Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Použití GraphQL ve webových aplikacích

Milan Vlasák

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Milan Vlasák

Informatika

Název práce

Použití GraphQL ve webových aplikacích

Název anglicky

Usage of GraphQL in web applications

Cíle práce

Práce se zabývá problematikou API podle specifikace GraphQL ve webových aplikacích. Hlavním cílem práce je vývoj moderní single-page webové aplikace za použití GraphQL. Práce se dále zabývá představením vlastností GraphQL, jeho implementací na straně serveru a klienta.

Metodika

Bakalářská práce sestává ze dvou částí – teoretické a praktické. Metodika zpracování teoretické části spočívá ve studiu odborných informačních zdrojů, zejména samotné specifikace GraphQL a dokumentace jeho implementací.

V praktické části bude navržena a implementována aplikace pro zjednodušení agendy související s vystavováním a přijímáním faktur postavená na serverové a klientské implementaci GraphQL. Backend aplikace bude implementován v jazyce Python za použití webového frameworku Django. Pro vytvoření API dle specifikace GraphQL bude použita knihovna Graphene. Frontend aplikace bude implementována v jazyce TypeScript s využitím některého z dostupných UI frameworků.

Aplikace bude nasazena a otestována. Závěrem budou shrnuty zjištění z její tvorby a zpětné vazby z testování a bude nastíněn případný další možný rozvoj aplikace v budoucnu.

Doporučený rozsah práce

35-40 stran

Klíčová slova

GraphQL, Python, Django, API, WEB, SPA, TypeScript

Doporučené zdroje informací

BUNA, Samer. Graphql in Action. Manning Publications, 2021. ISBN 9781617295683.
Graphene-Python [online]. [cit. 2021-5-21]. Dostupné z: https://graphene-python.org
GraphQL: A query language for your API [online]. [cit. 2021-5-21]. Dostupné z: https://graphql.org
TypeScript: Typed JavaScript at Any Scale [online]. [cit. 2021-5-21]. Dostupné z: https://www.typescriptlang.org

---

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 23. 11. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 06. 01. 2022

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci Použití GraphQL ve webových aplikacích jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 14. března 2023

Poděkování

Rád bych touto cestou poděkoval panu Ing. Jiřímu Brožkovi, Ph.D. za vedení této práce, jeho ochotu, připomínky a čas, který mi při konzultacích práce věnoval.

Dále děkuji rodičům a Daně Brabcové, DiS. za jejich podporu a pomoc s korekturou práce a dceři Štěpánce za každodenní motivaci.

Použití GraphQL ve webových aplikacích

Abstrakt

Tato bakalářská práce se zabývá implementací webového API a single-page webových aplikací pomocí dotazovacího jazyka GraphQL. Teoretická část obsahuje představení dotazovacího jazyka a postupů vývoje API postaveného na GraphQL. Zabývá se jednotlivými aspekty GraphQL na straně serveru, jako jsou datové typy podporované GraphQL, definice grafového schématu nebo mutace dat. Na závěr porovnává použití GraphQL a REST API.

V praktické části je popsán vývoj single-page webové aplikace, která využívá GraphQL pro komunikaci frontendu s backendem. Vývoj je rozdělen na dva dílčí projekty. Backend, který zastřešuje datovou a aplikační vrstvu aplikace. Frontend, který poskytuje vrstvu prezentační. V rámci vývoje backendu je popsán návrh relační databáze, implementací aplikační logiky a mapování relačních dat pro GraphQL API. Projekt frontendu se zabývá návrhem uživatelského rozhraní a přípravou GraphQL dotazů do API.

Klíčová slova GraphQL, Python, Django, API, WEB, SPA, TypeScript, NextJS, Graphene

Usage of GraphQL in web applications

Abstract

This Bachelor thesis is about implementation of web API and single-page web applications using GraphQL query language. The theorethical part contains introduction of the query language and methods of GraphQL based API development. It deals with particular aspects of GraphQL on server side like data types supported by GraphQL, definition of graphql schema or data mutations. In the end it compares usage of GraphQL and REST API.

The practical part describes development of single-page web application which use GraphQL for comunication between frontend and backend. The development is split into two subprojects. Backend which provides a data access layer and an application layer. Frontend which provides a presentation layer. There is an database design, application logic implementation and mapping of relation data to GraphQL API described as a part of the backend development. The frontend part is about user interface design and development of GraphQL API queries.

Keywords GraphQL, Python, Django, API, WEB, SPA, TypeScript, NextJS, Graphene

Obsah

Seznam obrázků

Seznam ukázek kódu

Seznam použitých zkratek

API Application Programming Interface

JSON JavaScript Object Notation

REST Representational State Transfer

URL Uniform Resource Locator

HTTP Hypertext Transfer Protocol

WWW World Wide Web

ACID Atomicity, Consistency, Isolation, Durability

HTML Hypertext Markup Language

SDL Schema Definition Language

OOP Object Oriented Programming

ORM Object Relational Mapping

JWT JSON Web Token

SSH Secure Shell

MVC Model-View-Controller

UWSGI Unified Web Server Gateway Interface

SPA Single-Page Application

DOM Document Object Model

DDOS Distributed Denial-of-Service

1 Úvod

Svět webových technologií se neustále vyvíjí. Největšího rozvoje se aktuálně těší jazyk JavaScript, který se se vznikem platformy Node.js roku 2009 stal plnohodnotným programovacím jazykem, který nemusí běžet pouze ve webovém prohlížeči. Zároveň zažívají velký rozmach webové single-page aplikace, které na rozdíl od klasického webu, kde server vrací celý statický HTML dokument, jsou tvořeny z velké části JavaScriptem, který se stará o dynamické vykreslování většiny obsahu stránky. Slovní spojení single-page v názvu tohoto konceptu pak naznačuje, že aplikace běží v rámci jediné HTML stránky, kterou není potřeba po celou dobu běhu aplikace obnovovat nebo načítat celou znovu. Aplikace je rozdělelna na jednotlivé komponenty, které má za úkol vykreslovat a obsluhovat použitý framework. Protože se stránka neobnovuje, přistupuje single-page aplikace k datům pomocí asynchronních HTTP požadavků, nejčastěji ve formátu JSON prostřednictvím REST API.

V současné době se často stane, že se některý z technologických gigantů rozhodne sdílet své know-how se světem a vytvořit silnou komunitu kolem vlastního nástroje, který vyvinul, osvědčil se mu a jemu samotnému usnadnil spoustu práce. Mezi takové patří například framework Angular vytvořený společností Google nebo knihovna React od Facebooku. Z dílny Facebooku však vyšla ještě jedna velice zajímavá technologie, kterou ostatní i konkurenční firmy začaly rychle a s oblibou zařazovat do svého technologického stacku. Tou technologií je dotazovací jazyk a také specifikace pro tvorbu grafového API jménem GraphQL. API postavené na GraphQL nabízí vývojářům zajímavou alternativu k současně rozšířenějšímu REST API. Spousta technologických firem proto začíná vedle RESTu poskytovat právě alternativní grafové API.

Přístup grafového API umožňuje lepší práci se vztahy jednotlivých datových entit na straně klienta a dotazovací jazyk GraphQL umožňuje klientským aplikacím pracovat s daty, které na rozdíl od REST API nemusí být v pevné, neměnné podobě. Není proto divu, že GraphQL nabývá na čím dál větší popularitě mezi vývojáři. Nejen tyto výhody byly i mou vlastní motivací ke zpracování GraphQL jako bakalářskou práci.

2 Cíle práce a metodika

2.1 Cíle práce

Práce se zabývá problematikou API podle specifikace GraphQL ve webových aplikacích. Hlavním cílem práce je vývoj moderní single-page webové aplikace za použití GraphQL. Práce se dále zabývá představením vlastností GraphQL, jeho implementací na straně serveru a klienta.

2.2 Metodika

Bakalářská práce sestává ze dvou částí – teoretické a praktické. Metodika zpracování teoretické části spočívá ve studiu odborných informačních zdrojů, zejména samotné specifikace GraphQL, dokumentace jeho implementací a dalších tištěných a on-line dostupných zdrojů. Zabývá se popisem definice grafového API na straně serveru pomocí jazyka SDL a věnuje se jednotlivým stavebním prvkům schématu API. Následně se věnuje dotazovacímu jazyku GraphQL, který se na straně klienta využívá ke komunikaci s API. Podobně jako v předchozím případě jsou popsány jednotlivé metody stavby dotazů. Závěr teoretické části se věnuje srovnání specifikací GraphQL a REST a zaměřuje se na jejich společné a rozdílné vlastnosti.

V praktické části bude navržena a implementována aplikace pro zjednodušení agendy související s vystavováním a přijímáním faktur postavená na serverové a klientské implementaci GraphQL. Backend aplikace bude implementován v jazyce Python za použití webového frameworku Django. Pro vytvoření API dle specifikace GraphQL bude použita knihovna Graphene. Frontendová single-page aplikace bude implementována v jazyce TypeScript s využitím některého z dostupných UI frameworků. Praktická část bude obsahovat stručný popis použitých technologií a odůvodnění jejich použití při vývoji. Praktická část popisuje také jednotlivé dotazy a mutace, které bude API aplikace poskytovat.

Aplikace bude nasazena a otestována. Závěrem budou shrnuty zjištění z její tvorby a zpětné vazby z testování a bude nastíněn případný další možný rozvoj aplikace v budoucnu.

3 Teoretická východiska

3.1 Pozadí GraphQL

Než se pustím do samotného popisu specifikace GraphQL, je třeba uvědomit si, že samotná specifikace, ani žádná z jejích implementací, nemohou fungovat sami o sobě a jsou postaveny na určitých teoretických základech a dalších existujících technologií. Popis těchto základů by však vydal na několik samostatných prací. V následující části se proto pokusím zmínit alespoň ty nejdůležitější, se kterými má GraphQL přímou spojistost.

3.1.1 Teorie grafů

Jak už název napovídá, GraphQL přináší do vývoje API[1] přístup podobný grafovým databázím, které vychází z teorie grafů.
V teorii grafů je graf definován následovně:

$$G = (V, E)$$

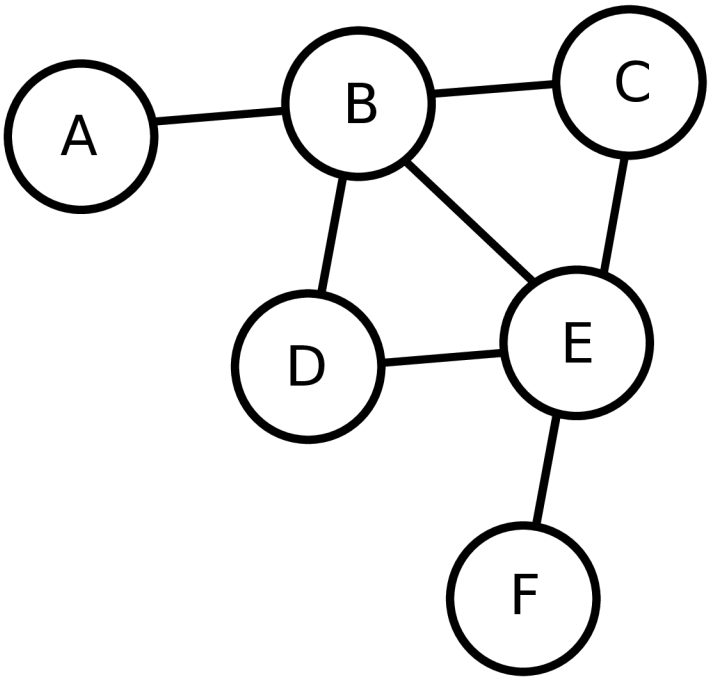
Graf je v teorii grafů uspořádaná dvojice, kde
V je množina vrcholů a
E je množina hran [1]

Grafy mohou být neorientované, kde hrana znamená relaci dvou vrcholů v obou směrech, nebo orientované, kde jedna hrana vede z jednoho vrcholu do druhého, nikoliv však v opačném směru. Hrany mohou být také ohodnocené nějakou sémantickou hodnotou (např. vzdáleností). [1]

Mezi nejčastější úlohy, které řeší teorie grafů, patří například hledání nejkratší cesty nebo cyklů v grafu. [1]

V GraphQL API jsou data uspořádány do grafu, přičemž jednotlivé datové typy (ať už objekty nebo primitivní typy) představují vrcholy grafu. Vazby mezi daty pak odpovídají hranám. [2]

[1] Application Programming Interface



Obrázek 3.1: Příklad grafu (zdroj: autor)

3.1.2 Grafové databáze

Grafové databáze jsou databáze z rodiny NoSQL, které svá data ukládají do grafových struktur místo předem definovaných tabulek, jak je k tomu u relačních databází. Na rozdíl od relačních SQL databází nekladou NoSQL databáze důraz na principy ACID[2], ale na větší flexibilitu a škálovatelnost.

Výhodou grafových databází je například, že místo cizích klíčů jsou entity (vrcholy), které jsou spolu v nějaké relaci, spojeny pomocí hran. Je tedy možné vytvářet komplexní stromové struktury, což je u relačních databází sice možné, ale neefektivní.

Použití grafových databází je vhodné například u sociálních sítí nebo jiných sociálních projektů s velkým množstvím dat. [3]

Jak brzy dále zmíním, GraphQL není grafovou databází, ani ke své funkci grafovou nebo jinou databázi nevyžaduje. Z pohledu klienta se však jako grafová databáze jeví.

[2] Atomicity, Consistency, Isolation, Durability

3.1.3 Protokol HTTP

GraphQL je webová technologie, která ke své komunikaci využívá HTTP[3]. HTTP je protokol aplikační vrstvy TCP/IP modelu používáný pro komunikaci ve WWW[4]. Protokol pracuje na systému požadavků a odpovědí, kde každý požadavek má cílovou URL[5] a je rozdělen na hlavičku s metadaty a případně tělo s obsahem zdroje.[6] [4]

Pro účely této práce není třeba celý protokol popisovat. Pro lepší chápání GraphQL komunikace by bylo však dobré zmínit tři věci: dotazovací metody protokolu, návratové kódy a rozdíl mezi synchronním a asynchronním HTTP požadavkem.

Metody HTTP

Každý požadavek na webový server poslaný pomocí HTTP začíná názvem přístupové metody. Metody se vzájemně liší operací, kterou provedou se zdrojem na dané URL. Pro účely této práce uvedu pouze metody, které využívá architektura REST[7].

