# Seminární práce pro předmět Softwarové inženýrství

Denisa Krebsová, Filip Stehlík, Prokop Mikulášek

Aktuální k: 26. 5. 2024

Univerzita Jana Evangelisty Purkyně v Ústí nad Labem

Git repositář: https://github.com/mikulpro/used_sources_database.git

Obraz na docker hubu: https://hub.docker.com/r/stehlfi1/used_sources_database_swi

# Obsah

# Popis řešitelského týmu

## Denisa Krebsová

Jakožto studentka programu Aplikovaná informatika na Univerzitě Jana Evangelisty Purkyně v Ústí nad Labem má Denisa bohaté zkušenosti s programovacím jazykem Python ve věci programování databází. Již ve druhém roce studia pomáhala s celouniverzitním programem pro správu přístupů do jednotlivých učeben pro recepci univerzity.

V našem projektu sehrála Denisa úlohu softwarové developerky, její hlavní úlohou byla komunikace s databází a objektový přístup k jejím entitám a její druhotnou rolí byla tvorba Docker image, protože s tím má ze členů řešitelského týmu asi největší zkušenosti. Zároveň obstarání databázové části práce, díky její předchozí zkušenosti.

## Filip Stehlík

Filip je bývalým studentem ČVUT v Praze (oboru Kybernetika a robotika) a současným studentem UJEP v Ústí nad Labem. Také Filip má bohaté zkušenosti s programovacím jazykem Python, které využil zejména při řešení problematiky zpracování a analýzy elektrického signálu EKG.

Filip v našem projektu zastává druhotnou roli testera a je zodpovědný za správnou implementaci restx modelů, aby náš program splňoval kritéria REST API rozhraní.

## Prokop Mikulášek

Prokop v současné době též studuje UJEP v Ústí nad Labem, kde se hodně sblížil s jazykem Python. Jeho vztah k programování začal již na gymnáziu, na němž Prokop jako svou seminární práci řešil problematiku šifrovaného zasílání zpráv.

V rámci tohoto projektu Prokop navrhnul základní strukturu aplikace a dále je zodpovědný za většinu textů jiné povahy než programovací, vzniknuvších v souvislosti s touto seminární prací. Prokop je také zodpovědný za většinu diagramů.

# Popis softwaru

Jako téma této seminární práce bylo zvoleno úložiště použitých textových zdrojů při psaní odborných prací, při nichž je nutné mít přehled o konkrétních použitých tiskovinách.

Software zahrnuje databázi Postgres s rozhraním splňujícím předlohu REST API (s komunikací pomocí .json souborů) připraveným k přijímání HTTP požadavků. Software dále zahrnuje instrukce pro Docker, takže je možné jej kdykoli složit do Docker obrazu a nasadit.

Software nezahrnuje – vyjma debugovacích a testovacích skriptů – žádné grafické rozhraní, nicméně je připraven pro nasazení nějakého GUI.

## Popis procesu vývoje softwaru a jeho konkrétní nastavení

Vývoj tohoto softwaru začínal návrhem funkcí pomocí use case diagramu a jeho rozšíření o návrhy, které funkce budou zprostředkovány kterým z GET, POST, PUT, DELETE požadavků.


Dále jsme si jako tým rozdělili programovací část práce na různé malé podúlohy, které jsme si zavedli do Github Projects jako položky v seznamu “to do”. Následný vývoj probíhal tak, že si každý člen týmu ve chvíli, kdy měl čas věnovat se tomuto projektu, přeložil položku ze seznamu “to do” do seznamu “in progress” a pracoval na ní od začátku až do konce a to včetně testování funkčnosti výsledku.

Náš proces vývoje, kdy jsme společně od začátku do konce prošli fázi sběru požadavků na finální produkt a zároveň bylo nutné, aby byl program při každé změně stále spustitelný, použitelný a testovatelný, bychom označili jako FDD (Feature-Driven Development).

Jeden z prvků extrémního programování, který jsme při vývoji použili, byla práce ve společné git branchi a neustálé testování funkčnosti všemi pracujícími členy týmu – jinými slovy nebylo možné přidat nefunkční část kódu, aniž by si toho někdo všimnul. (Tedy inkrementační metodika vývoje.)

Pokud budeme mluvit o programování jednotlivých funkcí jako o iteracích, byť to nejsou iterace vývoje softwaru v pravém smyslu slova, můžeme říci, že délka jednotlivých iterací byla v řádu jednotek hodin.

