FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Wowee: Komparace GraphQL a REST API
Student:	Michal Martinek
Vedoucí:	Ing. Petr Pauš, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je analyzovat využití GraphQL na již existující službě Wowee (bývalé ElateMe) a srovnat výhody a nevýhody řešení oproti použité REST API. Součástí práce bude i návrh struktury serveru, komunikace s klientem a zabezpečení dat. Za úkol si tato práce rovněž klade implementaci prototypu serverové i klientské části a otestování výkonu oproti současné implementaci.

Analyzujte:
- využití GraphQL na službě Wowee,
- optimalizaci výkonu oproti REST API.

Navrhněte:
- strukturu serveru a interakci s klientem.

Implementujte:
- prototyp rozhraní,
- prototyp klientské aplikace využívající GraphQL.

Otestujte:
- vytvořte unit testy,
- porovnejte výkon REST a GraphQL řešení.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 11. ledna 2018

Poděkování

Chtěl bych poděkovat vedoucímu mé práce Ing. Petr Pauš, Ph.D za vedení a cenné rady a Bc. Michalovi Maněnovi, vedoucímu projektu Wowee, za technické připomínky. Také bych rád poděkoval mé rodině za podporu během celého studia a za závěrečné korektury této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

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

V Praze dne 10. května 2018 .....

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

Martinek, Michal. *Wowee: Komparace GraphQL a REST API.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je analýza použití GraphQL na již existující službě Wowee, což je nová crowdfundingová platforma s elementy sociální sítě. Na rozdíl od jiných podobných projektů, jako je Kickstarter nebo Patreon, které podporují vývoj kreativních a komerčních projektů prostřednictvím zájemců, je Wowee zaměřena na plnění osobních přání s pomocí přátel uživatelů. V současné době je vyvinuta webová verze, mobilní aplikace pro Android a iOS a backend REST API.

Práce je zaměřena na porovnání výhod a nevýhod technologie REST API, jež je v současné době použita pro komunikaci serveru s klientem, s novou technologií GraphQL, jež slibuje efektivnější řešení. Součástí práce je i návrh a implementace prototypu serverové i webové klientské části pomocí GraphQL a porovnání výkonnosti oproti stávajícímu řešení.

Klíčová slova Wowee, crowdfundingová platforma, socíální síť, GraphQL, API

Abstract

The main purpose of this thesis is an analysis of the usage of GraphQL on the project Wowee, a new crowdfunding platform with elements of the social network. Unlike other crowdfunding platforms, such as Kickstarter or Patreon that support new creative and commercial projects, the main goal of Wowee is fulfillment of the personal wishes with help of other users and friends. There are already developed a web app, native mobile apps for Android and iOS and the REST API server.

This thesis is focused on the comparison of advantages and disadvantages of REST, that is currently used for communication between the server and client, with new technology GraphQL that should be faster and more effective. Part of this work are also prototypes of GraphQL server and client in form of a web app written in React and a performance comparison with the original version using REST API.

Keywords Wowee, crowdfunding platform, scoial network, GraphQL, API

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Wowee je nová internetová crowdfundingová platforma s elementy sociální sítě. Jejím cílem je umožnit lidem jednoduchý výběr peněz ke splnění osobních tužeb a přání, a nikoliv financování kreativních projektů nebo technologických inovací, jako např. Kickstarter nebo Patreon. Uživatel může vytvořit přání buď pro sebe, nebo pro někoho jiného jako překvapení. Při vytvoření přání má možnost vyplnit všechny detaily přání sám, nebo ho může vyhledat, na základě čehož je mu doporučeno zboží z různých eshopů a po výběru se detaily vyplní automaticky. Další uživatelé pak na toto přání mohou přispět finanční částkou, a to více způsoby – kartou, bankovním převodem či Bitcoinem, nebo tato přání komentovat. Po dosažení cílové částky nebo časovém vypršení přání cílový uživatel rozhodne, zda peníze poputují na jeho osobní účet, nebo se odešlou zpět na účty přispívajících.

Na téma Wowee již bylo vypracováno na Fakultě informačních technologií ČVUT několik bakalářských prací ještě pod starým názvem ElateMe, celkem tři na vývoj mobilních aplikací, jedna na projektový management a reklamní server, a pak také jedna na vývoj backendu ve formě REST API. Mou motivací je analyzovat případy použití GraphQL a zjistit vlastní zkušeností prostřednictvím realizace opravdového projektu, zda je tento způsob návrhu API v praxi efektivnější. Rovněž to, zda vývoj serverové části není pro programátora jednodušší, jelikož s touto částí implementace GraphQL ještě nemám osobní zkušenosti.

Cílem práce je analyzovat využití GraphQL na již existující službě Wowee a srovnat výhody a nevýhody řešení oproti v současnosti používané REST API. Součástí práce bude i návrh struktury serveru, komunikace s klientem a zabezpečení dat. Za úkol si tato práce klade rovněž implementaci prototypu serverové i klientské části a otestovaní výkonu oproti současné verzi.

Kapitola 1

Analýza

1.1 O projektu a struktuře Wowee

Wowee je nová internetová platforma pro crowdfundingové financování přání osob jejich přáteli i jinými uživateli. Nejedná se tedy o financování kreativních projektů či technologických inovací jako např. Kickstarter nebo Patreon a přispěvatelé za svůj příspěvek nic nedostávají. Uživatelé mohou vytvářet přání pro sebe, pro své přátelé na síti i mimo ní, buď ručně nastaví titulek, popis, cenu a fotku, nebo mohou vybrat produkt z katalogu přímo na stránkách, či použít URL adresu stránky produktu např. na eshopu a upravit jen předvyplněné detaily. Toto přání se poté zobrazuje přátelům či vybraným osobám v aplikaci a je možné jej sdílet na sociálních sítích. Uživatelé mohou tato přání komentovat a samozřejmě na ně přispívat a to vícero způsoby kartou, bankovním převodem či Bitcoinem. Po dosažení cílové částky nebo časovém vypršení přání se peníze převedou na účet cíleného uživatele, který má ale rovněž možnost tyto peníze poslat zpět na účty přispívajících.

Všechny tyto operace je možné provádět ve webové verzi nebo v nativních aplikacích pro Android a iOS. Webová verze, na jejíž implementaci jsem se podílel v rámci předmětů BI-SP1 a BI-SP2, je implementována jako webová aplikace v ReactJS a používá společně s nativními aplikacemi REST API psané v Pythonu.

1.2 Srovnání REST API a GraphQL

REST je architektura rozhraní, navržená pro distribuované prostředí, bez specifikace a s velkou volností v implementaci. V posledních desetiletích se stala oblíbeným způsobem jak navrhnout API. Ale postupem času se s rostoucími požadavky klientů ukázalo REST jako nedostačující. GraphQL bylo vyvinuto Facebookem pro efektivní a flexibilní komunikaci jejich mobilních aplikací se

serverem a postupem času bylo použito i ve webové aplikaci. V roce 2015 bylo Facebookem zveřejněno a v současnosti k tomuto dotazovacímu jazyku existuje specifikace a oficiální i neoficiální sada nástrojů.

Tyto technologie porovnám v různých aspektech na konkrétním případu z aplikace Wowee, a to získání úvodního kanálu (feedu) 20 příspěvků se všemi potřebnými daty.

1.2.1 Získání dat

V případě REST API nejprve stáhneme první stranu kanálu pomocí GET požadavku na */wishes/*, a pak pro každé přání v kanálu získáme první stranu komentářů pomocí požadavku GET na */wishes/:wish_id/comments/*. Pak musíme ještě provést řadu GET požadavků na */account/restricted/:pk* pro profily uživatelů, kteří komentovali přání, nebo jsou příjemci.

Naproti tomu v případě GraphQL se odešle jeden jediný požadavek na společný koncový bod (endpoint), ve kterém přesně specifikujeme, kolik příspěvků chceme a jaké detaily mají obsahovat, včetně zpropagovaných údajů o autorovi atd. Obsah požadavku může vypadat nějak takto:

```
{
  feed(first: 20) {
    id,
    title,
    imageUrl,
    gatheredAmount,
    neededAmount,
    expirationAt
    receiver {
      id,
      image,
      firstName,
      lastName
    },
    comments(first:5) {
      text,
      createdAt,
      author {
       id,
       image,
       firstName,
       lastName
      }
    }
  }
}
```

1.2.2 Odesílání dat na server

Odesílání dat, jako třeba přidání nového přání, je v obou případech docela podobné. V případě REST API se jedná o odeslání POST požadavku na adresu */wishes/* s potřebnými daty, např ve formátu JSON či FormData, a v případě přidaného obrázku další POST požadavek na */wishes/upload/image* s obrázkem.

V případě GraphQL se dá obojí udělat pomocí jednoho POST požadavku na společný koncový bod. Tento požadavek může vypadat třeba nějak takto:

```
mutation {
  postWish(
    title: "Testing wish",
    description: "description",
    productUrl: "http:/test.com",
    image: imageFile,
    amount: 150,
    currency:   CZK
  ) {
    id
    description
    title
  }
}
```

1.2.3 Síťové požadavky

Zatímco GraphQL odešle vždy 1 požadavek, REST API odešle po prvním požadavku na kanál příspěvků dalších 20 požadavků na komentáře. Poté se odešlou požadavky na profilů uživatelů, těch může být až pro 20 příspěvků s až 5 komentáři od neznámých lidí dohromady až 20*6. Celkem jsme tedy až na 121 požadavcích.

Samozřejmě počet požadavků při používání REST lze omezit optimalizací jak na straně serveru, kde budeme předávat více informací a případně pro toto vytvoříme speciální endpointy [1], a také na straně klienta, kde je možné již částečně vycházet z obdržených dat. Tyto optimalizace ale znamenají pro vývojáře práci navíc a mohou vést až protichůdným směrem oproti správnému návrhu REST zaručujícímu jednoduchost a robustnost.

GraphQL umožnuje mimo klasických požadavků na data navázat pomocí WebSocketu takzvaný *Subscription*, pomocí kterého se posílají na klienta aktualizace dat, jako například nová notifikace, komentář, příspěvek atd.

1.2.4 Over/Under Fetching

V případě použití REST je časté stahování více dat, než je potřebné, jelikož každý endpoint vrací stálou strukturu dat, která obsahuje všechny detaily. V našem případě jsou to pole v kanálu příspěvků `user_money_receiver`, `description`, `flat_donation`, `date_created`, `is_public`, `allowed_users`.