GET Výchozí a nejčastěji používaná metoda. Jejím účelem je získání dat zdroje. GET požadavek nemá tělo. Je však možné poslat serveru některé parametry požadavku jako součást URL, což je možné i u ostatních metod.

POST V těle požadavku jsou uživatelská data odesílaná na server. Používá se např. pro odesílání HTML[8] formulářů.

PUT Posílá se obvykle na URL konkrétního objektu. V těle požadavku této metody jsou podobně jako u POSTu data. Metoda slouží k aktualizaci nebo vytvoření zdroje, pokud neexistuje.

DELETE Nemá tělo, slouží ke smazání zdroje na konkrétní URL. [5]

[3] Hypertext Transfer Protocol
[4] World Wide Web
[5] Uniform Resource Locator
[6] Zdroj v kontextu HTTP lze chápat jako zobecnění obsahu dané URL. V praxi se nejčastěji jedná o HTML stránku, ale může jít o soubory libovolného typu nebo o dynamický obsah zcela generovaný webovým serverem.
[7] Representational State Transfer
[8] Hypertext Markup Language

OPTIONS Nemá tělo, dotazuje se na server, jaké metody zdroj na konkrétní URL podporuje. [4]

Návratové kódy HTTP

Důležitou součástí odpovědi HTTP serveru je návratový kód. Ten označuje, zda byl webový požadavek vyřízen úspěšně, nebo jestli došlo k nějakým problémům. Stejně jako v případě metod je návratových kódů definováno ve specifikaci protokolu velké množství a uvedu jen ty nejpoužívanější.

200 OK Požadavek byl úspěšně vyřízen a vrací požadovaná data.

201 Created Byla vytvořena nová entita na požadovaném URL.

301 Moved Permanently Zdroj na daném URL se nechází na jiném URL, které je uvedeno v těle odpovědi. Tento kód se obvykle používá k přesměrování v prohlížeči (např. po úspěšném zpracování dat z formuláře).

400 Bad Request Požadavek není syntakticky správně.

401 Unauthorized Pro vyřízení požadavku je vyžadována autorizace (např. přihlášení uživatele nebo token v hlavičce).

403 Forbidden Požadavek je správně, ale nemůže být vykonán (např. z důvodu, že uživatel nemá potřebné oprávnění).

404 Not Found Zdroj se na daném URL nenachází.

500 Internal Server Error Při vykonávání požadavku došlo k blíže nespecifikované chybě. Obykle se jedná o pád scriptu na serveru z důvodu neošetřené chyby. [4]

3.2 Definice GraphQL

GraphQL je specifikace komunikace mezi klientem a serverem, která definuje dotazovací jazyk pro klienta a vykonání dotazů na serveru pomocí vlastního typovacího systému pro definici dat. GraphQL není samo o sobě závislé na uložišti samotných dat. Data pro API[9] je možné poskytovat z různých zdrojů celé aplikace. Část dat může být například v souborech na disku, část v relační databází, část stažena z externího API. [6]

GraphQL není závislé na konkrétním programovacím jazyce. Jedná se pouze o specifikaci. Referenční implementace backendu je napsána v jazyce JavaScript. Je však možné při vývoji API použít některou nepřeberného množství implementací ve více než dvaceti různých programovacích jazycích nebo vytvořil podle specifikace implementaci vlastní. [6]

Pojem GraphQL však není nutné chápat pouze jako specifikaci. Jak může být z názvu GraphQL zřejmé, písmena QL jsou zkratkou pro Query Language. Často může být pojmem GraphQL myšlen dotazovací jazyk, který tato specifikace definuje. V neposlední řadě lze jako GraphQL rozumět službu, která definuje datové typy a schéma, slouží jako backend pro práci s daty a poskytuje API, se kterým je možné komunikovat výše zmíněným dotazovacím jazykem. [2]

3.3 Historie GraphQL

Technologie GraphQL byla vytvořena vývojáři sociální sítě Facebook roku 2012 během vývoje nových verzí mobilních aplikací pro systémy Adroid a iOS. Cílem bylo vytvořit platformně nezávislý způsob získávání dat z API, který by poskytoval data v jednoduše strojově čitelném formátu, např. JSON[10], a také abstrakci nejen od jednotlivých uložišť, což umožňoval např. už tehdy existující REST. Vývojáři se chtěli také odprostit od nutnosti mít ve zdrojovém kódu URL jednotlivých endpointů a spojování souvisejících dat na úrovni mobilní aplikace. Výsledkem byla první implementace GraphQL, která byla spolu se specifikací zveřejněna pod opensource licencí roku 2015.

[9] Application Programming Interface
[10] JavaScript Object Notation

„ As we transitioned to natively implemented models and views, we found ourselves for the first time needing an API data version of News Feed — which up until that point had only been delivered as HTML. We evaluated our options for delivering News Feed data to our mobile apps, including RESTful server resources and FQL tables (Facebook's SQL-like API). We were frustrated with the differences between the data we wanted to use in our apps and the server queries they required. We don't think of data in terms of resource URLs, secondary keys, or join tables; we think about it in terms of a graph of objects and the models we ultimately use in our apps like NSObjects or JSON. " [11] [7]

GraphQL postupně začínaly zařazovat do svých stacků další velké společnosti, jejichž podnikání je úzce spjato s jejich webovou či mobilní aplikaci. Mezi tyto společnosti patří například Netflix, Shopify, Airbnb nebo například GitHub. Roku 2018 se nadace Linux Foundation rozhodla založit pod svou správou organizaci GraphQL Foundation, která by byla zodpovědná správu a další vývoj GraphQL. [8]

V současné době existuje již šestá revize specifikace GraphQL, která byla vydána 26. 10. 2021. Nová verze specifikace je v přípravě. Část doby během psaní této práce byla v platnosti revize z roku 2018. [9]

[11] Během přechodu k nativně implementovaným modelům a pohledům jsme si uvědomili, že poprvé potřebujeme API verzi dat pro výpis novinek, který do teď existoval pouze ve formě HTML. Vyhodnocovali jsme možnosti poskytování dat pro výpis novinek v našich mobilní aplikacích včetně RESTových zdrojů a FQL tabulek (API facebooku podobné SQL). Byli jsme frustrováni z rozdílů mezi daty, která jsme chtěli v našich aplikacích, a potřebnými požadavky na server. Nepřemýšlíme o datech jako o URL zdrojů, cizích klíčích nebo spojených tabulkách, ale jako o grafu objektů a modelů, co používáme v našich aplikacích ve formátu NSObject nebo JSON.

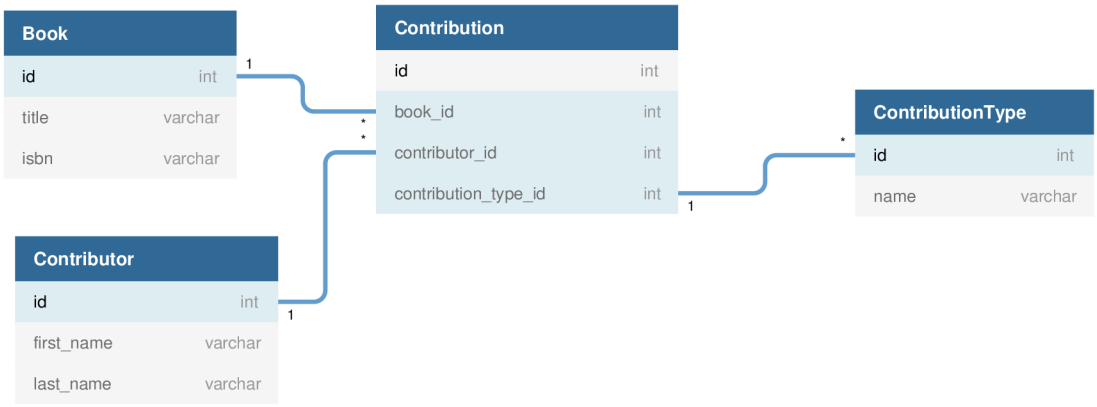
3.4 Stavební prvky GraphQL API

V následující části popíši jednotlivé části definice GraphQL API na straně serveru. Vzhledem k tomu, že ne všechny názvy z terminologie GraphQL mají v češtině jednoznačný nebo dostatečně výstižný překlad, rozhodl jsem se uvádět tyto názvy v angličtině.

3.4.1 Schéma

Každé GraphQL API popisuje tzv. schéma, které obsahuje definice jednotlivých datových typů a mutací. GraphQL pro tyto účely definuje jazyk označovaný jako Schema Definition Language (SDL). [2]

Způsoby zápisu definice schématu API se liší podle implementace GraphQL a také podle možností použitého programovacího jazyka. Většina implementací GraphQL (včetně té referenční) využívá přímo syntaxe SDL, kterou obvykle předá nějaké funkci pro další zpracování v podobě znakové řetězce. Jiné implementace používají formát JSON (nebo vlastní variantu pro zápis na vnořených slovníků), [6] případně zápis založený na deklaraci tříd a jejich atributů dle možností OOP[12] daného jazyka. [10]



Obrázek 3.2: Schéma databáze knih (zdroj: autor)

Pro lepší představu se podívejme na schéma jednoduché databáze knih. (obrázek 3.2) a rozdíl v zápisu mezi čistým SDL (ukázka kódu 3.1) a deklarací v knihovně Graphene

[12] Object Oriented Programming

pro programovací jazyk Python (ukázka kódu 3.2). Ve zbytku kapitoly budu používat pouze SDL.

3.4.2 Typy a pole

Nejmenším dělitelným prvkem každého schématu jsou typy[13]. Ty představují datové typy jednotlivých vrcholů v grafu. Z ukázek zápisu schématu z předchozí podkapitoly je zřejmé, že každý definovaný typ může mít libovolný počet tzv. polí. Každé pole musí mít uvedený svůj datový typ. Ten může být buďto typem definovaným ze schématu, což představuje hranu mezi dvěma vrcholy grafu, nebo některý z vestavěných typů označovaných jako skalární. Další možností je, že pole je výčtového typu. [6]

Typ Query

V ukázkách kódu 3.1 a 3.2 je možné si všimnout, že obě definují typ s názvem Query. Tento speciální typ představuje jakýsi vstupní bod do grafu a formou jeho polí definuje objekty, na které je možné se GraphQL API dotazovat.

Skalární datové typy

GraphQL definuje 6 skalárních typů. Skalární typy jsou nedělitelné a odpovídají obvyklým primitivním datovým typům v programovacích jazycích.
Jednotlivé skalární typy jsou:

ID Slouží jako unikátní identifikátor vrcholu. Je serializován stejně jako typ String a nemusí být nutně čitelný pro člověka.

Int 32-bitové celé číslo se znaménkem.

Float Číslo s pohyblivou řádovou čárkou s dvojitou přesností a znaménkem.

Boolean Binární typ nabývající hodnot „true“ a „false“.

String Znakový řetězec kódovaný v UTF-8.

[13] Pro lepší vnímání rozdílu mezi datovým typem obecně a typem v GraphQL, budu dále používat také označení „typ objektu“ nebo „objektový typ“

```
type Book{
        id: ID!
        title: String!
        isbn: String!
        contributions: [Contribution!]!
}

type Contribution{
        id: ID!
        book: Book!
        contributor: Contributor!
        contributionTypeEnum: ContributionTypeEnum!
}

type Contributor{
        id: ID!
        firstName: String!
        lastName: String!
        contributions: [Contribution!]!
}

enum ContributionTypeEnum{
        AUTHOR
        ILUSTRATOR
        EDITOR
}

input BookInput {
        title: String!
        isbn: String!
}

type Query {
        book(isbn: String!): Book
        books(limit: Int, offset: Int): [Book!]!
        author(firstName: String, lastName: String!): Contributor
}

type schema {
        Query: Query
}
```

Ukázka kódu 3.1: Zápis schématu GraphQL API v SDL (zdroj: autor)

```python
import graphene


class Book(graphene.ObjectType):
    id = graphene.ID()
    title = graphene.String()
    isbn = graphene.String()


class ContributionTypeEnum(graphene.Enum):
    AUTHOR = 'Author'
    ILLUSTRATOR = 'Illustrator'
    EDITOR = 'Editor'


class Contribution(graphene.ObjectType):
    id = graphene.ID()
    book = graphene.Field(Book)
    contributor = graphene.Field('Contributor')
    contribution_type = ContributionTypeEnum


class Contributor(graphene.ObjectType):
    id = graphene.ID()
    first_name = graphene.String()
    last_name = graphene.String()


class BookInput(graphene.InputObjectType):
    title = graphene.String()
    isbn = graphene.String()


class Query(graphene.ObjectType):
    book = graphene.Field(Book, isbn=graphene.String())
    books = graphene.NonNull(graphene.List(
        Book,
        limit=graphene.Int(),
        offset=graphene.Int()
    ))
    author = graphene.Field(
        Contributor,
        first_name=graphene.String(),
        last_name=graphene.String()
    )


schema = graphene.Schema(query=Query)
```

Ukázka kódu 3.2: Zápis schématu v knihovně Python-Graphene (zdroj: autor)

Date Datum serializovaný do unixové časové známky, tj. počtu sekund od 1. 1. 1970. Ne všechny implemetance GraphQL tento typ podporují. [6]

Seznamy

Pole mohou obsahovat také seznam určitého datového typu. Seznamy mají proměnnou délku. V SDL se seznam označuje hranatými závorkami, kde je uveden typ prvků v seznamu. Typ prvků může být objektový nabo skalární. [6]

Povinná pole a prázdná hodnota

Vykřičník v SDL označuje povinné pole. Povinná pole nesmí obsahovat prázdnou hodnotu, která se podobně jako ve většině programovacích jazycích označuje klíčovým slovem „null“.