## Seznam funkčních a mimofunkčních požadavků

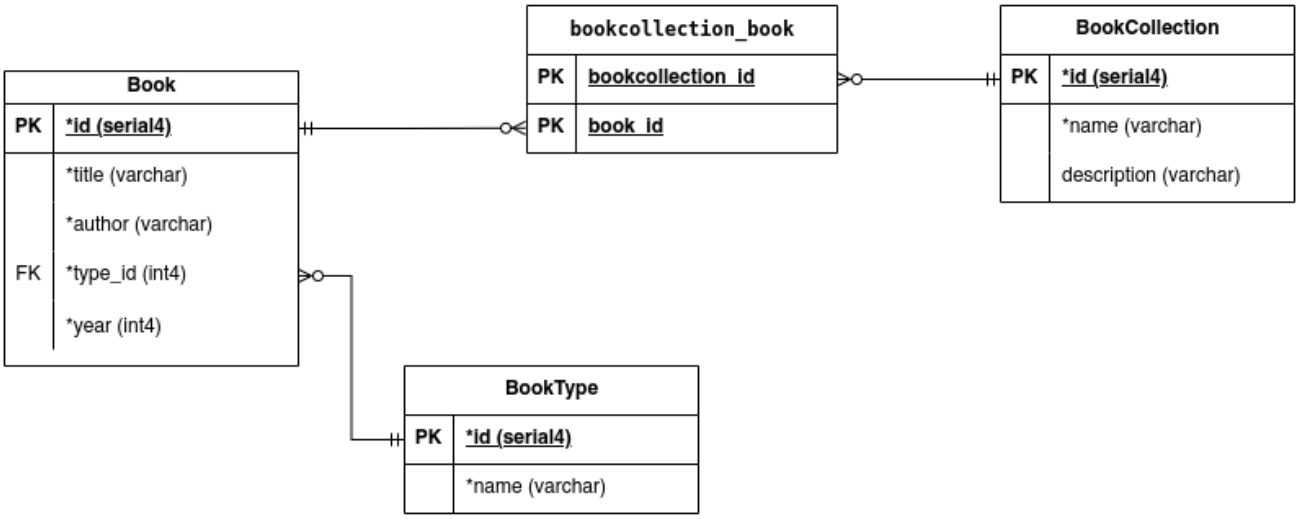
Funkční požadavky:

- Přijetí POST požadavku s .json souborem a vložení dodaných informací do databáze.
- Přijetí GET požadavku s parametry a odpověď na něj informacemi o záznamech v databázi se shodnými parametry.
- Přijetí PUT požadavku s .json souborem a případná úprava záznamu v databázi.
- Přijetí DELETE požadavku s parametry a následné odstranění záznamu z databáze s nejvíce shodnými parametry.
- Nakonfigurovat Docker, aby bylo možné každou verzi databáze a rozhraní zabalit a sdílet jako Docker image.
- Odpovědi na chybné požadavky zprávou o vzniklé chybě se správným HTTP kódem.
- Vytvoření automatizovaných testů pro průběžné ověřování funkčnosti jednotlivých součástí softwaru.

Mimofunkční požadavky:

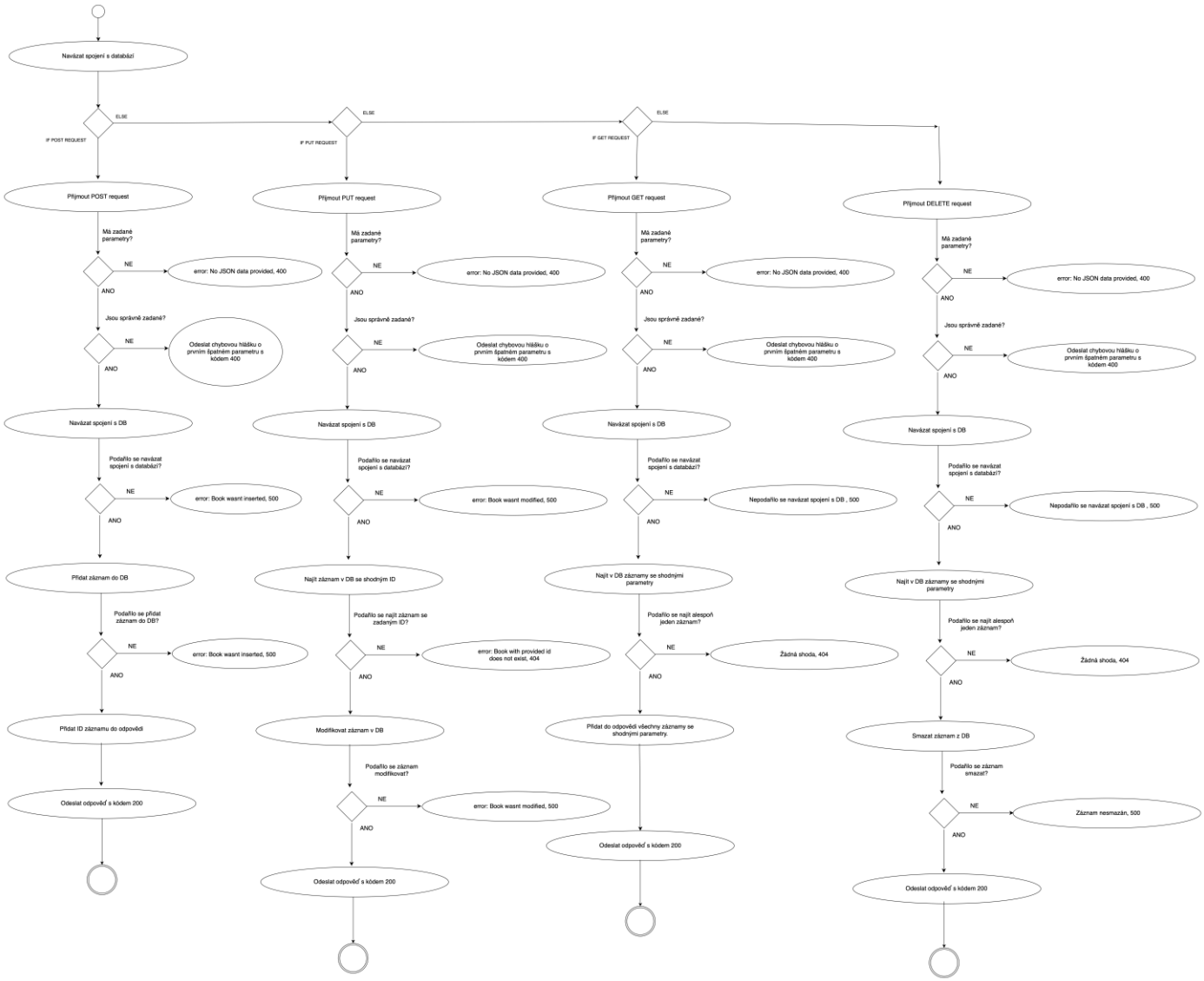
- (Bezpečnost) Pracovat s daty v databázi pomocí objektového přístupu, aby se předešlo SQL injection útokům.
- (Portabilita) Kvůli různorodosti zařízení musí software a v něm využité knihovny fungovat na Ubuntu 22, Windows 11, Windows 10 a MacOS 14.
- (Kompatibilita) Dodržet standard REST API s komunikací pomocí .json souborů a HTTP requestů, aby na náš software šlo nasadit libovolné uživatelské rozhraní napsané s ohledem na tyto standardy.
- (KISS) Mělo by být jednoduché porozumět tomu, co která část kódu dělá.
- Napsat k softwaru odpovídající dokumentaci.