Také je ale časté, že určitá data chybí, a je nutné provést řadu požadavků navíc, jako v našem případě chybějící komentáře k příspěvkům v kanále a profily uživatelů.

1.2.5 Ošetřování chyb

Zatímco REST API vrací typ chyby v HTTP stavovém kódu a případně více informací v odpovědi, GraphQL API na požadavky obsahující chybu v dotazu, nebo při vyhození chyby na serveru odpoví pomocí stavového kódu 200, který značí, že je vše v pořádku, chyby s více informacemi pak nahlásí server v odpovědi pomocí pole `errors`. Např. při neutorizovaném uživateli, ale vrací server klasický stavový kód 401.

1.2.6 Schéma a typová kontrola

Součástí GraphQL je silný typový systém. Všechny typy tohoto rozhraní jsou popsány v schématu pomocí GraphQL SDL. Toto schéma slouží jako definice toho, jaká data lze získávat ze serveru a jak volat Mutation a Subscription.

Něco obdobného je dokumentace REST API, která se tvoří např. pomocí externích nástrojů jako apiary.io, více informací je uvedeno v bakalařské práci Yevhena Kuzmovyche o backendu ElateMe [2].

Z analýzy pokrývající různé aspekty vyplývá, že nahrazením REST API za GraphQL můžeme výrazně optimalizovat komunikaci klientů se serverem bez zjevných nevýhod tohoto řešení.

1.3 Prototyp aplikace

Součástí bakalářské práce je návrh a implementace prototypu serverové a klientské části aplikace užívající GraphQL. Serverová část aplikace zajišťuje CRUD operace nad entitami, business logiku, včetně doporučování produktů a platebního systému, přes vystavené rozhraní, které je v tomto případě typu GraphQL. Tato část taky zodpovídá za ošetření a kontrolu transakcí.

Klientská část aplikace, v mém případě webová stránka typu SPA, je pouze vizuální obálka nad serverovou částí, se kterou komunikuje přes Internet.

Pro cíle mé práce není nutné implementovat celou aplikaci s veškerou business logikou, a proto pro přílišnou složitost a časovou náročnost vynechám část funkcionality, kterou současná verze disponuje, a to navrhování produktů, správu skupin uživatelů a darování s platebním a stavovým systémem. Jedná se totiž o části nutné pro reálnou verzi aplikace, ale mým cílem je pouze sestrojit prototyp, pomocí kterého porovnám REST API a GraphQL, a využití těchto funkcionalit nehledě na technologii většinou znamená jeden požadavek na server a následnou odpověď zpět s daty, kterých je ve zbytku aplikace i tak více než dost, takže by se projevila v těchto konkrétních případech spíš samotná implementace těchto náročnějších částí na serveru než vlastnosti REST API a GraphQL.

1.4 Funkční požadavky na aplikaci

„Funkční požadavky popisují chování produktu, výsledku nebo služby např. formou popsaného procesu nebo interakce s okolím“ [3]. Zde se jedná o popis funkcí, které naplňují požadavky na aplikace z hlediska nutných funkcí a vlastností.

F01 Přihlásit se pomocí FB.

F02 Nastavit svůj profil.

F03 Vytvořit přání pro sebe nebo přítele.

F04 Okomentovat přání.

F05 Smazat přání.

F06 Upravit přání.

F07 Zobrazit svůj profil s přáními.

F08 Zobrazit profil jiného uživatele.

F09 Zobrazit feed (kanál příspěvků).

F10 Zobrazit své přátelé a jejich narozeniny.

F11 Odhlásit se.

F12 Získat notifikace.

1.5 Nefunkční požadavky na aplikaci

„Nefunkční požadavky jsou doplňkem funkčních požadavků. Popisují další nezbytné vlastnosti potřebné vzhledem k prostředí a kontextu. Např. se jedná o požadavky na spolehlivost, bezpečnost, výkonnost, podporu během provozu atp.“ [3]

N1 Podpora: Google Chrome 57, Microsoft Edge 16, Firefox 58, Safari 11, iOS Safari 11, Chrome for Android 64.

N2 Serverová část bude s klientskou částí komunikovat přes HTTPS.

N3 Pro používání aplikace je nutné vlastnit facebookový účet.

N4 Aplikace bude podporovat zobrazení i na mobilech, tabletech a jiných médiích, které budou moci používat internetové prohlížeče vypsané v N1.

N5 Aplikace dokáže evidovat minimálně desetitisíce dárků a uživatelů.

1.6 Doménový model

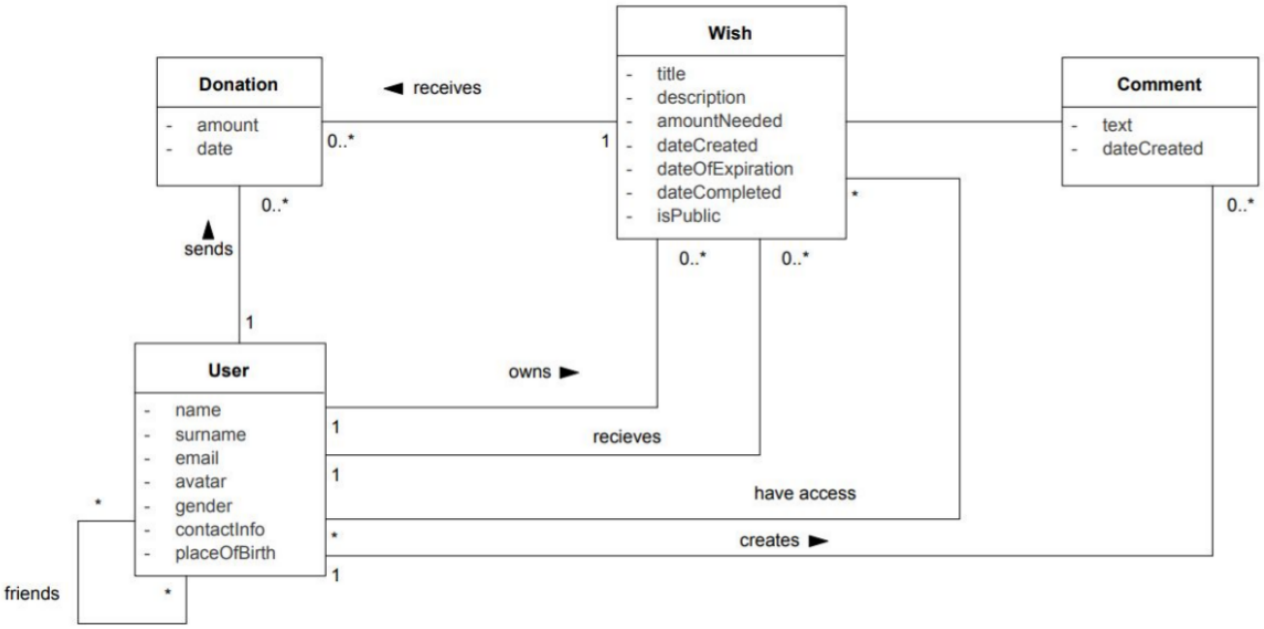
Doménový model slouží k navržení, ujasnění vztahů a struktury objektů v reálném světě, které budeme reprezentovat. Je zcela platformově nezávislý a pokrývá jak samotná data, tak i chování. Doménový model pro Wowee na obrázku 1.1 vypracoval Maksym Balatsko ve své bakalářské práci [4].

1.6.1 User

Jedná se o entitu reprezentující uživatele, tedy osobu registrovanou na Wowee. Tato entita má jako vlastnosti jméno, přijmení, e-mail, datum narození atd. Pro registraci a přihlašování je nutné vlastnit účet na Facebooku, ze kterého se rovněž získávají veškerá tato data stejně jako seznam přátel pro provázaní jednotlivých uživatelů i na Wowee.

1.6.2 Wish

Tato entita reprezentuje přání, které uživatel vytváří pro sebe nebo svého přítele. Přání obsahuje titulek, popis, potřebnou částku, datum expirace, datum dokončení a také informaci, zda je veřejné. Uživatel, který přání vytváří, má možnost nastavit, komu se přání zobrazí. Na toto přání pak ostatní uživatelé mohou do data expirace věnovat peníze nebo ho okomentovat.



Obrázek 1.1: Doménový model [4]

1.6.3 Comment

Uživatel může přidat k přání, na jehož zobrazení má opravnění, komentář, za jehož reprezentaci je zodpovědná právě tato entita. Uchovává pouze samotný text komentáře a datum vytvoření, jež je nastavováno automaticky.