V případě seznamu (tedy i v ukázce kódu 3.1) je možné v SDL pozorovat vykřičník uvnitř i vně hranatých závorek. V takovém případě vnitřní vykřičník znamená, že prvky seznamu nebudou prázné hodnoty. Vnější vykříční pak říká, že místo celého (byť prázdného) seznamu nesmí být použita prázdná hodnota. [6]

Výčtové typy

Podobně jako některé programovací jazyky, GraphQL podporuje výčtové typy. Definici výčtového typu je uvedena klíčovým slovem „enum“. Výčtový typ se serializuje ve formě znakového řetězce. [6]

3.4.3 Argumenty

Jednotlivá pole mohou mít v závorkách definované argumenty, kterými je možné ovlivnit data, která jsou od API vrácena v odpovědi na dotaz. U jednotlivých skalárních typů se může typicky uvádět jednotka, ve které má být hodnota vrácena. U objektových typů může být často argument typu ID, jenž určuje který konkrétní objekt se má vrátit. V případě seznamů jsou obvykle argumenty spojeny se stránkováním. [6]

3.4.4 Mutace

Mutace v GraphQL slouží ke speciálním dotazům, které požadují změnu dat na serveru. Mutace mívají argumenty pro předání vstupních dat. Pokud má mutace větší počet vstupních parametrů, bývá zvykem obalit argumenty mutace do speciálního vstupního typu, který se definuje podobně jako typ objektu. Jediným rozdílem je, že jeho definice je uvozena klíčovým slovem „input“. [6]

```
input BookInput{
    title: String!
        isbn: String!
    mainAuthorID: ID!
}

type CreateBookPayload{
    ok: Boolean
    output: book
}

mutation {
    createBook(input: BookInput!): CreateBookPayload
}
```

Ukázka kódu 3.3: Definice mutace pro přidání knihy (zdroj: autor)

Pokud bychom uvažovali objektový typ „Book“ z ukázky kódu 3.1, pak mutace pro přidání nové knihy by mohla být definována například jako v ukázce kódu 3.3. Mutace v ukázce vrací typ „BookCreatePayload“, který obsahuje příznak, zda se operace zdařila a také instanci nově vytvořené knihy.

3.4.5 Rozhraní

Pro zobecnění některých objektových typů, je možné definovat rozhraní. Rozhraní je speciální typ, který definuje společná pole, která budou mít všechny objektové typy implementující dané rozhraní. Rozhraní mohou připomínat dědičnost z objektově orientovaných programovacích jazyků. [6]

Použití rozhraní následně umožní dotazovat se v jednom dotazu na více typů současně, viz kapitola 3.5.5. V ukázce kódu 3.4 je uveden příklad použití rozhraní.

```
interface Car {
    id: ID!
    name: String!
    maxSpeed: Int!
}

type FuelCar implements Car {
    tankCapacity: Int!
}

type Electromobile implements Car {
    batteryLifetime: Float!
}
```

Ukázka kódu 3.4: Definice rozhraní a implementujících typů (zdroj: autor)

3.4.6 Union

Při tvorbě API se může stát, že vznikne požadavek na dotaz, který bude vracet dva nebo více typů, které nebudou implementovat žádné rozhraní, ani nebudou mít žádné společné pole. Tuto skutečnost je možné v SDL zapsat pomocí sjednocení typů. [6]

V případě použití sjednocení, je nutné v dotazu definovat dotazovaná pole zvlášť pro každý typ, viz kapitola 3.5.5. Definice sjednocení může vypadat například jako v ukázce kódu 3.5. [10]

```
union Property = Car | Book | RealEstate
```

Ukázka kódu 3.5: Definice sjednocení typů (zdroj: autor)

3.4.7 Resolvery

Nedílnou součástí definice grafového API jsou resolvery, které slouží k získání konkrétních dat. Resolvery nejsou součástí SDL. Jedná se o funkce a metody v konkrétním programovacím jazyce, které mají za úkol vrátit požadovaná data pro dané pole. Ne vždy je potřeba implementovat resolver pro každé pole. Některé implementace GraphQL umí triviálně dosažitelná data (např. na základě názvu pole objektu) získat sami. [10]

3.4.8 Introspekce

Instropekce je prostředek, jakým se klientská aplikace může dotazovat na dostupné dotazy, mutace a typy objektů. GraphQL API je schopné na základě svého schématu generovat dokumentaci ve formátu JSON. Kromě metadat samotných dotazů, mutací a typů obsahuje tato dokumentace datové typy a povinnost výskytu argumentů, výstupních typů a jejich polí. [6]

3.5 Dotazovací jazyk GraphQL

Tato kapitola popisuje jednotlivé stavební prvky dotazů na straně klienta. Na rozdíl od různých variant zápisu schématu GraphQL na straně serveru, které se liší dle implementace, na klientské straně poskytuje GraphQL jednotný dotazovací jazyk, který je pro všechny implementace stejný.

3.5.1 Komunikace se serverem

Ještě před popisem dotazovacího jazyka by bylo dobré uvést způsob, jakým se komunikuje s GraphQL API. GraphQL využívá HTTP protokol a to výhradně jeho metodu POST, přičemž v těle požadavku se nachází JSON obsahující následující pole:

query Řetězec se samotným dotazem.

variables Objekt obsahující hodnoty proměnných použitých v „query“, pokud dotaz proměnné nepoužívá, je pole nepovinné.

operationName Název operace v dotazu, viz kapitola 3.5.3.

Odpovědí API je opět JSON objekt obsahující pole s názvem data. Toto pole obsahuje požadavaná pole v dotazu s hodnotami, které k nim API vrátilo. [6]

3.5.2 Základní dotaz

Grafový dotaz v nejzákladnější formě (viz ukázka kódu 3.6) může lehce připomínat formát JSON, který vynechává uvozovky kolem názvů klíčů, a čárky oddělující jednotlivá pole. Dotaz bývá obalen složenými závorkami a obsahuje výčet typů a polí, která mají být vrácena od API, jenž bude dotaz zpracovávat. Na nejvyšší úrovni se mohou vyskytovat pouze pole z typu Query. Na dalších úrovních je výčet polí, které má API vrátit. [6]

```
{
    authors {
      firstName
      lastName
      contributions {
        book {
          title
          isbn
        }
      }
    }
}
```

Ukázka kódu 3.6: Základní GraphQL dotaz na seznam autorů a jejich knih (zdroj: autor)

Argumenty

Jak již bylo zmíněno v předchozí kapitole, jednotlivá pole objektových typů mohou obsahovat argumenty. Ty se API předávají u názvů daný polí v kulatých závorkách. [6] Předávání argumentů je znázorněno v ukázce kódu 3.7.

```
{
    author(firstName: "Karel" lastName: "Čapek") {
      firstName
      lastName
      contributions {
        book {
          title
          isbn
        }
      }
    }
}
```

Ukázka kódu 3.7: GraphQL dotaz na autora daného jména s použitím argumentů (zdroj: autor)

3.5.3 Pojmenované dotazy a proměnné

Pro lepší přehlednost je možné jednotlivé grafové dotazy pojmenovat. Pojmenovaný dotaz bývá uvozen klíčovým slovem „query“, za kterým následuje jeho název.

Pojmenování dotazů má kromě přehlednosti ještě jednu výhodu, a to je možnost použití proměnných namísto konstant v argumentech dotazu. Podobně jako se v jednotlivých programovacích jazycích nedoporučuje při komunikaci s databází uvádět hodnoty napřímo v SQL dotazech, ale předávat je jako zvláštní parametry, bývá v GraphQL upřednostňováno používání proměnných. Na rozdíl od SQL neplyne toto doporučení z bezpečnosti, ale z možnosti znovupoužitelnosti stejného dotazu s jinými parametry.

Proměnné se uvádějí do kulatých závorek za název dotazu. Jejich název začíná znakem „$“ a podobně jako v případě definice argumentů na straně serveru, se u nich definuje datový typ. Typ proměnné může být buďto skalár, výčet nebo vstupní objektový typ.

Pojmenovaný dotaz s proměnnými je možné si prohlédnout v ukázce kódu 3.8. [6]

```
query authorsBooks($firstName: String, $lastName: String!) {
  author(firstName: $firstName, lastName: $lastName) {
    firstName
    lastName
    contributions {
      book {
        title
        isbn
      }
    }
  }
}
```

Ukázka kódu 3.8: Pojmenovaný GraphQL dotaz na autora s použitím proměnných (zdroj: autor)

OperationName

Další použití názvu dotazu je v těle HTTP požadavku. V poli query je možné poslat více dotazů současně a název dotazu, který bude vyhodnocen serverem, je uveden v poli „operationName“. Pokud je jednoznačné, který dotaz se má vykonat, je pole „operationName“ nepovinné. [6]

Direktivy

Pomocí proměnných je možné také podmíněně dotazovat nebo odstranit z dotazu libovolná pole. Specifikace GraphQL definuje dvě tzv. direktivy, které toto umožňují.

V ukázce kódu 3.9 je použitá direktiva „include“, která zahrne pole do dotazu, jestliže je proměnná v argumentu rovna hodnotě „TRUE“. Druhá direktiva „skip“ naopak pole v případě hodnoty „TRUE“ z dotazu odstraňuje. Proměnné v argumentech obou direktiv musí být typu „Boolean“. [6]

```
query Authors($includeFirstName: Boolean!) {
    authors {
        id
        firstName @include(if: $includeFirstName)
        lastName
    }
}
```

Ukázka kódu 3.9: Dotaz s direktivou „include“ (zdroj: autor)

3.5.4 Mutace

Protože mutace způsobují změnu dat na serveru, jsou vždy uvozeny slovem „mutation“. Jinak není možné mutaci zavolat. Zavolání mutace pomocí dotazovacího jazyka GraphQL se jinak nikterak neliší od ostatních dotazů, viz ukázka kódu 3.10. [6]

```
mutation CreateBook($title: String! $isbn: String! $mainAuthorID: ID!) {
    createBook(
        input: {
            title: $title isbn: $isbn mainAuthorID: $mainAuthorID
        }
    ) {
        ok
        output {
            id
            isbn
            title
        }
    }
}
```

Ukázka kódu 3.10: Mutace přidávající novou knihu (zdroj: autor)

3.5.5 Fragmenty

Vypisování dotazovaných polí se může stát rychle repetitivní a může vést ke snížené přehlednosti v zápisu dotazů. Jazyk GraphQL proto umožňuje definovat z objektového typu tzv. fragment, který obsahuje podmnožinu polí z daného typu. Použití fragmentu je k nahlédnutí v ukázce kódu 3.11.

Pole ve fragmentu mohou stejně jako jiná pole přijímat argumenty. Hodnoty proměnných v datazu se zpropagují i do fragmentu. Vzhledem k tomu, že fragmenty jsou na rozdíl od objektových typů, definované klientem, není podoba fragmentu nijak omezená kromě toho, že může obsahovat pouze pole daného typu. Fragmenty mohou být znovupoužitelné napříč celou klientskou aplikací. [6]

```
fragment BookFragment on Book {
    title
    isbn
}

query AllAuthors {
    authors {
      firstName
      lastName
      contributions {
        book {
          ...BookFragment
        }
      }
    }
}
```

Ukázka kódu 3.11: Dotaz s použitím fragmentu (zdroj: autor)

Inline fragmenty

Velmi podobným způsobem se používají tzv. inline fragmenty, které slouží k definování dotazovaných polí v případě, že výsledkem dotazu může být více typů a to jak v rámci rozhraní, tak v případě sjednocených typů. Jak je vidět v ukázce kódu 3.12, společné pole je možné v dotazu uvést jako obvykle a pomocí inline fragmentů dotazují pole, která jsou specifická pro každý typ zvlášť. [6]

```
query Property {
    properties {
        __typename
        price
        ...on Car {
            maxSpeed
        }
        ...on Book {
            pages
        }
        ...on realEstate {
            area
        }
    }
}
```

Ukázka kódu 3.12: Dotaz s použitím inline fragmentů (zdroj: autor)

3.5.6 Aliasy

Může se stát, že je potřeba dotazovat stejné pole s různými argumenty v jednom dotazu. Výsledek by v takovém případě obsahoval dva shodné klíče a vrácený JSON by nebyl validní. Je ale možné shodná pole opatřit aliasy, které se použijí k jejich jednoznačnému rozlišení. [6] Viz ukázka kódu 3.13.

```
query TwoAuthors {
    capek: author(lastName: "Čapek") {
        firstName
    }
    vrchlicky: author(lastName: "Vrchlický") {
        firstName
    }
}
```

Ukázka kódu 3.13: Dotaz s použitím aliasů (zdroj: autor)

3.6 Srovnání s REST API

Za nejčastějšího konkurenta GraphQL se při stavbě API nad HTTP obvykle považuje REST. Obě tyto varianty komunikují na bázi modelu Klient - Server, což je dáno právě protokolem HTTP. Obě technologie umožňují požadovat data ze serveru i odesílat data na server. Komunikace je vždy inicializována klientem. REST API v nejčastější implementaci podobně jako GraphQL vrací data serializovaná ve formátu JSON.

REST využívá většinu metod definovaných HTTP protokolem (GET, POST, PUT, PATCH, DELETE), zatímco GraphQL používá jedno univerzální dotazovací rozhraní, na které jsou posílány dotazy pouze metodou POST.

REST vyžaduje, aby klienti stahovali všechna data, která jednotlivá URL poskytují, zatímco GraphQL umožňuje specifikovat pole, která mají být vrácena v odpovědi. Je tedy možné získat veškerá potřebná data v jednom jediném dotazu. Na druhou stranu, ani v případě GraphQL nebývá zvykem dotazovat se na všechna potřebná data najednou. Běžná praxe např. u SPA bývá taková, že jednotlivé komponenty aplikace volají jednotlivé dotazy podle toho, jaká data potřebují. Odstranění duplicity dotazů na straně klienta může řešit cachovací vrstva. Obvykle je také potřeba na úrovni GraphQL řešit omezení velikosti dotazů, protože GraphQL může být v opačném náchylný na útoky typu DDOS[14].

Na rozdíl od RESTu, implementace GraphQL mají vestavěnou silnou typovou kontrolu a validaci vstupních i výstupních dat. V případě REST API je potřeba řešit validaci buďto na úrovni konkrétní implementace nebo programovacího jazyka. [2, 11, 12]

[14] Distributed Denial-of-Service

4 Vlastní práce

Tato kapitola se zabývá praktickou částí bakalářské práce. Praktickou část tvoří vývoj single-page webové aplikace za použití GraphQL. V kapitole jsou popsány základní funkce aplikace, zvolené technologie a další rozhodnutí, které bylo potřeba v průběhu vývoje udělat.

