FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Rozšíření pokladního systému a jeho automatické nasazování do produkce přes CI/CD

Studijní program: AI – Aplikovaná informatika
Studijní obor: IT – Informační technologie
Autor práce: **Bc. Vancl David**
Vedoucí práce: Ing. Igor Kopetschke

Liberec 2024

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání diplomové práce

Rozšíření pokladního systému a jeho automatické nasazování do produkce přes CI/CD

Jméno a příjmení:	**Bc. David Vancl**
Osobní číslo:	M22000027
Studijní program:	N0613A140028 Informační technologie
Zadávající katedra:	Ústav nových technologií a aplikované informatiky
Akademický rok:	2023/2024

Zásady pro vypracování:

1. Na základě zpětné vazby uživatelů aktuálního systému proveďte rešerši a definici potřebných změn, úprav a rozšíření.
2. Navrhněte schéma rozhraní obecného frameworku pro dynamické přidávání dalších funkčních částí do pokladny bez dalších zásahů do aplikace.
3. Navrhněte a implementujte proces automatizace nasazování aplikace do produkce pomocí CI/CD včetně automatizovaných backendových i frontendových testů.
4. Nasaďte výsledný produkt do testovacího provozu a získejte zpětnou vazbu od uživatelů. Následně uveďte návrhy na další úpravy a rozšíření.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40 – 50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	čeština

Seznam odborné literatury:

[1] PORCELLO, Eve a Alex BANKS. Learning GraphQL: declarative data fletching for modern web apps. Sebastopol,CA: O´Reilly, 2018. ISBN 978-1-492-03071-3.

[2] GITHUB, INC. CI/CD: The what, why a nd how [online]. 2021 [cit. 2023-09-26]. Dostupné z: https://resources.github.com/ci-cd/

[3] PARMA, Pavel a Bohuslav KŘENA. Ověření možností migrace z architektury REST API do jazyka GraphQL. 2018.

[4] Banks, A., Porcello, E. (2020). Learning react: Modern Patterns for Developing React Apps. O'Reilly Media.

Vedoucí práce:	Ing. Igor Kopetschke Ústav nových technologií a aplikované informatiky
Datum zadání práce:	12. října 2023
Předpokládaný termín odevzdání:	14. května 2024

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. RNDr. Pavel Satrapa, Ph.D.
garant studijního programu

V Liberci dne 19. října 2023

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.
Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.
Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.
Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.
Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.
Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.
Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

8. 5. 2024 Bc. Vancl David

Rozšíření pokladního systému a jeho automatické nasazování do produkce přes CI/CD

Abstrakt

Tato diplomová práce se zaměřuje na rozvoj a optimalizaci pokladního systému vytvořeného v předchozí bakalářské práci. Hlavním cílem je rozšíření funkčnosti a implementace procesů Continuous Integration (CI) a Continuous Deployment (CD) pro automatizaci nasazování do produkčního prostředí. Práce analyzuje zpětnou vazbu od uživatelů, požadavky na systém a integruje nový platební terminál na základě rešerše trhu.

Významný důraz je kladen na propojení pokladního systému se skladovým systémem, optimalizaci procesů odečítání zásob, a zlepšení interakce mezi pokladnou a klienty prostřednictvím implementace navrhovaných vylepšení. Práce podrobně popisuje kroky automatizace nasazování, od přípravných procesů přes sestavování aplikace až po testování a nasazovací fáze.

Diplomová práce přináší komplexní pohled na moderní přístupy ve vývoji softwaru a demonstruje praktickou aplikaci CI/CD procesů na tomto pokladním systému, čímž zvyšuje jeho efektivitu, spolehlivost a adaptabilitu na měnící se požadavky trhu. Výsledky práce mají potenciál významně přispět k efektivnímu a dynamickému rozvoji pokladního řešení v praxi.

Klíčová slova: ReactJS, GraphQL, PHP, TS, JS, pokladna, platební terminál, CI/CD, Qt, C++, Java, Android

Extension of the Point of Sale system and its automatic deployment into production via CI/CD

Abstract

This thesis focuses on the development and optimization of the POS system created in the previous bachelor thesis. The main objective is to extend the functionality and implementation of Continuous Integration (CI) and Continuous Deployment (CD) processes to automate deployment into production environments. The thesis analyses user feedback, system requirements and integrates a new payment terminal based on market research.

Significant emphasis is placed on linking the POS system with the warehouse system, optimizing the inventory reading processes, and improving the interaction between the POS and the clients through the implementation of the proposed enhancements. The thesis details the steps of automation deployment, from the preparation processes through application build to the testing and completion phases.

The thesis provides a comprehensive view of modern approaches in software development and demonstrates the practical application of CI/CD processes on this POS system, thereby improving its efficiency, reliability and adaptability to changing market requirements. The results of this work have the potential to contribute significantly to the effective and dynamic development of e-commerce solutions in practice.

Keywords: ReactJS, GraphQL, PHP, TS, JS, point of sale, payment terminal, CI/CD, Qt, C++, Java, Android

Poděkování

Rád bych poděkoval vedoucímu diplomové práce Ing. Igoru Kopetschkemu za odbornou přípravu a metodologickou pomoc při zpracování mé práce.

Dále bych poděkoval celému koliktivu firmy WPJ s.r.o za dlouholetou trpělivost, cenné informace a příjemnou spolupráci.

Obsah

Seznam obrázků

Zdrojové kódy

Seznam zkratek

TCP *(Transmission Control Protocol)* komunikační protokol používaný pro spolehlivý přenos dat v počítačových sítích.

API *(Application Programming Interface)* sada pravidel a definicí, která umožňuje různým softwarovým aplikacím vzájemně komunikovat.

USB *(Universal Serial Bus)* je standardní technologie pro přenos dat a napájení mezi elektronickými zařízením.

SIM *(Subscriber Identity Module)* je integrovaný obvod, který se používá v mobilních telefonech.

CRA *(Create React App)* je nástroj od vývojářů pro ReactJS, který umožňuje vytvořit nový, konfigurovaný projekt.

NPM *(Node Package Manager)* je správce balíčků a závislostí pro JavaScript v prostředí Node.js.

E2E *(End-to-End)* je typ testování softwarových aplikací, který simuluje reálné interakce uživatele s aplikací z hlediska uživatele.

UI *(User Interface)* je prostředí, které umožňuje uživateli interakci s počítačovým programem nebo zařízením.

EET *(Elektronická evidence tržeb)* je systém povinné registrace tržeb.

HTML *(Hypertext Markup Language)* je značkovací jazyk používaný pro tvorbu a strukturování obsahu na webu.

URL *(Uniform Resource Locator)* je adresa používaná k identifikaci a přístupu k webovým stránkám na internetu.

CI *(Continuous Integration)* je proces kde se často integruje kód do sdíleného repozitáře s automatickými testy.

CD *(Continuous Deployment)* je proces, který zajišťuje, že veškerý kód, který prošel fází testování v rámci CI, je automaticky nasazen do produkčního prostředí.

DOM *(Document Object Model)* je objekt webové stránky, který reprezentuje jednu část struktury HTML nebo XML dokumentu.

EET *(Elektronická Evidence Tržeb)* je systém pro online evidenci tržeb podnikatelů.

1 Úvod

V dnešním rychle se měnícím technologickém světě je automatizace prostřednictvím Continuous Integration a Continuous Deployment nezbytná pro efektivní vývoj softwaru. Tento přístup umožňuje rychle integrovat a nasazovat kód, což značně zvyšuje produktivitu a snižuje možnost výskytu chyb. Díky automatizovaným testům a nasazení se vývoj může soustředit na inovace a optimalizaci pokladny místo rutinních úkolů. Klíčová pro moderní obchodní operace je implementace skladového systému do pokladny, neboť zajišťuje bezproblémovou synchronizaci mezi prodejem zboží a jeho skladovými zásobami. Tato integrace umožňuje okamžité aktualizace stavu zásob při každém prodeji, což vede k přesnějšímu sledování dostupnosti produktů a efektivnějšímu řízení.

V diplomové práci je rešerše zaměřena na analýzu reakcí klientů a jejich návrhy na vylepšení, což je zásadní pro optimalizaci služeb a zvyšování spokojenosti zákazníků. Dále jsou zkoumána systémová vylepšení, která by mohla přinést efektivnější a hladší běh operací. Navazující kapitola se zabývá implementací těchto vylepšení, které zahrnují široký rozsah aktualizací a inovací, od integrace nového platebního terminálu, které zavádí komunikaci mezi pokladnou a terminálem, až po rozšířenou podporu pro čtečky. Dále je v práci popsán a implementován systému slovenského EET, což je důležité pro splnění legislativních požadavků na tamním trhu. Implementace vylepšeného mechanizmu zaplacení a procesu příjmu zboží přes pokladnu přispívají k efektivnějšímu a plynulejšímu obchodnímu provozu. Také se klade důraz na nastavení oprávnění pro administrátory, což zlepšuje bezpečnost a správu systému.

Další velmi důležitou částí práce je implementace dynamického přidávání funkčních částí do pokladního systému, což umožňuje flexibilní integraci nových komponent bez nutnosti zásadních zásahů do stávajícího systému. Toto je demonstrováno na schématech a funkcích mechanizmu, které jsou implementovány jak na straně serveru, tak v samotné pokladně.

Celkově diplomová práce představuje komplexní přístup k rozvoji pokladního systému s využitím nejnovějších technologií a metodik v oblasti softwarového inženýrství, které zabezpečují jeho efektivitu, spolehlivost a schopnost rychlé adaptace na měnící se požadavky trhu.

2 Rešerše a průzkum trhu

V první části se kapitola zaměřuje na **analýzu zpětné vazby** od zákazníků a identifikaci **požadavků na systémová vylepšení**. Rešerše dále navazuje na poznatky získané v první části. Na základě požadavku od klienta je realizován **průzkum trhu platebních terminálů**. V souvislosti se systémovými vylepšeními je rešerše zaměřena na implementaci **automatizace prostřednictvím CI/CD**.

2.1 Analýza zpětné vazby

Zpětná vazba od klientů byla shromážděna prostřednictvím různých komunikačních platforem. Návrhy na nové funkce byly předány prostřednictvím **e-mailu**. Vylepšení existujících funkcí a opravy byly identifikovány prostřednictvím firemní **zákaznické sekce**. Někteří klienti byli přímo požádáni o testování pokladního systému. Své výsledky testování zaznamenávali do nově vytvořené tabule *(boardu)* v online aplikaci **Trello**. V případě nejasností nebo potřeby dodatečných změn probíhala komunikace **telefonicky**. Následující podsekce obsahuje seznam klientů, od kterých pocházely jednotlivé návrhy na úpravy a případná vylepšení.

2.1.1 Zpětná vazba od klientů a jejich návrhy na vylepšení

eJuice *"www.ejuice.cz"*

EJuice se specializuje na prodej elektronických cigaret, náplní (e-liquidů) a příslušenství pro vaping. Ejuice využívá pro rozlišení kamenných prodejen několik instancí pokladny. Klient požaduje, aby prodavači a prodavačky měli omezený přístup do systému. Je tedy nezbytné do pokladního systému přidat **oprávnění**, která umožní nastavovat specifická přístupová práva jednotlivým zaměstnancům. Oprávnění by se měla především vztahovat na nastavování slev, kupónů, stornování objednávky a další dodatečné funkce.

V situaci, kdy zákazník chce vrátit zboží, mohou zaměstnanci s omezeným přístupem do administrace e-shopu a pokladny narazit na obtíže. Proto dalším požadavkem je realizovat **příjem zboží** skrze pokladnu. I v tomto případě bude nutné přidat funkci pod oprávnění.

Jednou z klíčových funkcí pokladního systému je možnost vydávat online objednávky, které byly vytvořeny v e-shopu. Vzhledem k tomu, že klient používá více pokladen zároveň, tak požaduje, aby systém umožňoval **filtrování objednávek podle konkrétní prodejny**.

Od tohoto klienta tedy vycházejí tři klíčové požadavky na změny:

- Oprávnění
- Příjem zboží
- Filtrovnání objednávek

Kupkolo *"www.kupkolo.cz"*

Kupkolo se zaměřuje na prodej širokého spektra jízdních kol včetně horských, silničních, crossových, gravel bike, dětských kol a elektrokol. V případě prodeje mechanických dílů bývají jednotlivé součásti označovány sériovým číslem. Sériová čísla jsou jedinečné identifikační kódy přiřazené jednotlivým kusům výrobků nebo součástem, které výrobci používají k jejich sledování a identifikaci. Mít **možnost vybrat sériové číslo** přiřazené k produktu při jeho vložení do košíku je důležité pro zajištění, že zákazníci obdrží přesně specifikovaný výrobek, zejména v případech, kdy jsou produkty stejného typu ale s různými sériovými čísly odlišné kvůli výrobním dávkám, speciálním edicím, nebo různým konfiguracím.

Klient požaduje, aby byly u objednávek zobrazovány jejich čárové kódy. Pokud jsou tyto kódy dlouhé, jejich ruční přepisování se stává náročným a časově zdlouhavým. Pokud je **čárový kód vytisknut**, jeho načtení a manipulaci na skladu může efektivně zvládnout čtečka.

Od tohoto klienta proto pramení tyto požadované úpravy:

- Podpora zadávání sériových čísel
- Tisk/vykreslení čárových kódů objednávky

Xtreme *"www.xtreme.sk"* a Sadesport *"www.sadesport.sk"*

Xtremese specializuje na prodej širokého spektra outdoorového a sportovního vybavení pro různé druhy aktivit. Sadesport nabízí široký výběr jízdních kol, elektrokol, cyklodoplňků a cyklooblečení. Oba klienti mají sídlo na Slovensku, a tudíž zde také provozují své pokladny.

Před prvním nasazením pokladny pro slovenské klienty bylo nezbytné implementovat podporu pro jinou **měnu**, v tomto případě pro Euro. Po uvedení do provozu klienti nezávisle na sobě požadovali, aby do pokladního systému byla přidána **podpora pro slovenskou elektronickou evidenci tržeb**. Na rozdíl od České republiky, kde bylo EET zcela zrušeno s účinností od 1. ledna 2023, na Slovensku systém EET nadále funguje.

Změny vycházející od těchto klientů jsou:

- Podpora různých měn v pokladně *(v tomto případě Euro)*
- Implementace napojení na SK EET

Bjež *"www.bjez.cz"*

Bjež nabízí široký sortiment čelenek, čepic, nákrčníků, oblečení a doplňků pro běh, kolo a další aktivity.

Od tohoto klienta přišel pouze jeden požadavek. Tento požadavek se týká stránky se zaplacením objednávky. Pokud se zákazník rozhodne pro platbu hotově a následně zjistí, že nemá dost peněz, v systému pokladny již není možné vrátit se zpět a změnit způsob platby. Jediným řešením v této situaci je zrušení původní platby a vytvoření nové platby s upraveným způsobem platby. Při výběru platby se objednávka ukládá do systému a tím získává vygenerované číslo, které se následně přepisuje na platební terminál. Zákazník vyžaduje, aby bylo možné v pokladně jednoduše **rozlišit akce, vybrat typ platby a následně přijmout hotovost**.

Klient tedy požaduje změnu:

- Možnost změnit typ platby po výběru

KrmímKvalitně *"www.krmimkvalitne.cz"*

KrmímKvalitně nabízí široký sortiment krmiv a doplňků pro psy a kočky. Od tohoto klienta přišel požadavek na dvě menší úpravy. První úprava se zaměřuje na **zlepšení systému pro vyhledávání objednávek**. V současné době je funkce pro vyhledávání objednávek skryta v rozbalovacím menu. Na hlavním panelu je možné vyhledávat pouze produkty. Klient uváděl, že když zadali číslo objednávky do vyhledávacího panelu určeného pro produkty, neobdrželi očekávaný výsledek. Tato situace ukázala potřebu pro optimalizaci vyhledávacího rozhraní.

Druhá úprava se soustředí na vylepšení sekce pro rychlý přístup. V pokladně existuje sekce rychlého přístupu, do které lze v nastavení přiřadit vybrané produkty. Tato funkce umožňuje rychlejší a efektivnější práci s nejčastěji prodávanými nebo preferovanými produkty. Klient vyjádřil požadavek na **rozšíření sekce rychlého přístupu o funkci pro přidávání textových položek**. Klienti využívají textové položky místo produktů z několika důvodů, například pro přidaní "igelitové tašky", uplatnění "dodatečné slevy" nebo specifikaci služeb např. navážení granulí a jejich zabalení. Pokladní systém sice umožňuje zadávat a zaznamenávat textové položky do objednávky, ale dosud nemá funkci, která by tyto položky umožňovala ukládat pro budoucí rychlý přístup.

Zde je seznam změn, které klient navrhl:

- Vyhledávání objednávek na hlavním panelu.
- Rozšíření rychlého přístupu o textové položky.

HudebníCentrum *"www.hudebnicentrum.cz"*

E-shop HudebníCentrum nabízí široký sortiment hudebních nástrojů, zvukové techniky a nahrávacího vybavení, včetně kytar, kláves, bicích a dalšího příslušenství pro hudebníky.

Od tohoto klienta přišel jen jeden požadavek, který se však týká významného vylepšení. Požadavek od klienta se zaměřuje na začlenění platebního terminálu přímo do pokladního systému. V současné době si klienti zřizují vlastní platební terminály s individuálními podmínkami od poskytovatelů platebních služeb. Po vytvoření objednávky v pokladním systému zákazníci získají číslo objednávky, které následně musí ručně zadávat do platebního terminálu. Tento postup platby, kdy je nutné ručně přepisovat číslo objednávky z pokladního systému do samostatného platebního terminálu, je běžný u mnoha dostupných pokladních řešení na trhu. Klient navrhl **možnost přímého propojení platebního terminálu s pokladním systémem**, aby se zjednodušil proces platby. K vyhodnocení tohoto požadavku bylo nezbytné provést rešerši dostupných platebních terminálů, aby se zjistilo, zda je technicky možné takové propojení realizovat. Pokračování rešerše platebních terminálů naleznete v sekci 2.2.

Výsledek požadavku je následující:

- Podpora pro platební terminál.

AutoMotoVelo *"www.automotovelo.cz"*

Tento e-shop se specializuje na prodej produktů souvisejících s automobily, motocykly a historickými vozidly veteránů. Je to jeden z prvních klientů, kteří se zapojili do počátečního testování pokladního systému. Díky tomuto klientovi byla **opravena řada chyb**, **překlepů** a **grafických nedostatků**. Tyto chyby byly opraveny ihned po jejich nahlášení.

Zde je seznam některých chyb, které se v pokladním systému objevily. V seznamu jsou citovány přímo chyby zadané klientem:

- *"Po vyhledání (textovém ne čtečkou) produktu a kliknutí na něj se produkt přidá v pravé straně do seznamu, ale na levé zůstane celé vyhledávání viset a nesmaže se."*
- *"Při výpisu více položek je tam chyba, přepisuje se text přes sebe, screenshot přikládám níže."*
- *"Ikonka s křížkem k odebrání položky by měla být na konci každého řádku ne v rozklikávacím podmenu."*
- *"Při přidání položek v pokladně se vytvoří automaticky objednávka v e-shopu a to bez uložení!"*
- *"Nejde se vrátit do pokladny, musí se znovu kliknout na ikonku nahoře. Tlačítko zpět by bylo fajn nebo když kliknu na záložku Objednávka 1, dostanu se zpět do pokladny."*
- *"Seznam položek nákupu - pravá strana - není tam možnost změnit DPH na 10% pokud se jedná o servis."*

MushGO (*www.mushgo.cz*)

Nejnovějším klientem, kterému byla pokladna implementována, je e-shop MushGO. S tímto klientem teprve budeme pracovat na zpětné vazbě a na základě toho provádět případné úpravy nebo další vylepšení. Prozatím nebyla získána žádná zpětná vazba.

