VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

BACKEND NOTIFIKAČNÍHO SYSTÉMU PRO IOS

BACKEND OF AN IOS NOTIFICATION SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN MIHÁL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VLADIMÍR BARTÍK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Mihál Martin**
Program: Informační technologie
Název: **Backend notifikačního systému pro iOS**
Backend of a iOS Notification System
Kategorie: Informační systémy

Zadání:

1. Seznamte se s jazykem Python 3, problematikou REST API a GraphQL.
2. Analyzujte požadavky na backend notifikačního systému zahrnující systém pro přihlašování včetně odpovídající databáze, frontu zpráv, odesílání požadavků do APNS, agregaci notifikací a REST API/GraphQL zpracovávající jednotlivé požadavky.
3. Navrhněte řešení zahrnující požadavky z bodu 2.
4. Implementujte navržený backend a porovnejte řešení pomocí REST API a GraphQL.
5. Otestujte vytvořené řešení.
6. Zhodnoťte dosažené výsledky a diskutujte další možné pokračování tohoto projektu.

Literatura:

- Bielik, D.: Design and implementation of an iOS notification system. Bakalářské práce, FI MU Brno, 2018.
- Lacko, L.: Vývoj aplikací pro iOS. Computer Press, 2018. 480 s. ISBN 978-80-2514-942-3.
- Porcello, E., Banks, A.: Learning GraphQL. O'Reilly Media, Inc, USA, 2018. 175 s. ISBN 978-14-9203-071-3.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1-3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Bartík Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

Abstrakt

Cieľom tejto práce je navrhnúť a implementovať back-endovú časť notifikačného systému, ktorý dokáže prostredníctvom webovej požiadavky prijať notifikáciu zo softvéru na odoslať ju iOS zariadenie. Systému posiela notifikácie na iOS zariadenia prostredníctvom notifikačného serveru Apple. Systém taktiež poskytuje kompletný manažment účtu, ktorý používateľom umožňuje agregáciu notifikácií pod takzvané služby. Tie reprezentujú softvér používateľa, z ktorý notifikácie posiela. Systém komunikuje s iOS aplikáciou predstavujúcou používateľa cez REST alebo GrahpQL webové služby.

Abstract

The aim of this thesis is to design and implement backend of an iOS notification system, which can receive a notification through web request and send it to an iOS device. System sends notifications to iOS devices through the Apple notification server. System also provides a complete account management, which allows users to agregate notitifications into so-called services. These represent a user's software, which sends notifications. System communicates with the iOS application representing user through REST or GraphQL web service.

Kľúčové slová

Python, REST, GraphQL, PostgreSQL, Redis, RabbitMQ, Docker, APNs, iOS

Keywords

Python, REST, GraphQL, PostgreSQL, Redis, RabbitMQ, Docker, APNs, iOS

Citácia

MIHÁL, Martin. *Backend notifikačního systému pro iOS*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vladimír Bartík, Ph.D.

Backend notifikačního systému pro iOS

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Vladimíra Bartíka, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Martin Mihál
28.05.2020

Poďakovanie

Rád by som poďakoval vedúcemu mojej práce, pánovi Ing. Vladimírovi Bartíkovi, Ph.D., za jeho ochotu, pomoc, a že mi umožnil takúto špecifickú tému práce.

Obsah

Kapitola 1

Úvod

V dnešnom digitálnom svete prebieha cez internet množstvo procesov. Developeri internetového softvéru sa snažia zaručiť jeho stabilitu. K udržaniu stability je potrebný nejaký druh monitoringu. Developeri sledujú udalosti vo svojom softvéri a môžu napríklad zhotovovať štatistiky. V prípade chyby alebo výpadku musia okamžite zasiahnuť a problém odstrániť. Na tieto účely im slúži monitoring. Monitorovať softvér je možné viacerými spôsobmi, ale jeden konkrétny spôsob v dnešnej dobe ešte nie je bežný alebo veľmi známy. Tým je monitoring prostredníctvom mobilného zariadenia.

Mobilné zariadenie je forma prenosného počítača, ktorý má obvykle prístup na internet. Práve to ho vytvára atraktívnym riešením na monitoring udalostí v softvéri. Je schopné cez internet prijímať dáta. V dnešnom svete nie žiadna veľká služba, ktorá by tvorila pokročilý monitorovací softvér na mobilnom zariadení. Práve to viedlo k vytvoreniu tejto práce.

Táto práca dizajnuje a implementuje notifikačný systém, ktorý slúži na prenos notifikácie (dát), respektíve udalostí. Developeri softvéru môžu prostredníctvom tohto systému posielať dáta na svoje mobilné zariadenie. To im umožňuje monitorovať udalosti, ktoré v ňom prebiehajú, a teda mať okamžitú spätnú väzbu. Takýto princíp zvyšuje produktivitu a prehľad developera, pretože o všetkom môže okamžite vedieť a následne aj konať. Táto práca neobsahuje len jednoduchý notifikačný systém, ale rozširuje ho o takzvané služby. Jedna služba predstavuje nejaký softvér, ktorý patrí developerovi. Softvérov môže byť viac, rovnako aj služieb. Služba rozlišuje softvér od ďalšieho softvéru (služby). Každá notifikácia patrí pod nejakú službu a uľahčuje developerovi prehľadnosť v rámci jednotlivých notifikácií. Tento notifikačný systém obsahuje ešte jednu praktickú funkcionalitu, jedná sa o prihlasovací systém. Ten umožňuje developerom posielať notifikácie na viaceré zariadenia.

Prvá kapitola tejto práce je teoretická, zaoberá sa výmenou dát cez internet. Druhá vymenováva a dizajnuje potreby tohto notifikačného systému. V tretej je konkrétna implementácia súčastí systému, zvolené technológie a ich komunikácia.

Kapitola 2

Webové služby a jazyk Python

Jednotlivé elektronické zariadenia, služby, potrebujú medzi sebou navzájom komunikovať. Webové služby poskytujú možnosť takzvanej machine-to-machine komunikácie cez internet. Takáto služba má univerzálne rozhranie, ktoré dokáže spracovať počítač a je zároveň zrozumiteľné pre ľudí. Obe komunikujúce zariadenia si po sieti vymieňajú dáta nad týmto rozhraním, čo so sebou prináša obrovské možnosti pre tvorbu komerčného softvéru.

V dnešnej dobe používa takmer každý nový softvér takúto službu, pretože to je pre množstvo jej funkcií nevyhnutné. Táto práca nie je výnimkou. Existuje niekoľko druhov webových služieb.

V tejto práci sú použité a popísané dve takéto služby - Representational State Transfer (REST) a GraphQL. Dôvodom použitia týchto dvoch služieb nie je potreba ich vzájomnej kombinácie benefitov, ale ukážka použitia oboch variant pre vzdelávacie účely. Ďalším dôvodom pre ich výber je moja predchádzajúca skúsenosť s REST a aktuálna popularita GraphQL. Každá má svoje výhody a nevýhody. Samozrejme, aspoň jedna webová služba je v tejto práci potrebná na vytvorenie požadovaného softvéru.

2.1 Python

Python je interpretovaný, multi-paradigmový jazyk. Interpretovaný jazyk znamená, že sa v ňom zdrojový kód vykonáva priamo a neprekladá sa do strojového kódu. Kód vykonáva interpreter, ktorý je súčasťou Pythonu. Preto je potrebné mať Python nainštalovaný na počítači, kde sa má kód vykonať. Jeho multi-paradigmové vlastnosti podporujú viaceré štýly programovania. Podporuje teda objektovo orientované, štrukturované a funkcionálne programovanie. Je dynamicky typovaný, čo znamená, že typy sa nedeklarujú. Obsahuje takzvaný garbage collector, ktorý uvoľňuje už nepoužívanú pamäť. Má bohatú platformovú podporu (Windows, Linux, Mac).

Python je univerzálny jazyk, obsahuje prostriedky na programovanie takmer všetkých druhov softvéru. Syntax jazyka je veľmi jednoduchá a intuitívna, čo podporuje rýchle vyvíjanie softvéru ako aj čitateľnosť kódu. Jazyk Python je vhodný na tvorbu skriptov, ale aj back-endových serverov, príkladom je aj táto práca. Python má v sebe zabudovaných veľa užitočných knižníc, ktoré pokrývajú potreby programátora. Na ďalšie špecifické potreby je k dispozícií množstvo knižníc tretích strán. Na potreby notifikačného servera je jazyk Python viac než dostačujúci.

2.2 REST

Representational State Transfer (REST) je architektonický štýl, ktorý definuje požiadavky webovej služby. Systémy používajúce REST čítajú a prenášajú dáta pomocou ich textovej reprezentácie, ktorej štruktúra je dopredu a univerzálne definovaná. Použitie technológie REST je veľmi populárne v back-endových systémoch, s ktorými komunikuje front-end. V kontexte tejto práce je možné za front-end považovať aj iOS. Služba je považovaná za RESTful, ak spĺňa nasledujúce požiadavky.

Klient - server

Rozdeľuje úlohy klienta a servera. Klient nemusí riešiť narábanie s dátami, ukladanie dát, zabezpečenie. To umožňuje škálovateľnosť servera bez ohľadu na klienta. Taktiež umožňuje klientovu nezávislosť na platforme. Vzniknú teda dve rozdielne prostredia, ktoré majú svoje vlastné úlohy. Tento princíp urýchľuje vývoj jednotlivých strán, keďže každá sa bude zaoberať svojimi úlohami.

Bezstavovosť

Jednotlivé požiadavky klienta na server musia obsahovať všetky potrebné informácie na to, aby server vedel túto požiadavku spracovať. Klient dostane odpoveď presne podľa požiadavky, ktorú poslal. Pri ďalšej požiadavke toho istého klienta s inými parametrami bude odpoveď servera iná. Odpoveď servera nereflektuje predošlé požiadavky a ich stavy, ale len tu aktuálnu. Aktuálny stav požiadavky si ukladá klient a nie server.

V niektorých prípadoch, ako napríklad autorizácia, potrebuje server vedieť s kým komunikuje. Pretože pri niektorých operáciách server potrebuje poznať klienta, pod ktorým tieto operácie prebiehajú. V tomto prípade pošle klient potrebné údaje a odpoveď servera bude okrem iného obsahovať aj identifikátor relácie. Pri nasledujúcich požiadavkách, pri ktorých je teda potrebné byt prihlásený, pošle klient tento identifikátor relácie serveru. Ten na základe tejto informácie identifikuje daného používateľa a dokáže spracovať požiadavku a zároveň konkrétne danému klientovi odpovedať.

Táto vlastnosť prináša serveru lepšiu škálovateľnosť a menšie využitie zdrojov, keďže si nemusí ukladať jednotlivé stavy, ale môže znamenať aj väčšiu záťaž, pretože server dostane viac požiadaviek.

Využitie vyrovnávacej pamäte

Dáta z odpovede servera na dané požiadavky môžu byť znovupoužiteľné. O tom rozhoduje server a pri odpovedi tieto dáta do týchto dvoch kategórií patrične označí. V prípade znovupoužiteľnosti si ich klient uloží do vyrovnávacej pamäte a môže ich miesto ďalších požiadaviek použiť.

Táto vlastnosť umožňuje vynechať niekoľko požiadaviek, čo má ako pozitívny následok zníženie záťaže servera a zrýchlenie vykonávania operácií u klienta, ale negatívne vplýva na spoľahlivosť, pretože dané dáta nemusia byt identické skutočnej odpovedi servera.

Jednotné rozhranie

Vlastnosť ktorá oddeľuje architektonický štýl RESTu od ostatných. Rozkladá a zjednodušuje jednotlivé časti architektúry, umožňuje ich nezávislý vývoj. Zjednodušuje viditeľnosť a uľahčuje interakciu. Má nasledujúce podmienky.

Identifikácia zdroja

Jednotlivé zdroje sú identifikované v požiadavkách pomocou Uniform Resource Identifier (URI). Samotné zdroje sú odlišné od reprezentácie, aká je vrátená klientovi. Reprezentácia dát v odpovedi pre klienta je nezávislá od interného uloženia zdroja na serveri.

Manipulácia zdrojov cez ich reprezentáciu

Ak klient disponuje reprezentáciou zdroja so všetkými potrebnými informáciami, tak má byť schopný modifikovať alebo zmazať stav tohto zdroja. Použitím URI a metódy HTTP dokáže klient manipulovať so zdrojom.

Samo popisujúce správy

Každá správa zahŕňa všetky potrebné informácie, ktoré prijímateľ potrebuje, aby s ňou mohol zaobchádzať. Nemala by byť žiadna ďalšia potrebná informácia k zaobchádzaniu k danej správe nachádzajúca sa v inej.

Napríklad na obrázku 2.1 je samo popisujúca správa, pretože hovorí serveru aká HTTP metóda bola použitá (GET), a aký protokol bol použitý (HTTP 1.1). Na obrázku 2.2 posiela server klientovi v tele správy nejaký html text. V hlavičke správy je uvedený takzvaný Content-Type. Ten hovorí klientovi, aký je typ dát v tele správy. Keďže v hlavičke je ako typ označený text/html, tak klient vie, že má dáta v tele tejto správy interpretovať ako html. Týmto server spĺňa podmienku samo popisujúcej správy a klient má teda všetky potrebné informácie na zaobchádzanie.

```
GET / HTTP/1.1
Host: vutbr.cz
```

Obr. 2.1: Samo popisujúca správa pre server

```
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>
  <head>
    <title>Hello</title>
  </head>
  </body>
    <div>Hello</div>
  </body>
</html>
```

Obr. 2.2: Samo popisujúca správa pre klienta

Hypermédia

Server môže klientovi povedať, aké ďalšie požiadavky dokáže vykonať. Daný klient obvykle vykonáva akcie nad daným serverom. V tomto prípade môže server odporučiť klientovi

presmerovať svoju požiadavku na inú URI, kde klient nájde odpoveď na svoju požiadavku. Jedná sa o presmerovanie klienta na iný server bez toho, aby klient robil nejaké zmeny.

Vrstvený systém

Klient nemusí vždy komunikovať s jedným alebo koncovým serverom. Napríklad jeden server môže mať viacero inštancií svojho REST API obsluhujúceho požiadavky, ktoré vykonávajú tie isté operácie nad tými istými dátami. Ak má server vo svojej infraštruktúre vložený takzvaný load balancer, tak požiadavka klienta ide najprv cez tento bod. Load balancer rozhoduje, do ktorej inštancie túto požiadavku pošle.

Táto vlastnosť má za následok lepšiu škálovateľnosť servera, pretože dokáže naraz obslužiť viac požiadaviek. Na klienta to nemá žiaden vplyv a môže to dokonca urýchliť odozvu servera na jeho požiadavky.

Kód na požiadanie

Server bežne posiela klientovi dáta vo formáte XML alebo JSON. Kód na požiadanie je voliteľná možnosť, ktorá rozšíri funkcionalitu servera tým, že pošle klientovi rovno spustiteľný kód.

2.2.1 HTTP metódy

Protokol HTTP definuje niekoľko metód požiadaviek. Tieto metódy majú indikovať žiadanú akciu klienta nad daným zdrojom. V tejto sekcií sú popísané iba niektoré metódy, ktoré sú potrebné pre túto prácu. Uvedené metódy sú však dostatočné na vykonávanie všetkých druhov požiadaviek nad štýlom REST. Niektoré nižšie popísané metódy sú označené ako nie bezpečné. To znamená, že klient musí byť opatrný pri ich použití, pretože vie stratiť alebo zničiť dáta, ktoré má uložené na serveri. Nejedná sa o nebezpečné metódy z hľadiska skutočnej bezpečnosti.

GET

Využíva sa pri žiadaní reprezentácie nejakého špecifického objektu. Môže sa jednať o jeden objekt alebo aj množinu. Jedná sa o bezpečnú metódu, pretože dochádza len k čítaniu dát, nemení sa stav serveru. Táto metóda dokáže aj odosielať dáta, avšak server by nemal tieto dáta nijak ukladať, iba ich využiť pre spätnú väzbu na následné poskytnutie dát klientovi. Je to najčastejší druh požiadavky a zároveň aj predvolená metóda.

POST

Odosiela dáta na server. Využíva sa na zápis dát na serveri a teda vytvára zdroj alebo zdroje dát. Nejedná sa o bezpečnú metódu, pretože mení stav dát a môže to mať teda následky. Podobne ako GET, môže vrátiť dáta. Takáto požiadavka by mala vykonávať na serveri zmenu avšak nemala by byť použitá len na čítanie, čo je samozrejme možné tiež. Výsledok tejto operácie môže, ale aj nemusí, byť pri zopakovaní požiadavky s rovnakými dátami úspešný.

Používa sa napríklad pri registrácií používateľa do systému. Pokus o registráciu toho istého, už existujúceho, používateľa v systéme nemôže byť úspešný.

PUT

Rovnako ako pri metóde POST, posiela dáta na server. Taktiež nie je metóda bezpečná, pretože vykonáva zmenu dát. Používa sa na zmenu reprezentácie len jedného zdroja na základe dát v požiadavke. Táto metóda je idempotentná. To znamená, že opakovaná požiadavka na zmenu s rovnakými dátami, ktorá už bola predtým aplikovaná, nemôže byť úspešná.

Napríklad pri potrebe zmeniť údaje nejakého zákazníka v systéme. Opakovaná zmena na tie isté údaje nemôže vrátiť používateľovi informáciu o úspešnej zmene, pretože presne isté dáta sú aktuálne.

DELETE

Táto metóda vymaže zdroj na serveri. Rovnako ako pri metóde POST a PUT, táto metóda vykonáva zmeny v dátach a preto nie je bezpečná. Je taktiež idempotentná. Vymazanie už vymazaného zdroja nemôže byť úspešná operácia, pretože už jednoducho neexistuje.

2.2.2 Status kódy

Uplatňujú sa pri odpovedi servera klientovi. Súčasťou správy je aj kód, ktorý informuje o výsledku spracovania požiadavky. V nasledujúcej tabuľke sú popísané všetky status kódy, ktoré sú súčasťou tejto práce.

Status kód	Význam	Použitie
200	OK	Požiadavka bola úspešná.
204	Žiaden obsah	Požiadavka bola úspešná, žiadne dáta neboli vrátené.
400	Zlá požiadavka	Server nemôže spracovať požiadavku, pretože obsahuje chybu.
401	Neoprávnené	Klient nemá oprávnenie na tento zdroj.
404	Nenájdené	Uvedený zdroj neexistuje.
500	Interná chyba servera	Pri spracovaní požiadavky sa stala na strane servera chyba.

Tabuľka 2.1: Statusové kódy

2.3 GrahpQL

Graph Query Language (GraphQL) je požiadavkový jazyk pre systémy s klient-server architektúrou. Poskytuje alternatívu k architektúre REST. Zmysel GraphQL je eliminovať potrebu viacerých zdrojov (endpointov) a špecifikovať, o ktoré konkrétne dáta má klient záujem. Výsledkom je teda jeden jediný endpoint (koncový bod) a zníženie prietoku nepotrebných dát. Komunikácia klienta a servera prebieha vo formáte JSON.

GrahpQL má tri typy požiadaviek. Pre potreby tejto práce sú využité a nižšie popísané iba dva typy.

2.3.1 Query

Query (dopyt) je základný druh požiadavky. Jedná sa o čítanie dát. Je podobný metóde GET pri architektúre REST. Používa sa v požiadavkách, v ktorých sa nevykonáva zápis dát do databáze. Konkrétne sa používa pri hocijakých operáciách, ktoré nevyžadujú k fungovaniu databázu alebo využívajú ju len na príkaz SELECT.

2.3.2 Mutation

Mutation (mutácia) je požiadavka, ktorej hlavný cieľ je zápis dát. Podobá sa metódam POST, PUT, DELETE. Dáta vytvára, upravuje a vymazáva. Používa sa iba v takých požiadavkách, v ktorých sa vykonáva zápis dát do databáze. Rovnako ako pri idempotentných metódach u architektúry REST, opakovaná požiadavka s rovnakými dátami by nemala byť úspešná.

2.3.3 Typy dát

Dáta prenášané cez požiadavky musia mať a dodržiavať definované typy. V GraphQL sú definované nasledujúce skalárne typy [5].

- String (textový reťazec), sekvencia znakov s kódovaním UTF-8
- Int (celé číslo), celé číslo so znamienkom
- Float (desatinné číslo), číslo s desatinnou čiarkou so znamienkom
- Boolean (pravdivostná hodnota), táto hodnota je buď true (pravda) alebo false (nepravda)
- ID (unikátny identifikátor), slúži pre pamäť cache

Niektoré knižnice implementujúce jazyk GraphQL obsahujú aj ďalšie typy. Napríklad knižnica *Graphene* v jazyku Python, v ktorom je táto práca zhotovená, ponúka okrem vlastných typov aj nasledujúce.

- Date, dátum v tvare špecifikovanom podľa štandardu ISO8601.
- DateTime, dátum a čas v tvare špecifikovanom podľa štandardu ISO8601.
- Time, čas v tvare špecifikovanom podľa štandardu ISO8601.
- JSON String(reťazec JSON), dáta v JSON formáte

2.3.4 Výhody a nevýhody

GraphQL nemusí byť za každých podmienok výhodný. To závisí od architektúry systému a funkcií, ktoré ponúka.

Čo je možné považovať za výhodu aj nevýhodu, je špecifikácia chyby v prípade zlej formy požiadavky. Ak klient pošle zle naformátovanú alebo jednoducho chybnú požiadavku, tak odpoveď servera bude obsahovať špecifikáciu chyby a v ideálnom prípade napíše aj možné opravné riešenie. Výhodou je overenie korektnosti, jednoduchá zmena klienta pri zmene schémy u servera a celkovo teda rýchlejší vývoj softvéru. Nevýhodou je nižšia bezpečnosť koncového bodu, keďže štruktúra požiadavky by nemala byť pre tretie strany známa.

1. Výhody

    - Jeden koncový bod
    - Klient dostane presne také dáta, ktoré si vyžiada
    - Existuje len jedna špecifikácia. Kompatibilita GrahpQL systémov.
    - Striktné typy, čo eliminuje typové chyby.

2. Nevýhody

    - Potencionálne predĺženie odozvy API na strane back-endu. GrahpQL ešte stále nie je úplne podporovaný vo všetkých knižniciach pre tvorbu API v programovacích jazykoch. Niektoré knižnice sú vytvorené pre účely nízkej doby odozvy a ešte nemusia podporovať GraphQL.
    - Samotný back-end priamo neprofituje z tejto technológie, hlavný profit má front-end.
    - Komplikované využitie pamäte cache.
    - Vracia len 200, 401 statusové kódy. Môže teda potencionálne spomaliť klienta v dôsledku hľadania informácií o úspechu v tele samotnej správy.

Kapitola 3

Dizajn notifikačného systému

Pri dizajnovaní notifikačného systému je potrebná sumarizácia jeho vlastností. Hlavná vlastnosť tohto systému je prenos správy definovanej používateľom (aplikáciou, softvérom) na iOS zariadenie disponujúce aplikáciou na spracovanie a zobrazenie tejto správy. Jedná sa o dizajn a implementáciu serverovej časti takéhoto systému, ktorý pozostáva nie len zo samotného servera, ale aj iOS aplikácie.

Užívateľ pošle správu z jeho aplikácie, skriptu, ako webovú požiadavku do tohto systému cez jeho API. Táto správa je voľne prispôsobiteľná a obsahuje text, ktorý si užívateľ želá zobraziť na iOS zariadení. Systém obsahuje REST/GrahQL API, cez ktoré tieto požiadavky prijíma. Po následnom spracovaní ich systém posiela do fronty správ. Odtiaľ sú priradené službe systému, ktorá ich následne pošle do Apple Push Notification service (APNs). APNs je služba na strane Applu, ktorá tieto správy doručí na určené zariadenia iOS vo forme notifikácie.

Používateľom definovaná správa (notifikácia) má isté vlastnosti, ktoré ju obohacujú a prinášajú dodatočnú funkcionalitu, nie len samotné zobrazenie správy. Tieto vlastnosti sú popísane v sekcií 3.1. Vzhľadom na potreby notifikácií, funkcionalita servera je rozšírená o prihlasovací systém na ich manažment a agregáciu. Prihlasovací systém umožňuje používateľovi priradiť viacero iOS zariadení, na ktoré si môže notifikáciu poslať a zároveň umožňuje separáciu jednotlivých notifikácií na základe danej aplikácie, z ktorej ich posiela.

Na implementáciu prihlasovacieho systému bolo možné použiť knižnice tretích strán, napríklad Firebase [3], ktoré riešia tento problém. Táto práca implementuje vlastné riešenie. Dôvodom pre tento krok je naučiť sa danú problematiku.

3.1 Notifikácia

Jadrom tejto práce je práve notifikácia. Obsahuje text, ktorý sa má následne zobraziť na iOS zariadení používateľa. Okrem prenosu samotného textu obsahuje notifikácia aj nasledujúce parametre a dosahuje nasledujúcich funkcií.

- Používateľ môže jednoducho rozlíšiť notifikácie, ktoré pochádzajú z rôznych jeho aplikácií, do takzvaných services (služieb) 3.2. Na základe toho vie, odkiaľ jednotlivá správa pochádza a rozširuje prehľad o jeho aktivitách v projektoch.
- Každá notifikácia má tri levely (priority).
  - Warning

- Error
- Info

Priorita reprezentuje dôležitosť notifikácie. Určuje, či sa notifikácia vôbec pošle cez APNs na iOS zariadenie, a ako sa notifikácia na zariadení zobrazí. Žiadna priorita nie je striktne definovaná a užívateľ ich môže voľne použiť. Vie si tak prispôsobiť o aké notifikácie má záujem, a čo pre neho znamenajú. V tele správy, ktorá sa posiela na server, musí užívateľ určiť akú prioritu daná správa obsahuje.