4.1 Zaměření aplikace

Pro praktickou část této práce jsem si zvolil implementaci systému pro vystavování a přijímání faktur zaměřenou na potřeby malých živnostníků. Kromě samotné fakturační agendy jsem chtěl, aby systém poskytoval adresář kontaktů pro jednotlivé uživatele, roční přehledy s vypočtenými hodnotami pro účely přiznání daně z příjmů z nezávislé činnosti a integraci s aplikací Toggl. Většinu funkcionalit jsem navrhoval dle svých potřeb, protože aplikaci sám využívám.

4.1.1 Cílová skupina uživatelů

Cílová skupina uživatelů jsou již zmínění drobní živnostníci, kterým by aplikace měla ušetřit čas s fakturací a správou nákladů. Typickým uživatelem systému je osoba samostatně výdělečně činná, která ročně vystavuje menší množství faktur a pracuje na zakázkách pro menší množství klientů. Případně živnostník, který svým podnikáním nevykonává hlavní činnost.

Naopak aplikace pravděpodobně nebude dostatečná pro právnické osoby a živnostníky s větším obratem (např. plátci DPH). U uživatelů takových subjektů bude však pravděpodobnější, že budou vyhledávat pokročilejší účetní software.

4.1.2 Existující řešení

Existuje mnoho různých řešení pro agendu vystavování a přijímání faktur, která se liší svou funkčností, cenou a přístupností. Jeden z nejznámějších desktopových účetních systémů

je Pohoda, který poskytuje širokou škálu funkcí, ale může být pro některé uživatele příliš složitý. Existuje také verze, která je dostupná zdarma, ale s omezenými funkcemi. [13]

Webová aplikace Fakturoid je jinou alternativou, která poskytuje základní funkcionalitu zdarma. Uživatelé mohou vystavovat a přijímat faktury, spravovat kontakty a vést účetnictví. Nicméně, pro rozšíření funkcí, jako např. počet kontaktů nebo možnost exportu do účetních programů, je nutné zaplatit měsíční nebo roční poplatky. [14]

4.1.3 Funkce aplikace

Přihlášení uživatele

Uživatel se bude přihlašovat klasicky pomocí uživatelského jména a hesla. Bez přihlášování nebude možné přistupovat ke zdrojům, které poskytuje GraphQL API.

V této verzi nebude implementována registrace uživatelů a uživatelské účty budou vytvářeny prostřednictvím administračního rozhraní správcem aplikace na vyžádání.

Nastavení uživatele

Uživatel bude mít možnost změnit své heslo, křestní jméno, příjmení a e-mail. Dále také aktivovat nebo deaktivovat upozornění na e-mail a nastavení integrace se službou Toggl, viz níže.

Uživatel bude mít také možnost vyplnit údaje o své vlastní firmě, která bude figurovat na jeho fakturách jako vydavatel, či odběratel podle typu faktury. Bez vyplněných údajů o firmě nebude možné vystavovat faktury.

Adresáře firem

Systém bude poskytovat adresář firem. Ten bude individuální pro každého uživatele. Faktury půjdou adresovat pouze uloženým firmám, kde firma bude v roli druhé strany opět jako odběratel, nebo dodavatel podle typu faktury.

Fakturace

Systém bude umožňovat vystavování faktur, které budou moci být dvojího typu - vydané, nebo přijaté. Faktury budou moci mít několik stavů v rámci svého životního cyklu. Faktura bude moci být ve stavu Nová, Předaná, Opožděná a Uhrazená.

Aplikace po povolení uživatelem umožní odesílat upozornění na e-mail ohledně blížící se splatnosti faktur nebo opoždění platby v případě, že faktura nebude do data splatnosti přepnuta do stavu Uhrazená. Důležitou funkcí bude také možnost stažení evidovaných faktur ve formátu PDF.

Aplikace bude umožňovat vytvoření šablon faktur, např. pro případ, že uživatel fakturuje často stejnému odběrateli. Ze šablony bude možné vytvořit regulérní fakturu a to jak vystavenou, tak přijatou.

Roční přehledy

Na základě evidovaných faktur bude aplikace generovat roční přehledy příjmů a výdajů. Budou také poskytovat výpočty použitelné v daňovém přiznání fyzických osob, např. náklady vypočítané procentem z příjmů. Nebude však zohledňovat nově zavedenou paušální daň nebo příjmy, které nejsou evidovány v systému.

Integrace se službou Toggl

Toggl je webová aplikace, která slouží ke stopování času stráveném na jednotlivých úkolech. Úkoly je možné sdružovat do projektů. Integrace s Togglem bude spočívat ve stahování měsíčních reportů z REST API, které bude aplikace následně vykreslovat v podobě kalendáře s denním přehledem odpracovaných hodin a tabulkou, kde budou odpracované hodiny rozděleny po projektech za celý měsíc.

4.1.4 Architektura aplikace

Aplikace je striktně rozdělena na backend a frontend. Backend aplikace obsahuje veškerou aplikační logiku a především zprostředkuje přístup k datům uživatele, které jsou uloženy v relační databázi. Backend pro práci s databází využívá ORM a veškeré tabulky

v databázi definuje pomocí datových modelů. Data pro frontend poskytuje prostřednictvím GraphQL API.

Frontend je SPA[1], tedy aplikace v podobě jediné webové stránky, která je podle uživatelských vstupů překreslována JavaScriptem. Frontend se stará o vzhled aplikace a funkčnost uživatelského rozhraní. Dále definuje dotazy v GraphQL, pomocí kterých si vyžaduje data z API backendu.

4.1.5 Původní verze aplikace

První verzi aplikace jsem vytvořil v průběhu roku 2018 čistě pro své účely. Nešlo však o SPA, která by využívala GraphQL API. Jednalo se o webovou aplikaci postavenou na modelu MVC ve frameworku Django bez dalších rozšíření. Aplikace fungovala na již zastaralé verzi Pythonu a Djanga.

Z původní aplikace byl převzat především datový model, který je popsán v následující kapitole 4.3, administrace generovaná frameworkem a část aplikační logiky, která se přesunula z jednotlivých pohledů a formulářů Django frameworku do dotazů a mutací v GraphQL. Původní pohledy byly v kódu prozatím ponechány, ačkoliv se již aktivně nevyužívají, a větší část logiky musela být přepsána pro vhodnější použití s GraphQL.

[1] Single-Page Application

4.2 Výběr technologií

4.2.1 Backend

Python

Backend aplikace je implementován v programovacím jazyce Python. Python je vysokoúrovňový, interpretovaný, dynamicky typovaný jazyk, který umožňuje jak procedurální, tak i objektově orientované programování. Má rozsáhlou knihovnu standardních modulů, které poskytují širokou škálu funkcí a silnou komunitu vývojářů, kteří vytváří další knihovny. Pro tvorbu aplikace je použita verze 3.8. [15]

Django

Django je webový framework napsaný v jazyce Python. Kromě ORM[2] pro práci s relační databází, je dodáván s vlastním šablonovým systémem, nebo např. vrstvou pro key-value cache. Mezi jeho další výhody patří možnost vygenerování formulářů a Administračního rozhraní na základě datových modelových tříd. Django patří mezi nejpopulárnější frameworky v Pythonu. Django v základu funguje podle modelu MVC[3]. Pro jiné přístupy (např. REST nebo GraphQL) jsou k dispozici knihovny rozšiřující tento framework.

Schéma hlavní relační databáze aplikace je spravována migracemi, které jsou frameworkem generovány na základě datových modelů.

Aplikace využívá framework Django ve verzi 4.1. [16]

Graphene

Graphene je framework pro tvorbu GraphQL API v jazyce Python. Tento framework jsem zvolil na základě přechozích zkušeností a pro jeho existující integraci do Djanga, která umožňuje vytvářet datové typy z datových modelů ORM, podobně jako samotné django umí z modelů vytvářet formuláře.

[2] Object Relational Mapping
[3] Model-View-Controller

Integrace pro Django dále umožňuje použití Django formulářů nebo serializérům z knihovny Django Rest Framework pro validaci dat v mutacích. Tuto možnost jsem v aplikaci nevyužil, protože většina dat nevyžaduje větší validaci mimo datový typ. U větších aplikací jde však o velmi znatelné usnadnění práce.

Vývoj této implementace GraphQL se během práce na aplikaci na delší dobu zastavil a nebyla tak dostupná integrace kompatibilní s novějšími verzemi Djanga. Během této doby jsme zvažoval, zda refaktorizovat API do jiného frameworku s názvem Strawberry. Přibližně v polovině roku 2022 však vyšla nová verze Graphenu a rozhodl jsem se proto u něj zůstat.

Podobně jako jiné implementace GraphQL i Graphene nabízí webové rozhraní GraphiQL, které umožňuje testovat dotazy z webového prohlížeče. [10]

uWSGI

UWSGI[4] funguje jako most mezi webovým serverem a aplikací naprogramovanou v jazyce Python. Nabízí velké množství konfigurace škálovatelnosti na úrovni procesů a vláken. uWSGI se často používá pro provoz webových aplikací naprogramovaných v Pythonu, jako je Django nebo Flask. Podporuje ale i aplikace v jiných jazycích.

V rámci infrastruktury aplikace představuje UWSGI proces s webovým workerem, který přebírá HTTP požadavky od webového serveru a předává frameworku Django ke zpracování. [17]

Celery

Celery je implementace dávkového zpracování úloh ve frontě podle vzoru producent-konzument. Umožňuje zpracování úloh na pozadí, takže webový worker nemusí v rámci HTTP requestů vykonávat výpočetně náročné operace, ale jen zařadit novou úlohu do fronty a podstatně dříve vrátit uživateli odpověď.

Konzumentem úloh je proces (nebo více procesů), který se nazývá Celery worker a může mít volitelně více vláken, které jednotlivé úlohy vybírají z fronty. Celery je možné nakonfigurovat tak, že určité úlohy mají vyšší prioritu než jiné. Také je možné použití více oddělených front a jednotlivým workerům určit, které typy úloh mají vykonávat.

[4]Unified Web Server Gateway Interface

Celery dále nabízí další typ procesu s názvem Celery beat, který slouží jako plánovač periodických úloh.

Producentem může být teoreticky jakýkoliv jiný proces, který má možnost zapisovat do fronty v předepsaném formátu. V aplikaci může být do fronty zařazena úloha webovým workerem, Celery workerem z jiné právě vykonávané úlohy, nebo Celery beatem.

Největší výhodou je snadná integrace Celery do frameworku Django. I Celery je napsána v Pythonu a umí číst konfiguraci Djanga, takže jak webová aplikace v Djangu, tak i jednotlivé úlohy pro Celery mohou sdílet společný kód. [18]

JWT

Pro autentizaci uživatelů komunikujících přes API jsem se rozhodl použít metodu JWT[5] tokenů. Protože jde o obsáhlejší téma, rozhodl jsem se autentizaci věnovat samostatnou kapitolu 4.4.

PostgreSQL

PostgreSQL je robustní relační databáze dostupná jako opensource. Je vhodná pro použití nejen ve webových aplikacích. Podobně jako jiné databázové systémy podporuje atomické transakce, integritu dat pomocí cizích klíčů a jiné funkce.

PostgreSQL má také silnou podporu pro rozšíření, která umožňuje vývojářům rozšiřovat funkce databáze pomocí jazyků jako je C nebo PL/pgSQL. [19]

Redis

Jako databázi pro key-value cache a také message brokera pro Celery jsem zvolil NoSQL databázi Redis. V této databázi jsou uloženy relace uživatelů, které nevyužívají JWT (typicky pro přístup do Django administrace). Dále pak fronta pro Celery. [20]

Supervisor

Je správce procesů, který je vhodný pro spouštění Python aplikací jako služby. Umožňuje automatické restartování procesů, když dojde k chybě. Také umožňuje jednotlivé spravované

[5] JSON Web Token

procesy sdružovat do skupin a provádět různé oprace (např. start, stop, restart) nad celou skupinou.

Jako jeho subproces je definován webový worker, celery worker a celery beat. [21]

Nginx

Nginx je webový server, který může být nastavený také jako reverzní proxy nebo load balancer. [22]

Pro účely aplikace slouží primárně jako reverzní proxy před UWSGI webovým workerem, kterému předává HTTP požadavky skrze unix socket.

Zajišťuje také šifrovanou komunikaci mezi serverem a klienty přes protokol HTTPS. SSL certifikáty k tomu použité jsou vydány zdarma neziskovou certifikační autoritou Let's Encrypt. [23]

4.2.2 Frontend

NodeJS

NodeJS je běhové prostředí, které umožňuje spouštění kódu v jazyce JavaScript mimo webový prohlížeč. Při použití moderních frontendových frameworků je NodeJS základní požadavek pro sestavení produkčního balíčku webové aplikace. [24]

TypeScript

Pro vývoj frontendu jsem zvolil programovací jazyk TypeScript. Ten na rozdíl od JavaScriptu podporuje statickou typovou kontrolu a umožňuje tak vyhnout se chybám způsobeným častým implicitním přetypováním, které JavaScript provádí. Výstupem TypeScriptu je kód v JavaScriptu, který je spustitelný v prohlížeči. Výstupní kód obsahuje různé kontroly na základě statických typů z TypeScriptu a výše zmíněné problémy eliminuje. [25]

NextJS

Pro tvorbu uživatelského rozhraní je použit framework NextJS, který vychází z rozšířené knihovny React. Velkou předností je například routing single-page aplikace na základě

adresářové struktury projektu. Dále sám provádí různé SEO optimalizace, umožňuje moderní přístupy pro tvorbu webových aplikací jako je server-side rendering.

Z knihovny React si pak propůjčuje rozdělení celého uživatelského rozhraní na drobné komponenty, JSX/TSX syntax a manegement stavu aplikace a jednotlivých komponent. [26, 27]

Apollo Client

Apollo Client je knihovna pro JavaScript a TypeScript, která umožňuje konzumovat GraphQL API. Je navržena tak, aby pracovala bez problémů s Reactem, resp. NextJS, a může být také použita s dalšími JavaScriptovými frameworky a knihovnami.

Kromě samotných HTTP požadavků řeší také zpracování chyb a cachovací vrstvu, takže nemusí volat identické dotazy vícekrát po sobě a ušetřit tak počet HTTP požadavků a síťový provoz. [28]

GraphQL Codegen

Úkolem Codegen je z manuálně napsaných GraphQL dotazů vygenerovat kód pro provedení dotazu a získání výsledků a chyb pomocí Apollo Clienta. Codegen se API prostřednictvím introspekce dotáže na jednotlivé dotazy a mutace, které nalezne ve zdrojovém kódu aplikace a vyžádá si z introspekce jejich datové typy a další meta data.