2.1.2 Systémová vylepšení

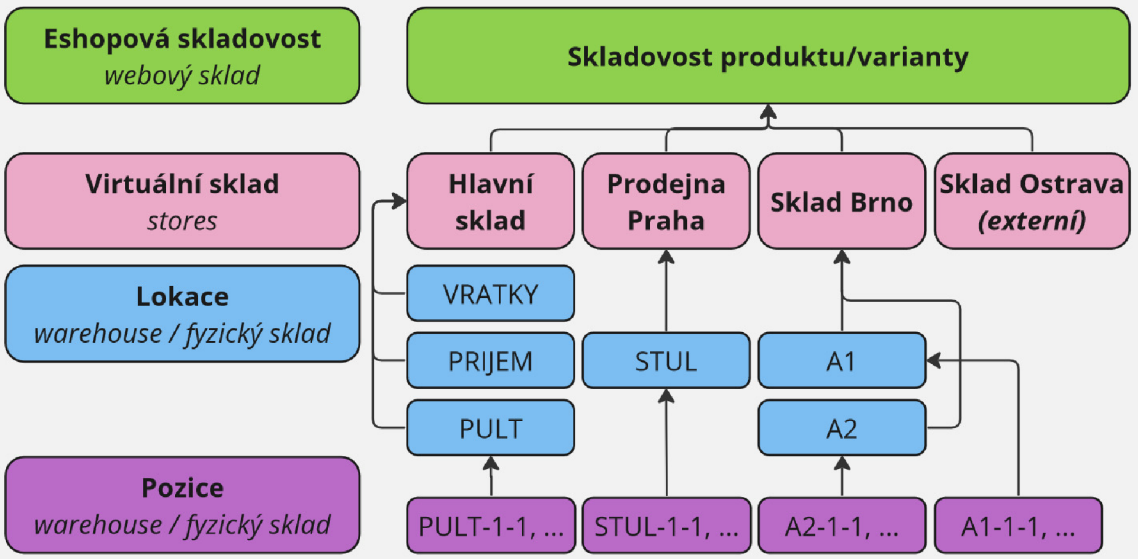
Vylepšení systému jsou primárně odvozena z původní implementace pokladního systému. Některé komponenty se pro zprovoznění ukázaly jako nedostatečné, jiné jako omezující, zatímco některé by představovaly užitečnou podporu.

Podpora skladového systému

Skladový systém od společnosti wpj je složen z několika vrstev. Na nejvyšší úrovni se umisťuje **webový sklad**. Tato vrstva odráží množství produktů v kontextu dostupnosti pro e-shop. Pokud je objednávka vytvořena, kusy jsou okamžitě odečteny. Pokud dojde ke stornu objednávky, kusy jsou ihned vráceny na sklad. Druhou vrstvou je **virtuální sklad**, který zaobaluje sklad fyzický a poskytuje souhrnné informace o jeho stavu. Tato vrstva nabízí řadu dalších funkcí, které nijak nezasahují do cíle práce a proto nebudou podrobněji rozvedeny. Třetí vrstvou je **fyzický sklad**. Fyzický sklad je tvořen lokacemi, jako jsou skladové budovy, a pozicemi, například regály ve skladu. Úroveň fyzického skladu odráží celkový počet přes všechny pozice přiřazené k lokacím. Zboží je přičítáno nebo odečítáno až v okamžiku vyskladnění/naskladnění, nebo v případě objednávky při zabalení balíku (vyprázdnění přípravného boxu).

K těmto třem základním úrovním jsou přidány další mezivrstvy. Jedním z příkladů může být *externí sklad*, který je považován za fyzický, ale není z něj možné nakupovat.

Následující obrázek 2.1 zobrazuje schéma, které popisuje propojení mezi jednotlivými částmi skladu. Je zřejmé, že pro správný provoz pokladny je nezbytné, aby byla implementována funkčnost pro správně odečítání zásob a pohybu zboží na skladě.



Obrázek 2.1: Schéma propojení skladového systému

Dynamické přidávání funkčních částí

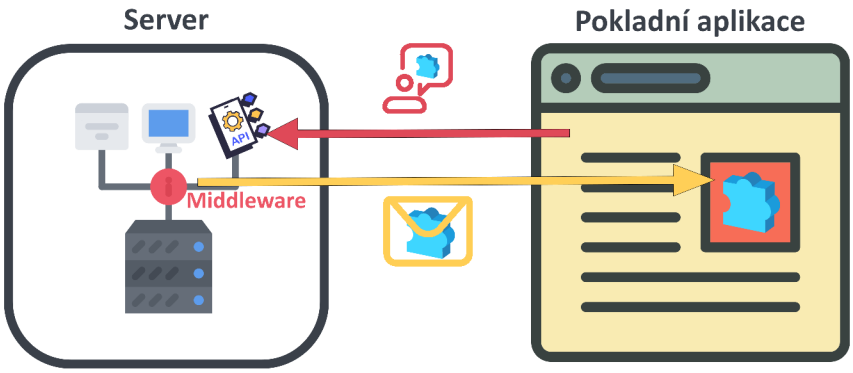
Modul by měl poskytovat flexibilní a rozšiřitelnou platformu pro integraci nových funkcí a služeb do stávajícího pokladního systému bez nutnosti jeho kompletního přepracování nebo významného zásahu do kódu.

Pokladní modul je implementován jako součást e-shopového řešení, využívající framework ReactJS. Po prozkoumání různých metod pro implementaci funkce dynamického přidávání jednotlivých částí se ukazují jako nejvhodnější tyto dva přístupy. První, poněkud komplexnější metodou, je generování HTML na základě předem definovaných pravidel. Nejnáročnější částí by pak byla komunikace se serverem.

Druhým, pravděpodobně nejuniverzálnějším řešením, je zavedení middleware[1] na straně serveru, který bude zodpovědný za vykreslování jednotlivých funkčních částí. Tyto části budou v pokladně zobrazeny jako iframe [2]. Díky jazyku JavaScript lze následně snadno zajistit komunikaci mezi tímto iframe a samotnou aplikací. Na serverové straně je možné vytvořit různé uživatelské prvky, jako jsou formuláře, seznamy, tabulky nebo jiné funkční komponenty, které budou dynamicky integrovány do pokladního systému. Hlavní předností tohoto systému je možnost dodat zákazníkům na míru určenou funkci do pokladny pouze prostřednictvím upravení šablony a vytvoření jednoduché PHP třídy, aniž by bylo nutné jakkoliv měnit samotnou pokladnu. Tímto způsobem je možné udržet jednotnou verzi pro všechny klienty. Další výhodou tohoto přístupu je, že i kolega, který nemusí být obeznámen s jazykem TypeScript nebo knihovnou ReactJS, bude schopen přidávat nové funkce do pokladny. Zásah do kódu pokladní aplikace by byl potřebný jen ve výjimečných případech, kdy by bylo nutné synchronizovat pokladní okno s iframe. Níže uvedený obrázek 2.2 představuje drátěný model, který ilustruje možný vzhled tohoto chování.

[1] Middleware je software sloužící jako prostředník mezi různými aplikacemi nebo komponentami systému, usnadňující jejich vzájemnou komunikaci a výměnu dat.

[2] Iframe je HTML element, který umožňuje vložení jiné webové stránky dovnitř aktuální stránky.



Obrázek 2.2: Drátěný model dynamického přidávání funkčních částí

Automatizované sestavování, nasazování do produkce a testování

Automatizace prostřednictvím CI/CD je klíčovým vylepšením, které bylo navrženo na základě zkušeností z první verze pokladního systému. Aktuálně je potřeba provést mnoho kroků při implementaci změn, a očekává se, že jejich počet bude v budoucnu narůstat. Níže naleznete seznam kroků nutných nebo vhodných k provedení před nasazením změn do produkce.

- Formát kódu
- Typová kontrola
- Build aplikace
- Testy front-end a back-end
- Nahrát source mapy do Sentry [3]
- Nasadit novou verzi klientovi

Vlastní sekce rešerše 2.3.1 je věnována popisu tohoto procesu, včetně nezbytných úkonů a přehledu dostupných technologií pro automatizaci.

Rozšířená podpora pro čtečku

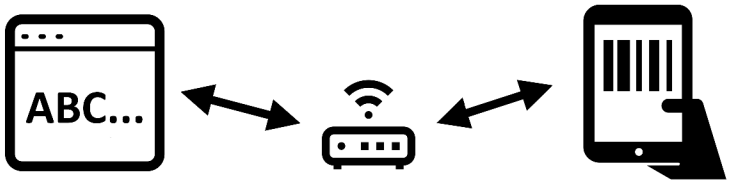
V první etapě rozvoje podpory pro čtečku čárových kódů bylo hlavním cílem rozšířit možnosti načítání dat do pokladního systému. Dosud bylo možné skenovat pouze produkty/varianty a slevy. Nyní je požadováno, aby systém podporoval také skenování objednávek, uživatelů, sériových čísel a dalších dat.

Klienti používají čtečku čárových kódů připojenou k stolnímu počítači, která je prostřednictvím windows aplikace integrována do pokladního systému. Komunikace mezi čtečkou a pokladnou probíhá skrze takzvaný pub-sub mechanismus zpráv,

[3] Sentry je nástroj pro sledování chyb v aplikacích, který umožňuje vývojářům identifikovat, monitorovat a opravovat problémy.

při kterém čtečka odesílá zprávu obsahující informace o skenovaných datech. V pokladně je odběratel, který tuto zprávu zachytí a extrahuje z ní údaje získané skenováním.

Při vývoji může nastat situace, kdy vývojář nemá fyzický přístup k čtečce čárových kódů. Jedním z řešení je simulace odesílání zpráv v konzoli prohlížeče, což však vyžaduje manuální zadávání čárového kódu a může být zdlouhavé. Po zvážení různých možností, jak tento problém vyřešit, je z dlouhodobého hlediska vytvořit android aplikaci, která se dokáže připojit k prohlížeči a umožnit tak automatické čtení a vkládání čárových kódů.



Obrázek 2.3: Alternativní metoda načítání čárových kódů

2.2 Dostupné platební terminály

Platební terminál je zařízení používané k provádění online plateb za zboží nebo služby. Terminál umožňuje zákazníkům platit pomocí různých platebních karet, telefonu, hodinek nebo jiných elektronických metod.

Na trhu existuje mnoho platebních terminálů specifických pro jednotlivé banky, což představuje nevýhodu při implementaci, pokud má každý zákazník jinou banku. V takovém případě by bylo nezbytné pro každou banku implementovat samostatné řešení, což je časově i finančně náročné, vzhledem k nutnosti každou takovou implementaci certifikovat. Proto je klíčovým požadavkem při výběru terminálu zvolit univerzální řešení, které se bude schopné ideálně připojit k libovolné bance. Samotný výběr takového terminálu dále závisí na několika důležitých faktorech. Zde jsou některé z nejdůležitějších faktorů, které jsou při výběru platebního terminálu brány v úvahu (vychází z [20]):

- **Poplatky za terminál a transakce** jsou jedny z nejdůležitější vlastností při výběru terminálu. Poplatky za platební terminál/transakce se mohou skládat z měsíčního poplatku, poplatku za transakci, poplatek za přenos dat, storno poplatky a další dodatečné (účtenky, údržba, ...).

- **Typy platebních terminálu** pokrývají široké spektrum použití. Zde je několik druhů terminálů, které jsou určeny pro různé činnosti: stacionární (připojené napevno k pokladně), přenosné (bezdrátový terminál, např. uvnitř restaurace), mobilní (určený například pro veletrhy, používají jiný typ komunikace), virtuální (bez přítomnosti platební karty, např. zadáním údajů), self-service (samoobslužné, nevyžadují přítomnost obsluhy)

- **Příchod peněz na účet** není omezující vlastností, ale je dobré mít alespoň přehled kdy je garantován příchod obnosu peněz na účet podniku. Některé terminály mohou platby pozdržet kvůli validaci či ověření platby.
- **Propojení terminálu s pokladním systémem** je důležité především pro vývojáře, který bude terminál napojovat. Nicméně je nutností zjistit, zda je vůbec technicky přijatelné implementovat napojení terminálu ještě před zřízením bankovních smluv.
- **Multiměnový terminál** je terminál s rozšířenou podporou pro cizí měny. Pokud je předpoklad využití pokladny například v Česku i na Slovensku, tak musí terminál podporovat platby v Korunách i Eurech, popřípadě provádět potřebné převody.
- **Další dodatečné vlastnosti** mohou zahrnovat nastavování spropitného, aplikace elektronických stravenek, stornovaní transakcí a mnohé další.

Z analýzy trhu vyplynuly jako nejpříznivější řešení tyto možnosti. Vlastnosti těchto terminálů a jejich možnost implementace k pokladnímu systému je detailněji popsána v následujících podsekcích.

2.2.1 Comgate: Platební brána a terminál

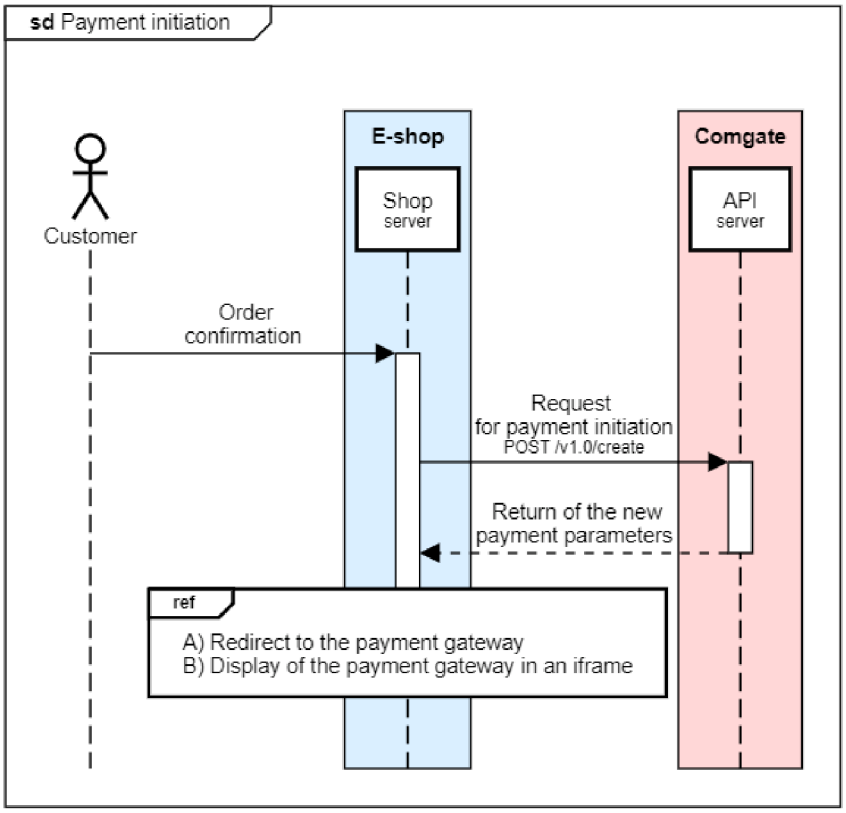
Společnost Comgate přichází nejen s platebním terminálem, ale i samotnou platební bránou. Služby zahrnují také pokročilé funkce pro správu plateb, reportování a analýzy. Comgate klade důraz na bezpečnost plateb a splňuje mezinárodní bezpečnostní standardy pro zpracování platebních karet.

Nabízeny jsou tři základní tarify. První tarif **Start** je určen menším podnikatelům, nebo začínajícím obchodům, které doposud nevyužívali platby kartou. Výhodu u toho tarifu jsou poplatky až od cenové hladiny. Druhý tarif **Easy** nabízí především fixní poplatky. Cílen je na již zaběhnuté obchodníky s ustáleným ziskem, ale nikoliv transakce dosahujících vyšších hodnot. Třetí tarif **Profi** rozděluje poplatky pro jednotlivé druhy karet, kupónů, stravenek atd. Je tedy především na zákazníkovi, zda v tomto tarifu najde výhodné uplatnění. Porovnání nejzajímavějších údajů je k dispozici v tabulce 2.2.1. Detailnější porovnání lze nalézt na stránkách Comgate [2].

Obrázek 2.4: Platební terminál od společnosti Comgate (Zdroj: [1])

	Start	Easy	Profi
Platební karty	do 50 tis. Kč zdarma, poté dle **Easy**	0,9%+1Kč [4]	individuálně, v EU: 0,65%+0,7 Kč, mimo EU: 1,15%+3 Kč
Stravenkové karty	0,9%+1 Kč	0,9%+1 Kč	individuálně, např. Lidl: 2,50%+0 Kč
Pronájem terminálu	zdarma (6 měsíců), dále dle **Easy**	dle přijmu, do 190 Kč	dle přijmu, do 190 Kč
Provoz služby	zdarma	zdarma	dle přijmu, do 200 Kč

Platební terminál je možné připojit pomocí WiFi, nebo Ethernet kabelu. K implementaci je dle dokumentace dostupné API, které zpracovává požadavky ze strany pokladny (serveru komunikujícího s pokladním systémem). Toto API je umístěno na serveru na straně Comgate, které dále pracuje s platebním terminálem. Nejedá se tedy o přímé napojení na terminál. Obrázek 2.5 zobrazuje jak by mohla být provedena implementace.



Obrázek 2.5: Schéma napojení na Comgate API (Zdroj: [1])

[4]Sazba zapsána ve tvaru **0,9% + 1 Kč** znamená 0,9% z uhrazované ceny + 1Kč

2.2.2 FiskalPRO: Pokladní řešení EET a platební terminály 3 v 1

FiskalPRO nabízí širokou škálu řešení platebních terminálů včetně stacionárních a mobilních zařízení. FiskalPRO je vhodnou volbou pro obchodníky hledající komplexní řešení spojující platební terminály s pokladními systémy.

Společnost nemá jednoznačně určené tarify. Zaměřuje se individuální vyhodnocení spolupráce a následné stanovení podmínek. FiskalPRO vychází ze standardně používaného modelu MIF++. [5] Po vyžádání detailnějších informací skrze emailovou komunikaci může spolupráce vypadat například takto:

průměrný obrat karta/měsíc/terminál	**procentuální sazba**
0 - 50 tis. Kč	0,99%
50 - 100 tis. Kč	0,89%
100 - 200 tis. Kč	0,79%
nad 200 tis. Kč	0,69% nebo individuálně

FiskalPRO nabízí různé modely platebních terminálů a pokladen, které jsou vhodné pro různé druhy zpracování:

- **FiskalPRO N3** - Univerzální zařízení kombinující čtečku čárových kódů, platební terminál a Qr kódy.
- **FiskalPRO N5** - Stejný druh zařízení, který nahrazuje čtečku za tiskárnu dokladů.
- **FiskalPRO N86** - Odlehčená verze. Pouze platební terminál s pokladním řešením.
- **FiskalPRO Orange sestava** - Celý systém pokladny, který je složen z terminálu, čtečky, tabletu, tiskárny dokladů a další.
- **FiskalPRO T6** - Samostatný platební terminál. Stacionární řešení vhodné k napojení na externí pokladnu.

Obrázek 2.6: Platební terminál od společnosti FiskalPRO (Zdroj: [15])

[5] MIF++ je model stanovení cen za transakce platebními kartami, který kombinuje mezibankovní poplatek, poplatek kartové společnosti a zpracovatelský poplatek, čímž obchodníkům poskytuje jasný přehled o celkových nákladech na akceptaci platebních karet.

Platební terminál FiskalPRO T6 je připojen do sítě pomocí ethernetového/USB kabelu. Terminál lze nastavit do několika módů. Pro napojení na pokladnu je vyžadován režim *klienta*. V tomto režimu terminál komunikuje pomocí TCP protokolu. Datový příkaz vypadá takto:

```
1 <Startovací znak1><Startovací znak2><Délka1><Délka2><Data><LRC>
```

Zdrojový kód 2.1: Příklad komunikace s terminálem dle dokumentace

2.2.3 Fio: Platební terminály a Platební brána

Fio banka poskytuje stacionární a přenosné platební terminály značky Ingenico, připojené k internetu buď přes ethernetový kabel nebo datovou SIM kartou.

Podmínky a sazby jsou přizpůsobeny individuálním potřebám klienta. Zde je přehled některých dostupných informací:

Poplatky za transakci	jednotná sazba pro karty
Poplatek na 500 000 Kč/měsíc	1,02 % pro běžné karty, + 1 % pro karty vydané mimo EU
Poplatek za terminál	Minimální výše poplatku 800 Kč
Instalace terminálu	Zdarma
Multiměnový terminál	Není podporován

V nabídce jsou dva druhy terminálů. Přenosný **Move 2500** pracuje pomocí datové SIM karty. U tohoto typu terminálů můžou být účtovány dodatečné poplatky za datový tarif. Druhým typem je stacionární **Desk 3200**, který je do sítě připojen pomocí ethernetového kabelu. Oba terminály využívají ke komunikaci API od Fio banky. Tedy i v tomto případě je komunikace mezi pokladnou a platebním terminálem obcházena skrze serverové API. Tyto informace vycházejí z webu a dokumentace Fio [4].