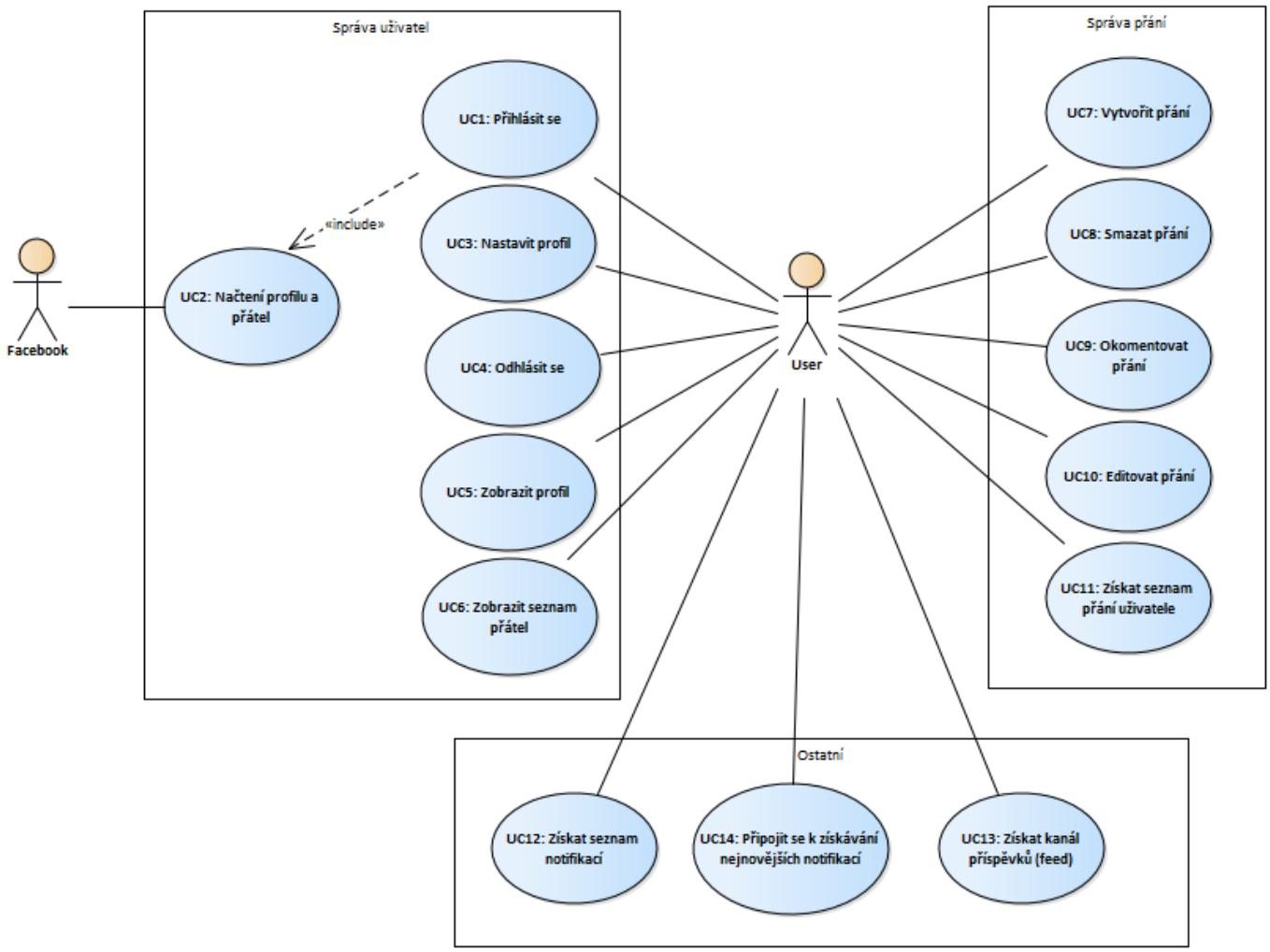
1.6.4 Donation

Poslední z entit reprezentující jednotlivý dar obsahuje konkrétní částku, datum platby a rovněž zdroj, odkud se vlastně vzala. Uživatel může darovat peníze na jakékoliv přání, nejen na ta, ke kterým má oficiálně přístup.

1.7 Případy užití

Připady užití definují interakci mezi systémem, uživatelem a případně dalšími subjekty. Jedná se o množinu akcí, která pokrývá dohromady funkcionalitu aplikace. Každá akce/případ užití by měl pokrývat jednu část funkcionality, jako třeba vytvoření přání.

Diagram případů užití je zobrazen na obrázku 1.2 a skládá se z několika různých modulů, které jsou ohraničeny do zvláštních systémů. Samotné případy



Obrázek 1.2: Diagram případů užití

užití jsou pak vloženy do nich. V rámci celého diagramu je několik různých aktérů, tedy rolí, přidělených vnějším entitám, které přímo komunikují se systémem. Základním aktérem je User, tedy uživatel, který je iniciátorem všech případů užití. Pak je tu aktér Facebook, který je v tomto případě zcela pasivní, slouží k přihlašování a k získání infomací o uživateli.

Na tyto případy užití se mapují funkční požadavky analyzované v kapitole 1.4, aby se zjistilo jestli jsou všechny pokryté případy užití. Toto mapování je zaneseno do tabulky 1.1, ve které jsou horizontálně zanesené případy užití. Vertikálně jsou zanesené funkční požadavky a jejich pokrytí případy užití, kde znak + značí, že funkční požadavek je pokryté právě daným případem užití.

Tabulka 1.1: Pokrytí funkčních požadavků případy užití

Požadavky	Případy užití													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F01	+	+												
F02			+											
F03							+							
F04									+					
F05								+						
F06										+				
F07					+						+			
F08					+						+			
F09													+	
F10						+								
F11				+										
F12												+		+

1.8 Analýza testování

Serverová i klientská část prototypu aplikace bude během vývoje pokryta automatickými testy, které budou kontrolovat správnou funkčnost implementace systému. Součástí budou jak unit testy, které kontrolují implementaci dílčích částí, tak i integrační testy, které kontrolují součinnost jednotlivých komponent. Serverová část bude pokryta systémovými testy, které testují funkčnost systému jako celku a přistupují k němu zcela přes vnější rozhraní.

Součástí mé práce je i otestování výkonnosti oproti současné implementaci. Toto bude provedeno řadou testů. Budou provedeny výkonnostní testy klientské části na rychlost načítání celé aplikace a domovské stránky s kanálem příspěvků včetně všech nutných dat. Jelikož je toto načítání ovlivněno rychlostí odezvy serveru, bude nejen proto otestováno GraphQL a REST API na

průměrný čas odpovědi při různém zatížení pro porovnání výkonu serverových částí.

Kapitola 2

Návrh

2.1 Použité technologie

2.1.1 Server

2.1.1.1 Nginx

Nginx je softwarový webový server s load managementem a reverzní proxy s otevřeným zdrojovým kódem. Zaměřuje se především na vysoký výkon a nízké nároky na paměť.

2.1.1.2 NodeJS

NodeJS je javascriptový serverový framework vyvinutý v roce 2009 na základě Google Chrome V8 JavaScript enginu. Používá event-driven, neblokující I/O model, je malý, efektivní a používá se často pro tvorbu API, webových stránek, ale používá se rovněž pro nástroje usnadnující vývoj jako bundling, a to i díky balíčkovému systému npm, který je v současnosti největší knihovnou softwaru na světě [5]. Javascript společně s NodeJS a Expressem, který představím níže, jsem si vybral díky několikaleté zkušenosti s touto platformou.

2.1.1.3 Express

Express je malá knihovna postavená nad NodeJS sloužící k jednoduššímu a rychlejšímu vývoji API a webových stránek, slouží jako mezivrstva mezi přímým rozhraním NodeJS pro tvorbu HTTP serveru, nabízí třeba routing atd.

2.1.1.4 PostgreSQL

PostgreSQL je open-source databázový systém. V současnosti se již používá na běžící verzi REST API, proto byl zvolen i v mém případě, a to pro získání

co nejvěrohodnějších výsledků při porovnávání výkonu.

2.1.1.5 GraphQL Yoga

GraphQL Yoga je knihovna pro provozování GraphQL serveru nad Express.js. Obsahuje v sobě Apollo Server i implementaci Subscriptions.

2.1.1.6 Prisma

Prisma je knihovna, která umožňuje přistupovat k databázi jako k GraphQL API. Slouží jako vrstva mezi databází a naší aplikací. Je to podobný princip jako ORM. Umožňuje nám jednoduše vytvořit databázové schéma pomocí GraphQL notace a pak k datům přistupovat přes GraphQL API. Obsahuje filtry, stránkování, Subscriptions a také lze provádět agregace nad daty. Ačkoliv díky novosti této technologie ještě není všechno dořešené a ne všechno je možné udělat tak efektivně, jak by mohlo být, je Prisma production-ready, běží v Dockeru a má v sobě již nástroje pro spuštění clusteru.

2.1.2 Klient

2.1.2.1 ES8

ECMAScript verze 8 je specifikace jazyka, která přináší mnoho zajímavých vlastností, jako třeba třídy, arrow functions, promises, generátory a mnoho dalšího. Javascript je pouhou implementací ECMAScriptu, a jelikož se používá mnoho různých a zastaralých verzí mnoha prohlížečů s vlastními interpretry Javascriptu, není zrovna aktuální verze ECMAScriptu rozšířená napříč zařízeními. Proto vznikly nástroje jako Babel, který kompiluje zápis v nové verzi ECMAScriptu do staršího, který je použitelný pro velkou množinu prohlížečů. A právě Babel budu používat i v tomto projektu.

2.1.2.2 React

React je malá javascriptová knihovna pro deklarativní vytváření komponent a jejich manipulaci v DOM. Byl vyvinut Facebookem pro vlastní potřebu a po několika letech byl v roce 2013 uvolněn jako open-source. Po počátečním nepochopení se ale brzy stal oblíbenou knihovnou s velkou komunitou. V Reactu je vyvinuta již současná webová verze Wowee, která komunikuje s REST API. Další výhodou je, že GraphQL bylo rovněž vyvinuto Facebookem, a to i pro využití v jejich webové verzi, která běží také na Reactu, a díky tomu fungují skvěle vedle sebe.

2.1.2.3 Redux

Redux je další malá javascriptová knihovna, která slouží k ukládání stavu aplikace. Umožňuje nám ukládat veškerá data do úložiště zvaného store. Celá

aplikace je pak obrazem storu. Stejně jako v případě Reactu vychází filozofie Reduxu z funkcionálního programování.

2.1.2.4 Apollo Client

Apollo Client je poslední důležítá javascriptová knihovna, kterou budu používat. Slouží jako GraphQL klient a integrace s Reactem je velmi jednoduchá. Obsahuje mnoho užitečných vlastností, jako cachování, jednoduché přihlášení k subscriptions, stránkování atd. Je vyvíjená komunitou, na rozdíl od konkurenční knihovny Relay, která je přímo od Facebooku a která zvláště ze začátku nebyla přívětivá k vývojářům i díky neúplné dokumentaci a použitelnosti pouze pro React. Prakticky umí obě knihovny více méně to samé, jen se liší některými koncepty.

Já jsem se rozhodl pro Apollo Client, neb s ním mám dobrou osobní zkušenost, má výtečnou dokumentaci, a na backendu se bude používat Apollo Server, který sice garantuje použitelnost i pro Relay, ale mít knihovny na klientu i serveru vyvinuté dohromady nemůže být na škodu.

2.1.3 Testování

2.1.3.1 Apache JMeter

Apache JMeter je open-source testovací nástroj. Byl vyvinut nejprve pouze pro testování webových stránek, ale dnes umožňuje mnohem více, včetně bezproblémového testování API. Ostatně jsou v něm již napsané testy pro REST API od Tomáše Grofka, což je primární důvod pro jeho volbu.

2.1.3.2 Selenium

Selenium je sada nástrojů pro automatizaci testů. Testy je možno spouštět v mnoha webových prohlížečích a také na mnoha platformách. Selenium využiji pro testování samotné webové aplikace, neb JMeter neobsahuje prohlížeč, a tedy nespouští ani JavaScript, ve kterém je celá aplikace napsaná.

2.1.3.3 Jest

Jest je testovací framework pro Javascript rovněž od Facebooku. Vybral jsem jej na základě osobní zkušenosti, snadné konfigurace, výkonu a podpory Reactu (zápisu v notaci JSX).