## Návrh databáze – ER diagram

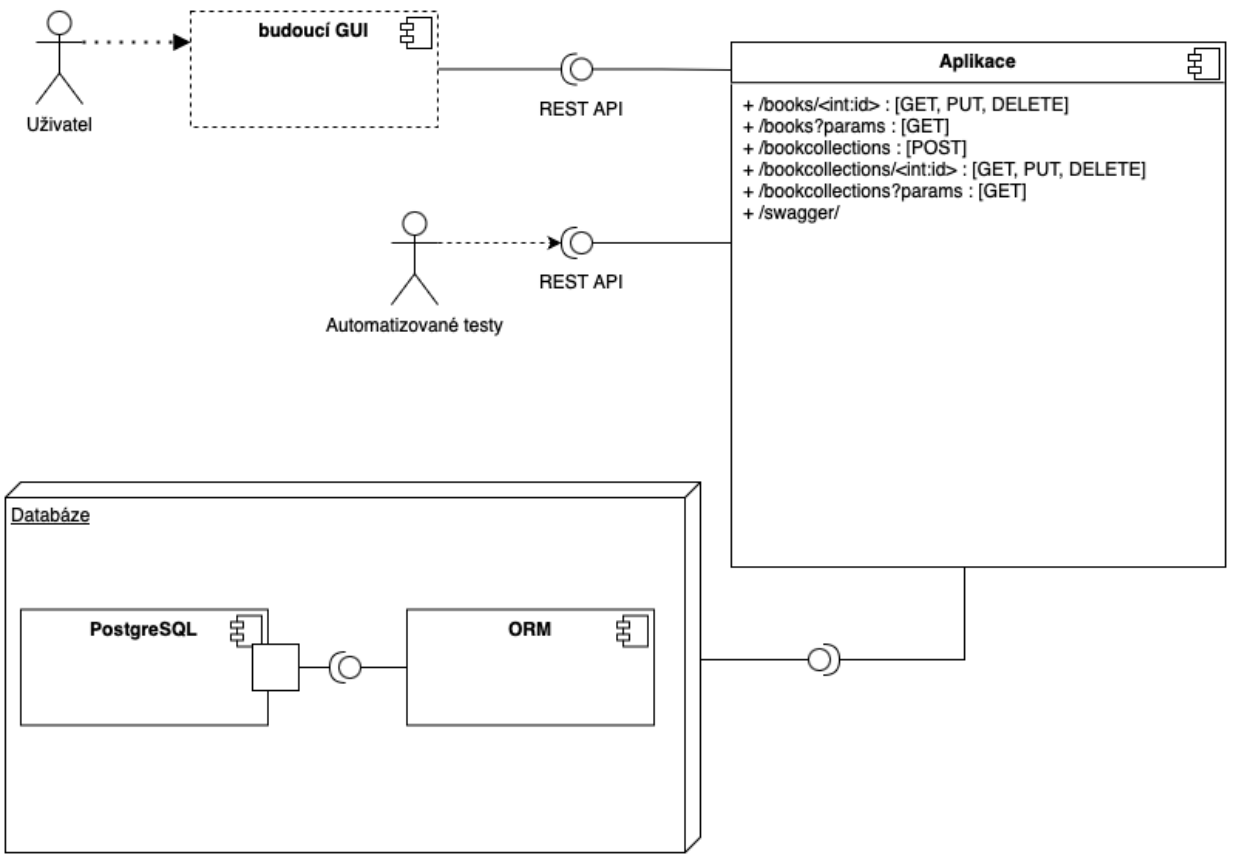


# UML diagramy popisující architekturu softwaru

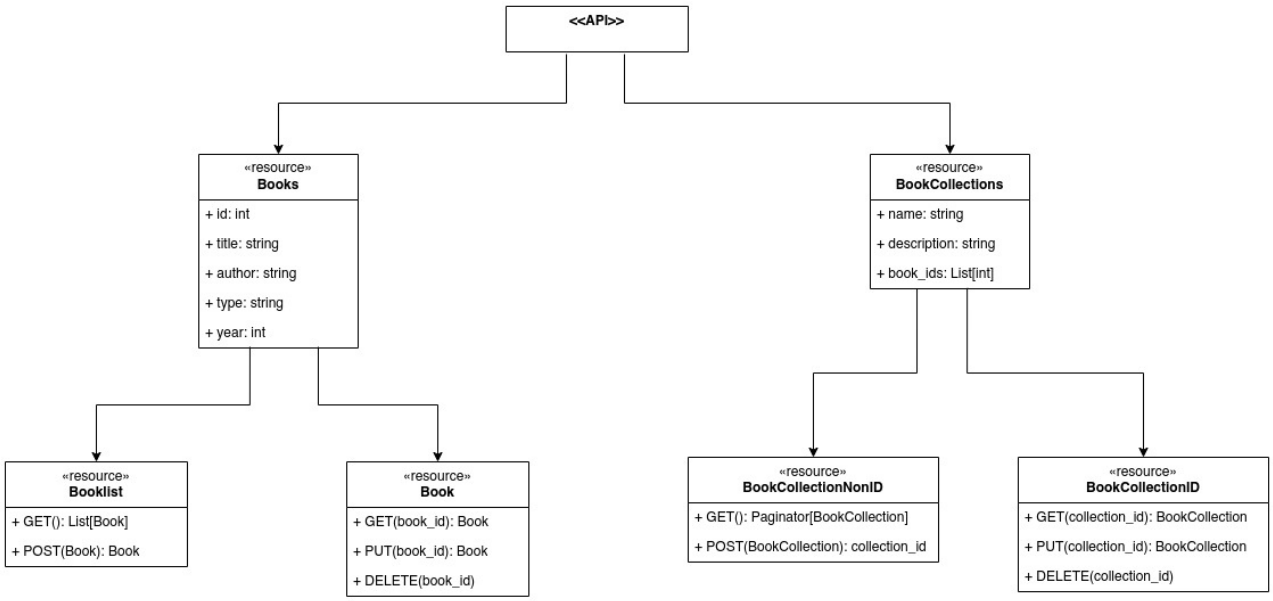
Activity diagram:



Component diagram:




Class diagram of resources:



## Popis implementovaných služeb

Vyvíjený software implementuje obsluhu GET, POST, PUT a DELETE požadavků odeslaných, respektive přijatých pomocí HTTP komunikace – splňuje REST API framework.

Software obsahuje interaktivní dokumentaci pomocí Swagger dostupnou přes /swagger.

Náš SW pomocí příslušných HTTP requestů umožňuje vkládat objekty reprezentující tiskoviny do databáze buď po jednom, nebo případně umožňuje poslat v requestu celou kolekci více objektů reprezentujících tiskoviny, z nichž každý bude uložen do databáze. Dále je užitím GET, PUT a DELETE požadavků možno objekty v databázi upravovat, mazat či je stahovat.

## Vybrané důležité partie kódu a jejich vysvětlení

Náš software je psaný v jazyce Python (verze 3.9 či novější) a využívá několik knihoven popsaných v requirements.txt.

LAMBDA je vždy cool

```
# Define individual book model
book_model = api.model(
    "Book",
    {
        "id": fields.Integer,
        "title": fields.String,
        "author": fields.String,
        "type": fields.String(
            attribute=lambda book: book.type.name
            if type(book) == Bookdb and book.type
            else None
        ),
        "year": fields.Integer,
    },
)
```

Kolekce Endpoint DELETE

Swag(ger) dokumentace api docs uvnitř codu

```
    @api.doc(description="Delete a book collection.")
    @api.response(200, "Collection deleted successfully.")
    @api.response(404, "Collection not found")
    @api.response(500, "Internal Server Error")
    def delete(self, collection_id):
        collection = (
            db.session.query(BookCollectiondb)
            .filter(BookCollectiondb.id == collection_id)
            .first()
        )
        if not collection:
            return {"error": f"Collection with id {collection_id} not found"}, 404
        try:
            db.session.delete(collection)
            db.session.commit()
            return {"message": f"Collection {collection_id} deleted successfully"}, 200
        except Exception as e:
            api.logger.error(
                f"Failed to delete collection with id {collection_id}! {e}"
            )
            return {"error": "Failed to delete collection"}, 500
```

## Jak nasadit a monitorovat naši aplikaci na server? Má nějaká omezení?

Stáhněte si a zprovozněte Docker.

Docker image je dostupný na: https://hub.docker.com/r/stehlfi1/used_sources_database_swi

Naklonujte si Git repozitář s aplikací dostupný na: https://github.com/mikulpro/used_sources_database.git

Ve složce s repozitářem (implicitně pojmenované used_sources_database) si otevřete terminál a příkazem "docker compose up" spustíte aplikaci. (Při prvním spuštění budete muset počkat na stažení Postgres image a instalaci dependencies.)

Aplikace je závislá na Pythonu verze alespoň 3.9 a těchto modulech:

Greenlet, Flask-SQLAlchemy, Flask, Flask-RESTful, Flask-RESTX, typing_extensions, werkzeug, requests, logging, psycopg2, psycopg2-binary

Aplikace je omezená tím, kolik položek se vejde do databáze vzhledem k volné paměti zařízení, na němž je nasazená.

Monitorovat lze aplikaci například automatickým logováním, které provádí Flask do souboru app.log

## Budoucnost softwaru

Je poměrně široká škála možností, kudy by se tento software mohl v budoucnu ubírat. Jako první by bylo prospěšné rozšířit možnosti zadaných informací o každé položce - např. přidat možnost zadání také ISBN a nahrání obrázku obalu knihy, také by bylo vhodné umožnit do databáze přidávat také webové stránky a jiné než knižní zdroje.