Obrázek 2.7: Platební terminál od FioBanky (Zdroj: [4])

2.2.4 Vybraný platební terminál pro implementaci

Na trhu existují i další řešení pokladních terminálů. Některé nelze samostatně implementovat k pokladnímu systému, jiné jsou vázány k určité bance nebo jsou limitována jinými faktory.

Jako nejvhodnější kandidát pro implementaci platebního terminálu se jeví řešení od **FiskalPro**. Klienti, kteří používají pokladnu od WPJ (pokladnu vytvořenou v rámci bakalářské práce), mají široké spektrum zisku. Je tedy výhodné nechat si klienty vyjednat individuální podmínky pro jejich podnikání, než se vázat na fixní poplatky. Druhým pozitivem je možnost pracovat na přímo s terminálem z pokladny. Není nutné komunikovat nejprve s API na straně serveru, která nakonec komunikaci s terminálem provede. V tomto případě terminál po obdržení požadavku sám provede zabezpečenou komunikaci s potřebnými servery.

Po vyjednání spolupráce s firmou FiskalPRO byl zaslán platební terminál *FiskalPRO T6*. Po obdržení terminálu byl ze strany FiskalPRO zřízen testovací sandbox[6], testovací údaje a potřebná dokumentace k implementaci.

2.3 Proces automatizace pomocí CI/CD

Zkratky CI a CD označují procesy, které se zaměřují na automatizaci fází vývoje softwaru, zejména na integraci *(CI, Continuous Integration)* a nasazování *(CD, Continuous Deployment)*. Tyto praktiky umožňují rychlejší, efektivnější a spolehlivější vývoj a nasazení softwaru.

Část pojmenovaná CI je proces, ve kterém vývojáři pravidelně (několikrát denně) integrují svůj kód do sdíleného repozitáře. S každou takovou integrací automaticky probíhá sada předdefinovaných testů, které pomáhají rychle odhalit a opravit chyby, zlepšují kvalitu kódu a snižují čas potřebný k integraci změn. CD je rozšíření CI, které zahrnuje automatizované nasazování všech změn kódu do testovacího prostředí po úspěšném projití testů.

Existuje několik platforem, které podporují zpracování pomocí CI/CD. Mezi nejrozšířenější patří GitLab a GitHub. Zde je několik dalších technologií, které podporují automatické zpracování (mohou se lišit, ale v konečné fázi jde o CI/CD): *Bitbucket, Azure DevOps, AWS CodeCommit* a další. Vzhledem k tomu, že pokladna je uložena na firemním GitLabu, bude následující rešerše zaměřena výhradně na něj.

2.3.1 Jednotlivé kroky automatizace

Automatizovaný proces skládající se z různých kroků, jako jsou sestavení kódu, testování, a nasazování se nazývá **pipeline**. Zpracování tohoto procesu lze popsat pomocí následujících kroků:

[6] Testovací "sandbox" je bezpečné, izolované prostředí, které vývojářům umožňuje experimentovat s novým kódem.

1. **Definice** - Celá definice jednotlivých kroků je umístěna v kořenovém adresáři v souboru *.gitlab-ci.yml*. Tento soubor specifikuje jednotlivé úlohy (jobs), které se mají vykonat, jejich pořadí, podmínky spuštění a prostředí, ve kterých se mají úlohy spustit.

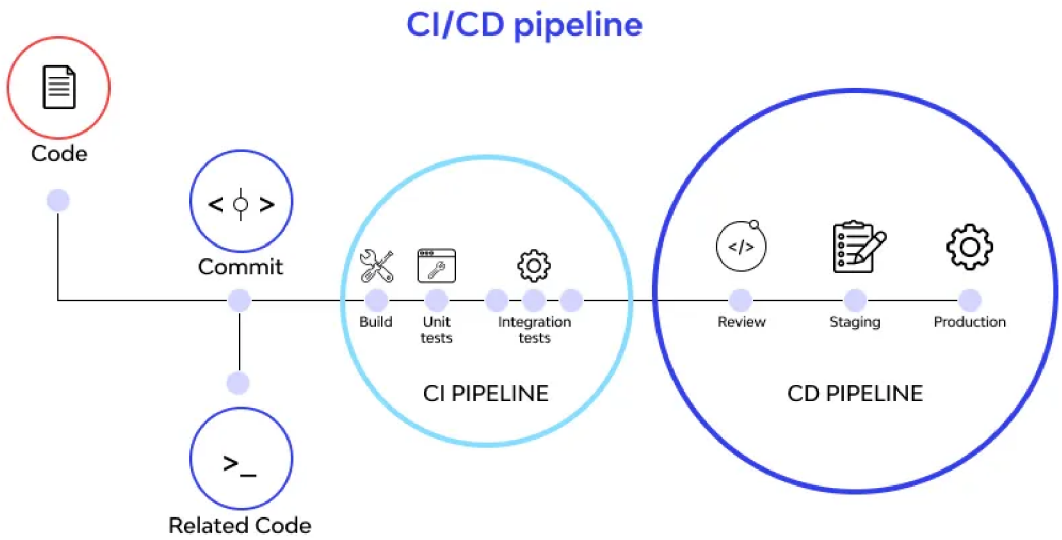
2. **Spuštění** - Pipeline může být spuštěna automaticky při událostech jako je vložení kódu do repozitáře, nebo manuálně přes webové rozhraní GitLabu.

3. **Výběr runnerů** [7] - GitLab informuje dostupný Runner o úlohách, které je potřeba vykonat. Runner, který je konfigurován pro daný projekt a má potřebné zdroje, přijme úlohu a začne ji zpracovávat.

4. **Zpracování** - Runner zpracovává úlohy v izolovaném prostředí, často pomocí Docker kontejnerů, což zajišťuje, že výsledky jsou konzistentní a nezávislé na prostředí, ve kterém běží.

5. **Zpětná Vazba** - GitLab poskytuje logy, včetně informací o úspěchu či selhání jednotlivých úloh. Výsledkem může být vytvořený artefakt, sestavení stránky, nasazení do produkce nebo pouze statistika o proběhlých testech.

Uvnitř konkrétní pipeline jsou definovány takzvané fáze (stage) a úlohy (job). Úlohy jsou zařazeny pod jednotlivé fáze. Ve výchozím stavu jsou sériově zpracovány jednotlivé fáze. Úlohy, které jsou v dané fázi se zpracují paralelně. Podle článku [9] o nastavení CI/CD, jak je uvedeno, základní fáze procesu CI/CD zahrnují sestavení (build), testování (test) a nasazení (deploy), což je ilustrováno na obrázku odkazovaném jako obrázek č. 2.8.



Obrázek 2.8: Proces automatizace pomocí CI/CD (Zdroj: [9])

[7] Runner je open-source aplikace nainstalovaná na serveru, která spouští úlohy definované v CI/CD

2.3.2 Dostupné knihovny pro automatické zpracování pomocí pipeline

Každá fáze pipeline má své specifické požadavky, a proto je výběr technologií pro jednotlivé fáze jako jsou build, test nebo deploy závisí na technologickém zpracování pokladny. Backend pokladny je napsán v jazyce TypeScript s využitím knihovny ReactJS. Frontend také využívá framework ReactJS, což je v podstatě HTML rozšířené o nadstavbové prvky. Styly jsou čerpány z online knihovny Bootstrap. Komunikace mezi serverem a pokladnou je realizována pomocí jazyka GraphQL. Serverová část je vytvořena v PHP s použitím Symfony. Zde jsou některé knihovny, které vyhovují těmto požadavkům a lze je využít v pipeline GitLabu.

Fáze BUILD:

Sestavení aplikace již používá nástroj **react-app-rewired**, který je specificky určený pro sestavení aplikací vytvořených pomocí Create React App (CRA). Pokud by používání tohoto nástroje bylo omezující, je možné využít základní nástroj pro sestavování aplikací **Webpack**. Knihovna **react-app-rewired** na pozadí tento balíček využívá. Vytváří vrstvu, která pomáhá vývojáři s nastavením projektu a usnadňuje práci při vývoji. Jak vychází z dokumentace [7] pro vytváření nového projektu, v současné době je již nedoporučuje používat nic jiného než **react-app-rewired**.

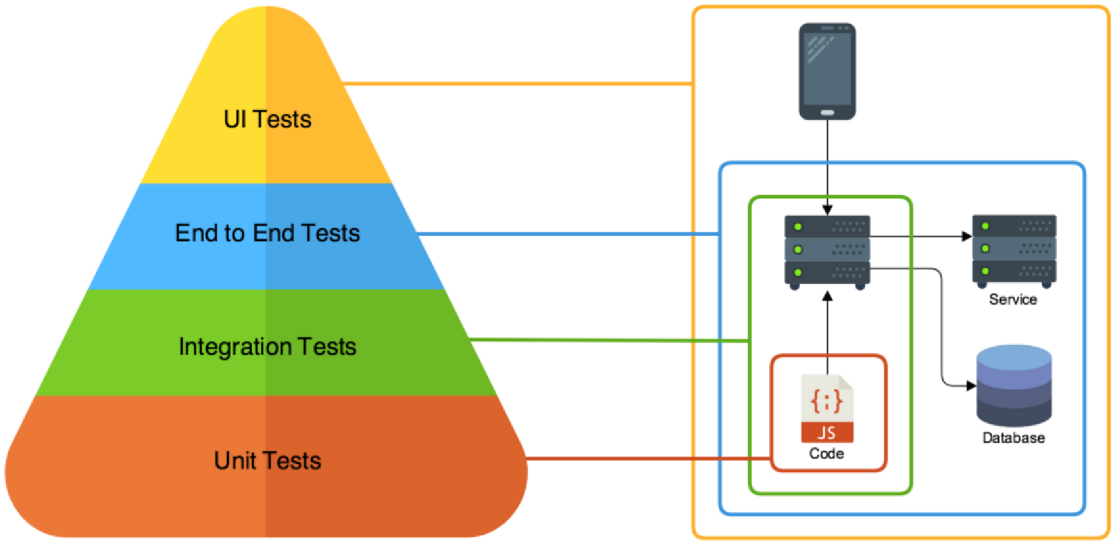
Tato knihovna samozřejmě nelze sama o sobě spustit, a proto správu projektu a jeho závislostí spravuje manažer **NPM**. Ten poběží v prostředí Node uvnitř pipeline.

Fáze TEST:

Existuje několik druhů testů, které se používají pro testování React aplikací. Tyto testy slouží k ověření správného chování části aplikace, včetně API endpoint, logiky a dalších funkcí.

- **Unit testy** testují jednotlivé části kódu (například funkce, metody, utility) izolovaně od ostatních částí aplikace. Příklad knihoven s podporou Unit testů: PHPUnit, Jest, Mocha, Enzyme, ...
- **Integrační testy** ověřují, zda různé části aplikace spolu správně komunikují. Příkladem může být ověření, že API endpoint komunikuje se serverem správně a vrací očekávané výsledky. Některé knihovny zastupující integrační testy: *React Testing Library, Cypress, TestCafe, ...*
- **End-to-end (E2E) testy** simulují chování reálných uživatelů a ověřují, zda aplikace pracuje správně jako celek. Dle článku [22] o testování aplikací bývají považovány za dostačující, protože testují jak UI, tak backend aplikace (tím i komunikační rozhraní). Možná implementace pomocí knihoven: *Jasmine, Cypress, Enzyme, ...*

- **Testy proti databázi** ověřují, že manipulace s daty v databázi probíhá správně. V React aplikacích se běžně nepoužívají. Klient aplikace většinou komunikuje skrze API se serverem, který následně pracuje s databází. Proto zde připadá v úvahu pouze PHPUnit knihovna.



Obrázek 2.9: Pyramida aplikačních testů (Zdroj: [22])

Fáze DEPLOY:

V této části automatizace by mělo proběhnout nahrání aplikace někam na server kde bude následně hostována. K takovému postupu by bylo vhodné použít některý z nástrojů jako je DeployBot, Bamboo, Jenkins nebo některé další.

Pokladna je napsána jako modul pro e-shopové řešení wpjshop. Jednotliví klienti jsou různě rozloženi po rozdílných serverech kvůli zátěži. Z tohoto důvodu není optimální nahrávat sestavenou pokladnu všem klientům a nutit tím znovu sestavení e-shopu. Jako efektivnější varianta se jeví sestavení (build) aplikace a následné vystavení tzv. artefaktu[8]. Při vložení změn do repozitáře konkrétního e-shopu se spustí vlastní pipelina, která sestaví e-shop jako takový. Během tohoto sestavení může proběhnout přiřazení posledního buildu pokladny. Tím se zajistí, že nedojde k rozdílným změnám mezi serverovým API a pokladnou.

Tento proces bude vyžadovat pouze nástroj pro stažení artefaktu z externího repozitáře a následně jeho rozbalení. Pro stažení lze použít **curl**[9] a jeho rozbalení **unzip**[10].

[8] V softwarovém vývoji se pojmem "artefakt" obvykle označuje výstup nebo produkt určitého procesu, který je vytvořen v průběhu vývoje softwaru

[9] Curl je nástroj příkazové řádky pro stahování obsahu z internetu podporující různé protokoly

[10] Unzip se používá k rozbalení jednoho nebo více souborů z archivu do cílového adresáře.

2.3.3 Vybrané technologie ke zpracování

V závěru této rešerše je zde stručný přehled vybraných nástrojů k implementaci. Tento list je doplněn o body, které výše nebyly zmíněny. Jedná se o kroky, které doplňují některé užitečné nástroje. Ty pomáhají s udržováním kódu, vylepšují struktury projektu nebo jinak pomáhají s vývojem aplikace.

- **SOURCE**: Přípravná fáze doplňuje dva balíčky. Balíček **ESLint** pro statickou analýzu kódu, případné hledání problematických vzorů. Druhá knihovna **Prettier** udržuje jednotné formátování kódu.
- **BUILD**: Nástroj **react-app-rewired**, který zjednodušuje nastavování aplikace. Tento nástroj je doporučován vývojáři ReacJS.
- **TEST**: Framework **Cypress** pro E2E testy. Přichází s vlastním studiem, nahráváním videí z testování, jednoduchým dotazováním na DOM elementy, podporou Component testování a další výhodami. Pro unit testování backendu je vybrána standardní knihovna **Jest**. Ze strany serveru bude probíhat testování GraphQL pomocí knihovny **PHPUnit** a nadstavby **GraphQLite**.
- **OPTIONAL**: Poslední fáze je volitelná. Spuštění je nutné vyvolat manuálně. Skrze knihovnu **JSDoc** se vygeneruje technická dokumentace na základě popisků z kódu.

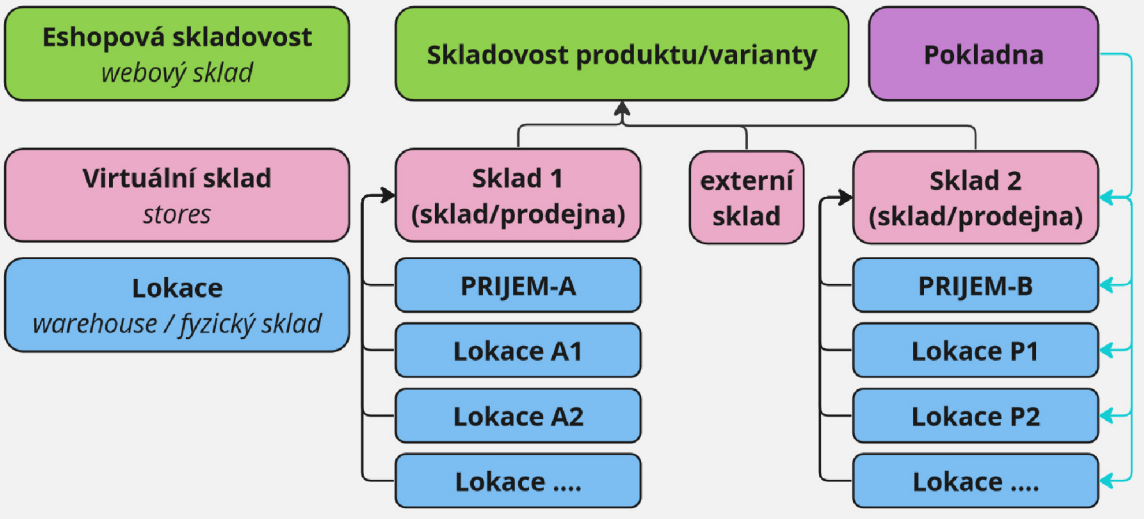
3 Implementace podpory skladového systému do pokladny

Tato kapitola popisuje postupy použité během vývoje, včetně ukázek implementovaného kódu. Stručný přehled o vrstvách skladového systému byl uveden již dříve v části věnované rešerši (viz 2.1.2). Jednotlivé vrstvy se dají považovat za moduly. Implementace těchto modulů by měla být navržena tak, aby jejich aktivace rozšiřovala stávající funkcionalitu, aniž by docházelo k její změně, přičemž jejich použití není pro nikoho povinné.

Integrace se skladovým systémem představuje složitý proces, který musí být intuitivně pochopitelný hned na první pohled a bez chyb. Proto kvůli lepší orientaci budou přikládány především vývojové diagramy namísto ne vždy přehledným úryvkům kódu.

3.1 Propojení skladového systému s pokladnou

První fází integrace bylo potřeba implementovat možnost výběru konkrétního skladu, z něhož bude pokladna čerpat zásoby. Podle předchozího průzkumu se sklady skládají z různých lokací a ty pak z konkrétních pozic. Z tohoto důvodu bylo zásadní umožnit nastavení specifických lokací v rámci vybraného skladu pro účely pokladny. Omezení jednotlivých pozic se již není vyžadována, protože rozlišení bude prováděno nanejvýše mezi jednotlivýma policemi regálu.



Obrázek 3.1: Propojení pokladny se skladovým systémem

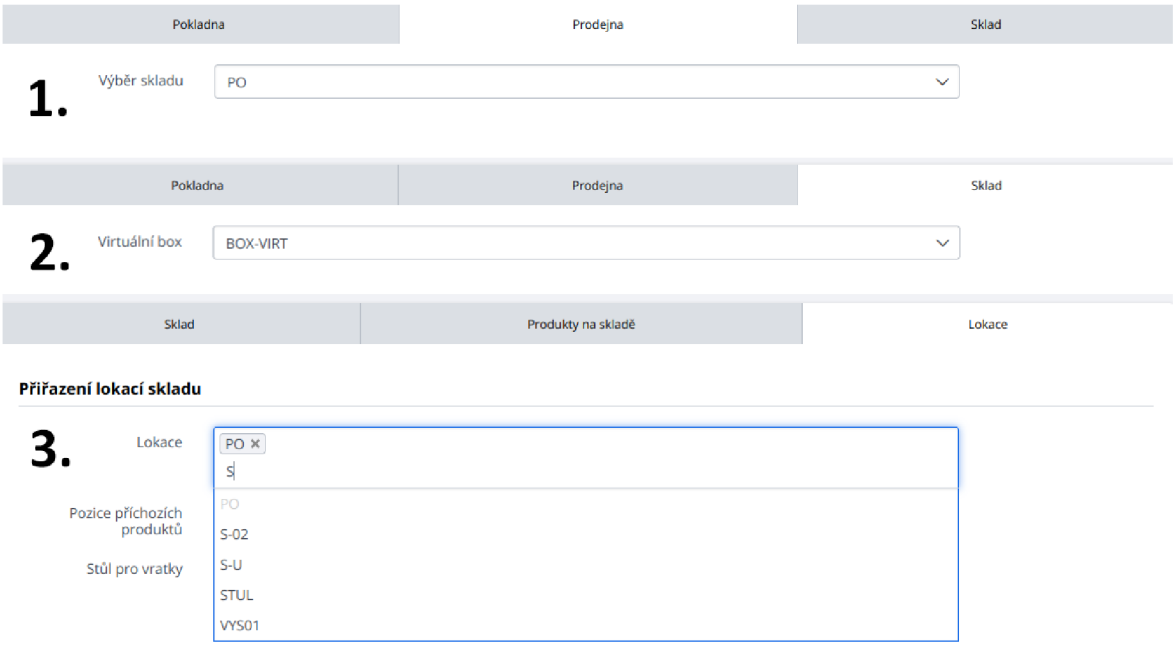
Z hlediska datové struktury reprezentuje pokladna pouze záznam v databázi, což umožňuje jednoduše přidat do tabulky odkaz na příslušný sklad. Na serveru je použit framework Symfony, který podporuje mechanismus aktualizací, známý jako migrace. Tyto aktualizace jsou aktivovány na základě specifikovaných kritérií. Jejich nastavení vychází z dokumentace [18] o migracích. Systém wpjshop má tento systém upravený na míru pro systémové potřeby. Níže je příklad, jak vypadá migrace ve wpjshopu:

```
1 public function check_columnStoresInPos() {
2     return $this->checkColumnExists('pos', 'store');
3 }
4 public function upgrade_columnStoresInPos() {
5     sqlQuery('SQL_COMMAND_HERE');
6     $this->upgradeOK();
7 }
```

Zdrojový kód 3.1: Upgrade script pro přidání pokladny

Pokladna je nyní propojena s databází. Pro ulehčení nastavení uživatelem byly do interaktivních dialogových oken, jak pro pokladnu, tak pro sklad, přidány možnosti výběru s funkcí autocomplete [1], usnadňující výběr přednastavených hodnot. Na přiloženém obrázku 3.2 jsou zobrazena tři dialogová okna, která umožňují konfiguraci dříve zmíněných nastavení. V obrázku je nastavení virtuálního boxu, který je blíže popsán v sekci 3.2 s pohyby na skladě.