Výsledný kód zahrnuje definici typů pro TypeScript a několik funkcí, které je možné v aplikaci použít. Vygenerované funkce jsou také k dispozici v podobě hooků pro knihovnu React. Integrace s Reactem, resp. NextJS je proto velice snadná. [29]

Vercel

Vercel je moderní platforma, která funguje jako webhosting pro statické weby. Ve službě Vercel je nasazen frontend aplikace. Jeho výhodou je možnost automatického nasazování bez nutnosti konfigurace NodeJS serveru. Další výhodou je optimalizace pro běh webových aplikací v NextJS, protože NextJS je jedním z produktů Vercelu. [30]

Tailwind

Tailwind je CSS framework založený na principu utility-first. Na rozdíl od nejznámějších frameworků jako např. Bootstrap, Tailwind neposkytuje hotové komponenty, které je obvykle možné přidat do stránky pomocí jedné nebo několika mála tříd. Naopak přístup utility-first poskytuje velké množství drobných tříd, které obvykle upravují hodnotu jediného atributu. [31]

4.2.3 Ostatní

Git

Git je verzovací nástroj, který vznikl za účelem verzování zdrojových kódů linuxového kernelu. V současné době se jedná o jeden z nejrošířenějších verzovacích systémů. Výhodou Gitu je jeho decentralizovanost. [32]

Ansible

Pro automatizaci nasazovacího procesu je použit nástroj Ansible. Jeho hlavní výhodou je, že pro komunikaci s cílovými servery nepotřebuje žádného tzv. agenta, který musí na serveru neustále běžet. Naopak využívá protokol SSH[6] a jednotlivé operace volá jako subprocesy vzdáleného shellu. [33]

Dbdiagram

Je webová aplikace pro tvorbu ER diagramů. Na rozdíl od jiných nástrojů pro tvorbu diagramů se výsledný diagram generuje na základě psaného kódu. Diagram je tak možné jednoduše verzovat a sdílet. Tento nástroj jsem použil pro vytvoření diagramu databázového schématu. [34]

Figma

Je designerský nástroj vhodný pro tvorbu wireframů a návrhů designu webových stránek a aplikací. V tomto nástroji jsem vytvářel wireframe aplikace. [35]

[6] Secure Shell

4.3 Návrh databáze

Na obrázku 4.1[7] je zobrazeno schéma relační databáze aplikace. Při návrhu databáze jsem použil standardní postupy pro návrh databází a snažil se, aby tabulky splňovaly druhou normální formu. Dále uvedu popis jednotlivých tabulek a zdůvodnění některých rozhodnutí při jejich návrhu.

Struktura databáze je spravována ORM a databázovými migracemi frameworku Django. To znamená, že namísto SQL scriptu, který by sloužil k vytvoření struktury databáze, jsou jednotlivé tabulky vytvářeny podle datových tříd (modelů) ve zdrojovém kód backendu.

User Reprezentuje uživatele systému. Obsahuje přihlašovací a základní kontaktní údaje. Dále některá pole potřebná k funkci frameworku Django a uživatelská nastavení jako zobrazování DIČ na fakturách nebo nastavení integrace s Togglem.

BaseCompany Je základní tabulkou, která drží společná data pro tabulky Company a OwnCompany. Při návrhu aplikace bylo potřeba logicky oddělit adresář jednotlivých uživatelů a informace o jejich vlastních firmách. Stejně jako ostatní tabulky, je tato tabulka vygenerována frameworkem na základě modelových tříd. Společná data tabulek Company a OwnCompany jsou v kódu reprezentována dědičnostní tříd.

Company Company je tabulka reprezentující firmy v adresářích uživatelů. Tabulka je připravena pro přidání dalších polí, která nebudou společná s OwnCompany. V současné době je v tabulce pouze cizí klíče na tabulku BaseCompany a uživatele.

OwnCompany Reprezentuje firmu, která patří uživateli a je proto vždy zobrazena na fakturách. Podobně jako v předchozím případě tabulka obsahuje cizí klíče na tabulky na BaseCompany a User, a mohou do ní být přidány další pole dle potřeby. Uživatel může mít pouze jednu vlastní firmu.

Address Reprezentuje adresu firmy. Aktuálně má tato tabulka vazbu na BaseCompany s četnostní 1:1. Rozšíření aplikace, které by umožňovalo, aby firma mohla mít více adres, by spočívalo pouze ve změně četnosti vazby na 1:N.

[7] Obrázek je k dospozici ve větším rozlišení v elektronické příloze.

Invoice Představuje v databázi vystavenou fakturu. Faktura může být vydaná, nebo přijatá. Tato vlastnost je určena polem invoice_direction. Tabulka má vazbu na uživatele, který fakturu vydal nebo přijal. OwnCompany tohoho uživatele bude použita jako dodavatel, či odběratel na faktuře. Druhá strana na faktuře je reprezentována vazbou na Company.

V rámci faktury je třeba zajistit, uchování dat firem v okamžiku, kde byla faktura vystavena. K tomu slouží pole primary_company_data a secondary_company_data, kde jsou tyto informace serializovány ve formátu JSON.

InvoiceTemplate Je tabulka s šablonami faktur. Protože je většina sloupců této tabulky shodná s tabulkou Invoice, je na úrovni datových modelů deklarována třída „BaseInvoice“, ze které dědí modely „Invoice“ a „InvoiceTemplate“. Na rozdíl od BaseCompany je však tato třída označena jako abstraktní, takže ORM nevygeneruje tabulku se společnými sloupci, ale dvě nezávislé tabulky, které mají všechny společné sloupce jako své vlastní. K tomuto postupu jsem se rozhodl, protože mezi fakturou a šablonou faktury není žádná semantická souvislost a i přes společná pole se jinak jedná o oddělené entity.

InvoiceItem V databázi reprezentuje jednu položku na faktuře.

4.4 Autentizace uživatelů

Autentizace uživatelů při komunikaci s API je řešena s pomocí JWT tokenů. JWT[8] je moderní standardizovaný formát pro ověřování autentizace a autorizace ve webových službách a aplikacích.

Skládá ze tří částí: hlavičky, těla a podpisu. Hlavička obsahuje informace o typu tokenu a algoritmu, který byl použit k podpisu. Tělo obsahuje informace o uživateli, který má být autentizován, jako jsou například ID uživatele nebo jeho přístupová oprávnění. Podpis je vytvořen pomocí hashovacího algoritmu, který je uveden v hlavičce a je vytvořen z hlavičky, těla a tajného klíče. První dvě části jsou objekty ve formátu JSON. Jednotlivé části tokenu jsou od sebe oddělené tečkou a jsou zakódované pomocí kódování BASE64.

Existuje více dostupných algoritmů obvykle vždy v 256, 384 a 512 bitových variantách. K dispozici jsou jak symetrické, které pro podpis i ověření podpisu používají stejný klíč

[8] JSON Web Token

(HMAC-SHA), tak i asymetrické, kde služba, která tokeny vydává, používá privátní klíč a k verifikaci klíče slouží veřejný klíč, který může být sdílený mezi více službami, které tokeny akceptují (RSA-SHA, ECDSA-SHA).

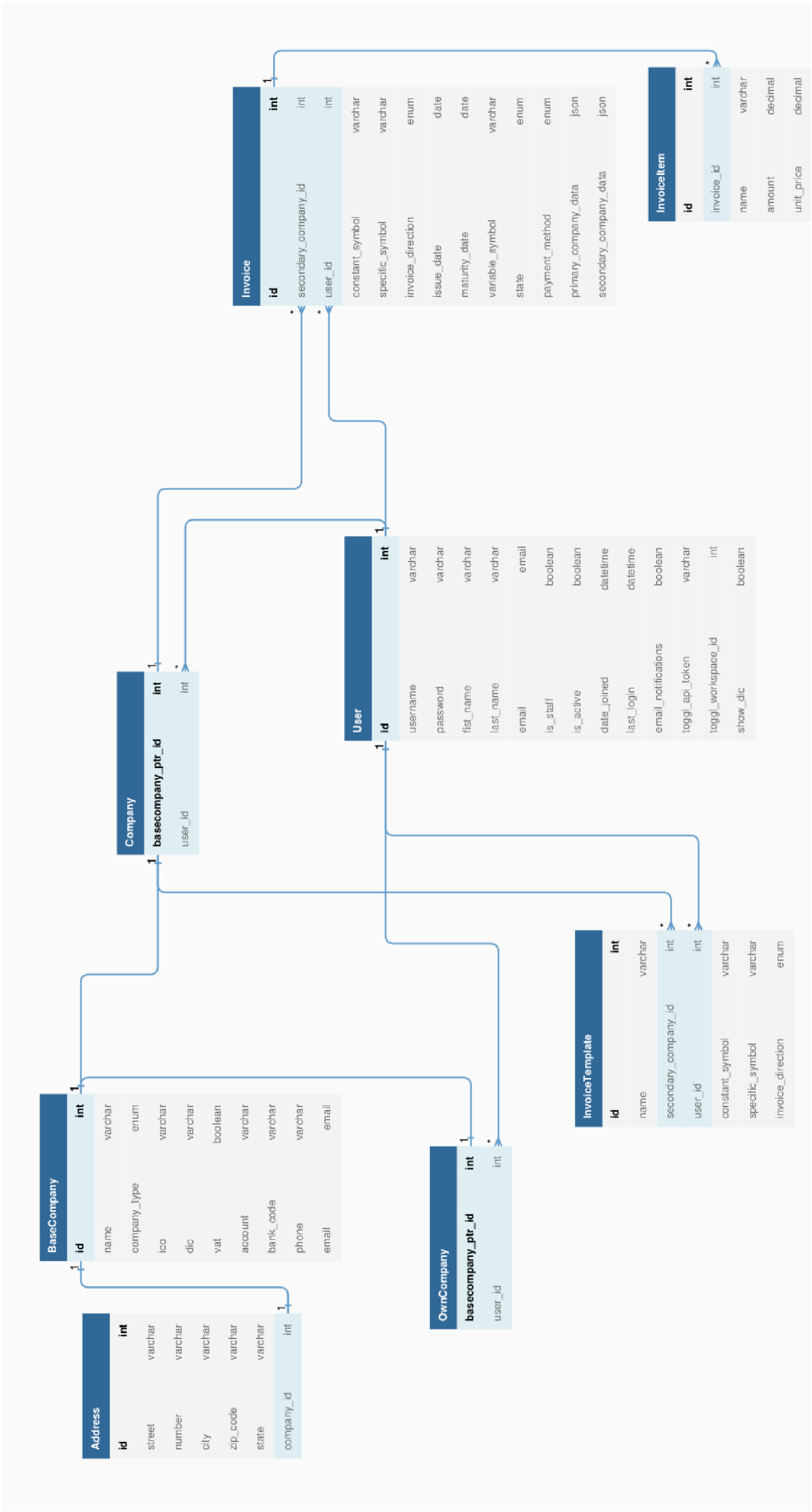
Jednotlivá pole v JSON objektu uvnitř JWT se označuje slovem „claim“. Pojmenování těchto polí může být libovolné. Existuje však seznam spravovaný organizací IANA, který obsahuje standardizovaná pole dle [36] a další ustálené claimy. [37]

JWT je stateless, což znamená, že server si nepamatuje stav klienta mezi jednotlivými HTTP požadavky. Informace o uživateli jsou proto uložené v JWT, které se předávají mezi klientem a serverem při každém požadavku. Tím se snižuje náročnost na paměť serveru a zvyšuje se rychlost a výkon aplikace. JWT se často používají pro autentizaci a autorizaci v SPA. JWT je také často používán pro autentizaci v různých cloudových službách. [38, 36]

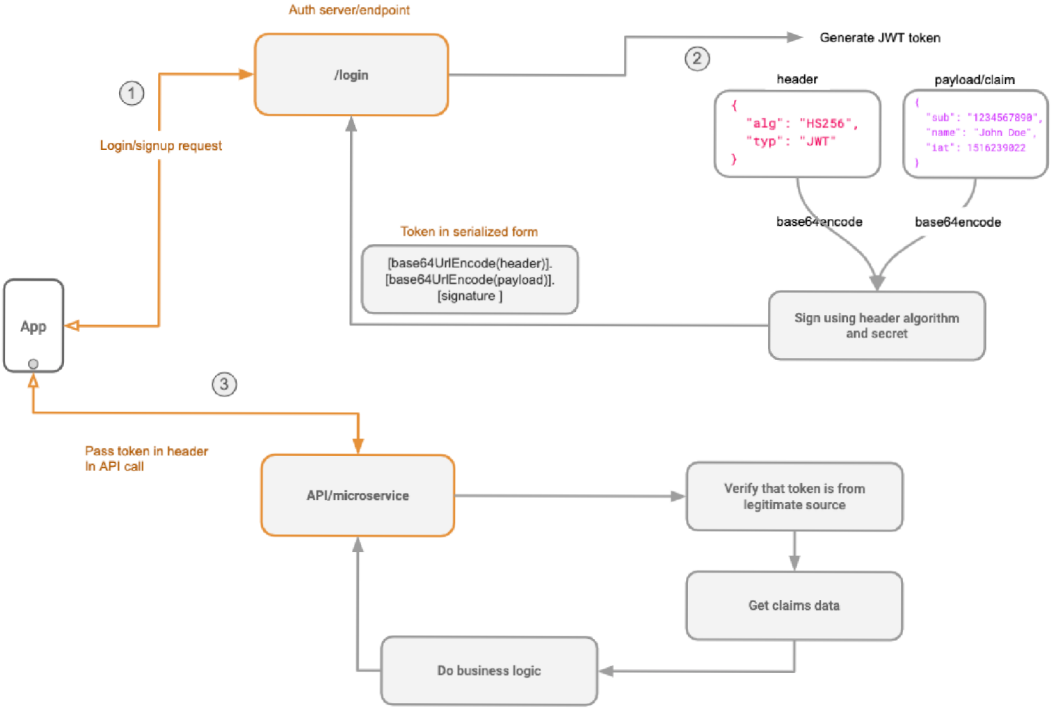
Jedním z častých případů použití JWT v single-page aplikací zhruba odpovídá standardu OAuth 2.0, který pracuje se dvěma tokeny. Access token, který je ve formátu JWT a je vydán s kratší platností (obvykle 15 až 30 minut), je serveru posílán s HTTP požadavkem v hlavičce „Authorization“ a slouží k autentizati uživatele. V aplikaci je tento token uložen buďto v session storage browseru nebo jako lokální proměnná, takže token není přístupný z vnějšku aplikace.

