FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	BillSaver – Technicka analýza a proof of concept
Student:	Patrik Malý
Vedoucí:	Ing. David Pešek
Studijní program:	Informatika
Obor / specializace:	Informační systémy a management
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

BillSaver je aplikace, která při bezhotovostní platbě mobilem přebírá a ukládá všechny položkové informace o nákupu. Uživatel pak má přístup ke svým účtenkám v elektronické podobě. U jednotlivé účtenky pak nalezne názvy, ceny a kategorie jednotlivých položek. Tyto informace pak tvoří přehled výdajů uživatele. Přehled je udělán formou názorných grafů a může pomoci uživateli s vedením osobního účetnictví.

Zadání: Proveďte detailní technickou analýzu aplikace a vytvořte proof of concept. Proveďte studii proveditelnosti, finanční analýzu a harmonogram práce nad projektem.

Elektronicky schválil/a Ing. David Buchtela, Ph.D. dne 8. března 2021 v Praze.

Bakalářská práce

BILLSAVER – TECHNICKA ANALÝZA A PROOF OF CONCEPT

Patrik Malý

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. David Pešek
5. ledna 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Za odborné vedení mé bakalářské práce, velkou míru trpělivosti a ochoty, rychlost, lidský přístup a také za cenné a velmi podnětné rady při zpracovávání práce děkuji vedoucímu práce Ing. Davidu Peškovi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 5. ledna 2022 .....

Abstrakt

Tato bakalářská práce se zabývá technickou a ekonomickou stránkou projektu BillSaver, který má za úkol vytvořit jednotný elektronický účtenkový systém, s přehledem jednotlivých pokožek na účtence a dostupný z mobilní aplikace. BillSaver dále usnadňuje vedení osobního učetnictví kvůli názornému přehledu spotřebitelské nákupní historie. Práce probírá různé procesy, architektonické vzory a technologie za účelem vytvoření prvotní architektury systému. Práce se dále zaměřuje na studii proveditelnosti ve které vytvoří plán budoucího vývoje, zanalyzuje finanční stránku celého projektu a zhodnotí potencionální rizika.

Vytvořené řešení poskytuje flexibilní, škálovatelnou a robustní architekturu která splňuje stanovené požadavky obchodníku a spotřebitelů. Dále architektura poskytuje jednoduchou cestu k budoucí expandibilitě. Studie proveditelnosti stanovuje konkretní postup při budoucím vývoji, stanovuje rozpočet, odhaduje budoucí výnos a popisuje odhadnuta rizika a navrhuje jejich případnou mitigaci.

Na závěr práce je vytvořen proof of concept, který testuje navrženou architekturu.

Klíčová slova studie proveditelnosti, technická analýza, mikroslužby, účtenkový systém, informační systém

Abstract

This bachelor thesis deals with the technical and economic side of BillSaver project, which aims to create a unified electronic receipt system with an itemized list of products on the receipt, available from a mobile app. BillSaver further facilitates personal accounting for a clear overview of consumer shopping history. The work discusses various processes, architectural patterns and technologies in order to create the system architecture. The work also focuses on a feasibility study in which it creates a plan for future development, analyzes the financial side of the entire project and evaluates potential risks.

The created solution provides a flexible, scalable and robust architecture that meets the set requirements of traders and consumers. Furthermore, the architecture provides an easily expandabile system. The feasibility study sets out a specific approach for future development, sets a budget, estimates future revenue and describes the potential risks and suggests possible mitigation strategies.

Finally a proof of concept is created that tests the proposed architecture.

Keywords feasibility study, technical analysis, microservice, receipt system, information system

Seznam zkratek

API	Application Programming Interface
CI/CD	Continuous integration (CI) / continuous deployment (CD)
DBMS	Database management system
DevOps	Development and Operations
EAN	European Article Number
EDA	Událostmi řízená architektura (Event-driven architecture)
EET	Elektronická evidence tržeb
GUI	Grafické uživatelské rozhraní
HR	Lidské zdroje
JVM	Java virtual machine
MD	Man day
PR	Vztahy s veřejností
PS	Pracovni smlouva
REST	Representational State Transfer
SDLC	Životní cyklus informačního systému
SWOT	S = Silné stránky, W = Slabé stránky, O = Příležitosti, T = Hrozby
SaaS	Software as a service
UI	Uživatelského rozhraní

Kapitola 1
Úvodní strana

1.1 Úvod

Pokud chce mít běžný spotřebitel celkový přehled o vlivu jeho nákupních zvyklostí na vlastní finanční situaci musí skladovat účtenky, extrahovat z nich ručně informace a tyto informace opět ručně zaznamenávat v nějaké mobilní aplikaci k tomu určené a nebo v horším případe v excelu. Toto úsilí zabírá čas, který snižuje pravděpodobnost pravidelně tyto informace zaznamenávat a na základě výsledku provést potřebné kroky k zlepšování finanční situace spotřebitele.

Další problém je s produkty které mají záruku. Od těchto produktů je nutné uchovávat účtenky, kterými se musí spotřebitel prokazovat u reklamace daného produktu. Účtenky jsou však tisknuty na papír který po čase tmavne a degraduje. Tím pádem musí spotřebitel tyto účtenky správně skladovat a nebo převádět do elektronické podoby.

Pro řešení těchto problémů zde navrhujeme aplikaci která automaticky sbírá účtenky od obchodníků, skladovat je pro spotřebitele, a nakonec z nich extrahovat užitečné data které se hodí v každodenním životě. Pro obchodníka také navrhneme užitečnou funkcionalitu, kterou mu ulehčí činit užitečné a kritické rozhodnuti která posunou jeho firmu před konkurenci.

1.2 Cíl práce

Hlavním cílem bakalářské práce je provést technickou analýzu projektu BillSaver. A vytvořit funkční prototyp formou proof of concept.

Proto se nejprve v teoretické části zaměřuji na stávající techniky vývoje softwaru. Především probírám různé architektonické návrhy aplikací a to zejména z pohledu použitelnosti pro BillSaver. Způsoby vývoje aplikací a časovou náročnost daného stylu vývoje. Dále se zaměřím na možnosti škálovatelnosti programu. Jaké vhodné databázové systémy existují. Jaké designové vzory se používají pro podobné typy aplikací. A také diskutuji nad typy aplikačních rámců pro zjednodušení tvorby různých aspektů programu.

Studie proveditelnosti se bude zaměřovat na technickou a ekonomickou stránku projektu. Navrhuji základní technické a technologické řešení projektu. Nastíním možné komplikace při vývoji projektu a k ním navrhnu možná řešení. Vytvořím harmonogram práce který bude zohledňovat efektivní využití dostupných pracovních sil. Vytvořím finanční analýzu ve které se pokusím minimalizovat cenovou náročnost řešení. Nakonec vytvořím proof of concept který bude moci být prezentovatelný potencionálním zájemcům.

Kapitola 2
Design aplikace

2.1 Základní řízení procesu vývoje softwaru

Řízení procesu vývoje softwaru se na první pohled může zdát jako velice jednoduchá záležitost. Dostaneme specifika programu od zákazníka, vytvoříme plán a začneme tvořit. Jako spousta věcí v životě ale ani toto není tak jednoduché. Zákazník bohužel nemůže přijít do obchodu a koupit si program, který má ty nejlepší parametry. Vůbec definovat kvalitu programu je velice obtížné. Měly by se parametry zaměřovat na počet řádků kódu, nebo na rychlost programu? Nejspíš ne. Více řádků kódu nemusí znamenat neefektivitu, může to být naopak známka promyšleného designu, který se v celém programu používá jednotně. Chybovost tak bude menší a bude jednodušší rozšiřovat kompatibilitu a funkčnost celého programu. Naopak i ta nejrychlejší aplikace nemusí byt dostatečně rychlá pro obsluhu milionu zákazníků současně. Není tedy jasné, jak specifikovat a monitorovat kvalitu softwarového vývoje.

Protože vývoj zahrnuje spoustu různých aktivit od tradiční programátorské práce, která tvoří základ každého vývoje, přes design uživatelského rozhraní, které umožní zákazníkům se jednoduše orientovat v digitálním světě, až po marketingový tým, jenž nejen vytváří propagační kampaň pro vyvíjený produkt, ale dostává i zpětnou vazbu jak od budoucích uživatelů, tak od zákazníků, kteří si daný produkt objednali. Proto je je velmi náročné sladit tyto týmy tak, aby vše proběhlo bez komplikací a aby nikdo na nikoho nečekal. V opačném případě vznikají zpoždění a kvalita celého projektu klesá.

Pro vývoj softwaru tedy musíme použít jiné inženýrské postupy než u tradičnějších způsobů výroby. Jinak bychom nevyužili všechny výhody vycházející z toho, že software je taková nehmatatelná substance, jež pohání náš svět. [1]

Výběr vhodného procesu pro vývoj BillSaveru nám zajistí jednoduchý a kvalitní vývoj. Více o tom jaký řídící proces jsme vybrali a proč si probereme v kapitole 3.1.5 a 3.3. Zde si vytvoříme pouze informovaný základ.

Celý vývoj by se dal rozdělit do několika etap: plánování, analýza, návrh, implementace a údržba. Tomuto rozdělení se říká SDLC (system development life cycle). V této studii se zaměříme na plánování, analýzu a převážně návrh.

- V plánovací fázi se zaměřujeme nejvíce na studii proveditelnosti. Vytvoříme a zkonsolidujeme vývojový tým. Vytvoříme strategii vývoje a vymezíme si vztyčné cíle. Identifikujeme možné budoucí problémy, vybereme vhodné technologie a ohraničíme podnikatelský rozsah celého projektu.

- V analýze bychom měli klasifikovat problémy, které se mohou vyskytnout při implementaci, a navrhnout jejich řešení. Vytvoříme rešerši podobných projektů a s týmem se dohodneme

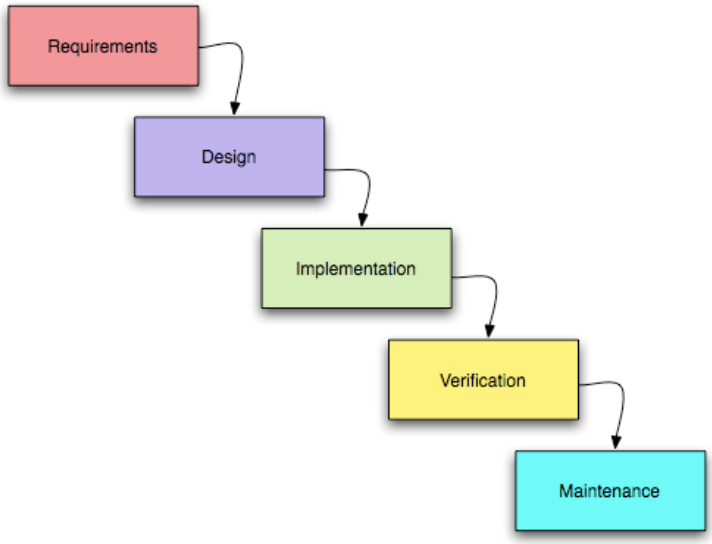
na tom, jaká implementace nejlépe odpovídá našemu případu použití. Zanalyzujeme systém s pomocí již vytvořené rešerše a nakonec poukážeme na nezbytné části a potřeby systému, které budeme podrobně navrhovat v další části.

- Při návrhu musíme zanalyzovat data, která se budou uchovávat, a k nim vytvořit vhodný databázový model. Dále je třeba analyzovat interakci jednotlivých objektů mezi sebou, vytvořit jejich výhodná rozhraní a nakonec navrhnout uživatelské rozhraní.
- Implementovat se bude již vyhotovená analýza, což znamená vytvoření databázového schématu podle databázového modelu a vytvoření aplikace z vyhotoveného návrhu. Po dokončení implementace se otestuje všechna funkcionalita a nakonec se opraví objevené chyby.
- Závěrečným krokem je údržba vytvořeného systému. Tato práce je prováděna administrátorem systému, jenž se stará o monitorování a údržbu celého systému. [2]

SDLC je však pouze teoretický základ pravidel, a proto vzniklo několik hlavních implementací. Tyto postupy se dělí na vodopádový model, metodologii ve tvaru V a agilní model. [3] [2] Každá z těchto implementací má své výhody a nevýhody, jež si nyní porovnáme:

2.1.1 Waterfall

Waterfall model neboli vodopádový model je specifická implementace SDLC, která byla prvně popsaná v roce 1970 hlavním vedoucím Lockhed Software Technology Center Dr. Winstonem W. Roycem. [4] Je to tedy jedna z nejstarších implementací SDLC.



Obrázek 2.1 Waterfall model [5]

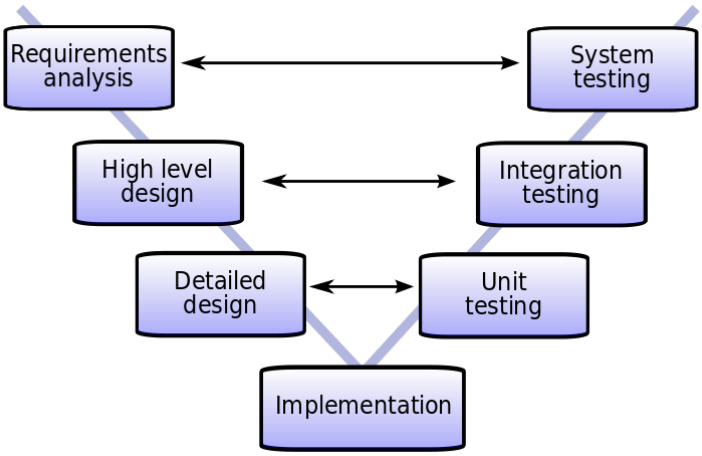
Jedná se o sekvenční model, kde každá sekce musí být kompletně hotová, aby se mohlo pokračovat dál. Dokumentace a testování se píše vždy na konci každé sekce. Potom se sekce uzavře a už se nesmí měnit. Požadavky by měly být specifikovány jasně a zřetelně již na začátku každé sekce, jinak by vznikal chaos v organizaci projektu.

Hlavní výhody tohoto modelu jsou jeho zřetelné požadavky na začátku každé sekce. Protože se jedná o lineární model, je celý proces velice jednoduchý na pochopení a implementaci. Udržení kvality dokumentace je také velice jednoduché při držení se daného modelu.

Hlavní nevýhody vyplývají z jeho výhod. Kvůli jeho linearitě mohou vznikat prostoje, které prodlužují vývojový čas a tím zapříčiňují zvyšující se finanční náročnost celého projektu. Dále je také u většiny projektů problém zjistit všechny informace již na začátku sekce, kvůli tomu se mohou nedostatky nabalovat v dalších fázích projektu. Bez možnosti upravování předchozích fází se z promyšleného projektu může velice rychle stát projekt, kterému nikdo nerozumí. [3]

2.1.2 V-model

V model neboli Validation & Verification model je specifikace, která navazuje na vodopádový model a doplňuje ho. Místo testování, které probíhá pouze v jednom bloku, se testy připravují postupně již od začátku vývoje. Testeři jsou tak zapojeni do celého projektu již od začátku.



Obrázek 2.2 V-model [6]

Výhody jsou podobné jako u vodopádového modelu, tzn. jednoduchá implementace, přesně definované pořadí kroků a relativně přímočará dokumentace projektu. Protože jsou však testy vytvářeny již na začátku, je šance na úspěch celého projektu daleko vyšší.

Nevýhody stejně jako u předchozího modelu představují jeho in-flexibilita k případným změnám, vysoká šance prodlevy, vyšší finanční náročnost a nutnost mít veškeré informace již na začátku projektu. I když jsou však testy vytvářeny již od začátku, nedávají jasné postupy řešení případných chyb. [3] [2]

2.1.3 Agilní model

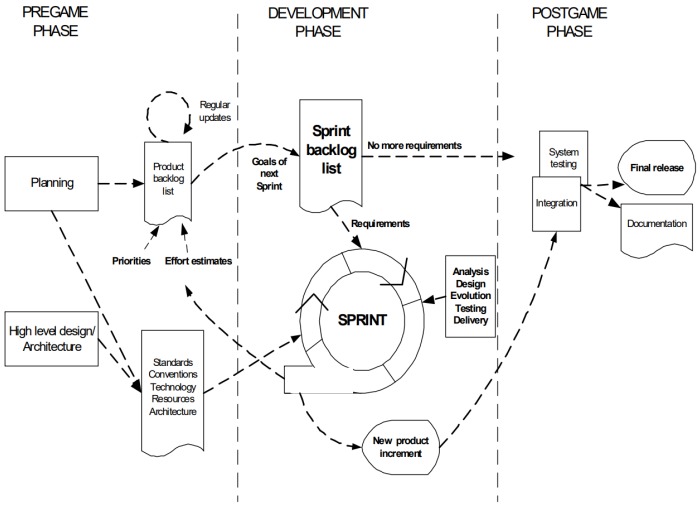
Agilní přístup je nový způsob vývoje, který se snaží nahradit in-flexibilitu lineárních přístupů, vytvářet malé propojené týmy, zvýšit spolupráci, dávat důraz na jednotlivce místo institucionalizovaných procesů a zvýšit týmového ducha. Agilní přístup se řídí několika zásadami, které jsou napsané v agilním manifestu:

- Jednotlivci a interakce před procesy a nástroji
- Fungující software před vyčerpávající dokumentací
- Spolupráce se zákazníkem před vyjednáváním o smlouvě
- Reagování na změny před dodržováním plánu
- Jakkoliv jsou body napravo hodnotné, bodů nalevo si ceníme více. [7]

Dále také obsahuje principy, jimiž by se měl každý agilní tým řídit:

- Naší nejvyšší prioritou je vyhovět zákazníkovi časnýma průběžným dodáváním hodnotného softwaru.
- Vítáme změny v požadavcích, a to i v pozdějších fázích vývoje. Agilní procesy podporují změny vedoucí ke zvýšení konkurenceschopnosti zákazníka.
- Dodáváme fungující software v intervalech týdnů až měsíců, s preferencí kratší periody.
- Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu.
- Budujeme projekty kolem motivovaných jednotlivců. Vytváříme jim prostředí, podporujeme jejich potřeby a důvěřujeme jim, že odvedou dobrou práci.
- Nejúčinnějším a nejefektnějším způsobem sdělování informací vývojovému týmu z vnějšku i uvnitř něj je osobní konverzace.
- Hlavním měřítkem pokroku je fungující software.
- Agilní procesy podporují udržitelný rozvoj. Sponzoři, vývojáři i uživatelé by měli být schopni udržet stálé tempo trvale.
- Agilitu zvyšuje neustálá pozornost věnovaná technické výjimečnosti a dobrému designu.
- Jednoduchost – umění maximalizovat množství nevykonané práce – je klíčová.
- Nejlepší architektury, požadavky a návrhy vzejdou ze samo-organizujících se týmů.
- Tým se pravidelně zamýšlí nad tím, jak se stát efektivnějším, a následně koriguje a přizpůsobuje své chování a zvyklosti. [7]

Z tohoto přístupu vzešlo několik hlavních vývojových postupů, z nichž nejznámější a nejpoužívanější je SCRUM, který zahrnuje 3 základní fáze: předehru, hru a post hru.