[1] Autocomplete je funkce, která předvídá slovo nebo frázi, kterou uživatel chce napsat, bez nutnosti psát celé slovo nebo frázi.

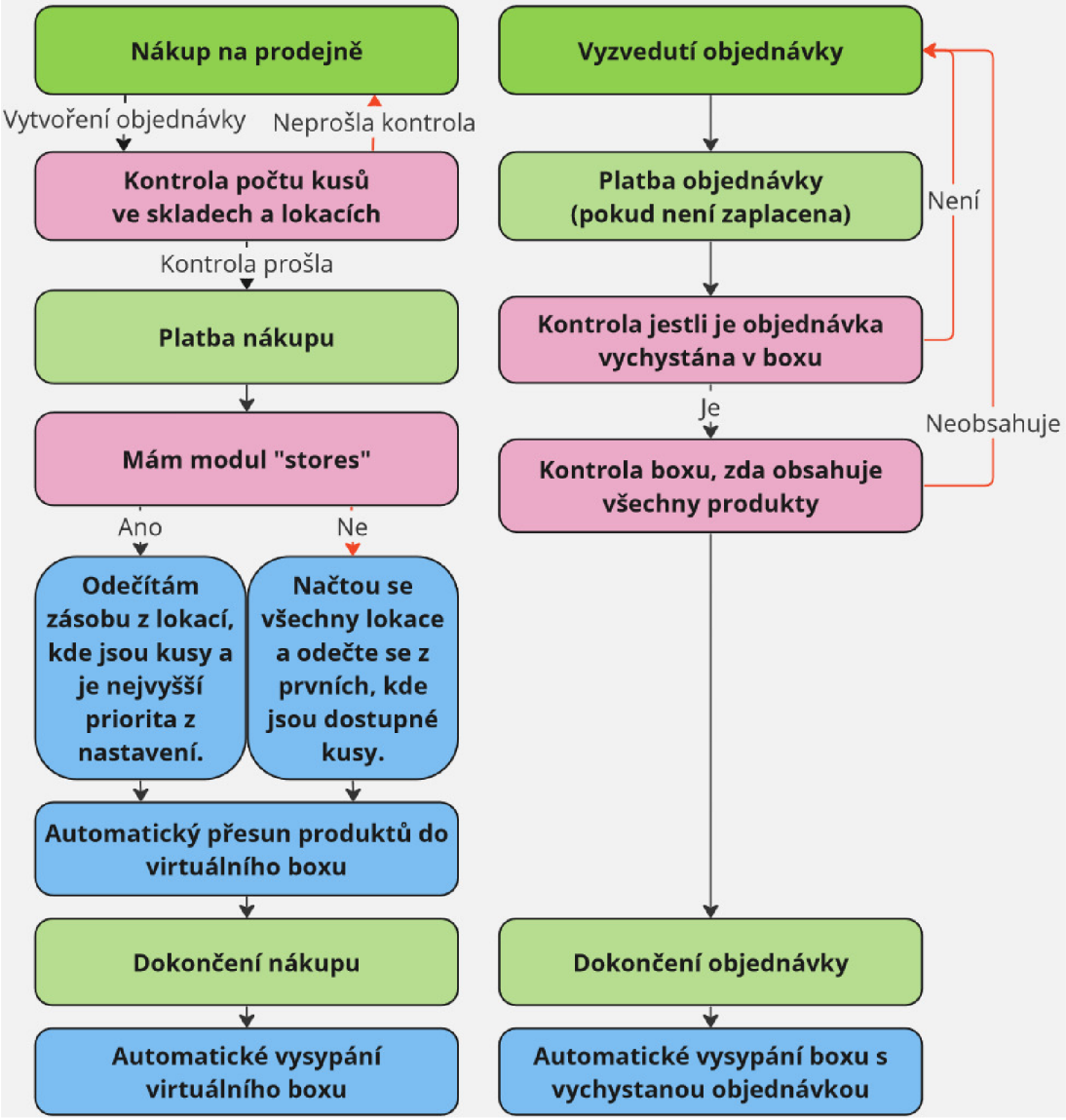


Obrázek 3.2: UI s nastavením propojení pokladny a skladu

3.2 Odečítání zásob ze skladu

Práce se skladem z pokladny je rozlišena na **"nákup na prodejně"** a **"vyzvednutí objednávky"** znázorněných v obrázky 3.3. Při nákupu v kamenné prodejně se provádí ověření dostupnosti zboží v porovnání s množstvím produktů v nákupním košíku zákazníka. Toto ověření však není potřebné v momentě, kdy zákazník vyzvedává online objednávku, jelikož požadované zboží bylo již rezervováno a odebráno ze stavu na webu.

Druhý rozdílný krok je v procesu **vychystávání objednávky** v případě, že je zapnutý modul pro fyzický sklad. Zaměstnanci kamenné prodejny musí online objednávku připravit a vybrat zboží ze skladu, přičemž umístí zboží do určeného boxu, například označeného jako *BOX-01*. Po příchodu zákazníka do obchodu zaměstnanec načte objednávku na pokladně, která zkontroluje, zda v boxu jsou všechny položky objednávky, v případě chybějících položek systém upozorní. Po úhradě zaměstnanec předá zboží zákazníkovi, což se označuje jako *„vysypání boxu"*. Pro nákupy přímo v prodejně tento krok není potřeba, jelikož zboží je obvykle již připraveno k předání. Proto byl zaveden koncept virtuálního boxu, který po zaplacení přesune zboží z prodejny nebo skladu do virtuálního boxu a následně ho „vysype", tedy předá zákazníkovi. Tento krok umožňuje efektivní sledování pohybu zboží a zajišťuje, že všechny transakce jsou zaznamenány, což pomáhá předejít například krádežím ze skladu.



Obrázek 3.3: Pohyb zásob na skladě při práci z pokladny

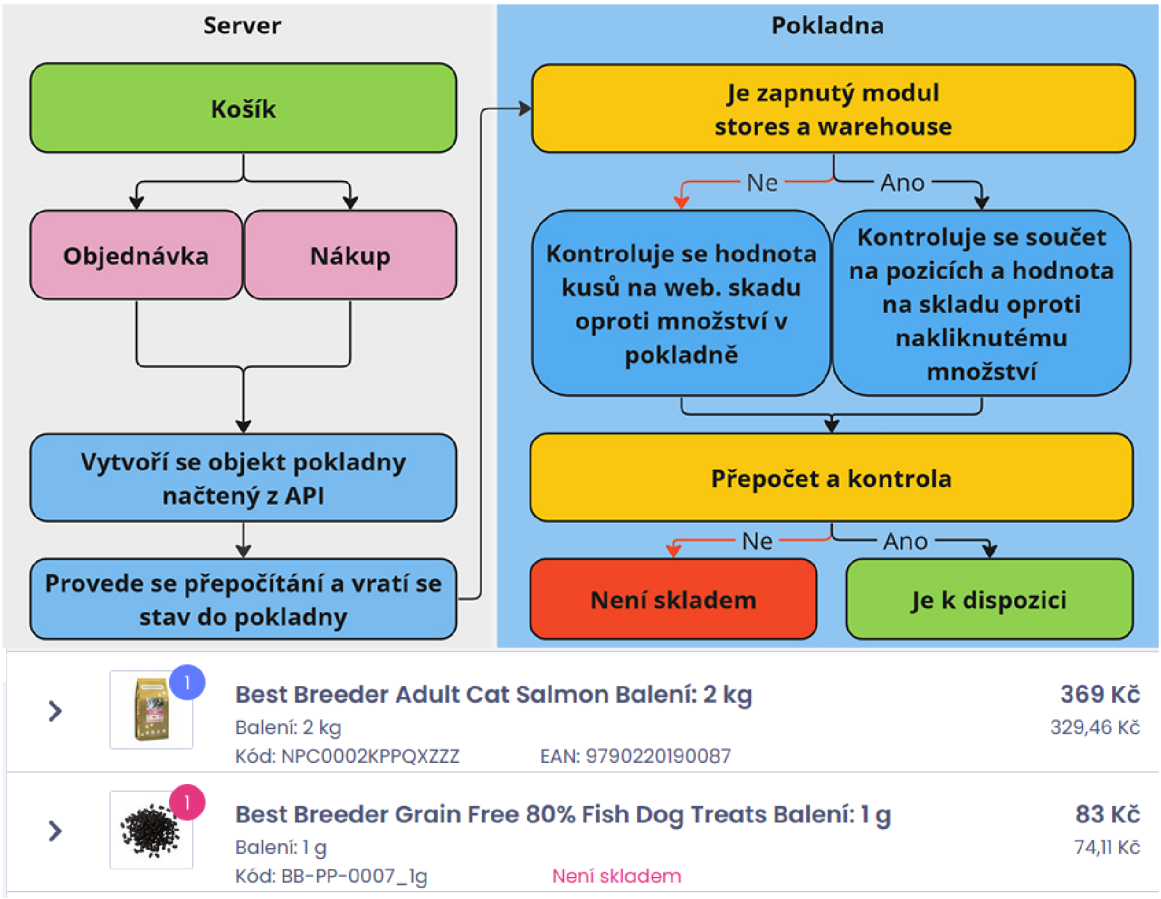
Upraven byl GraphQL endpoint pro vytvoření objednávky a platby. Konkrétně, endpoint *"posCreateOrder"* byl rozšířen o funkce pro vyhledávání a odečítání zásob z přiřazených lokací a jejich specifických pozic. Pro *"posCreatePayment"* byl implementován proces přesunu zboží do fyzického nebo virtuálního boxu. A nakonec, při dokončení nákupu či objednávky, endpoint *"posFinishOrder"* zajistí vysypání obsahu boxu.

```
1 sqlGetConnection()->transactional(function () use (...) {
2     if (findModule(\Modules::WAREHOUSE) &&
3         ($order->isPaid() || $isPaymentInvoice) && $purchase) {
4         $this->warehouseUtil->moveProductsToVirtualBox(...);
5     }
6 }
```

Zdrojový kód 3.2: Ukázka přesunu obsahu košíku do boxu

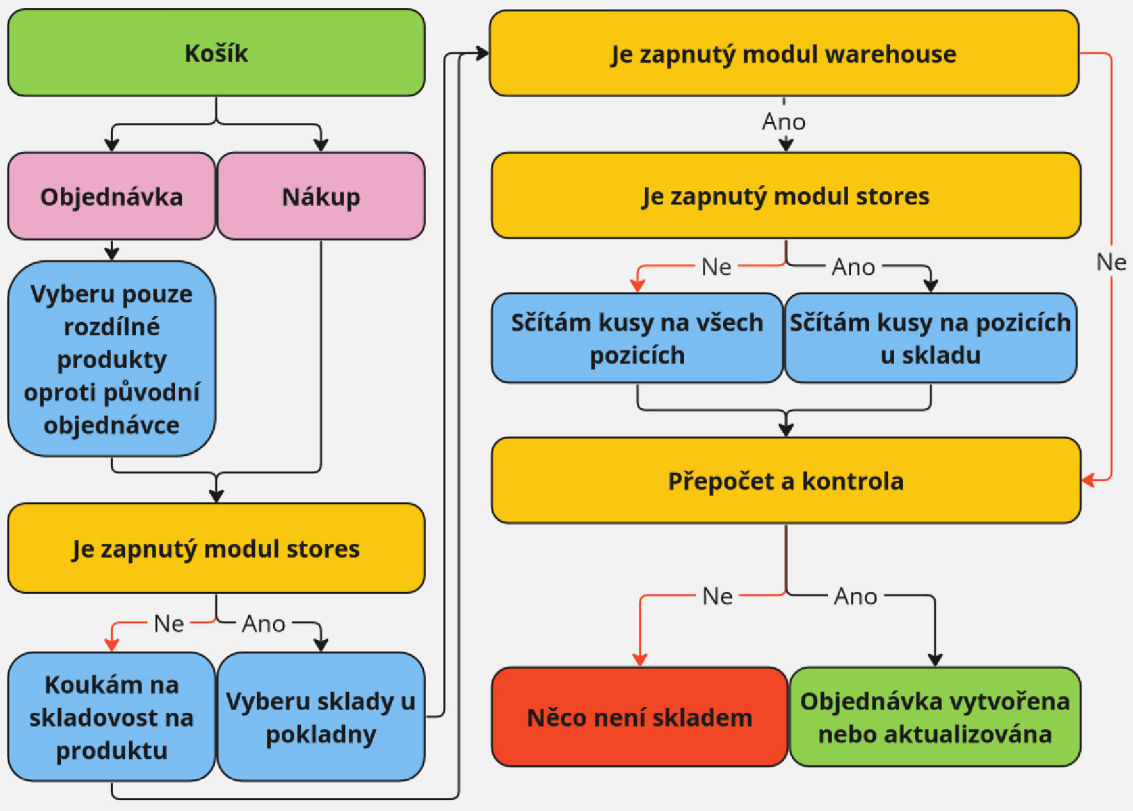
3.3 Ověření dostupnosti položek

Kontrola dostupnosti produktů se provádí ve dvou fázích. Iniciální kontrola dostupnosti pro nákup se odehrává v momentě přidání produktu do nákupního košíku. S každou následující změnou se informace o stavech pokladen zabalí a odesílají na serverové API, které pak přepočítává aktuální stav a provádí kontrolu dostupnosti jednotlivých položek v košíku. Proces ověření se může odlišovat podle toho, jaké moduly e-shopu jsou zapnuty, nebo podle toho, zda se jedná o nákup či objednávku. Jednotlivé kroky procesu jsou ilustrovány na obrázku 3.4.



Obrázek 3.4: Mechanizmus ověření dostupnosti s ukázkou v košíku

Tato fáze informuje obsluhu o dostupnosti zboží přidaného do košíku. Jedná se o pouhou informativní kontrolu, která obsluhu v její činnosti neomezuje. Pokud nedojde k žádné změně v obsahu košíku, pokladna nepožaduje znovu provést výpočet. Problém by mohl vzniknout, pokud by se objednávka z košíku vytvářela po delším čase bez nového přepočtu. Z toho důvodu byla zavedena ještě jedna kontrola, která se může v některých aspektech lišit. Standardně tato kontrola zastaví proces vytváření objednávky, pokud nezjistí dostatečný počet kusů skladem. Toto chování je možné změnit nastavovacím parametrem. Kontrola při vytvářené je zobrazena v obrázku 3.5.



Obrázek 3.5: Kontrola skladovosti při vytváření objednávky

Implementační zajímavostí z pohledu kódu je využitím událostí *(event)* pro framework Symfony. Při vytvoření objednávky je zavoláno odeslání události. V jiném adresáři, kde je umístěna kontrola skladu je zaregistrován posluchač *(listener*[2]*)*, který v případě odchycení příchozí události spustí kontrolu. Celý proces je volán pod transakcí. V případě potřeby stačí vyhodit výjimku *(exception)* a tím přerušit celý proces vytvoření objednávky. Při práci se Symfony byly čerpány informace z knihy [16] "Mastering Symfony" a konkrétně pro práci s událostmi byla využita oficiální dokumentace [19]. Zde je ukázka použití event:

```
1  /* Registrace eventy */
2  public static function getSubscribedEvents(){
3      $events[PosOrderEvent::PURCHASE_STATE_ORDER_CREATED]
4  }
5  /* Samotná metoda */
6  public function checkPurchaseStateStockIn(PosOrderEvent $event){
7      checkPurchaseStateItemsStockIn(...);
8  }
9  /* Zavolání eventy */
10 $this->sendOrderEvent(...,PosOrderEvent::PURCHASE_STATE_ORDER_CREATED);
```

Zdrojový kód 3.3: Registrace listeneru a spouštěné metody

[2]Listener je funkce v programování, která čeká na vznik určité události a reaguje na ni spuštěním předdefinovaného kódu.

4 Implementace vylepšeních zadaná klienty

Tato kapitola se zabývá modifikacemi a zdokonaleními, požadovanými klienty. Jednotlivé sekce jsou věnovány specifickým vybraným mechanismům, přičemž seznam těchto požadavků byl sestaven na základě zpracované rešerše 2.1.1. První část kapitoly se soustředí na nejzásadnější změny. Následně kapitola přechází k detailnějším úpravám, které mají rovněž velký význam.

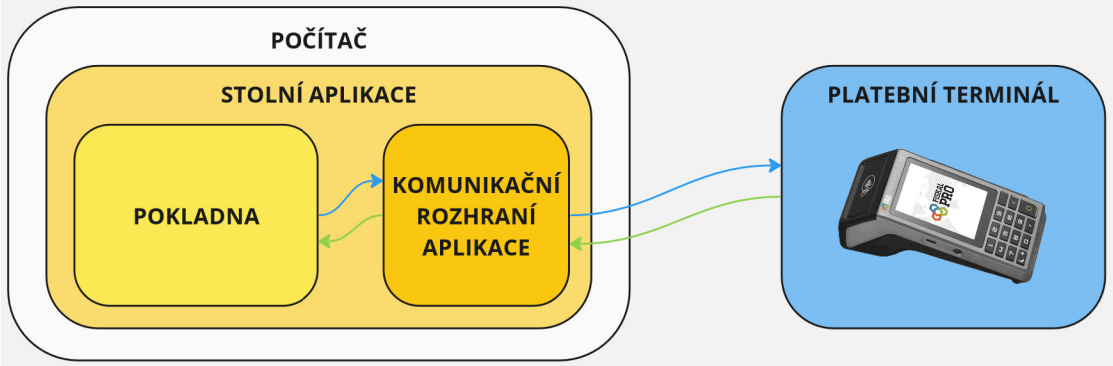
4.1 Implementace platebního terminálu

Jak je uvedeno v sekci s rešerší, tak došlo ke spolupráci s firmou FiskalPRO, která poskytla platební terminál **T6** a **sandbox** prostředí pro účely implementace a testování. Komunikace pokladny s terminálem probíhá pomocí TCP protokolu, přičemž terminál je připojen do sítě **ethernetovým kabelem**. Druhá varianta jak připojit terminál k pokladně je pomocí **USB**. Pokladna se spojuje přímo s terminálem, který následně zajišťuje komunikaci se svými servery. Detaily propojení a implementace jsou podrobně vysvětleny v následujících částech této práce.

4.1.1 Komunikace terminálu s pokladnou

Vzhledem k tomu, že pokladna je navržena jako webová aplikace, nemůže z prohlížeče přímo přistupovat k USB portům počítače. Toto omezení řeší desktopová aplikace napsaná v **Qt**, která integruje pokladnu a umožňuje tak komunikaci s externími zařízeními, jako je čtečka čárových kódů, tiskárna dokladů a v tomto případě platební terminál.

Aplikace vytváří **komunikační kanál** mezi pokladnou a samotnou aplikací. Pokladna řídí proces komunikace s platebním terminálem pomocí specifických datových zpráv, které jsou definovány v dokumentaci. Tyto zprávy jsou hexadecimálně zakódovány a odesílány z webové aplikace do stolní aplikace, odkud jsou přeposílány do platebního terminálu. Po zpracování požadavku terminál odesílá odpověď zpět do stolní aplikace, která ji následně přeposílá do pokladní aplikace. Tento proces komunikace je znázorněn na přiloženém obrázku 4.1.