2.2 Přihlašování

Do testovací aplikace se půjde přihlásit jediným možným způsobem, a to pomocí facebookového účtu. Tato možnost sice diskriminuje uživatele, kteří facebookový účet nemají a kterých je v ČR pořád ještě mírná většina [6], ale

ostatním nabízí jednoduché přihlašování, a také díky tomu lze s uživatelovým svolením získat automaticky osobní data, jako je jméno, přijmení, profilový obrázek a seznam přátel, který je důležitý pro automatické propojení uživatelů jako přátel, díky němuž mohou v aplikaci jednoduše interagovat s přáními ostatních uživatelů Wowee, se kterými jsou již jako přátelé na Facebooku. Díky tomu zabere i první přihlášení pouhé 2 kliky a uživatel rovnou vidí přání ostatních uživatelů, se kterými je ve spojení na Facebooku.

2.2.1 OAuth 2.0

OAuth2 je protokol pro bezpečnou autentizaci jednoduchým a jednotným způsobem pro webové i desktopové aplikace [7]. Umožňuje autorizaci přístupu třetí strany k určitým datům od uživatele bez samotných přístupových informací. Vznikl na základě OpenID vyvíjeného pro Twitter a stal se z něj standard pro autentizaci, který mimo Twitteru, Amazonu, Googlu a jiných používá právě i Facebook.

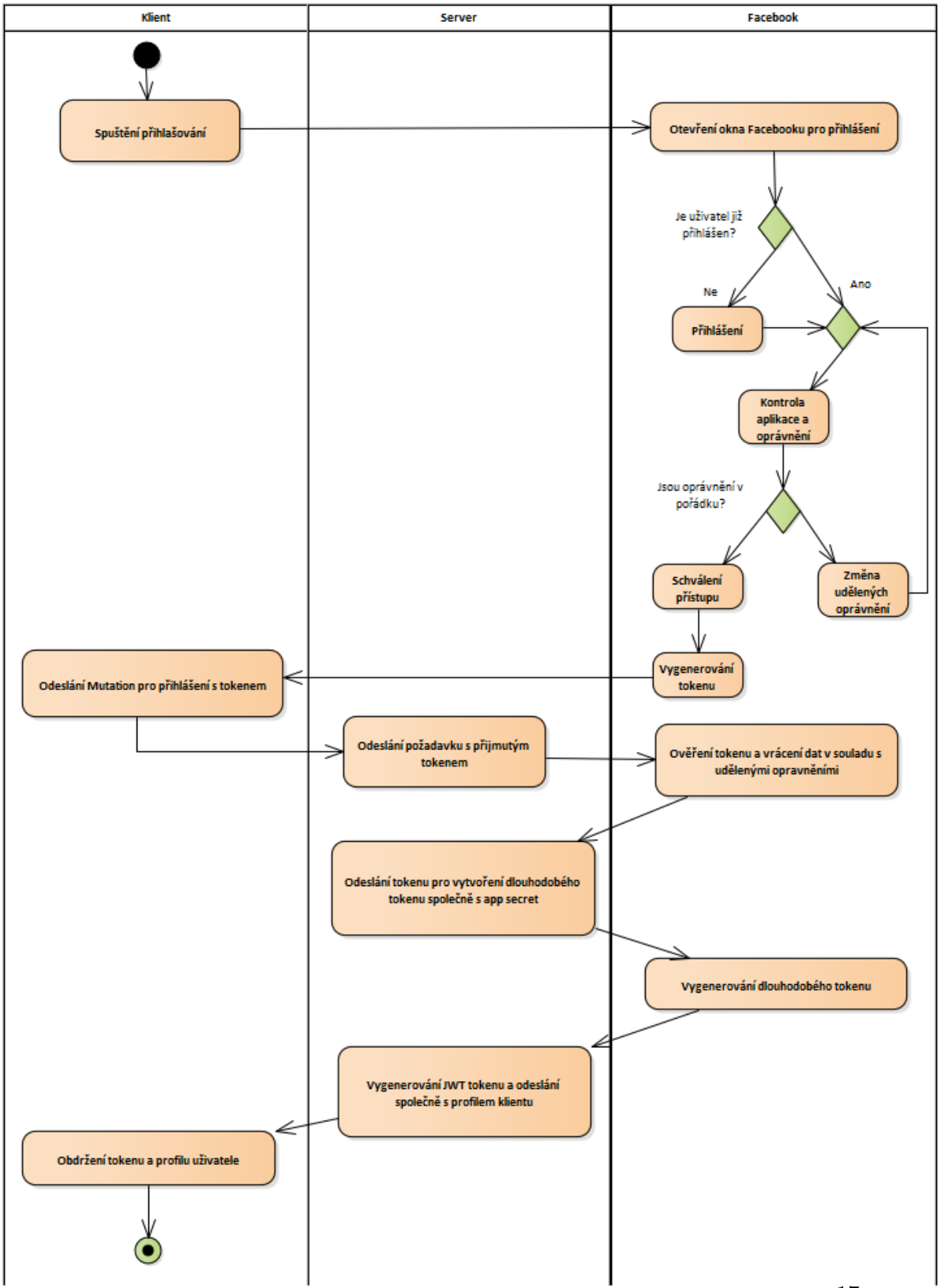
Pomocí tohoto protokolu nám uživatel umožní jednak jeho autentizaci, ale i přístup k jeho datům. Jak je naznačeno na activity diagramu 2.1, vše začíná získáním krátkodobého přístupového tokenu na klientu pro určitou aplikaci, ten se vygeneruje po kliknutí na Přihlášení a udělení práv k určitým datům, jako je v našem případě jméno, e-mail, přátelé a narozeniny. Tato práva může uživatel libovolně omezit na úkor funkčnosti aplikace. Tento token se pak odešle pro přihlášení na server, kde se použije pro přihlášení, získání potřebných dat o uživateli a vygenerování dlouhodobého tokenu pro aktualizaci dat.

Po ověření uživatele na základě facebookového profilu a aktualizaci dat odešle server zpět společně s profilem vygenerovaný token nutný pro další operace s API ve formátu JWT.

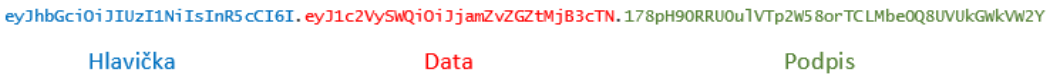
2.2.2 JWT

JSON Web Token je otevřený standard pro výměnu dat, která jsou v tomto případě uložena v podobě JSON objektu, který je digitálně podepsán pomocí JSON Web Signature (JWS). Tento typ tokenů oproti tradičnějším opaque tokenům obsahuje jak data, tak jejich podpis, takže je možné ho ověřit bez nutnosti volat server a taky není nutné držet samotná data v paměti na serveru.

Jak lze vidět na obrázku 2.2, každý token se skládá ze 3 částí oddělených tečkami, přičemž každá část je pak převedena do Base64url řetězce pro jednoduché použití, např. jako součást URL.



Obrázek 2.1: Login activity diagram



Obrázek 2.2: Detail JWT

2.2.2.1 Hlavička

Obsahuje informace o algoritmu použitém k vytvoření podpisu a o použití technologie JWT. V případě našich tokenů vypadá:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2.2.2.2 Data

Jedná se o JSON se samotnými daty. Není vhodné předávat větší množství informací, token pak může nepříjemně narůst na délce. V našem případě se jedná pouze o uživatelské id a datum expirace a vytvoření tokenu, takže tato část vypadá přibližně takto:

```
{
  "userId": "cjfodexruwp4y0a42hrxvu8oa",
  "iat": 1421234638,
  "exp": 14283219638
}
```

2.2.2.3 Podpis

Bez této části by token mohl být libovolně měněn a nebylo by možné ověřit jeho pravost. Takhle se zahashuje hlavička s daty pomocí soukromého klíče a obsah tokenu nelze zfalšovat. Většinou se pro vytvoření používá HMAC s SHA-256 či RSA s SHA-256.

2.3 GraphQL schéma

Každá GraphQL API obsahuje své schéma, které definuje strukturu API a specifikuje, jak má klient komunikovat se serverem a jaká data má očekávat. K jeho zápisu se používá speciálni notace GraphQL Schema Definition Language (SDL).

Základem GraphQL je silný typový systém a v tomto schématu tyto typy definujeme. Těchto typů existuje několik:

2.3.1 Skalární typy

Tyto typy jsou již předdefinované, jedná se o klasické typy jako Int, Float, String, Boolean či ID, který je sice implementovaný jako String, ale slouží jako unikátní identifikátor, který není pro člověka čitelný.

2.3.2 Výčtový typ

„Výčtový typ enum nám umožňuje vytvářet vlastní datové typy, které mohou nabývat pouze určitého omezeného množství hodnot. Každé z těchto hodnot poté odpovídá právě jedna instance výčtu“ [8].

V schématu je použit třeba pro definování zvoleného jazyka aplikace pod názvem `LANGUAGE` a nabývá hodnot `CZ` a `ENG`:

```
enum LANGUAGE {
  CZ
  ENG
}
```

2.3.3 Objekty

Základními komponentami schématu jsou typy objektů, které definují objekt a jeho vlastnosti pro výstup. Vlastnosti vrací vždy jeden ze skalárních či vlastních typů, nebo jeho seznam a mohou mít i vlastní argumenty specifikující např. formát dat.

V schématu jsou použity třeba pro definování přání, komentáře či uživatele, jehož zjednodušená podoba může vypadat např. takto:

```
type User {
  id: ID!
  name: String!
  birthday: DateTime!
  bankAccount: String
  email: String
  wishes(first: Int): [User]!
  language: LANGUAGE!
}
```

Definujeme typ `User`, který má různé vlastnosti, jako `id` vracející skalární typ `ID`. Znak vykřičníku znamená, že pole `id` nemůže nabýt hodnoty `null`. Poté

obsahuje povinně `name` typu `String`, nepovinně `bankAccount`, `email`. Povinně obsahuje také vlastnost `language` nabývající hodnot vlastního výčtového typu `LANGUAGE`. A poté i vlastnost `wishes` vracející `[User]!` , což znamená seznam uživatelů, který nenabývá nullové hodnoty, tedy v případě žádných přání vrátí prázdný seznam, a přijímací argument `first` skalárního typu `Int`.

2.3.4 Query, Mutation, Subscription

V rámci definovaných typů existují 3 zvláštní typy a to Query, který slouží k vystavení dat uživateli a je definován každou GraphQL API. Pak může být definován typ Mutation, ve kterém jsou uvedeny akce (změny dat), a typ Subscription, definující možné subscriptions, které průběžně zasílají nová data. Definujeme je podobně jako objektové typy, rovněž jednotlivé položky/vlastnosti mohou mít své argumenty. Bez těchto definovaných typů bychom nebyli schopni získat z GraphQL API žádná data, ani žádná data odesílat.