Obrázek 2.3 SCRUM model [8]

V předehře se sepíší všechny požadavky, které jsou prozatím známy. Odhadne se jejich pracnost a přidají se do takzvaného backlogu, který se nakonec seřadí podle priority. Požadavky

mohou přicházet od zákazníka, programátora, testera či jiných osob zapojených do vývoje. Do backlogu se mohou konstantně přidávat další požadavky, mohou se ale upravovat a zpřesňovat již vytvořené požadavky. V předehře se také vytvoří architektura programu v závislosti na stávajícím backlogu. V případě nutné změny se identifikují dopady a případné problémy na celý projekt. Tyto změny se pak prodiskutují a z výsledků této diskuze se zavedou případné změny. [8]

Hra se skládá z takzvaných sprintů, což jsou cykly trvající v rozmezí od jednoho do čtyř týdnů a opakující se přes celý vývojový proces. Každý sprint se skládá ze standardních SDLC fází. Existují však důležité milníky ve sprintu, a sice:

- Sprint planning, ve kterém se vyberou úkoly pro celý sprint a rozhodí se mezi programátory.
- Stand-UP, který se opakuje v pravidelných intervalech 1-3 dny, ve kterém všichni členové vývojového týmu řeknou, na čem zrovna pracují a jestli nepotřebují nějak pomoct.
- Na konci sprintu proběhne retrospektiva, kdy celý tým zanalyzuje a zhodnotí daný sprint a nakonec navrhne případné změny pro další sprint.

[8]

Poslední fáze SCRUMu je post hra. Do této fáze se celý tým dostane až tehdy, když nebyly identifikovány žádné další úkoly a celý backlog je prázdný. Celý systém je tak připraven na systémové testování, integraci a vytvoření dokumentace.

Výhody tohoto postupu jsou: krátká doba pro vytvoření výsledku; přímá komunikace mezi zákazníkem a vývojářským týmem; flexibilita pro případné změny v zadání; kontinuální zpětná vazba od zákazníka, která umožňuje zdokonalit celý projekt. Odhad pracnosti je vcelku přesný kvůli krátkému časovému úseku. Díky všem těmto výhodám má celý projekt větší šanci na úspěch. [9]

Hlavní nevýhody tohoto procesu jsou: komplikovaná struktura; nutnost týmové spolupráce, bez které se úspěch projektu neobejde; velká pravděpodobnost nedostačující dokumentace; integrační testy probíhají až na konci, kvůli tomu může dojít k nečekaným problémům. Ale především se tým může zaměřit na jeden aspekt programu a tím nemusí zbýt čas na jiné části projektu. [8]

2.2 Základní architektonické vzory a jejich návrhové vzory

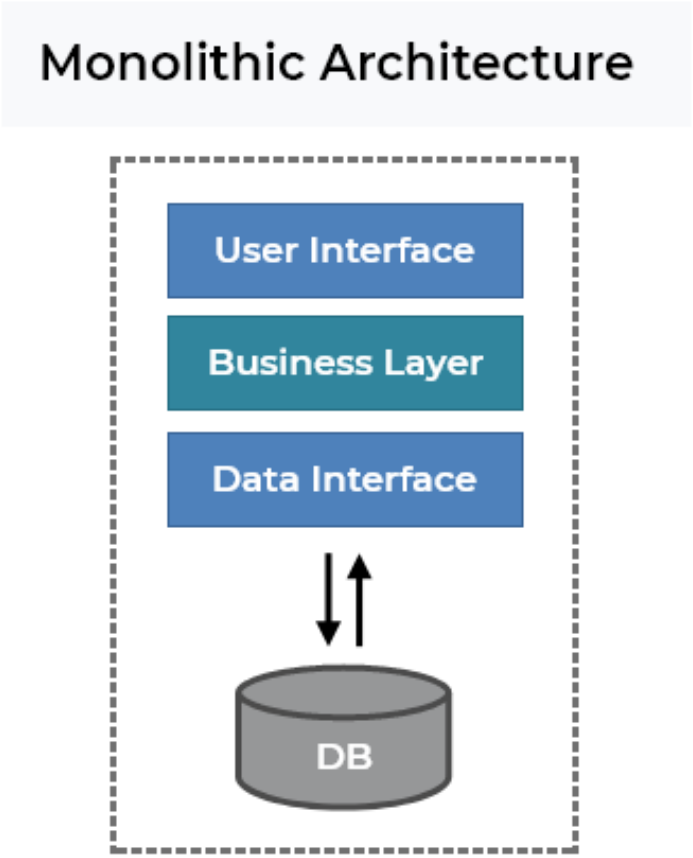
Existují dvě velice odlišné architektury programu: monolitická architektura a mikroslužební architektura. [10] Obě mají svoje výhody a nevýhody. Abychom v budoucnu nemuseli měnit kompletně design aplikace, aby se zamezilo případným problémům a snížily se tak budoucí náklady na údržbu, porovnáme tyto dvě architektury a vybereme pro BillSaver vhodnější variantu v kapitole 4.1.

2.2.1 Monolitická architektura

Monolitická architektura je považována za tradiční styl tvorby aplikací. Celý systém se staví jako jeden nedělitelný celek, jenž obsahuje celou logiku programu. Vnitřně se většinou dělí do logických vrstev, které obsluhují různé části aplikace.

Monolitická architektura se vyznačuje svou provázaností mezi jednotlivými vrstvami aplikace. Celý systém se píše do jediného projektu a je riziko, že tak všechny vrstvy budou silně provázány. Toto způsobuje u velkých projektů komplikace. Změny v jediné komponentě mohou kaskádovat v masivní změny v různých částech celého programu, které tak mohou prodloužit implementační dobu klidně i 10krát. To poté zvyšuje náklady na případné změny a prodlužuje vývojový čas. [10]

Na druhou stranu silná provázanost a přímá komunikace, která probíhá pouze uvnitř aplikace, může vést k lepší rychlosti a celkové odezvě systému. Rozdíl mezi mikroslužební a monolitickou



Obrázek 2.4 Příklad monolitické architektury [11]

architekturou může být až 6%. [12] Studie, která porovnávala rychlosti jednotlivých architektur, však nebrala v potaz škálovatelnost mikroslužební architektury.

Další výhoda monolitické aplikace je její relativně jednoduchá instalace na lokálním stroji, kde bude probíhat vývoj. Programátorovi potřebuje běžet na počítači pouze jedna aplikace, kterou může následně upravovat na jednom místě. Protože však každá změna vede k překompilování celého programu, může vývoj aplikace doprovázet dlouhé prostoje.

2.2.1.1 Výhody

- Snadné nasazení – Monolitická aplikace je pouze jedna, nasazení nové verze aplikace do serveru je tak velice jednoduché.
- Snadná konfigurace – Nastavení celé aplikace se většinou nachází pouze na jednom místě. Není tak potřeba mít dedikovaný systém, v němž se bude sledovat více konfiguračních souborů, stačí pouze jeden.
- Snadné testování – Díky tomu, že celý kód aplikace se nachází na jednom místě, můžeme se spolehnout, že případná chyba bude vždy v daném kódu. Dále také není potřeba spouštět více aplikací, které bychom potřebovali ke zprovoznění naší aplikace.

- Jednoduché monitorování – Monitorování aplikace je velice jednoduché. Není potřeba agregovat více souborů s detailní činností aplikace. Zjišťování stavu aplikace se většinou také nachází na jednom místě.

2.2.1.2 Nevýhody

- Komplexnost – Celý systém se stává komplexním a provázaným velice rychle.
- Velikost týmů – Čím více aplikace nabývá funkci, tím více lidí ji většinou musí udržovat. Domluva ve velkých týmech je neefektivní a dohoda na nových postupech a technologiích je tak pomalá.
- Pomalá kompilace – Čím větší je aplikace, tím narůstá nejen kompilační, ale i testovací čas.
- Pomalá adopce nových technologií – Pro přesun na novou technologii je zapotřebí upravit spoustu existujícího kódu. To navyšuje čas a tím i cenu adopce nových technologií.

2.2.2 Mikroslužební architektura

Jak je z názvu patrné, mikroslužební architektura se dělí na malé části. Každá část neboli služba funguje jako samostatně spustitelný program. Tyto služby zprostředkovávají jednu a pouze jednu logickou službu. Pro příklad můžeme uvést webový obchod, který je postaven na mikroslužební architektuře. Služby by mohly vypadat takto: registrační služba, platební služba, služba pro uživatelské rozhraní, služba pro řízení stavu skladu, účetní služba a mnoho dalších. Velikost jedné služby není přesně definovaná.

Pokud bude služba příliš velká, dostáváme se do stejných problémů jako u monolitické architektury. Pokud bude naopak příliš malá, budeme řešit spoustu problémů s komunikací mezi aplikacemi. Systém se stane celkově neúnosně roztříštěným. [13]

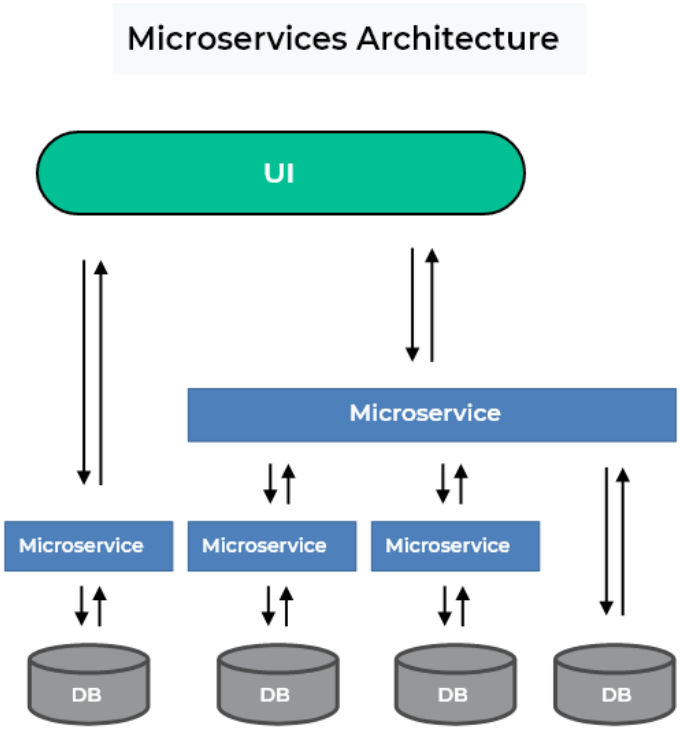
Protože by každá služba měla být nezávislá, můžou se používat různé technologie napříč všemi službami. Toto se však nedoporučuje kvůli případným problémům se znalostí velkého množství technologií. Pokud jeden tým nakonec bude muset udržovat více služeb, může nastat problém s tím, že nezná určité technologie, které jsou v některých službách.

Každá služba může vznikat nezávisle na ostatních. Tím pádem se do produkčního prostředí dají nasazovat také nezávisle. Kvůli velikosti obsahuje každá služba méně firemní logiky, a je tak většinou jednodušší na pochopení a rychlejší na vývoj.

Na rozdíl od monolitické aplikace se však jen těžko dá zabránit nekonzistentnosti dat mezi službami. Monitorování celé aplikace je také náročné kvůli množství nezávislých služeb. Prvotní návrh je složitější a vyžaduje komplexní znalost celé problematiky. [13] Dokonce vznikají i nové problémy, které vůbec neexistovaly v monolitické architektuře. Zajištění komunikace mezi službami může být náročné, protože se musí řešit i síťové problémy. Je nutné zajištění automatického nasazení služeb, bez něhož by to vyžadovalo neúměrné úsilí od administrátorů.

2.2.2.1 Výhody

- Flexibilita – Vývoj úplně nové funkcionality je velice jednoduchý kvůli neexistujícímu stávajícímu kódu. Vždy se začíná od začátku. [13]
- Jednoduchá adopce nových technologií – V případě, že jedna služba chce adoptovat novou technologii, není zapotřebí upravovat velké množství kódů. Toto však neplatí při adopci technologie napříč vícero službami.
- Robustnost – Pád jedné služby neznamená kolaps celé aplikace.
- Rychlý vývoj – Po vytvoření aplikačního rozhraní můžou týmy nezávisle pracovat na různých službách.



Obrázek 2.5 Příklad mikroslužební architektury [11]

- Rychlá kompilace – Kvůli velikosti jedné služby není potřeba kompilovat velké množství kódů.
- Jednoduché porozumění – Protože jedna služba dělá pouze určitou funkci, její obchodní logika není tak rozsáhlá a pochopení služby je jednodušší a rychlejší.
- Jednoduchá škálovatelnost – Architektura celého systému umožňuje nezávislou replikaci jednotlivých služeb. Například v případě vánoční nákupní horečky se může platební systém replikovat a odbavit tak více uživatelů najednou.
- Schopnost používat tu nejlepší technologii pro danou činnost – Každá služba může používat specializovanou technologii, která nejlépe řeší její problematiku. [13]

2.2.2.2 Nevýhody

- Obtížné testování – Pro testování je zapotřebí procházet více aplikací. Nastavení testovacího scénáře vytváří nutnost spouštění více služeb najednou. [13]
- Nutnost řešit datovou konzistentnost – Protože každá služba může vlastnit databázi, je zapotřebí informovat ostatní služby o změnách v uživatelských datech. [10]
- Velice obtížné zachování ATOM-icity – V případě chybného správcování je zapotřebí zrušit uživatelský požadavek mezi službami. [11]
- Obtížná komunikace mezi službami – Každá služba musí řešit komunikaci přes jednotné rozhraní a tak musí i řešit případné chyby v síťové komunikaci.

- Obtížný návrh služeb – Vytvořit správné rozvržení služeb vyžaduje komplexní znalost problematiky, která nemusí být na začátku celkově známá. [10]

2.2.3 Porovnání architektur

Monolitická aplikace		Mikroslužební aplikace	
Klady	**Zápory**	**Klady**	**Zápory**
Snadné nasazení	Komplexnost	Flexibilita	Obtížné testování
Snadná konfigurace	Velikost týmů	Jednoduchá adopce nových technologií	Nutnost řešit datovou konzistentnost
Snadné testování	Pomalá kompilace	Robustnost	Velice obtížné zachování ATOM-icity
Jednoduché monitorování	Pomalá adopce nových technologií	Rychlý vývoj	Obtížná komunikace mezi službami
		Rychlá kompilace	Obtížný návrh služeb
		Jednoduché porozumění	
		Jednoduchá škálovatelnost	
		Schopnost používat tu nejlepší technologii pro danou činnost	

Tabulka 2.1 Porovnaní jednotlivých architektonických stylů

2.3 Designové vzory

Aplikace, která by měla být používaná komunitou lidí od programátorů až po koncové uživatele musí splňovat jednu důležitou vlastnost – flexibilitu. Když mluvíme o flexibilitě, myslíme tím schopnost měnit svůj vzhled, design a funkcionalitu v závislosti na požadavcích uživatelů, majitelů a případné konkurence. Tyto informace následně zužitkujeme v kapitolách 4.1 a 5 kde na základě těchto informací budeme vytvářet aplikaci BillSaver.

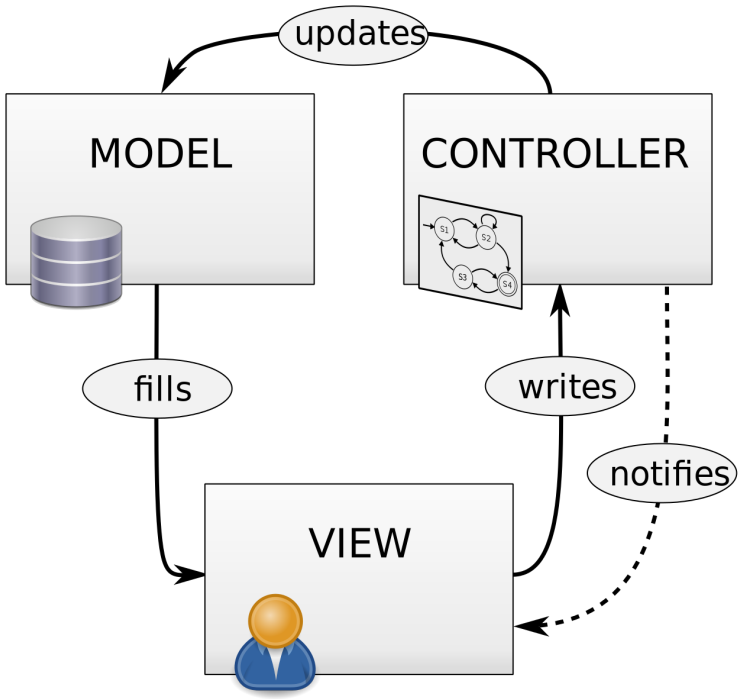
Aby byla aplikace flexibilní, musí být navržená tak, aby splňovala tyto vlastnosti:

- Rozšiřitelnost – Jednoduše přidávat funkcionalitu, tak aby nevznikla přílišná závislost na jiných částech aplikace.
- Udržitelnost – Schopnost upravovat, opravovat a odebírat již existující funkcionalitu, tak abychom neničili jinou funkcionalitu.
- Srozumitelnost – Aplikace by měla být napsaná a popsaná takovým stylem, že není pro kohokoliv problém, aby ji jednoduše, rychle a správně upravil.

Kvůli těmto třem vlastnostem se budeme převážně zaměřovat na designové vzory, které se dají propojit s mikroslužební architekturou. Pro podrobnější analýzu si důvody volby mikroslužební architektury rozebereme v sekci 3.3.

2.3.1 MVC

MVC je návrhový vzor, který dělí jednu službu na 3 hlavní funkční celky – model, view a controler. [14] MVC poté komunikuje přes standardní kanály mikroslužební architektury.



Obrázek 2.6 Diagram MVC architektury [15]

2.3.1.1 Model

Model obsahuje doménovou problematiku celé aplikace. Bude se tedy skládat z entit, které modelují a podporují základní problematiku, proto zůstávají vcelku neměnné. Musíme dodat, že entity v modelu o ostatních vrstvách nic nevědí.

2.3.1.2 View

View, přeloženo pohled, znázorňuje, jak budou data reprezentovaná uživateli či jinými aplikacemi. Těchto pohledů může být víc, například:

- GUI pohled
- API pohled
- Logovací pohled

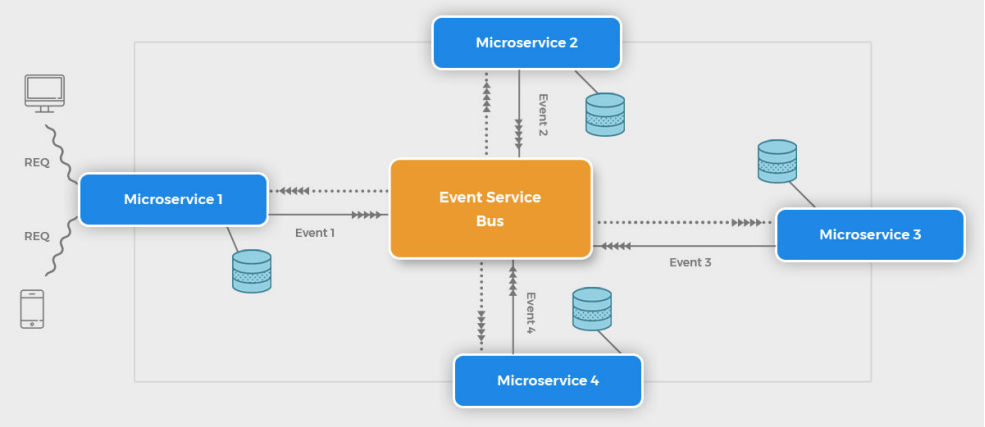
Představme si pohled, který vyřizuje obchodní objednávku. Obsahuje v sobě množství objednaných produktů. Tento pohled potom musí upravit počet produktů, jež patří uživateli, který kupuje dané zboží. Proto pohled potřebuje vědět o existenci modelu.