3.2 Services

Jednotlivé notifikácie spadajú pod takzvané services (služby). Services reprezentujú aplikácie a skripty, z ktorých sú notifikácie posielané. Vytvára ich používateľ a každý service patrí práve pod neho. Každý service obsahuje názov, notifikačný level a API kľúč. Meno a notifikačný level určuje používateľ.

3.2.1 Notifikačný level

Notifikačný level popísaný v sekcií 3.1 patrí priamo k danému Service-u. Každý Service má svoje vlastné nastavenie notifikácií. Keď používateľ pošle notifikáciu, ktorá je priradená pod nejaký Service, tak server následne vyhodnotí, aké priority má na ňom užívateľ zapnuté a správu ďalej pošle do APNs. Vypnuté priority server nespracuje a takéto správy ďalej nepošle. Táto funkcia umožňuje užívateľovi jednoducho počúvať na rôzne udalosti na rôznych zariadeniach v rámci jedného Service-u.

Napríklad service A má užívateľ nastavený na prijímanie error notifikácií, ktoré pre neho budú znamenať nejaké chybné udalosti v aplikácií. Service B bude počúvať iba na udalosti označené prioritou info, čo budú napríklad úspešné udalosti v jeho druhej aplikácií. To sa dá, samozrejme, škálovať a takéto procesy môžu prebiehať pod viacerými zariadeniami a v rámci jedného používateľa, kde budú nastavenia pre všetky jeho zariadenia jednotné. Takýmto spôsobom môže užívateľ použiť jeho iOS zariadenia na udalosti z jeho naprogramovaných aplikácií.

3.2.2 API kľúč

Je unikátny identifikátor každého service-u. Je generovaný serverom pri vytváraní service-u. Je súčasťou tela správy, ktorú používateľ posiela na notifikačný server. Slúži ako autentifikačný údaj. Server pri požiadavke na notifikáciu overí totožnosť API kľúča a zistí jeho priradený Service, pod ktorý notifikácia patrí. Aby tento mechanizmus fungoval, API kľúč musí dodržať nasledujúce pravidlá.

- Musí byť unikátny
- Musí byť zmeniteľný
- Musí byť ťažko uhádnuteľný, respektíve nesmie byť verejne získateľný

3.3 Device

Je iOS zariadenie (Device), ktoré má nainštalované notifikačnú aplikáciu. Každá nainštalovaná aplikácia obsahuje unikátny kľúč nazývaný **device token**. Pre funkčnosť a identi-

fikáciu aplikácie, ktorá dostane notifikáciu, musí server poznať jej príslušný device token. Používa sa pri posielaní notifikácie cez APNs na iOS zariadenie. Používateľ neprichádza do styku s device tokenom a tento mechanizmus je pre neho kompletne skrytý. Výmenu zabezpečuje iOS aplikácia webovou požiadavkou na server bez jeho vedomia. K tomuto procesu dochádza po prihlásení 3.4 používateľa do iOS aplikácie. Server musí taktiež zabezpečiť mechanizmus prípadnej zmeny tokenov, pretože nemajú neobmedzenú platnosť a môžu byť kedykoľvek zmenené zo strany Applu. To je zabezpečené tak, že iOS aplikácia periodicky posiela tieto tokeny na stranu servera. Ten každý nový token uloží do databázy a priradí k používateľovi.

Ďalšie využitie tohto device tokenu spočíva vo vypnutí a zapnutí notifikácií pre dané zariadenie. Ak je používateľ prihlásený 3.4, tak server má v databáze uložené, že je celkové počúvanie notifikácií zapnuté. Pri odhlásení z aplikácie 3.4, server vypne tento atribút a na toto zariadenie nebude posielať žiadne notifikácie.

Pre správnu interpretáciu je potrebné poznamenať, že server má dva stupne filtrácií notifikácií. Tou prvou a najväčšou je práve filtrácia na jedno zariadenie. Pri prihlásení-/odhlásení je celé zariadenie zapnuté alebo vypnuté od notifikácií. Druhý stupeň filtrácie je samotný service 3.2. Každý service má svoje vlastné levely (priority) notifikácií, ktoré prijíma, a nastavuje si ich podľa vôle používateľ. Týmto mechanizmom je umožnené prijímanie notifikácií pod jedným zariadením, ale nemusí prijímať všetky notifikácie z v rámci jednotlivých service-ov (aplikácií).

Jeden používateľ môže mať viacero iOS zariadení s touto notifikačnou aplikáciou. Každá ma vlastný token a server si potrebuje pamätať všetky tokeny, ak má byť schopný notifikovať všetky zariadenia. Jedno zariadenie má práve jednu aplikáciu a teda jeden device token.

3.4 Prihlasovací systém

Pre pokročilé možnosti, ktoré tento systém ponúka používateľovi, je potrebné implementovať prihlasovací systém. Musí obsahovať nasledujúce funkcie.

- Registrácia používateľa
- Prihlásenie
- Odhlásenie
- Zmena hesla
- Overenie existujúceho používateľského mena/emailu

Uvedené metódy sú štandardom pri prihlasovacích systémoch. V tomto notifikačnom systéme slúžia na identifikáciu používateľa a umožňujú pod neho priradiť jednotlivé iOS zariadenia s nainštalovanou aplikáciou. V rámci aplikácie priradia pod jeho správu ním vytvorené services, ktoré sú viazané API kľúčom na jeho skutočné aplikácie, z ktorých chodia notifikácie.

Pre potreby aplikácie sú uvedené metódy postačujúce. Napriek tomu tento systém obsahuje ešte ďalšiu vrstvu. Tou je emailová konfirmácia. V každom modernom systéme si musí používateľ po registrácií pre daný systém/softvér overiť jeho email. Tento proces prebieha tak, že po registrácií dostane na uvedený email odkaz alebo kód, ktorý musí buď odkliknúť

alebo vložiť do systému. Tým si verifikuje jeho email a systém tak bude považovať daného používateľa za verifikovaného, pretože mu daný email očividne patrí. Email sa používa ako bezpečnostná záloha, ktorou je možné meniť všetky vlastnícke práva, a preto jeho verifikácia znamená veľa v rámci bezpečnosti jeho účtu.

Po registrácií v tomto notifikačnom systéme používateľ obdrží email, ktorý obsahuje token, ktorý musí poslať naspäť do systému, aby sa verifikoval. Pre funkcionalitu tohto mechanizmu je potrebné pripojenie na SMTP server, cez ktorý sa na používateľov email aj spolu s tokenom pošle.

Emailová konfirmácia umožňuje serveru implementovať dodatočné funkcie v rámci autorizácie.

- Zmena emailu
- Reset zabudnutého hesla

Implementáciou týchto metód je používateľovi umožnená plná kontrola nad svojím účtom v tomto systéme. Proces je podobný ako pri verifikácií emailu. Token z prijatého emailu používateľ použije spolu s novým emailom/heslom na jeho zmenu/reset.

Registrácia

Pre registráciu sa od používateľa vyžaduje email, prihlasovacie meno a heslo. Heslo sa neukladá ako čistý text, ale používa sa hašovacia funkcia. Heslo sa teda uloží v zahašovanom formáte a používateľ sa nemusí báť, že by sa niekto dostal k jeho pôvodnej forme. Avšak stále mu hrozí nebezpečenstvo úniku hesla v prípade, že by bolo pri tejto komunikácií odchytené metódou takzvaného man-in-the-middle-attack [11]. Informácie sa uložia do databázy a používateľ je aktuálne v neverifikovanom stave. Pri registrácií server vytvorí a uloží do databázy takzvaný refresh token, viac v sekcií 3.7.4.

Prihlásenie

Na prihlásenie sa vyžaduje buď email alebo prihlasovacie meno a heslo. Po zadaní údajov sa porovná email alebo prihlasovacie meno a spolu s heslom. Heslo sa v tomto prípade opäť zahašuje a správnosť hesla sa porovnáva s uloženou zahašovanou verziou. Pre úspešné prihlásenie je potrebné, okrem iného, byť verifikovaný. Po prihlásení sa zmení atribút počúvania notifikácií na zariadení, cez ktoré sa používateľ prihlásil, na zapnuté. V odpovedi vráti používateľovi access token, refresh token a používateľské meno (username).

Odhlásenie

Funkcia odhlásenia funguje na princípe zapnutia a vypnutia notifikácií pre dané zariadenie. Funkcia vyžaduje parameter Device Token, ktorý je popísaný v sekcií 3.3. Po odhlásení sa vypne prijímanie notifikácií na zariadení, na ktorom sa používateľ odhlásil.

Zmena hesla

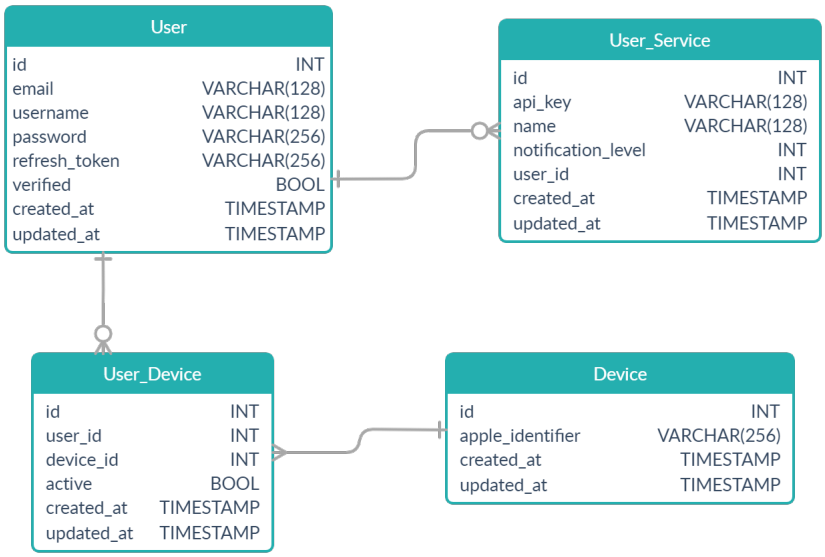
Táto funkcia umožňuje používateľovi zmeniť heslo, keď je prihlásený. Pre úspešnú zmenu hesla sa vyžaduje zadať aktuálne heslo a nové heslo. Rovnako ako pri registrácií a prihlasovaní, s heslom sa narába v zahašovanom formáte.

Overenie existujúceho používateľského mena a emailu

Funkcia overuje existenciu daného mena a emailu v systéme. Oba tieto parametre sú v rámci systému unikátne a používateľovi nie je umožnené mať duplikát. Používa sa pri registrácií používateľa. Pri zadávaní používateľského mena alebo emailu do vyplňovacieho formulára v iOS aplikácií, sa po každej zmene pošle webová požiadavka na server. Ten overí dostupnosť a iOS aplikácií okamžite pošle tento údaj. Používateľovi tak pri registrácií vyskočí upozornenie, ak pri písaní mena alebo emailu napíše už existujúci údaj bez toho, aby vôbec musel potvrdiť odoslanie požiadavky na registráciu.

3.5 Databáza

Ukladanie dát na strane servera je zabezpečené databázou. Vhodným kandidátom je relačná databáza (SQL). Slúži na trvalé uloženie dát, ktoré sa nestratia ani pri možnom výpadku systému. Dizajn databázových tabuliek pre tento systém je načrtnutý na diagrame 3.1.



Obr. 3.1: Entity Relationship Diagram

Každá tabuľka obsahuje primárny kľúč, ktorý slúži ako identifikátor na prepojenie tabuliek podľa ich skutočných vzťahov. Nie všetky identifikátory sú potrebné, ich prítomnosť je však praktická, a je teda prirodzené ich vložiť do každej tabuľky.

Posledné dve položky v každej tabuľke, konkrétne *created_at* a *updated_at*, slúžia na zaznamenanie dátumu vytvorenia/úpravy. Hoci v tomto systéme nemajú konkrétne využitie, môžu byť nápomocné pri prípadných problémoch. Napríklad pri výpadku servera v prípade nezaznamenania dát alebo sú jednoducho prítomné pre prípadne budúce pridávanie funkcií, pri ktorých sa môžu tieto dáta využiť. Taktiež môžu byť použité pre metriky a analýzy, napríklad denný počet zmien v údajoch používateľov.

User

Tabuľka *User* (používateľ) slúži na dáta súvisiace s používateľom. Pre potreby tohto notifikačného systému sa vyžaduje ukladať jeho email, username (používateľské meno), zahašované heslo, verified (verifikovaný). Tieto potreby sú popísané v sekcií 3.4. Refresh token je popísaný v sekcií 3.7.

User Device a Device

Tieto tabuľky slúžia na prepojenie používateľa s jeho iOS zariadeniami s nainštalovanou notifikačnou aplikáciou. Jeden používateľ môže mať viacero zariadení. Jedno zariadenie predstavuje tabuľku Device. Naopak, jedno zariadenie sa môže objaviť u viacerých používateľov. Pre každý záznam v tabuľke User existuje viacero záznamov v tabuľke *Device*. Zároveň pre každý záznam v tabuľke *Device* existuje viacero záznamov v tabuľke User. Tento vzťah sa nazýva **many-to-many** [6]. Nie je možné ho vytvoriť priamo, preto je potrebné vytvoriť tretiu tabuľku (User Device). Tá slúži na rozloženie vzťahu many-to-many na dva one-to-many vzťahy. Jeden one-to-many vzťah je medzi tabuľkami *User* a *User Device*. Druhý medzi tabuľkami *User Device* a *Device*. Táto tretia tabuľka (*User Device*) teda zaznamenáva každú existenciu tohto vzťahu. Musí obsahovať unikátne identifikátory z tabuliek *User* a *Device*, ktoré spojuje. V tomto prípade sú to *user_id* a *device_id*. Položka active je popísaná v sekcií 3.3.

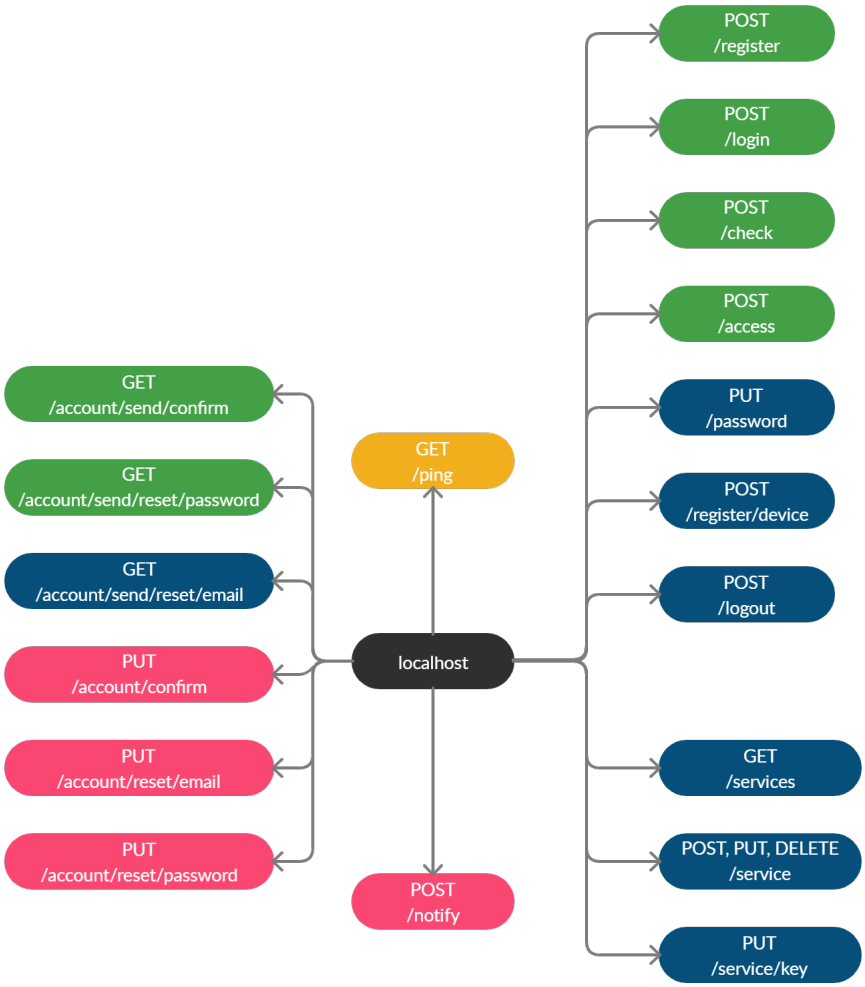
Tabuľka *Device* obsahuje takzvaný apple identifier, ďalej v zdrojovom kóde nazývaný device token. Identifikuje notifikačnú aplikáciu na iOS zariadení, viac v sekcií 3.3.

User Service

Reprezentuje reálnu aplikáciu používateľa, z ktorej posiela notifikácie na svoje zariadenie. Ukladá jej name (meno), notification level (notifikačný level), api key (api kľúč). Táto funkcia je podrobne popísaná v sekcií 3.2. Spája používateľa s jeho services (reálnymi aplikáciami), pod ktoré patria jednotlivé notifikácie. Tieto dve tabuľky sú preto vo vzťahu **one-to-many**. Jeden človek môže mať viacero záznamov v tabuľke *User Service*, ale nie naopak. Každý záznam v tabuľke *User Service* patrí len pod jedného používateľa.

3.6 API

API tohto systému obsahuje niekoľko koncových bodov (endpoints). Návrh vychádza z architektonického štýlu REST a je popísaný na grafe 3.2. Keďže táto práca implementuje štýly REST aj GrahpQL, tak pre GraphQL nie je uvedený graf celkom presný, pretože popisuje návrh endpointov (URL cesta a HTTP metóda). Pri GraphQL je len jeden endpoint, s jednou URL cestou, a používa sa striktne len HTTP metóda POST. To je však v poriadku, keďže cieľom tejto sekcie je popísať vlastnosti jednotlivých endpointov, ktoré sa uplatňujú aj pri GraphQL.



Obr. 3.2: Návrh REST API

Jednotlivé endpointy na grafe 3.2 sú podľa farieb oddelené do troch skupín.

- Zelenou sú vyznačené endpointy, ktoré nevyžadujú žiadnu autorizáciu, respektíve nevyžadujú v hlavičke webovej požiadavky autorizačné údaje.
- Modrou sú vyznačené endpointy, ktoré vyžadujú autorizáciu. V hlavičke webovej požiadavky musí byť správny autorizačný údaj.
- Červenou sú vyznačené endpointy, ktoré vyžadujú autorizáciu, ale autorizačné údaje sú prenášané v tele správy, nie v hlavičke. Využíva sa pre iný druh verifikácie používateľa, ktoré nemusí úplne súvisieť s prihlasovacími údajmi. Napríklad posielanie notifikácie na zariadenie, tam sa využíva API kľuč, ktorých má používateľ viac.

Čiernou farbou je vyznačená základná URL cesta, ktorú zdieľajú všetky endpointy, a definuje adresu servera. Žltou je vyznačený endpoint *ping*. Nie je zaradený do žiadnej skupiny, pretože nemá v tomto systéme logický význam. Funguje len na overenie dostupnosti a dĺžky odozvy servera. Klient naň nepotrebuje žiadne privilégiá.

Funkcie endpointov */register*, */login*, */logout*, */password* spolu s emailovými endpointami začínajúce sa na */account*, sú popísané v sekcií 3.4. Funkcia endpointu */register/device* je popísaná v sekcií 3.3 a */access* v 3.7.4.

Service endpointy

Koncept funkcií services je popísaný v 3.2. Endpoint */services* pri metóde *GET* vypíše zoznam všetkých services, vrátane ich dát. Endpoint */service* používa tri HTTP metódy. Pri metóde *POST* server vytvorí nový service záznam v databáze a vygeneruje príslušný API kľúč. Pri metóde *PUT* upraví uvedené dáta u daného Service-u identifikovanom prostredníctvom jeho API kľúča. Metóda *DELETE* vymaže service identifikovaný API kľúčom. Endpoint */service/key* mení API kľúč pre Service. Táto požiadavka vyžaduje samotný a ešte stále platný API kľúč, ktorým identifikuje Service, ktorého kľúč sa bude meniť. Server mu vygeneruje nový a starý stráca platnosť.

Notify

Tento endpoint prijíma webovú požiadavku, ktorá obsahuje správu, a tá sa následne zobrazí ako notifikácia na iOS zariadení. Je však chránený autorizáciou. Autorizačným údajom je API kľúč, ktorý sa vkladá do tela správy. Server na základe tohto kľúča identifikuje Service, pod ktorý notifikácia patrí, potom používateľa a následne nájde jeho Device Token. Týmto tokenom identifikuje jeho iOS zariadenie s danou aplikáciou. Vďaka tomuto tokenu vie presmerovať jeho správu ďalej na APNs 3.10 a tú Apple následne pošle na jeho iOS zariadenie.

Telo notifikačnej správy, ktorú posiela užívateľ na server, obsahuje nasledujúce parametre.

- apiKey - povinné, identifikuje Service, pod ktorý patrí.
- level - povinné, priorita notifikácie.
- body - povinné, samotná správa.
- text - nepovinné, dodatočný text správy.
- url - nepovinné, dodatočný url odkaz.

3.7 Autorizácia

Server potrebuje poznať identitu klienta pri istých webových požiadavkách. Pri požiadavke na prihlásenie sa vyžaduje email alebo používateľské meno a heslo. To stačí na identifikáciu klienta a server vie priradiť operácie, ktoré vykonáva, pod neho. Problém spočíva v tom, že klient musí pri každej požiadavke posielať meno a heslo. Tento problém je rieší JWT [4].

3.7.1 JWT - JSON Web Token

JWT Token sa používa na zakódovanie informácií do textového reťazca. Ten sa dá následne odkódovať a tým sa opäť sprístupnia dáta, ktoré do neho boli vložené. Štandardne sa používa na autorizáciu klienta, pretože dokáže držať a overiť jeho identitu. Pri vytváraní JWT tokenu sú vyžadované nasledujúce údaje.

- Header. Identifikuje algoritmus, ktorý bol použitý na zakódovanie dát
- Payload. Vlastné dáta, ktoré sa zakódujú. Sem môže byť vložené čokoľvek. Okrem vlastných dát sa sem zvykne pridávať dátum vydania a dátum expirácie tohto tokenu. Tieto dáta však nie sú povinné.
- Signature. Jedná sa o podpis vydavateľa tohto tokenu. Vytvára sa z headeru, payloadu a tajného kľúča. Využíva sa na verifikáciu správy zakódovanej vo vnútri, či nedošlo k jej zmene, a kto je jeho vydavateľ.

Výstupný formát tohto tokenu je Base64Url [7] textový reťazec zo zakódovanými dátami. Payload, teda miesto kde sú uložené vlastné dáta, by nemal obsahovať tajné informácie, pretože aj po zakódovaní sa dá jednoducho odkódovať a obsah tejto časti je viditeľný pre všetkých.

3.7.2 Rozdiel medzi autentifikáciou

Autentifikácia je proces určovania identity klienta [13]. Autorizácia je overenie právomocí klienta. Autentifikácia prebieha v tomto notifikačnom serveri na princípe prihlásenia prostredníctvom emailu/mena a hesla. Prihlásenie vráti klientovi JWT token, ktorý následne využíva na autorizáciu.

3.7.3 Použitie JWT

JWT tokeny sa v tomto systéme používajú na dva účely. Prvý je vo forme **Access token**u. Ten sa používa na autorizáciu a identifikáciu klienta, ktorý sa autentifikoval prihlásením. Po úspešnom prihlásení dostane klient v odpovedi tento access token. Umožňuje klientovi prístup k zabezpečeným endpointom. Pri webovej požiadavke sa vkladá do hlavičky správy. Klient musí pri každej webovej požiadavke na zabezpečený endpoint pridať aj tento access token. Na základe tohto tokenu vie server identifikovať jeho totožnosť.

Druhý je vo forme **Email token**u. Tento token sa používa pri emailových operáciach. Každý email, ktorý server odošle, obsahuje token na svoju funkcionalitu. Po úspešne vykonanej emailovej operácií, napríklad verifikácia, token okamžite stráca svoju platnosť. Všetky emailové tokeny môžu byť použité iba raz.

Obe formy tokenov majú dĺžku expirácie 10 hodín. To znamená, že po desiatich hodinách strácajú platnosť a musia byť vydané znova. To za normálnych okolností znamená, že používateľ sa musí opäť prihlásiť s menom a heslom. Tento problém rieši **Refresh token** 3.7.4.

Server pri vytváraní JWT tokenu do neho zakóduje nasledujúce informácie.

- *iss* - Meno samotného notifikačného servera.
- *action* - JWT token sa používa, okrem autorizácie na zabezpečené metódy, ako verifikačný emailový token.
- *user_id* - ID používateľa, čo je jeho primárny kľúč v databáze.
- *user_secret* - Využitie popísané v sekcií 3.8.
- *iat* - Dátum vydania tokenu.
- *exp* - Dátum expirácie tokenu.

3.7.4 Refresh token

Po expirácií access tokenu potrebuje používateľ jeho obnovu, aby sa nemusel znova prihlasovať. Po prihlásení mu server vráti tento refresh token, ktorý bude potrebovať, ak teda nechce byť samovoľne odhlásený. Existuje endpoint */access*, ktorý namiesto opätovného prihlásenia prijíma tento parameter a následne vráti používateľovi naspäť nový platný access token. Používateľ si teda musí uložiť refresh token. To však za neho rieši iOS aplikácia a tá periodicky obnovuje access token sama. Tento proces je teda pre používateľa skrytý.

3.8 Caching

Caching (vyrovnávacia pamäť) sa v tejto práci využíva na poskytnutie ďalšej vrstvy zabezpečenia účtu používateľa. Táto metóda spočíva v jeho automatickom odhlásení. Proces nastáva pri zmene hesla, či už klasicky vo vnútri aplikácie alebo cez email, alebo zmene samotného emailu. Dôvodom je možné zneužitie účtu.