Druhý, refresh token, bývá obvykle pouze sekvence náhodných znaků a jeho platnost se může pohybovat v řádu jednotek až menších desítek dní (obvykle 14 dní). Tento token je uložen v HTTP only cookie a slouží k vyžádání nového access tokenu v případě, že se blíží jeho expirace. [39, 40] Autentizace s použitím JWT je znázorněna na obrázku 4.2.

Protože práce s těmito tokeny není na úrovni GraphQL příliš elegantní (alespoň při použijí knihovny Graphene), rozhodl jsem se nezahrnovat autentizační operace do GraphQL API, ale implementoval jsem pro ně oddělené endpointy. Kromě endpointu pro GraphQL proto backend aplikace vystavuje také endpoint pro přihlášení uživatele a vydání dvojice (access token, refresh token), vyžádání nového access tokenu pomocí refresh tokenu a pro odhlášení a znehodnocení existujících tokenů.



Obrázek 4.1: Schéma databáze aplikace (zdroj: autor)



Obrázek 4.2: Autentizace za použití JWT (zdroj: [39])

4.5 Tvorba GraphQL API

Jak už jsem zmínil v kapitole 4.2.1, k vytvoření GraphQL API jsem použil knihovnu Graphene. Graphene je implementací GraphQL, která využívá tzv. code-first přístup, kde definice schématu nespočívá v použití SDL definovaným specifikací GraphQL, ale definováním jednotlivých stavebních kamenů GraphQL API pomocí tříd daného programovacího jazyka. Opakem code-first přístupu je schema-first, který definici v SDL využívá. Při volbě přístupu záleží především na preferencích programátora, protože většina implementací GraphQL umožňuje oba přístupy na sebe vzájemně převádět. Ze schématu v SDL lze vygenerovat datové třídy pro konkrétní programovací jazyk a naopak ze tříd schéma GraphQL API.

4.5.1 Objektové typy

Tvorba jednotlivých objektových typů pro GraphQL byla poměrně přímočará. Jak jsem zmínil v kapitole 4.2.1, knihovna graphene umožňuje vytvoření objektových typů přímo z datovým Django modelů. Vytvořené objektové typy obsahují většinu dostupných polí z databáze (viz 4.3). Některé z nich také vypočtená pole jako například celkovou sumu u faktur.

Asi nejvíce přidaných polí má typ „UserType“. Ten kromě polí převzatých z databázového modelu má navíc pole „hasOwnCompany“, což je příznak indikující, že uživatel má vyplněné údaje o vlastní firmě, „hasTogglSettings“ označující, že uživatel má vyplňenou konfiguraci pro integraci se službou Toggl a dále „displayName“ s celým jménem (křestním jménem a příjmením) a „yearJoined“ s rokem vytvoření uživatelského účtu.

Kromě typů vycházejících z datových modelů, definuje API ještě několik dalších typů. Jedním z nich je „OverviewType“, který poskytuje data ke zobrazení ročních přehledů.

Další typy jsou spojeny se službou Toggl. Kořenovým typem pro data je typ „TogglDataType“, jehož pole obsahují celkový počet naměřených hodin za jeden měsíc a seznamy tvořené z typů „TogglDayType“ s denním přehledem hodin a „TogglProjectType“ s měsíčním přehledem naměřených hodin podle jednotlivých projektů.

Speciálním definovaným typem je „ErrorType“, který slouží jako schránka pro chybové hlášky vracené z mutací.

Jako příklad je v ukázce kódu 4.1 zobrazena deklarace typu „UserType“. [9]

```
import graphene
from graphene_django import DjangoObjectType

from kraccount.core.models import User


class UserType(DjangoObjectType):
    has_own_company = graphene.Boolean()
    has_toggl_settings = graphene.Boolean()
    display_name = graphene.String()
    year_joined = graphene.Int()

    class Meta:
        model = User
        fields = [
            'id',
            'username',
            'first_name',
            'last_name',
            'email',
            'show_dic',
            'toggl_api_token',
            'toggl_workspace_id',
            'email_notifications',
        ]

    @staticmethod
    def resolve_display_name(root: User, info):
        if root.first_name and root.last_name:
            return root.get_full_name()
        return root.username
```

Ukázka kódu 4.1: Deklarace typu UserType (zdroj: autor)

[9] Ostatní typy, stejně jako dále uvedené mutace, jsou k nahlednutí ve zdrojovém kódu aplikace v elektronické příloze.

4.5.2 Dotazy

me Vrací aktuálně přihlášeného uživatele.

myCompany Vrací instanci OwnCompany aktuálně přihlášeného uživatele.

companies Vrací seznam firem v adresáři uživatele.

company Vrací detail firmy na základě ID v argumentu.

overview Vrací roční přehled příjmů a výdajů

togglData Vrací data ze služby Toggl. Konkrétně se jedná o přehled hodin za jednotlivé dny a poté čas strávený na jednotlivých projektech.

invoices Vrací seznam faktur. Pomocí argumentů je možné ofiltrovat výsledek na konkrétní rok vystavení faktury, určité stavy faktur, určit zda má jít o vystavené, či přijaté faktury nebo limitovat velikost výsledku.

invoice Na základě ID v argumentu vrací detail faktury.

4.5.3 Mutace

Pro tvorbu GraphQL mutací jsem vytvořil základní třídu „BaseMutation“, ze které dědí všechny ostatní třídy implementující mutace. Třída do všech mutací přidává výstupní pole „ok“ a „errors“. Ok slouží k rychlé indikaci, že mutace proběhla v pořádku. V případě, že mutace selhala, třída „BaseMutation“ transformuje známé vyjímky, které byly vyhozeny např. na základě neúspěšné validace nebo nalezení objektu v databázi, do pole errors. K tomu slouží výše zmíněný „ErrorType“. V poli „output“ vrací všechny mutace objekt, kterého se mutace dat týkala.

Většina mutací přijímá jako argumenty větší množství polí. Bylo proto nasnadě vytvořit pro ně vstupní typy. Konkrétní vstupní typy jsou zmíněny u jednotlivých mutací.

Jako příklad je uvedena mutace pro změnu stavu faktury v ukázce kódu 4.2.

```python
import graphene

from kraccount.core.graphql.mutations import BaseMutation
from kraccount.invoice.graphql.types import InvoiceType, InvoiceStateEnumType


class InvoiceChangeStateMutation(BaseMutation):
    output = graphene.Field(InvoiceType)

    class Arguments:
        id = graphene.ID(required=True)
        status = graphene.Argument(InvoiceStateEnumType, required=True)

    @classmethod
    def perform_mutate(cls, root, info, **kwargs):
        invoice = info.context.user.invoices.get(pk=kwargs['id'])
        target_state = kwargs['status']
        if invoice.is_state_allowed(target_state):
            invoice.state = target_state
            invoice.save()
        return cls(ok=True, output=invoice)
```

Ukázka kódu 4.2: Mutace pro změnu stavu faktury (zdroj: autor)

userSettingsUpdate Nastavení jména, příjmení a e-mailu uživatele, zobrazování DIČ, aktivace upozornění na e-mail a nastavení integrace s Togglem. Jako argument přijímá vstupní typ „UserSettingsInput“

userPasswordChange Změna hesla aktuálně přihlášeného uživatele. Na vstupu vyžaduje řetězce se starým a novým heslem.

companyCreate Vytvoření firmy v adresáři. Argumenty mutace jsou rozděleny na „CompanyInput“ a „AddressInput“. Společně s instancí firmy je v databázi vytvořena také adresa.

companyUpdate Editace firmy a adresy v adresáři firem. Kromě dvou vstupních typů jako v předchozím případě očekává tato mutace ID editované firmy.

companyDelete Odstranění firmy z adresáře. Mutaci není možné pokud je na firmu navázána existující faktura. Na vstupu očekává argument s ID firmy.

myCompanyUpdate Slouží k editaci dat vlastní firmy přihlášeného uživatele. Na vstupu přijímá „OwnCompanyInput“ a „AddressInput“.

invoiceCreate Vytvoření faktury. Jedním ze vstupních argumentů je mimo jiné informace, zda jde o fakturu vydanou, či přijatou. Faktura se vytváří bez položek. Vstupem je typ „InvoiceInput“.

invoiceUpdate Editace existující faktury. Jako argumenty má „InvoiceInput“ a ID faktury.

invoiceDelete Odstranění faktury a jejích položek podle ID faktury na vstupu.

invoiceAddItem Přidání položky k faktuře pomocí ID faktury a také vstupního objektu „InvoiceItemInput“.

invoiceUpdateItem Editace položky, která je navázána na fakturu. Podobně jako při přidání faktury kromě „InvoiceItemInput“ požaduje na vstupu ID položky faktury z databáze.

invoiceRemoveItem Odstranění položky z faktury podle ID položky v argumentu.

invoiceChangeState Změna stavu faktury. Součástí mutace je validace změny stavu. Např. uhrazená faktura nemůže být označena jako opožděná. Na vstupu má ID fakturu a hodnotu výčtového typu s požadovaným cílovým stavem.

invoiceTemplateCreate Vytvoření šablony faktury ze vstupního typu „InvoiceTemplateInput“

invoiceTemplateUpdate Editace šablony z dat v objektu „InvoiceTemplateInput“. Podobně jako ostatní editační mutace požaduje na vstupu také ID modifikovaného objektu.

invoiceTemplateDelete Odstranění šablony pomocí jejího ID.

4.6 Komunikace s GraphQL API

Pro komunikaci frontendu s GraphQL API slouží knihovna Apollo Client. S GraphQL je možné komunikovat i bez dalších knihoven s pomocí funkce „fetch“ vestavěné v browseru, ale Apollo Client přidává velké množství výhod, mezi které patří například snadnější odchytávání a zpracování chyb nebo možnost opakování dotazu v případě neúspěšné síťové komunikace. Tato funkcionalita je realizovaná pomocí zřetězených funkcí, které se v terminologii Apollo Clienta nazívají Apollo Links. Ve srovnání s webovými frameworky jde o podobný systém jako řetězení middlewarů.

Apollo Linky v aplikaci řeší autentizaci uživatele a přidání access tokenu do hlaviček HTTP požadavku, opakování požadavku při chybě v komunikaci, zpracování chyb v rámci mutace, zpracování obecných chyb GraphQL a konfigurace parametrů HTTP požadavků posílaných na GraphQL endpoint.

V kapitole 4.2.2 jsem zmiňoval použití generátoru GraphQL Codegen. Výstupem generátoru jsou kromě typů pro TypeScript také funkce pro volání jednotlivých dotazů a mutací. Všechny funkce jsou ve dvou variantách. První varianta dotaz provede okamžitě při přidání komponenty, která dotaz volá, do DOM[10] stránky. Druhá obsahující v názvu slovo „lazy“ umožňuje manuální zavolání dotazu kdykoliv během životního cyklu komponenty.

Jednotlivé dotazy a mutace jsou stručně popsány již v předchozí kapitole 4.5. Jeden z použitých dotazů je zobrazen v ukázce kódu 4.3. [11]

[10] Document Object Model

[11] Ostatní dotazy jsou zahrnuty ve zdrojovém kódu frontendové aplikace v elektronické příloze.

```
fragment AddressFragment on AddressType {
    id
    street
    number
    city
    zipCode
    state
}

fragment CompanyDetailFragment on CompanyType {
    id
    displayName
    name
    ico
    dic
    vat
    account
    bankCode
    phone
    email
    companyType
    address {
        ...AddressFragment
    }
}

query CompanyDetail($id: ID!) {
  company(id: $id) {
    ...CompanyDetailFragment
  }
}
```

Ukázka kódu 4.3: Dotaz použitý na stránce s detailem firmy (zdroj: autor)

4.7 Uživatelské rozhraní

Frontend aplikace jsem napsal za pomoci frameworku NextJS, který je postavený na knihovně React. Tato knihovna umožňuje rozčlenění webové stránky na izolované, znovupoužitelné komponenty, které je možné následně vkládat do stránky jako funkční celky. Tímto způsobem jsou vytvořeny společné prvky všech stránek, jako je např. vrchní navigační lišta, ale i prvky, které jsou použité pouze na jednom místě. Knihovna React dále usnadňuje práci s událostmi a vývoj interaktivního uživatelského rozhraní. Je tedy možné s ní efektivně řešit např. validaci a odesílání formulářů a obsluhu dalších akcí uživatele.