2.3.1.3 Controller

Controller dostává pokyny od uživatele nebo jiných aplikací a na jejich základě vyhodnotí a provede dané úkony. Podle změn v modelu upraví pohled a informuje uživatele. [14]

2.3.2 EDA

EDA neboli Event-driven architecture je architektura, která produkuje, konzumuje a reaguje na něco, čemu se říká event nebo také událost. Služby v EDA spolu komunikují přes tyto události.



Obrázek 2.7 Diagram EDA [16]

Událost je informace, která v sobě nese různé notifikace o změnách stavu. Události se rozdělují na tři základní typy.

- Normální události – Události, které v sobě nesou informace běžné a očekávané systémem. Například – objednávka byla zaplacena.
- Abnormální očekávané události – Události, které systém očekává, ale jen zřídka. Například – objednávka nebyla zaplacena.
- Neočekávané události – Události, na něž nedokážeme reagovat. Například – platební terminál shořel. [17]

Události se skládají ze dvou částí – hlavičky a těla. Hlavička obsahuje informace o události, jako jsou její identifikační číslo, typ události, časová značka události a autor události. Hlavička by měla být standardizovaná napříč systémem.

Tělo události obsahuje specifické informace o daných změnách. Události musí obsahovat všechny informace o dané události. Systémy by se neměly zpětně ptát na detailní informace. [18]

Každá událost začíná v generátoru událostí. To může být uživatel, jiná aplikace, senzor nebo i twitter zpráva. Generátor události ji upraví do předem definovaného standardu a nakonec ji pošle do kanálu událostí.

Kanál událostí má za úkol přenést tuto zprávu konzumentům. [18] Často má i garantovat doručení zprávy všem konzumentům alespoň jednou, zaznamenávat zprávy, které přesouval, a nahlásit případnou chybu, když zpráva nedorazí, nebo ji konzument nepřijme. [19]

Procesor zpráv nebo také konzument má za úkol zprávu zpracovat, v případě potřeby zahájit nějakou akci nebo generovat další událost.

2.4 Aplikační rámce

Abychom se nemuseli starat o implementační detaily, ale raději se mohli soustředit na obchodní logiku, využíváme aplikační rámce. Jedná se o strukturu, kterou využíváme na vyřešení obecné

funkcionality. Rámce nám pomáhají s připojením do databáze, zabezpečením aplikace, jednodušším využíváním designových vzorů, testováním, udržením konzistentity napříč aplikací a mnoho dalších věcí. [20] Vyberme zde ten nejvhodnější rámec pro BillSaver a následně jej využijeme v kapitolách 4.1 a 5 pomoci tohoto rámce budeme vytvářet aplikaci BillSaver.

Protože je Java jedním z nejvyužívanějších programovacích jazyků, který doporučují lidé pracující ve finančním oboru, [21, 22, 23, 24, 25, 26] zaměříme se zde na aplikační rámce pro Javu.

Na základě průzkumu provedeného v roce 2018 a 2021 jsou pro Javu nejpopulárnější rámce Spring Boot, Spring MVC, Java EE a Quarqus. [27, 28] Uvedeme si zde jejich výhody a rozdíly. Protože Spring Boot a Spring MVC patří pod jeden celkový rámec, budeme jej zde vnímat jako jeden celek. Dále zde nebudeme mluvit o Java EE kvůli tomu, že Spring je doplněk Java EE. [29]

2.4.1 Spring

Spring je nejpopulárnější rámec na světě s více jak 76 % java programátorů, využívajících tento aplikační rámec na tvorbu aplikací. [28] Je založen na Java EE, s nejnovější verzí 5.0 vyžadující Java EE 7 specifikaci.

Spring je složen z několika projektů, které spolu dohromady tvoří Spring Framework. Každý projekt má za úkol usnadnit tvorbu nějakého aspektu při vývoji aplikace. Zde je pouze několik příkladů vyhovujících našemu problému, tvorbě BillSaveru:

- Spring boot – Pomáhá s integrací jiných rámců. A definuje standardní konfigurace při tvorbě aplikací.
- Spring for Apatche kafka – Rámec, který spojuje aplikaci s jinými službami pomocí asynchronních událostí. A umožňuje tak jednoduchou implementaci EDA.
- Spring MVC – Umožňuje jednoduše vytvářet aplikace s MVC architekturou.
- Spring Repository – Umožňuje jednoduše propojovat aplikaci s databází.
- Spring Security – Automaticky zabezpečuje jednotlivé aplikace a stará se o bezpečnou komunikaci mezi jednotlivými službami v mikroslužební architektuře.
- Spring Restdocs – Zjednodušuje vytváření dokumentace pro jednotlivé služby. [29]

2.4.2 Quarkus

Quarkus je relativně nový rámec založený v roce 2019 od společnosti RedHat. [30] Stejně jako Spring i Quarkus je rozdělen na několik projektů, z nichž každý se snaží usnadnit tvorbu nějakého aspektu při vývoji aplikace.

Rozdíl oproti Spring představuje jeho zaměření na cloudové řešení. Je postaven pro mikroslužební architekturu. Dále se snaží zaměřit na nekonfigurační přístup pro vývoj, chce tím usnadnit a zrychlit vývoj aplikace. [31]

2.4.3 Spring x Quarkus

Shrneme si zde výhody jednotlivých aplikačních rámců.

I přes jejich rozdíly se jedná o velice podobné aplikační rámce, které se snaží docílit stejné věci, zrychlit a zefektivnit tvorbu aplikací. Jejich nejdůležitější parametry, jako jsou rychlost a podpora podpůrných aplikačních rámců, jsou velice podobné. Není tedy úplně těžké vybrat jeden, či druhý ekosystém. Opravdu se ale jedná o ekosystém, není tedy vhodné kombinovat tyto dva druhy aplikačních rámců mezi sebou. Protože potřebujeme co nejméně komplikovaný vývoj a díky jeho popularitě budeme v následujících kapitolách, především 4.1 a 5, používat Spring.

Spring	Quarkus
Rozsáhlejší dokumentace	Rychlá kompilace při vývoji
Lepší znalost mezi programátory	Menší velikosti aplikací po kompilaci
Zralý aplikační rámec	Konzumuje méně paměti
Větší komunita	Rychlejší start aplikace

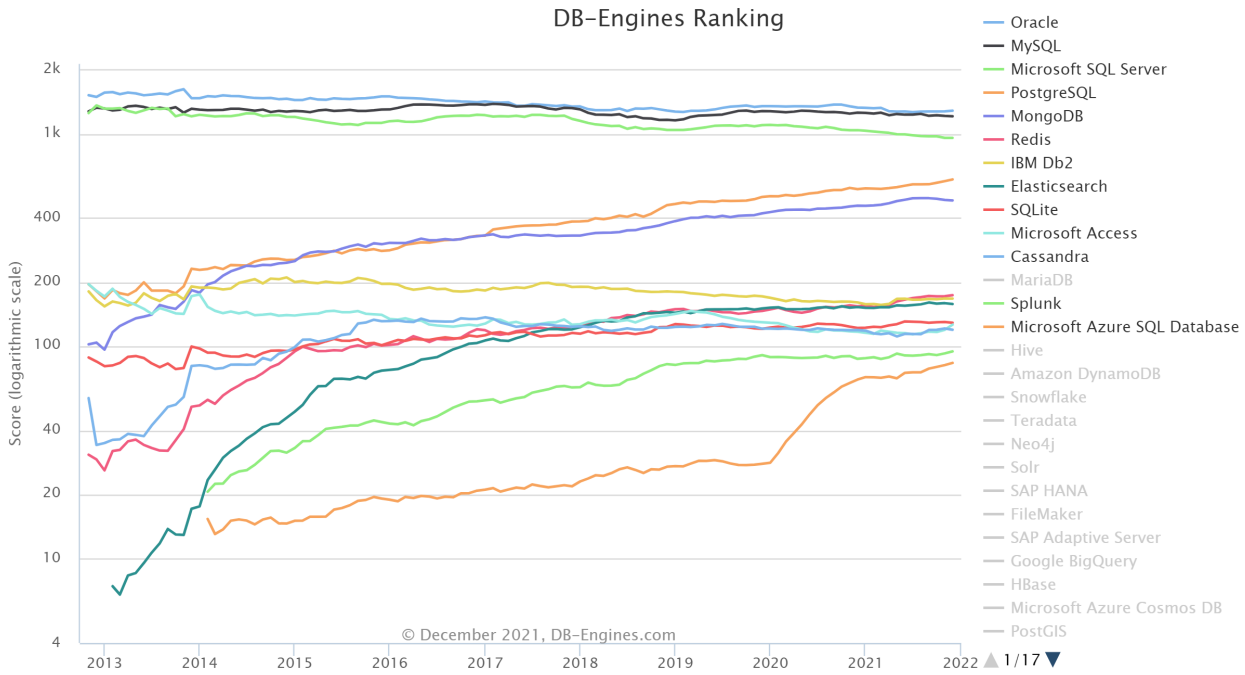
Tabulka 2.2 Porovnaní Spring Boot a Quarkus

2.5 Databáze

Databáze slouží k bezpečnému uchovávání a vyhledávání dat. Protože vytváříme systém, který bude používaný ve finančním sektoru, nechceme volit databáze, jež jsou obskurní, mají špatnou kompatibilitu s aplikačními rámci, špatně se s nimi pracuje, mají špatnou rychlost a nebo jsou zastaralé, tedy nezabezpečené. Kvůli této podmínce budeme vyhledávat pouze databáze neboli DBMS splňující všechna tato kritéria. Abychom si ušetřili čas, vybereme nejpoužívanější DMBS, rozebereme si, v čem se liší a na jaké věci se používají. Díky tomuto rozdělení využijeme ten nejvhodnější DBMS pro BillSaver v kapitolách 3.3 a 4.1.

Podle DBMS žebříčku od společnosti Solid IT 10 nejpopulárnějších DMBS systémů využívá jeden z těchto DBMS principů: relační, dokumentové, key-value a Search engine. [32]

Tyto čtyři DBMS se dají rozdělit do dvou hlavních skupin: SQL a NO-SQL databáze. Do SQL DBMS se řadí pouze relační databáze a do no-sql se řadí zbytek, tedy dokumentové, key-value a Search engine databáze.



Obrázek 2.8 Skóre jednotlivých DBMS [32]

2.5.1 Relační databáze

Relační databáze jsou jedním z nejpopulárnějších přístupů ke skladování aplikačních dat. Můžeme to vidět i na grafu 2.8, kde za posledních 9 let byly 3 nejpopulárnější DBMS právě relační.

Kvůli popularitě si relační DBMS udržují nejrozšířenější znalosti ve vývojářské komunitě. Podpora těchto systémů je také velice rozsáhlá a kvůli jejich rozšířenosti ve většině bankovních sektorů můžeme předpokládat, že mají velice robustní zabezpečení. [33]

Relační databáze disponují čtyřmi základními operacemi CRUD neboli: vytvořit (create), číst (read), editovat (update), smazat (delete). Tyto operace využívají programovací jazyk SQL neboli Structured Query Language.

Nevýhodou celého systému je nutnost definování celé datové struktury před provedením jakékoliv operace.To je může dělat in flexibilní pro účely, kde máme hodně různorodých dat a nebo kde se data můžou často měnit.

2.5.2 Dokumentové databáze

Dokumentové databáze se skládají z takzvaných dokumentů. Jeden tento dokument reprezentuje jeden záznam v databázi. Data se ukládají v podobě JSON textu, nevyžadují tak před-vytvoření datového schématu. Kvůli tomu mohou být data v průběhu doplňovaná a upravovaná bez ohledu na ostatní záznamy.

Hlavní výhoda, a to možnost upravovat datové schéma bez ohledu na jiné záznamy, je však i její velká nevýhoda. Uživatel se nemůže spolehnout na data, která bude vyhledávat. Na rozdíl od relačních databází ne všechny dokumentové databáze podporují ACID vlastnosti transakcí. ACID je zkratka, skládající se ze slov: atomicita (atomicity), konzistence (consistency), izolovanost (isolation), trvalost (durability). Zkráceně to znamená, že v případě selhání vložení jednoho záznamu by se neměly vložit ani ostatní záznamy, které jsou ve stejné transakci. [32]

2.5.3 Key-value databaze

Tento typ databáze ukládá svá data jako seznam klíčů a hodnot k těmto klíčům. Není tak možné vyhledávat jednotlivé záznamy na základě jejich vlastností. Celé vyhledávání se děje přes hlavní klíč, což vede k velice rychlému a efektivnímu vyhledávání.

Nevýhodou tohoto přístupu je, jak již bylo zmíněno, nemožnost vyhledávání podle jiných vlastností než podle hlavního klíče.

2.5.4 Search engine databáze

Search engine databáze je nadstavba různých NO-SQL databází a její vnitřní implementace záleží na jednotlivých produktech, které implementují tento DBMS.

Typicky nabízí komplexní vyhledávací metody, vyhledávání v textu a distributivnost pro rozsáhlé množství dat.

Jejich nevýhodami jsou typicky nepodporované ACID vlastnosti a vysoká náročnost na výkon kvůli jejich rozsáhlé indexaci záznamu.

Kapitola 3
Studie proveditelnosti

3.1 Popis projektu a jeho etap

3.1.1 Výchozí stav

Pokud chce mít spotřebitel celkový přehled o své finanční situaci, musí skladovat účtenky, extrahovat z nich informace a ty zaznamenávat v nějaké mobilní aplikaci k tomu určené, v horším případě v excelu. Toto úsilí zabírá čas, což snižuje pravděpodobnost pravidelně tyto informace zaznamenávat a na základě výsledků provést potřebné kroky ke zlepšení finanční situace spotřebitele.

Další problém nastává s produkty, které mají záruku. Od nich je nutné uchovávat účtenky, kterými musí spotřebitel prokazovat koupi daného produktu. Účtenky jsou však tisknuty na papír, který po čase tmavne a degraduje. Tím pádem musí spotřebitel tyto účtenky správně skladovat, nebo převádět do elektronické podoby.

3.1.2 Finální stav

Spotřebitel si založí účet na webové stránce. Ten si poté spojí se svými bankovními účty a půjde na nákup. Zaplatí mobilem, kreditní nebo debetní kartou a jde domů bez převzetí jakéhokoliv dokladu o platbě. Během chvilky spotřebiteli přijde upozornění na mobil o provedené platbě. V případě zájmu si otevře svoji aplikaci, kde najde:

- Údaje o provedené platbě s datem platby, celkovou cenou zboží, EET a jinými identifikačními údaji.
- Položkový seznam zboží, které nakoupil.
- Na základě své nákupní historie spotřebitel dostane doporučení na změny svých nákupních návyků pro celkové ušetření času a peněz.
- V případě konzumního zboží aplikace ukazuje kalorické a výživové údaje. V případě základních potravin i typy na přípravu rychlého, jednoduchého a zdravého jídla.
- Pokud zboží má státem garantovanou záruční dobu, zobrazují se kontakty a odkazy na případné reklamační oddělení.
- Jedná li se o nábytek, elektroniku, zahradní nářadí či jiné zboží, jež by v sobě obsahovalo návody, bezpečnostní instrukce či jiné příbalové letáky, tyto informace jsou poskytnuty ve formě PDF dokumentů v aplikaci.

- Spotřebitel také může vyhledávat stejné či podobné produkty podle klíčového slova, kde dokáže zjistit užitečné informace. Příklad: Spotřebitel jede na velký nákup. Potřebuje nakoupit mléko na další měsíc, protože je v akci. Podívá se do aplikace, vyhledá mléko a aplikace mu řekne, že minulý měsíc nakoupil $15 * 1,5l$ mléčných kartonů. Spotřebitel tak ví, že má nakoupit $22.5l$ mléka.

V aplikaci by se postupně objevovalo více funkcí na základě zpětné vazby od uživatelů.

Obchodník si musí ke svému prodejnímu systému připojit aplikaci, která spojuje systém BillSaver spolu se systémem obchodníka. Tato aplikace může běžet jak u obchodníka, tak na systémech BillSaveru za určitý měsíční poplatek. Obchodník se tak nemusí starat o zálohování svých dat, která si vždy může stáhnout z BillSaveru. Může se také dozvědět informace o svých zákaznících, například:

- Věková skupina zákazníků, pohlaví či jiné údaje, které zákazník vyplní při registraci nebo které systém vydedukuje, například koníčky zákazníků, jídelní preference a podobně.
- Nejoblíbenější produkty prodejce.
- Průměrná částka za nákup.
- Reklamovatelnost zboží.
- Nejčastější způsob platby.

V případě zájmu si obchodník může objednat další statistické a analytické služby. Obchodník se tak nemusí starat o zákaznické karty, protože informace o zákaznících dostane od BillSaveru. Místo zákaznických karet, které zákazníci mohou, či nemusí použít, BillSaver funguje automaticky pro každou platbu. Obchodník také může zavést slevy na určité zboží pro své věrné zákazníky.

3.1.3 Souhrnná informace o projektu

- Projekt řeší analýzu skladování a extrahování informací z účtenek, tak aby vyextrahované informace byly přínosné jak pro zákazníka, tak pro obchodníka.
- Systém dodává relevantní doplňkové informace k jednotlivým produktům.
- Celý systém je také lehce rozšiřitelný, aby se k němu dokázala přidávat další funkcionalita.
- Projekt se zaměřuje na mladší až středně staré věkové skupiny, které inklinují k automatizaci a ulehčení práce pomocí technologií.
- Dále se projekt zaměřuje na malé až středně velké obchodníky, kteří by potřebovali zjistit více informací ohledně svých zákazníků.

3.1.4 Lokalizace projektu

Pro jednoduchou správu celého systému a finanční nenáročnost bude systém běžet v cloudu u jedné ze společností Google cloud, AWS a nebo Azure. Tyto společnosti byly vybrány na základě jejich spolehlivosti, bezpečnosti, jednoduchosti škálování a dobré znalosti ve vývojářské komunitě.

Pro flexibilní výběr talentovaných lidí z České republiky a nebo sousedních zemí, snížení finančních nákladů a ověřenosti daného přístupu přes pandemii SARS-CoV-2 bude celý tým pracovat vzdáleně.

Pro sídlo firmy využijeme služeb od jedné ze společností, nabízejících virtuální firemní sídla.