V napríklad v prípade, že používateľ má viacero zariadení s iOS aplikáciou, v ktorých je prihlásený. Ak mu je jedno zariadenie odcudzené, tak si v druhom zmení heslo a systém ho automaticky odhlási zo všetkých zariadení. Týmto spôsobom má zabezpečenú dodatočnú ochranu.

Pri vytváraní access tokenu do neho server vloží takzvaný *user_secret*. Táto položka sa vytvára tak, že sa zahašuje kombinácia jeho emailu a jeho už zahašovaného hesla. To znamená, že akonáhle sa zmení jeho email alebo heslo, tak zahašovanie tejto kombinácie už nebude rovnaké. Bude teda znamenať zmenu jedného alebo oboch údajov. Keďže tento údaj sa vkladá do access tokenu, tak každý endpoint, ktorý vyžaduje autorizáciu, bude porovnávať aj tento *user_secret*. Ak sa nebude rovnať, server bude považovať token za neplatný a používateľa ďalej nepustí. Pri tomto procese sa využíva caching. Do tejto pamäte sa ukladá zahašovaná kombinácia emailu a hesla daného používateľa, ktoré sa načítajú z databázy, práve keď prechádza autorizáciou. Následne je držaná v tejto pamäti a pri ďalších zabezpečených endpointoch už nie je potrebné tieto dáta opäť načítavať z databázy. Vo vyrovnávacej pamäti sú držané istú dobu a následne expirujú. Použitie cachovania síce nie je potrebné, ale významne urýchľuje vybavenie týchto požiadaviek, pretože miesto získavania údajov z databáze sú už načítané v rýchlej pamäti. Odhlásenie používateľa prebieha tak, že pri zmene jeho emailu alebo hesla server odstráni jeho dáta z cache pamäti. To znamená, že pri hocijakej ďalšej požiadavke cez zabezpečený endpoint, bude musieť server tieto údaje znova načítať z databázy a už sa teda nebudú zhodovať s tým, čo má používateľ v access tokene.

Keďže sa *user_secret* ukladá do payloadu JWT tokenu, tak je viditeľný pre každého, kto sa rozhodne odkódovať payload. Tento problém je ale vyriešený zahašovaním kombinácie vložených údajov. Používateľ sa teda nemusí báť o svoju bezpečnosť.

3.9 Distribuovaná fronta správ

Po prijatí webovej požiadavky, respektíve notifikácie, na endpoint */notify*, sever spracuje údaje a správu presmeruje ďalej na APNs. Za normálnych okolností by spracovanie požiadavky a následné odoslanie ďalej na APNs vrátane odozvy trvalo dlho a používateľ by musel čakať, kým prebehne celý tento proces. Inak povedané, bol by to synchrónny proces a používateľ by čakal až kým nepríde notifikácia na jeho zariadenie. Spôsobilo by to veľké

spomalenie na strane jeho aplikácie. Tento problém je vyriešený použitím distribuovanej fronty správ.

Po spracovaní údajov v endpointe */notify* server vloží všetky potrebné údaje na presmerovanie notifikácie smerom na APNs do tejto fronty. Fronta je vlastný bežiaci proces nezávislý od servera. Prijíma jeho požiadavky a odbavuje ich. To umožňuje serveru spracovať dáta, vložiť ich do fronty a okamžite vrátiť používateľovi nejakú odozvu bez toho, aby sa čakalo na budúce spracovanie jeho notifikácie. Fronta vytvára tento endpoint **asynchrónnym**.

O nasledujúce spracovanie sa server už nestará a vybavenie požiadavky zabezpečuje fronta. Fronta postupne vyberá správy na princípe first in first out (prvý dnu prvý von) a odbavuje ich. Fronta obsahuje implementáciu APNs spojky 3.10, ktorá komunikuje s Applom, kam tieto notifikácie posiela. Keďže je fronta vlastný nezávislý proces, o priebehu a prípadných chybách sa server nedozvie a nedozvie sa ich ani používateľ. Kompetencia servera je získať dáta od používateľa a predať ich ďalej do fronty, ktorá zabezpečuje ďalší proces. To je následkom asynchronity. Výsledkom je príjem notifikácie na používateľovom iOS zariadení bez zbytočného zdržania v rámci jeho webovej požiadavky.

Ďalším dôvodom na použitie fronty je potencionálny veľký nápor notifikácií. V takomto prípade server nebude schopný vybaviť všetky požiadavky vo vhodnom čase a bude zbytočne zdržovať používateľov, respektíve ich aplikácie. Správy vo fronte odbavuje takzvaný worker (pracovník), respektíve proces. Ten si vezme z fronty jednu správu a obslúži ju. Workeri sa dajú škálovať a v prípade veľkého náporu ich môže vzniknúť niekoľko (viacero procesov), čo zabezpečí rýchle odbavenie notifikácií.

3.10 APNs

Apple Push Notification Service (APNs) [1] je služba na strane Applu, ktorá zabezpečuje prenos notifikácií na iOS zariadenia. Pre doručenie notifikácie na zariadenie je potrebná komunikácia s touto službou. Poskytovateľ (notifikačný server), ktorý táto práca implementuje, musí vytvoriť trvajúce a zabezpečené spojenie s APNs. Komunikácia s APNs prebieha cez protokol HTTP/2. APNs vyžaduje od poskytovateľa dodržať alebo zabezpečiť nasledujúce požiadavky.

- Získať device token, ktorý slúži na identifikáciu notifikačnej aplikácie v rámci zariadenia.
- Získať dáta, ktoré chce používateľ poslať na svoje zariadenie, teda samotný text notifikácie.
- Určiť, vzhľadom na dizajn notifikačného servera, kedy sa má notifikácia poslať.
- Vytvoriť a poslať notifikačnú požiadavku do APNs. Každá požiadavka musí obsahovať takzvaný payload (jadro správy) a informácie potrebné pre identifikáciu iOS zariadenia a jej aplikácie.

Každá požiadávka od poskytovateľa do APNs musí:

1. Vytvoriť vo formáte JSON notifikačné jadro správy (payload). Jedná sa informácie, ktoré chce používateľ dostať vo forme notifikácie.
2. Vložiť tento payload a device token do jednej HTTP/2 požiadavky.

3. Poslať webovú požiadavku cez HTTP/2 do APNs. Vyžaduje sa už nadviazané spojenie, ktoré sa autentifikuje kryptografickými údajmi.

Pre nadviazanie spojenia s APNs, je nutné trvalé zabezpečené HTTP/2 pripojenie. Pre autentifikáciu notifikačného servera do APNs Apple od neho požaduje kryptografické údaje. Existujú dva druhy údajov.

1. Tokenovo založené dôverné spojenie.
    - Pri tejto metóde sa vyžaduje vloženie tokenu do každej notifikačnej požiadavky posielanej do APNs.
    - Nie je potreba vytvárať certifikáty s limitovanou dobou platnosti.
2. Certifikačne založené dôverné spojenie.
    - Certifikát sa vkladá do úvodného vytvorenia spojenia s APNs a na rozdiel od tokenovej komunikácie ho nie je potrebné vkladať do každej notifikačnej požiadavky.
    - Certifikát má platnosť jeden rok, potom je potrebné vytvoriť nový a teda aktualizovať certifikát na serveri.

Obe metódy vyžadujú Apple developerský účet. V tejto práci je zvolená certifikačná metóda.

Kapitola 4

Implementácia

V tejto kapitole sú popísané implementácie jednotlivých súčastí systému. To okrem samotnej implementácie zahŕňa aj popis použitých technológií a knižníc, ktoré implementujú tieto súčasti. Implementácia serverovej časti je v jazyku Python.

4.1 Logovanie

Notifikačný server interpretuje webové požiadavky a niektoré stavy vo forme logov. Účel logovania je zachytiť tieto informácie, ktoré môžu byť užitočné pre potencionálnu analýzu. Implementácia logov prebieha pomocou Python knižnice **structlog**[1]. Nastavenie logovania pre potreby tohto servera sa nachádza v priečinku *src/notifire/log*. Logy môžu mať 3 úrovne, každá interpretuje rôzny stav a je na programátorovi, čo si pod jednotlivými úrovňami predstavuje. Táto práca definuje nasledovné úrovne.

- Exception, neočakávaná a vážna chyba.
- Warning, upozornenie, že niečo nemusí byť v poriadku, ale nie je to vážne.
- Info, informuje o obyčajnej udalosti alebo stave.

Na obrázku 4.1 je ukážka logu z endpointu ping.

```
16. 5. 2020 0:45:12 {"request_id": "1e9f9fc711a34d16819f7b1fbd34d7bc", "method": "GET",
"path": "/ping", "client": "10.42.19.202", "event": "request", "level": "info",
"timestamp": 1589582712.112}
16. 5. 2020 0:45:12 {"request_id": "1e9f9fc711a34d16819f7b1fbd34d7bc", "status": 200,
"event": "response", "level": "info", "timestamp": 1589582712.112}
```

Obr. 4.1: Výpis informácie vo forme logu

4.2 Databáza

Použitá technológia na implementáciu databázy je **PostgreSQL**[2]. Súbor *sql/schema.sql* obsahuje implementáciu databázových tabuliek podľa diagramu 3.1. Obsahuje konkrétne príkazy na vytvorenie tabuliek a takzvaných databázových triggerov (spúšťačov). Tento

[1] https://github.com/hynek/structlog
[2] https://www.postgresql.org/

súbor sa použije pri vytváraní samotnej databázy a implementácia serverovej časti teda nemusí riešiť ich vytváranie po naštartovaní aplikácie.

Všetky tabuľky obsahujú *id* ako svoj primárny kľúč, teda *PRIMARY KEY* [10]. Primárny kľúč musí byť číselná hodnota, ktorá nesmie byť prázdna alebo obsahovať duplikátnu hodnotu. Parameter *SERIAL* rieši problém s duplicitnými hodnotami. Serial automaticky priradí číselnú hodnotu stĺpcu *id*, čím splní podmienku pre nenulovú hodnotu. Po vzniku ďalšieho riadku inkrementuje číselnú hodnotu pre stĺpec id, a tak zabezpečí unikátnosť hodnôt v týchto stĺpcoch v rámci jednotlivých tabuliek.

Všetky tabuľky obsahujú stĺpce *created_at* a *updated_at*. Sú vytvorené databázou automaticky na základe databázových triggerov, ktoré sú implementované v uvedenom súbore. O ich manuálne vloženie sa teda server nemusí starať. Trigger *created_at* sa automaticky zapne pri vytváraní riadku v databáze a *updated_at* pri akejkoľvek zmene údajov v tomto riadku. Aby fungovali, musí byť najprv implementovaná tabuľka, v ktorej sa používajú a až potom samotný trigger.

Údaje vo forme textového reťazca sú vytvorené parametrom *VARCHAR*, kde sa do zátvorky udáva maximálny počet znakov reťazca pre daný stĺpec.

Parameter *BOOLEAN* umožňuje danému stĺpcu uschovávať hodnoty *True* alebo *False*. Umožňuje taktiež nastaviť východziu hodnotu pri vytváraní tohto stĺpca.

4.2.1 Tabuľky

Tabuľka *device* obsahuje stľpec *apple_identifier*, čo je identifikátor jedného iOS zariadenia s danou notifikačnou aplikáciou. Jeho maximálna dĺžka je 256 znakov, musí byť unikátny a nenulový. To zabezpečujú parametre *UNIQUE NOT NULL*.

Tabuľka *user* uschováva údaje *email* a *username* (používateľské meno). Oba majú nastavenú maximálnu dĺžku 128 znakov, musia byť unikátne a nenulové. Jeho heslo *password* musí byť nenulový stĺpec. Maximálna dĺžka je 256 znakov, nachádza sa tu však už zahašované heslo. Údaj *refresh_token* je unikátna nenulová hodnota maximálnej dĺžky 256 znakov. Unikátna je preto, pretože sa používa na získanie access tokenu potrebného na prihlásenie. Ak by to nebola unikátna hodnota, dvaja rôzni používatelia by sa vedeli prihlásiť ako jeden. Údaj *verified* typu *BOOLEAN* má nastavenú východziu hodnotu *False*. Pretože po vytvorení riadku, to znamená po registrácií, je používateľ automaticky neverifikovaný.

Tabuľka *user_device* vytvára vzťah many-to-many medzi používateľom a jeho zariadením. Preto obsahuje stĺpce *user_id* a *device_id* odkazujúce sa na primárne kľúče tabuliek *user* a *device*. Obe majú parameter *ON DELETE CASCADE*. To znamená, že pri vymazaní riadku v jednej z týchto dvoch tabuliek identifikované primárnym kľúčom, na ktoré tabuľka *user_device* odkazuje, sa vymaže aj celý riadok v tejto tabuľke. Napríklad používateľ má jedno zariadenie, to zodpovedá jednému riadku v tabuľke *device* a aj v *user_device*. Ak sa vymaže riadok v tabuľke *device*, na ktoré *user_device* odkazuje, tak sa vymaže aj riadok v tejto tabuľke, pretože musí odkazovať na nejaké zariadenie. Tento parameter oslobodzuje server od dodatočného vymazávania a vytvárania logiky, ktorou by sa tieto vzťahy museli sledovať. Údaj *active* reprezentuje počúvanie notifikácií na tomto zariadení (true alebo false). Táto tabuľka obsahuje aj unikátny kľúč (unique key) [10], ktorý sa skladá z unikátnej kombinácie používateľa a zariadenia *UNIQUE (user_id, device_id)*. Dôvodom je, že unikátnosť jedného riadku v tejto tabuľke je dosiahnutá práve týmito údajmi. Napríklad, používateľ A má zariadenie C. Používateľ B má taktiež zariadenie C. Aby bolo možné rozlíšiť tieto dva potencionálne stavy, je potrebné identifikovať riadok ako kombinácia používateľa a zariadenia.

Tabuľka *user_service* definuje vzťah one-to-many medzi používateľom. Obsahuje stĺpec *api_key* unikátnej nenulovej hodnoty s maximálnou dĺžkou 128 znakov. Unikátna je, pretože všetky notifikácie posielané na iOS zariadenie sa identifikujú práve týmto údajom, nie menom a heslom používateľa. Nenulový údaj *name* definuje názov tohto Service-u a je maximálnej dĺžky 128 znakov. Meno môže byť duplicitné, pretože nepredstavuje žiadne obmedzenie a používateľ môže mať viacero aplikácií s rovnakým názvom. Nie je preto potreba ho nijak obmedzovať. Údaj *notification_level* je uložený ako číslo, predstavuje aktuálne nastavenie notifikačných levelov, ktoré daný Service prijíma. Pre jednoduchosť ukladania týchto dát, ich nastavenie je transformované do čísla. Odkazuje na tabuľku *user* cez jeho primárny kľúč.

4.2.2 Rozhranie

Súbor *src/notifire/databases.py* obsahuje implementáciu spojenia servera s databázou. Komunikácia s databázou je implementovaná pomocou Python knižnice **asyncpg**[3]. Táto knižnica je nadizajnovaná špeciálne pre PostgreSQL a dosahuje vačšiu rýchlosť v porovnaní s najbežnejšie používanou knižnicou **psycopg2**[4].

Databázové operácie, ktoré sa tento server implementuje, potrebujú pripojenie. Každé databázové query je vo vlastnej funkcií a prvý parameter každej tejto funkcie je práve databázové pripojenie. To sa predáva do funkcie pomocou dekorátora *notifire_db*. Tento princíp uľahčuje čitateľnosť kódu, odstraňuje duplicitné riadky a umožňuje znovupoužitie.

Pripojenie do databáze môže byť nadviazané v dvoch formách [12].

Single connection

Jedno pripojenie (single connection) je bežne využívaný druh pripojenia servera do databázy. Funguje tak, že všetky databázové queries (požiadavky) používajú práve toto jedno spojenie na získanie dát. Tento princíp ma veľa nedostatkov. Ak veľa používateľov vytvorí veľa požiadaviek, ktoré vyžadujú dáta z databázy, tak všetky sa budú postupne po jednom odbavovať na tomto jednom pripojení. To znamená, že jednotlivé požiadavky čakajú vo fronte a čakajú, kým prídu na rad. Jedno pripojenie je implementačne nenáročné a dostatočné, ak je požiadaviek málo. Akonáhle začne stúpať počet a výpočtová náročnosť požiadaviek, začne byť toto pripojenie nedostatočné a nespoľahlivé.

Nevýhody jedného pripojenia sú nasledujúce.

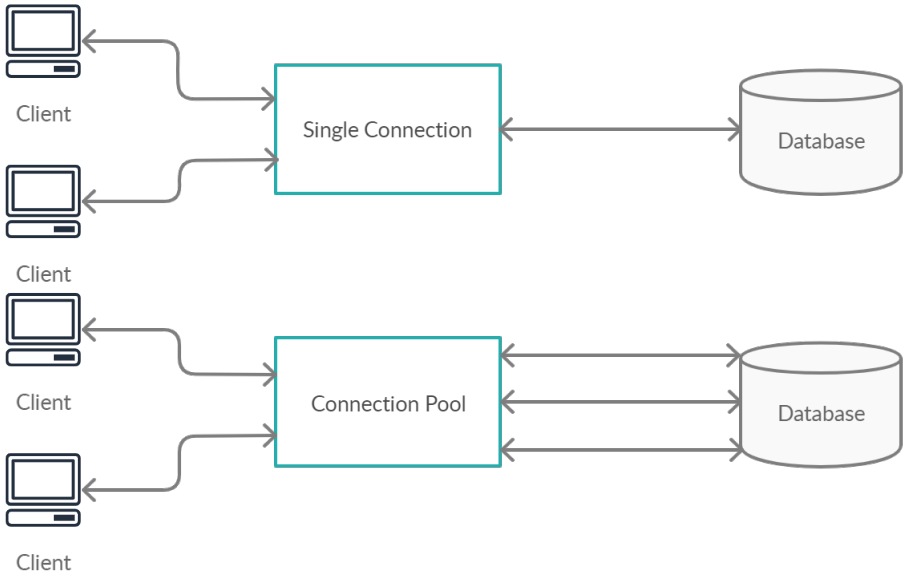
- Všetky požiadavky bežia na jednej inštancií pripojenia, ktorého rýchlosť je úmerná v počte požiadaviek, ktoré musí za sekundu vykonať.
- Server je závislý od jedného pripojenia. Ak sa pripojenie zavrie, tak server nemá ako obslúžiť používateľa. V tomto prípade musí implementovať mechanizmus otvárania spojenia.
- Pripájanie sa na databázu je pomalá operácia. Ak by teda server po vykonanej požiadavke zatváral pripojenie, musel by ho pri novej požiadavke opäť otvárať. To by znamenalo významné spomalenie.

[3] https://github.com/MagicStack/asyncpg
[4] https://github.com/psycopg/psycopg2

Connection pool

Takzvaná zbierka pripojení (connection pool) rieši problémy jedného pripojenia. Po vytvorení spojenia s databázou, je toto otvorené pripojenie vložené do zbierky (pool) a použité znova, keď je potrebné. Nie je teda nutné ho znova otvárať. Ak sa stane, že novú požiadavku nemá kto obslúžiť, všetky pripojenia sú zaneprázdnené, tak sa vytvorí nové pripojenie a vloží sa do zbierky. V zbierke sa nachádzajú voľné pripojenia, ktoré sú schopné obsluhy. Po priradení k požiadavke sa zo zbierky vyberie a obslúži požiadavku, následne sa do nej opäť vráti. Tento princíp je vhodný aj pre malé servery, kde nie je veľa požiadaviek v danom momente. Umožňuje škálovanie.

V tejto práci je využitá táto forma. Dôvodom je to, že je to odporúčaný spôsob vytvárania spojenia, škálovateľnosť. Patrí do to takzvaných best practices (najlepších riešení), čo sa týka tvorby kódu.



Obr. 4.2: Porovnanie jedného a zbierky pripojení [12]

4.3 API

REST API ale aj GraphQL sú implementované pomocou frameworku **sanic**[5]. Vyznačuje sa vysokou rýchlosťou a to je dôvodom pre jeho použitie. Tento framework neobsahuje niektoré funkcie, ktoré sú pre stabilitu a spoľahlivosť API vhodné. Jedná sa o nasledujúce vlastnosti.

- Maximálny počet požiadaviek, ktoré dokáže jeden worker (proces) tohto servera obslúžiť. Táto funkcia slúži na zamedzenie únikom pamäte. Po vykonaní určitého počtu požiadaviek sa worker reštartuje.
- Timeout. Jedná sa o čas, ktorý má k dispozícii worker na vykonanie požiadavky. Po uplynutí času je zrušený a reštartovaný.

[5] https://github.com/huge-success/sanic

Tieto funkcie sa dajú dosiahnuť použitím knižnice **gunicorn**[6]. Jedná sa o HTTP server, ktorý sa spojí frameworkom Sanic a rozšíri tak jeho funkcie. Tieto funkcie sú nastavené v súbore *.misc/gunicorn_config.py*. Okrem spomenutých funkcií tento súbor obsahuje ešte tri ďalšie parametre.

Parameter *max_requests_jitter* pridá k maximálnemu počtu požiadaviek, ktoré jeden worker obslúži, náhodne zvolené číslo z intervalu 0-100. Tento interval je voliteľný a slúži na to, aby sa nereštartovali všetci workeri naraz. To by znamenalo dočasný výpadok API a nebolo by teda možné prijímať požiadavky, keďže všetci workeri by boli v procese reštartovania. Týmto je zaručené náhodné reštartovanie workerov a nestane sa situácia, že sú všetci nedostupní.

Parameter *backlog* určuje maximálny počet klientov, ktorí môžu byť v danom momente obslúžení. V prípade tejto práce sa jedná o číslo 2048.

Počet workerov cez parameter *workers*. Sú to procesy, ktoré obsluhujú klienta. Knižnica Gunicorn odporúča mať počet workerov z intervalu 2-4krát počet jadier na procesore, pod ktorým beží. V tejto práci je použité množstvo rovnajúce sa počtu procesorových jadier. Toto číslo nie je pevne napísané v kóde, na ich zistenie je použitá knižnica *multiprocessing*, ktorá vráti aktuálne množstvo. Týmto je zabezpečená nezávislosť, pretože API môže bežať na viacerých strojoch, ktoré majú rôzne počty jadier.

Toto nastavenie obsahuje pevne daný port, na ktorom beží (80), maximálny počet požiadaviek pre workera (1000) a timeout (60 sekúnd).

Implementácia API sa začína súborom *src/notifire/api.py*. Obsahuje URL cesty a HTTP metódy jednotlivých endpointov. Každý endpoint reprezentuje funkcia v kóde. Všetky endpointy v tomto systéme obsahujú takzvanú **validáciu dát**.

4.3.1 Validácia dát

Pri posielaní požiadaviek, od klientov na API tohto servera, sú prenášané dáta. Server by mal byť schopný overiť, či tieto dáta dodržiavajú predom definovanú štruktúru a ich hodnoty sú správne. Tento proces by sa mal vykonať pred samotným spracovaním týchto dát, aby ich následne vedel správne prečítať a vyhodnotiť. Ak by došlo k nedodržaniu, server vráti klientovi špecifickú odpoveď, ktorá mu môže pomôcť identifikovať nezrovnalosti. Na tieto účely slúži validácia dát. Validácia je implementovaná pomocou knižnice **cerberus**[7].

Implementácia pozostáva z dvoch častí. Prvá je implementácia takzvaných validačných schém, v ktorých je definované, aké dáta daný endpoint prijíma. Jedná sa o názov, typ a hodnoty jednotlivých parametrov a štruktúru tela požiadavky. Klient posielajúci požiadavku musí presne dodržať všetky podmienky danej schémy. Typy parametrov sú integer, string, JSON, boolean. Ich hodnoty sú definované intervalom hodnôt v prípade čísel, minimálnou a maximálnou dĺžkou alebo regulárnym výrazom v prípade textového reťazca, true alebo false v prípade typu boolean. Parametre taktiež môžu byť označené ako povinné alebo nepovinné a mať východziu hodnotu. Tieto schémy sú implementované v priečinku *src/notifire/schemas*.

Druhá časť je implementácia validačného dekorátora *handle_params*, ktorý porovnáva prijaté dáta s definíciou schém. Nachádza sa v súbore *src/notifire/common/validations.py*. Schéma pre jeden endpoint sa taktiež môže meniť na základe jeho HTTP metódy. Definície

[6] https://github.com/benoitc/gunicorn
[7] https://github.com/pyeve/cerberus

schém ako aj tento dekorátor umožňujú endpointom zmeniť svoje schémy na základe tejto prijatej metódy. Dekorátor sa používa tak, že sa dá pred funkciu, ktorá reprezentuje daný endpoint, a ako parametrom tohto dekorátora je názov danej schémy, ktorou sa má riadiť. Ak je hodnota prázdna, hľadá schému podľa názvu funkcie.

V prípade, že klient pošle zlé údaje, odozva servera bude obsahovať presnú špecifikáciu danej chyby a statusový kód je vždy 400. Kód 400 podľa REST API štandardu znamená zlú požiadavku, kde sa chyba stala na strane klienta. Servera dodržiava tento štandard. Univerzálny príklad odpovede servera je znázornený na obrázku 4.3, kde parametre *status* a *msg* obsahujú vždy presne tieto hodnoty a teda hovoria klientovi, že poslal zlé dáta. Parameter *errors* špecifikuje konkrétne chyby.

```
{
    'status': 'error',
    'msg': 'invalid request data',
    'errors': {
        'field': 'error message',
        'field2': 'error message 2',
        }
}
```

Obr. 4.3: Odpoveď servera pri požiadavke, ktorá neprešla validáciou

Validácia pri GraphQL

GraphQL obsahuje svoju vlastnú implementáciu validácie dát. Implementuje ju samotná GraphQL knižnica a jedná sa o jednoduchšiu formu validácie. Nepoužíva schémy a jedná sa teda len o porovnanie správnosti typov a názvov. Validácia dát implementovaná na tomto serveri sa nevyužíva len pri REST API, ale aj pri GraphQL. Keďže všetky endpointy sú v kóde reprezentované jednou funkciou, tak sú pre obe implementácie API spoločné. To znamená, že GraphQL používa rovnaké funkcie s validačným dekorátorom, ktorý validuje dáta. Klient má tak k dispozícií rovnakú validáciu dát ako pri REST, tak aj pri GraphQL.