Obrázek 4.1: Diagram komunikace platebního terminálu s pokladnou

4.1.2 Rozšíření stolní aplikace

Desktopová aplikace byla rozšířena o funkce **signal**[1] a **slot**[2], což umožnilo zprovoznit komunikaci přes **datový kanál** s pokladnou. Nově přidaný slot *accessPaymentTerminal* zajišťuje zpracování požadavků od pokladny, které jsou realizovány prostřednictvím tohoto kanálu. V parametrech tohoto slotu se předává datová zpráva určená pro terminál. Při jeho zavolání je automaticky spuštěna přiřazená metoda, která pomocí klienta *QTcpSocket* zasílá data terminálu. Pro zpracování příchozích dat z terminálu je registrován další slot *readTcpData*, který se spouští, když dorazí data z terminálu.

Naopak přidaný signál *terminalReceived* řeší odeslání dat zpět do pokladny. Tento signál je emitován, když přijdou data z terminálu, a spustí přiřazenou funkci v pokladně, která přijímá datovou zprávu skrze parametr. Následuje zjednodušený příklad kódové implementace těchto funkcí.

```
1  void MainWindow::accessPaymentTerminal(QString data){
2      tcpSocket = new QTcpSocket( this );
3      connect(tcpSocket, SIGNAL(readyRead()), this, SLOT(readTcpData()));
4      tcpSocket->connectToHost(ipTerminal, 6090);
5      if( tcpSocket->waitForConnected() ) {
6          tcpSocket->write( qData );
7      }
8  }
9  void MainWindow::readTcpData() {
10     this->bridge.emitTerminalResponse(tcpSocket->readAll());
11 }
12 void Bridge::accessPaymentTerminal(QString data) {
13     this->window->accessPaymentTerminal(data);
14 }
15 void Bridge::emitTerminalResponse(QString response) {
16     emit terminalReceived(response);
17 }
```

Zdrojový kód 4.1: Rozšíření Qt desktopové aplikace pro komunikaci s terminálem

[1]Signal je vysílán objektem, když se stane něco důležitého, co může ovlivnit jiné objekty.
[2]Slot je metoda definována tak, aby reagovala na signál

4.1.3 Implementace komunikace v pokladně

Datový kanál v pokladně je registrován na objekt **window**, který je dostupný napříč celým projektem. Pro odeslání dat do terminálu stačí prostřednictvím **window.channel** zavolat slot na druhé straně. Přijetí dat je o něco složitější, jelikož posluchač pro příjem emitovaných zpráv ze stolní aplikace musí být zaregistrován již při vytvoření datového kanálu.

Tento problém je řešen skrze mechanismus publikace a odběru zpráv (pub-sub), který je implementován v pokladně. **Posluchač** pro zprávy z desktopové aplikace je zaregistrován během vytváření datového kanálu. Při přijetí zprávy je proveden **publish** této zprávy pod klíčem *TERMINAL_RESPONSE_RECEIVED* do pokladní aplikace. Na místě, kde je potřeba zprávu přijmout, je zaregistrován **odběratel** (subscriber), který zprávu zachytí a pomocí callback funkce uskuteční požadované úpravy. Zde je zjednodušená ukázka.

```
1  // Odchytávání zpráv z desktopové aplikace
2  window.channel.terminalReceived.connect(function (data) {
3      Signals.publish(TERMINAL_RESPONSE_RECEIVED, { response: data });
4  });
5  // řPijetí zprávy po odeslání
6  subscribeEvent(TERMINAL_RESPONSE_RECEIVED, (...) => {
7      ...
8  });
9  // Odeslání datové zprávy do terminálu
10 if (window.channel && window.channel.accessPaymentTerminal) {
11   window.channel.accessPaymentTerminal("01010005465252544351");
12 }
```

Zdrojový kód 4.2: Komunikace s platebním terminálem v pokladně

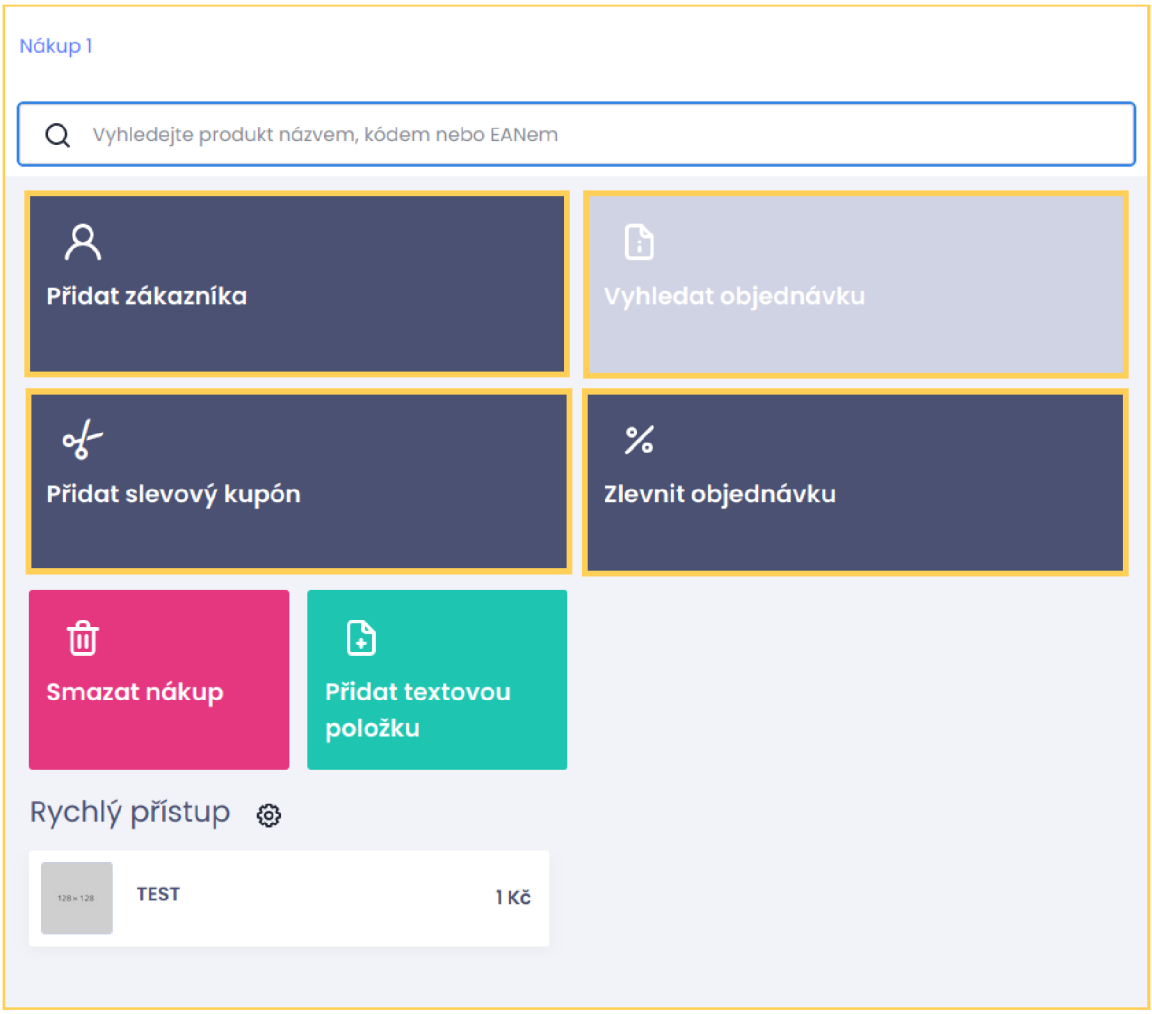
Datový řetězec *"01010005465252544351"* slouží jako zjednodušená ukázka zprávy pro získání času nastaveného na platebním terminálu. Z důvodů neveřejné dokumentace je tato ukázka pouze ilustrativní a slouží k vysvětlení základního principu komunikace, nikoli k detailnímu popisu skutečných datových zpráv.

4.2 Rozšířená podpora pro čtečku

V původní verzi pokladního systému byl v kořenovém souboru *app.tsx* zaregistrován jen jeden listener, očekávající přijetí hodnoty *CODE_RECEIVED*, což signalizovalo příjem dat ze skeneru. Asynchronní spouštění kódu vedla k potížím s určením, jaký kód byl právě skenerem načten.

Změna spočívá ve využití rozlišení, který modal[3] je v momentě zachycení události aktivní, díky čemuž systém rozpozná, jestli se načítá čárový kód objednávky nebo například slevový kód. Na obrázku 4.2 jsou změny v načítání kódů ilustrovány žlutě zvýrazněnými oblastmi, přičemž žlutý rámec okolo obrázku označuje výchozí režim načítání produktů.

[3]Modal v ReactJS je komponenta, která se zobrazí jako vrstva nad hlavním obsahem stránky, obvykle pro zobrazení důležitých informací nebo pro interakci s uživatelem, aniž by bylo nutné opustit aktuální kontext stránky.



Obrázek 4.2: Druh načítaných dat ze čtečky z pohledu UI

4.2.1 Změna implementovaná v pokladně

První úprava spočívá v posunutí listeneru o úroveň výše z kořenové části aplikace, aby měl přímý přístup k vnořeným stránkám, což umožňuje odstranění zbytečného kódu potřebného pro synchronizaci dat mezi komponentami.

Druhý krok zahrnuje aktualizaci registrovaného posluchače při každé změně okna, což bylo realizováno odstraněním starého a registrací nového posluchače. Stav každého okna je uchováván v pokladně pomocí *useState*[4], a pro synchronizaci se všemi stavy se využívá hook[5]. Tato metoda zajišťuje, že při změně stavu jakéhokoli okna dojde k vyvolání příslušné akce. Ukázka kódu ilustruje, jak je listener registrován a jak rozlišuje mezi různými typy kódů. K implementaci byla využita kapitola "Enhancing Components with Hooks" z knihy "Learning React" [3].

[4]Funkce useState je hook v ReactJS, který umožňuje přidat stavovou proměnnou do funkčních komponent.

[5]React Hook je speciální funkce, která umožňuje použití stavu a dalších ReactJS funkcí v komponentách bez tříd.

```
useEffect(() => {
  if (userModal) {
    props.handleEventSubscriberCallback(...);
  }
}, [userModal, ...]);

const handleEventSubscriberCallback = (...) => {
  Signals.clearAllSubscriptions();

  subscribeEvent(CODE_RECEIVED, (message: any, data: any) => {
      ...
      switch (action) {
        case ReaderAction.user: ...
          return;
        case ReaderAction.order: ...
          return;
    }
  });
};
```

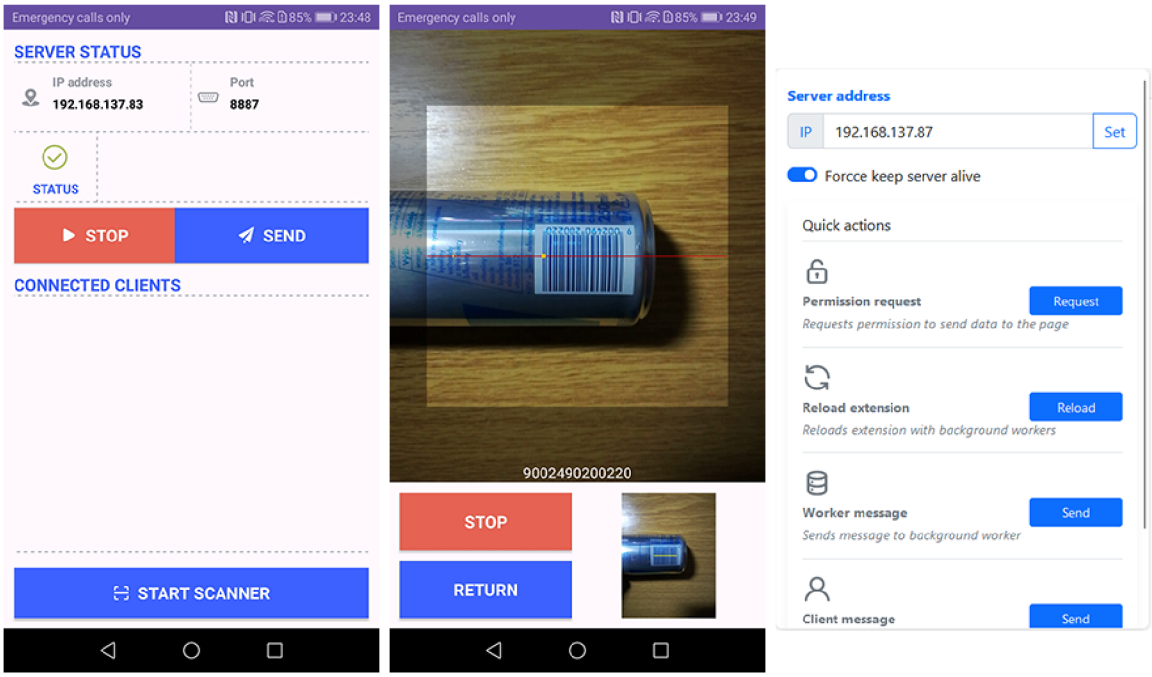
Zdrojový kód 4.3: Ukázka implementace hook a registrace pro listener

4.2.2 Simulátor čtečky pomocí telefonu

Během vývoje aplikace se občas vyskytl problém s nedostupností čtečky, což komplikovalo testování čtení čárových kódů. Bylo možné ručně přepisovat kódy do konzole, ale to nebylo efektivní z hlediska času.

V rámci diplomové práce byl vytvořen nástroj, který tento problém řeší. Vytvořena byla aplikaci pro Android v programovacím jazyce Java, která skenuje čárové kódy a posílá je do prohlížeče. Pro prohlížeč byl vytvořen multiplatformní doplněk *(polyfill-extension)*, který používá websockety k propojení s telefonem a přenáší načtená data do aktuální relace prohlížeče. V prohlížeči je JavaScript doplňku spouštěn v izolovaném prostředí, ze kterého není přímý přístup k načtené stránce. Proto spojení s telefonem realizuje script běžící na pozadí tzv. BackgroundWorker. Tento worker se v rámci prohlížeče spojí s ContentWorkerem *(script injektovaný do otevřené stránky v prohlížeči)* a ten následně vyšle pub-sub zprávu s načtenými daty. Tento mechanismus je v pokladně využit pro zachytávání událostí, a proto dokáže zachytit i simulované zprávy z rozšíření. Podpora ze strany prohlížeče vychází z oficiální dokumentace [5] pro prohlížeč Mozilla.

Protože vývoj této aplikace nebyl přímým cílem diplomové práce, ale spíše doplňkovým nástrojem, nebude zde dále podrobně rozebírán. Níže je přiložen obrázek aplikace a doplňku.

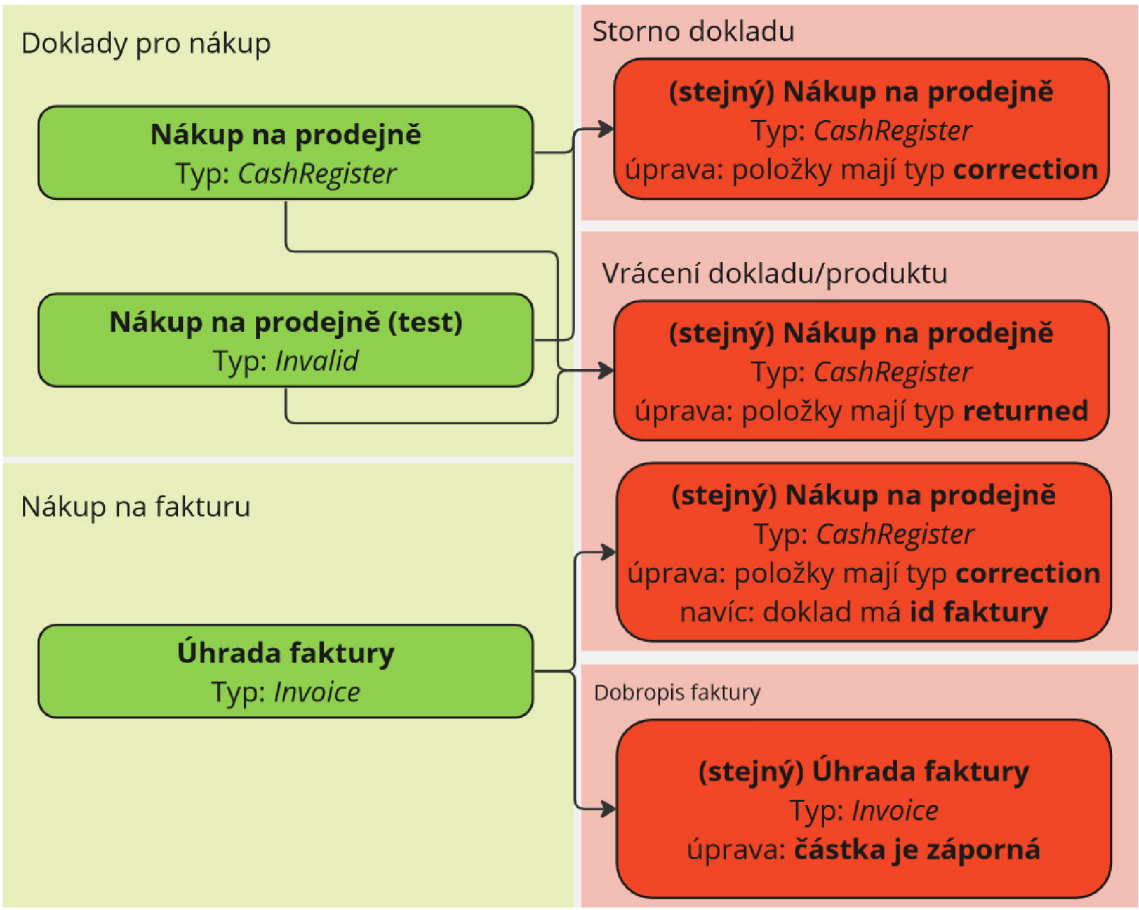


Obrázek 4.3: Android aplikace a rozšíření do prohlížeče

4.3 Implementace SK EET

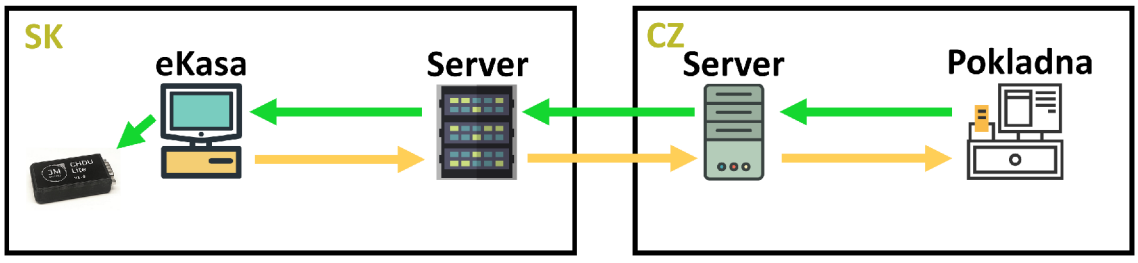
Implementace připojení k slovenskému systému EET byla provedena ve spolupráci s firmou NineDigit. Tato společnost je zodpovědná za vývoj systému Portos eKasa, který slouží k zajištění komunikace s eKasa systémem finanční správy, a je oficiálně certifikován Finanční správou Slovenské republiky.

Na serveru zpracovávajícím žádosti z pokladního systému bylo zapotřebí implementovat nový modul SKEET. Tento modul využívá knihovnu od NineDigit pro komunikaci se slovenským EET systémem. S pomocí této knihovny jsou generovány doklady různých typů, jejichž příklady jsou ilustrovány na přiloženém obrázku 4.4. Na levé straně se nachází originální doklad, zatímco na pravé straně je zobrazeno stornování nebo vrácení daného dokladu.



Obrázek 4.4: Doklady slovenského systému eKasa pro EET

Implementační podrobnosti jsou odvozeny z dokumentace [14] poskytnuté firmou NineDigit. Při provedení transakce na pokladně se obvykle používá typ dokladu CashRegister. Systém Portos eKasa je instalován na počítači klienta, ke kterému je připojen specifický typ hardwarového úložiště - tzv. "CHDU". Tento disk se liší od běžných USB flash disků tím, že umožňuje pouze zápis dat. Data přijatá z pokladny jsou na tomto disku následně ukládána. Komunikační schéma mezi pokladnou a tímto systémem je znázorněn na přiloženém obrázku 4.5.



