímek obrazovky dostupný na Githubu.
[74] Snímek obrazovky dostupný na Githubu.
[75] Framework *expo* jsem využila při vývoji aplikace jak v emulátoru, tak na testovacím telefonu. (Expo, 2024)

Android značně zjednodušen[76]. Pro zvýšení stability a spolehlivosti mobilní aplikace jsem rovněž integrovala monitorovací a reportovací službu Sentry (Sentry, 2024). Tato platforma poskytuje detailní vhledy do chování aplikace v provozu, automaticky detekuje a reportuje výskyt chyb a výjimek, což umožňuje rychlou diagnostiku a řešení potenciálních problémů.

[76] Expo navíc nabízí webového klienta, kde jsou přehledně dostupné veškeré informace o buildech. Viz obrázek na Githubu.

Závěr

Ve své bakalářské práci jsem se v její teoretické části zaměřila na představení technologií OCR a AI, které obě spadají pod strojové učení. Popsala jsem vysvětlení jejich základních principů, přičemž si uvědomuji, že skutečné implementace těchto algoritmů jsou mnohem komplexnější. Dále jsem se zabývala průzkumem současného (českého) trhu aplikací, které se zabývají evidencí výdajů, a definovala jsem požadavky na aplikaci, která by mezeru na trhu zaplnila. V praktické části práce jsem pak shrnula klíčové aspekty vývoje této aplikace a prezentovala stav, v němž se aplikace v tuto chvíli nachází.

V rámci vývoje software pro kategorizovanou evidenci výdajů jsem se snažila postupovat ve všech ohledech podle *best practice*, ať už z byznysového nebo vývojářského pohledu. Mým cílem bylo vyvinout funkční a použitelnou aplikaci, která uživatelům přinese užitek za minimální náklady. Z tohoto hlediska jsem uspěla, aplikace se dá nainstalovat do Android zařízení a je možné používat její základní funkce. Konkrétně se podařilo splnit tyto cíle:

- ✓ Aplikace umožňuje uživateli nahrát účtenku, a to buď pořízením fotografie nebo nahráním souboru z telefonu
- ✓ Aplikace zpracovává účtenku do textové podoby pomocí *Azure Document Intelligence*
- ✓ Aplikace kategorizuje položky na účtence pomocí *ChatGPT*
- ✓ Aplikace umožňuje uživateli editovat účtenku i její položky
- ✓ Aplikace poskytuje grafickou vizualizaci výdajů v hlavních kategoriích i v podkategoriích

Touto bakalářskou prací však vývoj aplikace nekončí. Je zde mnoho drobných problémů, které je třeba doladit, funkcionality, které si uživatelé přáli a které zatím nebyly implementovány, a samozřejmě mám v zásobě další nápady, které jsem v rámci této práce zmínila (viz kapitolu 2), ale na které nezbyla časová kapacita. Předpokládám také, že po několika měsících testování aplikace uživateli dostanu cennou zpětnou vazbu, od níž se budu moct při dalším vylepšování a ladění odrazit. Prioritu mají tyto cíle:

- o Umožnit uživatelům další formy přihlášení
- o Zpřístupnit aplikaci uživatelům operačního systému iOS
- o Optimalizovat počet odeslaných požadavků na server a cachování dat na frontendu

- Zlepšit přesnost kategorizace položek
- Testovat aplikaci a opravovat drobnosti, jako např. stylování, formátování nebo funkcionality tlačítek

Další, poněkud ambicióznější cíle, které by aplikaci posunuly na vyšší úroveň a udělaly z ní skutečně aktivního pomocníka při snaze ušetřit peníze, jsou pak:

- Umožnit vyhledávání cen položek v obchodech, nabízet levnější alternativy
- Na základě dat o cenách produktů v jednotlivých obchodech rozpoznat, co je skutečná sleva a co ne
- Sledovat cenovou hladinu položek

Do budoucna by také stál za zvážení přechod na vlastní implementaci OCR a LLM. Knihovny od Azure a OpenAI jsou jednoduché na konfiguraci a v malém měřítku takřka zadarmo. Pokud by se ale měla aplikace rozrůstat a přinášet profit, stálo by za zvážení nějaké vlastní řešení, které bude méně nákladné a podobně (nebo třeba více) spolehlivé. Tento cíl mi může sloužit jako inspirace na diplomovou práci.