4.3.2 Autorizácia endpointov

Endpointy, ktoré sú zabezpečené, vyžadujú autorizáciu vo forme JWT tokenu [4]. Klient najprv potrebuje tento token získať. Endpointy, ktoré token vytvárajú, volajú funkciu *create_jwt_token* zo súboru *src/notifire/security.py*. Táto funkcia prijíma argumenty, niektoré sú špecifikované v 3.7.3. Východzia hodnota premennej *action* je prázdny reťazec. Rozlišuje použitie JWT tokenu medzi autorizáciou a emailovou autorizáciou, ktorá taktiež využíva JWT token na svoju funkciu. V prípade obyčajnej autorizácie je použitá východzia hodnota, v druhom prípade sa hodnota rovná aktivite, ktorú emailová operácia vykonáva. Hodnota pre *iat* (dátum vytvorenia) je vytvorená knižnicou *datetime* a jej funckiou *datetime.utcnow()*, ktorá vráti aktuálny dátum a čas v časovom formáte UTC. Hodnota *exp* (dátum expirácie) je tvorená tou istou funkciou na aktuálny dátum a čas, ku ktorému sa pripočíta čas z argumentu *duration*, ktorý predstavuje čas expirácie v hodinách. Argument *duration* má východziu hodnotu importovanú zo súboru *src/notifire/constants.py*. Argument *key* predstavuje tajný kľúč, s pomocou ktorého sa dáta zakódujú. Jeho východzia hodnota je importovaná zo súboru *src/notifire/settings.py*. Sú dva druhy tajného kľúča,

jeden je pre štandardnú autorizáciu a druhý pre emailovú. Na zakódovanie týchto dát je okrem kľúča potrebné špecifikovať kódovací algoritmus. Ten je pre všetky účely rovnaký, a preto je pevne vložený do kódu pre zakódovanie ako aj dekódovanie. Na samotné vytvorenie ako aj dekódovanie je použitá knižnica **pyjwt**[8]. Dokumentácia tejto knižnice popisuje implementáciu vytvárania JWT tokenov a ako ukážku používa kódovací algoritmus *HS256*. Na základe tohto odporúčania je použitý aj v tejto práci.

Každý zabezpečený endpoint, predstavujúci funkciu v kóde, je dekorovaný dekorátorom *authorization*. Jeho implementácia je v súbore *src/notifire/decorators.py*. Webová požiadavka klienta na takýto zabezpečený endpoint musí obsahovať okrem iných dát aj JWT token v jej hlavičke. Ten sa dekóduje funckiou *decode_jwt_token* importovanou zo súboru *src/notifire/security.py*. Na dekódovanie je potrebný samotný token, tajný kľúč a algoritmus, ktoré sú rovnaké ako pri vytváraní tohto tokenu. Po dekódovaní nasleduje verifikácia JWT tokenu funckiou *verify_jwt_token* tiež importovaná z tohto súboru. Tá porovná hodnotu premennej *action*. Napríklad, používateľ po prihlásení disponuje access tokenom. Ak funkcia určená na verifikáciu emailu dostane platný access token, tak je možné ho odkódovať a mať všetky potrebné údaje. Lenže access token nebol na tieto účely vytvorený a zabráni tomu práve porovnanie hodnoty z premennej *action*, ktorá je iná pre emailový token. Práve toto porovnanie zabráni falošnému overeniu, ktoré by bolo inak možné. Funkcia taktiež porovnáva *user_secret*, viac v 4.5.

V tomto momente disponuje dekorátor všetkými údajmi, ktoré boli do JWT tokenu zakódované. Napríklad identifikátor používateľa, ktorý je pre ďalšie operácie nevyhnutný. Ak nedošlo k autorizačným chybám, údaje sú predané ďalej dekorovanej funkcií v cez jej argument.

Emailová autorizácia

Na účely emailových operácií je taktiež vhodný JWT token ako forma autorizácie. Pri vytváraní tokenu je použitý rovnaký spôsob ako pri štandardnej autorizácií vo forme access tokenu. Zavolá sa funkcia *create_jwt_token*, rozdiely sú v hodnotách *duration*, *action* a *key*. Obe sú importované z rovnakých súborov ako pri predošlej forme autorizácie. Expirácia je nastavená na 48 hodín, čo reflektuje štandardne používanú hodnotu. Hodnota *key* je importovaná z nastavení, jedná sa však o variantu pre email. Po zakódovaní sa tento token zakóduje do formy UrlBase64 [7]. Táto forma prijíma 8-bitové dáta a zakóduje ich iba pomocou znakov *a-z, A-Z, 0-9, _, -, =*, ktoré môžu byť prenesené cez komunikácie, ktoré nedokážu preniesť 8-bitové dáta ako napríklad **email**.

Pri dekódovaní sa najprv odkóduje token z formy UrlBase64 späť do zakódovaného JWT tokenu. Ďalší postup prebieha rovnako ako pri štandardnej autorizácií. Dekódovanie tokenu prebieha tou istou funkciou, ale s kľúčom vyhradeným pre emailové tokeny. Nasleduje verifikácia tokenu, ktorá overí, či je token určený na emailovú alebo štandardnú autorizáciu. Potom rovnako posunie odkódované dáta ďalej do dekorovanej funkcie.

Autorizačné chyby

Webové požiadavky na zabezpečené endpointy nemusia prejsť autorizáciou. Pri neplatnej autorizácií sú možné nasledovné stavy.

[8] https://github.com/jpadilla/pyjwt

1. Hlavička webovej požiadavky neobsahuje token (platí iba pri autorizácií vo forme access tokenu).
2. Token stratil platnosť.
3. Token je neplatný.

Server v odozve požiadavky pre klienta špecifikuje, ktorý stav nastal. V prípade chýbajúceho tokenu je stavový kód 400. To sa rovná zlej požiadavke, kde chyba je na strane klienta. V zostávajúcich dvoch prípadoch vráti kód 401. To znamená, že klient nie je oprávnený, respektíve neprešiel autorizáciou. Dôvodom pre kód 400 v prvom prípade je ten, že k samotnej autorizácií ani nedošlo. Požiadavka nespĺňa daný formát, ktorý sa aplikuje pre hocijakú požiadavku, ktorá napríklad nemá ani autorizáciu.

4.3.3 Middleware

Middleware je akýsi prepojovací most, ktorý sa v prípade API vloží do každej požiadavky. V knižnica Sanic implementujúca REST API loguje jednotlivé požiadavky s istými informáciami. V tejto práci je ale na logovanie použitá iná knižnica, než ktorú používa Sanic. Je preto nutné prepísať logy tak, aby používali serverom definovaný štandard. Na to slúži funkcia *logger* v súbore *src/notifire/common/middleware.py*, ktorá sa vloží do každej prijatej požiadavky. Táto funkcia vytvorí log, ktorý nesie informácie o HTTP metóde požiadavky, jej telo, URL ceste, IP adrese klienta. Ak bola použitá HTTP metóda, ktorá prenáša dáta v tele požiadavky (POST, PUT, DELTE), tak tieto dáta sú zobrazené v logu. Kvôli bezpečnosti sú, samozrejme, tieto dáta filtrované. Ak je hodnota *Content-Length*, ktorá indikuje veľkosť dát v tela tejto požiadavky v bajtoch, z hlavičky tejto požiadavky väčšia než 5000, tak sa dáta nevypíšu do logu. Ak sa v tele správy nachádza parameter *password* alebo *api_key* tak sú z logu odignorované, aby ich nikto nemohol zneužiť. Vytvorí aj log s informáciami o následnej odozve, ktorý obsahuje stavový kód.

4.4 Endpointy

Endpoity sú rozdelené do troch skupín, v ktorej každá reprezentuje podobné funkcie. Prvá je account management (manažment účtu), implementácie endpointov sa nachádzajú v súbore *src/notifire/api_modules/account_management.py*. Nachádzajú sa tu všetky funkcie, ktoré spravujú účet používateľa. Druhá skupina je v súbore *src/notifire/api_modules/services.py*, ktoré spravujú používateľove services. Tretia skupina, kde je implementácia notifikačného endpointu, je *src/notifire/api_modules/notifications.py*.

Register

Endpoint prijíma parametre, jedná sa o email, prihlasovacie meno a heslo. Validita emailu a prihlasovacieho mena je overená regulárnym výrazom, ktorý sa nachádza v priečinku schém v súbore *register.py*. Pre všetky taktiež platí podmienka maximálnej dĺžky, ktorá je v schéme uvedená. Po prijatí správnych dát sa najprv zahašuje heslo pomocou knižnice **passlib**[9]. Následne sa uložia do databázy používateľove údaje vrátane zahašovaného hesla funkciou *register_user* zo súboru *src/notifire/components/account.py*. Táto funkcia je dekorovaná databázovým dekorátorom, ktorý do nej vloží databázové spojenie. Do databázovej

[9] https://passlib.readthedocs.io/en/stable

tabuľky sa následne vloží email, meno, zahašované heslo a refresh token, ktorý sa vytvorí knižnicou *uuid4* a vznikne 32 znakový hexadecimálny reťazec. Endpoint vracia používateľovi parameter *success*, ktorý má hodnoty buď *true* alebo *false*. Neúspech môže vzniknúť v dvoch prípadoch. Prvý je v prípade, že daný email alebo heslo už existuje, vtedy sa pri REST verzií API vráti okrem hodnoty false aj stavový kód 200. Kód 200 znamená úspešnú operáciu, avšak v tejto práci nie je už existujúci údaj považovaný za neúspech. V druhom prípade neúspechu je na vine sám server, ktorému sa nepodarilo vložiť nového používateľa do databázy. Tentokrát server vráti stavový kód 500, čo znamená chybu na servera. Ak je nový používateľ vložený do databázy, server vloží do fronty správ nový proces na poslanie overovacieho emailu používateľovi. Tento mechanizmus posielania emailu je asynchrónnny a nezdrží tak používateľa pri registrácií. Posielanie emailu je náročnejšia operácia a trvá pár sekúnd.

Login

Endpoint prijíma parametre username, pod ktorým je buď email alebo prihlasovacie meno, a heslo. Ak prejde validáciou, vytiahne z databázy používateľa na základe jeho emailu alebo mena. Následne overí jeho heslo tak, že heslo poslané pri logine zahašuje a potom porovná, či sa tieto haše rovnajú. Ak je heslo správne, vráti používateľovi jeho prihlasovacie meno, access token a refresh token. Access token sa vytvára práve v tejto funkcií. Vloží do neho jeho identifikátor a kombináciu hašu z emailu a hesla. Všetky tieto údaje má z databázy. Pri prihlasovaní môžu vzniknúť 3 neúspešné stavy. Prvý vznikne, keď databázové query nenájde používateľa na základe jeho emailu alebo mena. Vtedy vráti chybu, v ktorej je uvedené, že email alebo prihlasovacie meno neexistuje. V druhom prípade je nesprávne heslo a opäť vráti chybu používateľovi, v ktorej ho s týmto oboznámi. V poslednom prípade sú údaje v poriadku, ale používateľ nie je verifikovaný, to znamená, že si ešte neverifikoval email. Pri REST API vracajú všetky druhy odozvy stavový kód 200.

Password

Endpoint prijíma parametre heslo a nové heslo. Ak je používateľ autorizovaný, funkcia z access tokenu vytiahne jeho identifikátor a cez funkciu *user_password* na základe tohto identifikátora zistí jeho heslo. Prijaté heslo opäť zahašuje a haše oboch hesiel porovná. Ak sa nerovnajú, vráti neúspech. Ak je heslo správne, porovná či sa prijaté heslo nerovná tomu novému. Bolo by zbytočné meniť heslo za to isté. V prípade identického nového hesla opäť vráti neúspech. Ak je všetko správne, nové heslo sa zahašuje a uloží do databázy funkciou *reset_user_password* a vytvorí nový refresh token, aby sa nemohol prihlásiť pod starým. Pri každom databázovom zápise existuje riziko, že sa zápis nepodarí. V takom prípade funkcia vráti neúspech so stavovým kódom 500. Ak je zmena úspešná, vymaže sa z cache pamäti položka *user_secret* slúžiaca na invalidáciu existujúcich access tokenov pre daného používateľa, ktoré sú z technického hľadiska stále platné. Následne vráti používateľovi jeho prihlasovacie meno, nový access a refresh token. Nový access token sa vytvorí z nového hesla, ktoré používateľ práve zadal.

Device

Endpoint prijíma device token s dĺžkou do 256 znakov. Vyžaduje autorizáciu. Tento endpoint vytvára a priradzuje device tokeny iOS zariadení daným používateľom. Má dve úlohy. Prvá je vytvoriť nový device token a priradiť ho používateľovi. Druhá je zapnúť notifikácie

používateľa na danom zariadení, ak už token existuje. Tento stav môže nastať vtedy, ak sa používateľ odhlási a znovu bude chcieť prihlásiť. Pomocou funkcie *add_device* sa vykonávajú databázové operácie. Funkcia sa skladá z dvoch databázových queries, kde prvé najprv skúsi uložiť device token do tabuľky *device* a vráti jeho identifikátor. V prípade konfliktu (existencie) taktiež vráti identifikátor. Druhé query vloží do tabuľky *user_device* identifikátor používateľa a device tokenu, takto ich navzájom prepojí. Taktiež nastaví položku *active* na true, čo znamená zapnutie notifikácií. Táto funkcia používa databázovú transakciu. Databázová transakcia slúži na pri viacerých queries, ktoré potrebujú byť naraz vykonané. Buď sa vykonajú všetky požiadavky alebo žiadna. Ochraňuje integritu dát. V tomto prípade sa využíva na to, aby v prípade uloženia nového device tokenu bol tento token priradený používateľovi. Môže sa stať, že sa uloží nový device token a v nasledujúcom query na priradenie tokenu pod používateľa toto query zlyhá. V takomto prípade by existoval riadok v tabuľke *device*, na ktorý by neexistovala referencia. Transakcia zaručí, že sa tento stav nikdy nestane.

Notifikačná aplikácia na iOS zariadení volá tento endpoint po prihlásení používateľa, aby zaregistrovala jeho zariadenie a spustila prijímanie notifikácií. Vracia stavový kód 204, čo znamená úspešnú požiadavku, ale odozva servera neobsahuje žiadne dáta.

Logout

Endpoint prijíma device token. Vyžaduje autorizáciu. Odhlási používateľa pomocou funkcie *logout_user* tak, že nastaví položku *active* v tabuľke *user_device* na false. To vypne prijímanie notifikácií. Vracia stavový kód 204, čo znamená úspešnú požiadavku, ale odozva servera neobsahuje žiadne dáta.

Check

Endpoint prijíma email alebo prihlasovacie meno vo forme parametra a nie v tele požiadavky. Pomocou funkcie *check_user* pozrie, či email alebo meno už v databáze existuje. Server vráti v odozve hodnotu true alebo false.

Access

Endpoint prijíma refresh token. Nevyžaduje autorizáciu, pretože vytvára nový access token na základe refresh tokenu. Pomocou funkcie *access_user* cez refresh token vytiahne id, meno a heslo, z ktorých následne vytvorí nový access token, ktorý vráti v odozve používateľovi. V prípade chyby vo vnútri tohto endpointu sa nevráti access token a statusový kód bude 500. Chyba po verifikácií môže nastať len na strane servera, preto je použitý kód 500.

Send_confirm_email

Endpoint prijíma email alebo prihlasovacie meno vo forme parametra, nie v tele požiadavky. Tento endpoint slúži na zaslanie opätovného verifikačného emailu. Pri registrácií sa email pošle automaticky, avšak môže sa stať, že email nepríde alebo používateľ zmešká časový úsek platnosti emailového tokenu. Pomocou funkcie *get_user* sa na základe emailu alebo mena vytiahne z databázy jeho id, email a heslo, z ktorých sa následne vytvorí nový email token. Do tokenu sa zakóduje, že sa jedná o verifikáciu emailu používateľa, aby nemohol byť tento token zneužitý na iné účely. V prípade, že sa email alebo prihlasovacie meno v nenájde, funkcia vráti neúspech. Server vloží do fronty novú požiadavku na poslanie

emailu a okamžite vráti používateľovi odozvu. Endpoint je asynchrónny, čiže používateľ nečaká, kým sa pošle email.

Confirm_email

Endpoint prijíma emailový token cez telo požiadavky. Verifikácia endpointu overí formát tokenu na základe regulárneho výrazu, ktorý je v súbore *confirm_email.py* v schémach. Vyžaduje emailovú autorizáciu, ktorá vytiahne token z tela požiadavky a nie hlavičke ako pri autorizácií access tokenom. Tento endpoint slúži na samotnú verifikáciu používateľa. Používa funkciu *confirm_user_email*, ktorá na základe jeho identifikátora nastaví položku *verified* v tabuľke *user* na true, ak ešte nebol verifikovaný. V prípade, že už používateľ verifikovaný bol, endpoint vráti neúspech. Pri úspešnej verifikácií endpoint vráti access a refresh token. Tieto autorizačné údaje vráti, pretože hocijaká operácia vykonaná s pomocou emailu znamená skutočnú identitu používateľa a nie je dôvod zdržovať proces prihlásenia sa prostredníctvom prihlasovacieho endpointu.

Send_reset_email

Endpoint neprijíma žiadne dáta v tele požiadavky. Vyžaduje autorizáciu, pretože slúži na odoslanie resetovacieho emailu, ktorý zmení aktuálny email na nový. Resetovací email sa pošle na aktuálny, aby overil, že skutočný majiteľ účtu chce naozaj zmeniť jeho email. Pomocou funkcie *get_user* na základe jeho id získa id, email a heslo potrebné vytvorenie email tokenu. Do tokenu sa zakóduje, že sa jedná o resetovací email, aby tento token nemohol byť zneužitý. Do fronty správ sa vloží nový proces na zaslanie tohto emailu, ktorý sa pošle asynchrónne.

Reset_email

Endpoint prijíma nový email a emailový token v tele požiadavky. Vyžaduje emailovú autorizáciu. Pomocou funkcie *reset_user_email* zmení aktuálny email používateľa na nový, ale iba ak nie je ten istý. V prípade úspechu vráti nový access a refresh token. Staré tokeny invaliduje a tie strácajú platnosť. V prípade neúspechu vráti stavový kód 200, ak bol nový email rovnaký ako starý, alebo kód 500, ak sa stala chyba na strane servera.

Send_reset_password

Endpoint prijíma email cez parameter požiadavky. Slúži na obnovu strateného hesla a preto nevyžaduje autorizáciu. Pomocou funkcie *get_user* na základe emailu vytiahne id, email a heslo používateľa, ktorému pošle resetujúci email na zmenu hesla cez frontu správ. Do tokenu je zakódované, že sa jedná o reset hesla. Ak email existuje, vráti chybu. Následne sa používateľovi asynchrónne pošle email na reset hesla.

Reset_password

Endpoint prijíma nové heslo a emailový token v tele požiadavky. Vyžaduje emailovú autorizáciu. Nové heslo sa najprv zahašuje cez funkciu *hash_password*. Pomocou funkcie *reset_user_password* sa na základe identifikátora používateľa zmení jeho heslo na nové. Rovnako sa zmení aj jeho refresh token, aby nebolo možné získať access token. Pri úspešnej zmene sa invalidujú všetky predošlé access tokeny a používateľ dostane nový access a refresh

token. V prípade chyby, ktorá môže nastať iba na strane servera, je vrátený stavový kód 500.

Notify

Endpoint prijíma v tele požiadavky api kľúč, notifikačný level a správu, ktorá sa skladá z hlavičky, tela, url a textu. Nevyžaduje autorizáciu, tú poskytuje api kľúč. Ak požiadavka prejde verifikáciou, ktorá je špecifikovaná v schéme *notify.py*, tak endpoint najprv pomocou funkcie *notification_level* vytiahne zapnuté notifikačné levely pre daný service, ktorý je špecifikovaný api kľúčom. Ak nič nenájde, tak bude to znamenať, že api kľúč je neplatný a vráti používateľovi chybu.

Následne pomocou funkcie *user_device_token* na základe api kľúča vytiahne device tokeny. Táto funkcia obsahuje databázové query, ktoré vytiahne device tokeny z tabuľky *device*. Aby to bolo možné, je najprv potrebné spojiť tabuľku *device* a *user_device* na základe primárneho kľúča riadku pre *device*, na ktorý sa odkazuje *user_device*. Tabuľka *user_device* obsahuje odkaz na tabuľku *user* rovnako ako tabuľka *user_service*, s ktorou ho treba spojiť. Tieto dve tabuľky sa spoja pomocou primárneho kľúča používateľa. V tomto momente je možné na základe api kľúča nájsť id používateľa v tabuľke *user_service*. Pomocou tohto id vieme nájsť všetky jeho riadky v tabuľke *user_device*. Pomocou device id, ktoré je teraz k dispozícií, je možné zistiť všetky device tokeny, ktoré reprezentujú používateľove iOS zariadenia. Podmienka je, že v tabuľke *user_device* má položka *active* hodnotu true, čo predstavuje prijímanie notifikácií na danom zariadení. V prípade, že je hodnota false, query odignoruje device token pod daným používateľom.

V tomto momente má endpoint k dispozícií všetky zariadenia, ktoré je potrebné notifikovať. Nasleduje filtrácia notifikačných levelov, ktoré daný service aktuálne prijíma. Notifikačné levely sú v databáze uložené ako číslo. Notifikačný level, ktorý endpoint prijíma, je reťazec. Funkcia *transform_notification_levels* transformuje toto číslo na slovník (dictionary), tá sa importuje zo súboru *src/notifire/utils/data_transformation.py*. Každý z troch notifikačných levelov je reprezentovaný mocninou čísla 2 [9]. Level *info* je reprezentovaný ako 2^0, *warning* ako 2^1 a *error* ako 2^2. Notifikačné levely, ktoré má daný service zapnuté, sú v databáze reprezentovane sumou jednotlivých levelov. Táto suma je následne transformovaná pomocou bitových operácií do dictionary, kde kľúč je názov notifikačného levela a hodnota je buď true alebo false. Endpoint porovná, či sa level obdržanej notifikácie zhoduje s nastavením. V prípade, že daný notifikačný level je vypnutý, server odignoruje ďalšie spracovanie a notifikácia sa teda zahodí.

Ak je všetko v poriadku, naformátuje sa notifikačná správa pre APNs, do ktorej sa vložia notifikačné dáta od používateľa. Server do nej navyše vloží aktuálny dátum čas vo formáte UTC a skontroluje veľkosť správy, ktorú APNs definuje. Ak veľkosť prekročí hranicu, server vráti v odozve chybu so stavovým kódom 400. Správa sa vloží do fronty a používateľ dostane prázdnu odpoveď. Endpoint je vďaka vloženiu správy do fronty asynchrónny.

Services

Endpoint vyžaduje autorizáciu a neprijíma žiadne dáta. Vracia list jednotlivých serviceov, ktoré má používateľ vytvorené. Každý service je v tomto liste vo forme dictionary (slovníka) a obsahuje meno, api kľuč, notifikačné levely. Tieto dáta získa pomocou funkcie *user_services* na základe id používateľa. Notifikačné levely sú však v číselnej forme, preto sú rovnako ako u 4.4 prevedené na typ dictionary.

Service

Pri REST API je pre tento endpoint možné použiť 3 HTTP metódy (POST, PUT, DELETE). Endpoint vyžaduje autorizáciu. Pri metóde POST endpoint prijíma názov a notifikačné levely. Táto metóda vytvára nový service. Najprv sa pomocou zabudovanej knižnice *secrets* vytvorí api kľúč. Knižnica vo svojej dokumentácií uvádza, že je vhodná na použitie pri účeloch ako napríklad bezpečný token, čo v tejto práci predstavuje api kľúč. Následne sa transformujú notifikačné levely z typu dictionary do čísla, ktoré sa uloží do databázy. Funkcia *transform_notification_levels* prevedie levely s hodnotami true na mocninu čísla 2 podľa definície z triedy *NotificationLevel* zo súboru konštánt. Následne sa vytvorí suma týchto levelov. Pomocou funkcie *add_user_service* sa vložia tieto hodnoty do databázy. Ak nastane pri databázovej operácií chyba, server vráti chybu so stavovým kódom 500.

Pri metóde PUT endpoint prijíma názov, levely a api kľúč. Táto metóda sa mení nastavenie jedného service-u na základe prijatých dát. Po úspešnej verifikácií sa opäť transformujú notifikačné levely do číselnej podoby. Pomocou funkcie *update_user_service* sa na základe api kľúča a id používateľa uložia nové hodnoty do databázy. Pre úspešný zápis je potrebné, aby bol názov alebo notifikačný level iný ako aktuálny. Ak nastane pri databázovej operácií chyba, server vráti chybu so stavovým kódom 500.

Pri metóde DELETE endpoint prijíma tie isté dáta ako pri metóde PUT. Táto metóda vymaže service z databázy. Tieto dáta síce nie sú potrebné, ale vyžadujú sa z bezpečnostných dôvodov. Pomocou funkcie *delete_user_service* sa na základe api kľúča a id používateľa vymaže service z databázy. Ak nastane pri databázovej operácií chyba, server vráti chybu so stavovým kódom 500.