Obrázek 4.5: Komunikace se slovenským systémem Portos eKasa

4.4 Zadávání sériových čísel u produktu

Sériová čísla se aplikují jen na specifickou skupinu produktů, převážně mechanických dílů, které se mohou nějakým způsobem odlišovat. Pokladní systém nyní obsahuje speciální výběrové pole pro sériová čísla, jež se zobrazí jen u produktů s povolenou podporou těchto čísel. Do pole lze ručně vkládat sériová čísla znak po znaku, přičemž systém nabízí funkci autodoplňování, která usnadňuje předvýběr zapisované hodnoty podle postupně přidaných znaků. Alternativně je možné sériové číslo naskenovat pomocí čtečky. Při kliknutí do pole se aktivuje listener pokladny a tím místo vložení nového produktu dojde k vložení kódu do pole. Po naskenování nebo opuštění pole se funkce listeneru automaticky resetuje na původní režim, tedy načítání produktů. Při zpracování objednávky je následně sériové číslo přiřazeno k dané položce, což umožňuje obsluze připravit přesně specifikovaný díl. V následujícím obrázku je ukázka produktu s již zadaným sériovým číslem.

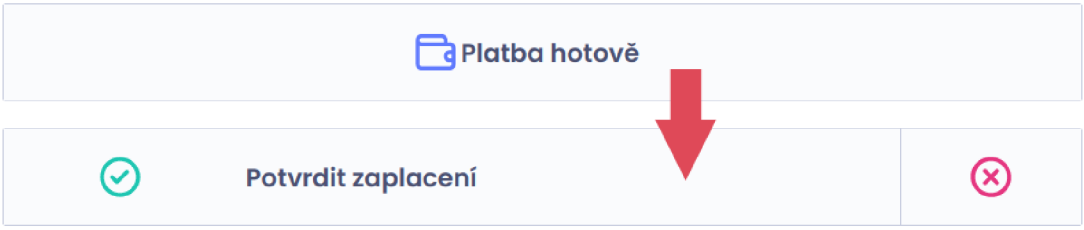
Obrázek 4.6: Produkt v košíku s načteným sériovým číslem

4.5 Vylepšení mechanizmu zaplacení

Jak bylo uvedeno v rešeršní části, objednávka byla v pokladním systému vytvářena až v okamžiku stisku tlačítka pro zaplacení. Tento postup komplikoval obsluze práci při přepisování čísla objednávky do platebního terminálu nebo při změně způsobu platby po již provedené akci. Aby se tento problém vyřešil, došlo k oddělení procesu vytvoření objednávky od jejího zaplacení.

V pokladním systému byla vytvořena nová react komponenta - tlačítko. Při jeho prvním stisku se odešle požadavek na vytvoření objednávky. Po změně stavu se tlačítko rozdělí na dvě části - potvrzení platby a možnost ukončit proces a vybrat jiný způsob platby. Obsluha si vybere způsob platby, načež pokladní systém aktualizu-

je a vytvoří objednávku s vygenerovaným číslem. V případě platby kartou obsluha přepíše číslo objednávky na platební terminál, zaplatí a v systému potvrdí přijetí platby. Při platbě hotovostí proces dokončí po přijetí peněz. Tato nová komponenta je prezentována na přiloženém obrázku.



Obrázek 4.7: ReactJS komponenta pro zaplacení objednávky

4.6 Příjem zboží přes pokladnu

Na první pohled se může zdát, že přidání funkcionality pro přijímání zboží představuje významné rozšíření systému. Nicméně, po prozkoumání vyplývá, že je možné pro tuto úlohu využít stávající mechanismus vytváření objednávek. Konkrétně, objednávky pro přijímání zboží budou charakterizovány záporným množstvím položek a odpovídající zápornou částkou.

V pokladně bylo zapotřebí upravit uživatelské rozhraní tak, aby umožňovalo zadávat záporné hodnoty pro množství kusů a finanční částky. Na serverové straně bylo nezbytné upravit API a mechanismus kontroly dostupnosti položek, aby reflektovalo možnost zadání záporných objednávek. Tato nová funkcionalita byla implementována pod oprávnění zapínatelné pro administrátory.

Z hlediska skladu se přijaté zboží eviduje pod specifickou pozicí, například *"PŘÍJEM"*. Pomocí záporné objednávky jsou položky přičteny do webového skladu. Zboží je následně zaskladněno zpět do příslušných pozic ve skladu.

4.7 Nastavení oprávnění pro administrátory

Administrátoři využívající pokladní systém musí mít jasně stanovená oprávnění, aby bylo zřejmé, ke kterým funkcím mají přístup a které jsou pro ně omezené. Konfigurace těchto oprávnění je začleněna do Smarty šablon v sekci nastavení pro administrátory. Zde je seznam všech doposud implementovaných oprávnění.

- **Povolit pokladnu**: Povolí používání ReactJS aplikaci pokladny. Po zapnutí se pokladna zobrazí v administraci.
- **Správa pokladen**: Povolí přidávat, upravovat a mazat jednotlivé instance pokladem v administraci.
- **Nastavení pokladny**: Povolí měnit nastavení uvnitř pokladní aplikace.

- **Přidávání slev**: Povolí nastavovat slevy k objednávce v pokladně.
- **Mínusový nákup**: Povolí v pokladně provést nákup se zápornými kusy a zápornou cenou (kvůli rychlému vrácení).
- **Nákup do mínusu**: Povolí v pokladně nakoupit produkty, které nejsou skladem (u produktu bude svítit "není skladem", ale povolí provedení nákupu).

Po dokončení nastavení jsou oprávnění prostřednictvím GraphQL API přenesena do pokladny. Při prvním spuštění aplikace pokladna využívá endpoint "posAdditionalData" k získání základních informací nutných pro její správnou funkci od serveru. V rámci tohoto požadavku jsou oprávnění přenesena do aplikace, kde jsou uložena v jejím nastavení. Aplikace poté podle těchto dat povoluje nebo omezuje přístup k jednotlivým funkcím. Následuje příklad, jak se do API přidávají oprávnění.

```
$permissions[] = new PosPermission(
    'POS_APP_SETTINGS',
    findRight('POS_APP_SETTINGS')
);
```

Zdrojový kód 4.4: Implementace nového oprávnění do API

4.8 Ostatní požadovaná vylepšení

Tato sekce obsahuje všechny ostatní požadovaná vylepšení. Jedná se o drobné úpravy nebo jednodušší změny, proto jsou sjednoceny do jedné sekce.

Filtrování objednávek určených pro pokladnu

Do pokladního systému bylo implementováno nové nastavení umožňující filtrování objednávek určených pro pokladnu. Když je toto nastavení aktivováno, vyhledávání objednávek vytvořených online se omezí pouze na ty, které jsou určeny k osobnímu odběru přímo u této pokladny. Filtrovací kritéria byla začleněna přímo do SQL dotazu, který tyto objednávky vyhledává. Existují dvě metody, jak zajistit propojení mezi pokladnou a objednávkami: První způsob spočívá ve výběru objednávek, jejichž způsob doručení (kombinace dopravy a platby) odpovídá nastavení dané pokladny. Druhý způsob využívá modul Prodejny, který pro e-shop přináší rozšířenou funkcionalitu a z pohledu pokladny zavádí vazební tabulku propojující konkrétní pokladnu s příslušnou kamennou prodejnou.

Vykreslování čárových kódů

Pro generování čárových kódů v pokladním systému byla využita knihovna **react-jsbarcode**. Kódy byly nakonfigurovány aby se vykreslovaly ve formátu **CODE128**. Implementace vykreslování čárových kódů proběhla na dvou různých místech. První případ se týká zobrazení čárového kódu pro již načtenou objednávku, kde čárový

kód reprezentuje číslo dané objednávky. V druhém případě je generován čárový kód pro box, ve kterém byla objednávka připravena ve skladu a je označena k vydání.

Obrázek 4.8: Čárové kódy vykreslované v pokladně

Textové položky v rychlém přístupu produktů

Funkce pro rychlý přístup k produktům prošla rozsáhlou úpravou, aby mohla ukládat také textové položky. Původní seznam položek pro rychlý přístup, který byl součástí nastavení, byl odstraněn. Namísto něj byl zaveden nový modální seznam, umožňující odstraňování těchto položek. Přidávání produktů zůstává beze změny a to vyhledáním produktu a následným kliknutím na ”přidat do rychlého přístupu” na detailu produktu. Pro vložení textové položky do košíku je k dispozici formulář. Tento formulář byl rozšířen o tlačítko, které namísto vložení do košíku uloží položku do rychlého přístupu.

5 Implementace dynamického přidávání funkčních částí do pokladny

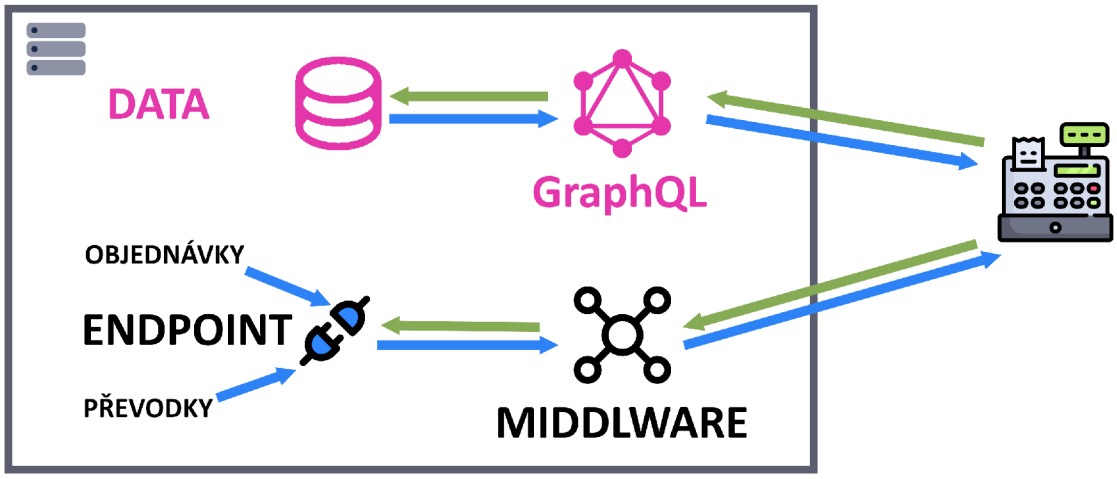
Tato kapitola se zabývá mechanismem, který umožňuje dynamicky přidávat funkční části do systému pokladny, aniž by bylo nutné zásadně zasahovat do jejího zdrojového kódu. Úvodní část kapitoly poskytuje obecný popis tohoto mechanismu, který je dále názorně vysvětlen na přiloženém obrázku. Následuje část zaměřená na implementaci na serverové straně, kde jsou podrobně rozebrány technické aspekty. Poté kapitola pokračuje popisem integrace této funkcionality do samotného pokladního systému. Závěr kapitoly je věnován procesu přidávání nových funkcí a přehledu již implementovaných funkcionalit.

5.1 Funkce mechanizmu

Dynamické načítání obsahu se skládá ze dvou zásadních součástí. Na serverové straně je to takzvaný middleware, který zpracovává požadavky od pokladny. Na základě přijatých dat middleware nastavuje parametry odeslané na endpoint s obsahem stránky, nebo v případě chybějícího ověření může požadavek odmítnout. Po úspěšné verifikaci je požadavek odeslán na endpoint, který zajišťuje vykreslení, v Symfony označovaném jako Renderer. Tento Renderer na své webové adrese zobrazuje určenou šablonu. Příkladem může být adresa pro seznam objednávek, která bude vypadat takto: *doména/admin/launch.php?s=list.php&type=posOrders*. Z parametrů endpointu vyplývá, že má sestavit seznam, který bude obsahovat objednávky pro pokladnu. Nastavení pokladny provádí middleware, který přiřadí entitu poklady do session[1]. List načte entitu a upraví list na základě přiřazeného nastavení. Po vykonání požadavku je obsah z endpointu načten do pokladny. Detaily implementace jsou popsány v dalších kapitolách.

Pro lepší pochopení využití serverového API pokladnou je přiložen obrázek, který ukazuje jak běžnou komunikaci, tak dynamické načítání obsahu. GraphQL endpoint pro standardní komunikaci je zvýrazněn růžově, zatímco REST middleware pro dynamický obsah je označen černou barvou. Zelené šipky ukazují směr požadavků a modré odpovědi na ně.

[1]Session je způsob, jak uchovávat informace napříč různými stránkami tím, že data jsou uložena na serveru.



Obrázek 5.1: Rozdělení serverového API určeného pro pokladnu

5.2 Implementace na straně serveru

Objekty, které server vykresluje, jako jsou formuláře, seznamy a podobně, jsou ve skutečnosti třídami, jejichž šablony jsou renderovány pomocí rendereru Symfony. Pro jednoduché přidání těchto elementů do pokladny byl vytvořen nový interface[2] "PosTabInterface". Použitím tohoto interface je element označen pro načtení do pokladny nezávisle na jeho umístění v adresáři.

```
1 interface PosTabInterface {
2     public function isAllowed();
3     public function getTitle();
4     public function getId();
5     public function getUrl();
6 }
```

Zdrojový kód 5.1: Interface označující strukturu pro pokladnu

Při spuštění pokladního systému se z serveru prostřednictvím GraphQL načítají doplňující informace, které nejsou uložené v localStorage prohlížeče. Součástí tohoto endpointu je také získávání seznamu dostupných záložek (tabů). K automatickému nalezení všech potřebných interfaců se využívají specifické funkce Symfony frameworku, které jsou detailně popsány v následujících krocích.

1. Služba je automaticky registrována do kontejneru a označena unikátním tagem, bez ohledu na její umístění v adresářové struktuře projektu.

```
1     $container->registerForAutoconfiguration(
2         PosTabInterface::class)->addTag('kupshop.pos.tab')
3         ->setPublic(false)->setAutowired(true);
```

Zdrojový kód 5.2: Automatická registrace služby do Symfony frameworku

[2]Interface je struktura, která umožňuje definovat metody bez jejich implementace, které musí být implementovány ve třídách, které dané interface používají.

2. Automatické vyhledávání všech zaregistrovaných interfaců je zajištěno pomocí tzv. "service locator", což vychází z článku [10] o generování lokátorů na základě jejich označení. Tento lokátor je nutné správně nakonfigurovat, aby věděl, jaký tag má vyhledávat. Konfigurace je umístěna v souboru *services.yml*, který Symfony používá pro nastavení všech služeb.

```
    GraphQLBundle\ApiPos\Util\PosUtil:
        arguments: [
            !tagged_locator { tag: 'kupshop.pos.tab' }
        ]
```

Zdrojový kód 5.3: Nastavení tagu pro service locator.

3. V GraphQL kontroleru jsou prostřednictvím locatoru načteny všechny struktury, které nejsou omezené právy nebo jinými konfiguračními parametry. V datech požadavku bude seznam stránek zpřístupněných pomocí middlewaru.

```
    $iframes = [];
    foreach ($this->locator->getServices() as $class) {
        if ($this->serviceLocator->has(class)) {
            $service = $this->serviceLocator->get(class);
            if ($service->isAllowed()) {
                $iframes[] = new PosIframe(...);
            }
        }
    }
```

Zdrojový kód 5.4: Načtení dat dynamického obsahu do GraphQL API struktury

V tuto chvíli pokladní systém disponuje daty o jednotlivých stránkách. Detaily o chování a struktuře implementace v pokladně jsou popsány v následující kapitole 5.3. Když pokladna přistupuje k dynamicky načtenému obsahu, nepřistupuje přímo na endpoint například seznamu, ale nejprve využívá dříve zmíněný middleware. Tento middleware je REST endpoint, který přijímá dva parametry od pokladny: ID pokladny a identifikátor stránky, na kterou se chce dostat. Middleware vytvoří objekt pokladny a ověří, zda byl na základě přijatých dat správně vytvořen. Pokud byl, nastaví tuto instanci do session a přesměruje na specifickou záložku, v opačném případě vrátí chybu. Krok s využitím middleware byl vytvořen kvůli sjednocení ověření, vytvoření a nastavení instance pokladny do session.

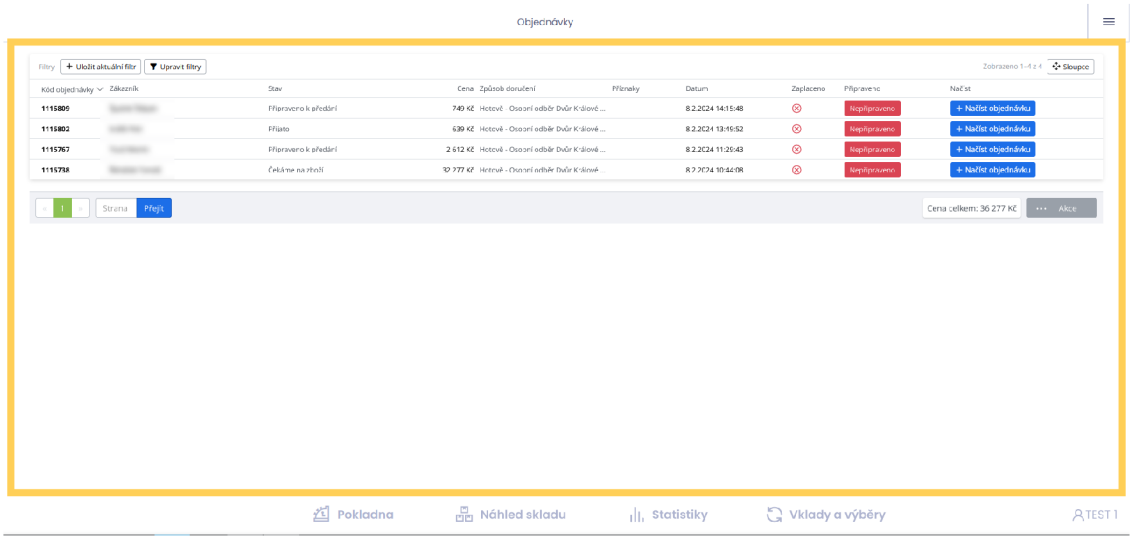
```
    /** @Route("URL", requirements={"url"=".+"}) */
    public function posTab(Request $request, PARAMS ...) {
        $this->posEntity->createFromDB($idPos);
        if (empty($this->posEntity->getId())) {
            return new Response('#403 Forbidden', 403);
        }
        $this->requestStack->getSession()->set(
            'pos_entity', $this->posEntity
        );
        return new RedirectResponse("URL");
    }
```

Zdrojový kód 5.5: Ukázka implementace pro middleware v Symfony

5.3 Implementace v pokladně

S implementací podpory v pokladně byla přidána knihovna GraphQL-Codegen, která z dotazu vygeneruje objekty do TypeScriptu. Díky těmto objektům mnohem jednodušší zpracovávat odpovědi z GraphQL API. Vylepšit takto strukturu kódu vzešlo z knihy [6] "Learning GraphQL: Declarative Data Fetching for Modern Web Apps". V knížce je část věnována imlpemntaci service komponentám pro klienta Apollo. Tento klient v pokladně použit nebyl, ale jedná se o alternativní knihovnu, která přináší spoustu typových kontrol a vylepšení.

V pokladním systému byla zavedena nová stránka (view), obsahující pouze navigační tlačítka a iframe roztažený na zbytek dostupného prostoru. URL adresa middlewaru, doplněná o GET parametry jako ID pokladny a identifikátor cílové stránky, je nastavena jako zdroj pro iframe. Umístění iframu je ilustrováno v obrázku 5.2 žlutým okrajem.



Obrázek 5.2: Iframe se seznamem objednávek určených na pokladnu

5.4 Komunikace mezi načteným oknem a pokladnou

Komunikace mezi okny je založena na vztahu rodič-potomek, přičemž pokladna plní roli rodiče a obsah iframe roli potomka. Implementace byla inspirována článkem [13], který vysvětluje zásady tohoto typu komunikace. Následující ukázka demonstruje, jak je vybraná objednávka načtena z seznamu do pokladního systému.