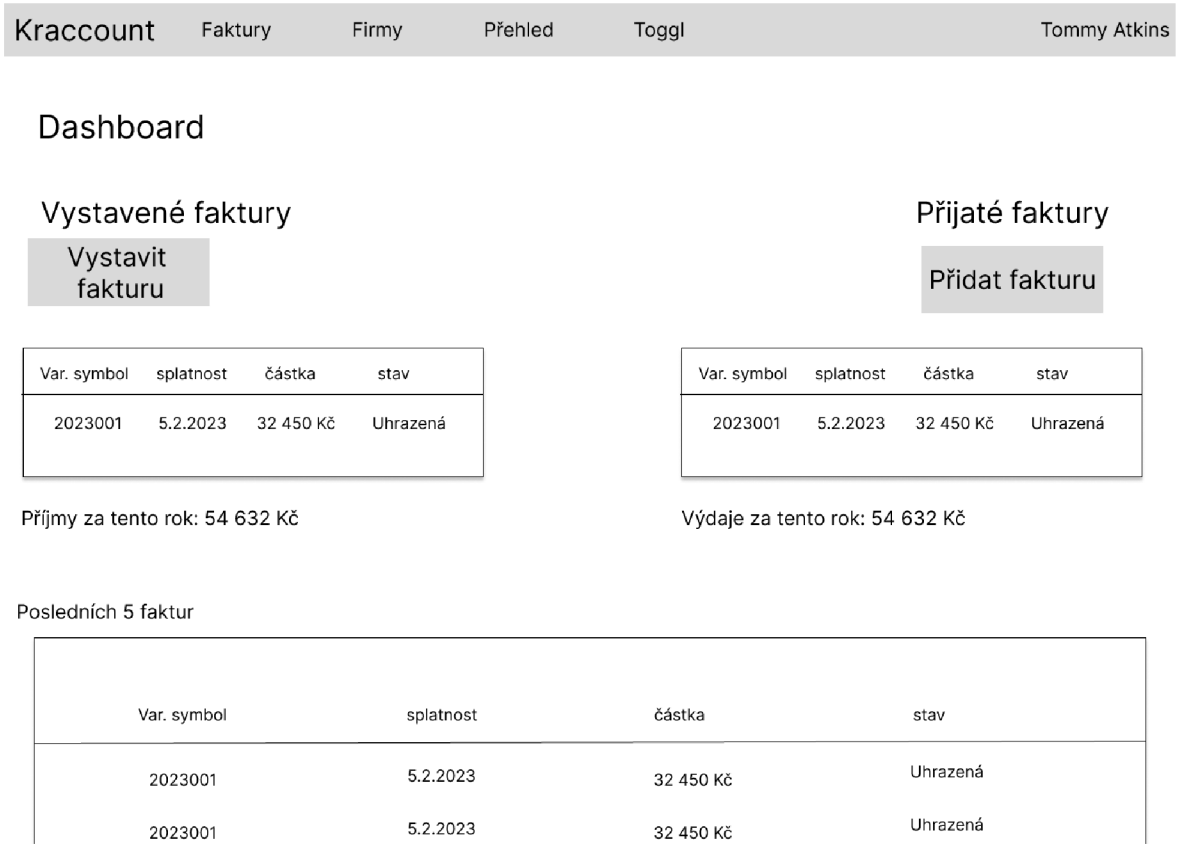
Pro tvorbu komponent jsem zvolil novější metodu založenou na deklaraci komponent pomocí funkcí a používání hooků definovaných v knihovně React. Hooky umožňují správu stavu a vedlejších efektů[12] jednotlivých komponent a také správu globální kontextu aplikace (např. aktuálně přihlášeného uživatele). Druhá možnost pro tvorbu komponent je deklarování komponent ve formě tříd, což je starší metoda, která se postupně stává méně rozšířenou.

Framework NextJS je v aplikaci použitý hlavně pro jeho implementaci routeru, což dovoluje překreslovat stránky aplikace podle změny URL v adresním řádku browseru, aniž by bylo nutné stránku celou obnovovat. I jednotlivé stránky jsou ve frameworku obhospodařovány jako komponenty. Jako hlavní komponenta je reprezentována i celá aplikace s rozvržením stránky spolu se společnými prvky, které jsou součástí všech stránek aplikace.

Pro vzhled stránky jsem zvolil jednodušší funkční design, kterého je dosaženo pomocí frameworku Tailwind. Stará verze aplikace využívala framework Boostrap a v rámci vývoje nové SPA verze jsem se snažil o vlastní implementaci některých komponent v Tailwindu. Jedná se například o tlačítka, formuláře, tabulky a navigační lištu. Kvůli jednoduchému designu jsem netrávil příliš času návrhem jednotlivých obrazovek. Pro účely návrhu jsem však sestavil alespoň wireframe dashboardu aplikace, který je možné vidět na obrázku 4.3[13].

Tailwind umožňuje mobile-first přístup, tedy vytvoření základního designu pro nejužší rozměr obrazovky (mobilního telefonu) a přetěžování některých atributů pro větší rozměry obrazovek. Pomocí Tailwindu je tedy možné vytvořit responzivní webový design, který bude vhodný pro mobilní telefony i pro monitory s velkou úhlopříčkou. Pří vývoji aplikace nebyla optimalizace pro mobilní zařízení priorita, nicméně snažil jsem se některé

[12] Překreslování konponent v rámci jejich životního cyklu na stránce.
[13] Obrázek je ve větším rozlišení součástí elektronické přílohy.

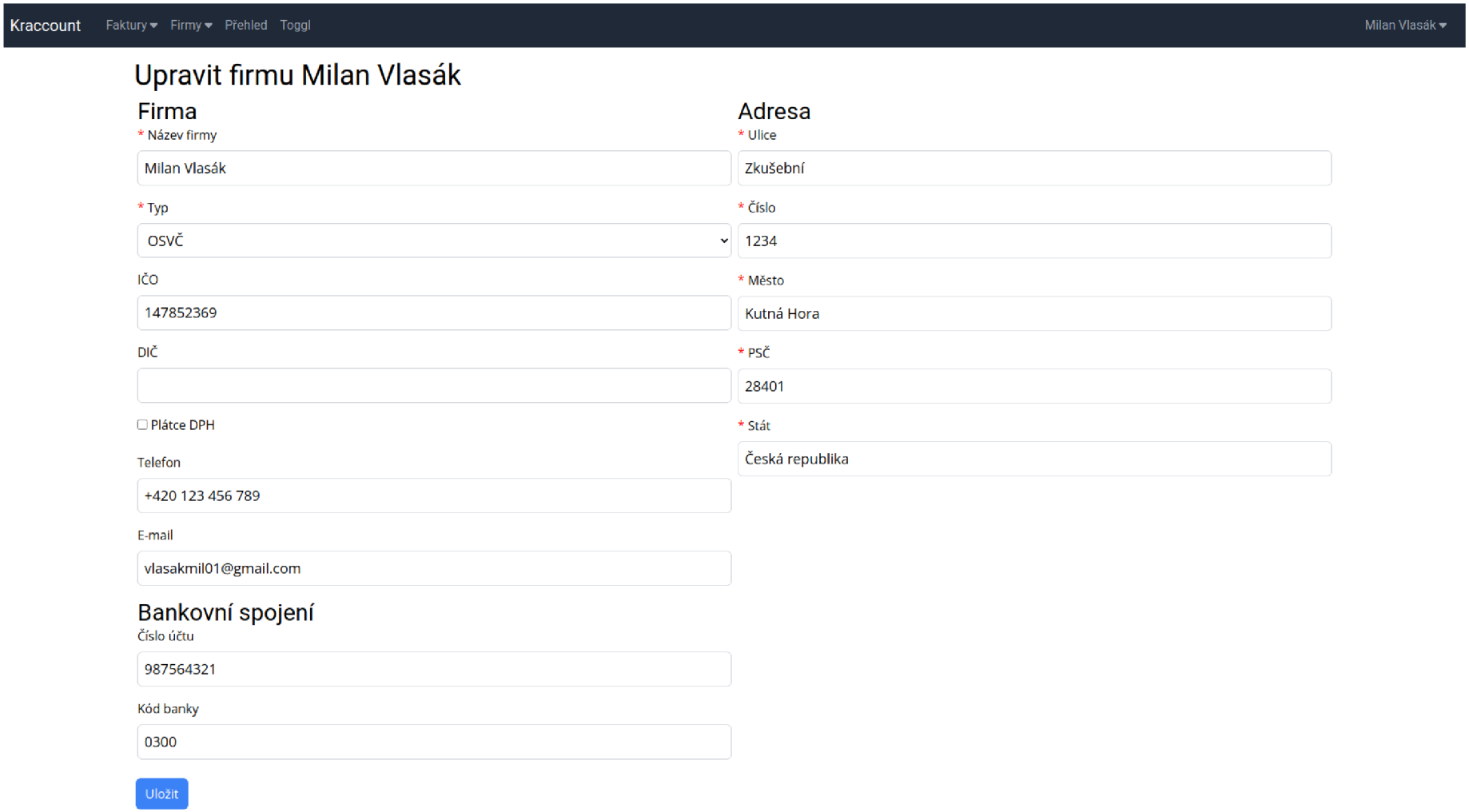


Obrázek 4.3: Wireframe aplikace (zdroj: autor)

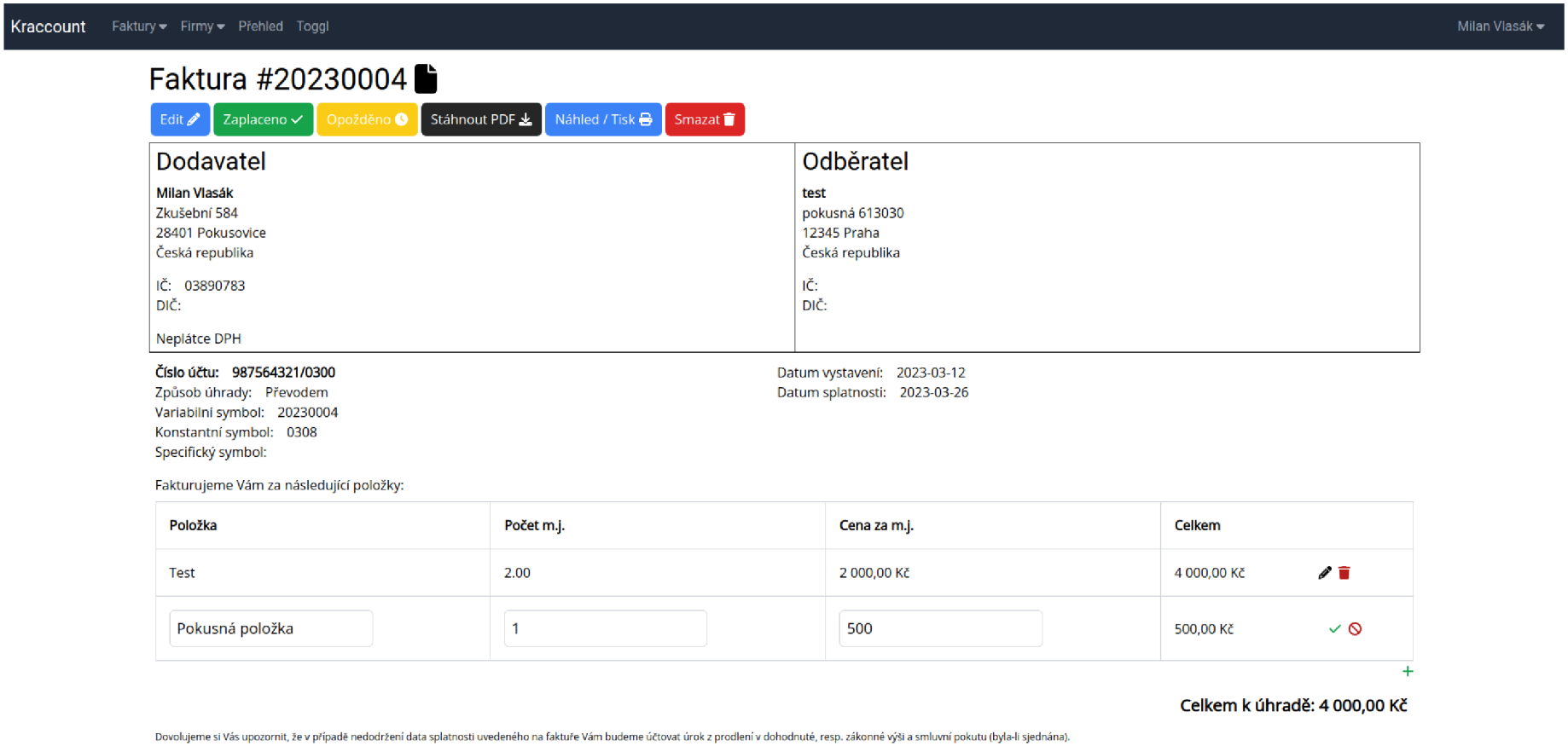
z komplikovanějších komponent vytvořit dostatečně responzivní. Například položky v horní navigační liště jsou na menších obrazovkách skryté a na místo nich je zobrazena ikona sloužící pro otevření nabídky. Na větších obrazovkách se naopak tato ikona nezobrazuje a na navigační liště jsou viditelné všechny hlavní položky, přičemž některé z nich mají ještě vlastní podnabídky, které se zobrazují po kliknutí myší.

Na obrázku 4.4 je screenshot z aplikace s formulářem pro zadání informací o vlastní firmě. Na dalším obrázku 4.5 je potom screenshot z detailu faktury[14].

[14]Oba obrázky jsou ve větším rozlišení součástí elektronické přílohy.



Obrázek 4.4: Screenshot z aplikace - formulář firmy (zdroj: autor)



Obrázek 4.5: Screenshot z aplikace - detail faktury (zdroj: autor)

4.8 Testování

Během vývoje jednotlivých GraphQL dotazů a mutací na backendu aplikace jsem používal manuální testování skrze rozhraní GraphiQL. Následně jsem pro testování backendu použil automatizované testy. Pro tvorbu testů jsem použil rozšíření standardní knihovny jazyku Python unittest, které poskytuje framework Django. Některé metody a funkce jsou pokryté jednotkovými testy. Konkrétně se jedná o generování variabilního symbolu vystavovaných faktur a autentizace pomocí JWT. Dále jsou použité integrační testy pro endpointy pracující s JWT, některé GraphQL dotazy a většinu mutací. Unittest v Djangu přidává možnost použití HTTP klienta, který umí volat jednotlivá URL z Django aplikace. Tímto způsobem je možné testovat volání jednotlivých API endpointů včetně HTTP komunikace. U všech testů jsem se snažil pokrýt většinu možných nevalidních vstupů.

Testování frontendu probíhalo pouze manuálně. Jednotlivé stránky a komponenty jsem testoval v průběhu jejich vývoje a následně souhrnně po jeho skončení. Během testování frontendu jsem se zaměřoval především na zpracování chyb na úrovni GraphQL v případě nevalidních vstupů mutací.

4.9 Nasazení aplikace

Nasazování backendu a frontendu aplikace funguje odděleně. Backend je nasazen na pronajatém virtuálním privátním serveru s linuxovou distribucí Ubuntu, který je plně automaticky konfigurován pomocí nástroje Ansible. Jeden konfigurační proces v Ansiblu se nazývá playbook, který může být členěn na menší role. Ty obsahují jednotlivé úlohy, jenž je potřeb na cílovém serveru vykonat.