Ak pri žiadnej z týchto metód nenastane chyba, endpoint vráti úspešnú odozvu.

Api_key

Endpoint prijíma názov, notifikačné levely a api kľúč pre daný service. Endpoint vyžaduje autorizáciu. Slúži na zmenu api kľúča. Hoci je potrebný len api kľúč, z bezpečnostných dôvodov sa vyžadujú aj ostatné údaje. Pomocou funkcie *update_service_api_key* sa na základe starého kľúča a id používateľa vytvorí nový kľúč. Ak nastane pri databázovej operácií chyba, server vráti chybu so stavovým kódom 500, inak vráti úspešnú odozvu.

4.4.1 GraphQL rozdiely endpointov

Implementácia GraphQL je v súbore *src/notifire/graphql.py*. Samotný GraphQL implementuje v Pythone knižnica **graphene**[10]. Knižnica **graphene-sanic**[11] umožňuje implementovať GraphQL v rámci frameworku Sanic. Kombinuje Graphene a Sanic.

Endpointy, ktoré pri webových požiadavkách nevyžadujú žiadne databázové operácie alebo na svoju funkciu vyžadujú databázové query typu SELECT, sú v GraphQL reprezentované typom **Query**. Naopak, endpointy, ktoré vykonávajú zmenu dát v databáze, napríklad query INSERT, UPDATE, DELETE, reprezentujú typ **Mutation**. Endpointy popísané v sekcií 4.4, sú funkcie v kóde, ktoré sú spoločné pre REST API a aj GraphQL. V prípade REST API sú tieto funkcie volané priamo. Pri GraphQL požiadavke sa na jeden jediný endpoint v tele správy špecifikuje táto funkcia, reprezentovaná mutáciou alebo typom query. V kóde sa zavolá trieda query alebo trieda mutácia, ktorá obsahuje metódu pod týmto menom, a až v nej sa zavolá funkcia s predošlej sekcie implementujúca logiku,

[10] https://github.com/graphql-python/graphene
[11] https://github.com/ethe/graphene-sanic

ktorá sa pri RESTe volá priamo. Rovnako ako pri RESTe, funkcia prijíma dáta. V prípade GraphQL sa však dáta vkladajú do zátvorky ohraničujúcej meno tejto funkcie. Funkcia dáta verifikuje ich tým istým spôsobom ako pri RESTe. Rozdiely oproti REST API sú:

- Všetky požiadavky majú jeden jediný endpoint, konkrétne *localhost/graphql*.
- Endpointy predstavujú funkcie, ktoré sa volajú na základe ich mena v tele požiadavky.
- Všetky webové požiadavky používajú HTTP metódu POST.
- V tele požiadavky je nutné špecifikovať, o ktoré dáta má klient záujem.
- Stavové kódy sú **vždy** 200, aj pri chybe.

V súbore *graphql.py* sú dva druhy tried. Prvý druh je trieda *Query* a zahŕňa všetky endpointy, kde je nie je vyžadovaná databáza alebo iba query typu SELECT. Jedná sa o endpointy *login*, *check*, *access*, *send_confirm_email*, *send_reset_email*, *send_reset_password*, *services*, *notify*. Druhá druh trieda *Mutation*, ktorú dedia všetky jednotlivé endpointy predstavujúce mutáciu, teda zmenu dát v databáze. Každý endpoint je vlastná trieda a dedí vlastnosti triedy *Mutation*. Oba druhy tried obsahujú premenné triedy, ktoré reprezentujú vstupné dáta. Každá premenná musí mať definovaný typ vstupných dát. V prípade mutácie je prítomná podtrieda *Arguments*, kde definovaný výstupný typ jednotlivých dát. Použité typy sú string (reťazec), boolean (true/false), scalar (skalárny typ). Vstup aj výstup môže obsahovať dáta vo formáte JSON, a preto je implementovaná trieda *JSONScalar*, ktorá dedí a rozširuje typ scalar, aby bolo možné prijať tento formát.

Výsledok pri GraphQL je na konci rovnaký ako pri REST API a teda výsledná funkcionalita sa nemení. Je len na používateľovi, ktorú formu API si zvolí.

4.5 Caching

Aby malo cachovanie dát zmysel, je potrebné použiť rýchlu vyrovnávaciu pamäť. Na tieto účely je použitá technológia **Redis**[12]. Jedná sa o NoSQL databázu [8], ktorá ukladá dáta do RAM pamäte. Na komunikáciu s Redisom je použitá knižnica **aioredis**[13]. Implementácia pripojenia na Redis je v súbore *src/notifire/databases.py* ako trieda *RedisPoolManager*. Rovnako ako pri pripájaní na databázu PostgresQL, aj pri Redise je pripájanie formou takzvaného Connection Pool 4.2.2. Pri zapínaní servera sa vytvorí inštancia tejto triedy, ktorá nadviaže s Redisom spojenie. Následne je možné importovať túto inštanciu a narábať s dátami v tejto databáze. Dáta sa do Redisu ukladajú formou kľúča a jeho hodnoty.

Cachovanie sa používa pri invalidácií JWT tokenov. Funkcia *verify_jwt_token* v súbore *src/notifire/security.py* porovnáva parameter *user_secret* z prijatého tokenu. Okrem časovej platnosti má JWT token v tejto práci platnosť obmedzenú k danému emailu a heslu. Akonáhle sa zmení email alebo heslo, token stráca platnosť. Na to slúži *user_secret*, ktorý v sebe drží tieto údaje v zahašovanej forme. Každý používateľ môže mať len jednu správnu kombináciu. Táto funkcia importuje inštanciu triedy *UserSecret* zo súboru *src/notifire/common/cache.py*.

Keď server obdrží webovú požiadavku na zabezpečený endpoint, pokúsi sa porovnať *user_secret* s tým, ktorý je pre daného používateľa v uložený v pamäti cache (Redise).

[12] https://redis.io/
[13] https://github.com/aio-libs/aioredis

Proces začne tým, že sa server snaží vytiahnuť z Redisu tento parameter cez identifikátor používateľa, ktorý sľuži ako kľúč. Jedná sa o metódu *get* tejto inštancie. Ak sa v pamäti Redis na základe kľúča nenájde jeho hodnota (user secret), tak sa zavolá metóda *refill_cache*. Tá zavolá ďalšiu metódu *load_from_source*, ktorá vezme tieto údaje z databázy, zahašuje kombináciu emailu a hesla a vráti ich späť. Následne sa tieto dáta uložia do cache pamäti pod daným kľúčom (id používateľa) a sú vrátené do funkcie *verify_jwt_token*, ktorá ich porovná. Dáta uložené do tejto pamäti môžu mať obmedzenú časovú platnosť, počas ktorej si ich Redis bude držať. To je vhodné na to, aby sa pamäť nezaplnila, napríklad ak sa prihlási veľa používateľov. Časovú platnosť určuje premenná *user_secret_expiry* zo súboru konštánt a jej hodnota je 12 hodín.

Každý endpoint vytvárajúci zmenu emailu alebo hesla volá metódu *delete_from_cache*, ktorá na základe kľúča (identifikátor používateľa) vymaže jeho hodnotu (user secret). Týmto je zaručené, že pri porovnávaní parametra *user_secret* pri verifikácií JWT tokenu bude musieť server znovu načítať dáta z databáze. Tým sa teda overí, či je token po zmene údajov platný.

Funkcia a implementácia cache pamäti nie je v tejto práci potrebná. Dá sa to vyriešiť neustálym vyťahovaním dát z databázy. To by znamenalo spomalenie každej zabezpečenej webovej požiadavky. Vďaka cachovaniu sa tento problém vyrieši a načítanie z databázy je potrebné len každých 12 hodín (voliteľné) alebo pri zmene emailu či hesla. Čím viac používateľov existuje, tým menšia by mala byť trvácnosť dát v cache pamäti, pretože sa môže ľahšie zaplniť.

4.6 Distribuovaná fronta správ

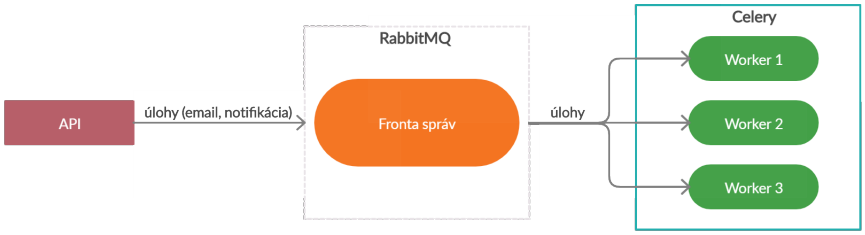
Fronta správ je implementovaná pomocou technológie **RabbitMQ**[14]. RabbitMQ poskytuje viacero funkcií, v tejto práci je využitý na implementáciu fronty správ [14]. RabbitMQ nie je jediná technológia, ktorá implementuje frontu správ. **Redis** taktiež implementuje túto funkcionalitu. Použitie ďalšej technológie (RabbitMQ) zvyšuje komplexnosť celej práce a nie je to ani nutné, pretože Redis sa už používa a môže byť využitý aj na implementáciu fronty správ. Dôvodom zvolenia RabbitMQ je to, že dáta po výpadku, respektíve vypnutí, nestratí. RabbitMQ zabezpečuje perzistenciu dát, čo Redis nedokáže. Druhý dôvod je ukázať v práci schopnosť pracovať s ďalšími alternatívnymi technológiami.

RabbitMQ dokáže prijímať správu od takzvaného producenta a následne ju dokáže konzument vybrať 4.4. Notifikačný server predstavuje rolu producenta, konkrétne *notify* API endpoint, ktorý do tejto fronty vkladá správy (notifikácie), ale aj endpointy posielajúce email, ktoré do fronty vkladajú emaily. Rolu konzumenta predstavuje opäť notifikačný server. Konzument musí byť nezávisle bežiaca aplikácia, ktorá vyťahuje správy z tejto fronty a obslúži ich. RabbitMQ slúži na prenos týchto správ, jedná sa o samotnú frontu, dokáže však správy prijímať a konzumentom ich vyťahovať.

[14] https://www.rabbitmq.com/

Obr. 4.4: Fronta správ v RabbitMQ, ich producenti a konzumenti

Funkciu konzumenta je nutné implementovať a na to slúži knižnica **celery**[15]. Táto knižnica rozširuje funkcionalitu fronty správ na takzvanú frontu úloh [15]. Funguje tak, že cez túto knižnicu sa vloží do fronty nová úloha (task). Nad touto frontou operujú pracovníci (workeri). Workeri konštantne monitorujú túto frontu a čakajú na nové úlohy, ktoré následne vykonajú. Tento princíp sa využíva na vykonávanie úloh pod vlastnými procesmi, ktoré môžu prebiehať na viacerých zariadeniach a teda distribuovať úlohy. Umožňuje separáciu niektorých úloh, ktoré by muselo vykonávať API notifikačného servera. Celery zabezpečuje rolu producenta správ, ale aj konzumenta. Producent je síce API notifikačného servera, ale samotné vloženie úlohy do fronty je realizované cez rozhranie knižnice Celery. Konzumenti sú workeri, ktorí čakajú na úlohy. Knižnica Celery na distribúciu správ (úloh) od producenta ku konzumentovi používa na prenos RabbitMQ. Celery vloží úlohu do fronty RabbitMQ a ten ju odovzdá workerovi 4.5.



Obr. 4.5: Proces distribúcie úlohy

Implementácia sa nachádza v súbore *src/notifire/celery.py*. Celery vyžaduje isté nastavenia a aj adresu na RabbitMQ, na ktorý sa pripojí. Premenné s jednotlivými nastaveniami importované zo súboru *src/notifire/settings.py*. Úlohy, ktoré obsluhujú workeri, sa nachádzajú v priečinku *src/notifire/tasks*, v ktorom súbor *email.py* obsahuje implementáciu posielania emailu a *apple_notification.py* posielania notifikácie do APNs. Obe tieto úlohy sú vykonávané workerom asynchrónne a bežia nezávisle od API notifikačného servera. Pri vkladaní úloh do fronty sa zavolajú tieto funkcie a vložia sa do nich potrebné údaje. Údaje sú v textovej podobe, pretože inštancia Celery je vlastný proces a teda nie je možné do nej prenášať údaje vo forme Python objektu.

4.7 APNs konektor

Funkcia tohto konektoru je pripojenie sa na systém APNs vo forme trvalého HTTP/2 spojenia . Pri posielaní notifikácií do APNs sa nesmie vytvoriť a zatvoriť veľa nových pripojení

[15] https://github.com/celery/celery

s APNs naraz, pretože APNs to bude považovať za DDoS útok [2]. To by znamenalo zablokovanie prístupu tejto služby pod IP adresu servera a teda nebolo by možné posielať notifikácie. Aby bolo možné posielať veľa notifikácií naraz, HTTP/2 umožňuje v jednom pripojení mať naraz viacero takzvaných streamov. Jedna inštancia HTTP/2 pripojenia povoľuje viacero nezávislých streamov, ktoré posielajú dáta. Jedná sa mechanizmus podobný webovej požiadavke, v tomto prípade sa posielajú cez zabezpečené a trvalé HTTP/2 spojenie. Na zabezpečenie optimálneho prietoku notifikácií je v tejto práci implementovaný mechanizmus, ktorý vie vytvoriť niekoľko pripojení, v ktorých každé jedno pripojenie dokáže mať naraz až niekoľko tisíc streamov.

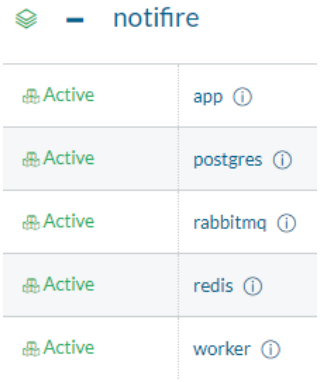
Implementácia sa nachádza v súbore *src/notifire/apns*. V súbore *client.py* sú dve triedy. Trieda *APNSClient* reprezentuje trvalé HTTP/2 pripojenie. Pri vytvorení inštancie tejto triedy sa nadviaže HTTP/2 spojenie s APNs pomocou URL cesty, portu a certifikátu, ktorý je vydaný Applom pre identifikáciu notifikačnej aplikácie. Toto spojenie prenáša dáta cez streamy, ktorých počet je určený konštantou *max_initial_apns_streams*. Jeden stream dokáže v danom čase preniesť jednu notifikáciu. Táto trieda importuje semafór zo súboru *semaphore.py*, ktorý implementuje logiku obsluhy streamov. Ak príde naraz veľa notifikácií, tak jedna zodpovedá jednému streamu. Streamov je obmedzený počet a ak sa všetky zaplnia, tak notifikácia bude čakať, kým sa uvoľní stream. Semafor slúži na blokovanie a odblokovanie streamu. Ak príde nová notifikácia, zníži sa počet voľných streamov a v momente, kedy sa notifikácia pošle, stream sa uvoľní. Metóda *send_async_notification* pošle notifikáciu do APNs. Na dodržanie správneho formátu daného Applom sa do metódy vkladá argument *notification*, ktorý je jedna inštancia triedy *Notification* zo súboru *payload.py*. Obsahuje správu vo formáte JSON naplnenú notifikačnými údajmi. Jej veľkosť nesmie presiahnuť veľkosť stanovenú danú Applom, ktorú reprezentuje konštanta *max_apns_message_size*. Správa sa nakoniec pošle cez stream na URL adresu, ktorú reprezentuje device token a teda konkrétne zariadenie.

Trieda *APNSConnectionPool* implementuje princíp connection poolu [12], ktorý sa využíva pri databáze a cachingu. Drží si v sebe inštancie triedy *APNSClient*, čo sú jednotlivé HTTP/2 pripojenia. Proces začne tak, že Celery worker dostane úlohu (poslanie notifikácie). Worker má k dispozícií inštanciu tejto triedy, cez ktorú zavolá metódu *send_notification_batch*. Tá dostane cez argument device tokeny a notifikáciu, ktorá sa na všetkých týchto zariadeniach má objaviť. Následne sa pokúsi použiť už existujúce pripojenie s APNs, ktoré je voľné. Voľné pripojenie znamená, že pripojenie má k dispozícií voľné streamy. Ak sú všetky streamy plné, pokúsi sa vytvoriť nové pripojenie. Pri vytváraní nového pripojenia sa využíva mechanizmus multiprocesingu, ktorý sa volá Lock (zámok). Lock slúži na to, aby sa jedna konkrétna udalosť nemohla vykonať z rôznych procesov alebo vlákien viackrát naraz. Môže sa stať, že viacerí workeri, ktorí predstavujú niekoľko procesov, môžu naraz chcieť otvoriť APNs pripojenia. Teoreticky by sa mohlo stať, že sa všetci v rovnakom čase ocitnú na tom istom riadku v kóde, ktorý by otváral nové pripojenia v prípade, že žiadne nie je otvorené alebo by boli všetky plné. Ak by sa nepoužil Lock, tak by sa naraz otvorilo viacero pripojení, čo by mohlo pre Apple znamenať DDoS útok. Lock zablokuje všetkým procesom, okrem toho prvého, prejsť cez daný riadok kódu. V tomto momente všetci až na jedného čakajú. Worker, ktorý prešiel cez Lock vytvorí nové APNs pripojenie zavolaním inštancie *APNSClient*, ktorú si uloží do prememnej *connections*. Následne sa zámok odblokuje a ďalší worker (proces) môže pokračovať. V tomto momente vzniklo nové pripojenie, ktoré logicky ešte nemá zaplnené svoje streamy. To znamená, že prvý worker môže začať odosielať notifikácie. Ten druhý si cez podmienku overí, či je k dispozícií voľné pripojenie. Ak je, tak neotvorí ďalšie a pošle notifikácie cez to, ktoré mu bolo pridelené.

Vytváranie nových pripojení ma svoj limit, ktorý reprezentuje konštanta *max_connections*. V prípade, že worker nemá k dispozícií voľné pripojenie a maximálny počet pripojení už existuje, musí počkať, kým sa uvoľní stream v niektorom z nich.

4.8 Docker

Notifikačný server tvorí týchto 5 služieb 4.6. Každá beží na vlastnom virtuálnom operačnom systéme, zabalená od vonkajšieho sveta do takzvaného kontajnera. Docker vytvára kontajnerizáciu, ktorá zabalí danú službu do separátneho kontajneru. Docker umožňuje komunikáciu kontajnerov medzi sebou a svojím mechanizmom poskytuje jednoduchý deployment a škálovanie. Každá služba sa zapína cez nejaký príkaz a na svoju funkcionalitu potrebuje dostať isté premenné. To zabezpečuje funkcia Dockeru nazývajúca sa **docker-compose**. Umožňuje jednoduché spustenie kontajneru s danou službou. Súbor *docker-compose.yml* obsahuje všetky potrebné parametre na spustenie jednotlivých služieb notifikačného systému.



Obr. 4.6: Notifikačný server

App

Jedná sa o časť notifikačného systému, ktorá implementuje API (REST, GrahpQL). Na vytvorenie Docker kontajneru potrebuje súbor *Dockerfile*, ktorého cesta je špecifikovaná pod parametrom *build*, v ktorom je cesta k tomuto súboru. API na svoju funkcionalitu potrebuje údaje, tie predstavujú súbor *src/notifire/settings.py*. Tieto premenné sú načítané do kontajneru cez takzvané environment variables. Tie musia byť prítomné v aktuálnom prostredí, z ktorého sa zapínajú alebo je tu špecifikovaná cesta k súboru, kde sú jednotlivé hodnoty. Parameter port určuje, na akom porte bude bežať API, v prípade tejto práce je port 80. Parameter *command* reprezentuje príkaz, pod ktorým sa spúšťa API server.

Db

Tento kontajner reprezentuje databázu. Používa Dockerfile, ktorý implementuje samotnú databázu. Keďže Dockerfile nie je prítomný v zdrojovom kóde, získava sa zo služby Docker Hub, kde je uložený. Ten je špecifikovaný v parametri *Image* a jedná sa o PostgreSQL verzie 10. Port určuje port, pod ktorým beží databáza. *Environment* sú environment variables (premenné prostredia), ktoré databáza k svojej funkcií potrebuje mať. Parameter *volumes* pridáva do kontajneru fyzický disk, ktorý sa nachádza na systéme, v ktorom je kontajner

spustený. Jedná sa o prepojenie disku medzi vonkajškom a vnútrom (kontajnerom). Slúži na zdieľanie dát. V prípade tejto práce sú prepojené dve zložky. Prvá ukladá dáta z databáze mimo svojho kontajneru, aby sa v prípade reštartu alebo výpadku dáta nestratili. Druhý volume vkladá sql schému zo súboru *sql/schema.sql* do kontajneru. To pri úplne prvom štarte tohto databázového kontajneru automaticky vytvorí tabuľky a triggery, ktoré sú v jazyku sql implementované v tomto súbore.

Rabbitmq

Používa Dockerfile zo služby Docker Hub, ktorý implementuje RabbitMQ verzie 3.7.7. Špecifikované sú dva porty, ktoré RabbitMQ potrebuje na svoju funkcionalitu a rovnako aj environment variables. Podobne ako pri PostgreSQL, prepája dve zložky na disku s vonkajším prostredím kvôli perzistencií dát v prípade výpadku, vypnutí a podobne.

Redis

Používa aktuálny Redis Dockerfile načítaný zo služby Docker Hub. Žiadne ďalšie parametre nie sú na jeho spustenie potrebné.

Worker

Jedná sa o celery workerov. Príkaz *«: *app* kopíruje nastavenia z *App* kontajneru, ktorý implementuje API. Paremeter *command* reprezentuje príkaz na spustenie tejto služby. Celery je nastavené tak, že sa workeri prispôsobujú náporu. Hodnota *10, 3* na konci príkazu znamená, že 10 workerov je maximálny možný počet, 3 je minimum. Workeri teda vznikajú a zanikajú dynamicky za behu.

Kapitola 5

Testovanie

Jednotlivé komponenty systému sú obsiahnuté vo funkciách obsluhujúcich jednotlivé endpointy. Ak funguje notifikačný systém, tak musí fungovať aj každý endpoint. Databáza PostgreSQL, cache pamäť Redis, fronta správ RabbitMQ, ale aj workeri pracujúci nad touto frontou cez knižnicu Celery, sú nepriame súčasti notifikačného systému. Bežia zvlášť a teda je nie absolútne zaručené, že sa nikdy nestane chyba. Priama súčasť systému je API.

Testovanie API prebieha v dvoch fázach. Prvá je verifikácia už spomenutá v 4.3.1. Tým je zaručená úplná kontrola nad nechcenými dátami, ktoré môže systém prijať. Druhá forma testu je skúška vstupu a výstupu endpointov. Týmto testovaním sa otestuje aj funkčnosť nepriamych súčastí systému. Funkcionalita týchto nepriamych súčastí je obsiahnuta v endpointoch. To znamená, že všetky endpointy dokopy využijú každú funkciu, ktorú jednotlivé komponenty poskytujú. Napríklad endpoint pre registráciu používateľa využíva funkciu databázy. Notifikačný endpoint databázu a frontu správ. Priame testovanie nepriamych súčastí nie je potrebné. Napríklad testovanie databázy PostgreSQL je vykonané jeho developermi, podobne to platí aj u ostatných vymenovaných nepriamych súčastí. V prípade notifikačného systému sa testujú jeho priame súčasti a aj ich komunikácia s nepriamymi.

Testovanie tak prebieha na základe testu vstupu a výstupu každého endpointu. Ak vstupy a výstupy všetkých endpointov prebehnú správne, je možné považovať systém za funkčný. Testy vstupov a výstupov obsahujú okrem správnych príkladov aj nesprávne. Nesprávne testujú, či je systém stabilný aj pri zlých dátach, a či spĺňa vlastnosti návrhu. Testovanie API prebieha nie len cez REST ale aj GraphQL.

Testy vstupov a výstupov API sú v prílohe B. Jedná sa o snímky obrazovky a každá jedna webová požiadavka je autentická.

Kapitola 6

Záver

Cieľom tejto práce bolo implementovať back-endovú časť notifikačného systému. Prihlasovací systém obsahuje všetky potrebné funkcie pre manažment účtu používateľa ako aj dodatočné emailové procesy (verifikácia, zabudnuté heslo). Systém obsahuje frontu správ, ktorá vytvára endpointy asynchrónnymi a teda nespomaľuje používateľa, respektíve jeho softvér, ktorý odosiela notifikáciu. Agregácia správ je dosiahnutá rozdelením notifikácií do takzvaných services (služieb), ktoré od seba odlišujú jednotlivé softvéry (aplikácie) používateľa. Umožňujú odlíšiť notifikácie v iOS aplikácií ako aj na obrazovke zariadenia v momente, keď sa objaví notifikácia. Odosielanie správ na iOS zariadenie je dosiahnuté komunikáciou s APNs. Tento komunikačný konektor obsahuje implementáciu, ktorá kompletne vystihuje požiadavky Applu a zaručuje pokojný priebeh notifikácie bez prípadných chýb. Jedna z eliminácií prípadných chýb je časté otváranie nových spojení s APNs, čo by Apple vyhodnotil ako DDoS útok. Konektor a aj notifikačný endpoint dosahujú vysokú rýchlosť odbavenia notifikácií, taktiež umožňujú vysoký prietok notifikácií. To je zaručené škálovateľnou frontou správ a veľkým množstvom postupne otváraných pripojení na APNs podľa záťaže. Webové požiadavky používateľa prichádzajú do systému cez REST alebo GraphQL API. Obe varianty sú implementované a používateľ si môže vybrať, s ktorou sa mu pracuje lepšie. Funkcionalita sa pri oboch druhoch API nemení. Ich porovnanie je dostupné z implementačného hľadiska, ako aj z pohľadu webových požiadaviek.