3.1.5 Fáze projektu a jeho harmonogram

Projekt je rozdělen do 5 fází, jež by měly kopírovat SDLC, viz 2.1. Plánování a analýza by patřily do předinvestiční fáze. Návrh a implementace by patřily do investiční fáze, údržba spolu s kontinuálním vývojem by patřily do provozní fáze.

3.1.5.1 Fáze předinvestiční

1. Plánování
   - **Tvorba studie proveditelnosti.**
   - **Založení vývojového týmu.**
   - **Vymezení styčných cílů.**
   - Vymezení cílové skupiny.
   - Vydefinování buisness potřeb.
   - **Identifikování budoucích problémů.**
   - **Výběr vhodných technologií.**
   - Ohraničení buisness cílů.
   - Zanalyzování zájmu o BillSaver z pohledu spotřebitelů a obchodníků.
2. Analýza
   - Provedení rešerše podobných projektů.
   - **Klasifikování problémů, které mohou nastat, a navrhnout k nim řešení.**
   - **Poukázat na nezbytné části systému.**

Protože plánování a analýza velice úzce souvisí s druhou bakalářskou prací od Anastasiie Bernikové, na kterou tato práce navazuje. Musíme si zde velice jasně vymezit rozsah. Protože se tato bakalářská práce zaměřuje především na technickou implementaci BillSaveru, nebudeme se okruhy jako například vymezení cílové skupiny, zanalyzování zájmu o BillSaver, provedení rešerše podobných projektů atd., zabývat. Budeme se zde věnovat pouze tučně vyznačeným bodům, a to převáženě po technické stránce. Plánování a analýza jsou vytvořeny v gant diagramu, viz obrázek: 3.1

3.1.5.2 Fáze investiční

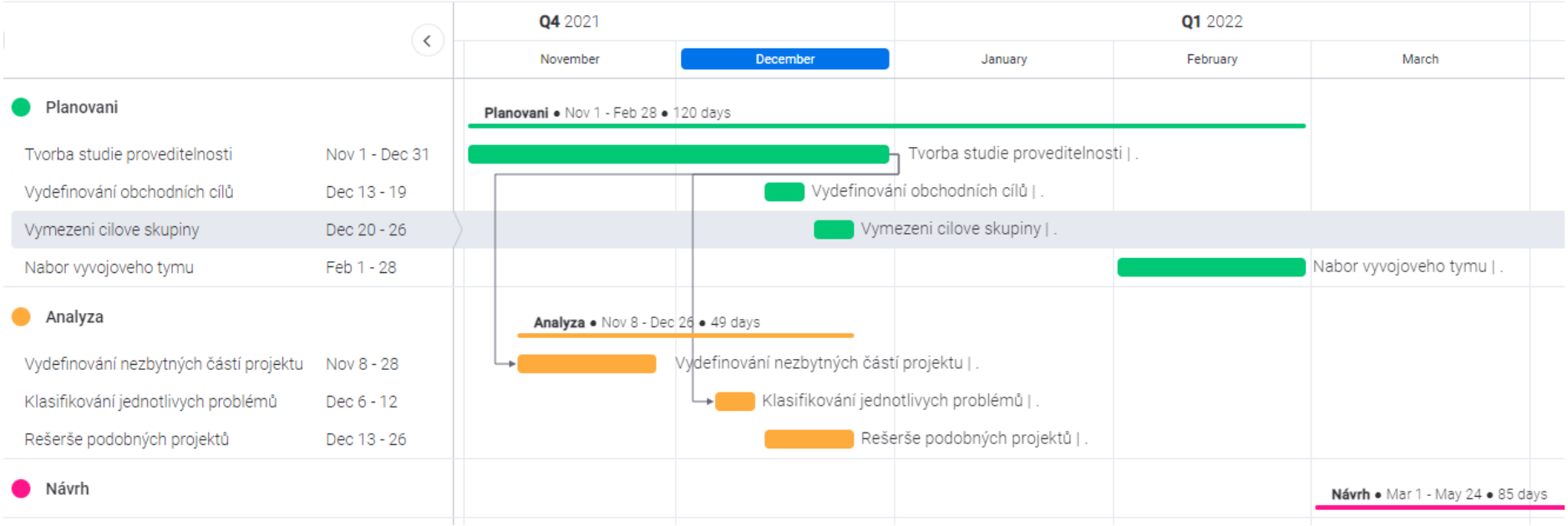
1. Návrh
   - Zanalyzování dat, která se budou uchovávat v databázi.
   - Vytvoření databázového modelu.
   - Zanalyzování interakce buisness entit mezi sebou.
   - Navržení komunikačního rozhraní jednotlivých buisness entit.
   - Návrh uživatelského rozhraní.
2. Implementace
   - Na základě analýzy vytvořit seznam úkolů.
   - Vytvoření datového modelu na základě databázového modelu.
   - Na základě designu aplikace vytvořit implementaci.
   - Otestovat aplikace.

Pro podrobnější návrh jednotlivých servis, rozdělení komunikace, odůvodnění, proč jsme použili zrovna Microservice architecture a dalších implementačních detailů se prosím obraťte na kapitolu3.3. Zde si pouze ukážeme harmonogram práce. Návrh je vytvořen v gant diagramu, viz obrázek:3.2Pro rozsáhlost implementační části jsme museli rozdělit gant diagram na dvě části, viz obrázky: 3.3, 3.4

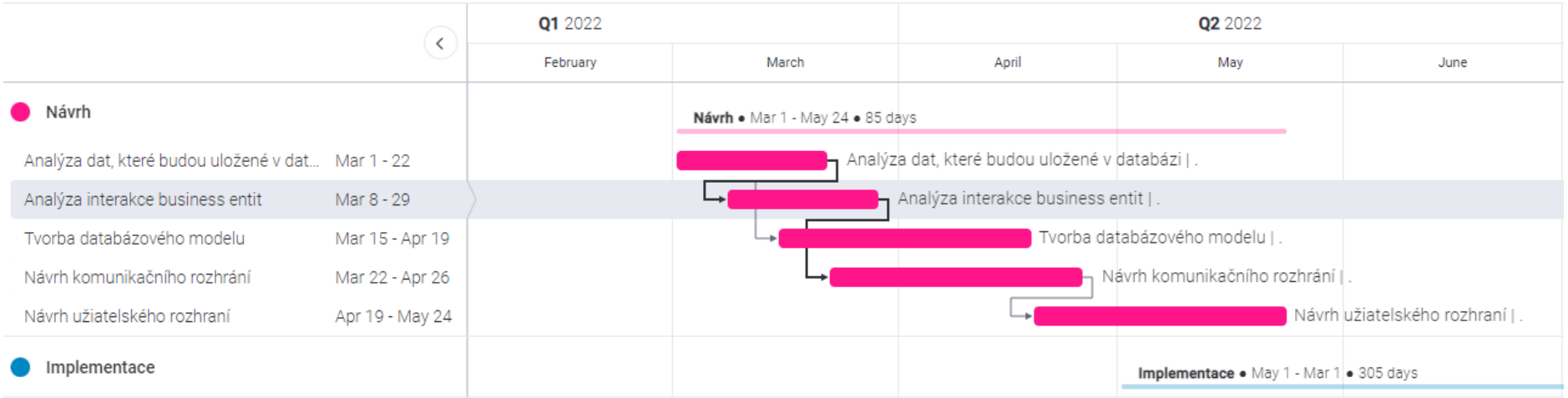
3.1.5.3 Fáze provozní

Tato fáze nebude mít sadu úkolů, které budou mít začátek a konec. Nebudeme zde tak tvořit harmonogram práce. Pouze zde popíšeme jednotlivé pozice, které se budou starat o správný chod celé aplikace.

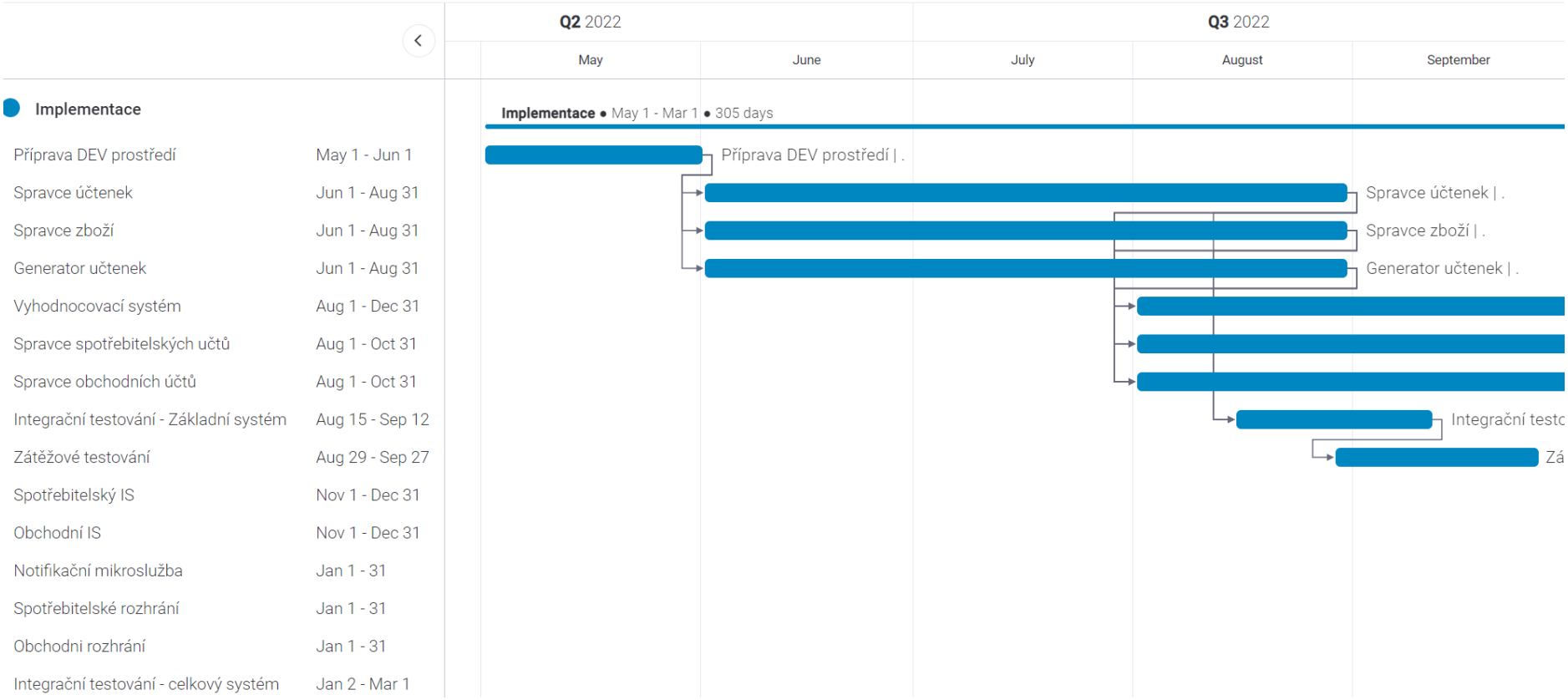
- Admin tým
  1. Monitorování stavu všech servis.
  2. V případě výskytu chyby buď chybu vyřešit, nebo eskalovat problém na technický tým.
- Technický tým
  1. Řešit a opravovat vzniklé chyby v systému.
  2. Vyvíjet novou funkcionalitu na základě připomínek zákazníků, PR nebo buisness partnerů.
- PR tým
  1. Zahajovat a vyhodnocovat promoční kampaně.
  2. Zjišťovat a zajišťovat zpětnou vazbu od zákazníků.
  3. Navrhovat novou funkcionalitu, kterou by zákazníci využili.
  4. Nábor nových obchodníků do systému.
- Support tým
  1. Řešení problémů spotřebitelů.
  2. Řešení problémů obchodníků.
  3. Případná komunikace s admin, technickým a nebo PR týmem.



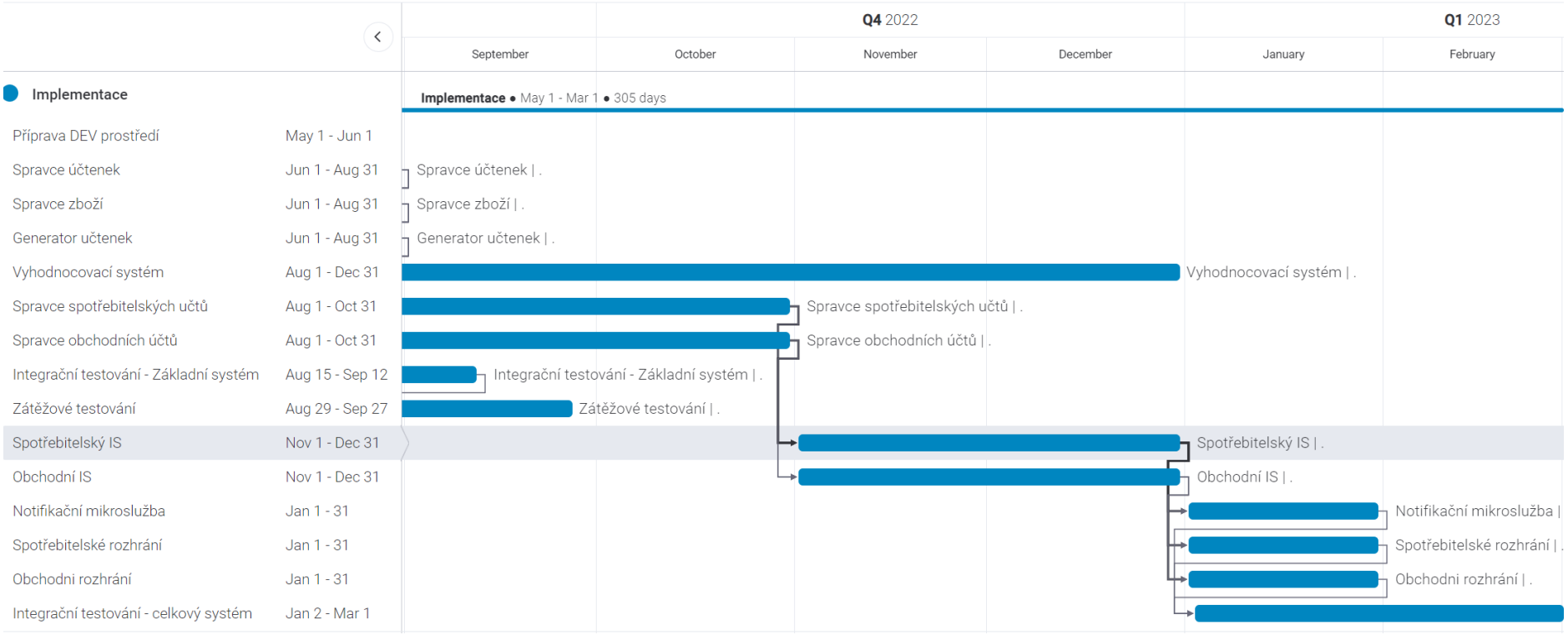
Obrázek 3.1 Gant diagram s plánováním a analýzou



Obrázek 3.2 Gant diagram návrh



Obrázek 3.3 Gant diagram implementace část 1



Obrázek 3.4 Gant diagram implementace část 2

3.2 Lidské zdroje

V teto kapitole si rozepíšeme pracovní kapacity které budou potřeba k dokončení jednotlivých fázi. Ke každé pozici přikládáme její odpovědnost a kompetence, procentuální úvazek, druh pracovní smlouvy, a očekávaná hrubá mzda či MD sazba.

Týmové složení						
Pozice v týmu	Jméno	Název organizace	Odpovědnost a kompetence člena týmu	Úvazek na projektu v %	Druh pracovně právního vztahu	Hrubá mzda (PS) nebo MD sazba (Kontrakt)
Před-investiční fáze						
Architekt	Patrik Malý	BillSaver	Vytvořit prvotní návrh celého systému	100	Pracovní smlouva	80 000
Byznys Konzultant	Anastasiia Berníková	BillSaver	Definovat byznys potřeby projektu	100	Pracovní smlouva	80 000
Investiční fáze						
Architekt	Patrik Malý	BillSaver	Koordinovat vývoj mezi týmy, dohlížet na dodržování směru vývoje a případně upravovat architekturu systému	100	Pracovní smlouva	80 000
Byznys konzultant	Anastasiia Berníková	BillSaver	Jednat s případnými obchodníky s napojením do BillSaveru, řešit požadavky obchodníků a spotřebitelů a podílet se na vytváření reklamních kampaní	100	Pracovní smlouva	80 000
HR / PR	Není relevantní	BillSaver	Starat se o nábor pracovníků, řešit jejich případné stížnosti, vytvářet reklamní kampaně, podílet se na jednání s obchodníky	100	Pracovní smlouva	50 000
Účetní	Není relevantní	Externí firma	Vyplácet mzdy vytváření faktur, zaúčtovávání nákladů a řešení finančních problémů.	20	Kontrakt	3 500
DevOps / Admin	Není relevantní	BillSaver	Vytvoření a starání se o celý CI/CD proces a starání se o testové, preprod a prod prostředí.	100	Pracovní smlouva	70 000
Senior developer – tým 1	Není relevantní	BillSaver	Starat se o hlavní vývoj v týmu 1, pomáhat kolegům, pořádat všechny SCRUM procesy, které jsou potřeba, a starat se o dodržování postupů.	100	Pracovní smlouva	80 000
Medior developer – tým 1	Není relevantní	BillSaver	Vyvíjet naplánovanou funkcionalitu, řešit chyby, které zjistí tester	100	Pracovní smlouva	60 000
Medior developer – tým 1	Není relevantní	BillSaver	Vyvíjet naplánovanou funkcionalitu, řešit chyby, které zjistí tester	100	Pracovní smlouva	60 000

Tester – tým 1	Není relevantní	BillSaver	Vytvářet testové scénáře a testovat funkcionalitu služeb	100	Pracovní smlouva	50 000
Senior developer – tým 2	Není relevantní	Externí firma	Starat se o hlavní vývoj v týmu 2, pomáhat kolegům, pořádat všechny SCRUM procesy, které jsou potřeba, a starat se o dodržování postupů.	100	Kontrakt	4 000
Medior developer – tým 2	Není relevantní	Externí firma	Vyvíjet naplánovanou funkcionalitu, řešit chyby, které zjistí tester.	100	Kontrakt	3 000
Medior developer – tým 2	Není relevantní	Externí firma	Vyvíjet naplánovanou funkcionalitu, řešit chyby, které zjistí tester	100	Kontrakt	3 000
Tester – tým 2	Není relevantní	Externí firma	Vytvářet testové scénáře a testovat funkcionalitu služeb	100	Kontrakt	2 500
Provozní fáze						
Architekt	Patrik Malý	BillSaver	Koordinovat vývoj mezi týmy, dohlížení na dodržování směru vývoje a v případě upravovat architekturu systému	100	Pracovní smlouva	80 000
Byznys konzultant	Anastasiia Berníková	BillSaver	Jednat s případnými obchodníky s napojením do BillSaveru, řešit požadavky obchodníků a spotřebitelů a podílet se na vytváření reklamních kampaní	100	Pracovní smlouva	80 000
HR / PR	Není relevantní	BillSaver	Řešit problémy zákazníků, řešit problémy zákazníků, vytvářet reklamní kampaně, podílet se na jednání s obchodníky	100	Pracovní smlouva	50 000
PR	Není relevantní	BillSaver	řešit problémy zákazníků, vytvářet reklamní kampaně, podílet se na jednání s obchodníky	100	Pracovní smlouva	50 000
Účetní	Není relevantní	Externí firma	Vyplácet mzdy, vytvářet faktury, zaúčtovávání nákladů a řešení finančních záležitostí.	20	Kontrakt	3 500
DevOps / Admin	Není relevantní	BillSaver	Vytvoření a starání se o celý CI/CD proces a starání se o testové, preprod a prod prostředí.	100	Pracovní smlouva	70 000
Senior developer – tým 1	Není relevantní	BillSaver	Starat se o hlavní vývoj v týmu 1, pomáhat kolegům, pořádat všechny SCRUM procesy, které jsou potřeba, a starat se o dodržování postupů.	100	Pracovní smlouva	80 000
Medior developer – tým 1	Není relevantní	BillSaver	Vyvíjet naplánovanou funkcionalitu, řešit chyby, které zjistí tester.	100	Pracovní smlouva	60 000

Medior developer – tým 1	Není relevantní	BillSaver	Vyvíjet naplánovanou funkcionalitu, řešit chyby, které zjistí tester.	100	Pracovní smlouva	60 000
Tester – tým 1	Není relevantní	BillSaver	Vytvářet testové scénáře a testovat funkcionalitu služeb	100	Pracovní smlouva	50 000

3.3 Technické a technologické aspekty

3.3.1 Technické a technologické aspekty projektu

3.3.1.1 Celková architektura

Pro správné fungování celého projektu je kritický prvotní návrh systému, tedy správné rozdělení obchodních entit, navržení vzájemné komunikace mezi entitami a zjištění potřebných dat, která bude BillSaver uchovávat. Kvůli potřebným vlastnostem popsaným v kapitole 4.1 jsme se rozhodli navrhnout systém a vypracovat jako mikroslužební architekturu. Kvůli tomu jsme rozdělili celý systém na jednotlivé služby, jež by se daly rozčlenit do dvou kategorií – na hlavní a vedlejší služby.