Pro vlastní účely jsem vyvinul několik playbooků. Instalační playbook má za úkol nainstalovat obecné systémové balíčky, nainstalovat PostgreSQL databázi, Redis, Nginx a další služby. Instalační playbook není součástí repozitáře aplikace. Playbook, který slouží k nasazování aplikace, má za úkol vytvoření adresářové strukturu a databáze pro aplikaci. Dale vytváření virtuálního prostředí pro jazyk Python, instalaci závislostí aplikace, vytvoření konfiguračních souborů pro Supervisor, Nginx a uWSGI. Nasazovací playbook je zahrnut v repozitáři, ale jednotlivé role, které se v playbooku volají, jsou odkazovány jako závislosti playbooku z jiných repozitářů.

K nasazení frontendu jsem zvolil hostingovou službu Vercel. Ta funguje zcela automatizovaně a podstatně jednodušejí, než v případě backendu. Nasazení na Vercel spočívá v přidání repozitáře do Vercelu a schválení přístupových práv Vercelu do repozitáře. Vercel si v repozitáři zaregistruje webhook, který je spuštěn při každém uploadu změn kódu do repozitáře a spouští sestavení produkčního buildu a nasazení aplikace.

5 Diskuse a výsledky

Aplikaci, která byla v rámci této práce vytvořena, mám v plánu (stejně jako její starší verzi) nadále využívat pro své účely při vystavování faktur mým klientům, neboť splňuje všechny základní funkcionality v plném rozsahu definované v kapitole 4.1.3. Aplikaci bych dále mohl nabídnout také známým kolegům, kteří rovněž pracují jako OSVČ a k vystavování faktur využívají komerční nástroje, které jim nemusí zcela vyhovovat.

Backend aplikace by měl být nejen díky frameworku Django dostatečně robustní. Pro lepší škálovatelnost v případě většího počtu uživatelů by mohlo být výhodnější kontejnerizovat aplikaci pomocí nástroje Docker nebo jiné kontejnerové platformy a následně přepracovat nasazování do kontejnetové orchestrace jako např. Docker Swarm nebo Kubernetes. Tato změna by umožnila provozovat backend ve větším počtu replik na clusteru o velikosti dvou a více serverů. Na druhou stranu, dokud nebude provoz aplikace vyžadovat více replik, nebude tato úprava potřeba.

Naopak frontendová webová aplikace je dostatečně škálovatelná sama o sobě díky použitému hostingu. Zároveň díky architektuře SPA nebude mimo stažení aplikace browserem generovat pro servery žádnou větší zátěž a aplikace následně pouze komunikuje s backendem přes API. Vzhledem k tomu, že působím primárně jako backendový vývojář, nejsem zcela zběhlý ve vývoji frontendu a jistě se najdou na straně frontendové aplikace různé technické nedostatky. Například ne vždy se mi dařilo dosáhnout reaktivnosti celé stránky při změně globálního stavu aplikace, což nastává obvykle při zobrazování hlášek po provedení (nebo selhání) GraphQL mutací. Během testování docházelo občas k tomu, že ne všechny momentálně vykreslené komponenty se o těchto změnách dozvěděly, aniž by uživatel nepřešel na jinou stránku aplikace. Pravděpodobně se jedná o problém použité metody pro aktualizaci stavů komponent a mělo by být možné tento problém vyřešit volbou jiného postupu. I přes tento občasný nedostatek nic nebrání produkčnímu užívání aplikace.

5.1 Možná další rozšíření aplikace

Kromě známých problém na straně frontendu, je několik cest, kudy by mohl směřovat další vývoj aplikace. Jedná se o větší nebo měnší aktualizace a technické úpravy, které by rozšiřovaly funkcionalitu aplikace.

Registrace uživatelů V současné verzi aplikace nemá registrační formulář a potenciální uživatelé jsou odkázání na spolupráci se správcem aplikace, aby jim vytvořil uživatelský účet.

Lepší vizuální design Aplikace má pouze jednoduchý, čistě účelový design a pokud by měla být k dispozici širší veřejnosti, bylo by vhodné zlepšit její vizuální stránku.

Rozšíření integrace s Togglem Současná integrace se službou Toggl slouží k alternativnímu zobrazení dat z Togglu. Rozšíření integrace by mohlo spočívat například v převedení odpracovaných hodin v Togglu na jednotlivé položky ve faktuře s předem definovanou hodinovou sazbou.

Položky v šablonových faktur Šablony faktur aktuálně neumožňují přidávání položek a jedná se pouze šablony hlavičky faktury s kontaktními a platebními údaji.

Faktury s QR kódem Na faktury by bylo možné přidat QR kód, který by při naskenování do bankovní mobilní aplikace, usnadňoval placení faktur.

Refaktorizace API do knihovny Strawberry Jak jsem zmiňoval v kapitole 4.2.1, během výoje jsem zvažoval přepsání API do knihovny Strawberry. Tato knihovna má kolem sebe aktivnější komunitu, než Graphene a dá se předpokládat, že nové verze knihovny budou vycházet častěji a pravidelněji. Strawberry navíc využívá modernější přístupy, které přišly s novými verzemi Pythonu v posledních letech, což vede k čitelnějšímu a úsporněji napsanému kódu aplikace.

6 Závěr

Bakalářská práce se zabývá problematikou API podle specifikace GraphQL ve webových aplikacích. Jejím cílem bylo popsat způsob stavby API a grafových dotazů podle specifikace GraphQL. Dále vytvoření single-page webové aplikace používající GraphQL.

V úvodu teoretické části nastiňuje teorii grafů, základní informace o grafových databázích a protokolu HTTP, kde představuje jednotlivé metody požadavků a návratové kódy, se kterými se dá nejčastěji setkat při vývoji webových API. Teoretická část se dále zabývá základní definicí GraphQL specifikace a popisem stavebních prvků GraphQL API, přičemž ukazuje na příkladech veškeré možnosti definice schématu grafového API pomocí jazyka SDL ze specifikace GraphQL.

Druhá polovina teoretické části popisuje dotazovací jazyk GraphQL, který slouží klientským aplikacím ke komunikaci s GraphQL API. Představuje způsoby zápisu dotazů, mutací a fragmentů včetně použití proměnných a dalších postupů, které přispívají ke znovupoužitelnosti dotazů. Všechny tyto prvky jsou podobně jako v případě SDL ukázány na příkladech. V závěru teoretické části je rozebráno stručné srovnání se starší a rozšířenější alternativou ke GraphQL, REST API, kde jsou popsány společné a rozdílné vlastnosti obou přístupů.

Praktická část práce popisuje vývoj SPA aplikace s pracovním názvem Kraccount. Cílem aplikace je zjednodušení agendy s vystavováním faktur pro drobné živnostníky. Backend aplikace je implementován v programovacím jazyce Python za použití webového frameworku Django a pro tvorbu GraphQL API využívá knihovnu Graphene. Frontend aplikace je napsát v jazyce TypeScript, využívá framework NextJS a s GraphQL API komunikuje pomocí knihovny Apollo Client.

Praktická část se zabývá návrhem architektury a databáze aplikace. Popisuje, jak je řešena autentizace uživatelů a představuje definované GraphQL dotazy a mutace, které aplikace pro svou funkci využívá, z pohledu serveru i klienta. V práci je také vysvětleno, jakým způsobem je zajištěna komunikace frontendu s backendem na úrovni GraphQL a HTTP. Následně je popsáno testování aplikace a jakým způsobem je řešeno její nasazování.

Poslední část práce diskutuje výslednou realizaci aplikace, rozebírá známé problémy a diskutuje další možná rozšíření aplikace nad rámec této práce. Vytvořená aplikace může

být produkčně nasazena a nabídnuta dalším uživatelům. Mimo to by tato práce spolu s přiloženým zdrojovým kódem mohla posloužit například jako podklad k vytvoření kurzu základů GraphQL.

7 Seznam použitých zdrojů

1. JIROVSKÝ, Lukáš. *Teorie grafů* [online]. 2010-09-23 [cit. 2021-07-18]. Dostupné z: https://teorie-grafu.cz.
2. BUNA, Samer. *GraphQL in Action*. Manning Publications Co, 2021. ISBN 9781617295683.
3. RAMBA, Jaroslav. *Grafová databáze Neo4j: Grafová terminologie a dostupné technologie* [online]. 2013-10-21 [cit. 2021-07-25]. Dostupné z: https://zdrojak.cz/clanky/grafova-terminologie-a-dostupne-technologie/.
4. THE INTERNET SOCIETY. *rfc-2616: Hypertext Transfer Protocol – HTTP/1.1* [online]. 1999 [cit. 2021-07-25]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc2616.
5. MALÝ, Martin. *REST: Architektura pro webové API* [online]. 2009-08-03 [cit. 2021-07-25]. Dostupné z: https://zdrojak.cz/clanky/rest-architektura-pro-webove-api/.
6. GRAPHQL FOUNDATION. *GraphQL* [online]. 2018 [cit. 2021-07-11]. Dostupné z: https://graphql.org/.
7. BYRON, Lee. *GraphQL: A data query language* [online]. 2015-09-14 [cit. 2021-07-11]. Dostupné z: https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/.
8. LINUX FOUNDATION. *The Linux Foundation Announces Intent to Form New Foundation to Support GraphQL* [online]. 2018 [cit. 2021-07-11]. Dostupné z: https://www.linuxfoundation.org/press-release/2018/11/intent_to_form_graphql/.
9. *GraphQL Specification Versions* [online]. 2023 [cit. 2023-02-19]. Dostupné z: https://spec.graphql.org.
10. *Graphene: GraphQL in Python Made Easy* [online] [cit. 2021-07-26]. Dostupné z: https://graphene-python.org.

11. *How to GraphQL: The Fullstack Tutorial for GraphQL* [online]. 2021 [cit. 2021-07-18]. Dostupné z: https://www.howtographql.com.

12. *GraphQL vs REST* [online] [cit. 2023-02-07]. Dostupné z: https://hasura.io/learn/graphql/intro-graphql/graphql-vs-rest/.

13. *Pohoda: ekonomický a informační systém* [online] [cit. 2023-03-10]. Dostupné z: https://www.stormware.cz.

14. *Fakturoid* [online] [cit. 2023-03-10]. Dostupné z: https://www.fakturoid.cz.

15. *Python* [online] [cit. 2022-12-30]. Dostupné z: https://www.python.org.

16. *Django: The web framework for perfectionists with deadlines* [online] [cit. 2022-12-30]. Dostupné z: https://www.djangoproject.com.

17. *uWSGI* [online] [cit. 2023-01-19]. Dostupné z: https://uwsgi-docs.readthedocs.io/en/latest/.

18. *Celery: Distributed Task Queue* [online] [cit. 2022-12-30]. Dostupné z: https://docs.celeryq.dev.

19. *PostgreSQL: The World's Most Advanced Open Source Relational Database* [online] [cit. 2023-01-19]. Dostupné z: https://www.postgresql.org.

20. *Redis* [online] [cit. 2023-01-19]. Dostupné z: https://redis.io.

21. *Supervisor* [online] [cit. 2023-01-19]. Dostupné z: http://supervisord.org.

22. *Nginx: Advanced Load Balancer, Web Server and Reverse Proxy* [online] [cit. 2023-01-19]. Dostupné z: https://www.nginx.com.

23. *Let's Enrypt* [online] [cit. 2023-01-19]. Dostupné z: https://letsencrypt.org.

24. *Node.js* [online] [cit. 2023-02-03]. Dostupné z: https://nodejs.org.

25. *TypeScript: JavaScript With Syntax For Types* [online] [cit. 2023-02-03]. Dostupné z: https://www.typescriptlang.org.

26. *React: A JavaScript library for building user interfaces* [online] [cit. 2023-02-03]. Dostupné z: https://reactjs.org.

27. *Next.js by Vercel: The React Framework* [online] [cit. 2023-02-03]. Dostupné z: https://nextjs.org.

28. *Apollo GraphQL: Supergraph: unify APIs, microservices, & databases in a composable graph* [online] [cit. 2023-02-03]. Dostupné z: https://www.apollographql.com.

29. *GraphQL Code Generator* [online] [cit. 2023-02-03]. Dostupné z: https://the-guild.dev/graphql/codegen.

30. *Vercel* [online] [cit. 2023-02-03]. Dostupné z: https://vercel.com.

31. *Tailwind: Rapidly build modern websites without ever leaving your HTML* [online] [cit. 2023-02-03]. Dostupné z: https://tailwindcss.com.

32. *Git* [online] [cit. 2023-02-03]. Dostupné z: https://git-scm.com.

33. *Ansible is Simple IT Automation* [online] [cit. 2023-02-03]. Dostupné z: https://www.ansible.com.

34. *dbdiagram.io: Database Relationship Diagrams Design Tool* [online] [cit. 2023-03-10]. Dostupné z: https://dbdiagram.io.

35. *Figma: the collaborative interface design tool* [online] [cit. 2023-03-10]. Dostupné z: https://www.figma.com.

36. *RFC 7519: JSON Web Token (JWT)* [online] [cit. 2023-02-07]. Dostupné z: https://www.rfc-editor.org/rfc/rfc7519.

37. *JSON Web Token (JWT)* [online]. 2015-01-23 [cit. 2023-02-19]. Dostupné z: https://www.iana.org/assignments/jwt/jwt.xhtml.

38. *JWT: JSON Web Tokens* [online] [cit. 2022-12-30]. Dostupné z: https://jwt.io.

39. *The Ultimate Guide to handling JWTs on frontend clients (GraphQL)* [online] [cit. 2023-02-07]. Dostupné z: https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/.

40. *RFC 6749: The OAuth 2.0 Authorization Framework* [online] [cit. 2023-02-07]. Dostupné z: https://www.rfc-editor.org/rfc/rfc6749.

8 Přílohy

8.1 Obsah elektronických příloh

```
/
├── README.md ..... Stručný popis obsahu příloh ve formátu Markdown
├── src/
│   ├── backend/ ..... Zdrojový kód backendu aplikace
│   ├── frontend/ ..... Zdrojový kód frontendu aplikace
│   └── thesis/ ..... Zdrojový kód práce a tezí ve formátu XeLaTeX
├── images/
│   ├── db_schema.png ..... Schéma databáze aplikace ve formátu PNG
│   ├── screenshot.png ..... Screenshot z aplikace ve formátu PNG
│   ├── screenshot_2.png ..... Screenshot z aplikace ve formátu PNG
│   └── wireframe.png ..... Wireframe aplikace ve formátu PNG
├── zaverecna_prace.pdf ..... Text práce ve formátu PDF
└── teze.pdf ..... Teze práce ve formátu PDF
```