Vytvorené riešenie viac ako dostačujúce, zlepšenie je však možné v prípade širšieho použitia alebo skvalitnenia istých funkcií. Jedna z vhodných zmien do budúcna je zmeniť endpointy pre services (služby) na socketovú komunikáciu s iOS aplikáciou. To by umožnilo pozerať a meniť services v reálnom čase na viacerých zariadeniach. Okrem toho by sa zamedzila nechcená možnosť takzvanej duálnej zmeny. Ak používateľ na jednom zariadení zmení nastavenie jedného service-u (služby), na druhom zariadení sa pod tým istým účtom táto zmena neukáže, kým používateľ neobnoví stránku. Na druhom zariadení by teoreticky mohol opäť zmeniť niečo, čo už medzitým spravil na prvom zariadení, ale zmenu kvôli neobnoveniu stránky nevidel. Takto by prepísal prvú zmenu. Ďalšie zlepšenie by bola implementácia takzvaného pagination. Pagination je technika načítania dát po častiach a nie ako celok. V kontexte tejto práce by bola aplikovateľná na načítanie services. Tie by sa načítali na základe daného filtra po častiach a znížili by tak prietok dát a čas odozvy.

Literatúra

[1] *APNs Overview* [online]. Apple. revidované 2018-06-04. [navštívené 2020-05-1]. Dostupné z: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html.

[2] *Communicating with APNs* [online]. Apple. revidované 2018-06-04. [navštívené 2020-05-1]. Dostupné z: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/CommunicatingwithAPNs.html.

[3] *Firebase Authentication* [online]. Google. [navštívené 2020-05-10]. Dostupné z: https://firebase.google.com/docs/auth.

[4] *Introduction to JSON Web Tokens* [online]. Jwt.io. [navštívené 2020-05-11]. Dostupné z: https://jwt.io/introduction/.

[5] *Schemas and Types* [online]. The GraphQL Foundation. [navštívené 2020-05-10]. Dostupné z: https://graphql.org/learn/schema/.

[6] *Table Relationships* [online]. Launch School. [navštívené 2020-05-10]. Dostupné z: https://launchschool.com/books/sql/read/table_relationships#manytomany.

[7] *Base64* [online]. Wikipedia, 2020. [navštívené 2020-05-12]. Dostupné z: https://en.wikipedia.org/wiki/Base64#URL_applications.

[8] *NoSQL* [online]. Wikipedia, 2020. [navštívené 2020-05-12]. Dostupné z: https://en.wikipedia.org/wiki/NoSQL.

[9] Bielik, D. A. *Design and implementation of an iOS notification system [online].* 2019 [cit. 2020-05-15]. Bakalářská práce. Masarykova univerzita, Fakulta informatiky, Brno. Vedúci práce Bühnová, B. Dostupné z: https://is.muni.cz/th/c5lo5/.

[10] Bisht, A. *Difference between Primary key and Unique key* [online]. Geeksforgeeks. [navštívené 2020-05-10]. Dostupné z: https://www.geeksforgeeks.org/difference-between-primary-key-and-unique-key/.

[11] Chen, Z., Guo, S., Duan, R. a Wang, S. Security Analysis on Mutual Authentication against Man-in-the-Middle Attack. In: *2009 First International Conference on Information Science and Engineering.* December 2009. DOI: 10.1109/ICISE.2009.1051. ISSN 2160-1291.

[12] Marton, V. *Database Tutorial – Single Connection vs Connection Pool* [online]. Guidearea. [navštívené 2020-05-10]. Dostupné z: https://www.guidearea.com/best-database-practices-single-connection-vs-connection-pool/.

[13] Siddiqui, A. *Authentication vs Authorization* [online]. Medium. [navštívené 2020-05-11]. Dostupné z: https://medium.com/datadriveninvestor/authentication-vs-authorization-716fea914d55.

[14] Tišnovský, P. *RabbitMQ: jedna z nejúspěšnějších implementací brokera* [online]. Root.cz, december 2018. [navštívené 2020-05-15]. Dostupné z: https://www.root.cz/clanky/rabbitmq-jedna-z-nejuspesnejsich-implementaci-brokera/.

[15] Ximenes, F. *Celery: an overview of the architecture and how it works* [online]. Október 2017. [navštívené 2020-05-15]. Dostupné z: https://www.vinta.com.br/blog/2017/celery-overview-archtecture-and-how-it-works/.

Príloha A

Obsah priloženého CD

Štruktúra priečinku:

- **/doc** Priečinok s dokumentáciou
- **/src** Priečinok so zdrojovými súbormi
- **README.md** súbor s inštrukciami na spustenie

A.1 Spustenie notifikačného servera

Na spustenie servera je potrebné nainštalovať Docker, Docker Compose. Notifikačný server sa zapne spustením príkazu:

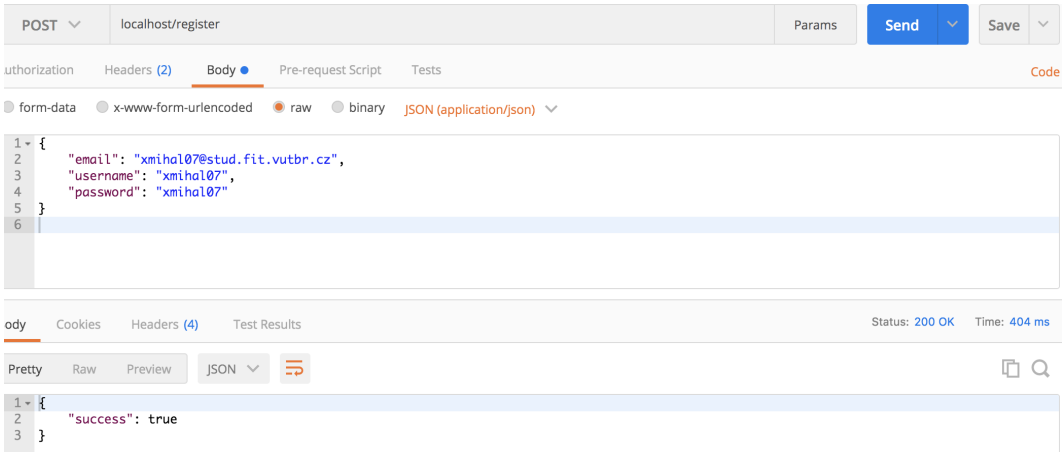
```
$ docker-compose -f docker-compose-dev.yml up --build
```

K plnej funkcionalite servera je vyžadovaný Apple certifikát, ktorý slúži na identifikáciu aplikácie. Z bezpečnostných dôvodov nie je súčasťou práce a teda notifikačný server spustený lokálne nebude schopný notifikácie posielať. Notifikačný server je nadizajnovaný a implementovaný produkčným spôsobom, nie je vytvorený pre plnú funkcionalitu v prípade lokálneho používania.

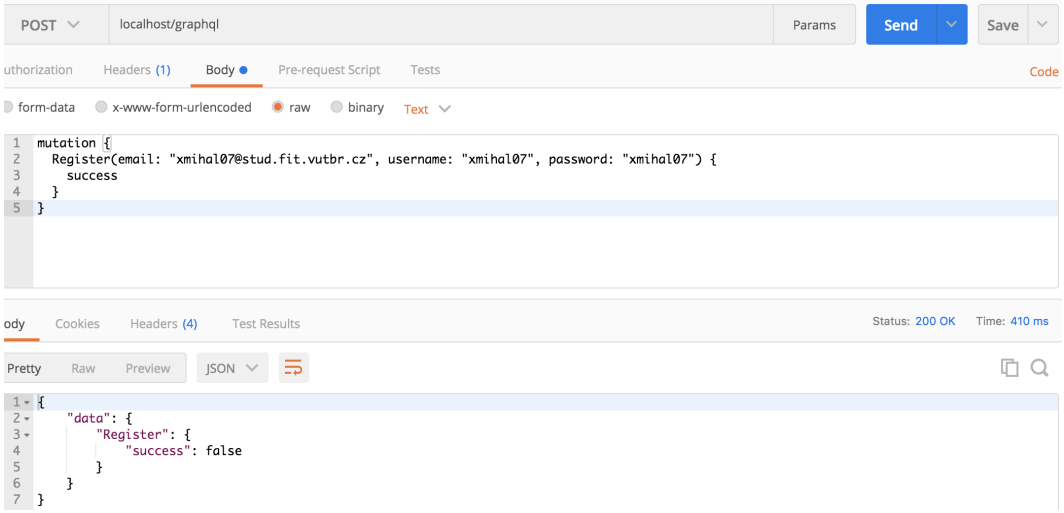
Príloha B

Vstupy a výstupy REST a GraphQL API endpointov

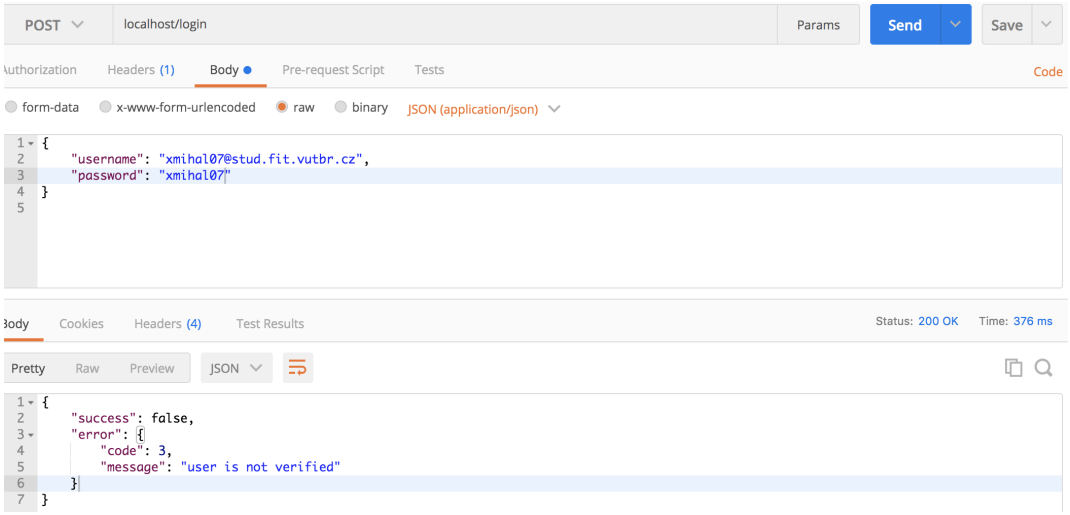
Nasledujúce snímky obrazovky reprezentujú webové požiadavky na jednotlivé endpointy oboch implementovaných druhov API. Obsahujú ukážku autentickej webovej požiadavky klienta s odpoveďou servera.



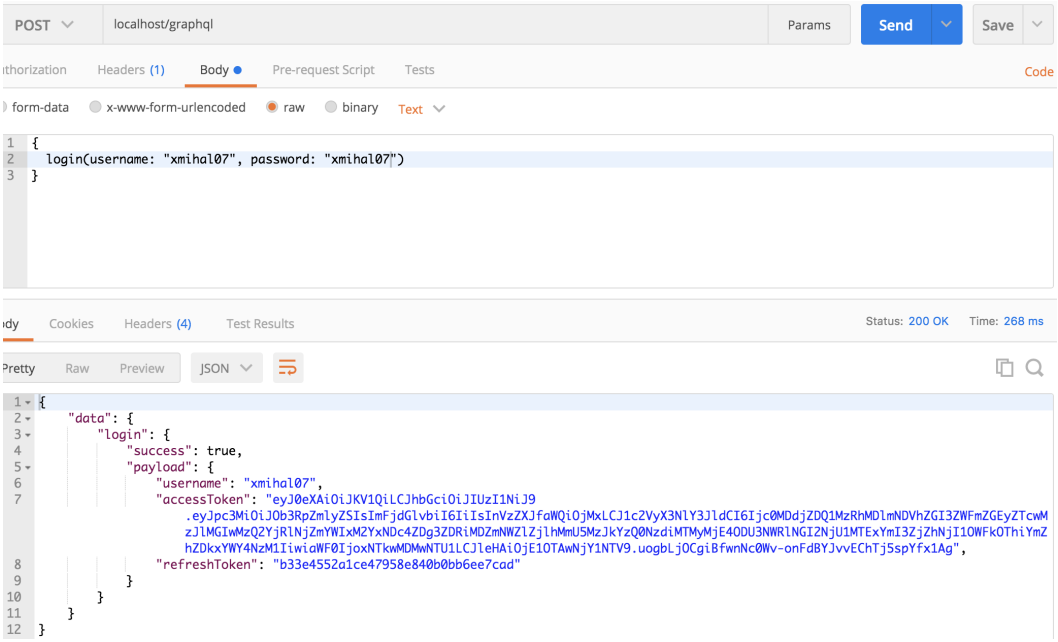
Obr. B.1: Registrácia cez REST



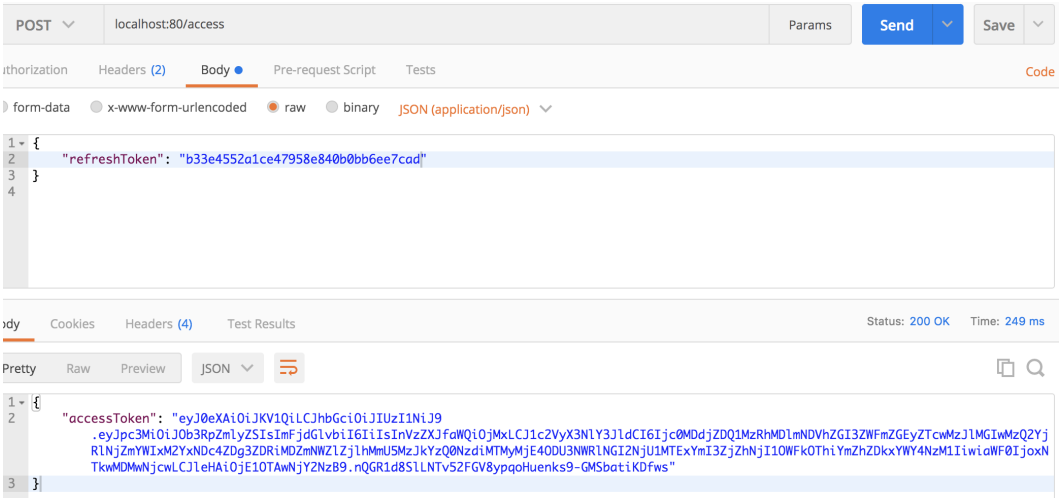
Obr. B.2: Registrácia už existujúceho používateľa cez GraphQL



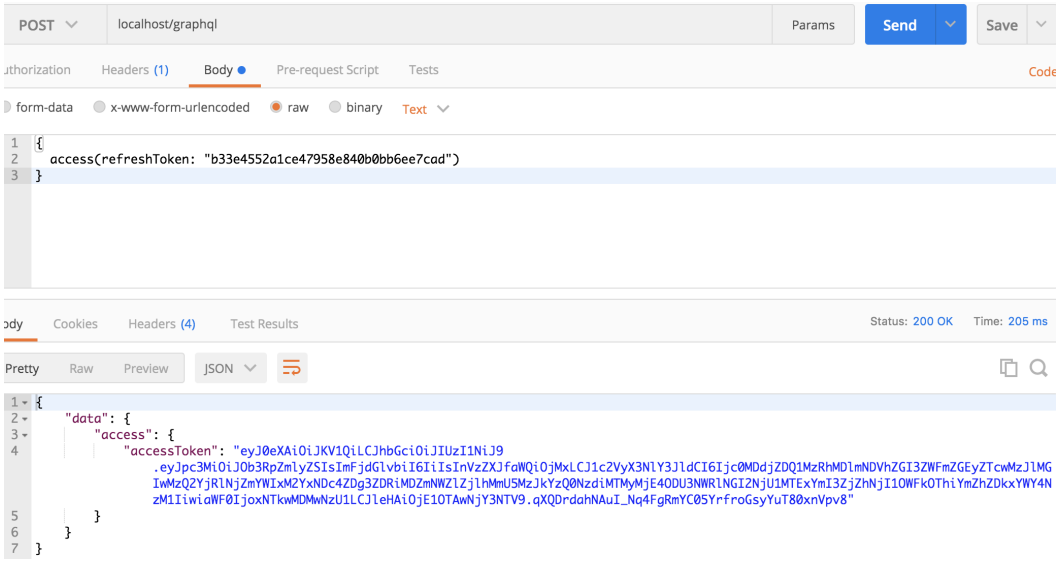
Obr. B.3: Prihlásenie neverifikovaného používateľa cez REST



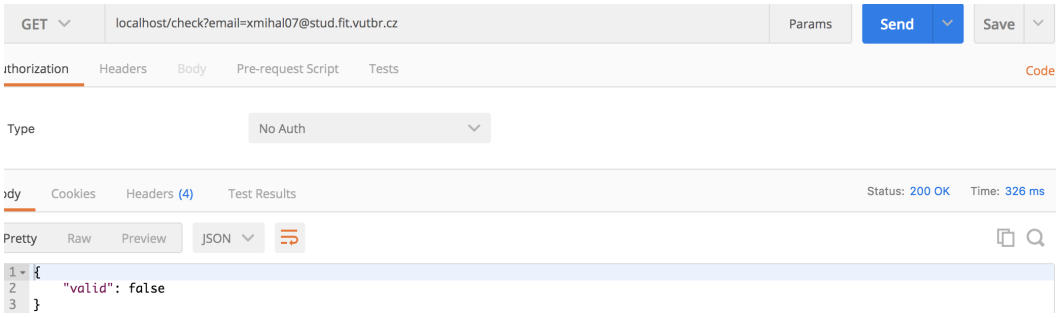
Obr. B.4: Prihlásenie používateľa cez GraphQL



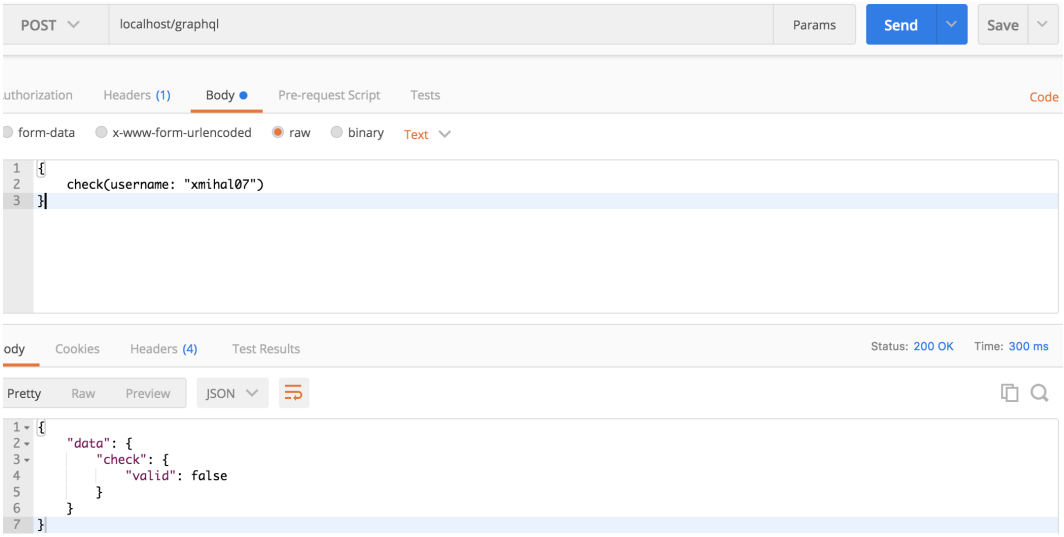
Obr. B.5: Obnova access tokenu cez REST



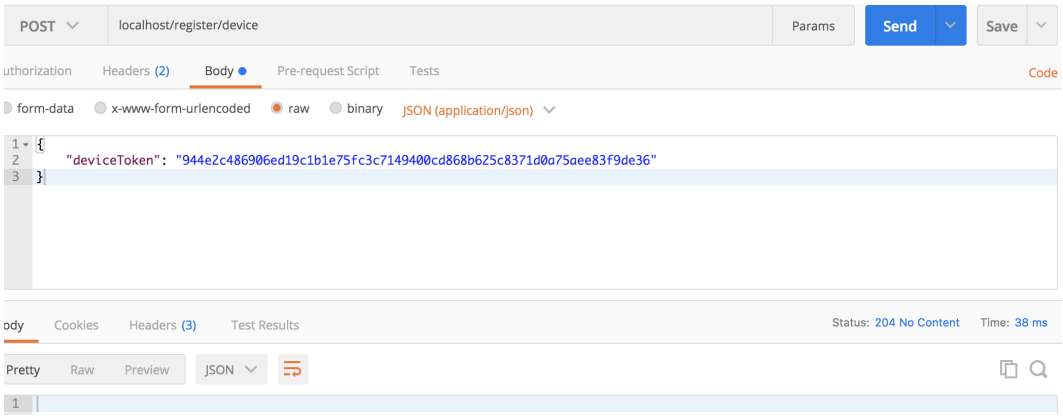
Obr. B.6: Obnova access tokenu cez GraphQL



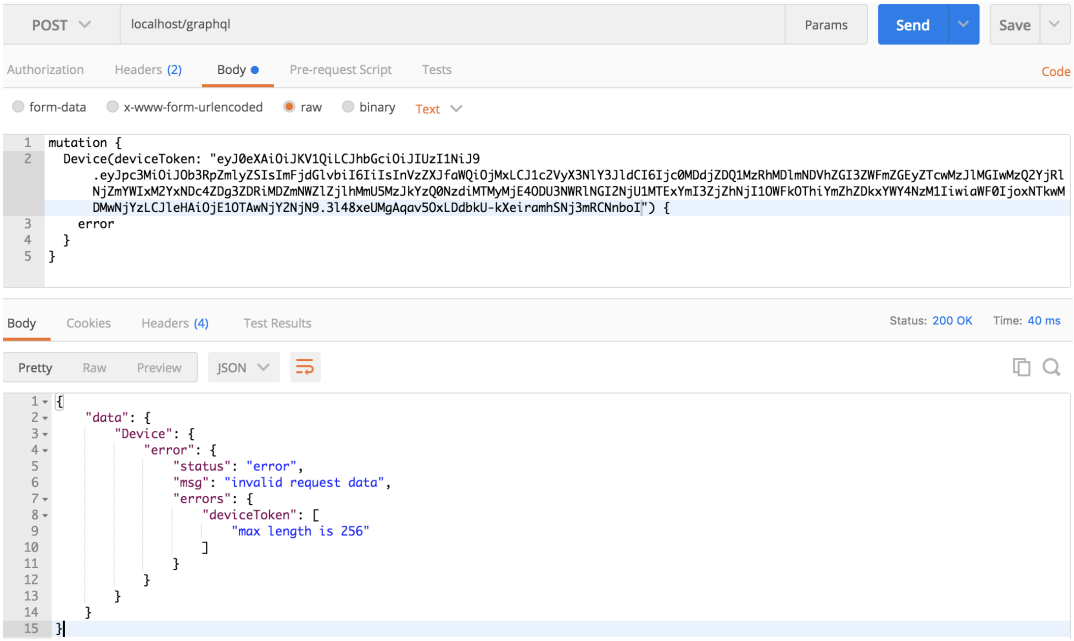
Obr. B.7: Overenie dostupnosti emailu cez REST



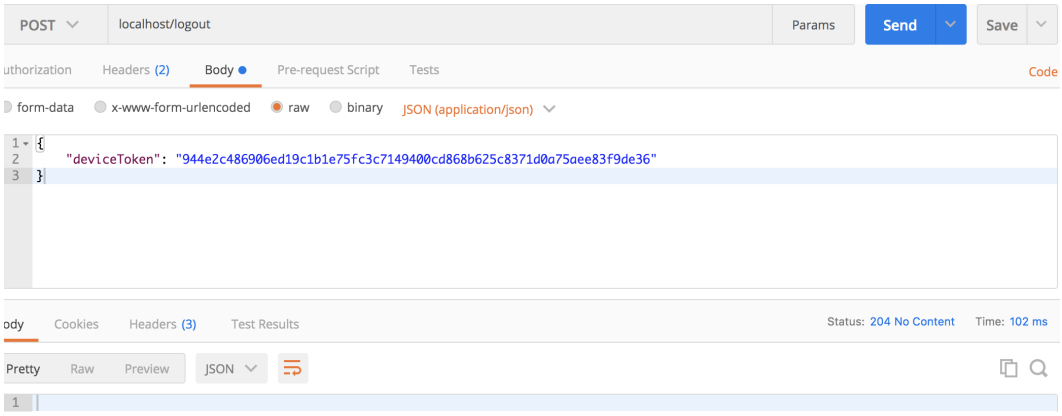
Obr. B.8: Overenie dostupnosti používateľského mena cez GraphQL



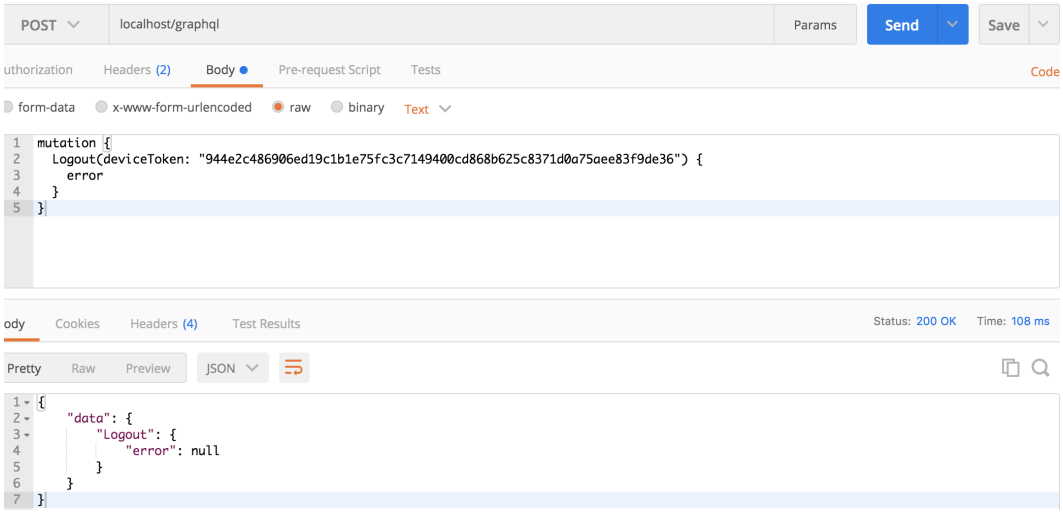
Obr. B.9: Registrácia iOS zariadenia cez REST



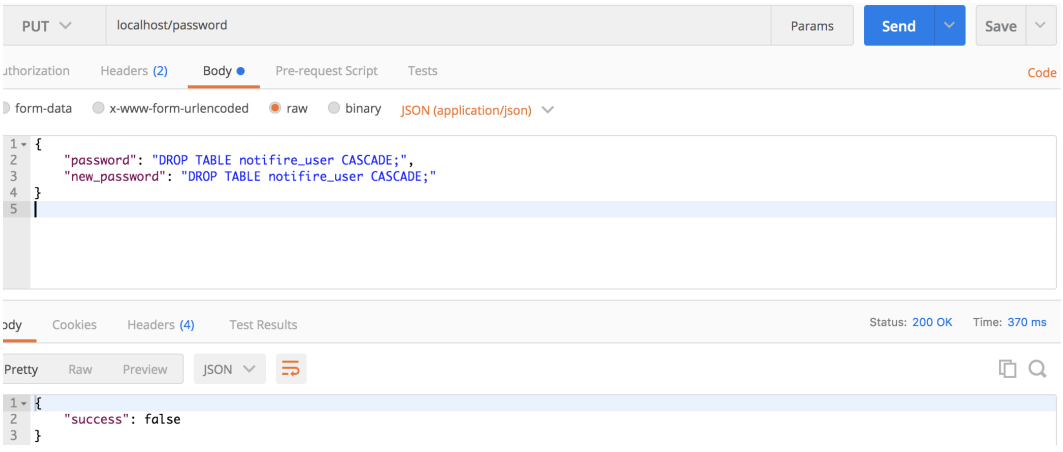
Obr. B.10: Registrácia iOS zariadenia s chybným device tokenom cez GraphQL



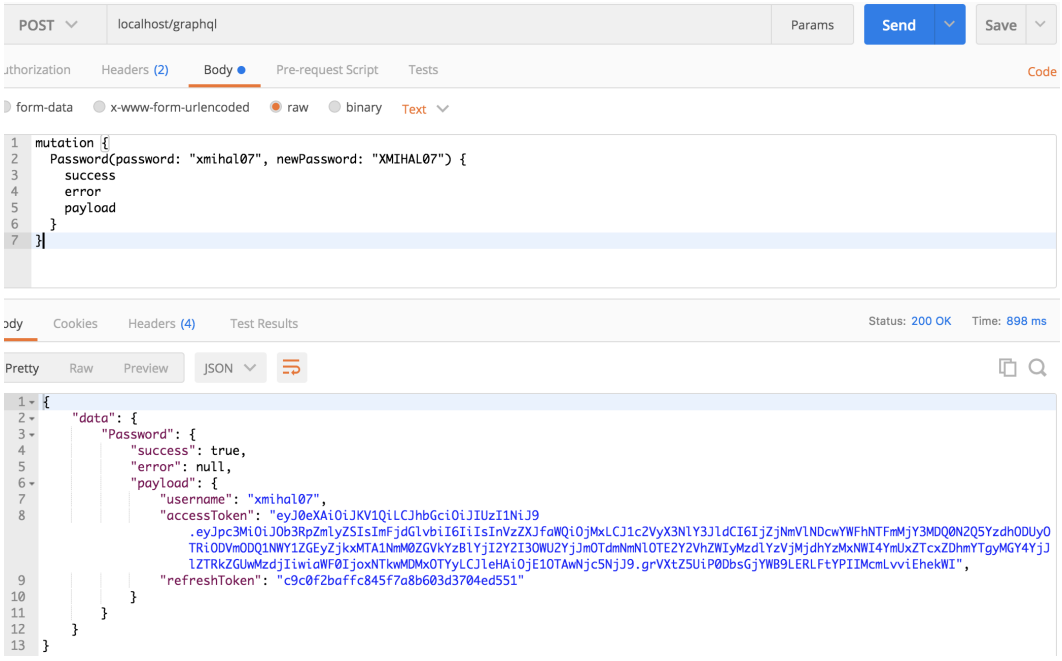
Obr. B.11: Odhlásenie cez REST



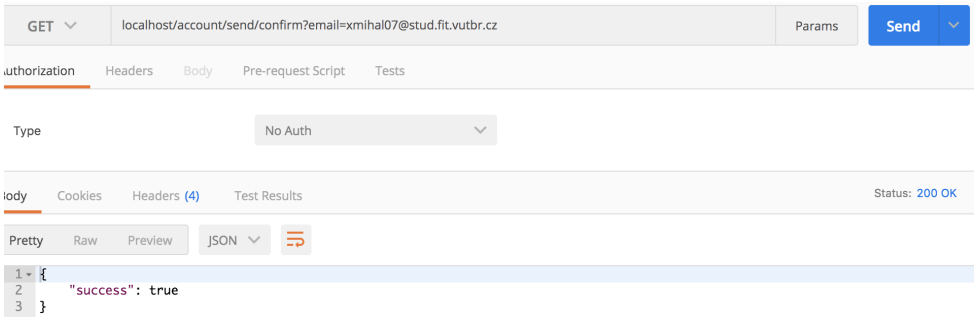
Obr. B.12: Odhlásenie cez GraphQL



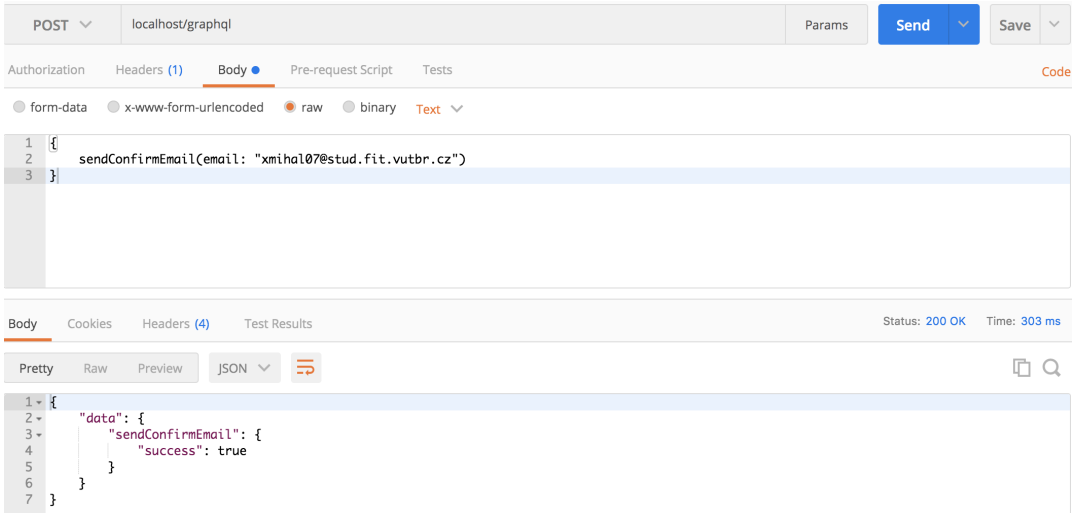
Obr. B.13: Neúspešná zmena hesla s SQL injekciou cez REST



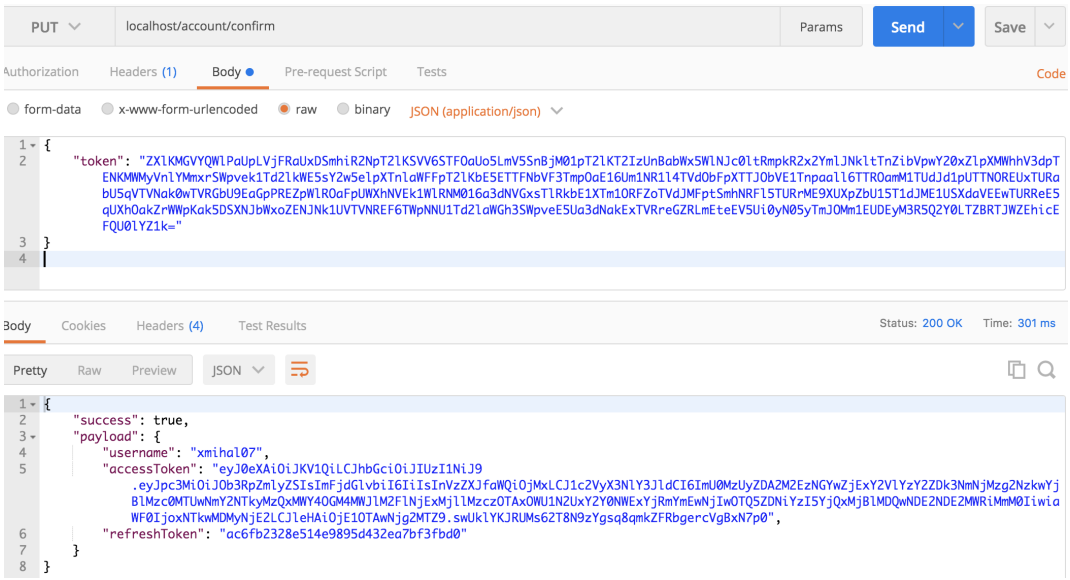
Obr. B.14: Zmena hesla cez GraphQL



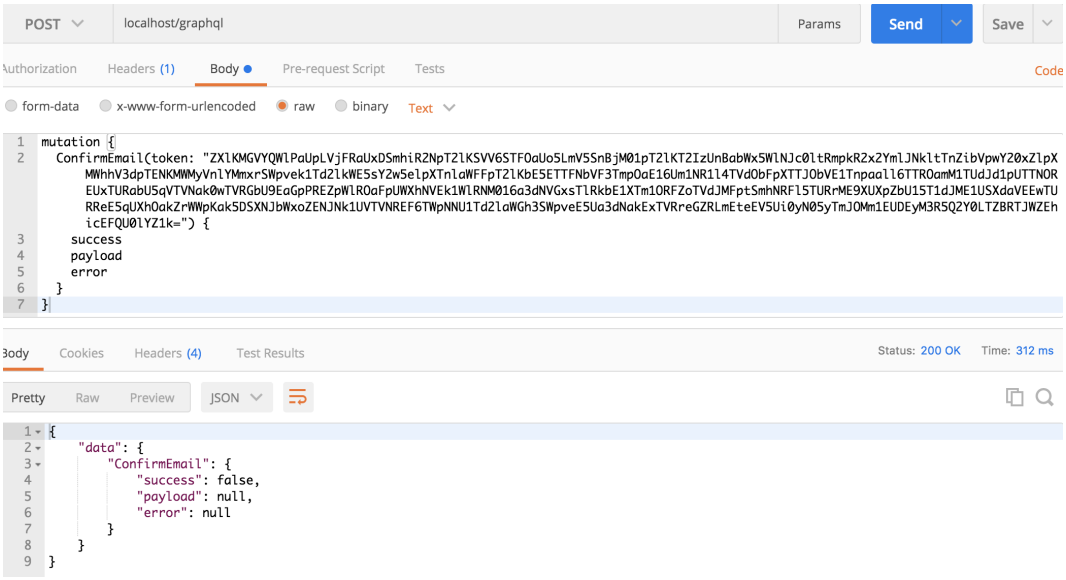
Obr. B.15: Požiadavka na odoslanie verifikačného emailu cez REST



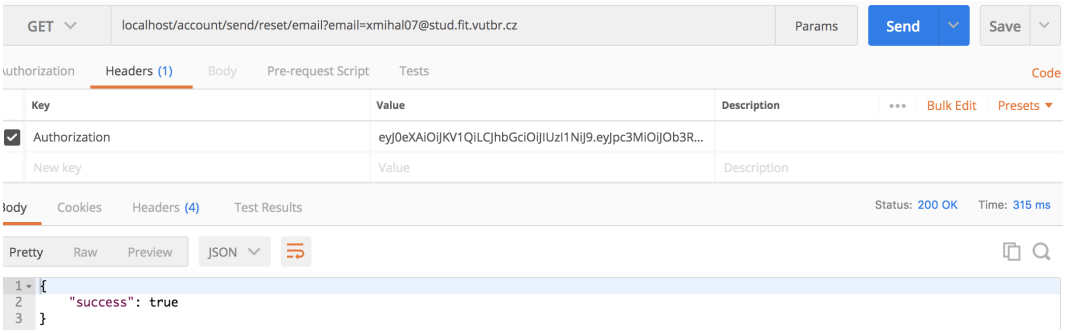
Obr. B.16: Požiadavka na odoslanie verifikačného emailu cez GraphQL



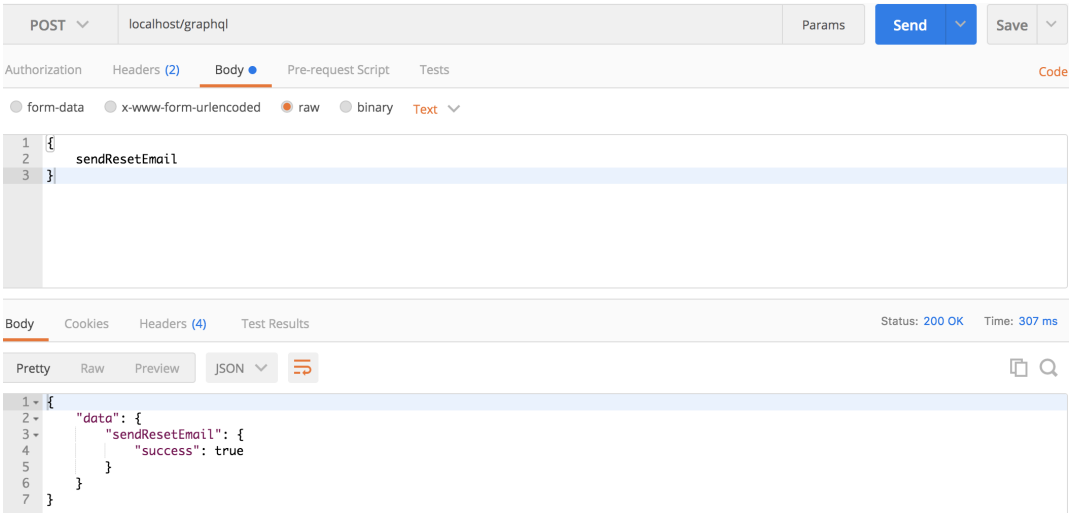
Obr. B.17: Verifikácia emailu cez REST



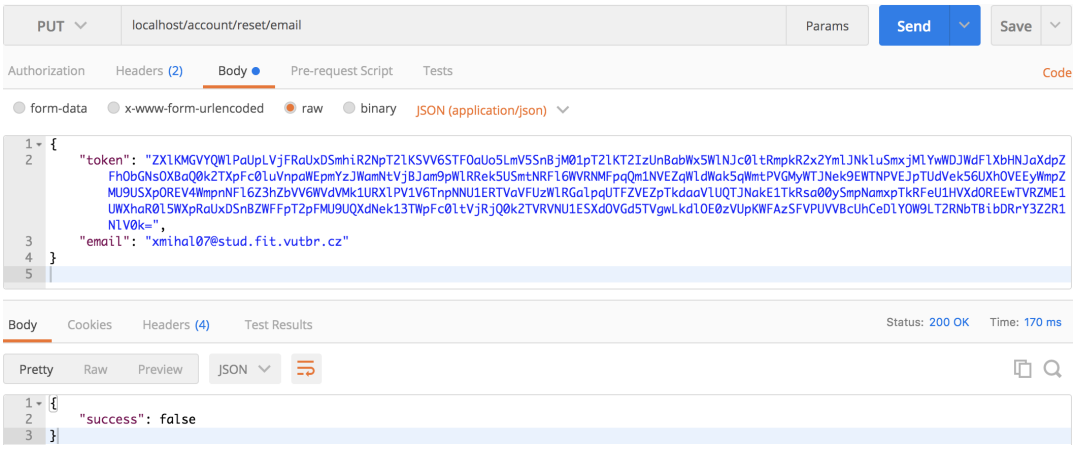
Obr. B.18: Verifikácia emailu cez GraphQL



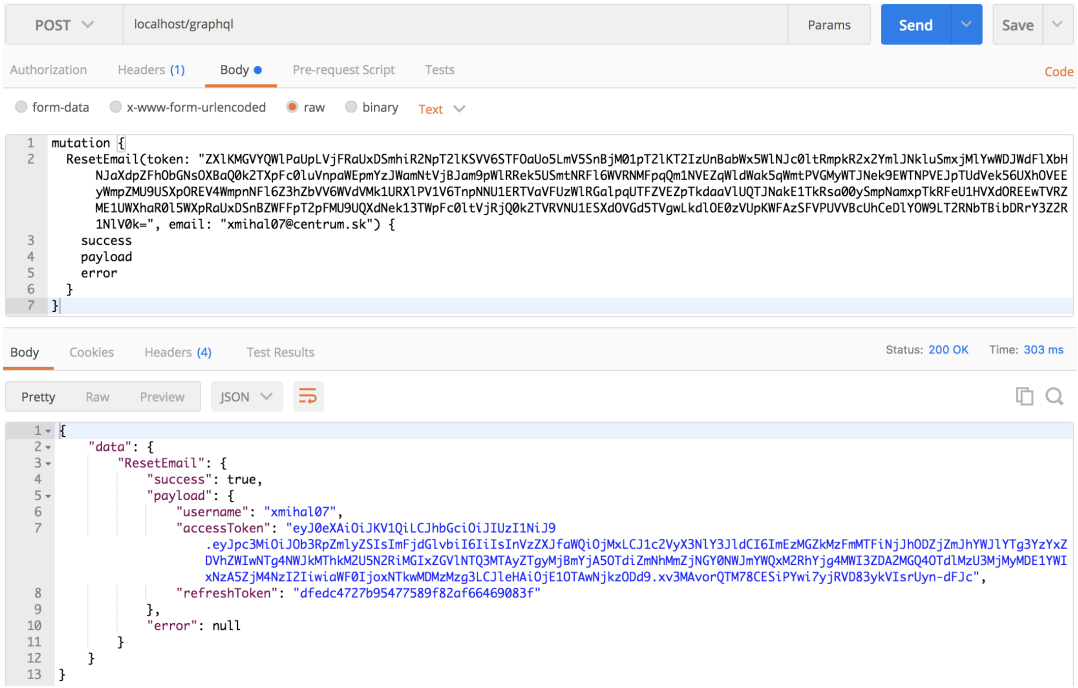
Obr. B.19: Požiadavka na odoslanie emailu na reset emailu cez REST



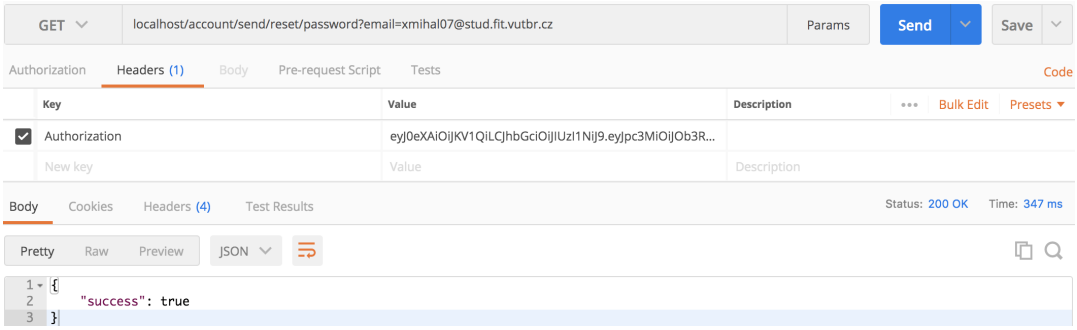
Obr. B.20: Požiadavka na odoslanie emailu reset emailu cez GraphQL



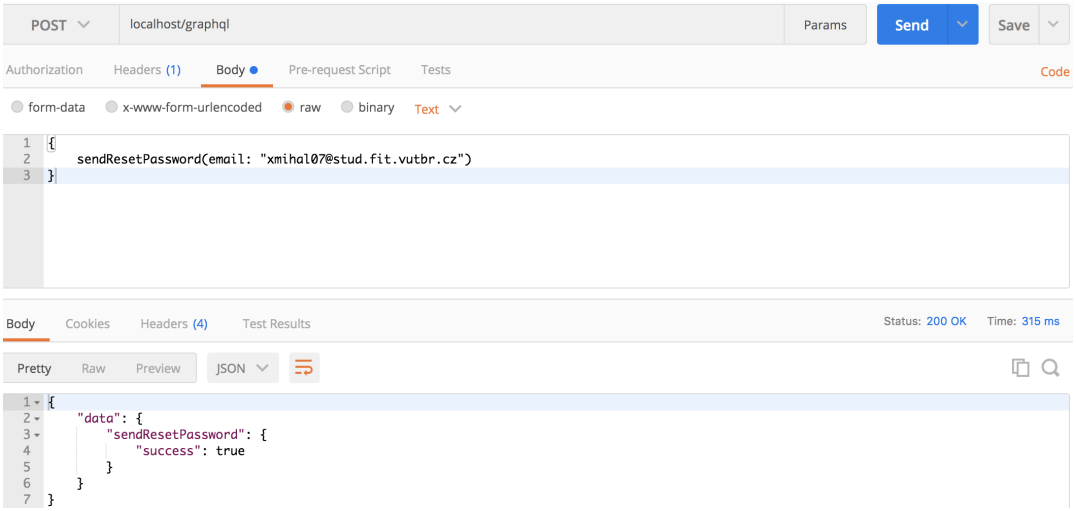
Obr. B.21: Reset emailu cez REST



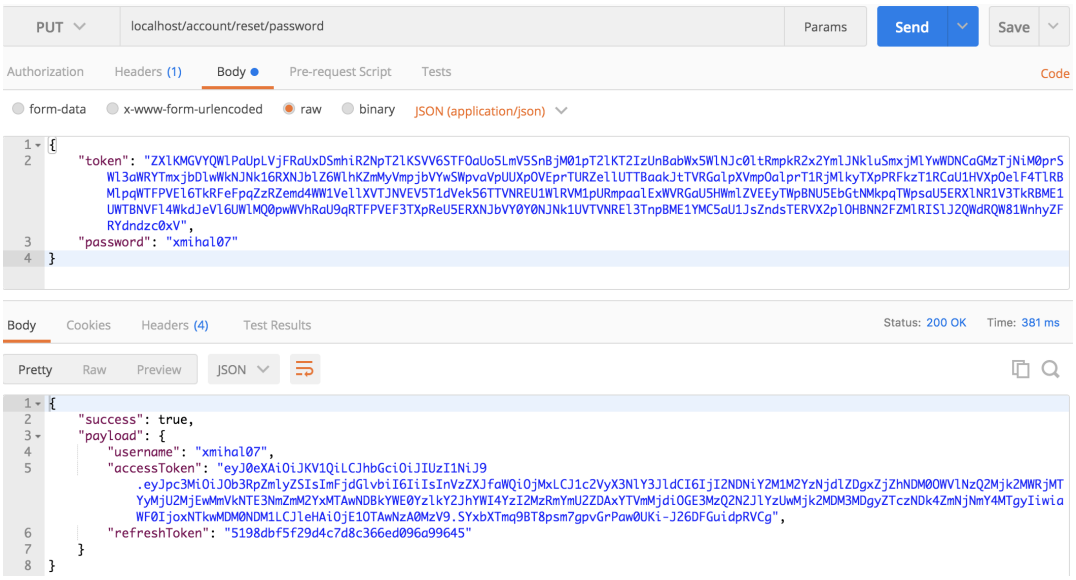
Obr. B.22: Reset emailu cez GraphQL



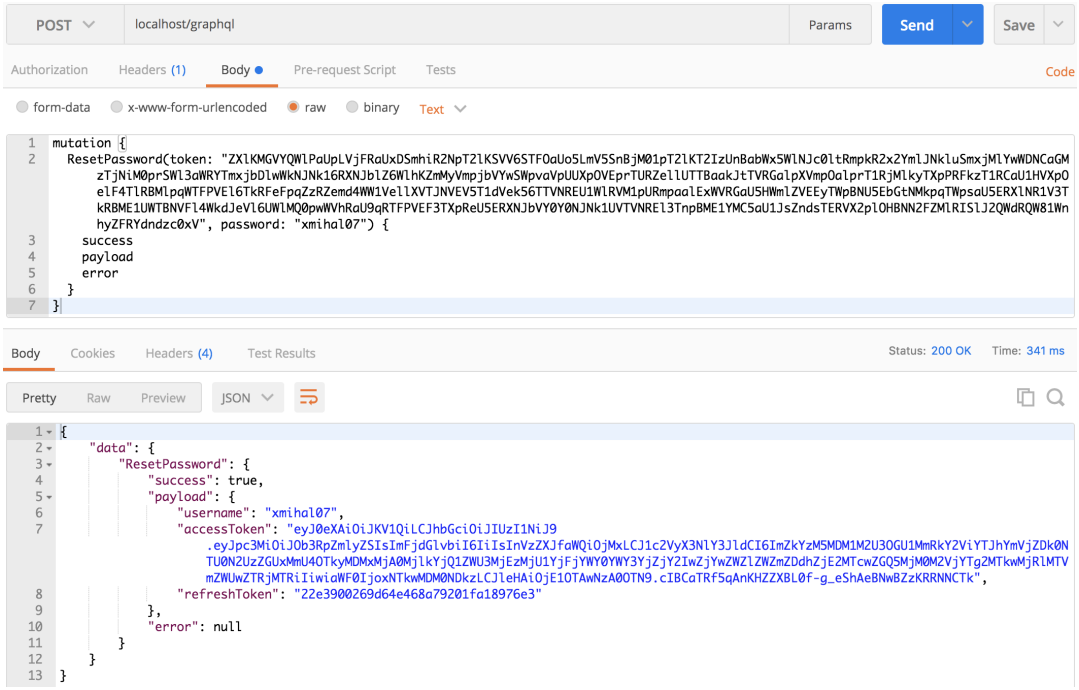
Obr. B.23: Požiadavka na odoslanie emailu na reset hesla cez REST