Hlavní služby mají za úkol fakticky spravovat BillSaver. Do těchto služeb se počítají služby, které berou účtenky od zákazníka, skladují účtenky a informace o jednotlivých položkách na účtence a spravují obchodní a uživatelská data. Každá tato služba je navržena jako nezávislá entita, proto bude moct fungovat úplně nezávisle.

Vedlejší služby mají umožňovat komunikaci uživatele se systémem. Do těchto služeb se řadí uživatelská rozhraní, notifikační služby a informační centra. Tyto služby závisí na fungování hlavních služeb a v případě vypadnutí hlavních služeb vypadnou i ty vedlejší.

3.3.1.2 Použité softwarové technologie

Všechny služby jsou postaveny na stejném základu, tedy Java, Spring Boot, Kafka a Rest. Všechny tyto technologie jsou popsané v technické analýze, kde je také odůvodněno, proč byly použity zrovna tyto technologie, viz 4.1.

Kvůli použití stejných základních technologií máme možnost vytvořit standardizované procesy pro samotné nasazení těchto technologii na server. Zkráceně se vše bude nasazovat na kubernetees cluster, jenž poběží u jednoho z cloud dodavatelů. Bližší informace budeme probírat v kapitole 4.6.

Díky těmto technologiím a procesům budeme mít možnost poskytovat jednoduchou, flexibilní a rozšiřitelnou platformu, na níž budeme moct budovat další funkcionalitu.

3.3.2 Provozní řešení

Pro jednoduchý a rozšiřitelný vývojový proces budeme používat cloud služeb, přesněji SaaS neboli software jako služba. Nejenom nám to ulehčí koordinaci všech vývojových prostředí, ale nebude ani zapotřebí kupovat a starat se o drahý hardware. Stačí nám rychlé porovnání, kde zjistíme, že se nám SaaS služby vyplatí, v tabulce 3.2.

Specificky využijeme SaaS službu od Jetbrains, která nabízí Space, jež v sobě komponuje všechny vývojové služby potřebné k úspěšnému dokončení projektu, vedení týmu, projektový management, komunikace, vývoj projektů, dodávku projektů a mnoho dalšího.

Další výhodou použití SaaS je to, že není potřeba kupovat výkonný hardware, ale postačí nám tak základní počítače jak pro vývojáře, tak pro HR, PR a ostatní. Určitě bude zapotřebí koupit

	Lokální vývojové prostředí	SaaS vývojové prostředí	Odůvodnění
Počítač	10000	3750	Pro lokální vývoj musí být počítač dostatečně rychlý, u SaaS toto není zapotřebí, veškeré výpočetní operace probíhají na serverech provozovatele služby. Cena počítače je rozložena na 4 roky, což je jeho životnost.
Software k počítači	0	30000	Lokální prostředí může používat software, který je zdarma, a ušetřit tak na SaaS službě.
Cena administrátora	48000	24000	U lokálního počítače může selhat aktualizace a zničit tak systém, uživatel může stáhnout vir, popřípadě může nastat i fyzická chyba, administrátor je tak musí řešit. S hardwarem, na kterém běží SaaS, se také může něco stát, ale většina těchto chyb se dá vyřešit rychlým přeinstalováním počítače. Protože uživatel nemá žádná data na stroji, který používá SaaS, může to být tak první volba, kterou administrátor zvolí, a ušetří si tak čas hledáním chyby. K této ceně jsem dospěl z mých osobních zkušeností. Tedy administrátorská práce vychází na 1 MD měsíčně a u SaaS 0.5 MD měsíčně, protože nemusí hledat chybu, která nastala. Cenu 1 MD počítáme jako 4000 Kč.
Neproduktivita při problémech	48000	0	V případě lokálního prostředí si programátor musí počkat na opravu, je proto neefektivní, nastavení vývojového prostředí by trvalo déle než samotná oprava. U SaaS prostředí má celé vývojové prostředí na internetu a může tak pokračovat na jakémkoliv jiném stroji. Nemá tak žádné prostoje.
Celkem	106000	57750	

Tabulka 3.2 Porovnání lokálního a SaaS vývojového prostředí za jeden rok

i jiné vybavení od monitoru po židle, nebo alespoň nabídnout tuto možnost zaměstnancům na hlavní pracovní poměr, kteří budou pracovat z domova.

3.4 Dopad projektu na životní prostředí

Netisknutí účtenek ušetří spoustu stromů. Pokud usoudíme, že jedna účtenka je dlouhá 33 cm a široká 8 cm [34]. Můžeme podle velikosti jedné účtenky zjistit, kolik účtenek připadá na velikost jedné A4 $0.0625/(0.33 * 0.08) = 2.36742$. Váha jednoho A4 papíru je 5 g, z toho nám vyjde, že pro tunu papíru je zapotřebí $1000000/5 = 200000$ listů A4 papíru neboli 473484 účtenek. Podle dat z World Atlas je k výrobě jedné tuny papíru zapotřebí 24 stromů. Dále podle ČNB činil

celkový počet transakcí za rok 2020 1,5 miliardy. [35] V Česku se tedy pokácelo přibližně 73 tisíc stromů jen kvůli účtenkám.

Pro výrobu jedné tuny papíru je zapotřebí 75700 litrů vody, [36] to vychází na 2,3 milionu kubických metrů vody za rok. Takovéto množství vody spotřebuje 7000 obyvatel za rok. [37]

Tato čísla jsou pouze orientační a mají poukázat na zbytečné plýtvání přírodními zdroji v době, kdy řešíme globální oteplování. Toto řešení, i kdyby nezredukovalo dopad na životní prostředí ve velkém měřítku, by alespoň částečně přispělo ke zmírňování jedné z největších přírodních krizí, které zažijeme. Detailnější rozbor této problematiky najdete v bakalářské práci Anastasiie Bernikove.

3.5 Finanční analýza

3.5.1 Rozpočet projektu (výdaje projektu v realizační fázi)

Z Ganttova diagramu a výpisu z lidských zdrojů a technické analýzy nám vychází následující výdaje v realizační fázi:

Zaměstnanci

Vývoj					
Role	Počet pozic	Úvazek	MD Cost	MD	Cena pozice
Architekt	1	100	4 000,00 Kč	314	1 256 000,00 Kč
Senior developer	2	100	4 000,00 Kč	218	1 744 000,00 Kč
Medior developer	4	100	3 000,00 Kč	218	2 616 000,00 Kč
Tester	2	100	2 500,00 Kč	218	1 090 000,00 Kč
DevOps / Admin	1	100	3 500,00 Kč	218	763 000,00 Kč
Buisness					
Buisness konzultant	1	100	4 000,00 Kč	314	1 256 000,00 Kč
HR / PR	1	100	2 500,00 Kč	279	697 500,00 Kč
Účetní	1	20	3 500,00 Kč	218	152 600,00 Kč
				Suma	9 575 100,00 Kč

Vybavení

Software / Hardware	Počet kusů	Cena licence	Cena
Jetbrains Space	12	30 000,00 Kč	360 000,00 Kč
Notebook	8	20 000,00 Kč	160 000,00 Kč
Monitor	8	7 000,00 Kč	56 000,00 Kč
PC příslušenstvý	8	5 000,00 Kč	40 000,00 Kč
Home office nábytek	8	10 000,00 Kč	80 000,00 Kč
		Suma	696 000,00 Kč

Celková cena	10 271 100,00 Kč
Cena s rezervou 20%	12 325 320,00 Kč

Obrázek 3.5 Cena realizační fáze

3.5.2 Výdaje v provozní fázi

Jako většina projektů ani BillSaver nebude po roce vývoje dokončen, po počáteční fázi vývoje zpětně zhodnotíme celý projekt a zjistíme, co bychom dělali jinak. Po shrnutí našich zkušeností,

připomínek obchodníků a spotřebitelů vytvoříme plán pro expanzi nové funkcionality. Dále se zaměříme na expanzi do více obchodů, integraci s bankami a jinými aplikacemi. Proto ponecháme jeden vývojový tým a rozšíříme PR tým o jednoho člena.

Budeme muset také pořídit robustní server, který bude vysoce dostupný a dostatečně výkonný. Spolu se serverem budeme každý rok muset kupovat Space licence pro zachování původního ekosystému.

Zaměstnanci

Vývoj					
Role	Počet pozic	Úvazek	MD Cost	MD	Cena pozice
Architekt	1	100	4 000,00 Kč	261	1 044 000,00 Kč
Senior developer	1	100	4 000,00 Kč	261	1 044 000,00 Kč
Medior developer	2	100	3 000,00 Kč	261	1 566 000,00 Kč
Tester	1	100	2 500,00 Kč	261	652 500,00 Kč
DevOps / Admin	1	100	3 500,00 Kč	261	913 500,00 Kč
Buisness					
Buisness konzultant	1	100	4 000,00 €	261	1 044 000,00 Kč
HR / PR	2	100	2 500,00 €	261	1 305 000,00 Kč
Účetní	1	20	3 500,00 €	261	182 700,00 Kč
				Suma	7 751 700,00 Kč

Vybavení

Software / Hardware	Počet kusů	Cena licence	Cena
Jetbrains Space	9	30 000,00 Kč	270 000,00 Kč
Produkcni server	1	500 000,00 Kč	500 000,00 Kč
		Suma	770 000,00 Kč

Celková cena	8 521 700,00 Kč
Cena s rezervou 20%	10 226 040,00 Kč

Obrázek 3.6 Náklady v provozní fázi

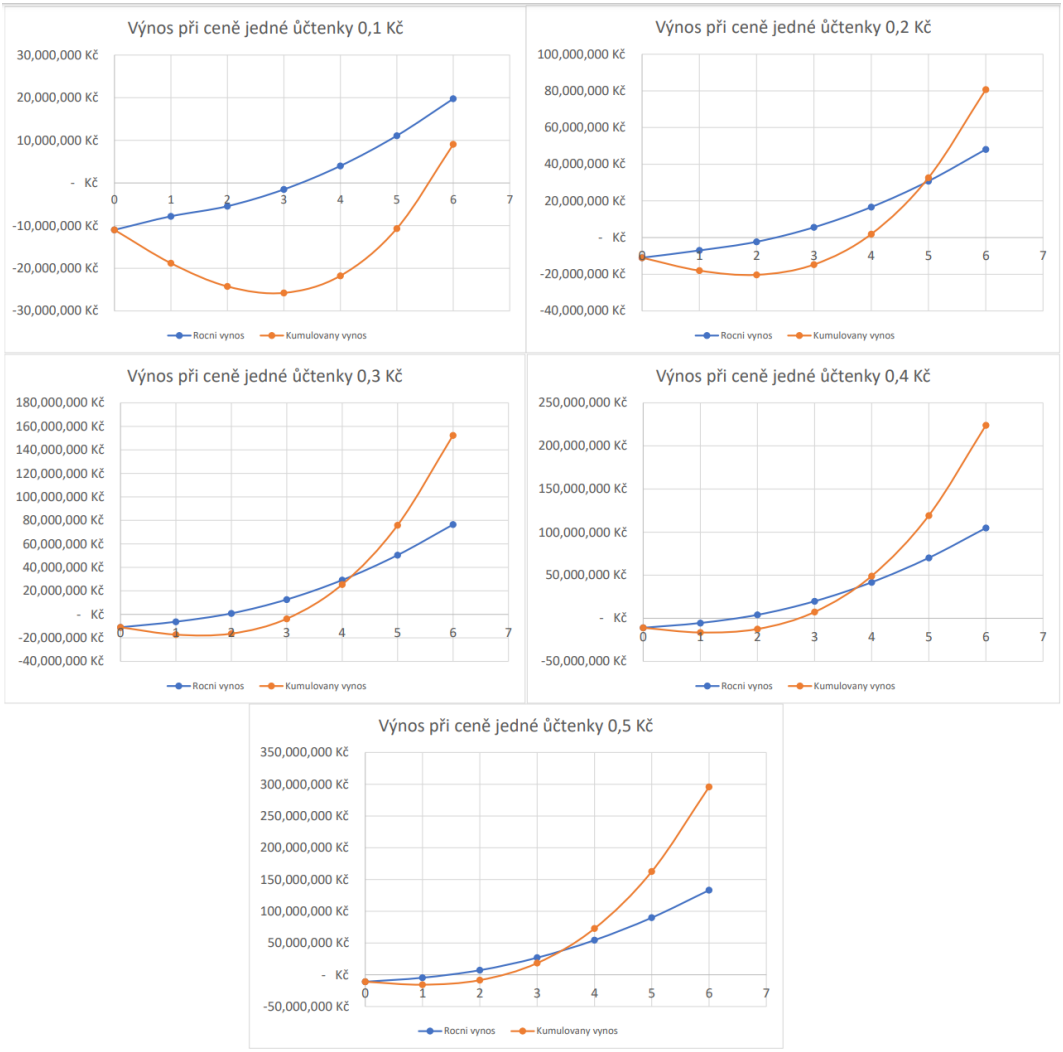
3.5.3 Výnosy projektu v provozní fázi

3.5.3.1 Provozní příjmy generované projektem

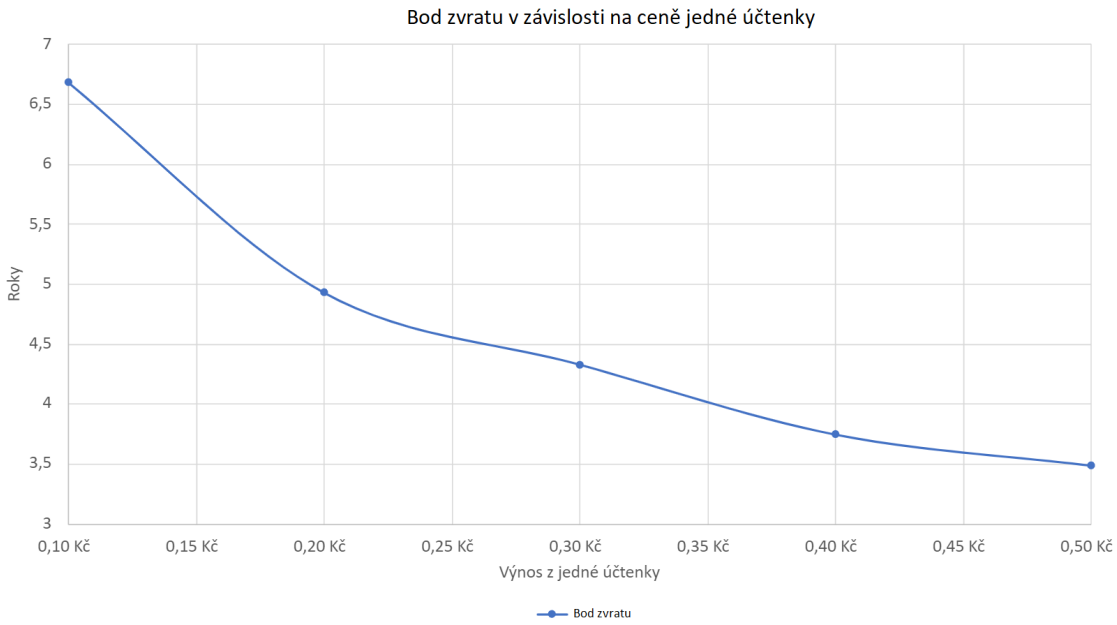
Díky datům z České národní banky můžeme zjistit, že v roce 2020 proběhlo na území České republiky přes 1,5 miliardy karetních transakcí. [35] Dále můžeme z dat Světové banky zjistit, že v České republice žije 10676000 obyvatel. [38] Z toho můžeme vyvodit, že občan provede průměrně 140 transakcí za rok. Toto číslo jistě v následujících letech vzroste, zaokrouhlíme ho tedy na 150 transakcí za rok.

Budeme počítat s 10% nárůstem adopce systému u obchodníků za rok až do 60% adopce. Podobně tak budeme počítat s adopčním nárůstem 5% u spotřebitelů za rok až do 30%. Jak jsme již vypočítali, náklady za jeden rok budou 10, 5 milionu a počáteční investice činila 12,5 milionu korun. Díky těmto faktorům dokážeme odhadnout cenu, za kterou bychom museli zpracovat jednu účtenku, viz: 3.7. Z těchto grafů vyplývá že prvotní čistý výdělek nastane mezi 2,5 až 5,5 lety.

Stejně tak můžeme porovnat postupně klesající bod zvratu v souvislosti s cenou jedné účtenky, viz: 3.8.



Obrázek 3.7 Výnos jedné účtenky v dalších letech – návratnost investice při ceně zpracovaní jedné účtenky



Obrázek 3.8 Bod zvratu v závislosti na výnosu z jedné účtenky

3.5.3.2 Zdroje financování provozu projektu

Zdroje financování mohou přicházet od obchodníků, spotřebitelů, reklam nebo od třetí strany, jako je banka, která by chtěla zaintegrovat tuto funkcionalitu do svého systému. Průměrná cena jednoho metru ekologického termo kotoučku neboli účtenkového papíru se pohybuje okolo 0,35–0,9 Kč/m. [34] Při odhadu průměrné délky jedné účtenky 33 cm se náklady na ni pohybují okolo 0,1–0,3Kč. Tím pádem můžeme stanovit cenu zpracování jedné účtenky BillSaverem na 0,2 Kč pro obchodníky.