Dále je software připraven na rozšíření o funkci exportu uložených zdrojů do čistě textové podoby, která by se dala vložit jako bibliografie konkrétní práce.

Jinou možností ještě je přidat uživatelské účty, aby mohlo stejnou databázi využívat více osob nezávisle na sobě. S tím se také pojí možnost rozšiřovat databázi na více serverů najednou a optimalizovat úložiště zdrojů podle polohy uživatele, který je nejvíce využívá.

Při případném růstu bude připadat v úvahu řešit problematiku škálování a přechodu na databázový systém, který podporuje distribuci dat na více systémů, protože postgres distribuci nativně nepodporuje.

Pro pokročilé monitorování aplikace v budoucnu lze k aplikaci připojit také syslog server.

# Diskuzní část

## Metodika vývoje softwaru

vizte část *Strategie kolaborace pro efektivní verzování softwaru*

## Nástroje využité během jednotlivých fází vývoje softwaru

Pro tvorbu dokumentace tohoto softwaru jsme využili nástroj Swagger.

K verzování jsme použili Git a Docker.

K samotnému vývoji došlo v jazycích Python a minoritně v SQL a dalších.

Kvůli jednoduchosti jsme si vybrali SQLite databázi, tu ale později nahradila MySQL díky dostupnosti Docker obrazu, a později Postgres.

- K testování jsme využili Swagger, Postman, unittest a vlastní Python skripty.
- Unittest – Unit testing
- Swagger – Integrované testování
- Postman – Funkční testování

Jako webový framework jsme zvolili Flask a Flask RESTx.

Pro komunikaci s databází byl použit jazyk SQL a knihovna SQLAlchemy pro objektový přístup.

K analýze kódu posloužili (za zmíněné je zodpovědný Filip Stehlík):

- Pylint a Ruff k statické analýze kódu a formatingu
- Bandit k testování bezpečnosti
- Isort pro formátování importů
- Black pro automatický formating kódu

Tvorba diagramů proběhla buďto ručně, nebo pomocí nástrojů Visual Paradigm a Draw.io

## Strategie kolaborace pro efektivní verzování softwaru

Hlavním pilířem naší strategie efektivního verzování byla společná domluva na základní kostře aplikace, která měla většinu položek označenou jako “neimplementováno”. Verzování tak bylo vyřešené tím, že bylo zakázané publikovat do git repozitáře změny, které by zamezily v chodu aplikace – jinými slovy na začátku software na vše odpovídal “error: ještě neimplementováno” a postupně se s každou další verzí zlepšoval až do konečné podoby, kdy dokázal na všechny požadavky vypracovat adekvátní odpověď a provést příslušné operace nad databází.

Jak už jsem zmiňoval v popisu vývoje softwaru, jednalo se o feature-driven development, kdy se v Github Projects vypisovaly požadavky na funkcionality, které jednotliví členové týmu označovali jako “rozpracované” či “dokončené” podle toho, jakou měrou přispěli k jejich naplnění. Využili jsme prvky test-driven developmentu, protože jednou z požadovaných funkcionalit byl skript na automatické otestování REST API funkcí a otestování softwaru pomocí nástrojů Postman, Swagger a případně PyTest.

Velkou roli při rozdělování práce sehrála přímá komunikace mezi členy týmu realizovaná prostřednictvím online chatování a videohovorů. Pokud se nám podařilo najít společný čas, aplikovali jsme párové programování.

## Které problémy během vývoje nastaly a jaké bylo jejich řešení

Jedním z problémů bylo nalezení termínů ke společným konzultacím částí projektu, které jsme potřebovali probrat všichni hromadně, aby byly jednotné. Důvodem byly rozdílné denní rozvrhy aktivit každého ze členů týmu. Problém byl nakonec vyřešen nalezením společné chvíle ke komunikaci.

Dalším problémem bylo rozchození téhož Docker image MySQL DB na dvou rozdílných počítačových architekturách. Tento problém byl vyřešen individuální úpravou docker-compose.yml souborů pro každou architekturu a nakonec přechodem na Postgres DB, jejíž Docker obraz funguje v totožné podobě jak na ARM architektuře, tak na x86.