Ve zjednodušené podobě uvádím část schématu:

```
type Query {
  feed(filter: String, skip: Int, first: Int): [RestrictedWish!]!
  profile(id: ID): User!
}
type Mutation {
  postWish(
    title: String!,
    description: String!,
    amountNeeded: Float
  ): Wish!
  deleteWish(id: ID!): Wish
  login(token: String!): AuthPayload
}
type Subscription {
  newNotification: NotificationPayload
}
```

2.3.5 Interface

Pomocí Interface definujeme abstraktní typ, jehož implementace musí obsahovat vlastnosti definované v tomto abstraktním typu.

Definování typu interface a jeho implementace by mohla v naší aplikaci vypadat takto:

```
interface SurpriseReceiver{
  name: String!
}
```

```
type User implements SurpriseReceiver{
  id: ID!
  name: String!
}
```

2.3.6 Input

Argumenty nemusí být pouze skalárního typu, ale je možné definovat vlastní komplexnější datové struktury. K tomu slouží právě Input, který je velmi podobný objektům, ale jednotlivé vlastnosti nemohou mít definované vlastní argumenty. Tyto typy slouží pouze jako definice formátu vstupu a není je možné použít jako typ sloužící k výstupu. K tomu slouží právě objekty.

Input pro vytvoření skupiny může vypadat následovně:

```
input GroupCreateInput {
  color: String!
  users: UserCreateManyWithoutGroupsInput
}
```

Kompletní schéma je k nalezení v přiložených zdrojových kódech.

2.4 Databázová vrstva

Jak již bylo napsáno výše, rozhodl jsem se pro použití technologie Prisma jakožto databázové vrstvy. Ta nám umožní komunikovat s databází jako s další GraphQL službou. Prisma je nejen open-source knihovna, ale i společnost, která za touto technologií stojí, a která nabízí cloud pro tuto technologii.

Hlavní částí je definování databázového modelu pomocí notace GraphQL SDL. Jedinou dosud nezmíněnou zvláštností jsou *field constraint*, pomocí kterých lze přidat další sémantiku jako výchozí hodnoty atd.

```
type Wish {
  id: ID! @unique
  createdAt: DateTime!
  title: String!
  description: String!
  productUrl: String
  isPublic: Boolean @default(value: "false")
  postedBy: User!
  comments: [Comment]!
  amountNeeded: Float
  donations: [Donation]!
}
```

Na ukázce přání vidíme právě použití *field constraint* za návratovým typem pomocí znaku @ a názvu omezení. Pro `id` je typicky použité `@unique` zaručující unikátnost. U vlastnosti `isPublic` nastavujeme výchozí hodnotu pomocí `@default` přijímajícího argument `value` typu String. Podobně je definován i zbytek databázového schématu, který je k nalezení v přiložených zdrojových kódech.

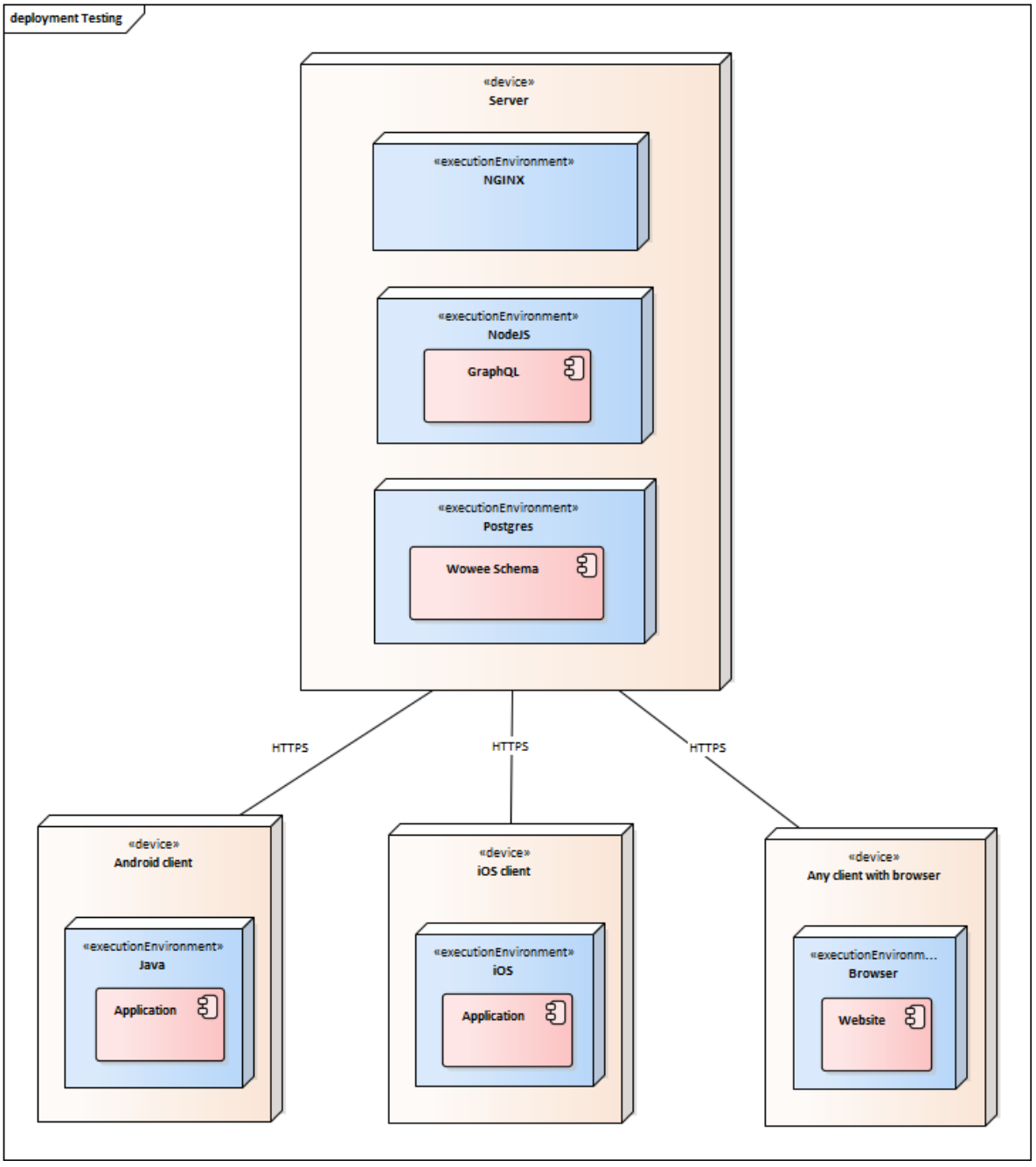
Na základě tohoto schématu pak Prisma vygeneruje jak opravdové databázové schéma použité pro skutečnou databázi, tak i standardní GraphQL schéma, se kterým pak můžeme pracovat v aplikaci.

Prisma umožňuje jednoduší vývoj, snadný přístup a manipulaci s daty, sdílení schémat databáze a aplikace či delegování resolverů aplikace na resolvery běžící Prisma GraphQL služby. Nevýhodou je to, že je to poměrně nová technologie, a s tím spjaté nedodělky. Dalo by se říci, že jde o alternativu k ORM určenou pro GraphQL, a ani tato technologie není schopná plně efektivního návrhu a dotazování, jakých by se dalo dosáhnout pomocí optimalizovaného návrhu a SQL dotazů. V současné chvíli se ale pracuje na mnoha zajimavých issues slibujících do budoucna možnost efektivnější komunikace a dotazů, jako je třeba složitější agregace.

2.5 Nasazení

Pro popsání „rozložení jednotlivých softwarových komponent na hardwarových zdrojích (uzlech) a jejich spolupráci“ [9] se používá diagram nasazení, který můžete vidět na obrázku 2.3. Naše konkrétní nasazení pro testovací účely bude velmi jednoduché, databáze a backend bude běžet pouze v jedné instanci od každého na jednom stroji. Pro vystavení rozhraní ven pro klienty jako webová aplikace bude použit Nginx jakožto reverse proxy odkazující na aplikaci a server hostující statické soubory klientské webové stránky.

Samozřejmě pro použití v produkčním prostředí by bylo toto nasazení nešťastné. Proto jsem uvažoval nad řešením, které by bylo mimo jiné horizontálně škálovatelné. Vzhledem k tomu, že GraphQL API lze jednoduše připravit pro běh jako Docker kontejner a databázová vrstva – Prisma – Docker sama používá, použil bych pro tyto účely nějaký container orchestrator jako Kubernetes, s tím, že pro statické soubory klienta by byl použit Nginx image.



Obrázek 2.3: Diagram nasazení

Kapitola 3

Implementace

3.1 Server

3.1.1 Struktura projektu

Struktura prototypu serveru je docela jednoduchá, ve složce *database* se nachází konfigurace a schémata pro Prismu zajišťující databázovou vrstvu. Ve složce *resolvers* najdeme samotnou logiku naší aplikace. Jedná se o resolvery, funkce implementující navržené GraphQL schéma. Resolvery jsou v projektu ještě rozděleny na *queries*, *mutations* a *subscriptions*. Vedle těchto složek najdeme i naše specifikované schéma, jádro serveru v *index.js*, kde se inicializuje Express server a propojuje se Prisma s resolvery a schématem, a také složky s integračními testy a pomocnými funkcemi.

3.1.2 Databázová vrstva

Jak již bylo rozebráno v kapitole 2, jako databázová vrstva je použita knihovna Prisma. Ta běží jako zvláštní GraphQL služba, ale nabízí i NodeJS API používající vystavené GraphQL API pro snadné propojení. Právě toto API je používáno v aplikaci a z důvodů přístupnosti je předáváno resolverům v rámci *contextu*.

3.1.3 Přihlašování

Pro přihlašování se používá hned několik knihoven z npm. Jedna je *fb*, které lze jednoduše předat *access token* a volat Facebook Graphl API. Tohle slouží k ověření uživatele a získání uživatelských dat pomocí předaného *access tokenu*. Ná základě výsledků se pak pomocí Prisma API vytvoří nový uživatel a pomocí knihovny *jsonwebtoken* se vytvoří nový JSON Web token obsahující uživatelský identifikátor. Tato knihovna pak slouží také k ověření těchto to-

kenů a získání identifikátoru. To celé je zabaleno do resolveru pro login, který vrací dle schématu token a profil uživatele.