Do seznamu s objednávkami je přidáno tlačítko pro načtení, to je zobrazeno na výše přiloženém obrázku 5.2. Stisknutím tlačítka je odeslána zpráva do rodiče (pokladny). Zpráva obsahuje typ události co se má v pokladně provést a data, kde je například id objednávky, která se bude načítat.

```
1 $message = {
2     event_id: 'load_order_request',
```

```
    data: {
        id: '{$values['id']}'
    }
};
window.parent.postMessage({$message}, '*');
```

Zdrojový kód 5.6: Odeslání zprávy do pokladny v iframe

Při otevření stránky s iframe je v React aplikaci na objektu window zaregistrován posluchač (listener), který čeká na příchozí zprávy. Po přijetí zprávy je zavolán callback posluchače a následně je provedena implementovaná akce, jak je ukázáno v příkladu.

```
useEffect(() => {
  window.addEventListener('message', handleIframeCallback);
  ...
}, [props.iframeUrl]);

const handleIframeCallback = useCallback((event) => {
  if (event?.data?.event_id === 'load_order_request') {
    loadOrderObjectFromApi(...);
  }
}, []);
```

Zdrojový kód 5.7: Zpracování požadavku přijatého z iframe

5.5 Proces přidávání a dosud přidané funkce

Závěrem této kapitoly je postup jak přidat nové funkční celky do pokladny bez žádného, nebo minimálního zásahu do kódu pokladny.

1. Pokud je třeba upravovat již existující funkce, je nezbytné buď vytvořit nový objekt nebo zdědit stávající objekt a předefinovat jednotlivé funkce.
2. Přiřadit interface a implementovat vyžadované metody.
3. Pokud je nutná komunikace mezi pokladnou a oknem, je vhodné v pokladně rozšířit callback o vlastní přidanou funkci.

Tento přístup umožňuje do pokladny přidávat téměř jakékoliv funkce, avšak je na uvážení vývojáře, do jaké míry je vhodné tento mechanizmus využívat. Primárně je tento mechanizmus navržen pro zobrazování hodnot z administrace pro obsluhu s příslušnými oprávněními k obsahu. Pokud by přidávaná funkcionalita vyžadovala komplexnější úpravy, je vždy lepší tyto úpravy implementovat přímo do kódu pokladny.

Do pokladny byly doposud přidány dva seznamy. První je **seznam objednávek**, jak bylo ukázáno v předchozí implementaci, určených pro konkrétní pobočku. Druhý seznam, **seznam převodek**, se týká příjmu zboží na prodejně a zahrnuje dovoz zboží ze skladu do kamenného obchodu. Díky tomu je obsluha pokladny schopna přebírat zboží ze skladu přímo na prodejně.

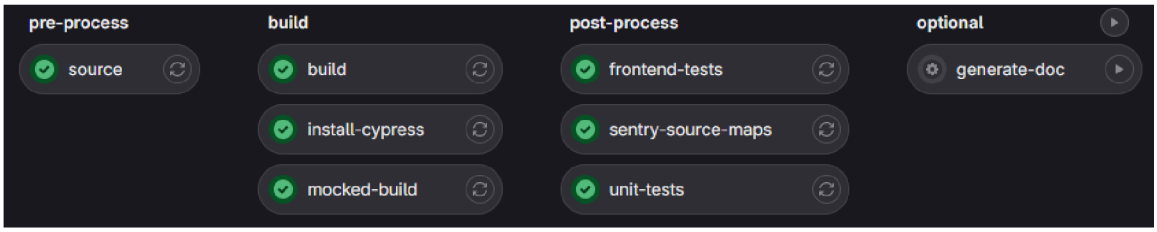
6 Implementace automatizace nasazování do produkce pomocí CI/CD

Tato kapitola se zabývá nastavením a implementací automatizovaného zpracování pokladny v rámci CI/CD procesů. Na začátku je uveden popis, jak jsou jednotlivé části automatizace strukturované a jaký mají účel. Následně je prezentována implementace vybraných technologií, doplněná o příklady pro každý spuštěný proces. Kapitola je zakončena popisem provozu aplikace, s důrazem na monitoring a zachytávání chyb.

6.1 Kroky implementované automatizace

Bloku o fungování CI/CD byl věnován prostor již v sekci 2.3 rešerší. Tato část poskytuje podrobný pohled na dění uvnitř pipeline. Proces začíná ve chvíli, kdy programátor provede vložení změn nebo nových úprav *(git push)* do větve repozitáře. S každou nově vloženou změnou se spouští proces zvaný pipeline, který je rozdělen do několika dílčích sekcí, neboli stages. Ke zpracování automatizace se jako vhodná pomůcka ukázala kniha [12], která podrobně vysvětluje řadu užitečných technik a metodik pro zpracování s využitím CI/CD. V rámci tohoto projektu se jeví jako optimální následující čtyři stage:

- **pre-process** je přípravná fáze, ve které dochází ke kontrole a manipulaci s kódem ještě před samotným sestavením aplikace.
- **build** je proces sestavení react aplikace, během kterého se vytvoří artifact. Z důvodu frontendového testování, kde je API aplikace simulováno *(namockováno)*, se vytvářejí dva buildy aplikace.
- **post-process** spouští frontendové i backendové testy a zároveň generuje source mapy pro nástroj Sentry.
- **optional** obsahuje volitelné úkoly, mezi které patří například možnost generování dokumentace přímo z kódu.



Obrázek 6.1: Fáze a procesy automatizovaného zpracování pokladny pomocí CI/CD

6.2 Přípravná fáze *(pre-process)*

Do této sekce je zahrnuto i samotné nastavení pipeline, jelikož se jedná o přípravu před samotným spuštěním. Prvním krokem před spuštěním je definovat jaký image (obraz) má docker kontejner použít. Pro práci s JavaScriptovými aplikacemi je ideální NodeJS. Následně je třeba definovat jednotlivé stage jak je ukázáno v ukázce.

```
1  image: node:18.17.0
2
3  stages:
4    - pre-process
5    - build
6    - post-process
7    - optional
8
9  cache: &global_cache
10   key: ${CI_COMMIT_REF_SLUG}
11   paths:
12     - node_modules/
13     - .npm/
14     - .cache/*
15     - cache/Cypress
16     - mocked-build/
```

Zdrojový kód 6.1: Konfigurace pipeline před spuštěním

Pro zrychlení opakovaných spuštění byla přidána sekce s cache [1]. Cache je konfigurována s klíčovým označením "${CI_COMMIT_REF_SLUG}", což znamená, že pro každou větev repozitáře bude vytvořena samostatná cache. Dále jsou specifikovány adresáře, které mají být ukládány do cache.

[1] Cache (mezipaměť) je technologie, která dočasně ukládá kopie dat nebo výsledků operací.

6.3 Fáze sestavení *(build)*

V této fázi jsou vytvořeny dva klíčové artefakty pro pokladnu: produkční build a mockovaný build. Produkční artefakt je vystavený na URL adrese GitLab repozitáře. Naopak mockovaný build není publikován, ale je uložen v cache pro další zpracování. Součástí této fáze je také paralelní instalace Cypress balíku a jeho závislostí do cache, což značně urychluje průchod testy na frontendu, jelikož systém nevyžaduje opakované stahování a instalaci balíčků, ale pouze jejich načtení z cache.

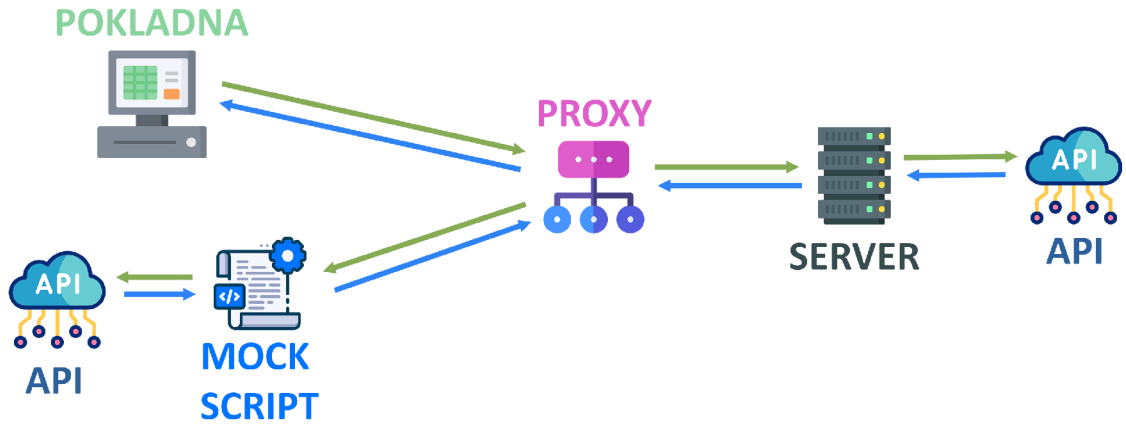
6.3.1 Produkční build aplikace vs mocked build

Sestavení aplikace se provádí pomocí standardního skriptu *react-app-rewired*. Příkazy pro build jsou specifikovány v konfiguračním souboru *package.json*. V rámci pipeline je proces sestavení zjednodušen na spuštění příkazu *npm run build*. Ukázka publikování buildu obsahuje definici cesty *./build*, kam byl projekt sestaven, a podmínku, že publikace proběhne jen v případě, že celý proces sestavení proběhl bez chyb.

```
1  artifacts:
2    paths:
3      - build
4    when: on_success
5    expire_in: 1 week
```

Zdrojový kód 6.2: Nastavení pro vydání produkčního balíku

Hlavním rozdílem mezi produkčním a mocked buildem je způsob komunikace s API. Produkční build odesílá požadavky na adresu skutečného serveru, což může vést k chybovým hláškám, pokud server není dostupný. To je případ spuštění v rámci pipeline. Z tohoto důvodu byl vytvořen mocked build, který simuluje API přímo uvnitř aplikace. Tento přístup umožňuje nezávislý vývoj a testování bez nutnosti připojení k externímu serveru. Rozdíly v komunikaci mezi těmito dvěma typy buildů jsou vizualizovány na přiloženém obrázku 6.2. Vysvětlení funkce mockovaného buildu je vysvětlena v následujícím bloku 6.3.2.



Obrázek 6.2: Typy komunikace s API pro jednotlivá sestavení aplikace

6.3.2 Testovací build s mock API

Sestavení probíhá stejným způsobem jako v produkčním případě. Jediný rozdíl při spuštění sestavení je přiložená proměnná **REACT_APP_MOCK_GRAPHQL=true**, která v pokladně upraví komunikaci, tak aby neodcházela na server, ale byla odchytávána pomocí mock scriptem. Mockovat lze i jiné části než je API.

Pro implementaci mocku byla využita knihovna MSW, která podporuje jak GraphQL, tak REST API. K zapracování knihovny do projektu byl nápomocen článek [11] o této knihovně. Do projektu byla přidána následující struktura.

- **./src/msw** - kořenový adresář mocku
  - **./mocks** - obsahuje browser.ts, kde je implementován a nakonfigurován worker, který simuluje funkcionalitu prohlížeče.

```
export const worker = setupWorker(...handlers);
```

Zdrojový kód 6.3: Vytvoření mocked worker pro prohlížeč

  - **./handlers.ts** - Obsahuje definice jednotlivých endpointů API, včetně REST i GraphQL.

```
graphql.mutation('PosOrderCreate', ({ ... }) => {
  return HttpResponse.json(posOrderCreate);
})
```

Zdrojový kód 6.4: Registrace mockovaného rozhraní

  - **./response** - Adresář obsahuje soubory xxx.json, které obsahují odpovědi poskytované mockovaným API.
- **./public/mockServiceWorker.js** - Worker vygenerovaný pomocí knihovny msw, který začne po spuštění odchytávat odchozí zprávy a přesměrovávat je na mock worker prohlížeče.

Aby bylo možné zapnout mockování, slouží právě výše zmíněná proměnná *REACT_APP_MOCK_GRAPHQL=true*. Mockování je umístěno v souboru index.tsx, kde probíhá vkládání React aplikace do HTML elementu s id root. Pokud je proměnná aktivní, mockServiceWorker.js je zaregistrován před spuštěním aplikace. Z tohoto důvodu nelze aplikaci spustit pouze s parametrem, ale je nutné vytvořit nový build. Ukázka v příloze ilustruje proces registrace tohoto mocku.

```
async function enableMocking() {
  if (process.env.REACT_APP_MOCK_GRAPHQL == 'true') {
    const { worker } = await import('./msw/mocks/browser');
    return await worker.start({...});
  }
  return;
}
enableMocking().then(() => {
    // standardní redern react aplikace do elementu root
});
```

Zdrojový kód 6.5: Spuštění mock workeru před přeložením apliakce

6.4 Dokončovací fáze *(post-process)*

Tato část pipeline je primárně zaměřena na testování aplikace, přičemž se rozlišuje mezi backendovými a frontendovými testy. Backendové testy se soustředí na funkcionalitu pokladny běžící v pozadí, zatímco frontendové testy testují uživatelské rozhraní pokladny z pohledu uživatele.

Paralelně k tomu probíhá generování zdrojových map, které jsou odesílány do systému Sentry. V případě, že dojde k výjimce způsobené chybou v pokladně, Sentry tuto výjimku zachytí. Chyby se nejčastěji objevují v nově přidaném kódu, a proto je zásadní mít v Sentry nejaktuálnější zdrojové mapy pro přesné lokalizování původu chyby.

6.4.1 Backendové testování

Backendové testy jsou implementovány s využitím knihovnu Jest. Jak již bylo výše zmíněno, tak tyto testy pokrývají pouze funkce pracující na pozadí. Nastavení těchto testů je umístěno v kořenovém adresáři pokladny v souboru jest.config.js. Zde je pár důležitých nastavení.

- **preset: 'ts-jest'** - testy používají preset pro TypeScript, tedy i samotné testy mohou být napsány v TypeScriptu.
- **testRegex: "..."** - specifikace umístění testů, aby se nespouštěli všechny soubory v pokladně.
- **moduleNameMapper: {}** - seznam modulů pokladny, zjednodušení cesty k zanořeným souborům. Např. *@components* nahradí cestu *<rootDir>/src/components/*

Testy jsou v rámci CI spouštěny příkazem 'npm run test – –testResultsProcessor="jest-junit" –watchAll=false –ci –coverage'. Přiložené parametry říkají jak se mají testy zpracovat a jak vygenerovat výsledek proběhnutí pro GitLab. V GitLabu jsou tyto statistiky zobrazeny pro konkrétní doběhnutou pipeline.

```
1 it('Test: get total price', () => {
2   let response: any = getRecalculatedOrderData();
3   updateOrder(response, order);
4   expect(getOrderTotalPriceWithVat(order)).toBe('1223,00');
5   expect(getOrderTotalPriceWithoutVat(order)).toBe('1010,74');
6 });
```

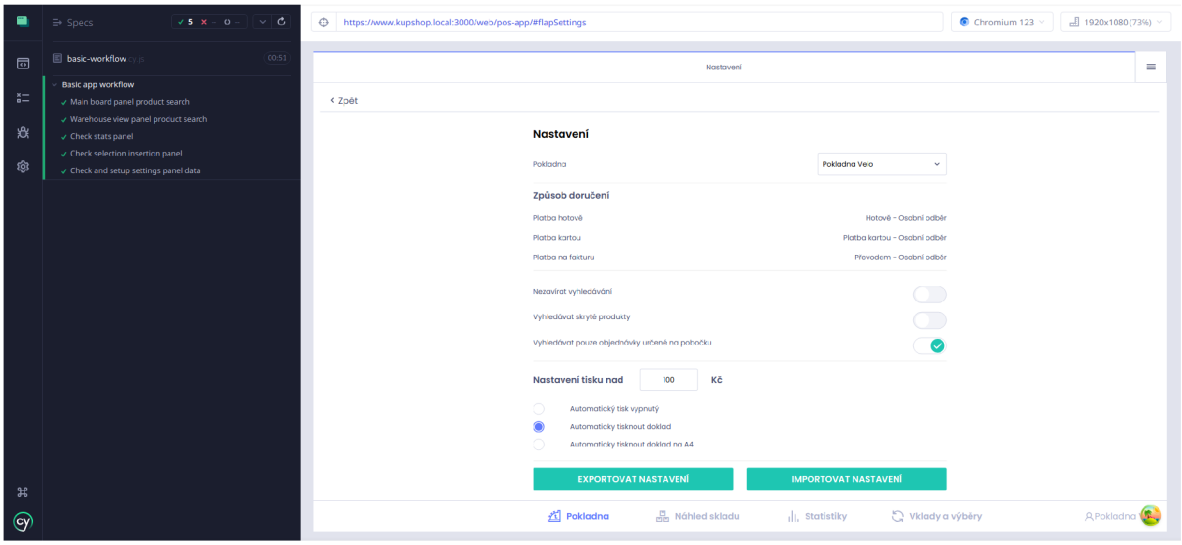
Zdrojový kód 6.6: Ukázka struktury backendového testu pro přepočet objednávky

Jelikož se jedná o ReactJS aplikaci, kde velkou část aplikace tvoří frontend a na něj navěšené hooky a callbacky, tak backendové testování není zcela nejefektivnější metodou jak testovat běh pokladny. Z tohoto důvodu byly přidány frontend testy popsané v následující sekci.

6.4.2 Frontendové testování

Pro testování frontendu byl použit framework **Cypress**, konkrétně pro provádění **E2E** testů. Konfigurace testů se provádí skrze soubor *cypress.config.ts*, který umožňuje například zapnutí záznamu videí z testování pomocí *video: true*, nastavení environmentálních proměnných pomocí *env:* nebo vypnutí izolace testů pomocí *testIsolation: false*. Testy a související nástroje jsou uloženy v adresáři *cypress/e2e*. Jednotlivé testy jsou označeny příponou *.cy.js*, například soubor *basic-workflow.cy.js* obsahuje test základního průchodu všemi stránkami v pokladním systému.

Cypress také nabízí vlastní editor pro psaní a spouštění testů. V adresáři s pokladnou lze pomocí příkazů **npx cypress install** nainstalovat závislosti do cache a **npx cypress open** poté otevře editor pro správu testů. K nastavení a zprovoznění frameworku s react aplikací byl nápomocen článek [21], který je věnován tomuto tématu.



Obrázek 6.3: Cypress editor s testem na základní průchod pokladnou

Spuštění testů pro frontend v rámci CI je složitější než provádění backendových testů. Pro účely testování frontendu je nezbytné, aby byla aplikace hostována, a zároveň musí být dostupné API. Jak již bylo dříve zmíněno, komunikační rozhraní s API je nahrazeno mockem. Následuje postup, jak aplikaci hostovat a jak spustit samotné testy v rámci CI.

1. Doinstalování balíčku **serve** pro možnost "light-weight" hostování z konzole.
2. Instalace **pm2** (process manager) aby se hostování aplikace spustilo jako process do pozadí.
3. Spuštění **hostování mockované aplikace** a krátké vyčkání na hosting.

```
1    pm2 start "serve -p 3000 -s mocked-build -n" --name FMS
2    echo "Waiting for web server run ..." && sleep 5s
```

Zdrojový kód 6.7: Hosting aplikace uvnitř CI

4. Spuštění samotných E2E cypress testů s nastavovacími parametry.

```
1     npx cypress run
2         --browser chrome
3         --config baseUrl=...
4         --reporter junit
5         --reporter-options "toConsole=true"
```

Zdrojový kód 6.8: Spuštění E2E testlů v CI

Pokud jeden z testů selže, automaticky se uloží video a snímky obrazovky jako artefakty. Doba, po kterou jsou tyto soubory dostupné, je závislá na nastavení, které v tomto případě činí jeden den. Výsledky úspěšně dokončených testů jsou zobrazeny v konzoli příslušného procesu.

6.4.3 Generování zdrojových map do Sentry

Generování zdrojových map a jejich odesílání do Sentry je implementování hlavně pro sledování a diagnostiku chyb v produkčním prostředí. Když Sentry zachytí výjimku, k dané chybě je vypsán stack trace [2]. Vzhledem k tomu, že chyby nejčastěji vznikají v nově přidaném kódu, je důležité, aby se s každou změnou kódu automaticky generovaly nové zdrojové mapy a nahrávaly do Sentry pomocí procesů CI/CD.