Dále můžeme postupně přidávat placené funkce do BillSaveru pro uživatele a tím pádem dostávat finanční zdroje i od uživatelů. Ti jsou však často rezistentní k placeným aplikacím, a tak se na počátku budeme zaměřovat na rozšíření aplikace mezi veřejnost a financování bude od obchodníků.

Další možnost představuje spojení se s nějakou bankou, jež by chtěla mít takovouto funkcionalitu zabudovanou do své aplikace. Uživatel by si tak mohl prohlížet seznam nakoupeného zboží spárovaný se svými transakcemi. Na tuto variantu se však nebudeme spoléhat.

3.6 Analýza rizik a jejich předcházení

3.6.1 SWOT analýza

Silné stránky	Slabé stránky
▬ Na trhu neexistuje tak automatizované řešení ▬ Šetření přírody a tak snížení CO_2 stopy ▬ Robustní a škálovatelná architektura ▬ Několik možností monetizování projektu	▬ Velký počáteční kapitál ▬ Žádná velká obchodní korporace zatím nespolupracuje s projektem ▬ Každý obchodník se musí jednotlivě připojit k projektu, tak může být pomalá adopce.
Příležitosti	**Hrozby**
▬ Široká cílová skupina ▬ Možnost rozšíření služby do dalších států	▬ Únik citlivých informací a tak porušení GDPR ▬ Obchodníci nebudou ochotni spolupracovat ▬ Nezájem zákazníků o poskytovanou službu ▬ Konkurence vyjde s podobným řešením

Tabulka 3.3 SWOT analýza

3.6.2 Zhodnocení rizik a navrhovaná opatření k jejich předcházení

Riziko č. 1 Problematický nábor – Kvůli relativně velkému nedostatku programátorů může dojít k prodloužení náborového procesu.

Riziko č. 2 Nekvalifikovaní pracovníci – Nábor pracovníků může vést k výběru málo kvalifikovaných pracovníků, kteří nebudou schopni efektivně pracovat na daném zadání. Mitigaci tohoto problému můžeme řešit dvěma způsoby. První, méně vhodný způsob představuje vytvoření obtížného přijímacího řízení. Toto řešení by fungovalo velmi dobře, pokud by bylo hodně programátorů. V této situaci však jediné kandidáty spíše odradíme dlouhým přijímacím řízením. Druhý, podle mě lepší způsob je dělat detailní interview, kde vysvětlíme, co je to BillSaver, a postupně s případným kandidátem probereme architekturu a technologii. Podle průběhu tohoto rozhovoru se rozhodneme o jeho přijetí do týmu. Jedinou nevýhodou této metody je velká časová náročnost pro architekta.

Riziko č. 3 Rozpad hlavního týmu po dokončení projektu – V případě rozpadnutí hlavního vývojového týmu bychom ztratili velké know how celého projektu a tím také vývojové mementum, což by způsobilo zpomalení následného rozšíření funkcionality. Jediné efektivní řešení tohoto problému je dbát na důkladnou dokumentaci projektu.

Riziko č. 4 Uniknutí nebo ztráta informací – Zamezení úniku informací zajistí pouze správné šifrování, logování a správné audit logování. Pro zamezení ztráty informací bychom měli zajistit automatické zálohování celé databáze.

Riziko č. 5 Špatný prvotní návrh systémů – Jediný způsob, jak správně navrhnout prvotní návrh, je detailní analýza problému a konzultace architektury s jinými architekty mikroslužbových aplikací.

Riziko č. 6 Špatná volba technologií – Díky využité architektuře, i když je toto riziko reálné, by neměla způsobit velký dopad na vývoj systému.

Riziko č. 7 Nízká adopce od obchodníků – V případě nízké adopce obchodníků nedojde k rozšíření systému mezi veřejnost, nevznikne tak podnět pro spotřebitele využít danou aplikaci. V tomto případě by bylo zapotřebí vytvořit obchodníkům další větší podnět. Odhady na adopci jsou však přiměřené, a tak odhaduji, že pravděpodobnost, že tato situace nastane, je malá.

Riziko č. 8 Nezájem veřejnosti – V případě nízké adopce veřejnosti bychom nedokázali zpracovávat tolik účtenek, a proto bychom nedokázali generovat dostatečný obrat, takže by se BillSaver stal ztrátovým. Mitigace tohoto problému je dobrá reklamní kampaň.

Riziko č. 9 Nízký odhad pracnosti – Toto by způsobilo prodloužení doby vývoje a tím zvýšení nákladů na vývoj. Ke zmenšení pravděpodobnosti výskytu takovýchto následků vede stejně jako u návrhu systému detailní analýza problému.

Název	Dopad	Detail dopadu	Pravděpodobnost výskytu	Riziko
Riziko č. 1	Střední	Náklady: Nárůst menší než 5% Čas: Nárůst až 10% Kvalita: Neznatelný dopad	Střední	Riziko střední
Riziko č. 2	Vysoké	Náklady: Nárůst až 15% Čas: Nárůst až 15% Kvalita: Významný dopad na kvalitu	Nízká	Riziko střední
Riziko č. 3	Střední	Náklady: Nárůst menší než 5% Čas: Nárůst až 10% Kvalita: Významný dopad na kvalitu	Střední	Riziko střední
Riziko č. 4	Velmi vysoký	Náklady: Nárůst vetší než 20% Čas: Neznatelný Kvalita: Produkt nelze používat	Velmi nízké	Riziko střední
Riziko č. 5	Střední	Náklady: Nárůst až 10% Čas: Nárůst až 10% Kvalita: Neznatelný dopad	Střední	Riziko střední
Riziko č. 6	Nízký	Náklady: Nárůst menší než 5% Čas: Nárůst menší než 5% Kvalita: Neznatelný dopad	Střední	Riziko střední
Riziko č. 7	Vysoký	Náklady: Nárůst až 15% Čas: Nárůst až 15% Kvalita: Nepřijatelný dopad	Nízké	Riziko střední
Riziko č. 8	Střední	Náklady: Nárůst až 10% Čas: Nárůst až 10% Kvalita: Nízký dopad	Nízké	Riziko střední
Riziko č. 9	Vysoký	Náklady: Nárůst až 15% Čas: Nárůst až 15% Kvalita: Neznatelný dopad	Nízké	Riziko střední

Tabulka 3.4 Tabulka zhodnocení rizik

3.6.3 Heat mapa rizik

Heat mapa nám pomůže zobrazit nečekaná seskupení a zobrazí nám trendy kde se naše rizika pohybují. Jak můžeme vidět v tabulce 3.5, většina rizik se pohybuje ve střední části grafu tedy

Pravděpodobnost výskytu \ Dopad	Velmi nízký 5%	Nízký 20%	Střední 40%	Vysoký 60%	Velmi vysoký 80%
Velmi vysoký 80%					
Vysoký 60%					
Střední 40%		Riziko č. 6	Riziko č. 1 Riziko č. 3 Riziko č. 8		
Nízký 20%			Riziko č. 5	Riziko č. 2 Riziko č. 7 Riziko č. 9	
Velmi nízký 5%					Riziko č. 4

Tabulka 3.5 Heat mapa rizik – zobrazující všechny rizika v závislosti na jejich dopadu a pravděpodobnosti výskytu

střední pravděpodobnost výskytu a střední dopad. Jedině Riziko č. 4 je na jednom z extrémů grafu a to velmi vysoký dopad, takovéto riziko má však skoro každý projekt.

Kapitola 4
Technické řešení

4.1 Architektura

Pro správné fungování celého projektu je kritický prvotní návrh systému, tedy správné rozdělení buisness entit, navržení komunikace a popis zodpovědnosti.

Ze zadání nám vychází, že existují dvě skupiny uživatelů – obchodníci a spotřebitelé. Pro pochopení si všechny potřeby obou skupin zapíšeme do tabulky.

Potřeby uživatelů	
Obchodníci	Spotřebitelé
Informativní UI	Jednoduché a intuitivní UI
Souhrnný přehled demografických charakteristik svých zákazníků	Informovaný náhled nad svými útratami
Snadná integrace s již existujícími systémy	Podpora většiny uživatelských zařízení
Zabezpečení prodejních dat, tedy převážně účtenky	Zabezpečení svých osobních dat
Podpora všech buisness funkcionalit, které byly stanoveny v kapitole 3.1.2	
Přidávat další funkcionalitu na základě požadavků	
Aplikace musí být dostupná	

Tabulka 4.1 Porovnaní potřeb spotřebitelů a obchodníků

Z těchto potřeb nám vyplynulo několik nutných aspektů, které musí naše architektura podporovat:

- Flexibilita – Pro uspokojení obou skupin uživatelů musí být náš systém flexibilní pro případné změny nebo doplňky funkcionality.
- Škálovatelnost – Abychom se nedostali do komplikací při nákupních horečkách, například Black Friday či Cyber Monday, musíme být schopni přidat více výpočetních výkonů, abychom nekolabovali pod nátlakem uživatelů a účtenek.
- Robustnost – V případě výpadku nebo plánované odstávky bychom neměli přestat přijímat účtenky od obchodníků. Obchodníci i uživatelé by se měli spolehnout na to, že jejich data budou v bezpečí a neztratí se.

Kvůli těmto potřebám se nejlépe hodí použití mikroservisní architektury. Z toho vyplývá, že musíme klást velký důraz na návrh buisness entit, jednotného komunikačního standardu a ohraničení jednotlivých odpovědností.

4.2 Buisness služby

V této kapitole si rozepíšeme jednotlivé buisness služby, jež figurují v systému, a ohraničíme jejich odpovědnosti. Na tyto služby následně navazuje obrázek 4.1 který ukazuje primární komunikační cesty které systém bude používat.

1. Spravce účtenek: První buisness služba je určitě úložiště účtenek. Má za úkol přijímat účtenky ve správném formátu a ukládat je. Při přijetí účtenky odešle notifikaci, že danému uživateli přišla účtenka. V případě dotazu na účtenky služba požadované účtenky vrátí.

2. Správce Zboží: Další služba uchovává data o daných produktech. Poslouchá, jestli nepřišly nové účtenky, a pokud ano, vyextrahuje produkty, které přišly, zkontroluje, že daný produkt zná a má všechna jeho potřebná data. Pokud by nějaká data chyběla, vyšle žádost o doplnění informací. Dále poskytuje informace o daných produktech na základě dotazů od uživatelů.

3. Generátor účtenek: Třetí služba propojuje systém obchodníka se systémem BillSaver. Upraví účtenku do podoby podporované BillSaverem a následně ji odešle dál.

4. Správce spotřebitelských účtů: Služba, která má na starosti uživatelská data, je vstupním bodem pro spotřebitele. Má na starosti uživatelskou registraci, úpravu uživatelských dat a propojování platebních metod s účtenkami.

5. Spotřebitelský informační systém: Spotřebitel bude pravidelně komunikovat s informační entitou, jež má na starosti monitorování správy ohledně účtenek a správy od obchodníků. Uživatel bude moct hodnotit své nákupy a případně posílat připomínky obchodníkům.

6. Spotřebitelské rozhraní: Toto rozhraní zajišťuje synchronní komunikaci se všemi systémy se kterými musí spotřebitel komunikovat.

7. Správce obchodních účtů: Stejně jako služba, která propojovala spotřebitele, existuje i služba propojující obchodníky s BillSaverem. Má na starosti registraci a úpravu obchodníkových dat.

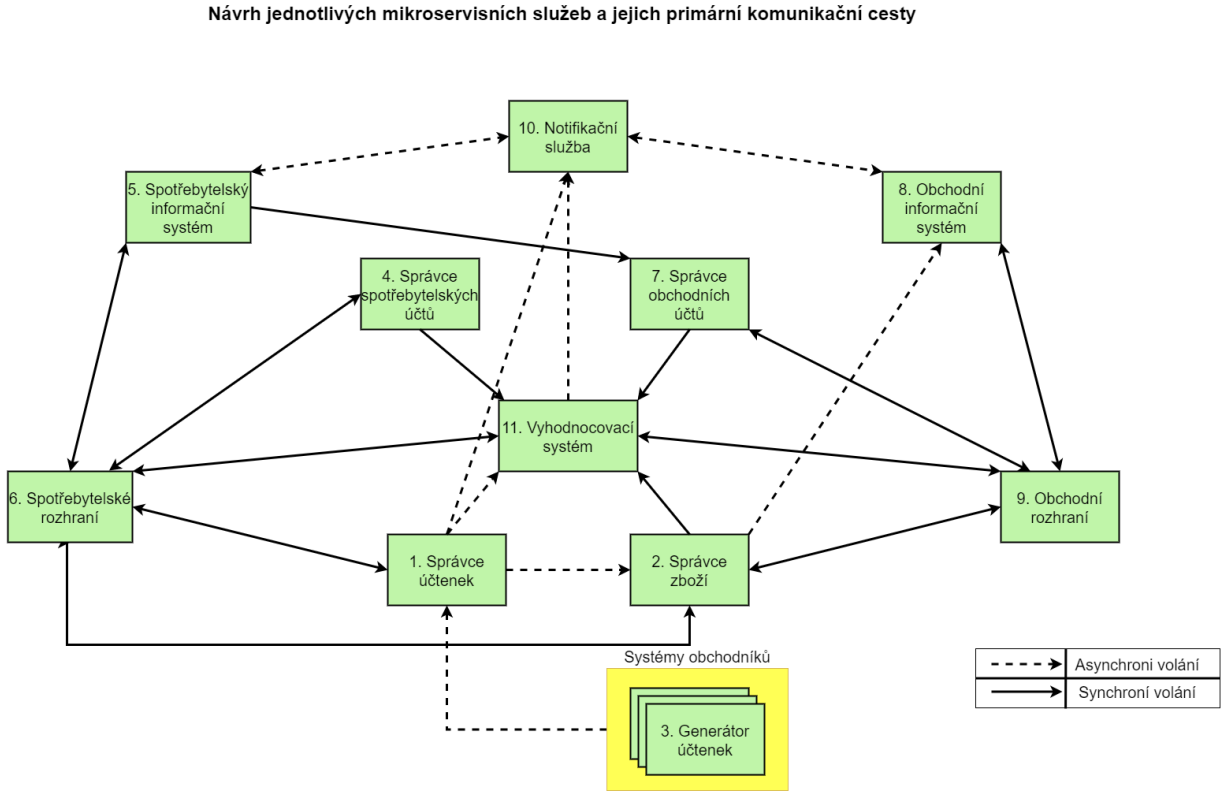
8. Obchodní informační systém: Pro obchodníky existuje separátní informační služba, jež sbírá zpětnou vazbu od uživatelů, shromažďuje jejich reakce na jednotlivé produkty a informuje o dotazech k doplnění informací. Dále monitoruje případné dotazy k doplnění informací o daných produktech. Pokud takový dotaz přijde a bude se týkat daného obchodníka, obchodníkovi přijde zpráva o doplnění informací o daném produktu. Až obchodník doplní informace o zboží, služba odešle požadavek na úpravu informací o daném produktu.

9. Obchodní rozhraní: Toto rozhraní zajišťuje synchronní komunikaci se všemi systémy se kterými musí obchodník komunikovat.

10. Notifikační služba: Notifikační služba sleduje žádosti o poslání notifikací jak spotřebiteli, tak obchodníkovi a na základě jejich nastavení je informuje v uživatelsky nastavené frekvenci buď posíláním notifikací do mobilu, e-mailem či bude posílat informace zprávami SMS.

11. Vyhodnocovací systém: Vyhodnocovací systém má na starosti vyhodnocování uživatelských nákupních trendů a na jejich základě vytváří prediktivní uživatelské modely, jež doporučují spotřebiteli zefektivnění jeho nákupních zvyků.

K těmto službám přibudou ještě dvě důležité služby, které nemají nic společného s buisness kontextem aplikace, ale jsou určitě velice nutné ke správnému fungování celého systému.



Obrázek 4.1 Návrh jednotlivých mikroservisních služeb a jejich primární komunikační cesty

- Logování: Tato služba bude poslouchat všechny ostatní služby a bude zapisovat všechno, co ostatní služby pošlou a co bude jejich log. To bude sloužit k identifikování případných chyb a zajistí nám tak rychlejší řešení problémů v budoucnu. V této službě se budou skladovat úplně všechny logy vygenerované během činnosti systému, tedy auditní logy, server logy, logy přijatých zpráv a popřípadě i další.
- Monitorování: Tato služba bude mít za úkol sledování stavu celého systému, tedy monitorování zatíženosti systému, sledování, jestli jednotlivé služby jsou funkční, a případně bude informovat admin tým ohledně problémů.

4.3 Data

V této kapitole se zaměříme na vydefinování důležitých dat, která musí jednotlivé služby uchovávat a odesílat.

Hlavní data uchovávaná v BillSaveru jsou účtenky. Tyto informace můžeme rozdělit na hlavičky a těla. Hlavička obsahuje všechny identifikační údaje o dané tržbě a celkovou cenu, tělo obsahuje a cenu jednotlivých produktů spolu s jejich identifikačními parametry. Pro detailní informace, jaká data se ukládají a přenášejí, si vypíšeme v tabulkách: 4.2, 4.3, 4.4, 4.5. Všechny tyto informace jsou vcelku neměnné, a proto tyto informace ukládáme v relační databázi.

Důležitá jsou dále data jednotlivého zboží, která jsou definované číslem EAN. Protože se mohou tyto informace hodně lišit, ukládáme je v dokumentové databázi pro její jednoduchou expadibilitu pro více informací, viz: 2.5.2. Tyto informace se vydefinují s jednotlivými obchodníky. Pro obrovskou variabilitu si zde ukážeme možné hodnoty pro dva typy zboží: jídlo a mobilní telefony. Viz tabulky 4.6, 4.7.

Hlavička			
Název	Popis	Povinnost	Datový typ v Java
UUID obchodníka	Zpráva definující, od koho účtenka přišla, systém poté spáruje UUID obchodníka s jeho účtem. IČO, DIČ a jiné identifikátory si bude moct obchodník nastavovat v jeho účtu.	Ano	UUID
UUID provozovny	Zpráva definující, z jakého obchodu účtenka přišla. Účtenka se poté spáruje s názvem, adresou a nebo telefoním číslem specifické provozovny. Stejně jako u UUID obchodníka i u UUID provozovny si bude moct obchodník nastavit různé parametry ve svém systému.	Ano	UUID
Datum tržby	Datum a specifický čas, kdy obchodník zaznamenal tržbu.	Ano	LocalDateTime
ID Pokladny	Označení pokladního zařízení, na kterém je tržba evidována.	Ano	String
Číslo účtenky	Pořadové číslo účtenky zaznamenané pokladním systémem.	Ano	String
Režim tržby	Termín stanovující zákon, obchodník může uvést buď „běžný", nebo „zjednodušený".	Ano	String
FIK	Fiskální identifikační kód vygenerovaný finanční správou.	Ne	String
PKP	Podpisový kód poplatníka vygenerovaný pokladním zařízením.	Záleží na režimu tržby a FIK	String
BKP	Bezpečnostní kód poplatníka vygenerovaný z PKP	Ano	String

Tabulka 4.2 Informace v hlavičce účtenky

Poslední, co uchováváme, jsou osobní data obchodníků a uživatelů. Ty zde nebudeme rozepisovat pro jejich relativně jednoduchou strukturu, pouze si zde vypíšeme ty důležité. Spotřebitel: jméno, příjmení, heslo, kontaktní údaje, připojené kreditní a debetní karty. Obchodník: název společnosti, identifikační údaje jednotlivých poboček, údaje o společnosti pro finanční úřad jako IČO, DIČ a podobně a kontaktní udaje.