3.1.4 Query

Query slouží k získávání dat, v rámci aplikace jich je několik, např. pro kanál příspěvků, pro seznam vlastních či cizích přání či pro profil uživatele. Tyto resolvery často pouze kontrolují přístupová práva a jinak se delegují na resolvery Prisma databázové vrstvy.

3.1.5 Mutation

Pomocí *Mutation* se zasílají nová data na server a mění se stávající záznamy. V případě Wowee tedy umožní vytvoření přání, jeho komentování či aktualizaci profilu uživatele.

3.1.6 Subscription

Zatímco *Query* slouží k získání dat v konkrétní chvíli, *Subscription* existuje pro průběžné zasílání aktuálních dat bez nutnosti volání dalších požadavků. V případě Wowee je *Subscription* vystaveno pro získávání aktuálních notifikací. Podpora pro *Subscription* je již vestavěná v rámci používané *graphql-yoga* a je implementovaná pomocí WebSocket serveru, který přihlašeným klientům odesílá nové notifikace přes otevřený socket.

3.2 Klient

3.2.1 Struktura projektu

Struktura projektu se řídí vesměs doporučeními v oficiální dokumentaci [10]. Soubory jsou oddělené podle jednotlivých částí aplikací, jako například vytvoření přání, feed, nastavení atd. K tomu existují sdílené komponenty a pomocné funkce. Tento způsob má zajistit přehlednost a dostatečnou granularitu, i když se kód aplikace rozroste, ale zase nevyžaduje zbytečné zanořování složek, které vede až k zbytečnému rozsložkování [11].

```
node_modules ..... instalované balíky
public ..... složka obsahující statické soubory hostované aplikací
    index.html ..... HTML šablona aplikace
    manifest.json ..... soubor s nastavením PWA
src
    common ..... sdílené komponenty pro celou aplikaci
        Wish.js
        ...
    utils ..... pomocné funkce
```

```
    ├── styles ..... SASS styly
    ├── feed ..... modul s feedem
    │   ├── index.js ..... vstupní soubor pro feed
    │   ├── feed.test.js ..... testy feedu
    │   └── ... ..... další soubory pro feed
    ├── createWish ..... modul pro vytvoření přání
    ├── ... ..... další moduly
    ├── registerServiceWorker.js ..... inicializace service workeru
    └── index.js ..... vstupní soubor aplikace
├── README.md ..... stručný popis projektu a návod na ovládání
├── package.json ..... soubor definující závislosti a skripty
└── yarn.lock ..... soubor ukládající instalované verze balíků
```

3.2.2 Typy komponent

React komponenty nacházející se v projektu můžeme rozdělit na dva typy lišící se svým účelem. Toto rozdělení a dodržování zásad napomůže znovupoužitelnosti komponent a jednoduchosti kódu [12].

3.2.2.1 Presentational Components

Prezentační komponenty slouží jako čistě vykreslovací komponenty. Neobsahují logiku ani získávání dat, převádí vstup, tzn. data a callbacky, pomocí props do výstupu ve formě elementů v Reactu. Jsou většinou bezestavové (stateless) a bez závislostí na zbytku aplikace jako na komunikační vrstvě s API nebo na routovacím systému.

Lze je zapsat často jako *functional components*, jak je ukázáno na zjednodušené verzi komponenty vykreslující přání:

```
const Wish = ({ id, title, description, postedBy, amountNeeded,
    comments }) => (
      <div>
        <h3>{`${postedBy.firstName} ${postedBy.lastName}`} wishes</h3>
        <UserImage user={this.props.wish.postedBy}/>
        <h2>{title}</h2>
        <p>{description}</p>
        {amountNeeded && (
          <p>Amount: {amountNeeded}</p>
        )}
        <CommentList comments={comments}>
        <AddCommentContainer wishId={id}/>
      </div>
  )
```

3.2.2.2 Containers

Kontejnery mají na starost funkční stránku komponent. Zajišťují získávání dat, odesílání dat z formulářů a logiku kolem ní. Rozhodují o vykreslení konkrétních prezentačních komponent a mají na starosti předávání potřebných dat pomocí props. Jsou nezřídka stavové a velice časté je vygenerování pomocí higher-order component [13], které lze vidět třeba na kontejneru pro uživatelova přání:

```
class MyWishesContainer extends Component {
  render() {
    if (this.props.myWishesQuery && this.props.myWishesQuery.loading)
        {
      return <Loading />
    }

    if (this.props.myWishesQuery && this.props.myWishesQuery.error) {
      return <Error />
    }

    const wishes = this.props.myWishesQuery.wishes

    return (
      <div className="container">
        <div>{wishes.map(wish => <Wish key={wish.id} wish={wish}
            showLink/>)}</div>
      </div>
    )
  }
}

export const MY_WISHES_QUERY = gql`
    {
        wishes {
            ...WishFragment
        }
    }
    ${WishFragment}
export default graphql(MY_WISHES_QUERY, { name: 'myWishesQuery' })
    (MyWishes)
```

3.2.3 Apollo

Jak již bylo zmíněno, pro komunikaci s API je na klientu použita knihovna *apollo-client*. Ta obstará jak dotazování a odesílání dat, tak jejich cachování, zjednoduší optimistic UI, subscriptions, stránkování, prefetching atd. Její nastavení včetně odesílání hlavičky s *access tokenem* vypadá přibližně takto:

```
const httpLink = createUploadLink({ uri: env.API_URL })

const middlewareAuthLink = new ApolloLink((operation, forward) => {
  const token = localStorage.getItem('accessToken')
  const authorizationHeader = token ? `Bearer ${token}` : null
  operation.setContext({
    headers: {
      authorization: authorizationHeader
    }
  })
  return forward(operation)
})

const httpLinkWithAuthToken = middlewareAuthLink.concat(httpLink)

const client = new ApolloClient({
  link: httpLinkWithAuthToken,
  cache: new InMemoryCache()
})
```

Nastavený Apollo Client se pak předá přes props obalující komponentě Apollo-Provider z balíčku *react-apollo*. Tato komponenta zprostředkovává funkcionalitu Apollo Client ostatním vnořeným komponentám bez nutností explicitního předávání.

```
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
```

Vnitřní komponenta, která pak například získává data z API, se napojuje na Apollo pomocí HoC *graphql*, ta pak na základě vstupních GraphQL dotazů vrací přes props údaje o stavu dotazu a data. Ukázka kódu pro získání vlastních přání od uživatele přes Apollo se nachází v 3.2.2.2

3.2.4 Přihlašování

Pro přihlašování přes Facebook Login se používá knihovna react-facebook-login, do níž jsem sám nepatrně přispěl [14]. Výstupem této knihovny je React komponenta, u které je důležité správně nastavit mimo callbacku, který se zavolá po přihlášení s *access tokenem*, i appID a *scope*, který určuje, ke kterým informacím o uživateli bude mít vygenerovaný *access token* přístup.

```
      <FBLogin
        appId={constants.APP_ID}
        scope="public_profile,email,user_birthday,user_friends"
```

```
        callback={this.responseFacebook}
      />
```

Tento *access token* od Facebooku se pak použije pro přihlášení přes LoginMutation, která vrací *access token* pro GraphQL API. Ukázka volání LoginMutation:

```
const LOGIN_MUTATION = gql‘
    mutation LoginMutation($token: String!) {
        login(token: $token) {
            token,
            user {
                id
                firstName
                lastName
                email
            }
        }
    }
‘
...
// V callbacku pro FBLogin
const result = await this.props.loginMutation({
      variables: { token: accessToken },
    })
```

3.2.5 Routování

Routování pomocí knihovny *react-router* obstarává v rámci naší aplikace navigaci. Zajišťuje vykreslování komponent podle URL, rovněž jako samotnou manipulaci s URL. V aplikaci je nutné napojit komponentu BrowserRouter podobně jako ApolloProvider. Pak můžeme použít komponenty pro samotné routování a nastavit adresy s odpovídajícími komponentami. V rámci adresy je možné použít proměnnou, jak lze vidět na */wish/:id*, ke které lze pak v komponentě přistupovat.

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route exact path="/wishes" component={privateRoute(MyWishes)} />
  <Route exact path="/create" component={privateRoute(CreateWish)} />
  <Route exact path="/wish/:id" component={WishDetail} />
  <Route exact path="/profile" component={privateRoute(Profile)} />
</Switch>
```

Až na domovskou stránku a detail přání jsou všechny adresy zabezpečené

pomocí *higher-order component* nazvané *privateRoute* [13], která kontroluje, zda je uživatel přihlášen. V kladném případě vykreslí komponentu, v záporném přesměruje uživatele na domovskou stránku, kde se může přihlásit.

```
export const privateRoute = function (WrappedComponent) {
  class PrivateComponent extends React.Component {
    render() {
      if (this.props.profileQuery && this.props.profileQuery.loading)
          {
        return <Loading</div>
      }

      if (this.props.profileQuery && this.props.profileQuery.error) {
        this.props.history.push('/');
        return <div>Not log in</div>
      }
      return <WrappedComponent {...this.props} />;
    }
  };

  return graphql(PROFILE_QUERY, { name: 'profileQuery'
      })(withRouter(PrivateComponent))
};
```

3.2.6 Optimistic UI

Optimistic UI je technika, která zrychluje reagování UI v případě formulářů apod. Při odeslání Mutation se předvypočítá překpokládaná odpověd serveru v případě kladného přijetí a na jejím základě se překreslí UI. Až dorazí opravdová odpověd serveru, tato dočasná odpověd se zahodí a UI se překreslí dle reálných dat.

V rámci aplikace se toto používá např. při odesílání komentáře, kdy se automaticky po odeslání jeví jako již odeslaný, i když jsou teprve data odesílána na server. V případě chyby se pak komentář odstraní a objeví se chybová hláška, jinak se UI po odpovědi serveru již nezmění.

3.2.7 PWA

„Progresivní webová aplikace (PWA) je relativně krátce používaný termín k označení nové metodologie vývoje softwaru. Na rozdíl od tradiční aplikace je progresivní webová aplikace jakýmsi hybridem běžné webové stránky a mobilní aplikace. Tento nový aplikační model má za cíl kombinaci vlastností nabízených většinou moderních webových prohlížečů s benefity user experience mobilních aplikací. Tento termín poprvé použil Google v 2015 k popsání aplikací využívajících výhod nových funkcí moderních webových prohlížečů, jako

jsou service workers a webové aplikační manifesty, které umožňují uživatelům vylepšit webové aplikace na prvotřídní aplikace v jejich nativním OS.“ [15]

Základními rysy je responzivita, progresivnost, nezávislost na připojení na internetu, zabezpečnost pomocí HTTPS nebo instalovatelnost na domácí obrazovku.

3.2.7.1 Manifest

Základem je manifest, podle kterého prohlížeč rozpozná, že jde o PWA. Jde o soubor ve formátu JSON uložený ve složce public a nalinkovaný do HTML šablony. Uvádí se v něm například jméno, odkazy na ikonky v různých velikostech, barvy aplikace apod.