Přílohy

K této bakalářské práci se pojí Github repozitář **thesis_VM_2024** dostupný na adrese https://github.com/vlastami/thesis_VM_2024, který má následující strukturu a obsah:

- App-views (Snímky obrazovky s ukázkami Android aplikace)
- Code-snippets (Snímky obrazovky s ukázkami kódu z backendu a frontendu aplikace)
- Diagrams (Diagramy prezentované v této práci)
- README.md (Zadávací text)
- Výsledky dotazníku.pdf
- Bakalářská práce.pdf

Reference

Apache Maven. (11. 04 2024). Získáno 09. 03 2024, z Introduction to the Build Lifecycle: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

Aydin, A. (04. 10 2023). *Aysel Aydin Medium*. Získáno 10. 04 2024, z 1 — Text Preprocessing Techniques for NLP: https://ayselaydin.medium.com/1-text-preprocessing-techniques-for-nlp-37544483c007

Baeldung. (2024). Získáno 09. 03 2024, z What Is a POJO Class in Java?: https://www.baeldung.com/java-pojo-class

Basiková, A. (17. 02 2022). *Forbes*. Získáno 09. 03 2024, z Pořádek v peněžence: Těchto šest aplikací vám usnadní správu osobních financí: https://forbes.cz/poradek-v-penezence-techto-sest-aplikaci-vam-usnadni-spravu-osobnich-financi/

Buckland, M. (2006). *Goldberg Statistical Machine*. Získáno 09. 03 2024, z https://people.ischool.berkeley.edu/~buckland/statistical.html

Clu, P., & další. (11. 04 2021). *Microsoft*. Získáno 09. 03 2024, z Convert Word to Vector in Azure Machine Learning: https://learn.microsoft.com/en-us/azure/machine-learning/component-reference/convert-word-to-vector?view=azureml-api-2

Expo. (2024). Získáno 19. 03 2024, z Documentation: https://expo.dev

Fakturoid. (2024). Získáno 09. 03 2024, z Finance pod kontrolou: Tipy na chytré nástroje a aplikace pro freelancery: https://www.fakturoid.cz/almanach/pruvodci/zdrave-finance/chytre-nastroje-a-aplikace

Farley, P., Mehrotra, N., & Urban, E. (18. 07 2023). *Microsoft*. Získáno 09. 09 2024, z Overview of the OCR technology: https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/overview-ocr

FreeCodeCamp. (2024). Získáno 09. 03 2024, z SOLID Principles: Single Responsibility Principle Explained: https://www.freecodecamp.org/news/solid-principles-single-responsibility-principle-explained/

Gallo, N. (21. 09 2023). *The Evolution of Personal Finance, From the 70s to Now*. Získáno 09. 03 2024, z Finmasters: https://finmasters.com/evolution-of-personal-finance/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

Christensson, P. (18. 04 2018). *TechTerms*. Získáno 09. 04 2024, z OCR Definition: https://techterms.com/definition/ocr

IBM. (24. 03 2023). Získáno 09. 03 2024, z Advantages of Java: https://www.ibm.com/docs/en/aix/7.3?topic=monitoring-advantages-java

Miles, J. (06. 10 2020). *Open Access Government*. Získáno 09. 03 2024, z The evolution of the personal finance industry and where it is headed: https://www.openaccessgovernment.org/the-evolution-of-the-personal-finance-industry-and-where-it-is-headed/95566/

OpenAI. (2024). Získáno 09. 03 2024, z GPT-3.5 Turbo Model Documentation: https://platform.openai.com/docs/models/gpt-3-5-turbo

React Native. (08. 12 2023). Získáno 09. 03 2024, z Introduction to React Native Components: https://reactnative.dev/docs/intro-react-native-components

Render. (2024). Získáno 19. 03 2024, z Cloud Hosting for Developers: https://render.com/

Sentry. (2024). Získáno 19. 03 2024, z React Native Documentation: https://docs.sentry.io/platforms/react-native

SheCodes. (2024). Získáno 19. 03 2024, z Color Palettes for Developers: https://palettes.shecodes.io/

Spring. (2024). Získáno 09. 03 2024, z Spring Boot: https://github.com/spring-projects/spring-boot

Swagger. (2020). Získáno 10. 04 2024, z OpenAPI Specification - Version 3.0: https://swagger.io/specification/v3/

Tesseract OCR. (18. 01 2024). Získáno 11. 04 2024, z Open Source OCR Engine: https://github.com/tesseract-ocr/tesseract

The PostgreSQL Global Development Group. (2023). Získáno 31. 10 2023, z PostgreSQL: The World's Most Advanced Open Source Relational Database: https://www.postgresql.org/about/

Tripathi, P. (27. 09 2023). *Docsumo*. Získáno 09. 04 2024, z The Brief History of OCR Technology: https://www.docsumo.com/blog/optical-character-recognition-history

Urban, E., & další. (29. 02 2024). *Microsoft*. Získáno 09. 03 2024, z Concept Receipt in Azure Document Intelligence: https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/concept-receipt?view=doc-intel-4.0.0