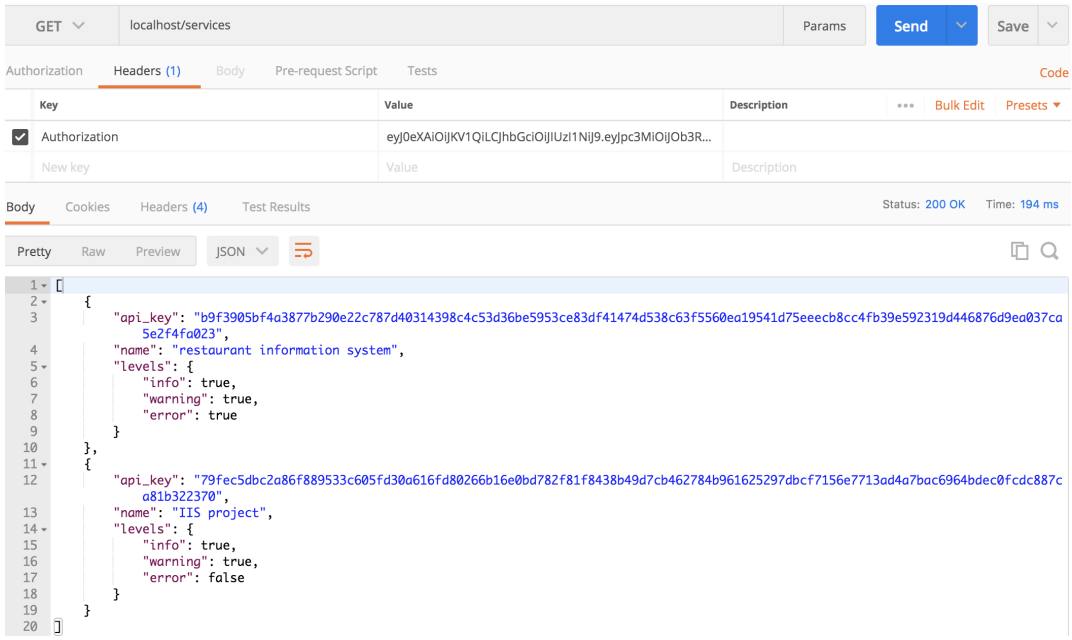
Obr. B.24: Požiadavka na odoslanie emailu na reset hesla cez GraphQL



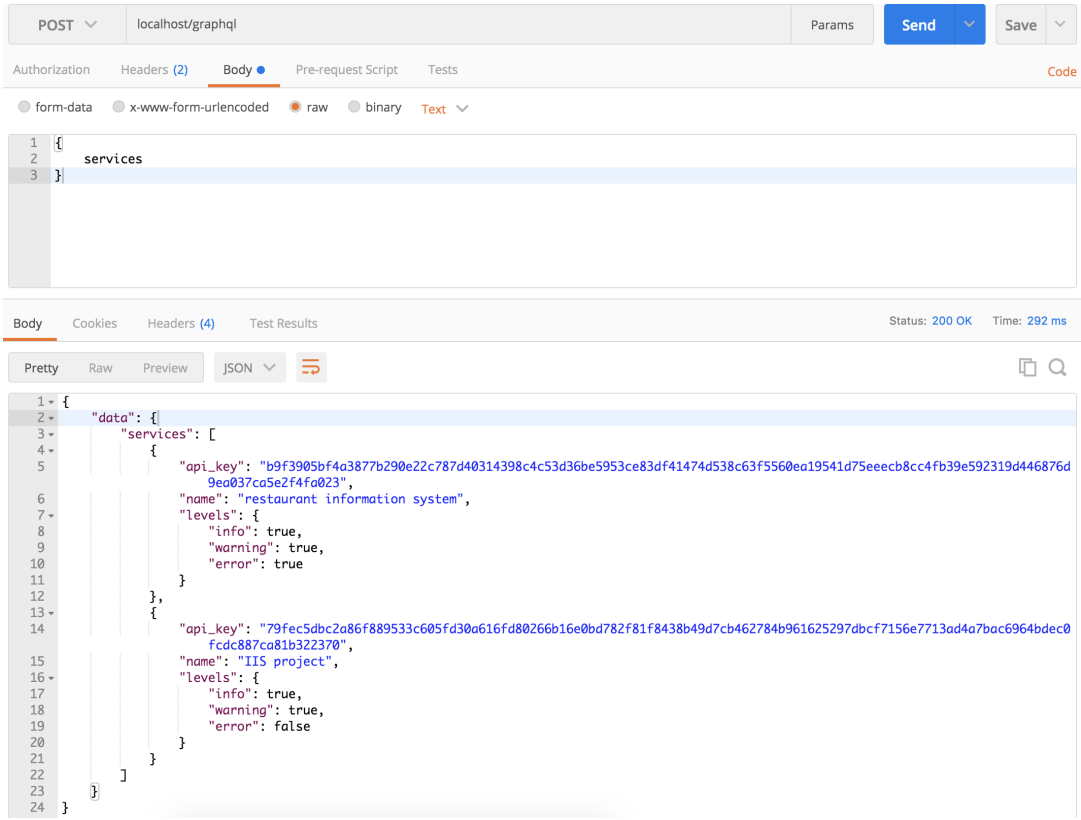
Obr. B.25: Reset hesla cez REST



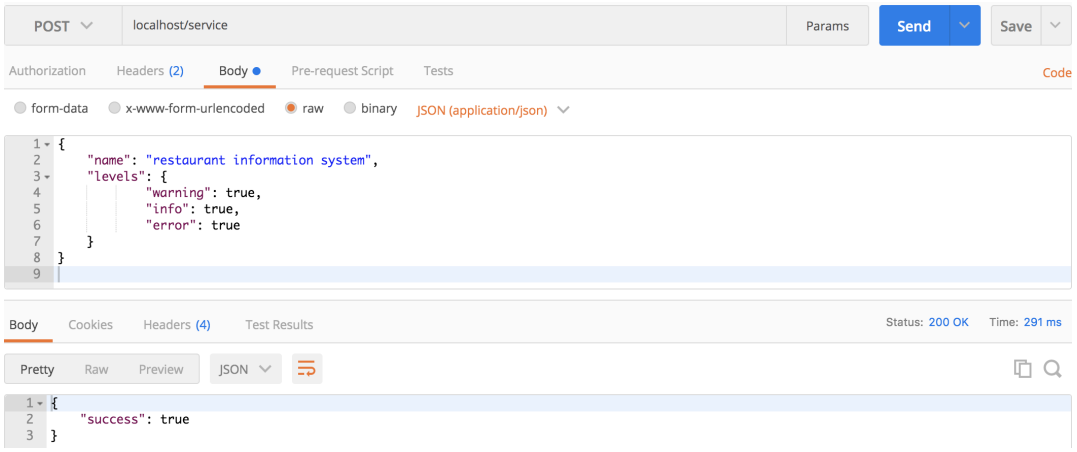
Obr. B.26: Reset hesla cez GraphQL



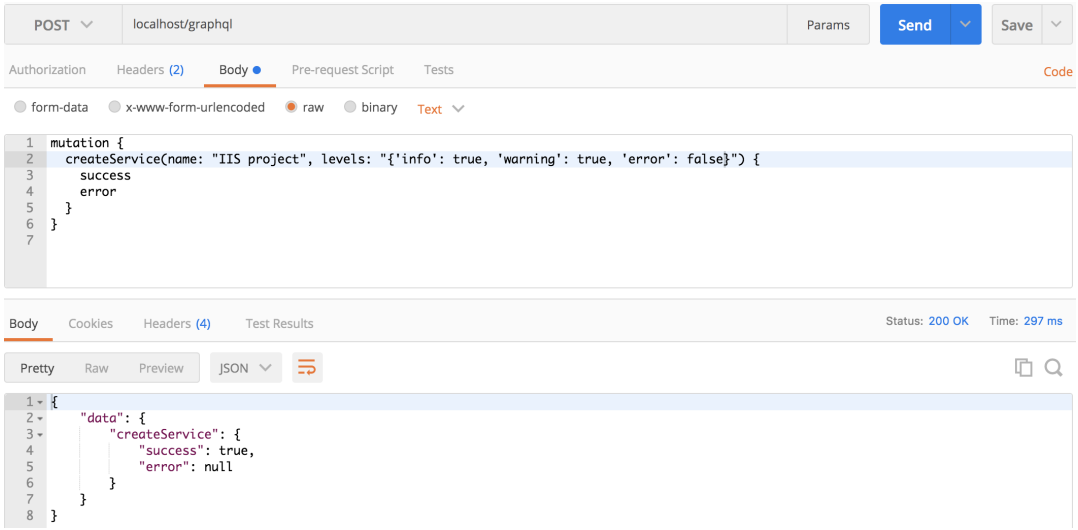
Obr. B.27: Výpis zoznamu services cez REST



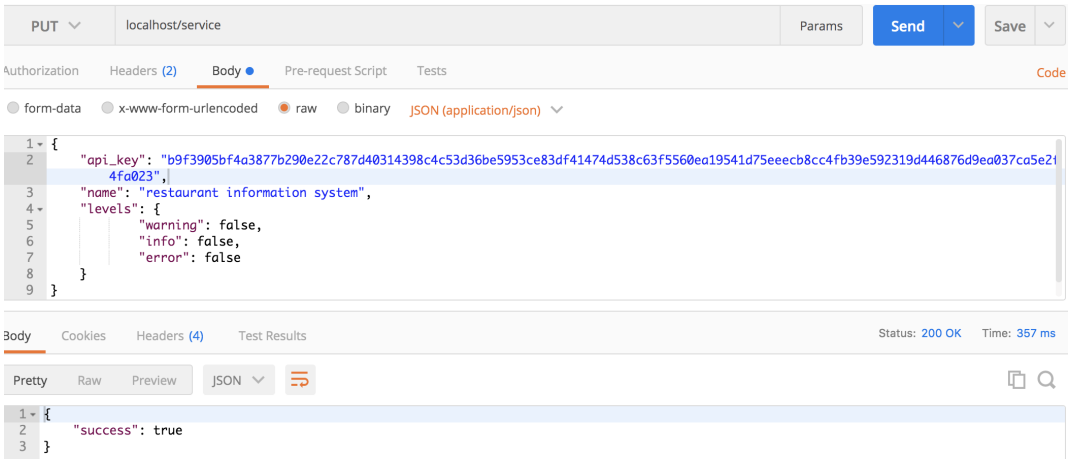
Obr. B.28: Výpis zoznamu services cez GraphQL



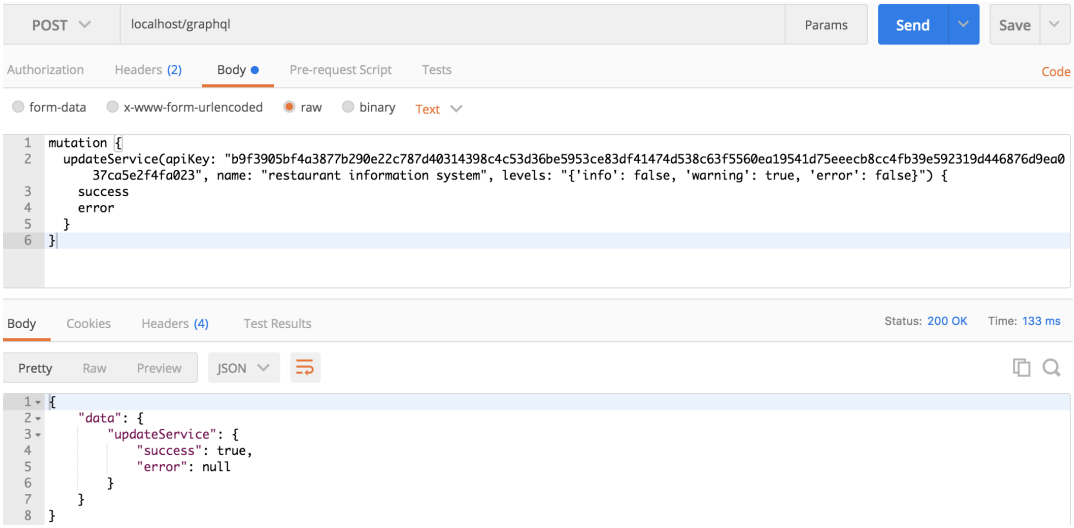
Obr. B.29: Vytvorenie service-u cez REST



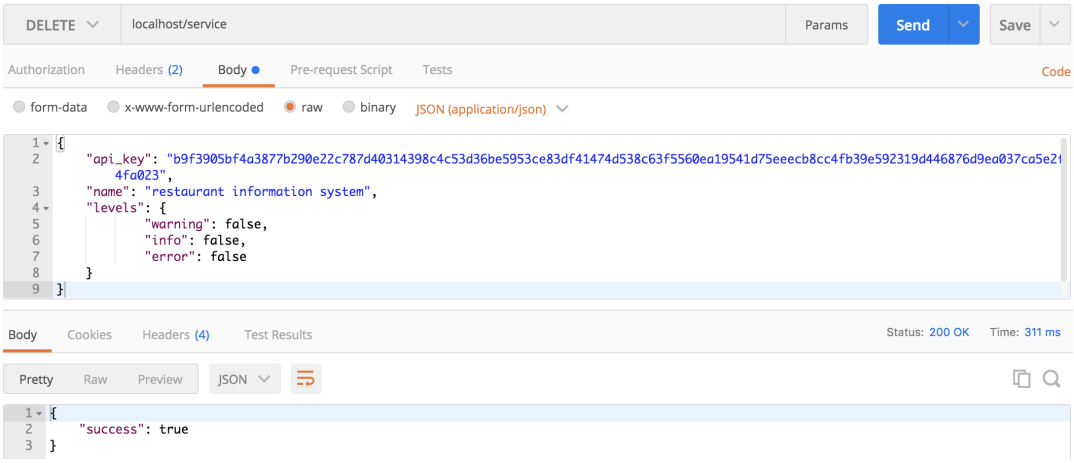
Obr. B.30: Vytvorenie service-u cez GraphQL



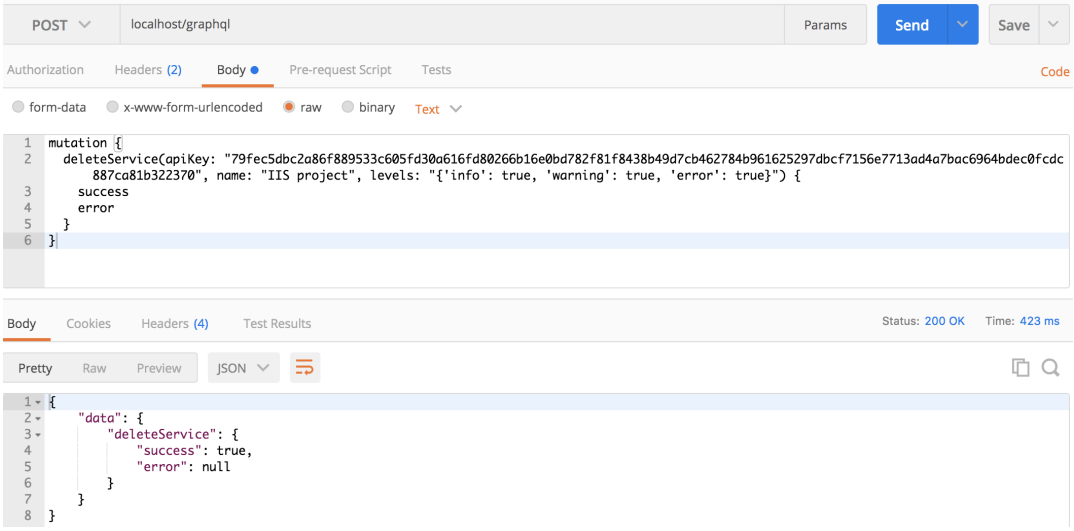
Obr. B.31: Zmena service-u cez REST



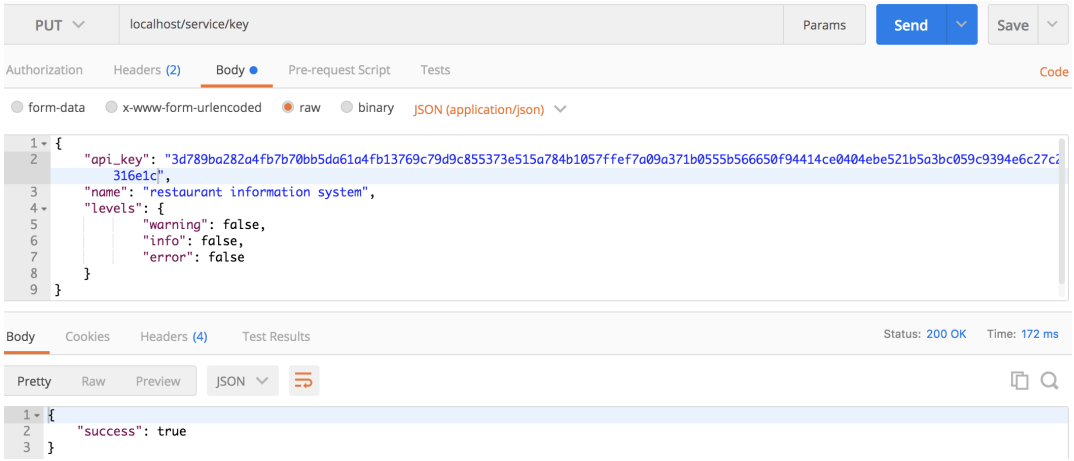
Obr. B.32: Zmena service-u cez GraphQL



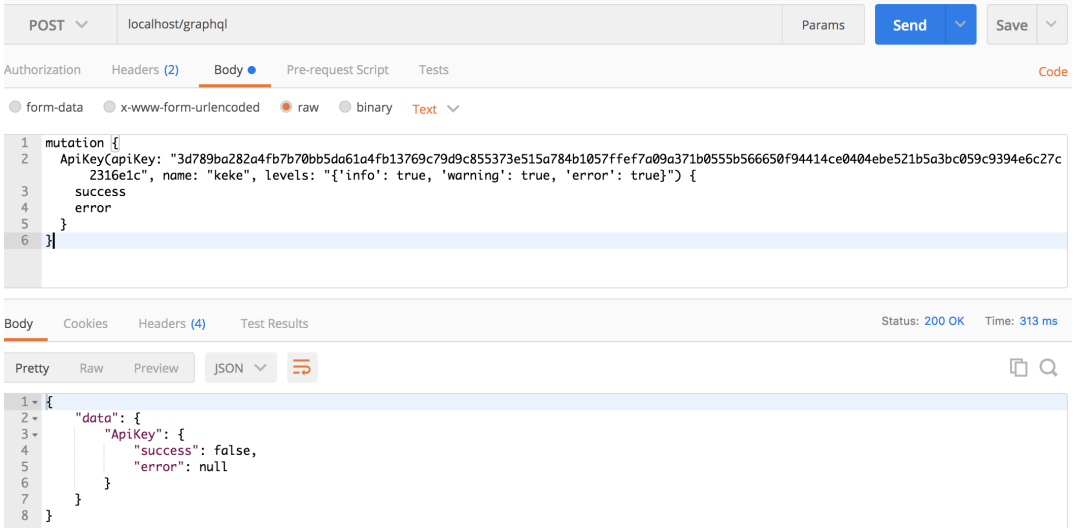
Obr. B.33: Vymazanie service-u cez REST



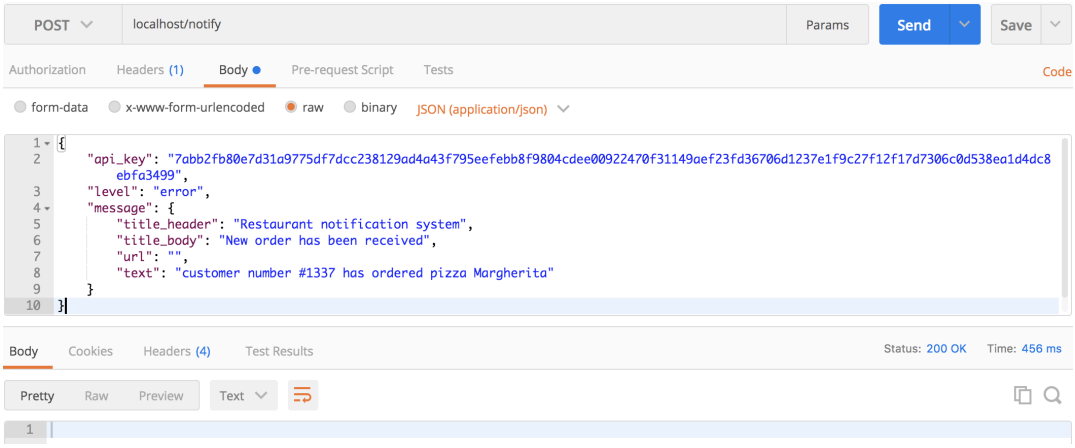
Obr. B.34: Vymazanie service-u cez GraphQL



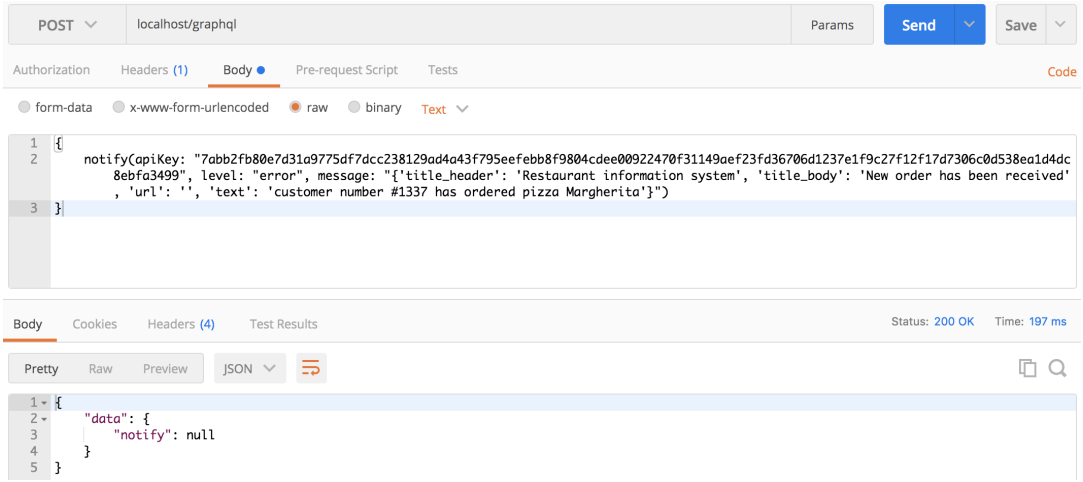
Obr. B.35: Zmena api kľúča service-u cez REST



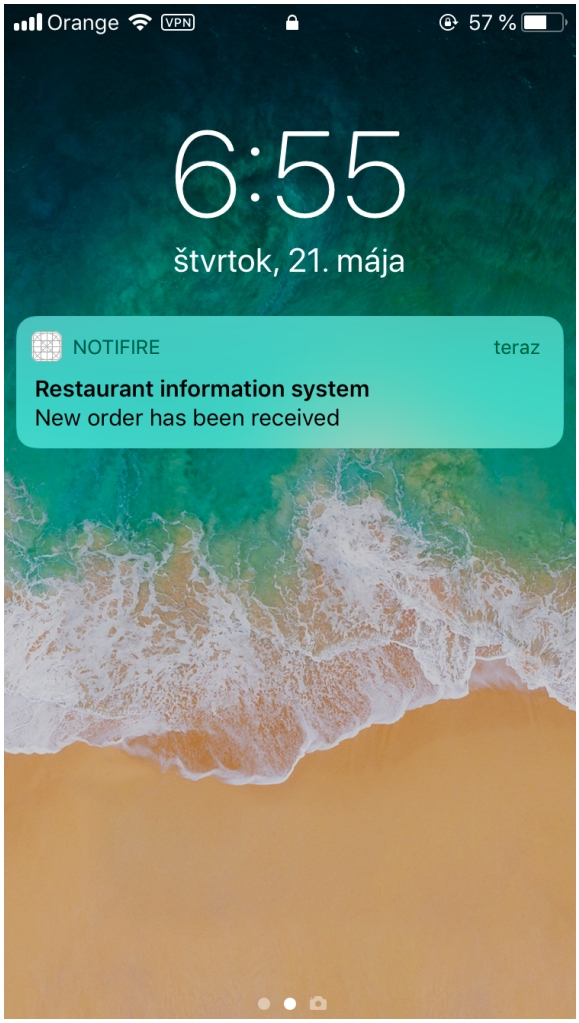
Obr. B.36: Zmena api kľúča service-u cez GraphQL



Obr. B.37: Notifikácia cez REST



Obr. B.38: Notifikácia cez GraphQL



Obr. B.39: Notifikácia prijatá na iOS zariadení