```
{
  "short_name": "Wowee.cz",
  "name": "Wowee.cz",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    }
  ],
  "start_url": "./index.html",
  "display": "standalone",
  "theme_color": "#7e3f91",
  "background_color": "#ffffff"
}
```

3.2.7.2 Service Worker

Service workers poskytují programovatelnou síťovou proxy ve webovém prohlížeči k obsluze webových (HTTP) požadavků. Service worker se nachází mezi sítí a zařízením k doplnění obsahu. Service workers používají efektivní cachovací mechanismus a umožňují používat aplikaci i v případě odpojení od sítě [15]. Tak je to i v případě Wowee.cz, pomocí service workeru, který se nachází v *src/registerServiceWorker.js*, se cachuje celá aplikace a reaguje teda částečně i v případě výpadku internetu.

Kapitola 4

Testování

4.1 Server

4.1.0.1 Systémové testy

Serverová část je převážně o konfiguraci knihoven, GraphQL schématu a resolverech, které z velké částí fungují jako fasáda pro resolvery Prisma služby. Správnost dotazů zaručuje již GraphQL server, dotazy, které neodpovídají schématu, jsou automaticky zamítnuty, ať už se jedná o nesprávnost datového typu nebo např. chybějící argumenty. Proto jsem se rozhodl v rámci prototypu pouze pro systémové testy, pomocí kterých se ověří funkčnost jako celku oproti vystavenému rozhraní.

Během systémových testů se volá API tak, jak ji volá i klientská aplikace. Simuluje se proces, kterým bude API dotazováno včetně možných negativních průběhů. A mimo kladného či záporného provedení dotazů se validují i samotné výstupy.

Tyto systémové testy využívají knihovny *node-fb* pro vytvoření testovacích uživatelských profilů na FB a získání access tokenu pro následné volání GraphQL API. Pro komunikaci se samotnou API je použita knihovna *graphql-tester* zjednodušující psaní dotazů v GraphQL SDL. Samotné testování je pak provedeno pomocí testovacího frameworku *jest*.

4.1.0.2 Testy výkonosti

Pro porovnání API byla provedena sada testů na testování rychlosti odpovědi serveru při různém zatížení. Testy byly provedeny pomocí nástroje *Apache JMeter*, kterým se spouštělo dotazování na server z několika současných připojení pomocí Thread Group. Každé připojení pak spouštělo v cyklu dotaz pouze na seznam vlastních přání. Na základě změřených dat se pak vypočítala

průměrná doba odpovědi, propustnost a APDEX, což je údaj vyjadřující uživatelskou spokojenost dle vzorce $\frac{s+\frac{t}{2}}{c}$, kde s udává počet požadavků, které byly provedeny v uspokojivém čase, což je v tomto případě 500 ms, t udává počet požadavků splněných v tolerovaném čase, který je ve výpočtu nižší než 1 500 ms, a c udává celkový počet provedených požadavků.

Pro porovnání byla tato sada testů se stejnými parametry spuštěna i na REST API. Obě API byly nasazeny na testovací servery ve shodné konfiguraci, pro co nejmenší odchylku v měření. Výsledky byly převedeny do tabulky 4.1, ve které jsou uvedeny výše zmíněné údaje: APDEX, průměrný čas odpovědi na požadavek v ms a u vysoké zátěže i propustnost v požadavcích za sekundu. Z porovnání těchto údajů vychází implementace v GraphQL, hlavně v případě vyšší zátěže, o dost lépe, což je částečně dáno tím, že REST provádí více logiky spjaté s širší funkcionalitou. Toto se potvrdilo i po otestování obou API na jinných endpointech, viz příloha B.

Tabulka 4.1: Testování výkonu API

Zatížení	Nízké		Střední		Vysoké		
Počet připojení	1		10		100		
Celkem dotazů	10		100		1000		
	APDEX	Průměr	APDEX	Průměr	APDEX	Průměr	Propustnost
GraphQL	0.950	118 ms	0.975	290 ms	0.373	1205 ms	49.76 r/s
REST	0.750	496 ms	0.435	1254 ms	0.003	12771 ms	6.98 r/s

4.2 Klient

4.2.0.1 Unit a integrační testy

Jelikož se jedná pouze o prototyp, byly opět pomocí *jest* vytvořeny unit testy pouze pro kritické části kódu, jako manipulace se storem, s daty atd. Nejsou otestované samotné komponenty, které většinou převádí data do DOM a u nichž by byla tvorba testů obsahově náročnější, testovala by pouze vykreslování dat a správnost vygenerované DOM struktury.

4.2.0.2 Systémové testy

Systémové testy nebyly pro klientskou část vytvořeny tak důkladně jako pro serverovou část, jedná se spíše o smoke testy. Ověřují pouze základní funkčnost, jako je vyrendrování aplikace, funkčnost přihlášení a následné vykreslení kanálu příspěvků po přihlášení. Jsou vytvořeny pomocí nástroje Selenium, což je nástroj pro automatické testování webových aplikací. A na základě nich jsou pak implementovány i testy výkonnosti.

4.2.0.3 Testy výkonosti

Klient byl otestován na rychlost rozhraní a načítání potřebných dat. Byla provedena sada 10 testů měřících rychlost přihlášení a následného načtení domovské stránky včetně potřebných dat, což zahrnuje kanál příspěvků včetně profilů uživatelů a komentářů, 5 posledních vlastních přání a seznam přátel s jejich narozeninami.

Tyto testy pak byly spuštěny i na aktuální verzi aplikace využívající REST. Stejně jako v případě testování výkonosti serveru byly testy spouštěny na stejném stroji ve shodném prostředí, rovněž API pořád běžely na serverech ve shodné konfiguraci s nulovou zátěží. Výsledky pak byly převedeny do tabulky 4.2, ze které můžeme vyčíst, že verze používající GraphQL server je rychlejší o více než 1.8 s, což je dáno nepatrně rychlejší odpovědí, ale hlavně efektivní komunikací s API.

Tabulka 4.2: Testování výkonu klienta

Typ API	Průměr	Počet požadavků na API
REST	4694.67 ms	15
GraphQL	2827.68 ms	6

Závěr

V této práci jsem se zabýval porovnáním typů API, a to REST a GraphQL, a analyzoval použití GraphQL na již existující službě Wowee. Nejprve jsem srovnával tyto dvě technologie v různých aspektech, jako jsou způsoby získávání a odesílání dat, ošetřování chyb či síťové požadavky. Pak jsem se věnoval analýze prototypů serverové a klientské části, jejichž návrh a implementace byla rovněž cílem práce. Detailněji jsem se zaobíral funkčními a nefunkčními požadavky na prototyp aplikace, doménovým modelem, případy užití a v neposlední řadě i analýzou testování výkonu, které jsem prováděl pro porovnání efektivity již výše zmíněných řešení.