Prvním krokem je vytvoření zdrojové mapy pro pokladnu. V rámci CRA je k dispozici proměnná **GENERATE_SOURCEMAP**, která při zapnutí při buildu automaticky generuje složku .map obsahující zdrojové mapy. Pro nahrání zdrojových map do Sentry je potřeba doinstalovat nástroj sentry-cli. Tento nástroj nejprve doplní build o metadata, která obsahují informace o vztahu zdrojových map k odpovídajícím minifikovaným souborům. Poté je pomocí stejného nástroje obsah nahrán na server Sentry. Zde je ukázka CI/CD procesu pro přegenerování zdrojových map v Sentry:

```
1   variables:
2     SENTRY_AUTH_TOKEN: ...
3   only:
4     - master
5   script:
6     - GENERATE_SOURCEMAP=true npm run build
7     - sentry-cli sourcemaps inject --org wpj --project pos
     ./build
8     - sentry-cli --url URL sourcemaps upload --org wpj --
     project pos ./build
```

Zdrojový kód 6.9: Generování zdrojových map do Sentry

Při nahrávání je proces ověřován na serveru skrze token, který je specifikován v proměnných prostředí. Proces je nastaven tak, aby se spouštěl pouze na hlavní

[2] Stack trace je seznam volání funkcí nebo metod, které vedly k chybě nebo výjimce v programu.

větvi master. Toto nastavení zajišťuje, že v produkčním monitoringu se neobjeví změny, které ještě nebyly nasazeny do produkce.

6.5 Vygenerování dokumentace (volitelné)

Generování dokumentace je zatím považováno za nepovinnou činnost. Tato dokumentace je určena pro vývojáře a vytváří se na základě komentářů přímo v kódu. V budoucnosti má dokumentace usnadnit jiným vývojářům přidávání nových funkcí a úpravy stávajícího kódu pokladny. V rámci bakalářské práce již byla vytvořena dokumentace pro klienty, avšak proces seznámení se s kódem pokladny nebyl dosud formálně zdokumentován.

Pro vygenerování dokumentace ve formě webové stránky byla využita knihovna **JSDoc**. Vzhledem k tomu, že pokladna je napsána v TypeScriptu, bylo nutné implementovat také podporu Babel transpileru prostřednictvím knihovny **jsdoc--babel**. Výsledná webová stránka s dokumentací je generována do složky ./docs. Níže je příklad postupu generování dokumentace.

```
1   when: manual
2   script:
3     - npm install -g jsdoc
4     - npm install -D jsdoc-babel
5     - npm run docs
```

Zdrojový kód 6.10: Vygenerování dokumetace kódu pokladny

Jako optimální řešení pro hosting dokumentace se nabízelo využít stránky GitLab Pages přímo související s projektem. Avšak podle dokumentace [8] musí být webová stránka umístěna ve složce *./public* repozitáře a obsahovat soubor *index.html* v této složce. Tento požadavek však koliduje s existující React aplikací, která již složku *./public* a soubor *index.html* využívá. Kvůli této kolizi by bylo nutné zřídit samostatný hosting pro dokumentaci. Z těchto důvodů zůstává dokumentace prozatím dostupná ke stažení jako artefakt.

6.6 Nasazení pokladny do produkce

Posledním a zároveň klíčovým krokem této automatizace je automatické nasazení do produkčního prostředí. Tento proces se mírně liší od standardního nahrání aplikace na hosting nebo server. Vzhledem k tomu, že pokladna je navržena jako modul pro e-shop, vyžaduje specifický přístup k nasazení. Z tohoto důvodu není nasazení zahrnuto přímo v procesu CI/CD pokladny, ale místo toho je integrováno do procesu sestavení samotného e-shopu. Podrobný popis tohoto postupu je vysvětlen v následujících blocích.

6.6.1 Sestavení pokladny s e-shopem

E-shop je strukturován do několika částí, přičemž největší a nejdůležitější je jeho jádro, které zahrnuje serverové GraphQL API a za ním stojící logiku. Na toto jádro navazují specificky upravené části pro daný e-shop. Do systému jsou poté integrovány různé moduly, včetně pokladny. V konečné fázi kdy jsou poskládány všechny části, tak je build e-shopu publikován na příslušný hosting.

Tento přístup nabízí výhodu v tom, že pokud jsou správně vytvořeny verze, je možné předejít rozdílům mezi verzí serverového API pokladny a samotnou pokladnou. Proces sestavování e-shopu je však složitý, a proto je dokumentace implementace omezena jen na části týkající se pokladny. V rámci CI konfigurace je spouštěn skript build.sh, který zajišťuje stažení a rozbalení jednotlivých modulů. Pro modul pokladny je zde připojena specifická část.

```
  if has_module "new_pos"; then
   DIR=$PWD
   if [[ -z "${BUILD_POS}" || "${BUILD_POS}" == "1" ]]; then
     BUILD_POS="master"
   fi
   mkdir -p web/pos-app
   cd web/pos-app
   curl --location --output data.zip --header "PRIVATE-TOKEN" "URL"
   unzip data.zip > /dev/null
   mv build/* .
   rm -rf build/ data.zip
   cd ${DIR}
fi
```

Zdrojový kód 6.11: Sestavení pokladny s e-shopem v CI/CD

Skript nejprve ověří, zda je modul pokladny aktivní, a z načteného konfiguračního souboru .yml získá informace o nastavené větvi. Pokud není specifikovaná, použije výchozí větev master. Poté se vytvoří a přepne do specifického podadresáře určeného pro pokladnu. Pomocí příkazu curl stáhne archiv pokladny a následně jej rozbalí. Na závěr skript odstraní všechny zip soubory, které již nejsou potřebné. V e-shopu je pak pokladna dostupná podle metodiky popsané v bakalářské práci.

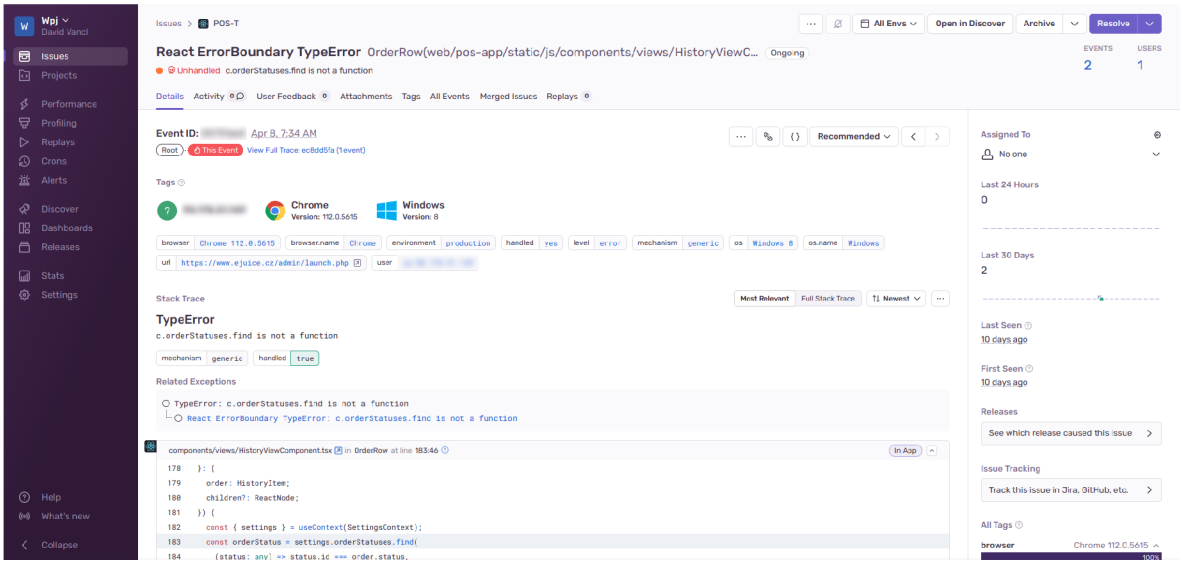
6.6.2 Běh a monitoring aplikace

Aplikace v produkčním prostředí může být ovlivněna různými faktory, které nejsou vždy během vývoje plně zohledněny. Chyby se nikdy nevytvářejí úmyslně a je nevyhnutelné, že dříve nebo později nějaká nastane. Proto je zásadní mít možnost získat zpětnou vazbu o tom, kdy a kde došlo k chybě.

Sentry

Pro sledování a řešení chyb v reálném čase je použit nástroj Sentry. Sentry monitoruje, zachytává výjimky a odesílá je na server. Implementováno je také zasílání

notifikací vývojářům přes Slack[3] pomocí Slack bota. Odchycená chyba z pokladny je znázorněna na přiloženém obrázku 6.4.



Obrázek 6.4: Sentry: odchycená chyba z pokladny

Implementace Sentry je integrována přímo do kódu pokladny s použitím knihovny **@sentry/react**, což vychází z její dokumentace [17]. Nejprve je knihovna inicializována pomocí přístupových údajů a poté je hlavní komponenta *<App/>* obalena komponentou pro zachycení chyb, jak ukazuje následující příklad.

```
if (process.env.NODE_ENV === 'production') {
  Sentry.init({
    dsn: '...',
    integrations: [new Sentry.BrowserTracing()],
  });
}
<Sentry.ErrorBoundary fallback={...}>
    <App />
</Sentry.ErrorBoundary>
```

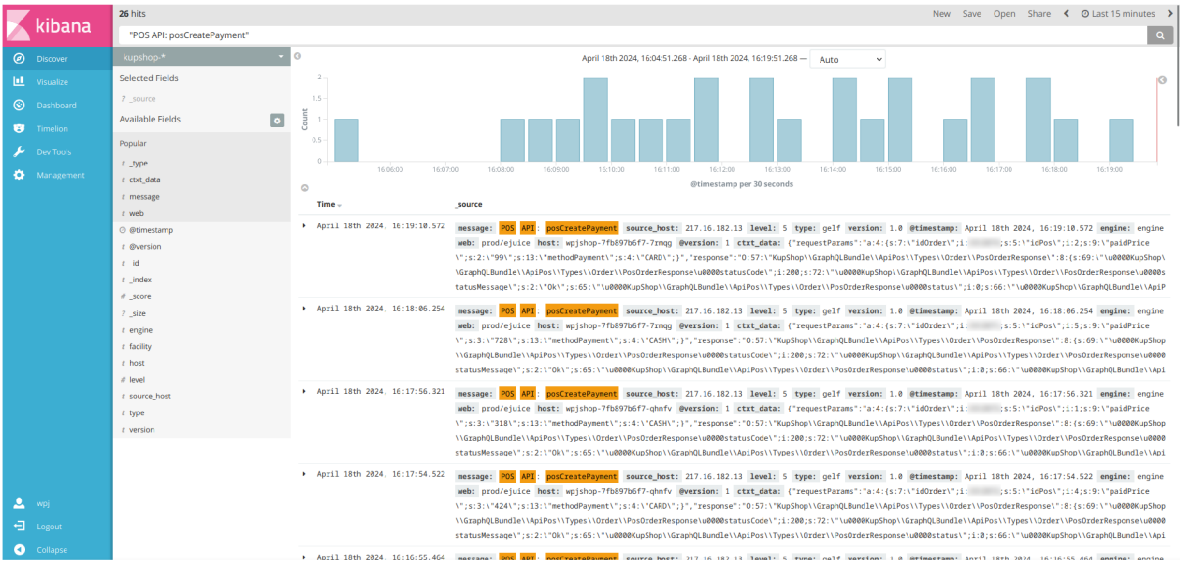
Zdrojový kód 6.12: Odchytávání chyb do Sentry v React aplikaci

Kibana a Elasticsearch

K zaznamenávání komunikace je využita webová aplikace Kibana, která slouží jako uživatelské rozhraní pro Elasticsearch[4]. V případě pokladny uchovává logy komunikace mezi pokladnou a serverem. Tyto data jsou velmi užitečná při zpětnému dohledávání jak chyb, tak například využitelnosti jednotlivých částí pokladny.

[3]Slack je platforma pro komunikaci, která nabízí chatovací místnosti (kanály), soukromé zprávy a integrační nástroje.

[4]Elasticsearch je vyhledávací a analytický engine, který se používá pro velké množství dat.



Obrázek 6.5: Kibana: Provedení plateb skrze pokladnu za posledních 15 minut

Logování na straně serveru je implementováno přímo v controlleru GraphQL API. Pro záznam dat do databáze je používána služba Symfony, pojmenovaná jako Logger, jak je vidět v přiložené ukázce. Tato služba obsahuje kód nezbytný pro navázání spojení s databází a uložení dat. Logování je aplikováno na všech klíčových endpointech, jako je například vytváření objednávek nebo proces plateb.

```
    $this->logger->notice('POS API: posCreatePayment', ['data' => [
        'requestParams' => serialize($request),
        'response' => serialize($response),
    ]]);
    /** @required */
    public function setLogger(LoggerInterface $logger): void {
        $this->logger = $logger;
    }
```

Zdrojový kód 6.13: Logování komunikace v GraphQL API do Kibany

7 Závěr

Diplomová práce představila rozšíření a optimalizaci pokladního systému, který byl navržen a vytvořen v předchozí autorově bakalářské práci. Hlavním cílem bylo implementovat procesy Continuous Integration (CI) a Continuous Deployment (CD), což výrazně zlepšilo schopnost systému reagovat na změny prostřednictvím automatizace testování a nasazení. Hlavním vylepšením byla integrace nového platebního terminálu a zejména podpora skladového systému, což umožnilo plynulejší transakční procesy a lepší správu zásob. Implementace podpory skladového systému do pokladny znamenala významný posun v efektivitě správy zásob a inventury. Tento krok vede k okamžitému aktualizování stavu skladu při každé transakci.

V rámci diplomové práce byla realizována řada důležitých vylepšení pokladního systému, odpovídajících požadavkům klientů. Implementace podpory pro slovenskou elektronickou evidenci tržeb (SK EET) je zásadní pro splnění legislativních norem v oblasti evidování tržeb. Byla také rozšířena funkčnost čtečky, včetně nové implementace v pokladním systému a vytvoření simulátoru prostřednictvím mobilního telefonu, což zvyšuje flexibilitu v případě nedostupnosti fyzické čtečky. Dále byly vylepšeny procesy zaplacení a příjmu zboží, které nyní usnadňují manipulaci se zbožím a finanční transakce. Rozšíření nastavení oprávnění pro administrátory zlepšuje kontrolu přístupu a zabezpečení systému. Navíc byla implementována řada dalších vylepšení, která celkově zlepšují funkčnost, efektivitu a uživatelskou přívětivost systému.

Klíčová část práce je věnována implementaci dynamického přidávání funkčních částí do pokladního systému, což představuje značnou výhodu v oblasti modularity a rozšiřitelnosti systému. Tento přístup umožňuje snadné a rychlé začlenění nových funkcí a komponent do stávajícího systému bez nutnosti komplexních zásahů do jeho kódu. Tento mechanismus nejen zvyšuje flexibilitu a adaptabilitu pokladního systému na měnící se požadavky trhu, ale také významně urychluje proces vývoje a nasazení nových funkcí.

Celkově práce dokázala, že rozšíření pokladního systému a zavedení moderních vývojových praktik může zásadně přispět k efektivitě, spolehlivosti a bezpečnosti systému. Podařilo se dosáhnout významného pokroku ve zjednodušení a automatizaci operací, což přispívá k lepšímu pochopení požadavků moderního obchodního prostředí. Taková flexibilita je zvláště cenná v prostředí, kde se obchodní požadavky a technologie neustále vyvíjejí. Tato diplomová práce přináší jak technické, tak praktické výhody a poskytuje solidní základ pro další rozvoj pokladního systému a jeho dalších oblastí.

Seznam použité literatury

[1] A.S., Comgate. *Example of Background Payment Setup – HTTP response* [online]. 2024. [cit. 2024-03-21]. Dostupné z: https://apidoc.comgate.cz/?lang=en#payment-process.

[2] A.S., Comgate. *Platební terminál* [online]. 2024. [cit. 2024-03-21]. Dostupné z: https://www.comgate.cz/platebni-terminal.

[3] ALEX BANKS, Eve Porcello. *Learning React, 2nd Edition*. 2. vyd. O'Reilly Media, Inc., 2020. ISBN 9781492051725. Dostupné také z: https://www.oreilly.com/library/view/learning-react-2nd/9781492051718/.

[4] BANKA, Fio. *Platební terminály* [online]. 2024. [cit. 2024-03-22]. Dostupné z: https://www.fio.cz/bankovni-sluzby/platebni-karty/platebni-terminaly-brana#Zadost_POS_Platebni_brana.

[5] CONTRIBUTORS, MDN. *Browser extensions* [online]. 2024. [cit. 2024-04-06]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions.

[6] EVE PORCELLO, Alex Banks. *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. 1. vyd. O'Reilly Media, Inc., 2018. ISBN 978--1492030713. Dostupné také z: https://www.oreilly.com/library/view/learning-react-2nd/9781492051718/.

[7] FACEBOOK, Inc. *Getting Started* [online]. 2022. [cit. 2024-03-25]. Dostupné z: https://create-react-app.dev/docs/getting-started.

[8] GITLAB. *GitLab Pages* [online]. 2023. [cit. 2024-04-19]. Dostupné z: https://docs.gitlab.com/ee/user/project/pages/.

[9] KOLA-BALOGUN, Adetoyese. *Configuring a GitLab CI/CD pipeline in your react app* [online]. 2022. [cit. 2024-03-23]. Dostupné z: https://toyesebalogun95.medium.com/configuring-a-deployment-pipeline-in-gitlab-7e3e09aeda1b.

[10] LAVIALE, Olivier. *Generate locators from tagged services with Symfony 4.3* [online]. 2019. [cit. 2024-04-11]. Dostupné z: https://olvlvl.com/2019-09-symfony-tagged-service-locator.html.

[11] MANTHA, Kiran. *Implementing and Unit testing Graphql in React using msw* [online]. 2023. [cit. 2024-04-17]. Dostupné z: https://dev.to/kiranmantha/implementing-and-unit-testing-graphql-in-react-using-msw-2667.

[12] MERODE, Henry van. *Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines*. 1. vyd. Apress, 2023. ISBN 978-1484292273. Dostupné také z: https://www.amazon.com/Continuous-Integration-Delivery-Practical-Developing/dp/1484292278.

[13] MRAJAEIM. *Understanding window.postMessage and window.parent.postMessage in JavaScript* [online]. 2023. [cit. 2024-04-12]. Dostupné z: https://medium.com/@mrajaeim/understanding-window-postmessage-and-window-parent-postmessage-in-javascript-f09d4eac68ba.

[14] NINE DIGIT, s.r.o. *Portos eKasa - Dokumentácia* [online]. 2024. [cit. 2024-04-07]. Dostupné z: https://developer.ninedigit.sk/ekasa/.

[15] S.R.O., A3 Soft. *FiskalPRO T6* [online]. 2024. [cit. 2024-03-21]. Dostupné z: https://fiskalpro.cz/fiskalpro-t6/.

[16] SALEHI, Sohail. *Mastering Symfony*. 1. vyd. Packt Publishing Limited, 2016. ISBN 9781784390310. Dostupné také z: https://www.nejlevnejsi-knihy.cz/kniha/mastering-symfony.html.

[17] SENTRY. *React* [online]. 2023. [cit. 2024-04-19]. Dostupné z: https://docs.sentry.io/platforms/javascript/guides/react/.

[18] SYMFONY. *DoctrineMigrationsBundle* [online]. 2024. [cit. 2024-04-04]. Dostupné z: https://symfony.com/bundles/DoctrineMigrationsBundle/current/index.html.

[19] SYMFONY. *The EventDispatcher Component* [online]. 2024. [cit. 2024-04-05]. Dostupné z: https://symfony.com/doc/current/components/event_dispatcher.html.

[20] ŠKOP, Luděk. *Platební terminály 2024 – Velké nezávislé srovnání poskytovatelů* [online]. 2024. [cit. 2024-02-06]. Dostupné z: https://blog.inspirum.cz/platebni-terminaly-2024-velke-nezavisle-srovnani-poskytovatelu-cr/.

[21] TECHNOCRAT, Community Contributor. *How to Run Cypress Tests for your Create-React-App Application* [online]. 2023. [cit. 2024-04-18]. Dostupné z: https://www.browserstack.com/guide/run-cypress-tests-for-create-react-app.

[22] ZAIN SAJJAD, Dan Ackerson. *Why End-to-End (E2E) Testing is Often Good Enough* [online]. 2022. [cit. 2024-03-25]. Dostupné z: https://semaphoreci.com/blog/e2e-testing.

Přílohy

K diplomové práci je přiloženo CD, které obsahuje následující materiály:

Adresář přiloženého CD:

- Diplomová práce David Vancl 2024.pdf
- Diplomová práce David Vancl 2024 - LaTeX.zip
- Diplomová práce David Vancl 2024 - zdrojové kódy.zip
- Diplomová práce David Vancl 2024 - cypress test video.mp4
- Diplomová práce David Vancl 2024 - android aplikace a rozšíření.zip