4.4 Komunikace

Škálovatelná a jednotná komunikace napomůže k udržení flexibility a robustnosti jednotlivých služeb. Abychom však dokázali komunikaci udržet jednotnou a škálovatelnou, musíme vybrat standardní technologii, která se bude používat napříč všemi službami. Existují dva typy komunikace pro mikroslužbovou architekturu, synchronní a asynchronní.

Synchronní komunikace funguje na principu odeslání žádosti a čekání na odpověď. Tato odpověď může být jak kladná, tedy vrátila se správná data, tak záporná, tzn. něco se pokazilo na serveru. Synchronní komunikace se většinou používá na takovou komunikaci, kde je odpověď vyžadovaná okamžitě. Takováto komunikace se většinou vyžaduje u interakce s uživatelem, jenž chce informace dostat okamžitě na svůj dotaz. Nejznámější technologie, které na takovouto

Tělo			
Název	Popis	Povinnost	Datový typ v Java
Celková cena	Celková cena tržby včetně DPH	Ano	BigDecimal
Sumární sazba	Seznam jednotlivých sazeb DPH spolu se základem daně, samotnou daní a celkovou cenou	Ano	List<Sazba>
Seznam položek	Seznam produktů, které spotřebitel nakoupil	Ano	List<Produkt>

Tabulka 4.3 Informace v tělu účtenky

Sazba			
Název	Popis	Povinnost	Datový typ v Java
Sazba daně	Procentuální sazba daně	Ano	BigDecimal
Základ daně	Základ daně všech produktů zahrnutých v sazbě DPH	Ano	BigDecimal
Daň	Daň vypočítaná ze sazby daně a základu daně	Ano	BigDecimal
Celkem	Celková částka sečtená z daně a základu daně	Ano	BigDecimal

Tabulka 4.4 Informace o sazbě

Produkt			
Název	Popis	Povinnost	Datový typ v Java
EAN	European Article Number položky neboli čárový kód	Ano	String
Sazba daně	Procentuální sazba daně, do které položka spadá	Ano	BigDecimal
Cena	Celková cena daného produktu	Ano	BigDecimal
Množství	Celkové množství jedné položky	Ano	BigDecimal
Jednotka	Jednotka množství, tedy kg, kus, litr atd.	Ano	Enum

Tabulka 4.5 Informace o produktech v účtence

Jídlo			
Název	Popis	Povinnost	Datový typ v Java
EAN	European Article Number položky neboli čárový kód	Ano	String
Název	Název zboží	Ano	String
Výrobce	Výrobce produktu	Ano	String
Země původu	Země, kde bylo zboží vyrobeno nebo vypěstováno	Ano	String
Kalorie	Počet kalorií v suchém / nepřipraveném stavu	Ano	BigDecimal
Popis	Popis produktu vytištěný na obalu či příbalovém letáku	Ne	String
Složení	Složení produktu	Ne	String
Nutriční hodnoty	Tabulka s nutričními hodnotami	Ne	List<Nutriční Hodnota>

Tabulka 4.6 Informace o produktu: Jídlo

Elektronika			
Název	Popis	Povinnost	Datový typ v Java
EAN	European Article Number položky neboli čárový kód	Ano	String
Název	Název zboží	Ano	String
Výrobce	Výrobce produktu	Ano	String
Země původu	Země, kde bylo zboží vyrobeno.	Ano	String
Popis	Popis produktu vytištěný na obalu či příbalovém letáku	Ne	String
Specifikace	Specifikace zboží. Například: rozlišení obrazovky, velikost úložiště, velikost baterie.	Ano	List<Specifikum>
Manuál	Odkaz na soubor s manuálem	Ano	String

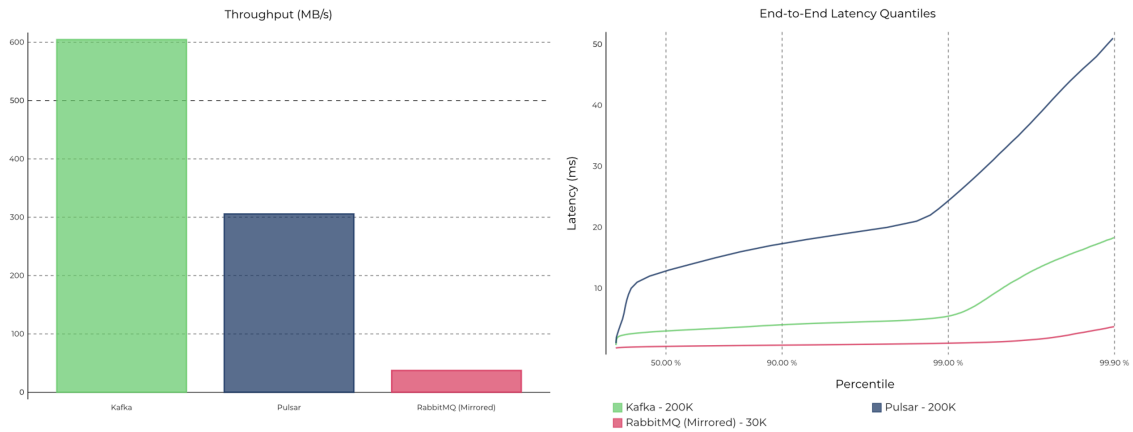
Tabulka 4.7 Informace o produktu: Elektronika

komunikaci existují, jsou REST, SOAP, GraphqQL a gRPC. Každá z nich má své výhody a nevýhody. Zde si je porovnáme a na základě výsledku vybereme tu nejvhodnější technologii pro synchronní komunikaci.

Kvůli velké rozšířenosti a vysoké podpoře v aplikačních rámcích vybereme REST rozhraní k synchronní komunikaci. Velké transfery dat budou zajišťovat asynchronní volání, kvůli tomu nepotřebujeme benefity gRPC. Dále i když by graphQL byla vhodnější volba, její nízká rozšířenost a komplexní zavedení do jednotlivých systémů ji diskvalifikuje z výběru. Nakonec výhody soap rozhraní rychle zastíní jeho vysoká režie a tím pádem pomalá komunikace oproti REST.

Asynchroní komunikace funguje na principu odeslání zprávy, která obsahuje fakt nebo instrukci. Tato zpráva se za nějakou dobu zpracuje, ale služba, jež ji odeslala, nečeká na odpověď. Služby na sebe nejsou vůbec vázané a systém se kvůli tomu stává jednoduše škálovatelným, flexibilním a robustním. Proto se pokusíme najít nejrychlejší technologii pro posílání dat. Takováto komunikace bude v našem systému používaná pro přenos a zpracování velkého množství dat. Kvůli tomu se pokusíme najít nejrychlejší technologii pro takovouto asynchronní komunikaci.

Základní technologie, které se používají pro přenos asynchronních dat, jsou RabbitMQ, Pulsar a Kafka. Jak můžeme vidět na obrázku 4.2, rychlost Kafky je 2× větší než Pulsar a 10× větší než RabbitMQ. Díky těmto výsledkům je jisté, že pro asynchronní komunikaci použijeme technologii Kafka.



Obrázek 4.2 Porovnání rychlostí jednotlivých asynchronních technologií [42]

	Synchroni komunikace			
	REST	Soap	GraphQL	gRPC
Popularita	Velká	Malá	Střední	Střední
Jednoduchost použití	Velice jednoduché	Jednoduché	Středně těžké	Středně těžké
Rychlost komunikace	Střední	Střední	Pomalá	Rychlá
Podpora v aplikačních rámcích	Velká	Střední	Malá	Střední
Výhody:	Rozhraní používané v celém webu.	Rozhraní, které je stále používáno ve finančním sektoru	Klient si dokáže vybrat data, která mu server pošle	Velice rychlá komunikace
	Velká znalost ve vývojářské komunitě	Podpora starších systémů	Vysoká míra flexibility pro uživatelské rozhraní	Obousměrná komunikace založená na HTTP/2
	Jednoduché nastavení ukládání do mezipaměti	Značné rozšíření pro zabezpečení komunikace	Není potřeba verzovat rozhraní	
Nevýhody:	Neexistuje jeden správný způsob, jak vytvářet rest api	Podpora pouze XML formátu	Při množství dotazů můžeme přetížit službu	Vysoká provázanost mezi službami
	Velké množství obecných dat, jako jsou odkazy na jiné api	Vysoká režie	Velice obtížné nastavení ukládání do mezipaměti	Dokumentace musí být psaná zvlášť, neexistuje žádná introspekce rozhraní

Tabulka 4.8 Tabulka porovnávající jednotlivé komunikační standardy [39, 40, 41]

4.5 Základní technologie

Jak jsme již rozebrali v kapitole 2.4, nejpopulárnější aplikační rámce jsou Spring Boot a Quarkus. Kvůli široké dokumentaci a daleko větší popularitě vybereme pro náš projekt Spring Boot. Díky této volbě a kvůli tomu, že chceme používat standardní technologie, se nám automaticky zúží výběr pomocných aplikačních rámců.

Java spolu se Spring Boot stále potřebují JVM pro běh celé aplikace. Běh těchto aplikací zajišťuje Java servlet container, což je aplikační server, který v sobě spouští Java EE aplikace, tedy Spring Boot aplikace. Základní server, který Spring Boot používá, se jmenuje Tomcat.

Spring Boot používá pro komunikaci s DBMS Spring data JPA, který vnitřně používá Spring Data a Hibernate pro komunikaci s databázemi.

Pro REST, tedy synchronní komunikaci Spring Boot, využívá Spring WebFlux. WebFlux vytváří jednoduché REST rozhraní s automatickou dokumentací koncových bodů.

Pro asynchronní komunikaci Spring Boot obsahuje Spring for Apache Kafka. Toto rozšíření přidává celému projektu jednoduché použití Kafky. Dále existuje i modul Spring Cloud Stream, který přidává stream processing do našeho systému.

Abychom vytvářeli jednoduchý a přehledný kód, využíváme autogeneraci kódu od projektu Lombook, a abychom psali ještě méně kódů v případě potřeby konverze jednoho objektu do jiného, využíváme Mapstruct projekt, který automaticky vytváří konverzní funkce.

Spring Boot také obsahuje automatické testovací rozhraní pro náš kód. K tomu využívá Spring Test, který obsahuje testování jak jednotlivých funkcí (unit testing), tak aplikačního rozhraní (intergation testing). Pro unit testing využívá knihovnu Juint 5.

Pro rekapitulaci uvedeme tabulku 4.9, kde sjednotíme všechny základní technologie, které naše služby využívají.

4.6 CI/CD

CI/CD se zaměřují na zautomatizování procesů potřebných k nasazení jednotlivých služeb do produkčního prostředí. Použití tohoto procesu nám umožní rychleji odhalovat chyby, zvýšit kvalitu kódu a svým zaměřením se velmi hodí pro použití SCRUM modelu vývoje.

Protože používáme Intellij IDEA se vzdálenou kompilací odůvodnění, viz: 3.3, využijeme i ostatní produkty od Jetbrains. Budeme tak mít jednotné prostředí, které spolu velice dobře komunikuje a operuje.

Tedy pro automatické testování, kompilaci, sledování problému, správu celého projektu využijeme Space od Jetbrains.

Pro běh jednotlivých služeb na produkčním serveru použijeme standardní virtualizační technologii Kubernetes. Tato technologie zajistí jednoduché škálování, automatické monitorování aplikací a jednoduchou migraci celé služby na jiné fyzické servery buď lokální, nebo k poskytovateli cloud computing služeb.

4.7 Procesní flow

V teto sekci se zaměříme na popis důležitých procesů, které v systému probíhají. Jde především o uložení účtenek, verifikaci dat, přidání platební karty, přidání a úpravu zboží. Zobrazení dat, účtenky, uživatelské informace zde nebudeme ukazovat, bude se jednat o jednoduché volání do databáze.

Použité technologie		
Název technologie	Oblast použití	Popis
Java	Základ	Výchozí programovací jazyk
Spring Boot	Základ	Základní aplikační rámec, který propojuje ostatní aplikační rámce
Tomcat	Základ	Výchozí Java servlet container
Spring Data	Komunikace	Zajišťuje komunikaci s DBMS
Hibernate	Komunikace	Zajišťuje transformaci DBMS objektu do Java entit
Spring WebFlux	Komunikace	Zajišťuje vytváření a komunikaci přes standardní REST rozhraní
Spring for Apache Kafka	Komunikace	Zajišťuje asynchronní komunikaci s ostatními službami
Lombook	Přehlednost kódu	Auto-generuje kód ze základních anotací
Mapstruct	Přehlednost kódu	Auto-generuje transformační funkce mezi objekty
Spring Test	Testování	Zjednodušuje vytváření jak unit, tak integration testů
Junit 5	Testovaní	Využívá se k testování jednotlivých funkcí.
Git	DevOps	Správce souborů a verbovací nástroj pro kód
Space	DevOps, Management	Tvorba a udržování CI/CD a management celého projektu

Tabulka 4.9 Použité základní technologie v projektu

4.7.1 Uložení účtenky

1. Obchodník přijde do obchodu, nakoupí a začne platební proces. Pokladní systém zpracuje všechny položky, které nakoupil, a vytvoří platební příkaz. V tomto procesu vygeneruje veškeré náležité údaje, které patří k účtence, jako jsou FIK, datum a čas nákupu a ostatní identifikační čísla. Jakmile uživatel zaplatí, pokladní systém pošle číslo karty generátoru účtenek.
2. Generátor účtenek zaheshuje tato data a zkontroluje v jeho interní databázi zaheshovaných karetních údajů, jestli systém může přijmout tuto účtenku, která byla zaplacena poskytnutou kartou. Generátor účtenek tedy vrátí pokladnímu systému true/false.
3. Pokud pokladní systém dostane zpět false, pokračuje starým způsobem. V případě, že dostane true, pošle účtenku generátoru účtenek. Následně řekne spotřebiteli, že účtenku najde ve svém BillSaver účtu.
4. Generátor převede účtenku po přijetí do standardního tvaru. Číslo karty zaheshuje, podepíše účtenku obchodním privátním klíčem a odešle ji přes Kafku do streamu neověřených účtenek.
5. Správce obchodníka přijme tuto účtenku, ověří její podpis a pošle ji do streamu ověřených účtenek.
6. Správce uživatelských účtů přijme ověřenou účtenku a přiřadí k ní uživatele na základě zaheshovanych karetních údajů. Následně účtenku pošle do streamu přiřazených účtenek.
7. Správce zboží přijme, ověřenou a přiřazenou účtenku si uloží a odešle zprávu, že uživateli byla přidána nová účtenka.

4.7.2 Přidání platební karty

1. Uživatel se přihlásí do systému a vyplní formulář s novou kartou

2. Nová karta se ověří s bankovním účtem
3. Správce účtu kartu s jejími údaji uloží do databáze
4. Z údajů na kartě vytvoří hash a odešle ho ve zprávě, že systém zaregistroval novou kartu.
5. Generátor účtenek tuto informaci zachytí a uloží si daný hash do vlastní databáze.
6. V případě zjišťování pokladního systému, jestli dokáže BillSaver přijmout hash dané karty, generátor bude moct odpovědět na základě interní databáze zaheshovaných karet, které systém přijímá.

4.7.3 Verifikace dat

1. Když správce účtenek ohlásí příchod nové účtenky pro daného spotřebitele, správce zboží tuto zprávu zachytí.
2. Správce zboží ze zprávy vyextrahuje EAN kódy zboží.
3. Tyto EAN kódy porovná se svojí databází produktu.
4. Správce ověří, že zboží existuje a je kompletně vyplněné.
5. V případě, že je zboží vyplněno, systém nic nedělá. Pokud však zaznamená chybějící informace o zboží ze zprávy, vyextrahuje, od kterého prodejce účtenka přišla, a pošle zprávu obchodníkovi o doplnění informací o zboží.
6. Zprávu zachytne notifikační služba, která mu na základě obchodních preferencí tuto zprávu pošle do předdefinovaných komunikačních kanálů.
7. Stejně tak tuto zprávu zachytí i obchodní informační služba, která přiřadí dotaz o doplnění těchto informací k obchodnímu profilu.

4.7.4 Přidání a úprava zboží

1. Pokud bude chtít obchodník přidat nové zboží, nebo stávající zboží upravit, přihlásí se do obchodního informačního systému.
2. Tam pomocí automatického načítacího systému nebo vyplnění dotazníku uvede informace o novém či upraveném zboží.
3. Informační systém tuto informaci odešle.
4. Správce zboží tuto zprávu zachytí.
5. Data ve zprávě se verifikují a doplní se do databáze.

Kapitola 5
Proof of concept

Pro zjištění funkčnosti technologií a architektury vytváříme proof of concept. Zaměřujeme se na komponenty, které zajišťují ukládání a správu účtenek a správu zboží. Ostatní komponenty jako správu obchodníků a spotřebitelů, implementujeme pouze do míry potřebné k uložení účtenek. Hlavními komponenty celého systému jsou generátor účtenek, správce zboží a správce účtenek.

5.1 Generátor účtenek

Generátor je velice specifická komponenta která se přizpůsobí účetnímu systému obchodníka. Kvůli tomu zde implementujeme ukázkovou implementaci od které se ostatní generátory účtenek odvíjejí.

Jak již bylo zmíněno, generátor účtenek má na starosti převážně transformaci účtenek do kompatibilní struktury a ověřovaní zda systém může přijmout danou transakci. Z toho nám vyplývá že jediné informace které ukládáme jsou hashe platebních karet které dokážeme zpracovat. Protože nepotřebujeme ukládat mnoho informací nebudeme vytvářet externí databázi a spolehneme se na Kafku. Při startu aplikace se načtou data z Kafka topiku ve kterém jsou události o platebních kartách do in-memory databáze (viz. EDA architektura sekce 2.3.2). Do tohoto topiku postupně přibývají další události o platebních kartách, které se zpracovávají a tím tak zůstává generátor účtenek aktuální.