Na základě analýzy se po rozebrání použitých technologií mohlo přistoupit k návrhu struktury projektu, databázovému schématu, přihlašováni a komponent. Dle tohoto návrhu jsem pak pokračoval v implementaci serverové části GraphQL API a prototypu klienta ve formě webové aplikace. V práci jsem rovněž detailněji popsal samotnou technologii GraphQL a její používání.

Tento prototyp byl pak nasazen na testovací server a otestován oproti současné verzi aplikace řadou výkonnostních testů. Těmito testy se ukázalo, že pomocí GraphQL je opravdu možné navrhnout efektivnější komunikaci klienta se serverem a díky tomu např. signifikantně zrychlit prvotní načtení aplikace o cirka 40 %. Rovněž se ukázalo, že si GraphQL API výkonnostně nevede oproti REST API vůbec zle a zvládá lépe vyšší zátěž. Toto je avšak dané i samotným rozsahem implementace, jelikož v prototypu GraphQL API není zahrnuta veškerá funkcionalita REST verze např. možnost nastavit viditelnost přání pro jednotlivé uživatele.

Povedlo se mi splnit cíle kladené na tuto práci a díky tomu, že jsem ověřil, že implementace pomocí GraphQL může být opravdu rychlejší, je možné na základě prototypů v budoucnosti přejít k implementaci reálné produkční verze aplikace, která bude podstatně rychlejší než současná verze využívající REST.

V této verzi bych se ale vyhnul použití knihovny Prisma pro databázovou vrstvu, a raději bych zvolil přímější způsob komunikace s databází. Pro prototypování se Prisma hodí skvěle, ale během implementace jsem se několikrát potýkal se změnou rozhraní bez zpětné kompatibility i v rámci aktualizací minor verzí a řadou menších chyb, což bylo pro mne docela frustrující, a po hlubší zkušenosti s ní si nemyslím, že je zatím vhodná pro produkci. Pomocí přímější metody by se rovněž dalo docílit daleko efektivnější komunikace s databází a díky tomu celkovému zrychlení API.

Literatura

[1] *Optimizing API against latency [online].* Polidea — Design and development studio delivering digital products, červen 2018, [cit. 2018-04-17]. Dostupné z: `https://www.polidea.com/blog/Optimizing_API_against_latency/`

[2] Kuzmovych, Y.: *ElateMe - Backend.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[3] Jan Doležal, J. K.: *Projektový management v praxi.* Grada, 2016, ISBN 978-80-247-5693-6, 82 s.

[4] Balatsko, M.: *Elateme - Project management and Advert server.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[5] Marinos, A.: *npm now the largest module repository [online].* Resin.io, červenec 2014, [cit. 2018-04-17]. Dostupné z: `http://alexandros.resin.io/npm-now-the-largest-module-repository/`

[6] *Facebook and Instagram user demographics in Czech Republic [online].* Napoleon cat, srpen 2017, [cit. 2018-04-17]. Dostupné z: `https://napoleoncat.com/blog/en/facebook-instagram-user-demographics-in-czech-republic-july-2017/`

[7] *OAuth 2.0 [online].* OAuth 2.0, [cit. 2018-04-13]. Dostupné z: `https://oauth.net/`

[8] *Java (19) - Enum [online].* Algoritmy.net, [cit. 2018-04-17]. Dostupné z: `https://www.algoritmy.net/article/30320/Enum-19`

[9] Alena Buchalcevová, L. P., Jarmila Pavlíčková: *Základy softwarového inženýrství.* Praha: Vysoká škola ekonomická, 2017, ISBN 80-245-0346-8, 202 s.

[10] *File Structure [online].* ReactJS Documentation, [cit. 2018-04-24]. Dostupné z: https://reactjs.org/docs/faq-structure.html

[11] Mangin, A.: *How to better organize your React applications? [online].* Medium, duben 2016, [cit. 2018-04-24]. Dostupné z: https://medium.com/@alexmngn/how-to-better-organize-your-react-applications-2fd3ea1920f1

[12] Abramov, D.: *Presentational and Container Components [online].* Medium, březen 2015, [cit. 2018-04-20]. Dostupné z: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

[13] *Higher-Order Components [online].* ReactJS Documentation, [cit. 2018-04-10]. Dostupné z: https://reactjs.org/docs/higher-order-components.html

[14] *Fix autoloading [online].* Github, [cit. 2018-04-20]. Dostupné z: https://github.com/keppelen/react-facebook-login/commit/fd766d0550c8ba3919c09738a022ef7085767082

[15] *Proč a jak psát progresivní webové aplikace [online].* Ackee.cz, leden 2017, [cit. 2018-04-16]. Dostupné z: https://www.ackee.cz/blog/proc-a-jak-psat-progresivni-webove-aplikace/

Příloha **A**

Seznam použitých zkratek

APDEX Application Performance Index

API Application Programming Interface

CRUD Create, Read, Update, Delete

DOM Document Object Model

GUI Graphical user interface

GraphQL SDL GraphQL Schema Definition Language

HMAC Keyed-hash Message Authentication Cod

HS256 HMAC-SHA256

HoC Higher-order component

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

JSX JavaScript XML

ORM Object-relational mapping

REST Representational State Transfer

RSA Rivest–Shamir–Adleman

SPA Single-page application

SHA256 Secure Hash Algorithm 2

XML Extensible markup language

Příloha B

Porovnání výkonu API

Tabulka B.1: Výsledky testů na získání kanálu přání

Zatížení	Střední		Vysoké		
Počet připojení	10		100		
Celkem dotazů	100		1000		
	APDEX	Průměr	APDEX	Průměr	Propustnost
GraphQL	0.940	336 ms	0.132	1973 ms	35.58 r/s
REST	0.040	2567 ms	0.002	23327 ms	3.92 r/s

Tabulka B.2: Výsledky testů na získání detailu přání

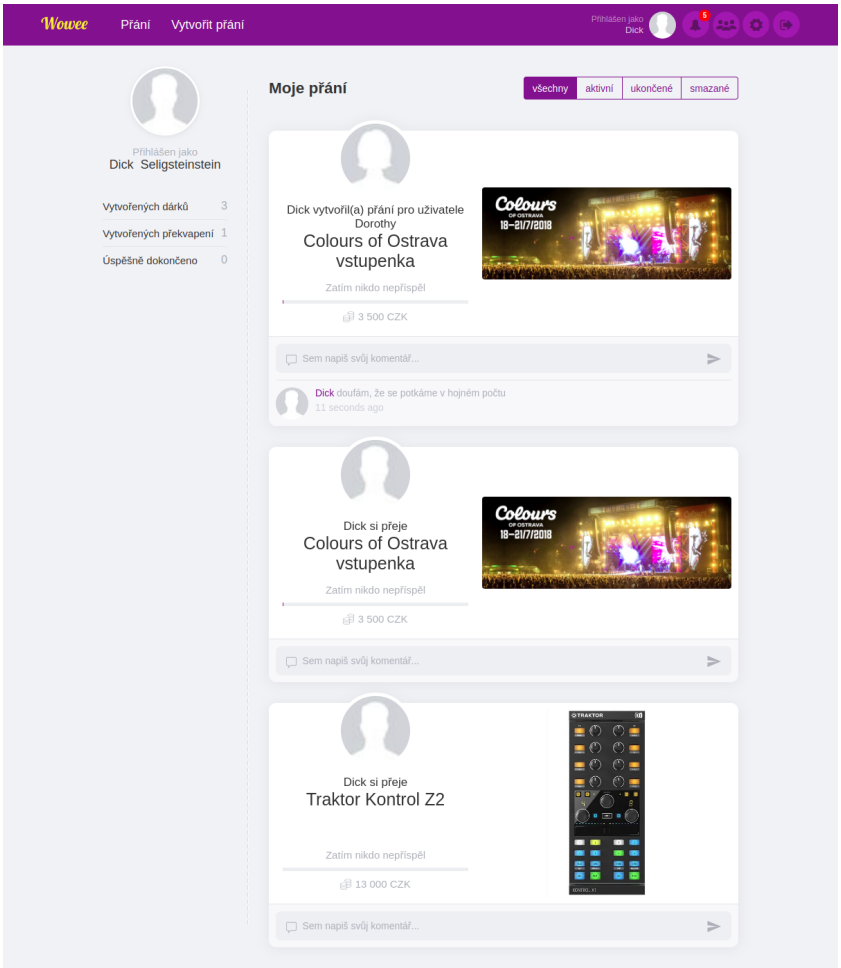
Zatížení	Střední		Vysoké		
Počet připojení	10		100		
Celkem dotazů	100		1000		
	APDEX	Průměr	APDEX	Průměr	Propustnost
GraphQL	0.995	247 ms	0.569	956 ms	56.63 r/s
REST	0.760	511 ms	0.057	3184 ms	23.37 r/s

Tabulka B.3: Výsledky testů na získání profilu uživatele

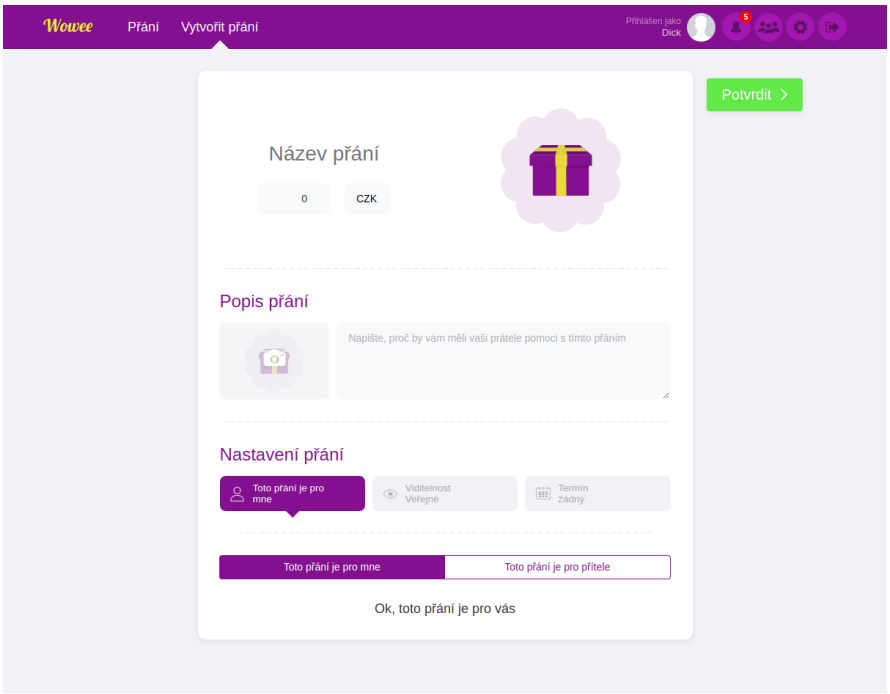
Zatížení	Střední		Vysoké		
Počet připojení	10		100		
Celkem dotazů	100		1000		
	APDEX	Průměr	APDEX	Průměr	Propustnost
GraphQL	0.980	314 ms	0.122	1877 ms	37.67 r/s
REST	0.990	246 ms	0.519	1064 ms	47.74 r/s

Příloha C

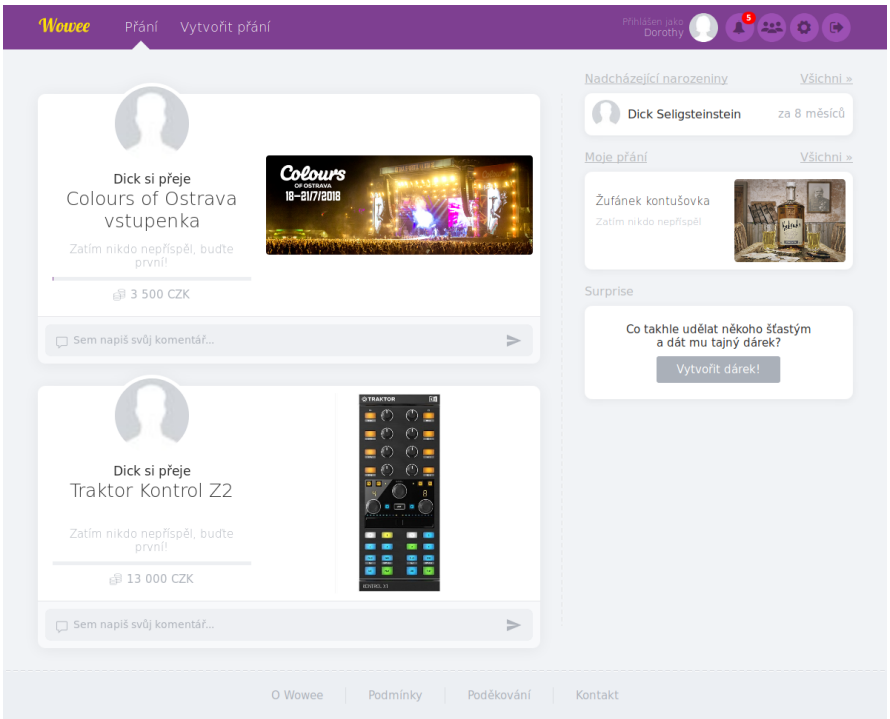
Snímky obrazovky aplikace



Obrázek C.1: Seznam vlastních přání



Obrázek C.2: Vytvoření přání



Obrázek C.3: Kanál příspěvků (feed)

Příloha D

Obsah přiloženého CD

- `src`
  - `client` ..... zdrojové kódy implementace klienta
  - `server` ..... zdrojové kódy implementace serveru
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `installation_guide.txt` ..... návod na instalaci a spuštění
- `readme.txt` ..... stručný popis obsahu CD
- `thesis.pdf` ..... text práce ve formátu PDF