Služba obsahuje pouze dva koncové body a to ověření že karta kterou spotřebitel zaplatil je evidovaná v systému a poslaní účtenky do systému. Zde si ukážeme důležité části kódu generátoru účtenek.

```
//Tato anotace rika ze se jedna o entitu v databazy
@Entity
public class PodporovaneKarty {
    //Tato anotace rekne hibernate
    //ze se tato promena ma pouzit jako primarni klic
    @Id
    private String hashKaretnichDetailu;
}
```

Zdrojový kód 5.1 Data potřebná k uchovávaní dat v generátoru účtenek

```
//Tato anotace řekne že se má tato třída používat ve Springu
@Service
```

```
public class UdalostiKaret {

^^I//Tento objekt zpracovává všechny karetní procesy
^^I@Autowired
^^Iprivate ProcessorKaret processorKaret;

^^I//Zde se nastaví jaky topik se má zpracovávat
^^I@KafkaListener(topics = "KaretniUdalosti",
^^I               groupId = "GeneratorUctenek")
^^I//Objekt KaretniUdalost má v sobe všechny potřebné informace
^^I//k zpracovaní karetní události
^^Ipublic void zpracovani(KaretniUdalost udalost){

^^I^^I//Tato funkce přidá a nebo odebere hashe karty které
^^I^^I//mohou být zpracovávány
^^I^^IprocessorKaret.zpracovaniKaretniUdalosti
^^I^^I    (udalost.getUdalostKarty(), udalost.getHashKarty());

^^I}
}
```

Zdrojový kód 5.2 Zpracovaní karetních událostí

```
//Tyto anotace řeknou že se má tato třída používat ve Springu jako REST api
@RestController
@RequestMapping("/uctenka")
public class UctenkovyController {
^^I//Tento objekt zpracovává všechny karetní procesy
    @Autowired
    private ProcessorKaret processorKaret;
    @Autowired
    //Tento objekt zpracovává všechny uctenkove procesy
    private ProcessorUctenek processorUctenek;

    //Tato anotace řekne že Spring má vytvořit REST GET koncový bod
    //Celkově tento koncový bod ověří že daný hash karty se nachází v systému
    @GetMapping(value = "/{hashKarty}")
    public ResponseEntity<Void> podporaKarty(@PathVariable String hashKarty) {
        boolean podporaKarty = processorKaret.podporaKarty(hashKarty);
        if(podporaKarty){
            return new ResponseEntity<>(HttpStatus.ACCEPTED);
        }else{
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);

        }
    }

    //Tato anotace řekne že Spring má vytvořit REST POST koncový bod
    //Celkově tento koncový bod posle účtenku do systému.
    //Systém tuto účtenku zpracuje a přiřadí spotřebiteli
    @PostMapping
    public ResponseEntity<Void> odeslaniUctenky(@RequestBody Uctenka uctenka) {
```

```
        processorUctenek.odelsatUctenku(uctenka);
        return new ResponseEntity<>(HttpStatus.ACCEPTED);
    }

}
```

Zdrojový kód 5.3 API generátoru účtenek

5.2 Správce účtenek

Jedna z nejhlavnějších služeb je správce účtenek. Když se však koukneme do technické dokumentace uvidíme velice jednoduchý set operací které musí podporovat. Jediná operace která nás zde bude zajímat je uložení účtenky a ohlášení že přišla účtenka uživateli.

Pro přehlednost jsem zde nebudeme ukazovat jak vypadá databázová struktura vypadala by vsak podstatě stejně jak byla data popsaná v tabulkách: 4.2, 4.4, 4.3, 4.5.

Účtenka se vygeneruje v generátoru účtenek ověří se a přiřadí se k ní uživatel. Účtenka která má přiraženého uživatele přijde do správce účtenek a ten ji následně uloží. Pokud dojde k úspěšnému uloženi účtenky systém pošle notifikaci o projmuté účtence.

```
//Tato anotace řekne že se má tato třída používat ve Springu
@Service
public class UdalostiUctenek {

^^I//Tento objekt zpracovává všechny účtenkové procesy
^^I@Autowired
^^Iprivate ProcessorUctenek processorUctenek;

^^I//Zde se nastaví jaký topik se má zpracovávat
^^I@KafkaListener(topics = "PrirazenaUctenka", groupId = "spravceUctenek")
^^I//Objekt PrirazenaUctenka má v sobě všechny potřebné informace
^^I//k uložení a odeslaní notifikace o příchozí účtence
^^Ipublic void zpracovani(PrirazenaUctenka prirazenaUctenka){

^^I^^I//Tato funkce uloží účtenku
^^I^^IprocessorUctenek.ulozitUctenku(prirazenaUctenka.getUctenka(),
^^I^^I                               prirazenaUctenka.getUzivatel());
^^I^^I//Tato funkce oznámí všem kteří poslouchají,
^^I^^I//že přišla uživatelovi účtenka
^^I^^IprocessorUctenek
^^I^^I    .oznameniOPrijetiUctenky(prirazenaUctenka.getUctenka(),
^^I^^I                             prirazenaUctenka.getUzivatel());

^^I}

}
```

Zdrojový kód 5.4 Zpracovaní účtenkových událostí správce účtenek

5.3 Správce zboží

Stejně jako správce účtenek i správce zboží je relativně jednoduchá služba která má na starosti relativně málo úloh. Pro tento proof of concept vyžadujeme pouze ověření že je dané zboží uloženo a kompletní v databázi. V případě že zboží v databázi není toto zboží se založí a nastaví se jako nekompletní. Informace o nekompletním zboží se následně pošlou obchodníkovy.

Stejně jako u správce účtenek ani zde nebudeme ukazovat databázoví model, vypadal by stejně jako u tabulek: 4.7, 4.6. Ukážeme zde pouze Kafka rozhraní pro přehled všeho co správce bude dělat.

```
//Tato anotace rekne ze se ma tato trida pouzivat ve Springu
@Service
public class UdalostiZbozi {

^^I//Tento objekt zpracovava vsechny procesy se zbozim
^^I@Autowired
^^IProcessorZbozi processorZbozi;

^^I//Zde se nastavi jaky topik se ma spracovavat
^^I@KafkaListener(topics = "PrijateUctenky", groupId = "spravceZbozi")
^^I//Objekt PrijataUctenka ma v sobe vsechny potrebne informace
^^I//k zpracovani prijate uctenky
^^Ipublic void zpracovani(PrijataUctenka prijataUctenka){
^^I^^I
^^I   for(Zbozi zbozi : prijataUctenka.getZbozi()){
            Optional<Zbozi> zboziPotrebneKDoplneni =
               processorZbozi.overeniUlozeniZbozi(zbozi);

            zboziPotrebneKDoplneni.ifPresent(
               i->processorZbozi.odeslatPozadavekODoplneniZbozi(
                  i,prijataUctenka.getObchodnik()
               )
            );
^^I   }
    }

}
```

Zdrojový kód 5.5 Zpracovaní událostí správce zboží

Kapitola 6
Závěr

Hlavním cílem této bakalářské práce bylo provést technickou analýzu projektu BillSaver, vytvořit funkční prototyp formou proof of concept a nakonec také zformovat studii proveditelnosti se zaměřením na technickou stránku projektu.

Výsledkem je část služeb potřebných k přijímání a skladování účtenek. Sada služeb je napsaná ve stylu mikroslužební architektury, jak je popsaná v technické analýze. Celá komunikace začíná v generátoru účtenek, jež se následně uskladní v manažeru účtenek, a poté se ověří dostupnost informací v manažeru zboží. Vytvořené služby mezi sebou komunikují přes Kafku.

Všechny služby jsou vytvořené pro SaaS prostředí, jsou tak nezávisle spustitelné a aktualizovatelné. Všechny zdrojové kódy jsou prozatím na soukromém GitLabu.

V následujících krocích tohoto projektu bude vytvořen vyhodnocovací systém, který bude vytvářet statistiky pro uživatele a obchodníky. Další důležitý krok představuje tvorbu UI komponenty, která umožní komunikaci uživatelů se systémem.

Všechny kroky a postup tvorby spolu s popisem jednotlivých služeb a jejich funkcionalitou jsou popsané v technické dokumentaci, jež pomůže se zahájením projektu a urychlí počáteční vývoj.

Ze studie proveditelnosti a plánování projektu jsme zjistili, že implementace bude trvat přibližně 1 rok a její cena se bude podle našeho modelu pohybovat okolo 12,3 milionů Kč. Bod zvratu nastane během 2,5 až 5,5 let a bude záviset na počtu uživatelů, objemu účtenek a množství podporovaných obchodníků. Provoz spolu s kontinuálním vývojem se bude podle našeho modelu pohybovat okolo 10,2 milionu ročně.

Bibliografie

1. KACVINSKÝ, Martin. *Agilné metodológie riadeniavývoja softwaru* [https://is.muni.cz/th/sjquf/kacvinsky-dp.pdf]. 2012. Dis. pr. Masarykova univerzita, Fakulta informatiky.
2. NUGROHO, Suryanto; HADI, Sigit; HAKIM, Luqman. Comparative Analysis of Software Development Methods between Parallel, V-Shaped and Iterative. *International Journal of Computer Applications*. 2017, roč. 169, č. 11, s. 7–11. ISSN 0975-8887. Dostupné z DOI: 10.5120/ijca2017914605.
3. BALAJI, Sundramoorthy; MURUGAIYAN, M Sundararajan. *Waterfall vs. V-Model vs. Agile: A comparative study on SDLC* [online]. 2012 [cit. 2021-12-10]. Č. 1. Dostupné z: https://mediaweb.saintleo.edu/Courses/COM430/M2Readings/WATEERFALLVs%20V-MODEL%20Vs%20AGILE%20A%20COMPARATIVE%20STUDY%20ON%20SDLC.pdf.
4. RERYCH, Markus. *Wasserfallmodell> Entstehungs context* [online]. 2007 [cit. 2021-12-10]. Dostupné z: http://cartoon.iguw.tuwien.ac.at/fit/fit01/wasserfall/entstehung.html#weltbild.
5. HOADLEY, Paul A. *Waterfall Model* [online]. https://commons.wikimedia.org, 2005-11 [cit. 2021-12-10]. Dostupné z: https://commons.wikimedia.org/wiki/File:Waterfall_model.png.
6. BRUYNINCKX, Herman. *V Model* [online]. https://commons.wikimedia.org, 2008-03 [cit. 2021-12-10]. Dostupné z: https://commons.wikimedia.org/wiki/File:V-model.svg.
7. BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifesto for Agile Software Development*. Manifesto for Agile Software Development [online]. 2001 [cit. 2021-12-10]. Dostupné z: http://www.agilemanifesto.org/.
8. ABRAHAMSSON, Pekka; SALO, Outi; RONKAINEN, Jussi; WARSTA, Juhani. *Agile Software Development Methods: Review and Analysis*. 2017. Dostupné z arXiv: 1709.08439 [cs.SE].
9. MAHALAKSHMI, M; SUNDARARAJAN, Mukund. *Traditional SDLC vs scrum methodology–a comparative study* [online]. Citeseer, 2013 [cit. 2021-12-12]. Č. 6. Dostupné z: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.413.2992&rep=rep1&type=pdf.
10. RATHOUSKÝ, Jakub. *ANALÝZA MODERNÍCH SOFTWAROVÝCH ARCHITEKTUR*. 2021. diplomathesis. CVUT FIT.

11. DOMARESKI, Henrique Siebert. *Monolithic & Microservices Architecture* [online]. Medium, 2021-05 [cit. 2021-12-13]. Dostupné z: https://henriquesd.medium.com/monolithic-microservices-architecture-239e8799d3e1.
12. AL-DEBAGY, Omar; MARTINEK, Peter. A Comparative Review of Microservices and Monolithic Architectures. In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2018, s. 000149–000154. Dostupné z DOI: 10.1109/CINTI.2018.8928192.
13. WOLFF, Eberhard. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016. ISBN 9780134650449.
14. QURESHI, M. Rizwan Jameel; SABIR, Fatima. *A comparison of model view controller and model view presenter*. 2014. Dostupné z arXiv: 1408.5786 [cs.SE].
15. XINFE, initial work by Deltacen. *MVC Diagram (Model-View-Controller)* [online]. wikimedia, 2013-05 [cit. 2021-12-16]. Dostupné z: https://commons.wikimedia.org/wiki/File:MVC_Diagram_(Model-View-Controller).svg.
16. KUMAR, Lakhwinder [online]. Softobiz Technologies India, 2021-02 [cit. 2021-12-16]. Dostupné z: https://softobiz.com/understanding-the-event-driven-architecture/.
17. CHANDY, K Mani. Event-driven applications: Costs, benefits and design approaches. *Gartner Application Integration and Web Services Summit*. 2006, roč. 2006.
18. MICHELSON, Brenda M. Event-driven architecture overview. *Patricia Seybold Group*. 2006, roč. 2, č. 12, s. 10–1571.
19. GARG, Nishant. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.
20. SINGH, Vijay. *What is frameworks? [definition] types offrameworks* [online]. https://hackr.io/, 2021-05 [cit. 2021-12-19]. Dostupné z: https://hackr.io/blog/what-is-frameworks.
21. SLYUSAR, Arthur [online]. LinkedIn, 2020-03 [cit. 2021-12-17]. Dostupné z: https://www.linkedin.com/pulse/what-best-fintech-programming-languages-your-project-arthur-slyusar/?articleId=6640212976618078208.
22. STAFF, UNOSQUARE. *6 top programming languages for Fintech* [online]. unosquare, 2021-08 [cit. 2021-12-17]. Dostupné z: https://blog.unosquare.com/6-top-programming-languages-for-fintech.
23. SANYAL, Sayantani. *Top 7 programming languages for Fintech and finance in 2021* [online]. 2021-07 [cit. 2021-12-17]. Dostupné z: https://www.analyticsinsight.net/top-7-programming-languages-for-fintech-and-finance-in-2021/.
24. TRIKHA, Ritika. *Emerging languages overshadowed by incumbents Java, python in coding interviews* [online]. 2016-08 [cit. 2021-12-17]. Dostupné z: https://blog.hackerrank.com/emerging-languages-still-overshadowed-by-incumbents-java-python-in-coding-interviews/.
25. BLOGGER, Guest. *5 of the best programming languages for Fintech* [online]. 2021-03 [cit. 2021-12-17]. Dostupné z: https://www.yoh.com/blog/5-of-the-best-programming-languages-for-fintech.
26. BELLSOFTWARE, Bellsoftware. *Coding languages for Finance and Fintech: Java, Kotlin and more* [online]. Bellsoftware, 2021-07 [cit. 2021-12-17]. Dostupné z: https://bell-sw.com/announcements/2021/01/05/Coding-Languages-for-Fintech/.
27. MAPLE, Simon; BINSTOCK, Andrew. *JVM Ecosystem Report 2018* [online]. 2018 [cit. 2021-12-19]. Dostupné z: https://res.cloudinary.com/snyk/image/upload/v1539774333/blog/jvm-ecosystem-report-2018.pdf.
28. MAPLE, Simon; BINSTOCK, Andrew. *JVM Ecosystem Report 2021* [online]. 2021 [cit. 2021-12-19]. Dostupné z: https://res.cloudinary.com/snyk/image/upload/v1623860216/reports/jvm-ecosystem-report-2021..

29. ROD, Johnson; JUERGEN, Hoeller; KEITH, Donald; COLIN, Sampaleanu; ROB, Harrop; THOMAS, Risberg; ALEF, Arendsen; DARREN, Davison; DMITRIY, Kopylenko; MARK, Pollack; THIERRY, Templier; ERWIN, Vervaet; PORTIA, Tung; BEN, Hale; ADRIAN, Colyer; JOHN, Lewis; COSTIN, Leau; MARK, Fisher; SAM, Brannen; RAMNIVAS, Laddad; ARJEN, Poutsma; CHRIS, Beams; TAREQ, Abedrabbo; ANDY, Clement; DAVE, Syer; OLIVER, Gierke; ROSSEN, Stoyanchev; PHILLIP, Webb; ROB, Winch; BRIAN, Clozel; STEPHANE, Nicoll; SEBASTIEN, Deleuze; JAY, Bryant; MARK, Paluch [online]. 2021-11 [cit. 2021-12-19]. Dostupné z: `https://docs.spring.io/spring-framework/docs/current/reference/html/`.
30. BERNARD, Emmanuel. *Announcing Quarkus 1.0* [online]. 2019-11 [cit. 2021-12-20]. Dostupné z: `https://quarkus.io/blog/announcing-quarkus-1-0/`.
31. SOTO, Alex. *Home of Quarkus Cheat-Sheet* [online]. Quarkus [cit. 2021-12-20]. Dostupné z: `https://lordofthejars.github.io/quarkus-cheat-sheet`.
32. *Engines ranking* [online]. 2021-12 [cit. 2021-12-20]. Dostupné z: `https://db-engines.com/en/ranking`.
33. MALIK, Ahsan; BURNEY, Aqil; AHMED, Fawad. *A Comparative Study of Unstructured Data with SQL and NO-SQL Database Management Systems* [online]. Scientific Research Publishing, Inc., 2020 [cit. 2021-12-20]. Č. 04. Dostupné z DOI: `10.4236/jcc.2020.84005`.
34. MORAVA, Roličky. *Nejlevnější Ekologické Kotoučky UŽ OD 10-Ti Kusů* [online] [cit. 2021-12-21]. Dostupné z: `https://www.rolickymorava.cz/kategorie/termo-kotoucky-ekologicke/`.
35. *Komentář Ke Statistice Platebního Styku* [online]. 2020 [cit. 2021-12-20]. Dostupné z: `https://www.cnb.cz/cs/statistika/menova_bankovni_stat/harm_stat_data/komentar-ke-statistice-platebniho-styku/index.html`.
36. KIPROP, Joseph. *How many trees does it take to make 1 ton of paper?* [Online]. WorldAtlas, 2018-11 [cit. 2021-12-21]. Dostupné z: `https://www.worldatlas.com/articles/how-many-trees-does-it-take-to-make-1-ton-of-paper.html`.
37. A.S., Severočeské vodovody a kanalizace. *Spotřeba Vody* [online]. 2019 [cit. 2021-12-21]. Dostupné z: `https://www.scvk.cz/vse-o-vode/pitna-voda/spotreba-vody/`.
38. *Population, total - czech republic* [online] [cit. 2021-12-20]. Dostupné z: `https://donnees.banquemondiale.org/indicateur/SP.POP.TOTL?locations=CZ`.
39. NEZNAMY, Neznamy. *Comparing API architectural styles: Soap vs rest vs graphql vs RPC* [online]. AltexSoft, 2020-05 [cit. 2021-12-20]. Dostupné z: `https://www.altexsoft.com/blog/soap-vs-rest-vs-graphql-vs-rpc/`.
40. DASCALU, Andrei. *Soap vs rest vs grpc vs graphql* [online]. DEV Community, 2021-05 [cit. 2021-12-20]. Dostupné z: `https://dev.to/andreidascalu/soap-vs-rest-vs-grpc-vs-graphql-1ib6`.
41. RRESELMA, Bob. *An architect's guide to apis: Soap, rest, GraphQL, and grpc* [online]. Red Hat, Inc., 2020-10 [cit. 2021-12-20]. Dostupné z: `https://www.redhat.com/architect/apis-soap-rest-graphql-grpc`.
42. CHANDAR, Vinoth; NIKHIL, Alok. *Benchmarking Kafka vs. Pulsar vs. Rabbitmq: Which is fastest?* [Online]. Confluent, 2020-08 [cit. 2021-12-20]. Dostupné z: `https://www.confluent.io/blog/kafka-fastest-messaging-system`.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src` ..... zdroj
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF