Autentizace a autorizace aplikačního rozhraní s využitím RFC 7519 a databáze Redis

Bc. Filip Kroča

Diplomová práce
2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav elektroniky a měření

Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Filip Kroča**
Osobní číslo:	**A18395**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Bezpečnostní technologie, systémy a management**
Forma studia:	**Kombinovaná**
Téma práce:	**Autentizace a autorizace aplikačního rozhraní s využitím RFC 7519 a databáze Redis**
Téma práce anglicky:	**Authentication and authorization of application interface using RFC 7519 and Redis database**

Zásady pro vypracování

1. Zpracujte rešerši literatury z oblasti technologií aplikačních rozhraní (API) a možností jejich zabezpečení pomocí autentizace a autorizace.
2. Proveďte analýzu konvenčního HTTP REST API.
3. Zvolte vhodné moderní prostředky a metody pro návrh řešení.
4. Proveďte implementaci tohoto návrhu v libovolném programovacím / skriptovacím jazyce.
5. Otestujte toto rozhraní.
6. Zhodnoťte přínosy implementovaného rozhraní.

Rozsah diplomové práce:
Rozsah příloh:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. LEK, Kamol a Naruemol RAJAPAKSE. Cryptography: protocols, design, and applications. New York: Nova Science Publishers, c2012, ix, 242 s. Cryptography, steganography and data security. ISBN 9781621007791.
2. TILBORG, Henk C. A. van a Sushil JAJODIA. Encyclopedia of cryptography and security. 2nd ed. New York: Springer, c2011, xl, 1416 s. Springer reference. DOI: 9781441959065. Dostupné také z: http://www.springerlink.com/content/978-1-4419-5905-8/contents/
3. PORCELLO, Eve a Alex BANKS, 2018. *Learning GraphQL*. 1. O'Reilly Media. ISBN 978-149-2030-713.
4. SIRIWARDENA, Prabath, 2014. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. 1. Apress. ISBN 9781430268178.
5. VARGHESE, Shiju, 2015. *Web Development with Go: Building Scalable Web Apps and RESTful Services*. 1. Apress. ISBN 9781484210529.

Vedoucí diplomové práce: **doc. Ing. Roman Šenkeřík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce:	**9. prosince 2019**
Termín odevzdání diplomové práce:	**29. května 2020**

L.S.

doc. Mgr. Milan Adámek, Ph.D.
děkan

Ing. Milan Navrátil, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne

Filip Kroča, v. r.
podpis diplomanta

ABSTRAKT

Tato diplomová práce se zabývá vytvoření a zabezpečení aplikačního rozhraní pomocí moderních programovacích přístupů a s využitím kryptografie. Práce má za cíl poukázat na možné přínosy těchto progresivních technologií do průmyslu komerční bezpečnosti. Z tohoto důvodu bylo jakožto praktický příklad vybráno aplikační rozhraní systému kontroly vstupu ACCESS. Práce je rozdělena na teoretickou a praktickou část.

Teoretická část obsahuje literární rešerši, která byla zpracována na základě české a zahraniční odborné literatury. Tato rešerše obsahuje popis technologií používaných pro šifrování přenosu, kontroly konzistence dat a deklarace identit. Dále jsou popsány technologie aplikačních rozhraní SOAP, REST a GraphQL.

V praktické části je pomocí zvolených technologií implementováno aplikační rozhraní, které je zabezpečeno pomocí TLS šifrování a JWT tokenů. Toto rozhraní je otestováno pomocí testovacího nástroje Insomnia a pomocí sady automatických testů Mocha.

Ze závěru práce plyne, že pronikání těchto moderních prostředků do průmyslu komerční bezpečnosti může přinést zvýšení bezpečnosti výsledných aplikací a také zrychlení vývojového cyklu těchto aplikací.

Klíčová slova:

Aplikační rozhraní, GraphQL, Apollo, JWT, autentizace, autorizace, bearer token, OAuth

ABSTRACT

This thesis is devoted to creating and securing an application interface with a modern programming approach and cryptography. The goal of the thesis is to show which benefits can offer these technologies into the commercial security industry. Due to these reasons was as use-case selected an application interface for the access system. The thesis is divided into the theoretical and the practical part.

The theoretical part contains literary research that was created over the Czech and world professional literature. The literary research contains a description of technologies used for cryptography, consistency check, and identity declaration. Also, application interface technologies SOAP, REST, and GraphQL are described in this part.

In the practical part is the application interface implemented using the TLS encryption and JWT tokens. This interface is tested with the Insomnia tool and Mocha - automated testing framework.

The conclusion follows that the implementation of these technologies into the commercial security industry can bring better security of applications and faster development cycle.

Keywords:

Application interface, GraphQL, Apollo, JWT, authorization, authentification, bearer token, OAuth

Tímto bych rád poděkoval vedoucímu mé diplomové práce doc. Ing. Roman Šenkeřík, Ph.D. za všestrannou pomoc, množství cenných a inspirativních rad, podnětů, doporučení, připomínek a zároveň za velkou trpělivost s obdivuhodnou ochotou při konzultacích poskytnutých ke zpracování této práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD 11

I. TEORETICKÁ ČÁST 13

1 APLIKAČNÍ PROGRAMOVATELNÉ ROZHRANÍ 14

1.1 AUTENTIZACE 15

1.1.1 Proces autentizace *15*

1.2 AUTORIZACE 15

1.3 ZABEZPEČENÍ PŘENOSU 17

1.3.1 RSA *17*

1.3.2 SSL *17*

1.3.3 TLS *19*

1.3.4 Shrnutí verzí SSL a TLS: *20*

1.4 SEZNAM POUŽITÝCH HASHOVACÍCH FUNKCÍ 20

1.4.1 SHA1 ..... *20*

1.4.2 HMAC-SHA1 ..... *20*

1.5 METODY POUŽÍVANÉ PŘI DEKLARACI IDENTITY 21

1.5.1 HTTP Autentizace: Základní a přehledové ověřování přístupu *22*

1.5.2 Shrnutí HTTP Autentizace *22*

1.5.3 Vzájemné ověřování *23*

1.5.4 OAuth 1.0 ..... *23*

1.5.5 OAuth 2.0 ..... *28*

1.5.6 JWT (JSON Web Tokens) *33*

2 TECHNOLOGIE APLIKAČNÍCH ROZHRANÍ 37

2.1 SOAP 37

2.2 HTTP REST API 37

2.2.1 První úroveň – URI *38*

2.2.2 Druhá úroveň – metody CRUD *38*

2.3 GRAPHQL – NÁSTUPCE REST 41

3 VOLBA VHODNÝCH PROSTŘEDKŮ 44

3.1 NODE.JS 44

3.1.1 ECMAScript *45*

3.2 APOLLO GRAPHQL 46

3.3 JWT 48

3.4 DATABÁZE REDIS 48

II. PRAKTICKÁ ČÁST 49

4 NÁVRH ŘEŠENÍ 50

4.1 STRUKTURA ACCESS SYSTÉMU ... 50
4.1.1 Úrovně oprávnění ... 51
4.1.2 Sada oblastí ... 51
4.1.3 Sada testovacích uživatelů ... 52

5 IMPLEMENTACE API ... 53

5.1 INICIALIZACE NOVÉHO NPM PROJEKTU ... 53
5.2 FUNKCE PRO OBSLUHU JWT TOKENŮ ... 53
5.2.1 Import klíčů a samotné knihovny ... 57
5.2.2 Vytváření nových tokenů ... 58
5.2.3 Kontrola tokenu pomocí online nástroje jwt.io ... 61
5.2.4 Funkce pro kontrolu validity tokenu v rámci řetězce autentizace API ... 62
5.3 IDENTIFIKACE UŽIVATELE A NAČTENÍ ÚROVNĚ JEHO PRÁV Z DATABÁZE REDIS ... 63
5.3.1 Získání databáze Redis ... 63
5.3.2 JS knihovna Node Redis ... 65
5.3.3 Funkce umožňující autorizaci požadavků ... 65
5.4 IMPLEMENTACE GRAPHQL SERVERU ... 67
5.4.1 Schéma GraphQL ... 67
5.4.2 Resolverové funkce Query ... 71
5.4.3 Resolverové funkce Mutace ... 85
5.4.4 Sestavení https a Apollo serveru ... 95

6 TESTOVÁNÍ ... 99

6.1 INSOMNIA ... 99
6.1.1 Nastavení autentizace ... 100
6.1.2 Nastavení TLS ... 100
6.1.3 Otestování Query bez platného JWT tokenu ... 100
6.1.4 Otestování Query getUsers ... 101
6.1.5 Otestování Query getToken ... 102
6.1.6 Otestování Query getAccess ... 102
6.1.7 Otestování Mutace addUser ... 103
6.1.8 Otestování Mutace blockUser ... 104
6.1.9 Otestování Mutace addArea ... 105
6.2 MOCHA ... 106
6.2.1 Testy pomocí Mocha ... 106

7 ZHODNOCENÍ PŘÍNOSŮ ... 110

ZÁVĚR ... 111

SEZNAM POUŽITÉ LITERATURY ... 112

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 114

SEZNAM OBRÁZKŮ 115

SEZNAM TABULEK 116

SEZNAM VÝPISŮ 117

ÚVOD

Aplikační rozhraní jsou zcela zásadní technologie, bez niž by vznik současných digitálních služeb nebyl možný. Současný svět se stává stále více propojeným a digitalizovaným, přičemž množství aplikací roste geometrickou řadou a mnohé z těchto aplikací obsahují prvky kritické pro bezpečnost a chod společnosti. Sběr dat je téměř všudypřítomný a probíhá v masovém měřítku. Převážná část této komunikace probíhá pomocí aplikačních rozhraní, u nichž je potřeba velmi dobrého zabezpečení. Tímto zabezpečením je především šifrování přenosu, autentizace uživatelů a autorizace požadavků. Potřeba správné volby a správná implementace těchto tři prvků je klíčová při návrhu správného řešení komunikace.

Tato práce má za cíl vytvořit a zabezpečit univerzální aplikační rozhraní systému kontroly vstupu ACCESS, který bude sloužit k řízení přístupu do budov a jejich prostor. Na toto vytvořené aplikační rozhraní bude možné napojit libovolný ACCESS systém.

V teoretické části budou shrnuty existující technologie a provedena jejich analýza. Jedná se o technologie SOAP, HTTP REST a GraphQL. Na základě této analýzy budou vybrány technologie vhodné k implementaci aplikačního rozhraní pro ACCESS systém.

V praktické části bude implementováno aplikační rozhraní se zabezpečením přenosu pomocí TLS šifrování v kombinaci s HTTPS protokolem.

Autentizace uživatelů – deklarace identity bude provedena pomocí JWT tokenů. Jedná se o zakódované textové řetězce, které jsou zabezpečeny pomocí asymetrické kryptografie a tím jsou ochráněny proti podvržení. Tyto tokeny budou obsahovat užitečná data ve formě uživatelského ID, časového razítka vydání tokenu a časového razítka expirace tokenu. Tokeny budou vydávány na uživatele, a to s platností 365 dní. Po uplynutí této doby se token automaticky stane neplatným a je nutné ho znovu vygenerovat. Tokeny bude uživatelům vydávat administrátor systému pomocí funkce určené k tomuto účelu. Tyto tokeny budou podepsány privátním šifrovacím klíčem, který bude vygenerován v praktické části.

Autorizace požadavků bude implementována pomocí víceúrovňového systému oprávnění, tyto informace budou uloženy v databázi Redis a při každém požadavku na rozhraní budou z této databáze načteny a bude rozhodnuto, zda má daný uživatel dostatečné oprávnění k požadované akci. Tato databáze bude také obsahovat údaje o všech oblastech ACCESS systému.

V části testování bude funkční aplikační rozhraní otestováno pomocí testovacího nástroje Insomnia. Tento nástroj je velmi rozšířený a oblíbený pro své široké testovací možnosti a širokou škálu možných nastavení testovacích požadavků. Umožňuje například testovat JWT tokeny a HTTPS. Aplikační rozhraní bude také vybaveno sadou automatických testů Mocha pro možnou integraci do CI/CD prostředí.

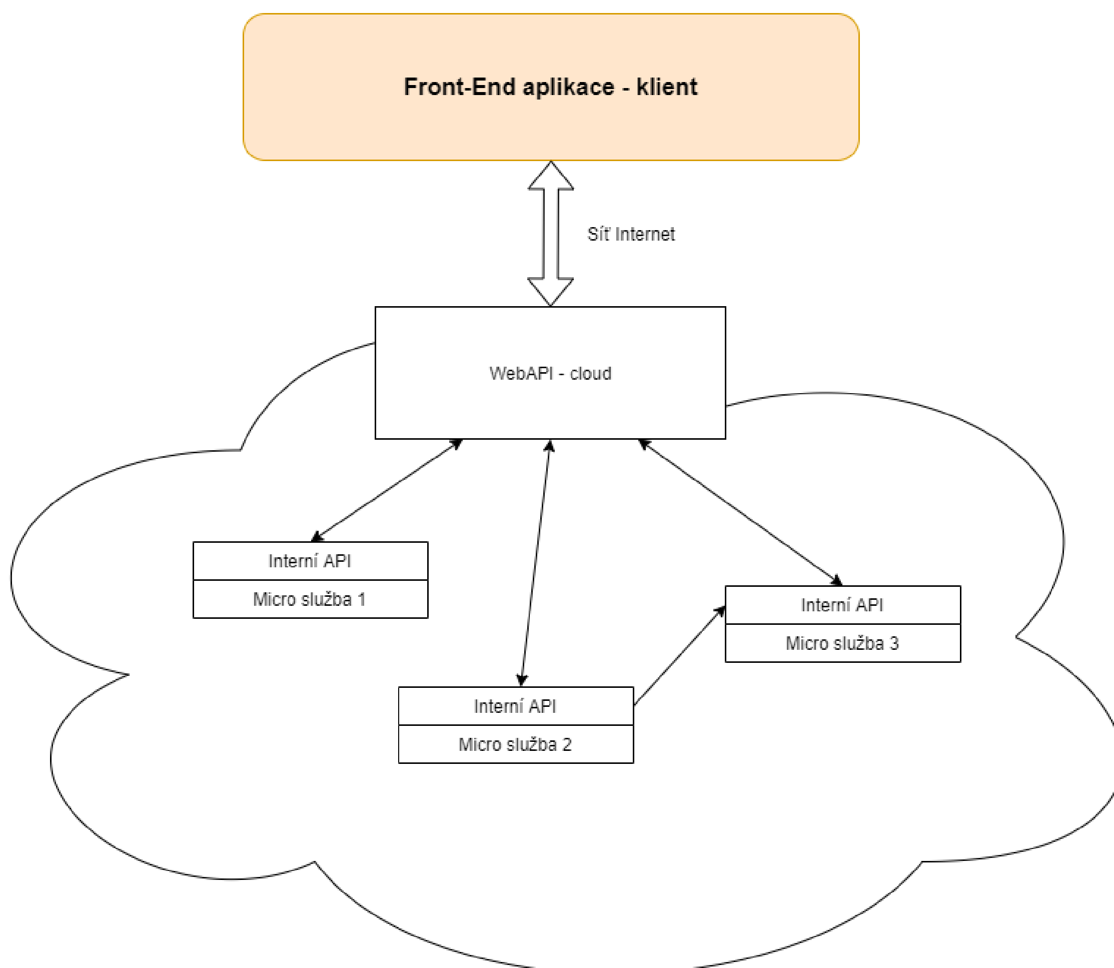
Závěr práce bude obsahovat vyhodnocení vytvořeného aplikačního rozhraní a použitých metod. Budou také popsány možné přínosy použitých technologií do průmyslu komerční bezpečnost.

I. TEORETICKÁ ČÁST

1 APLIKAČNÍ PROGRAMOVATELNÉ ROZHRANÍ

Application programming interface je obecné rozhraní, nebo komunikační protokol, zajišťující komunikaci mezi rozdílnými částmi programů a mezi rozdílnými systémy. Tato rozhraní jsou široce implementována na mnoha úrovních informačních systémů, od nízko úrovňových služeb (například POSIX) až po internetové služby (WebAPI viz. *Obr. 1*).

API poskytuje abstrakci problému a poskytuje specifikaci toho, jakým způsobem musí klient interagovat se softwarovými komponentami, které poskytují řešení tohoto problému. Tyto komponenty jsou většinou distribuovány jako softwarové knihovny, díky čemuž je možné je využívat v mnoho různých aplikacích. Ze své podstaty API definuje znovupoužitelné stavební bloky poskytující jednotlivé funkcionality, které poté mohou být zahrnuty do softwaru pro konečné uživatele [1].



Obr. 1. Funkční schéma API.

1.1 Autentizace

Proces autentizace je často referován jako „Autentizace, determinování a ověřování uživatelské identity“ [2]. V případě privátního API musí být tento proces opakován u každého požadavku z důvodu potřeby kontroly uživatelem proklamované identity. Z hlediska výběru metod jsou rozhodující především dva parametry, a to rychlost procesu kontroly a odolnost vůči podvržení. Také je nutné zabezpečit uživatelskou přívětivost tak, aby proces autentizace po prvotním ověření probíhal nejlépe zcela na pozadí a pro koncového uživatele byl tedy prakticky neviditelný.

1.1.1 Proces autentizace

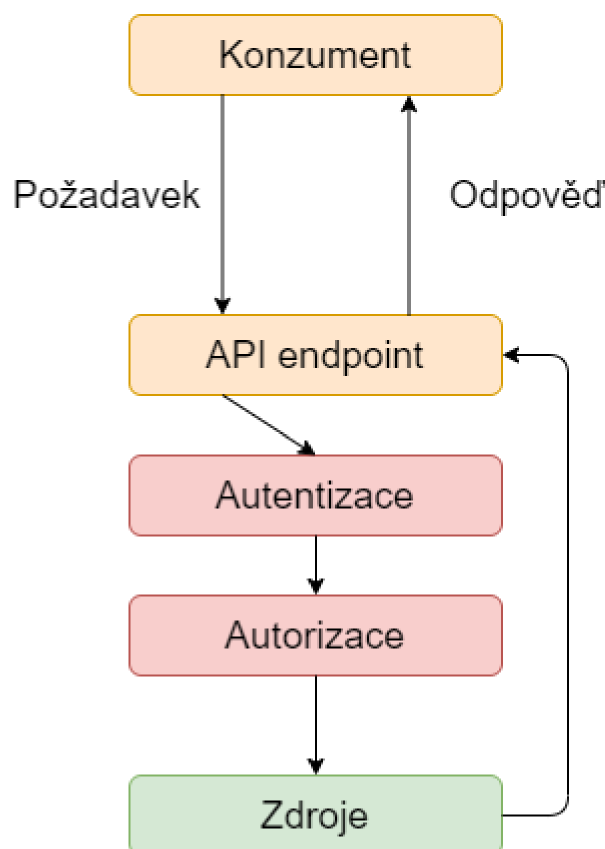
V prvním kroku autentizace aplikačního rozhraní je vystavena metoda API, která slouží k ověření uživatelské identity a zahájení uživatelského sezení. Tato metoda obecně přijímá některý z identifikačních prvků (například uživatelské jméno, heslo, biometrický údaj, RFID token, BLUE COIN ID). Po přijetí požadavku na ověření dojde k nahlédnutí do interní databáze uživatelů a v případě shody je zahájeno uživatelské sezení. V případě potřeby přístupu k chráněným zdrojům třetích stran jsou implementovány složitější mechanismy, které mají za cíl ochránit uživatelské přístupové údaje. Jako příklad lze uvést přihlašování pomocí tlačítka Google a systému oAuth, kdy uživatel použije k autentizaci svůj Google účet, který může být velmi dobře zabezpečený například dvoufaktorovou autentizací (2FA) a technologií Google Authenticator. Google vystaví přihlašovací token, který nenese žádné uživatelské přihlašovací údaje a tento token je dále používán k autentizaci uživatele v rámci další komunikace s API. Protože uživatel zadává jeho přihlašovací údaje pouze v rámci služby Google, nedojde nikde k jejich vyzrazení, což je velmi žádané z hlediska bezpečnosti a snížení možnosti kompromitace těchto údajů.

Obecně dochází k ústupu od autentizace pomocí přihlašovacích údajů k autentizaci pomocí tokenů a párů RSA klíčů (které mohou být dále chráněny heslem). Tento přechod přináší významné zvýšení bezpečnosti a je také velmi výhodný pro M2M komunikaci.

1.2 Autorizace

Proces autorizace poskytuje uživatelům přístup pouze k takovým zdrojům, které jsou oprávněni používat. Zároveň poskytuje prevenci před přístupem ke zdrojům, které nejsou pro daného autentizovaného uživatele povoleny [2].

Krok autorizace následuje bezprostředně po kroku autentizace a je na něm zcela závislý. Z hlediska náročnosti se jedná o mnohem náročnější úkol, autorizace může být mnohoúrovňová a velmi komplexní. Jako příklad může posloužit autorizace požadavků číst, editovat a mazat data pro různé skupiny uživatelů – systémový administrátor, administrátor, majitel dat, zaměstnanec, registrovaný uživatel, anonymní uživatel. Každý požadavek na API musí projít tímto procesem a musí být determinováno, zda je požadovaná akce legitimní viz. *Obr. 2*. Jakákoliv chyba v tomto kroku je považována za bezpečnostní zranitelnost a může pro provozovatele systému znamenat až fatální bezpečnostní hrozbu. Nejčastěji se provádí autorizace náhledem do interní databáze uživatelských práv a rozhodnutím, zda má daný uživatel oprávnění na danou akci. S pomocí kryptografie lze ovšem tento proces částečně zjednodušit, a to díky možnosti zakódovat do přihlašovacího tokenu vlastní data (například uživatelská práva). Tyto data je poté možné ověřit pomocí veřejného šifrovacího klíče a není tedy možné jejich podvržení. Tento systém má výhodu v tom, že nemusí být udržována žádná centrální databáze uživatelských oprávnění.



Obr. 2. Funkční diagram autentizace a autorizace.

1.3 Zabezpečení přenosu

Při procesu autentizace a samotné komunikaci s aplikačním rozhraním ve většině případů dochází k přenosu citlivých informací pomocí nedůvěryhodného přenosového kanálu – například sítě Internet, lokálních Wi-Fi sítí, LAN sítí. Tento přenos je tedy nutné zabezpečit tak, aby ho nebylo možné odposlechnout například pomocí Sniffingu a aby nebylo možné s přenášenými daty manipulovat například pomocí Man-in-the-middle útoku. Pokud by došlo ke kompromitaci přenosové cesty, vedlo by to k možné krádeži identity uživatele.

1.3.1 RSA

Algoritmus RSA dostal název podle iniciálu svých autorů (Rivest, Shamid a Adleman), kteří tento algoritmus zveřejnili v roce 1977. Algoritmus pracuje s koncepcí veřejného a privátního klíče a jedná se tedy o asymetrickou kryptografii. Lze ho použít k šifrování i podepisování dat. Jeho princip je postaven na rozkladu velkých čísel na součin prvočísel [18].

1.3.2 SSL

Protokol SSL je protokol zabezpečující transportní vrstvu. Tohoto zabezpečení je dosaženo tak, že mezi transportní vrstvu (například TCP/IP) a vrstvu aplikační (například HTTP) je vložena další vrstva, která pomocí kombinace asymetrického a symetrického šifrování (Hybridní šifrování) zabezpečuje komunikaci a také umožňuje autentizaci účastníků komunikace. Protokol byl vyvinut v 90. letech společností Netscape Communications Corporation pro nasazení na produkt webového serveru, který tato společnost nabízela. Je primárně určen pro zabezpečení HTTP komunikace. Protokol byl vydán ve třech verzích – SSL 1.0, SSL 2.0 a SSL 3.0. Vývoj verze SSL 3.0 finálně přešel do nové evoluce tohoto protokolu – zcela nového protokolu TLS [8].

SSL je klient/server protokol a poskytuje tyto základní služby:

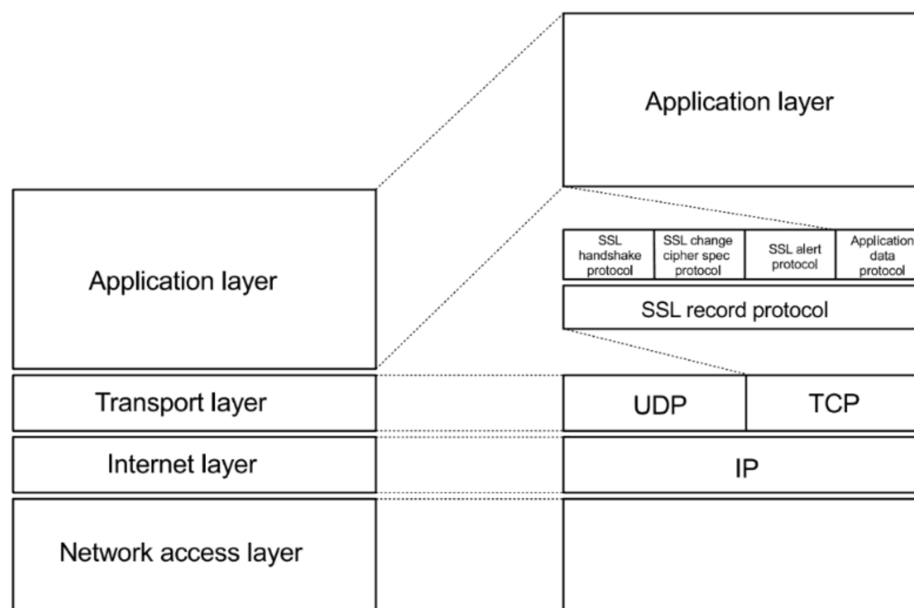
- autentizaci obou stran komunikace
- důvěryhodnost spojení
- zaručuje integritu spojení bez možnosti obnovení.

Protože SSL používá kryptografií s veřejnými klíči, nemůže tento protokol poskytnout potvrzení o identitě odesílatele. Jak napovídá název, tento protokol je orientován na sockety a to znamená, že cokoliv je odesláno do daného socketu je automaticky pomocí protokolu SSL kryptograficky chráněno.

Toto pravidlo platí také pro veškerá data přijata ze sítě na daný socket. SSL může být nejlépe chápáno jako mezivrstva, která:

- naváže zabezpečené (autentizované a důvěryhodné) spojení mezi dvěma stranami
- toto spojení je používáno k bezpečnému přenosu přenosu protokolů vyšších vrstev modelu OSI (HTTP, FTP, IMAP, SSH) viz. *Obr*. 3. SSL fragmentuje data těchto vyšších protokolů do jednotlivých fragmentů, které poté mohou být komprimovány, autentifikovány, zašifrovány a s připojenou hlavičkou připraveny k odeslání adresátovi pomocí nezabezpečené transportní vrstvy TCP, nebo UDP. Každý tento fragment je odeslán jako samostatný SSL záznam
- po obdržení tohoto fragmentu příjemcem je fragment dešifrován, autentifikován, dekomprimován, znovu sestaven a předán vyšší vrstvě modelu OSI, nejčastěji aplikační [8].

Protokol SSL je plně autonomní a pracuje zcela bez asistence TCP/UDP. Většina serverů je pro případ, že by klient nepodporoval SSL protokol, nakonfigurována tak, aby naslouchala také na nezabezpečeném portu a umožňovala tak komunikaci také nezabezpečenou cestou. Nejpoužívanější SSL porty jsou port `443` pro HTTP over SSL a port `465` pro SMTP over SSL.



Obr. 3. Schéma protokolu SSL a (pod)vrstev [8].

Na *Obr. 3* jsou znázorněny jednotlivé vrstvy komunikace v případě použití SSL. Z tohoto obrázku je patrné, jakým způsobem je SSL protokol vnořen do komunikace.

1.3.3 TLS

Protokol TLS je přímou evolucí protokolu SSL 3.0. Rozdíly mezi SSL 3.0 a TLS 1.0 jsou sice minimální, ale dostatečné na to, aby vedly k nekompatibilitě těchto protokolů. Specifikace TLS 1.0 byla uvolněna v roce 1999 organizací Internet Engineering Task Force (IETF) v dokumentu RFC 2246. Tato verze umožňuje downgrade na SSL 3.0. TLS na rozdíl od protokolu SSL umožňuje začít komunikaci nešifrovaně a šifrování je spuštěno až po inicializaci služeb jako VirtualHost (hostování více domén na jedné veřejné IP adrese). Tím odpadá nutnost poskytovat každé doméně dedikovanou veřejnou IP adresu.

TLS také umožňuje navázání spojení pomocí asymetrické kryptografie a poté pomocí výměny symetrického klíče další šifrování realizovat mnohem rychlejšími symetrickými šiframi. TLS tedy vykazuje mnohem lepší výkon než SSL. TLS také odstraňuje mnoho zranitelností protokolu SSL. Na *Výpis* 1 je znázorněn TLS certifikát v textovém formátu PEM.


```
-----BEGIN CERTIFICATE-----
MIIDazCCAlOgAwIBAgIUbAr8ZhIAmHsMwaIspCwodWbEDJIwDQYJKoZIhvcNAQEL
BQAwRTELMAkGA1UEBhMCQVUxEzARBgNVBAgMClNvbWUtU3RhdGUxITAfBgNVBAoM
GEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZDAeFw0yMDA3MTYxNTA2MjlaFw0yMDA4
MTUxNTA2MjlaMEUxCzAJBgNVBAYTAkFVMRMwEQYDVQQIDApTb21lLVN0YXRlMSEw
HwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwggEiMA0GCSqGSIb3DQEB
AQUAA4IBDwAwggEKAoIBAQDr60UcMmyntz05bc8nmYjKs5tDx2ISKvNuTuPkgeTa
zK82DD4LEUM4ltH2F3C1VUBLrhRFCdOCVCj6GERYMs2AusuSusm/DoeMvmp+XfZN
KI49Vba45ir77FiueuIs5uaCWsbL0o6IxRBYAEpOpWmI7fuX4FZv1YaXVArKkKrr
iejheMu9cW8HmFZ9Ufm0/W0cHMThkNZiZZvT+YAkGB3nUacwobQtrdSZzF3Plt1b
3xs7Pn9fLz+XDR0TzfEyRQu4MbYeeF668AxfSzbDrvSNz8/UnU80Vjdrapx4JVp9
FvBIPLaOQxt6qti0/FestgXvGyJFWKrW0n//Q0FRzIbdAgMBAAGjUzBRMB0GA1Ud
DgQWBBSrm5wneJpIwNXpdzRKSY7kDlSHhzAfBgNVHSMEGDAWgBSrm5wneJpIwNXp
dzRKSY7kDlSHhzAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3DQEBCwUAA4IBAQCp
qMsPsQpWGfxthK9Tyr6HtFhO4r44WhtYz9orA4QLO2fJvHgNFTjoJryraawvJnee
2YpmnVo5zyy7c9XIWOoYYgyBwRcldFeiwiSOHsSSUJ+qDRJieksb1CtfqI1ye4sd
db+s9aW/zTU1wRdhtR2BN/UiGN1uOL2SF4Ae65dXdUmzUdRDiFryBQAoLe2Kgv6a
Wa399at5QGx10fxfjU8jaa2tHCZqoEmMDpSCk5lb44X8MJZUINyFTBItcwBWEM8c
iDM6b9I6RC3o3eyU/+xKEgE7SN02Hlju5UJWuyv8t/r627VD8xTsZyHP7mAArWuY
u3b5b2Lk4PHxBCtA8q5a
-----END CERTIFICATE-----
```


Výpis 1. Ukázka TLS certifikátu.

1.3.4 Shrnutí verzí SSL a TLS:

SSL 1.0 – nebylo nikdy zveřejněno, specifikace obsahovala bezpečnostní zranitelnosti.

SSL 2.0 – uvolněno v roce 1995. Od roku 2011 vyřazeno. Obsahuje známé bezpečnostní zranitelnosti.

SSL 3.0 – uvolněno v roce 1996. Od roku 2015 vyřazeno. Obsahuje známé bezpečnostní zranitelnosti.

TLS 1.0 – uvolněno v roce 1999 jako upgrade SSL 3.0. Plánované vyřazení v roce 2020.

TLS 1.1 – uvolněno v roce 2006. Plánované vyřazení v roce 2020.

TLS 1.2 – uvolněno v roce 2008.

TLS 1.3 – uvolněno v roce 2018.

Z toho seznamu vyplývá, že je velmi doporučované v rámci konfigurace serveru zakázat všechny verze SSL a používat pouze TLS 1.2 výše.

1.4 Seznam použitých hashovacích funkcí

Tato kapitola se zabývá hashovacími funkcemi použitými v této práci. Hashovací funkce má obecně za úkol vytvořit digitální otisk dat (kontrolní součet) o předem stanovené délce. Její základní vlastnost musí být nekoliznost. Pro jeden kontrolní součet by tedy ideálně neměla existovat sada dvou různých zdrojových dat. Mezi známé hashovací funkce patří MD2, MD4, MD5, MDC2, SHA1 (také známá jako DSS1) a RIPEMD-160 [9].

1.4.1 SHA1

Hashovací funkce SHA1 produkuje 160 bitový hash libovolného vstupu. Ve formě čitelné pro lidi je tento hash reprezentován jako 40 číslic dlouhé hexadecimální číslo. Tato funkce již není považována za bezpečnou, stejně jako dříve MD5, a je doporučeno její nahrazení funkcemi SHA-256, nebo SHA-3 [9].

1.4.2 HMAC-SHA1

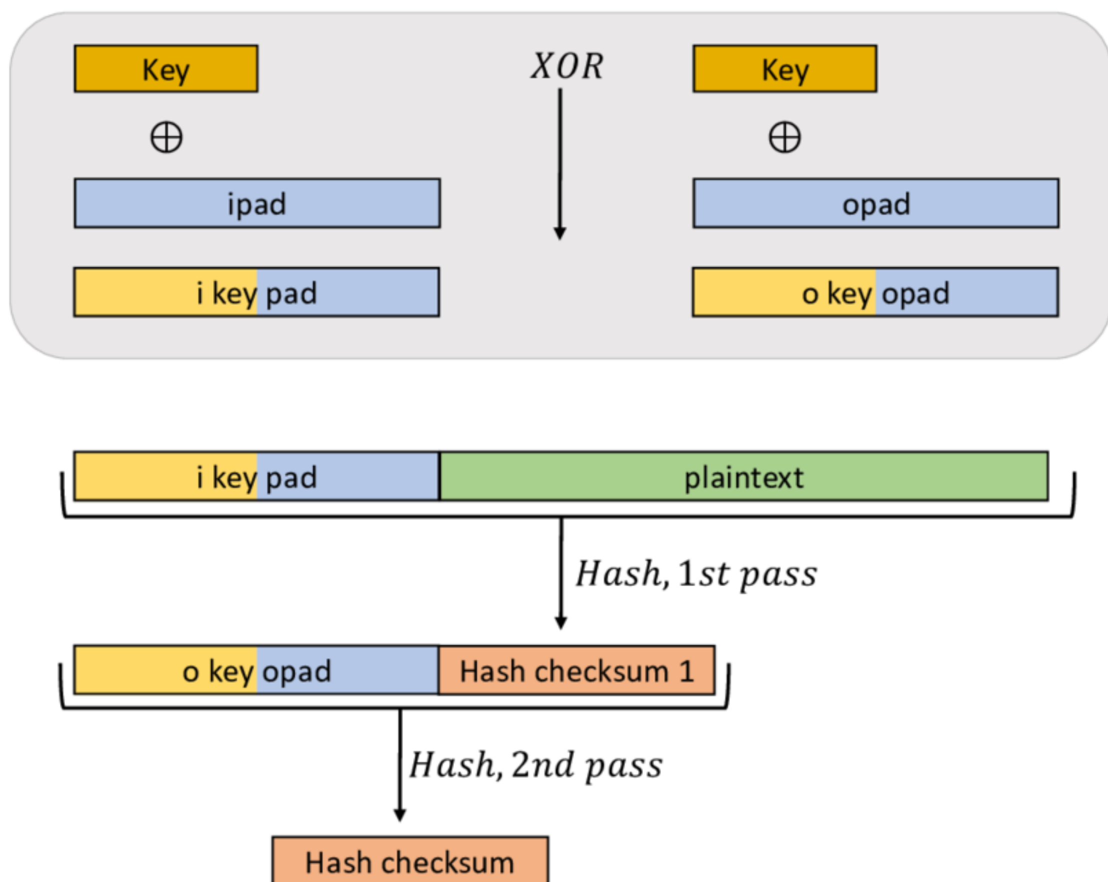
Jedná se o autentizační kód zprávy MAC založený na hash funkci SHA1. Tento kód se používá k ověření obsahu zprávy. HMAC byl navržen v roce 1996 M. Bellarem, R. Canettim a H. Krawczykem. Stal se populárním v 90. létech, protože tehdy nebyl k dispozici žádný efektivní a bezpečný MAC algoritmus.

Ve formě čitelné pro lidi se jedná o 40 znaků dlouhý textový řetězec, viz. *Výpis 2*. Tento řetězec reprezentujíce hexadecimální číslo.

```
HMAC_SHA1("tajný klíč", "Důležitá zpráva") =
a59c780aa0639dce3261b428e11c17ba421eb00c
```

Výpis 2. Ukázka výstupu HMAC-SHA1.

Hash funkce v tomto případě (například MD5, nebo SHA1) nabízí mnohem lepší softwarový výkon, než blokové šifry [6]. Princip funkce tohoto algoritmu je znázorněn na *Obr. 4*.



Obr. 4. Schéma získání kódu HMAC [10].

1.5 Metody používané při deklaraci identity

V této sekci jsou shrnuty metody, které jsou používány při autentizaci a částečně mohou být použity také při autorizaci. Tato oblast zažila v posledních letech velmi dramatický rozvoj a vzniklo mnoho standardů a knihoven, které vývojářům velmi usnadňují práci. V sekci bude shrnut vývoj těchto technologií od HTTP Authentication: Basic and Digest Access

Authentication definované v dokumentu RFC 2617 z roku 1999 společností Microsoft [3] až po moderní technologii JSON Web Token z roku 2015 [4].

1.5.1 HTTP Autentizace: Základní a přehledové ověřování přístupu

Jedná se o jednu z prvních specifikací určených k autentizaci uživatelů v rámci služby WWW. K autentizaci je použito uživatelské jméno a heslo přenášené ve formě čistého textu – cleartext, nebo hashované pomocí dnes již nevyhovující funkce MD5. K přenosu je použit protokol HTTP, konkrétně jeho hlavička a její pole. Oba způsoby lze považovat za nezabezpečené a je tedy nutné použít další vrstvu zabezpečení, například TLS. Definuje ji dokument RFC 2617.

HTTP poskytuje jednoduchý mechanismus autentizace pomocí výzvy, která může být zobrazena serverem při vypořádání požadavku. Klient poté poskytne ověřovací informace a vloží je do hlavičky požadavku do pole WWW-Authenticate. Tato autentizace využívá rozšiřitelné tokeny nerozlišující velká a malá písmena k identifikaci schématu autentizace, následovaný seznamem dvojic atribut-hodnota oddělených čárkami, které nesou parametry nezbytné pro dosažení autentizace prostřednictvím tohoto systém [3].

1.5.2 Shrnutí HTTP Autentizace

Výhody:

- nativní podpora ve všech moderních prohlížečích.

Nevýhody:

- přihlašovací údaje jsou vkládány do každého požadavku
- není možné poskytnout přístup třetím stranám bez vyzrazení přihlašovacích údajů
- není zahájena žádná relace
- překonaný typ autentizace.

1.5.3 Vzájemné ověřování

Tato metoda nabízí oboustranné ověření identity jak klienta, tak serveru. Toto ověření probíhá nejčastěji pomocí certifikátů, a to u služeb typu SSH. Vzhledem k použití certifikátů je tento způsob ověření nasazován především v B2B aplikacích, kde spolu komunikuje menší počet účastníků a zvýšená administrativní zátěž na správu certifikátů tedy nepředstavuje velkou překážku.

Shrnutí Mutual Authentication

Výhody:

- vysoká bezpečnost
- ověření identity všech účastníků.

Nevýhody:

- nutnost práce s šifrovacími klíči
- potřeba dodržení pravidel při práci s šifrovacími klíči
- vhodné spíše pro B2B aplikace.

1.5.4 OAuth 1.0

OAuth je otevřený standard vytvořený přímo za účelem standardizace autentizace přístupů do API rozhraní. Hlavní motivací pro vytvoření tohoto standardu byla snaha poskytnutí API rozhraní třetím stranám v průmyslovém rozsahu. Práci na standardu zahájil kanadský softwarový inženýr Blaine Cook pracující na API rozhraní internetové služby Twitter v roce 2006.

Tento standard využívá tři typy tokenu a zcela zamezuje vyzrazení přihlašovacích údajů. Při správné implementaci přihlašovací údaje neopustí klientské zařízení.

Terminologie

Konzument: klient (služba webové tiskárny).

Poskytovatel služby: server (cloudové fotoalbum).

Uživatel: vlastník chráněných zdrojů (uživatel František).

Konzumentský identifikátor a klíč: client credentials.

Požadavkový identifikátor a klíč: dočasné přihlašovací údaje.

Přístupový identifikátor a klíč: token credentials.

Příklad použití dle RFC 5849

V dokumentu RFC 5849 [5] je uveden následující názorný příklad.

František (vlastník zdroje) nahrál fotky ze soukromé dovolené na jeho oblíbený web pro sdílení fotografií 'photos.example.net' (server). Chtěl by použít web 'printer.example.com' (klient) pro tisk jedné z těchto fotografií.

František se obvykle přihlašuje do „photos.example.net" pomocí svého uživatelského jména a hesla. František si však nepřeje sdílet své uživatelské jméno a heslo do služby serveru (photos.example.net) s klientem (printer.example.com).

Klient 'printer.example.com', který potřebuje přístup k fotografii, aby mohl realizovat objednávku pro tisk s cílem poskytnout svým uživatelům lepší službu si zaregistruje konzumentský identifikátor a klíč vygenerovaný serverem „photos.example.net" pro klienta „printer.example.com":

Klientský identifikátor

```
dpf43f3p2l4k3l03
```

Klientský klíč:

```
kd94hf93k423kf44
```

U klientského klíče se jedná o Shared Secret, tedy klíč sdílený konzumentem i serverem.

Po obdržení tohoto páru provede klient konfiguraci svého rozhraní tak, aby používal END point uvedený v dokumentaci serveru „photos.example.net", který musí používat podpisovou metodu "HMAC-SHA1".

END Point dočasné přístupové požadavky

```
https://photos.example.net/initiate
```

END Point autorizační URI vlastník chráněných zdrojů:

```
https://photos.example.net/authorize
```

END Point požadavků na tokeny URI:

```
https://photos.example.net/token
```

Předtím, než klient „printer.example.com“ může požádat vlastníka chráněných zdrojů Františka o přístup k jeho fotkám, musí nejprve požádat server „photos.example.net“ o sadu požadavkového identifikátoru a klíče sloužícího k identifikaci daného požadavku.

Klient tedy pošle následující HTTPS požadavek:

```
POST /initiate HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
      oauth_consumer_key="dpf43f3p2l4k3l03",
      oauth_signature_method="HMAC-SHA1",
      oauth_timestamp="137131200",
      oauth_nonce="wIjqoS",
      oauth_callback="http%3A%2F%2Fprinter.example.com%2Fready",
      oauth_signature="74KNZJeDHnMBp0EMJ9ZHt%2FXKycU%3D"
```

Server provede validaci tohoto požadavku a jako odpověď zašle sadu dočasného identifikátoru a klíče.

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded
      oauth_token=hh5s93j4hdidpola&
      oauth_token_secret=hdhd0244k9j7ao03&
      oauth_callback_confirmed=true
```

Klient přesměruje uživatele Františka na END Point autorizační URI vlastníka chráněných zdrojů, kde uživatel provede autorizaci požadavku na přístup serveru k požadovaným chráněným zdrojům.

```
https://photos.example.net/authorize?oauth_token=hh5s93j4hdidpola
```

Server si vyžádá přihlášení uživatele František pomocí uživatelského jména a hesla a pokud je toto přihlášení úspěšné, požádá o potvrzení udělení přístupu 'printer.example.com' k jeho soukromým fotkám. Uživatel František provede toto potvrzení, načež je přesměrován zpět na URI poskytnutou klientem v předcházejícím požadavku.

```
http://printer.example.com/ready?
      oauth_token=hh5s93j4hdidpola&oauth_verifier=hfdp7dh39dks988
```

Tento zpětný požadavek informuje klienta o tom, že uživatel dokončil autorizační proces. Klient nyní zašle požadavek na server a vyžádá si sadu přístupového identifikátoru a klíče pro trvalý přístup.

Toto volání musí být uskutečněno pomocí TLS zabezpečeného kanálu tak, aby nemohlo dojít k vyzrazení páru identifikátoru a klíče.

```
POST /token HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
     oauth_consumer_key="dpf43f3p2l4k3l03",
     oauth_token="hh5s93j4hdidpola",
     oauth_signature_method="HMAC-SHA1",
     oauth_timestamp="137131201",
     oauth_nonce="walatlh",
     oauth_verifier="hfdp7dh39dks9884",
     oauth_signature="gKgrFCywp7rO0OXSjdot%2FIHF7IU%3D"
```

Server provede validaci požadavku a zašle vygenerovaný identifikátor a klíč v těle HTTP odpovědi.

```
HTTP/1.1 200 OK
     Content-Type: application/x-www-form-urlencoded
     oauth_token=nnch734d00sl2jdk&oauth_token_secret=pfk-
     kdhi9sl3r4s00
```

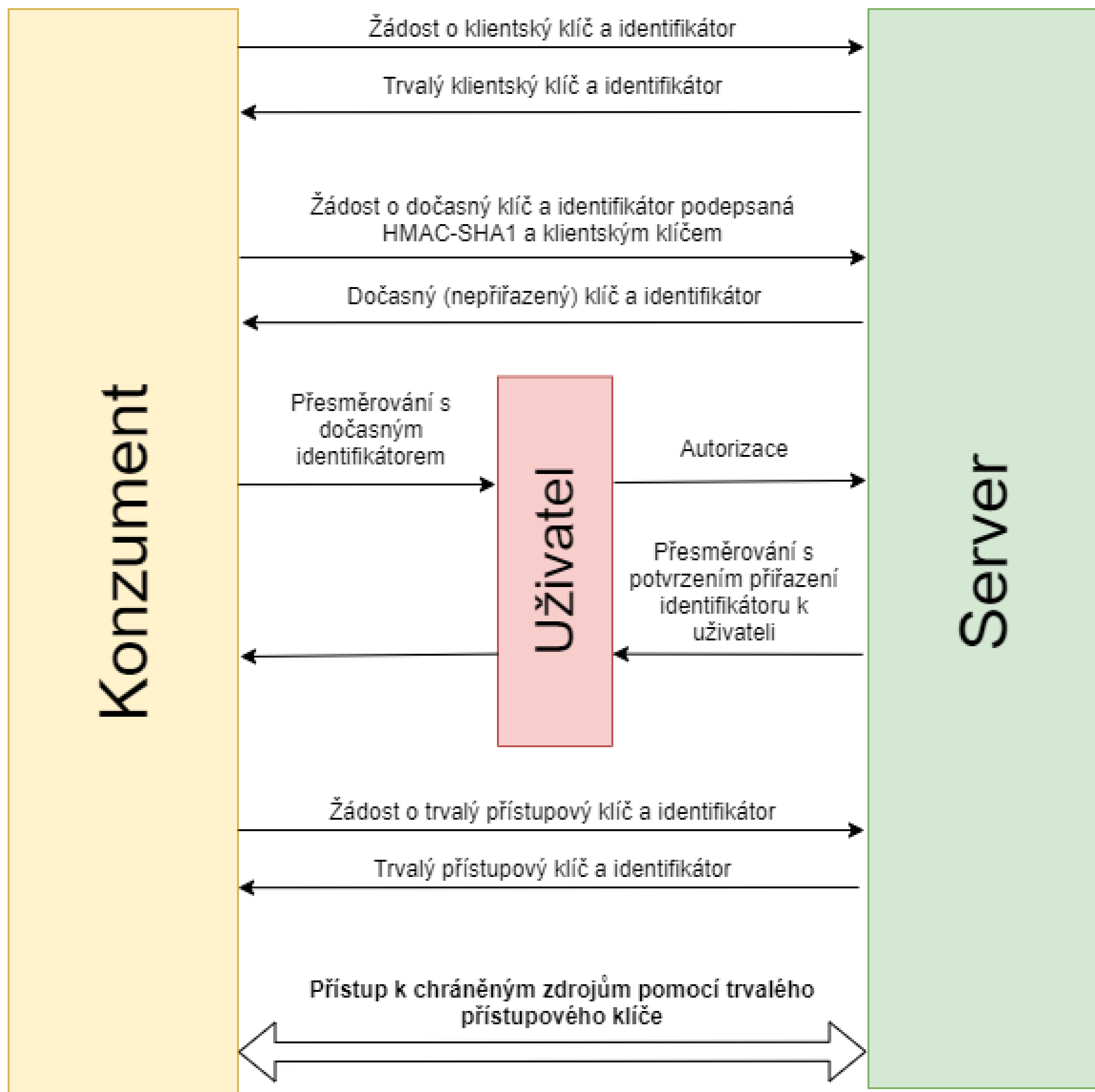
Pomocí dvojice identifikátoru a klíče nyní může konzument přistupovat k soukromým fotografiím uživatele František uloženým na serveru.

```
GET /photos?file=vacation.jpg&size=original HTTP/1.1
Host: photos.example.net
Authorization: OAuth realm="Photos",
     oauth_consumer_key="dpf43f3p2l4k3l03",
     oauth_token="nnch734d00sl2jdk",
     oauth_signature_method="HMAC-SHA1",
     oauth_timestamp="137131202",
     oauth_nonce="chapoH",
     oauth_signature="MdpQcU8iPSUjWoN%2FUDMsK2sui9I%3D"
```

Server 'photos.example.net' nyní bude validovat každou žádost o přístup k fotkám a odpovídat požadovanými daty, dokud nedojde ke zneplatnění identifikátoru a klíče anebo dokud

nedojde k vypršení jejich platnosti. Konzument 'printer.example.com' tedy přistupuje k těmto zdrojům pomocí stále stejné dvojice identifikátoru a klíče [5].

Princip funkce oAuth 1.0 je shrnut na *Obr. 5.*



Obr. 5. Schéma fungování OAuth 1.0.

Délky tokenů

Authorizační token: 256 bajtů.

Přístupový token: až 2048 bajtů.

Obnovovací token: 512 bajtů.

Z uvedených velikostí tokenů je zřejmé, že se jedná o velmi bezpečnou metodu z pohledu útoku hrubou silou. Pro srovnání velikost 2048 B odpovídá heslu o délce 2048 náhodných ASCII znaků, přičemž u hesel obecně nelze předpokládat entropii srovnatelnou se správně vygenerovaným náhodným tokenem.

Shrnutí OAuth 1.0

Výhody:

- možnost propojení s širokou škálou služeb jako Google, Facebook, Twitter
- při propojení například s Google API velmi jednoduchá implementace 2 faktorového ověřování
- velmi dobře prověřený a v průmyslu široce nasazený standard
- široká škála dostupných knihoven pro mnoho programovacích jazyků

Nevýhody:

- přílišná komplikovanost
- v případě autentizace pouze pomocí hesla a přihlašovacího jména zbytečně náročné řešení.

1.5.5 OAuth 2.0

OAuth 2.0 je nová specifikace OAuth, která odstranila některé nedokonalosti OAuth 1.0. OAuth 2.0 bylo přepsáno zcela od začátku a sdílí s OAuth 1.0 pouze hlavní cíle. OAuth 2.0 reprezentuje roky diskusí mezi společnostmi jako Google, Facebook, Microsoft, Deutsche Telekom, Mozilla a dalších.

Byla přidána lepší podpora pro aplikace, které nejsou založeny na webu a byly tedy definovány nové mechanismy, které umožňují uživatelské autorizace i bez přesměrování do webového prohlížeče. Toto vede především ke zlepšení uživatelského zážitku.

Většina selhání pokusů o implementaci OAuth 1.0 měla společného jmenovatele – kryptografické požadavky protokolu. OAuth 2.0 nevyžaduje, aby klientská aplikace obsahovala kryptografii. V případě OAuth 1.0 je nutné připojit HMAC-SHA1 hash zprávy, OAuth 2.0 umožňuje zpracovat takový požadavek bez HMAC-SHA1. Je dostačující, pokud konzument zašle požadavek pomocí HTTPS s připojí platný token.

Podpisy OAuth 2.0 jsou mnohem jednodušší na implementaci, protože nevyžadují žádné speciální parsování a kódování.

Přístupové tokeny OAuth 2.0 mají definovanou časovou platnost. Konzument může také požádat o obnovovací token, který nemá omezenou platnost. Pomocí tohoto obnovovacího tokenu poté může žádat o přístupové tokeny s omezenou platností.

Terminologie

Klient (služba webové tiskárny).

Poskytovatel služby (cloudové fotoalbum).

Uživatel (uživatel František).

Klientský identifikátor a klíč: client credentials.

Bearer token

Bearer token je přístupový token na doručitele. Tento token není určen pro uživatele a je pro něj nesrozumitelný. Tokeny mohou být buď přímo řetězce hexadecimálních znaků, nebo případně strukturované objekty (např. JSON Web Token). Tyto tokeny musí být uloženy na důvěryhodném místě tak, aby k nim neměly přístup další strany a je tedy nutné s nimi zacházet ve stejném režimu jako s privátním klíčem. Také je vyžadováno tyto tokeny přenášet pouze pomocí zabezpečeného kanálu (HTTPS), protože jejich odposlechnutí, například pomocí man-in-the-middle útoku, by jinak bylo triviální.

Autorizace pověřením klienta

Tato metoda je používána, pokud aplikace požadují přístup k vlastním zdrojům a nedochází tedy k autorizaci přístupu ke službám žádného poskytovatele služby. Používá se u M2M API volání.

Požadované parametry:

grant_type Musí být nastaveno na "client_credentials".

client_id ID klienta

client_secret heslo klienta

Příklad požadavku:

```
POST /oauth/token HTTP/1.1
Host: authorization-server.com


grant_type=authorization_code
&code=xxxxxxxxxxx
&redirect_uri=https://example-app.com/redirect
&client_id=xxxxxxxxxx
&client_secret=xxxxxxxxxx
```

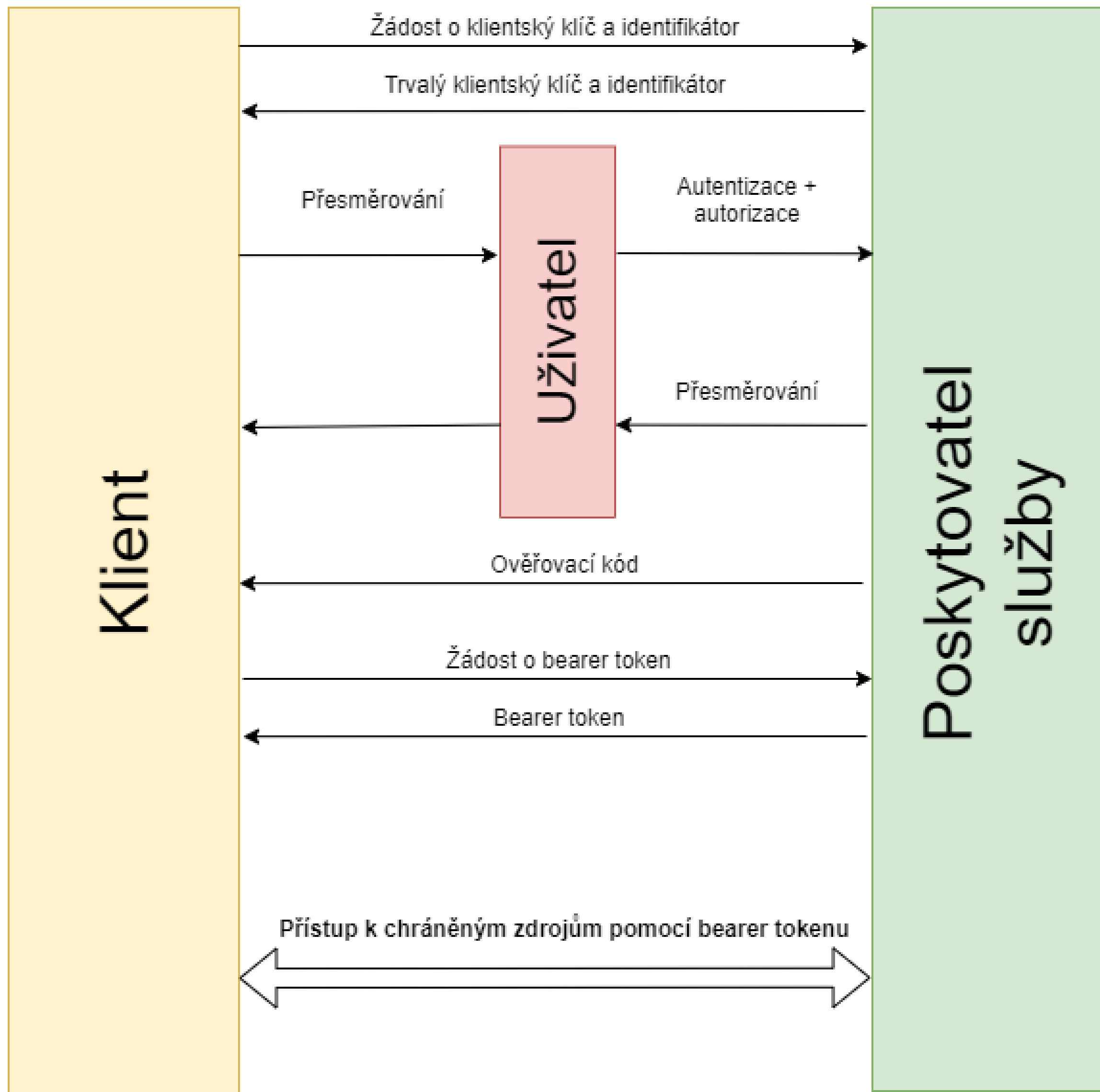
Odpověď serveru:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache
{
  "access_token":"MTQ0NjJkZmQ5OTM2NDE1ZTZjNGZmZjI3",
  "token_type":"bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2YxOTQ5MGE3YmNmMDFkNTVk",
  "scope":"create"
}
```

Autorizace pomocí ověřovacího kódu

Tato metoda je určena pro případy, kdy klient požaduje přístup k poskytovateli služby. Pro její použití musí mít klient vygenerovaný a zaregistrovaný klientský identifikátor a klíč. Klient provede přesměrování uživatele na přihlašovací stránku poskytovatele služby, kde uživatel, pomocí svých uživatelských údajů, provede autentizaci a autorizaci přístupu klienta k chráněnému zdroji. V případě úspěšné autentizace a autorizace poskytovatel služby vygeneruje přístupový klíč, který je předán klientovi. Klient poté odešle požadavek na výměnu ověřovacího kódu za přístupový token, který je dále používán k přístupu k chráněným zdrojům.

Důležitá vlastnost ověřovacích kódu je jejich jednorázovost a také provázanost s daným uživatelem. Princip funkce oAuth 2.0 je shrnut na *Obr. 6*.



Obr. 6. Schéma metody autorizace pomocí ověřovacího kódu.

Příklad požadavku:

```
POST /oauth/token HTTP/1.1
Host: authorization-server.com


grant_type=authorization_code
&code=xxxxxxxxxxx
&redirect_uri=https://example-app.com/redirect
&client_id=xxxxxxxxxx
&client_secret=xxxxxxxxxx
```

Odpověď serveru:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache


{
  "access_token":"MTQ0NjJkZmQ5OTM2NDE1ZTZjNGZmZjI3",
  "token_type":"bearer",
  "expires_in":3600,
  "refresh_token":"IwOGYzYTlmM2YxOTQ5MGE3YmNmMDFkNTVk",
  "scope":"create"
}
```

Shrnutí OAuth 2.0

Výhody:

- nová verze ověřeného oAuth
- široce rozšířeno a podpora u mnoha služeb
- nižší nároky na implementaci než oAuth 1.0.

Nevýhody:

- vyžaduje uživatelskou interakci
- vhodné spíše pro spojení se službami třetích stran.

1.5.6 JWT (JSON Web Tokens)

JSON WebTokens je otevřený průmyslový standard definovaný dokumentem RFC 7519. Tento standard definuje způsob bezpečné deklarace nároků mezi dvěma stranami [11].

Encoded PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiaWF0IjoxNTE2MjM5MDIyLCJ1xb5pdmF0ZWwiOiJGcmFudGnFoWVrIiwicG96aWNlIjoiemFtxJtzdG5hbmVjIiwidXNlcklkIjoxfQ.LeFaM8Hgn-NXs0jweQ6HNjtwz2g14j-IFE3L67CtVIg

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "iat": 1516239022,
  "uživatel": "František",
  "pozice": "zaměstnanec",
  "userId": 1
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Obr. 7. JSON zašifrovaný do JWT tokenu [11].

JWT využívá formát JSON, tento formát je definován v dokumentu RFC 8259 a vychází ze syntaxe skriptovacího jazyka JavaScript. JWT token obsahuje hlavičku, payload a podpis viz. *Obr. 7*.

JSON je určen k přenosu strukturovaných dat a má definované následující datové typy viz. *Tab. 1* [12].

Tab. 1. Datové typy JSON.

Datový typ	Možné hodnoty
JSONString	textový řetězec
JSONNumber	reálné číslo včetně zápisu exponenta
JSONBoolean	logická hodnota reprezentující TRUE, nebo FALSE
JSONNull	hodnota null
JSONArray	pole
JSONObject	objekt

Díky JWT lze vytvořit vlastní bearer tokeny, které nesou užitečné informace vložené do JSON objektu (na *Obr. 7* v části PAYLOAD). Do tokenů je zakódován payload a poté jsou podepsány pomocí jednoho z mnoha možných algoritmů. Informace v nich obsažené jsou tedy věrohodné a není možná jejich kompromitace třetí stranou ani držitelem.

Hlavní vlastnosti:

- díky jejich kompaktní velikosti mohou být velmi snadno připojeny do každé HTTP hlavičky, POST parametru, nebo URL parametru
- díky jejich malé velikost je jejich přenos efektivní a rychlý
- tokeny mohou obsahovat všechny potřebné informace a po jejich dekódování není nutné stále dotazovat databázový systém, což přináší mnoho benefitů například z hlediska škálovatelnosti API
- jeden token může být využit pro více API a více domén pouze pomocí sdílení klíče
- tokeny zajišťují stateless API – není nutné udržovat žádný sdílený datový sklad.

Příklady použití:

- autentizace, po přihlášení je uživateli vydán bearer token, který je použit v další komunikaci k autentizaci
- autorizace, vydaný bearer token může nést informaci nutnou k autorizaci API volání
- výměna informací, bearer token je dobrý způsob výměny důvěryhodných informací mezi různými stranami.

Struktura JWT tokenů

JWT tokeny se skládají ze tří částí spojených tečkou do jednoho textového řetězce.

1. Hlavička viz. *Výpis 3.*
2. Libovolná užitečná data viz. *Výpis 4.*
3. Podpis viz. *Výpis 5.*

Skladba tokenu je následující: `111111.2222222.3333333`

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

Výpis 3. Příklad hlavičky JWT tokenu [11].

Na *Výpis 3* je zobrazena dekódovaná hlavička JWT tokenu, která obsahuje typ použitého algoritmu a typ tokenu.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Výpis 4. Příklad užitečných dat JWT tokenu [11].

Na *Výpis 4* je zobrazen payload, který obsahuje identitu subjektu sub, jméno subjektu a časové razítko vystavení tokenu.

Na *Výpis 5* je zobrazen způsob výpočtu tokenu.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  256-bit-secret
)
```

Výpis 5. Následný výpočet podpisu JWT tokenu [11].

Shrnutí JWT tokenů

Výhody:

- možnost zakódování vlastních užitečných dat ve formě payloadu
- není nutné provozovat centralizovanou databázi
- tokeny lze sdílet napříč různými systémy
- velmi rozšířený způsob autentizace s širokou podporou.

Nevýhody:

- zvýšené nároky na kryptografii
- tokeny nejsou čitelné pro lidské uživatele.

2 TECHNOLOGIE APLIKAČNÍCH ROZHRANÍ

V této sekci jsou shrnuty hlavní technologie používané pro provoz aplikačních rozhraní.

2.1 SOAP

SOAP je standardizovaný protokol pro zprávy sdílené mezi aplikacemi. Specifikace nedefinuje nic víc než XML obálku pro data, která se mají přenést, viz. *Výpis 6*. Dále obsahují sadu pravidel pro překládání konkrétních datatypů specifických pro dané platformy do formátu XML [21].

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"


<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Výpis 6. Ukázka HTTP požadavku SOAP [21].

Poprvé byl tento protokol definován v roce 1998 Davem Winerem, Donem Boxem, Bobem Atkinsonem a Mohsenem Al-Ghoseinem jako XML-RPC.

Jedná se o v současné době již velmi zastaralou technologii, která je postupně nahrazována.

2.2 HTTP REST API

REST je architektura, která byla reprezentována po roce 2000 Royem Fieldingem v disertační práci [22] jakožto odpověď na krizi škálovatelnosti, která nastala na World Wide Webu. Architektura REST je zaměřena na distribuované prostředí. Vzhledem k velmi dynamickému růstu WWW vyvstala potřeba vyvinout nové způsoby provozu aplikací tak, aby

bylo možné rychle navyšovat množství odbavených klientů. To znamenalo přechod od monolitických aplikací k architektuře navzájem propojených microslužeb. Toto propojení ve většině případů obstarává právě REST API.

Architektura REST také umožňuje přechod z aplikací renderovaných na serveru, které vyžadují při každé interakci nový HTTP požadavek a kompletní render aplikace ze serveru, na jednostránkové aplikace – SPA, kdy je každému klientovi poskytnuta celá aplikace pouze jedenkrát a poté již veškeré renderování probíhá na straně klienta a na pozadí je uskutečňována pouze komunikace s API.

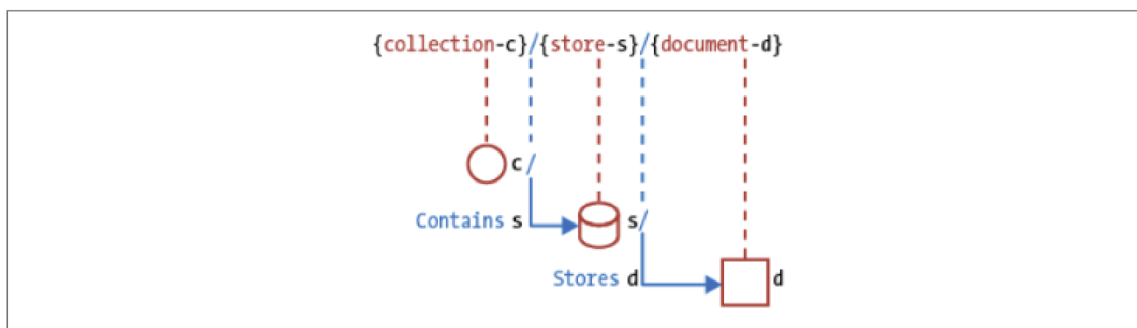
2.2.1 První úroveň – URI

Architektura REST je založena na klíčové vlastnosti a to, že každý jednotlivý zdroj má vlastní END Point, který má definovanou strukturu tzv. URI viz. *Obr. 8*.

Příklad URI:

```
GET /company/10/users
Host: https://api.example.com
```

Toto API volání vrátí seznam všech uživatelů společnosti s ID 10.



Obr. 8. Schéma URI [13].

2.2.2 Druhá úroveň – metody CRUD

Architektura REST využívá čtyři typy HTTP metod k oddělení funkcí volání. Tyto metody byly nazvány zkratkou CRUD – Create, Read, Update a Delete. K ukázkám požadavků u jednotlivých metod je využita služba JSONPlaceholder určena pro prototypování a testování REST API.

HTTP metoda GET

GET (CRUD Retrieve) slouží k získání zdrojů v režimu read-only. Zavoláním této metody na konkrétní URI dojde k vrácení požadovaných zdrojů.

Příklad požadavku:

```
GET /posts/1
Host: https://jsonplaceholder.typicode.com
```

Příklad odpovědi:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expe-
  dita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum
  rerum est autem sunt rem eveniet architecto"
}
```

Možné návratové HTTP stavy této metody jsou 200 (OK), 404 (Not Found), pokud nebylo ID nalezeno, nebo není validní.

HTTP metoda POST

POST (CRUD Create) slouží k vytváření dat na serveru klientem. Musí být využita přesně stanovená URI, protože klient nezná identifikátor vytvářeného dokumentu, který je přidělen až serverem při procesu tvorby tohoto dokumentu. Dobrá praxe je navrácení vytvořeného dokumentu serverem v odpovědi z důvodu možné kontroly časových razítek a ID klientem.

Příklad požadavku:

```
POST /posts
Host: https://jsonplaceholder.typicode.com
userId=1&title=Titulek&body=Body
```

Příklad odpovědi:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "userId": 1,
  "id": 2,
  "title": "Titulek",
  "body": "Body"
}
```

Možné návratové HTTP stavy této metody jsou 404 (Not Found), 409 (Conflict) pokud zadané ID již existuje.

HTTP metoda PUT

PUT (CRUD Update/Replace) slouží k úpravě a náhradě dat na serveru klientem. Musí být využita přesně stanovená URI obsahující identifikátor dokumentu u kterého má být provedena požadovaná akce.

Příklad požadavku:

```
PUT /posts/1
Host: https://jsonplaceholder.typicode.com
{
      "id": "1",
      "title": "foo",
      "body": "bar",
      "userId": "1"
}
```

Příklad odpovědi:

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "id": 1,
  "title": "foo",
  "body": "bar",
  "userId": "1"
}
```

Možné návratové HTTP stavy této metody jsou 200 (OK), 204 (No Content), nebo 404 (Not Found) pokud ID nebylo nalezeno, nebo není validní.

HTTP metoda DELETE

DELETE (CRUD Delete) slouží k odstranění dat na serveru klientem. Musí být využita přesně stanovená URI obsahující identifikátor dokumentu, který má být odstraněn.

Příklad požadavku:

```
DELETE /posts/1
Host: https://jsonplaceholder.typicode.com
```

Příklad odpovědi:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

Možné návratové HTTP stavy této metody jsou 200 (OK), nebo 404 (Not Found) pokud ID nebylo nalezeno, nebo není validní.

2.3 GraphQL – nástupce REST

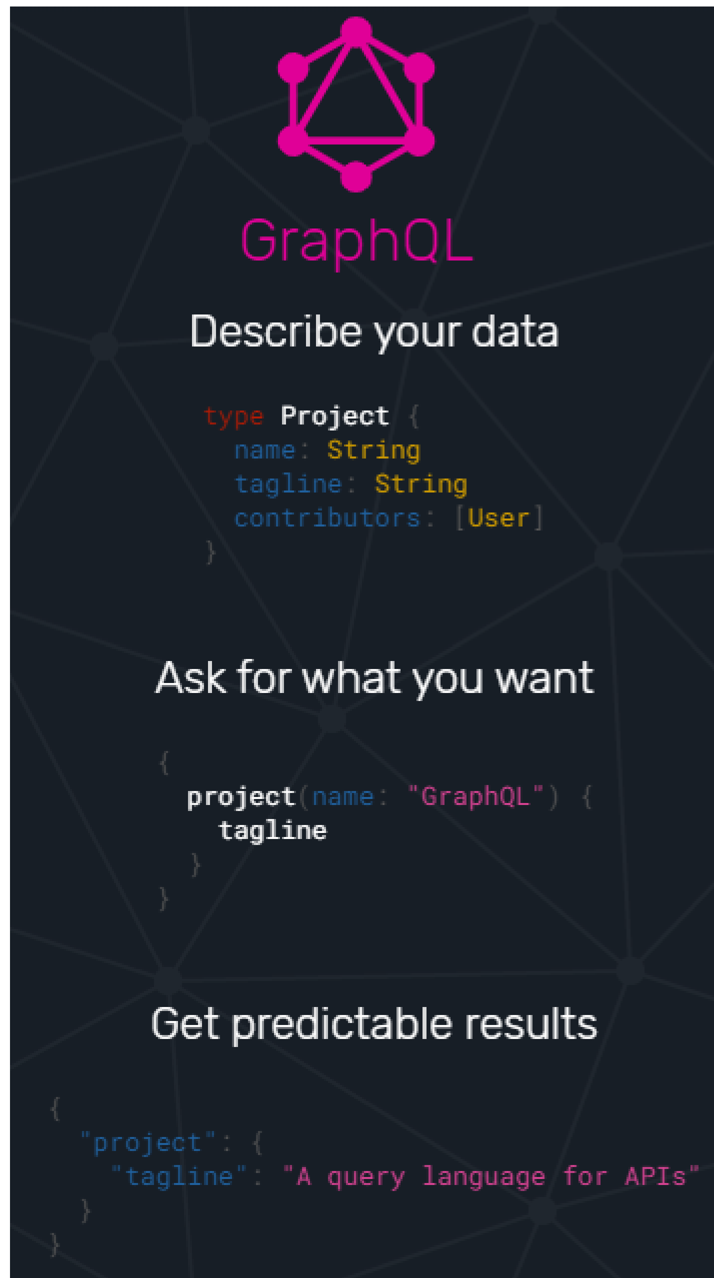
Z výše provedené analýzy REST aplikačního rozhraní je patrná jeho složitost z hlediska implementace a údržby. Pro každou metodu je nutné implementovat patřičné metody a také pro každý typ dat je nutné implementovat dedikovaný URI endpoint. Dále se přidává velmi složité verzování, kdy je nutné udržovat řadu verzí endpointů z důvodu nutnosti souběžného provozu více verzí klientských aplikací, což časem vede k neudržitelné a nepřehledné situaci.

GraphQL je specifikace dotazovacího jazyka pro API, který je přímou náhradou architektury REST a má za cíl odstranit výše zmíněné problematické aspekty REST API. GraphQL bylo vyvinuto společností Facebook nejprve jako interní projekt, aby bylo následně v roce 2015 uvolněno veřejnosti jako OpenSource [15].

Tato specifikace definuje jediný endpoint, kdy namísto velkého počtu různých endpointů pro jednotlivé typy dat je celý dotaz klienta sloučen do jednoho dotazu a poté doručen na server pomocí HTTP metody POST. Podobně jako u databází tedy klient přímo pomocí dotazu definuje, o jaká data má zájem a server GraphQL poté provede rozparsování dotazu a jeho vyřízení. Toto vyřízení probíhá pomocí resolverů viz. *Výpis 7*, které jsou definovány pro každý typ dat a mohou obsahovat zcela libovolné napojení na další API, nebo do rozličných datových zdrojů a microslužeb. Tyto resolvery jsou dotázány s ohledem na požadavky aktuálně zpracovávaného dotazu, jejich návratová hodnota je dále zpracována a poté vrácena klientovi ve formě odpovědi na jeho dotaz. Resolvery mají formu asynchronních funkcí a jsou spouštěny asynchronně, což dále zvyšuje rychlost a škálovatelnost API.

Princip funkce znázorňuje *Obr. 9*, kdy celý proces je rozdělen do tří kroků:

1. Popište svá data.
2. Definujte svůj požadavek.
3. Dostanete předvídatelnou odpověď.



Obr. 9. Znázornění funkce GraphQL [15].

GraphQL má přesně definované a staticky typované schéma a z tohoto důvodu je kdykoliv zcela zřejmá jaká data lze požadovat a jakou odpověď lze očekávat.

GraphQL je zcela nezávislé na platformě a má nespočet implementací v široké škále programovacích jazyků. Také je vysoce multiplatformní, protože klient může být implementován ve zcela jiném jazyce než server. Těchto klientů může být také na server napojeno libovolné množství.

Koncepce CRUD (CREATE, READ, UPDATE, DELETE), kterou implementoval předchůdce GraphQL – REST, byla zjednodušena na koncepci Query (čtení) a Mutací (vytváření, editace a mazání dat), kdy jsou oba typy dotazů realizovány pomocí HTTP metody POST. Toto vedlo k zásadnímu zjednodušení implementace a údržby rozhraní.

Dalším velkým vylepšení je odpadnutí nutnosti verzování z důvodu kompatibility, protože požadavek je přesně určen přímo v dotazu a různé verze klienta tedy mohou dotazovat různé datové struktury, přičemž server je stále schopen je obsloužit.

V neposlední řadě došlo k optimalizaci datových přenosů, protože GraphQL po zpracování resolverových funkcí provede vyfiltrování návratových hodnot a dále na klienta již ve formě odpovědi odešle jen konkrétně vyžádána data.

```
const resolvers = {
  Query: {
    books: () => books,
  },
};
```

Výpis 7. Ukázka resolverové funkce v implementaci GraphQL pomocí knihovny Apollo Server. [16]

3 VOLBA VHODNÝCH PROSTŘEDKŮ

V této sekci budou zvoleny vhodné prostředky pro vytvoření aplikačního rozhraní ACCESS systému a jeho zabezpečení pomocí šifrování přenosu, autentizace a autorizace.

3.1 Node.js

Jedná se o Javascriptový serverový runtime naprogramovaný v jazyce C++ využívající jádro V8 pocházející z prohlížeče Chrome. Tento runtime doplnil V8 o funkce typické pro serverové použití tak, aby mohl být plnohodnotně nasazen na server. Typicky se jedná o funkce pro práci se soubory, práci se sítí a podobně. Tyto funkce jsou doplněny ve formě nízkoúrovňového C++ a C kódu a jsou nabaleny do V8 Javascriptového engine. Ve výsledku vznikl velmi robustní runtime, který umožňuje používání technologie typické pro frontend (Javascript) také na backendu a tím šetří spoustu času, kdy vývojáři nejsou nuceni používat rozdílné technologie v různých částech aplikace.

Obr. 10. Logo runtime Node.js [7].

Node.js umožňuje psaní velmi škálovatelných serverových aplikací s využitím asynchronních I/O operací.

Tento runtime je poskytován pro všechny hlavní architektury – Linux, Windows, macOS.

Součástí ekosystému Node.js je také balíčkovací systém NPM, který v současné době obsahuje přes jeden milion balíčků, které v souhrnu dosahují padesáti miliard měsíčních stažení. Většina těchto balíčků má formu knihoven pod licencí OpenSource a stojí za nimi aktivní komunita přispěvovatelů. Lze tedy získat velmi dobře otestované a stabilní balíčky.

Node.js je distribuován pod OpenSource licencí MIT. Projekt má přes 370 vývojářů a velikost komunity vývojářů se odhaduje na statisíce. Mezi korporátní uživatele se řadí například

společnosti Mastercard, Netflix, Wallmart, Uber, PayPal, LinkedIn, Microsoft, Ebay, NASA, Aliexpress a další.

Díky asynchronní povaze runtimu lze například velmi jednoduše sestavit webový server schopný obsloužit velké množství požadavků. Jak je patrné z *Výpis 8*, každý HTTP požadavek u takového server je obsluhován pomocí izolované asynchronní funkce. Tyto funkce jsou vytvářeny a spuštěny přímo za běhu.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Výpis 8. Ukázka sestavení webového serveru pomocí Node.js [7].

3.1.1 ECMAScript

V rámci implementace aplikačního rozhraní je použit JavaScript ve verzi ECMAScript 2017 a to z důvodu své velmi široké rozšířenosti, dostupnosti knihoven a vysoké produktivity práce vykazované při používání tohoto jazyka.

ECMAScript je sada standardů vydávaná společností European Computer Manufacturers Association (ECMA).

Jedná se o novou specifikaci jazyka Javascript, která implementuje pokročilé funkce pro obsluhování asynchronních operací, například koncept asynchronních funkcí async / await.

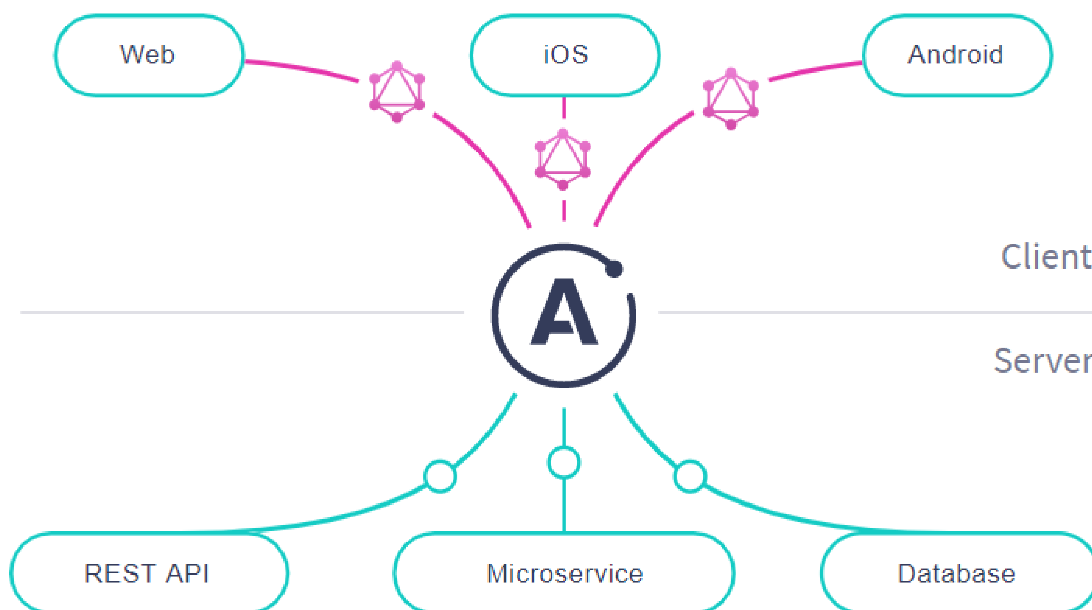
3.2 Apollo GraphQL

Apollo GraphQL je OpenSource knihovna napsaná v jazyce ECMAScript a TypeScript. Je vyvíjena společností Meteor Development Group Inc. a hostuje na adrese www.apollo-graphql.com. Jedná se o jednu z implementaci specifikace GraphQL.

Knihovna je distribuována pomocí balíčkovacího systému NPM a je rozdělena na serverovou (apollo-server, aktuálně 400 tisíc stažení týdně) a klientskou (apollo-client, aktuálně 1,3 milionu stažení týdně) část. Dle počtu stažení lze potvrdit, že se jedná o velmi rozšířenou a dobře otestovanou implementaci GraphQL.

Tato technologie je zvolena především kvůli následujícím výhodám:

- velmi rozšířená
- multiplatformní viz. *Obr. 11*
- rozsáhlá komunita
- na server lze napojit jakéhokoliv klienta dodržující specifikaci GraphQL
- jednoduchá implementace TLS pomocí knihovny https
- jako datový zdroj lze napojit jakoukoliv databázi, mikroslužbu, nebo další API
- podpora websocketů.



Obr. 11. Schéma fungování Apollo GraphQL serveru [16].

Při implementaci API je použit balíček apollo-server, který poskytuje OpenSource GraphQL server.

```
const { ApolloServer, gql } = require('apollo-server');

// The GraphQL schema
const typeDefs = gql`
  type Query {
    "A simple type for getting started!"
    hello: String
  }
`;

// A map of functions which return data for the schema.
const resolvers = {
  Query: {
    hello: () => 'world',
  },
};

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

Výpis 9. Sestavení GraphQL serveru pomocí knihovny apollo-server [16].

Výpis 9 obsahuje zjednodušenou ukázku kompletního Apollo GraphQL serveru. Proměnná typeDefs obsahuje jednoduché schéma s jedním Query, proměnná resolvers obsahuje jednu resolverovou funkci pro odbavení definované Query. Poté je zkonstruován server a na tomto serveru je zavolána metoda listen(). Po zavolání této metody server začne naslouchat na defaultním portu.

3.3 JWT

Z analýzy technologií pro autentizaci je vybrána technologie JWT z důvodu její jednoduchosti a pro daný případ zcela dostačující funkcionality. Dále z důvodu vysoké bezpečnosti a efektivní implementace. Do tokenu JWT je pomocí algoritmu RS256 zakódován a digitálně podepsán payload ve formě uživatelského ID, času vystavení a času expirace. Technologie JTW je rozebrána v # 1.5.7 JWT .

3.4 Databáze Redis

Redis je open source (pod licencí BSD) úložiště datových struktur v operační paměti, používané jako databáze, mezipaměť a zprostředkovatel zpráv. Podporuje datové struktury, jako jsou řetězce, hashe, seznamy, množiny, tříděné množiny s dotazy na rozsah, bitmapy, hyperlogy, geoprostorové indexy s dotazy na poloměry. Redis má vestavěnou replikaci, Lua skriptování, transakce a různé úrovně persistence na disku a poskytuje vysokou dostupnost prostřednictvím služby Redis Sentinel [18].

Obr. 12. Logo databáze Redis. [18]

Tato databáze je při implementaci aplikačního rozhraní použita pro uložení úrovně oprávnění uživatelů.

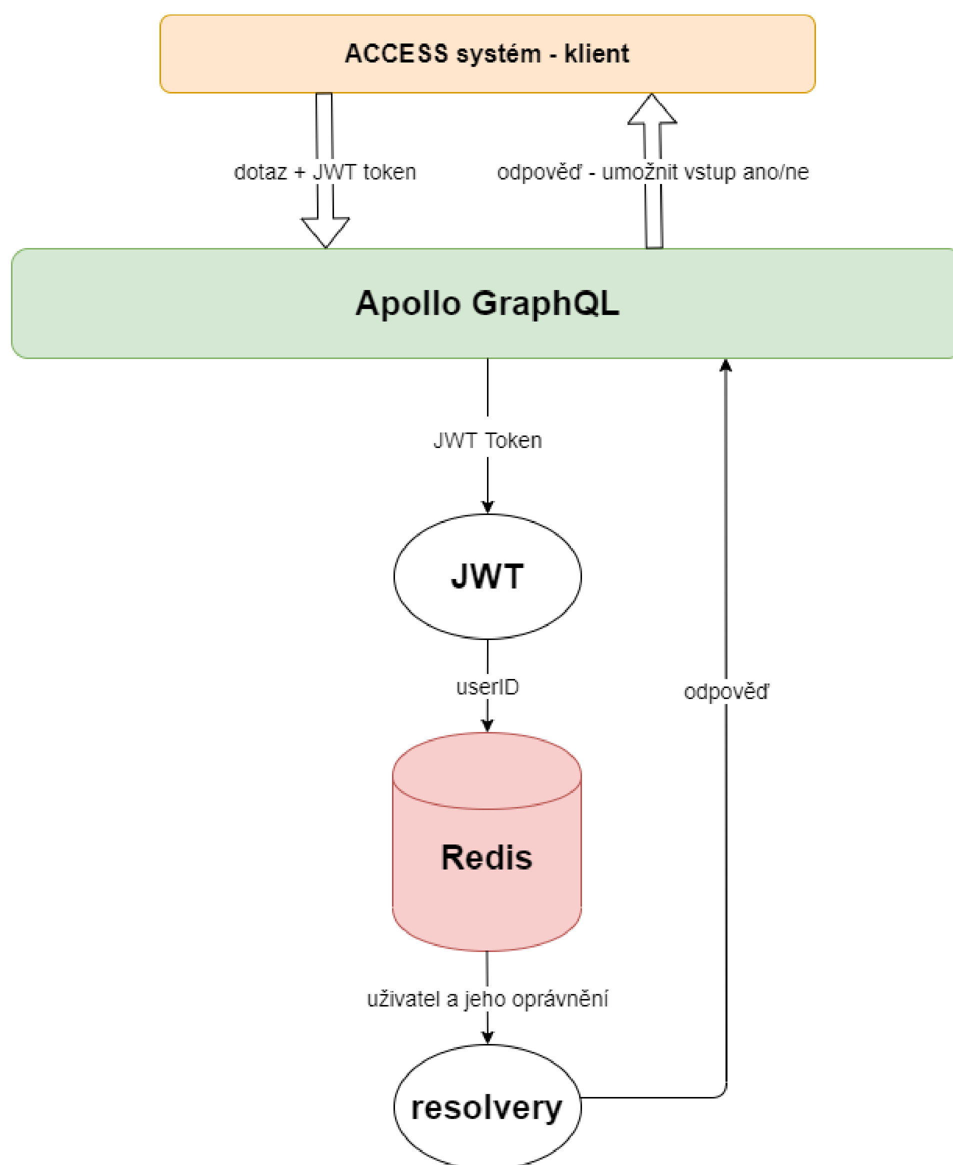
II. PRAKTICKÁ ČÁST

4 NÁVRH ŘEŠENÍ

Jako řešení uvažovaného aplikačního rozhraní je zvolen GraphQL server, který bude zabezpečen pomocí TLS. K autentizaci budou použity JWT tokeny, k autorizaci bude použita databáze Redis. Autorizace bude probíhat na úrovni jednotlivých resolverových funkcí.

4.1 Struktura ACCESS systému

Navrhovaný ACCESS systém obsahuje sadu oblastí, které mají definované konkrétní časy, ve kterých je možné do prostoru přistupovat. Přístup v úsecích mimo tyto konkrétní pásma musí být odepřen. Každá oblast vyžaduje minimální úroveň oprávnění uživatele požadovanou k umožnění vstupu. Blokové schéma tohoto ACCESS systému je zobrazeno na *Obr. 13*.



Obr. 13. Diagram aplikačního rozhraní.

4.1.1 Úrovně oprávnění

Úrovně oprávnění jsou rozděleny do tří skupin, viz. *Výpis 10*. CEO, plný přístup a návštěvník. Uživatelům tedy mohou být přiděleny následující úrovně oprávnění:

```
const accessLevels = [
  { level: 0, name: 'CEO' },
  { level: 1, name: 'fullAccess' },
  { level: 2, name: 'visitor' },
];
```

Výpis 10. Úrovně oprávnění uživatelů.

4.1.2 Sada oblastí

Jako oblasti jsou ve *Výpis 11* definovány tři oblasti s různou minimální úrovní oprávnění potřebnou pro přístup.

```
const areas = [
  {
    name: 'Laboratoř',
    accessLevel: accessLevels[2],
    grantedHours: [
      { from: '08:00', to: '12:00' },
      { from: '13:00', to: '18:00' },
    ],
  },
  {
    name: 'Jednací místnost',
    accessLevel: accessLevels[2],
    grantedHours: [{ from: '06:00', to: '22:00' }],
  },
  {
    name: 'Toaleta',
    accessLevel: accessLevels[3],
    grantedHours: [{ from: '00:00', to: '24:00' }],
  },
];
```

Výpis 11. Sada oblastí ACCESS systému.

4.1.3 Sada testovacích uživatelů

Aby bylo možné rozhraní řádně otestovat, je ve *Výpis 12* vytvořena následující sada testovacích uživatelů.

```
const users = [
  {
    userID: '5f2034519c9263e3d8a8191f',
    name: 'Petr Novák',
    accessLevel: accessLevels[0],
  },
  {
    userID: objectID(),
    name: 'Pavel Novotný',
    accessLevel: accessLevels[1],
  },
  {
    userID: objectID(),
    name: 'František Kuba',
    accessLevel: accessLevels[2],
  },
];
```

Výpis 12. Sada testovacích uživatelů.

Každý uživatel je identifikovaný pomocí jedinečného ID ve formě ObjectID. Jedná se o unikátní ID s délkou 12 bajtů, například:

```
ObjectId("54759eb3c090d83494e2d804")
```

5 IMPLEMENTACE API

V této části je zvažované API implementováno dle bodů 3 a 4 této práce.

5.1 Inicializace nového npm projektu

V prvním kroku je založen nový npm projekt následujícím příkazem:

```
$ npm init
```

Tento příkaz slouží k vygenerování souboru package.json, který obsahuje všechny závislosti a skripty. Obsah takto vygenerovaného souboru je zobrazen ve *Výpis 13*.

```
{
  "name": "api",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Výpis 13. Soubor package.json nově založeného projektu.

5.2 Funkce pro obsluhu JWT tokenů

Tyto funkce mají za úkol generování a následné ověřování JWT tokenů. Jedná se o primární funkcionalitu autentizace aplikačního rozhraní, protože tokeny vygenerované pomocí těchto funkcí slouží ke ztotožnění daného uživatele. Pro práci s JWT je zvolena knihovna jsonwebtoken z ekosystému NPM, která je distribuována pod OpenSource licencí MIT. Jedná se o přímou implementaci standardu RFC7519.

Stažení a instalaci knihovny jsonwebtoken pomocí balíčkovacího systému npm je proveden následujícím příkazem:

```
$ npm install jsonwebtoken
```

Tato knihovna obsahuje funkce pro práci s JWT tokeny, především funkce pro podepsání a ověření tokenu.

Tyto funkce obsahuje soubor jwt.js, který se nachází v adresáři /src/usr/, funkce jsou z tohoto souboru exportované pomocí systému module.exports.

Vygenerování páru klíčů RS256

V podepisování a ověřování JWT tokenů je použit algoritmus RS256. Jedná se o asymetrický algoritmus a je tedy nutné vygenerovat pár privátního a veřejného klíče.

K vygenerování páru klíčů je použita rozšířená OpenSource knihovna kryptografických funkcí OpenSSH viz. *Výpis 14*, konkrétně její utilita ssh-keygen. Vygenerované klíče jsou uloženy do souboru jwtRS256.key, v případě produkčního nasazení je tento klíč nutné uložit na bezpečné místo.

```
$ ssh-keygen -t rsa -b 4096 -m PEM -f
jwtRS256.key

Generating public/private rsa key pair.

Enter passphrase (empty for no passphrase):
Enter same passphrase again:

Your identification has been saved in
jwtRS256.key

Your public key has been saved in
jwtRS256.key.pub

The key fingerprint is:
SHA256:ckq0wSTLSVpJDGhg65I11pW5cYGPVKkvTD3Y
sIcH7DM

The key's randomart image is:

+---[RSA 4096]----+
|oo.+=oo=oo       |
|o..*oBX o        |
|..= =o+/         |
|.+ . .EoB        |
|o.   o+*S.       |
|.    .o+.        |
|      ..         |
|                 |
|                 |

+----[SHA256]-----+
```

Výpis 14. Generování RSA klíčů pomocí utility ssh-keygen.

Tímto krokem byl vygenerován pár veřejného a privátního klíče. Ukázka privátního klíče je zobrazena ve *Výpis 15*.

```
-----BEGIN RSA PRIVATE KEY-----
MIIJKAIBAAKCAgEAvKUEq1p6WHbCH49sDXBuqUTiz0l22+kqCpZtlVU1hie8B5YB
nAheT+z9Rv4o/QkAg8hvSvs9M5bCwGEij5PpQXMP4TQ4Cuvf3ZEajwRPz2rhtCsM
oh4XwY4Ytb+LliRiTuV5SuxXui+2nS4e9WSh0b9te8xlhhbBY++IEzXQGYFBGPfe
aimqtiR36Ute4wF/ZSoLKj/5ixdmwRUTfB2nN6rbnERpvZjswHqmeP3wMR+Ci/j1
q+ksi3rZcSbCLUC2qcTbLYSJuU7c2BHbzsQ2pVP+LyQ5pUslnwVHFEilsZUK/5sm
PThfPfd6BbnY0h5OmqR3x+TzpDZtVIFLdVffZzlN0xwB4x+ewsF/9xzOPcjlfV0S
9kO11Q0juQrLoOz+Cju7Xkga44xxsO8K7p+x1AU3yR+k+sz9PbsoTZgKDaKiKiRR
h3F7HKjQ6LAiXmu47diqFGDdmvrPHls0Bs5ncX9kwaRYQLAZneU66zd17EAv+9E/
rm4870sdkkfSuIYBQMpBDDQFuWYSj2xrJB3yNzKEZOmsL17F33OWSbOe1NY=
-----END RSA PRIVATE KEY-----
```


Výpis 15. Část privátního RSA klíče.

Pomocí OpenSource knihovny kryptografických funkcí OpenSSL ve *Výpis* 16, konkrétně utility openssl je z privátního klíče odvozen veřejný klíč ve formátu PEM. Tento klíč je uložen do souboru jwtR256.key.pub.

```
$ openssl rsa -in jwtRS256.key -pubout -outform PEM -out jwtRS256.key.pub
```

Výpis 16. Odvození veřejného klíče PEM z privátního klíče RSA.

Výstupem je vygenerovaný veřejný klíč, viz. *Výpis* 17. Tento klíč je použit ke kontrole autenticity JWT tokenů.


```
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAvKUEq1p6WHbCH49sDXBu
qUTiz0l22+kqCpZtlVU1hie8B5YBnAheT+z9Rv4o/QkAg8hvSvs9M5bCwGEij5Pp
QXMP4TQ4Cuvf3ZEajwRPz2rhtCsMoh4XwY4Ytb+LliRiTuV5SuxXui+2nS4e9WSh
0b9te8xlhhbBY++IEzXQGYFBGPfeaimqtiR36Ute4wF/ZSoLKj/5ixdmwRUTfB2n
N6rbnERpvZjswHqmeP3wMR+Ci/j1L9ulBielk/g98zbLIsUcdp6tlCiGwIqolL9g
A7gOpqfWBCCxNlslw7qY2fWdwJDz4uxJp7yocpXPwCS+r9WgTM4jyvMkIhoiRApN
I2466DiaUDQH7e5OVq4R2TLbntZ1rE6FY5TKAo45Cst9tYOBMKblSrXT1oA1L7Hd
bgEN+zg9mToeFTHN1B1dlBlwcDqfig+xC8OzcdCPlAYvcHgy0Etybm7JocswJzOr
rrvu2drRIujlPjAKoSPI8qGHoEHxqk3w6ywaP6S+3FaDfeQaOMambVb6bYOWaoH9
uIkMTqzZ8Ip0r8SJIe1a4AZNgsWitruniLpC+S6wB84+0u1v8gPVhrHbldVDDDNw
Hbf06eKti9RtZVQ9kdEIfDckPBUPkygsfP6YltOcW79omwWJnQvW63kE8ZmDc9w8
CF1tsd7Vx0mosoULGYQF2eMCAwEAAQ==
-----END PUBLIC KEY-----
```


Výpis 17. Vygenerovaný veřejný RSA klíč.

5.2.1 Import klíčů a samotné knihovny

Import samotné knihovny je proveden pomocí funkce require() ve *Výpis 18*, jsou také načteny privátní a veřejný klíč. V případě produkčního nasazení by tento klíč byl uložen v prostředí určenému pro správu secrets a jeho načtení by tedy probíhalo odlišně. Pro příklad ukázky je postačující textový soubor PEM na disku.

```
const jwt = require('jsonwebtoken');
const fs = require('fs');

const privateKey = fs.readFileSync('./jwtRS256.key');
const pubKey = fs.readFileSync('./jwtRS256.key.pub');
```

Výpis 18. Soubor jwt.js - import knihovny jsonwebtoken a načtení privátního klíče z disku.

5.2.2 Vytváření nových tokenů

Za pomocí jsonwebtoken je vytvořena funkce pro generování nových tokenů createToken() ve *Výpis 19*. Tato funkce bere jako vstupní parametr uživatelské ID ve formátu ObjectID a vrací Javascript Promise – příslib budoucí hodnoty. Obsahuje kontrolu validity ID a v případě neúspěchu provede reject(). V případě úspěchu pomocí metody resolve() vrací vygenerovaný JWT token.

Knihovna jsonwebtoken je pomocí konfiguračního objektu nakonfigurována tak, aby byl použit algoritmus RS256 a platnost tokenu byla nastavena na 365 dní.

```
// function createToken creates signed token with userID as payload
const createToken = ({ userID }) => {
  return new Promise((resolve, reject) => {
    // check an userID length
    if (userID.length !== 24) {
      reject('Length of userID is not correct!');
    }

    // sign and return a token
    jwt.sign(
      {
        userID,
      },
      privateKey,
      { expiresIn: '365d', algorithm: 'RS256' },
      (err, token) => {
        // reject error
        if (err) reject(err);

        // resolve signed token
        resolve(token);
      },
    );
  });
};
```

Výpis 19. Soubor jwt.js – funkce pro vytváření JWT tokenu.

Tato funkce je otestována jejím zavoláním:

```
const { createToken } = require('./src/user/jwt');

// generate random objectID
const objectID = (
  m = Math,
  d = Date,
  h = 16,
  s = (s) => m.floor(s).toString(h),
) => s(d.now() / 1000) + ' '.repeat(h).replace(/./g, () => s(m.ran
dom() * h));

// create a token and log to the console
createToken({ userID: objectID() })
  .then((token) => {
    generatedToken = token;
    console.log(generatedToken);
  })
  .catch((err) => {
    console.log(err);
  });
```

Výpis 20. Otestování funkčnosti vytváření tokenů.

Aby bylo možné funkci createToken zavolat, je nejprve nutné vytvořit náhodné ID, k tomu slouží funkce objectID() ve *Výpis 20*.

Funkce createToken je asynchronní Javascriptová funkce a lze tedy využít metod then() a catch(), pomocí kterých lze efektivně zajistit řízení případných chybových stavů.

Výstupem této zavolání této funkce je dle předpokladu podepsaný JWT token, viz. *Výpis* 21.


```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySUQiOiI1ZjE0NjJlNTYwM
jk2N2ZmNmU1YzZiMjEiLCJpYXQiOjE1OTUxNzE1NTcsImV4cCI6MTYyNjcwNzU1N30.
d6HpDsjsLlQTcXkmx5jHOXsONlSwo0GBmUaAO-gD3bKUC9EPIS1GRzQpNXwA1z1IAmd
rHqwLApNmD-K-S0eIAzJjASebX7g77hXK-2F99LLAzm3WnviudIjCEdNAHvhh5O8oMZ
FDpUjcPCGzFfPOYnloHE9wYA7y3tBaE_0qwXxh1xrC5NinfxCnOTUxNzE1NTcsImV4c
CI6MTYyNjcwNzU1N30.d6HpDsjsLlQTcXkmx5jHOXsONlSwo0GBmUaAO-gD3bKUC9EP
IS1GRzQpNXwA1Kw8uDL_30vO2vJt4S-oA7iOwGdgkLU7JOVI_iWr3Jt19UmAnOitTKc
G950tqqLvSSbTav8CtR2iNhTZHpRSCw1osYsBK_LzaDUfM0VhzQVl8JT6r4X5p0T-Dl
ruo-R6yUpN_8r4LmkcKVFsFBMSdWA_Rnsxk7PJHwy2Z5an7WDzg1WqdD1M7Z06VAYTA
9dt2gjoW-VUIaOxKNkRGXH1hbLq3qxphz1IAmdrHqwLApNmD-K-S0eIAzJjASebX7g7
7hXK-2F99LLAzm3WnviudIjCEdNAHvhh5O8oMZFDpUjcPCGzFfPOYnloHE9wYAS-NNj
wYwFxEnUzXvRpLaAxc8PPCYO-dWTufKFY_i6RVNO3qYroFBMtWshpdAK-lUiPPYqMMr
umuWMXT1wYxNA7y3tBaE_0qwXxh1xrC5NinfxCn_Qbz0Nu9iomQRfqhTo9RSAuddEgC
nzAaMxxI0OxBjHD-R6unuvgkwJB8cyaTMCa2Dp3N0i-gb1NkBDToJazOSiIGVy5qyn7
dvKw8uDL_30vO2vJt4S-oA7iOwGdgkLU7JOVI_iWr3Jt19UmAnOitTKcG950tqqLvSS
bTav8CtR2iNhTZHpRSCw1osYsBK_LzaDUfM0VhzQVl8JT6r4X5p0T-Dlruo-R6yUpN_
8r4LmkcKVFsFBMSdWA_Rnsxk7PJHwy2Z5an7WDzg1WqdD1M7Z06VAYTA9dt2gjoW-VU
IaOxKNkRGXH1hbLq3qxph7AjlrSmcvDMPlg6WIioLjE9ShdCi63wsVGLxbfEUEplRxQ
BnSacuNGmqf4NFwhiuxj8e7k7Esn7RGIRiGz_YReY6f5INyhVi_S-NNjwYwFxEnUzXv
RpLaAxc8PPCYO-dWTufKFY_i6RVNO3qYroFBMtWshpdAK-lUiPPYqMMrumuWMXT1wYx
NA
```


Výpis 21. Vygenerovaný JWT token, který je podepsaný pomocí privátního klíče.

5.2.3 Kontrola tokenu pomocí online nástroje jwt.io

Přímo domovská stránka JWT tokenů nabízí přehledný online nástroj pro práci s JWT tokeny. Tento nástroj je použit ke kontrole správnosti vygenerovaného tokenu na *Obr. 14*.

Algorithm RS256

Encoded PASTE A TOKEN HERE

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
1c2VySUQiOiI1ZjE0NjJlNTYwMjk2N2ZmNmU1YzZ
iMjEiLCJpYXQiOjE1OTUxNzE1NTcsImV4cCI6MTY
yNjcwNzU1N30.d6HpDsjsL1QTcXkmx5jHOXsON1S
wo0GBmUaAO-
gD3bKUC9EPIS1GRzQpNXwA1z1IAmdrHqwLApNmD-
K-S0eIAzJjASebX7g77hXK-
2F99LLAzm3WnviudIjCEdNAHvhh5O8oMZFDpUjcP
CGzFfPOYnloHE9wYA7y3tBaE_0qwXxh1xrC5Ninf
xCn_Qbz0Nu9iomQRfqhTo9RSAuddEgCnzAaMxxI0
OxBjHD-R6unuvgkwJB8cyaTMCa2Dp3N0i-
gb1NkBDToJazOSiIGVy5qyn7dvKw8uDL_30vO2vJ
t4S-
oA7iOwGdgkLU7JOVI_iWr3Jt19UmAnOitTKcG950
tqqLvSSbTav8CtR2iNhTZHpRSCw1osYsBK_LzaDU
fM0VhzQVl8JT6r4X5p0T-Dlruo-
R6yUpN_8r4LmkcKVFsFBMSdWA_Rnsxk7PJHwy2Z5
an7WDzg1WqdD1M7Z06VAYTA9dt2gjoW-
VUIaOxKNkRGXH1hbLq3qxph7AjlrSmcvDMPlg6WI
ioLjE9ShdCi63wsVGLxbfEUEplRxQBnSacuNGmqf
4NFwhiuxj8e7k7Esn7RGIRiGz_YReY6f5INyhVi_
S-NNjwYwFxEnUzXvRpLaAxc8PPCYO-
dWTufKFY_i6RVNO3qYroFBMtWshpdAK-
lUiPPYqMMrumuWMXT1wYxNA

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "userID": "5f1462e5602967ff6e5c6b21",
  "iat": 1595171557,
  "exp": 1626707557
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  -----BEGIN PUBLIC KEY-----
  MIICIjANBgkqhkiG9w0BAQEFAAOC
  Ag8AMIICCgKCAgEAvKUEq1p6WHbC
  H49sDXBu
  ,
  Private Key. Enter it in plain
  text only if you want to gener
  ate a new token. The key never
  leaves your browser.
)
```

Signature Verified

SHARE JWT

Obr. 14. Kontrola tokenu pomocí jwt.io a veřejného klíče.

Z výše uvedeného obrázku plyne, že se jedná o platný JWT token zabezpečený pomocí RS256 algoritmu obsahující payload zobrazený ve *Výpis* 22.

```
{
  "userID": "5f1462e5602967ff6e5c6b21",
  "iat": 1595171557,
  "exp": 1626707557
}
```

Výpis 22. Payload vygenerovaného tokenu.

Payload tokenu obsahuje vygenerované userID a dále dva parametry, které byly přidány přímo knihovnou jsonwebtoken:

1. iat, čas vystavení tokenu ve formátu Epoch Unix Time Stamp
2. exp, čas expirace tokenu ve formátu Epoch Unix Time Stamp.

Tyto dva přídavné parametry jsou následně použity ke kontrole validity tokenu.

5.2.4 Funkce pro kontrolu validity tokenu v rámci řetězce autentizace API

Tato funkce má za úkol ověřit platnost tokenu, podpis a jako návratovou hodnotu vrátit Promise, viz. *Výpis* 23. V případě úspěchu je tato Promise vyřešena navrácením dekódovaného uživatelského ID, což znamená faktickou autentizaci. V případě neúspěchu je vyvolán chybový stav funkcí reject(), což vede k neúspěšné autentizaci. Funkce bude volána při každém požadavku zaslaném na aplikační rozhraní.

```
// function getuserID verify tokens and returns userID if valid
const getuserID = ({ token }) => {
  return new Promise((resolve, reject) => {
    jwt.verify(token, pubKey, (err, decoded) => {
      // token is not valid, return false
      if (err) reject(err);

      // token is valid, return userID
      resolve(decoded.userID);
    });
  });
};
```

Výpis 23. Soubor jwt.js – autentizace ověřením tokenu.

Export těchto funkcí je proveden tak, aby bylo možné je libovolně používat v rámci celého balíčku. Toho je docíleno pomocí konceptu module.exports, viz. *Výpis* 24.

```
module.exports = { createToken, getUserID };
```

Výpis 24. Export funkcí pro práci s JWT.

5.3 Identifikace uživatele a načtení úrovně jeho práv z databáze Redis

Funkce z bodu 5.1.2 Funkce pro obsluhu JWT tokenů umožňuje pomocí JWT autentizovat uživatele a získat jeho userID. Ke každému userID je přiřazen stupeň oprávnění, který reprezentuje jedna z hodnot pole definovaného v bodě 4.1.1 Úrovně oprávnění.

Tato funkcionalita je realizována uložením této hodnoty do 3.4 Databáze Redis .

5.3.1 Získání databáze Redis

Projekt databáze Redis je přímo spojen se společností Redis Labs, která nabízí cloudovou službu Redis Enterprise: Ultimátní Redis zkušenost. Pomocí této služby lze spustit managovanou Redis databázi doslova během několik minut viz. *Obr. 15*.

Create Database

Database Name	server
Subscription	#1186252 Essentials/AWS/eu-west-1/Standard/30MB
Protocol	Redis
Replication	Disabled
Data Persistence	None
Access Control & Security	Redis Password l0tE9e6nLR6Fn3ISuxJRKtaHVnMCWcah
Data Eviction Policy	volatile-lru
Alert Settings	Total size of datasets under this plan has reached 80 % of plan limit
	Number of connections has reached 80 % of plan limit

Cancel Activate

Obr. 15. Vytvoření databáze pomocí služby Redis Labs.

Tato služba nabízí několik typů předplatného, kdy základní 30 MB databáze je zdarma viz. *Obr.* 16. Tato základní varianta je zvolena pro účely této práce. V případě 100 MB plánu lze získat managovanou 100 MB databázi včetně metrik, persistence a záloh za poplatek 8 USD měsíčně.

Standard plans offer a rich set of features, including in-memory replication, auto-failover, data persistence and backups.

Plan	Price
30MB	free
100MB	$8/mo
250MB	$20/mo
500MB	$41/mo
1GB	$80/mo
2.5GB	$194/mo
5GB	$369/mo
Pay-As-You-Go	$369/mo+usage*

Feature	
Memory size	100MB
Infinite auto-scalability	
Multi-core Redis	
Replication	✓
Auto-failover	✓
Data persistence	✓
Daily and instant backups	✓
Dedicated databases	4
Connections	256
Security Groups / Source IP authentication rules	1/4
24/7 toll-free support hotline	✓

Obr. 16. Datové a cenové plány služby Redis Enterprice.

Konfigurace databáze

Pro potřeby konfigurace je vytvořen soubor config.js v adresáři /src. Do tohoto souboru jsou překopírované přihlašovací údaje získané ze služby Redis Labs, viz. *Výpis* 25. V případě produkčního nasazení je nutné heslo uložit na místo k tomu určené – secrets.

```
const redisCnf = {
  port: 13807,
  url: 'redis-13807.c233.eu-west-1-1.ec2.cloud.redislabs.com',
  password: 'l0tE9e6nLR6Fn3ISuxJRKtaHVnMCWcah',
};

module.exports = { redisCnf, apolloCnf };
```

Výpis 25. Soubor config.js obsahující konfiguraci databáze Redis.

5.3.2 JS knihovna Node Redis

JS knihovna Node Redis je high performance Redis klient, jedná se o OpenSource balíček distribuovaný pomocí balíčkovacího systému NPM. Knihovna mapuje nativní Redis příkazy do jazyka JavaScript 1:1. Všechny Redis příkazy jsou volány asynchronně a je tedy nutné využití callback funkcí.

```
const redis = require("redis");
const client = redis.createClient();

client.on("error", function(error) {
  console.error(error);
});

client.set("key", "value", redis.print);
client.get("key", redis.print);
```

Výpis 26. Ukázka použití knihovny Redis [19].

5.3.3 Funkce umožňující autorizaci požadavků

Tyto funkce jsou exportovány ze souboru auth.js v adresáři /src/user. V rámci uzávěry exportu bude vytvořen a připojen Redis klient, jak je patrno ve *Výpis* 27. Tento klient bude k dispozici v rámci daného exportu.

```
const { getuserID } = require('./jwt');
const { AuthenticationError } = require('apollo-server');
const redis = require('redis');
const { redisCnf } = require('../config');

const client = redis.createClient(redisCnf.port, redisCnf.url);
client.auth(redisCnf.password);

client.on('connect', function (err, res) {
  console.log('redis is connected!');
});
```

Výpis 27. Soubor auth.js – připojení Redis klienta a importy potřebných funkcí.

Následující funkce getUser() má jako vstupní parametr JWT token a jako návratová hodnota je příslib budoucí hodnoty objektu uživatele a jeho úrovně oprávnění. Pro získání této informace je nutno procesovat 2 kroky:

1. Dekódování JWT tokenu a ověření jeho platností
2. Dotázání Redis databáze na úroveň oprávnění daného uživatele.

Protože jak získání uživatelského ID, tak dotazování databáze je asynchronní úloha, je využita koncepce try – catch, viz. *Výpis* 28. Pokud kterákoliv volaná funkce vyvolá chybový stav, dojde k selhání celé autentizace a spuštění bloku catch.

```
// získání a ověření uživatele z databáze, nebo vyvolání chyby
const getUser = async ({ token }) => {
  try {
    const userID = await getuserID({ token });

    const user = await new Promise((resolve) => {
      client.hget(`user`, userID, function (getError, result) {
        if (getError) throw getError;
        resolve(result);
      });
    });

    // navrácení objektu obsahujícího údaje o uživateli
    return { user: JSON.parse(user) };
  } catch (error) {
    throw new AuthenticationError(error);
  }
};
```

Výpis 28. Soubor auth.js - získání úrovně oprávnění z databáze Redis.

5.4 Implementace GraphQL serveru

Server GraphQL se skládá z několika částí, především je nutné vytvořit jeho statické datové schéma a resolverové funkce. Poté je možné sestavit server a připojit middleware – mezivrstvy. Touto mezivrstvou je právě autentizační funkce.

5.4.1 Schéma GraphQL

Pro účely datového schématu je vytvořen soubor schema.js v adresáři /src/schema.

```
const { gql } = require('apollo-server');

// Schema obsahuje kompletní datastrukturu serveru
const typeDefs = gql`
  # "Query" obsahuje všechny spustitelné dotazy
  type Query {
    getAccess(areaID: ID!): Boolean!
    getUsers: [User]
    getToken(userID: ID!): String!
    getAreas: [Area]
  }

  # "Mutation" obsahuje všechny spustitelné mutace
  type Mutation {
    addUser(user: UserInput): User!
    addArea(area: AreaInput): Area!
    blockUser(userID: ID!): User
  }
`;
module.exports = typeDefs;
```

Výpis 29. Soubor schema.js - datové schéma GraphQL serveru.

API má následující Query funkcionality:

1. dotaz ACCESS systému na povolení přístupu do oblasti – getAccess
2. výpis všech uživatelů – getUsers
3. žádost o vygenerování nového tokenu – getToken
4. výpis všech oblastí ACCESS systému – getAreas.

Modifikace dat je obsloužena pomocí mutací:

1. přidání nového uživatele – addUser
2. přidání nové oblasti – addArea
3. zablokování existujícího uživatele – blockUser.

Všechny tyto funkcionality jsou znázorněny ve *Výpis* 29.

Vstupní datatypy:

Tyto datatypy lze používat v rámci mutací a vytvářet tak nové záznamy v data zdrojích.

Datatyp input UserInput:

Tento datatyp slouží pro potřeby vytvoření nového uživatele.

```
input UserInput {
  name: String
  accessLevel: Int
}
```

Výpis 30. Soubor schema.js – datový typ vstupu nového uživatele.

Datatyp input AreaInput:

Tento datatyp slouží pro potřeby vytvoření nové oblasti.

```
input AreaInput {
  name: String
  accessLevel: Int
  grantedHours: [GrantedHourInput]
}
```

Výpis 31. Soubor schema.js – datový typ vstupu nové oblasti.

Datatyp input GrantedHourInput:

Tento datatyp slouží pro potřeby vytvoření nového časového úseku v rámci datatypu Area-Input. Každá oblast obsahuje pole těchto časových úseků.

```
input GrantedHourInput {
  from: String
  to: String
}
```

Výpis 32. Soubor schema.js – datový typ vstupu nového časového úsek

.

Výstupní datatypy:

Tyto datatypy lze v rámci API dotazovat a získat tak předvídatelnou odpověď.

Datatyp Area:

Tento datatyp obsahuje strukturu dat oblasti, která je uložena v databázi a může být dotazována na přístup.

```
# This "Area" type defines the queryable fields for every area
type Area {
  id: ID!
  name: String
  accessLevel: AcessLevel
  grantedHours: [GrantedHour]
}
```

Výpis 33. Soubor schema.js – datový typ oblasti.

Datatyp GrantedHour:

Tento datatyp obsahuje strukturu dat časového úseku, ve kterém je možné vstupovat do oblasti.

```
type GrantedHour {
  from: String
  to: String
}
```

Výpis 34. Soubor schema.js – datový typ časového úseku.

Datatyp AccessLevel:

Tento datatyp obsahuje úroveň oprávnění uživatele, nebo úroveň nutnou pro přístup do oblasti.

```
  type AccessLevel {
    level: Int
    name: String
  }
```

Výpis 35. Soubor schema.js – datový typ úrovně oprávnění.

Oprávnění může nabývat hodnot znázorněných ve *Výpis* 36.

```
const accessLevels = [
  { level: 0, name: 'CEO' },
  { level: 1, name: 'fullAccess' },
  { level: 2, name: 'visitor' },
  { level: 3, name: 'blocked user' },
];
```

Výpis 36. Soubor config.js – možné hodnoty datatypu oprávnění.

Datatyp User:

Tento datatyp obsahuje strukturu dat reprezentující uživatele.

```
  type User {
    userID: ID!
    name: String
    accessLevel: AcessLevel
  }
```

Výpis 37. Soubor schema.js – datový typ reprezentující uživatele.

Pro demonstrační účely jsou využiti testovací uživatelé znázornění ve *Výpis* 38.

```
const users = [
  {
    userID: '5f2034519c9263e3d8a8191f',
    name: 'Petr Novák',
    accessLevel: accessLevels[0],
  },
  {
    userID: objectID(),
    name: 'Pavel Novotný',
    accessLevel: accessLevels[1],
  },
  {
    userID: objectID(),
    name: 'František Kuba',
    accessLevel: accessLevels[2],
  },
];
```

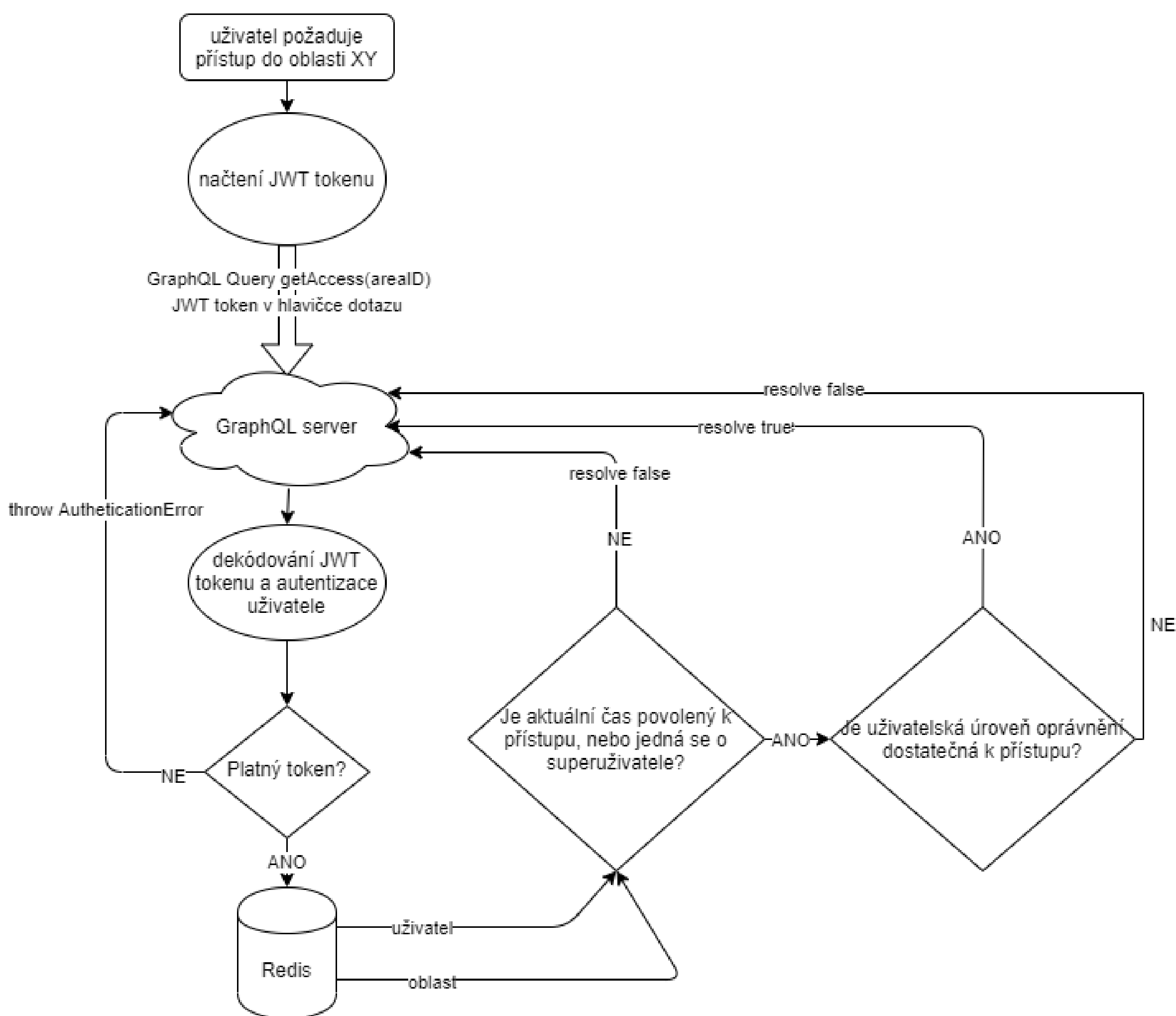
Výpis 38. Sada demonstračních uživatelů.

5.4.2 Resolverové funkce Query

Tyto funkce obsahují kompletní business logiku a napojení do zdroje dat, jsou exportovány ze souboru resolvers.js v adresáři ./src/resolvers.

Resolverová funkce getAccess()

Funkce getAccess je volána v případě požadavku na přístup do oblasti, jedná se tedy o nosnou funkci celého systému. Blokové schéma tohoto volání je znázorněno na *Obr. 17.*



Obr. 17. Blokové schéma resolverové funkce getAccess().

Dle schématu GraphQL tato funkce musí jako parametr přijímat areaID, což je ID požadované oblasti. Toto ID je extrahováno pomocí techniky zvané destructuring a je uloženo do proměnné areaID. Pomocí funkce contextu bylo v případě do proměnné context injektován objekt obsahující autentizovaného uživatele.

Tento context je testován na přítomnost tohoto údaje, viz. *Výpis 39*. V případě nepřítomnosti parametru je vyvolán chybový stav AuthenticationError a poté vyvolán návrat z funkce, což způsobí přerušení volaní resolveru.

```
getAccess: async (parent, { areaID }, context) => {
      return new Promise(async (resolve, reject) => {
        // kontrola oprávnění
        if (!context.user || !context.user.accessLevel) {
          reject(new AuthenticationError('Unauthenticated'));
          return;
        }
      });
}
```

Výpis 39. Soubor resolvers.js – autorizace resolveru getAccess.

V následujícím kroku je připojena databáze Redis a přečtena daná oblast pomocí jejího ID, viz. *Výpis 40*.

```
// připojení Redis db
   const client = redis.createClient(redisCnf.port, redisCnf.url);
   client.auth(redisCnf.password);
   await new Promise((resolve) => client.on('connect', resolve));

   let area = await new Promise((resolve) => {
   client.hget(`area`, areaID, function (getError, result) {
       console.log(getError);
       if (getError) throw getError;
         resolve(result);
       });
   });

   if (!area) {
     reject(new Error('Area not found!'));
     return;
   }
   area = JSON.parse(area);
```

Výpis 40. Soubor resolvers.js – načtení požadované oblasti z databáze Redis.

Pokud není oblast nalezena, je vyvolán chybový stav 'Area not found!' a vyvolán návrat z resolveru, viz. *Výpis 41*. Pokud je oblast v pořádku nalezena, je provedeno rozparsování do JSON objektu.

Dále je testováno, zda má daný uživatel potřebné oprávnění k přístupu, viz. *Výpis 41*.

```
    // kontrola oprávnění
    if (area.accessLevel.level < context.user.accessLevel.level) {
      resolve(false);
    }
```

Výpis 41. Soubor resolvers.js – testování oprávněnosti akce.

Jako poslední krok je testováno, zda je přístup požadován v povolený časový úsek tak, jak je to definované v rámci parametru grantedHours na objektu dané oblasti viz. *Výpis 42*.

```
// kontrola povolených časů
        let permitedTime = false;
        let now = moment();
        area.grantedHours.forEach((x) => {
          let from = moment(x.from, 'HH:mm');
          let to = moment(x.to, 'HH:mm');
          if (now > from && now < to) {
            permitedTime = true;
          }
        });

        // CEO může vstupovat do všech oblastí 24/7
        if (context.user.accessLevel.level === 0) {
          permitedTime = true;
        }

        resolve(permitedTime);
```

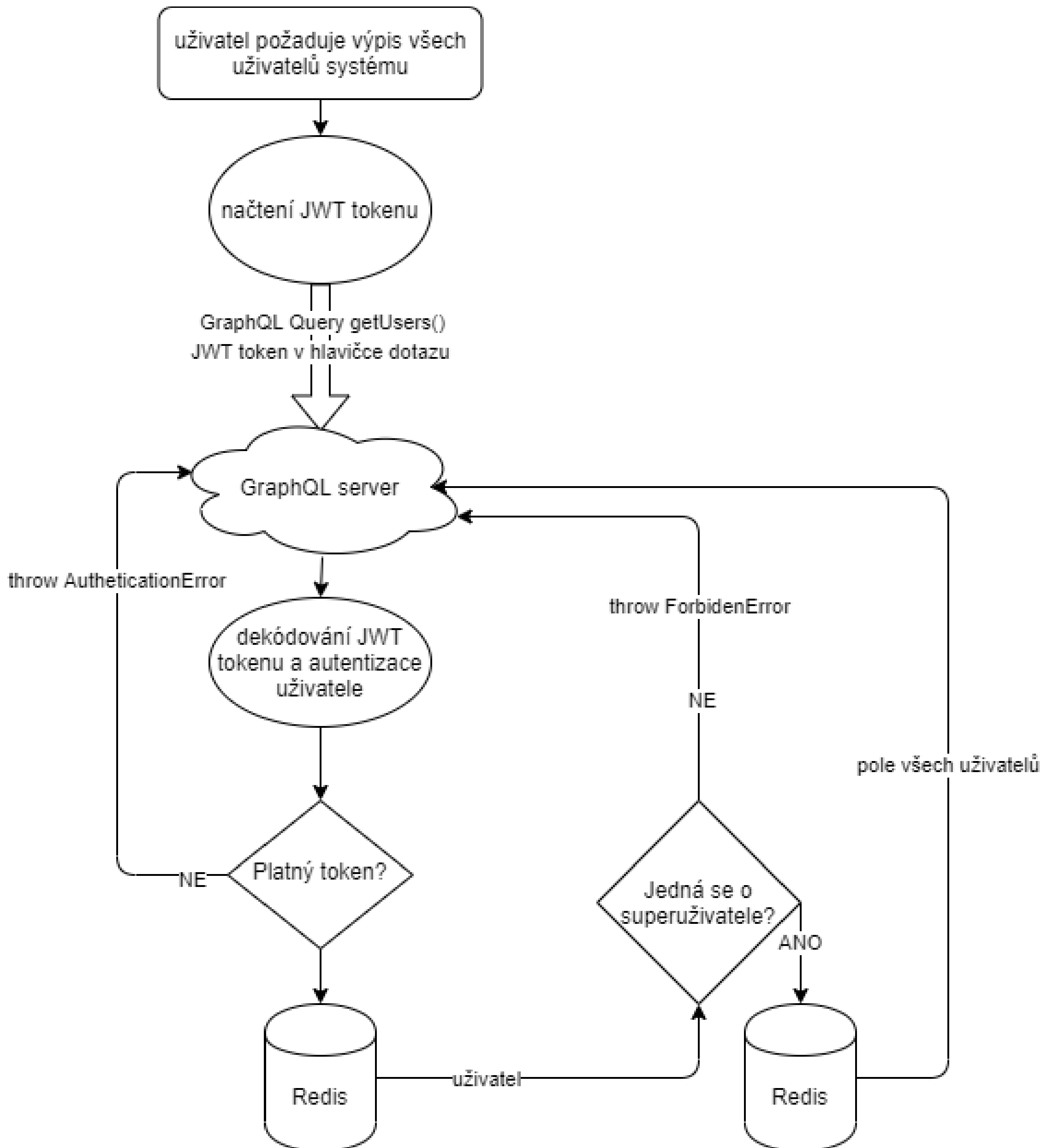
Výpis 42. Soubor resolvers.js – testování časových úseků.

K tomuto účelu je použita knihovna moment, která umožňuje velmi rozsáhlé manipulace s časem v jazyku JavaScript. Proměnná permitedTime je inicializována s hodnotou false a posléze jsou pomocí metody forEach projity všechny prvky pole grantedHours. Z každého času from to je vytvořen objekt moment a ten poté porovnán s aktuálním časem. Je-li

nalezeno časové okno, které obsahuje aktuální čas, je proměnná permitedTime přepsána na hodnotu true. Pokud se jedná a superuživatele s levelem 0, je přístup umožněn bez ohledu na časová okna, proměnná permitedTime je tedy přepsána na hodnotu true. Po zpracování je proměnná permitedTime navrácena jako návratová hodnota pomocí funkce resolve().

Resolverová funkce getUsers()

Tato funkce má za úkol poskytnout výpis všech uživatelů pomocí query getUsers. Pro vykonání tohoto resolveru musí uživatel disponovat oprávněním 0 – superuživatel, jinak dojde k vyvolání autorizačního chybového stavu. Blokové schéma volání tohoto dotazu je znázorněno na *Obr. 18*.



Obr. 18. Blokové schéma resolverové funkce getUsers().

Tato funkce nepřijímá žádné vstupní parametry, protože nevrací specifické uživatele, ale seznam všech uživatelů uložených v databázi.

Funkce je dostupná pouze superuživatelům a proto je v prvním kroku při volání funkce provedena kontrola úrovně oprávnění daného uživatele, viz. *Výpis 43*.

```
getUsers: async (parent, args, context) => {
      return new Promise(async (resolve, reject) => {
        // check for privilegies
        if (
          !context.user ||
          !context.user.accessLevel ||
          context.user.accessLevel.level !== 0
        ) {
          reject(
          new ForbiddenError(`User doesn't have
                                sufficient privileges.`),
          );
          return;
        }
      )
}
```

Výpis 43. Soubor resolvers.js – autorizace resolveru getUsers.

Po autorizace požadavku následuje připojení do databáze Redis a přečtení všech uživatelů, viz. *Výpis 44*.

```
// připojení Redis db
const client = redis.createClient(redisCnf.port,
                                  redisCnf.url);
client.auth(redisCnf.password);
await new Promise((resolve) => {
                  client.on('connect', resolve)
                  });

let users = await new Promise((resolve) => {
  client.hgetall(`user`, function (getError, result) {
    if (getError) throw getError;
    resolve(result);
  });
});
```

Výpis 44. Soubor resolvers.js – získání seznamu uživatelů z databáze.

Databáze Redis pracuje pouze s daty ve formě textových řetězců, viz. *Výpis 45*. Tyto data je tedy nutné deserializovat a vyparsovat do Javascriptového pole.


```
{
  '5f21e743a4a57057b97a55e7': '{"userID":"5f21e743a4a57057b97a55e7
","name":"Pavel Novotný","accessLevel":{"level":1,"name":"fullAcce
ss"}}',
  '5f2034519c9263e3d8a8191f': '{"userID":"5f2034519c9263e3d8a8191f
","name":"Petr Novák","accessLevel":{"level":0,"name":"CEO"}}',
  '5f21e74365c2229017aa62a0': '{"userID":"5f21e74365c2229017aa62a0
","name":"František Kuba","accessLevel":{"level":2,"name":"visitor
"}}'
}
```


Výpis 45. Syrová data získaná z databáze Redis.

Data ve formě objektu jsou namapována do jednotlivých uživatelů a vyparsována do formátu JSON, viz. *Výpis 46*.

```
        let usersArray = [];
        Object.keys(users).forEach((key) => {
          usersArray.push(JSON.parse(users[key]));
        });

        resolve(usersArray);
```

Výpis 46. Soubor resolvers.js – namapování uživatelů do pole jakožto návratové hodnoty.

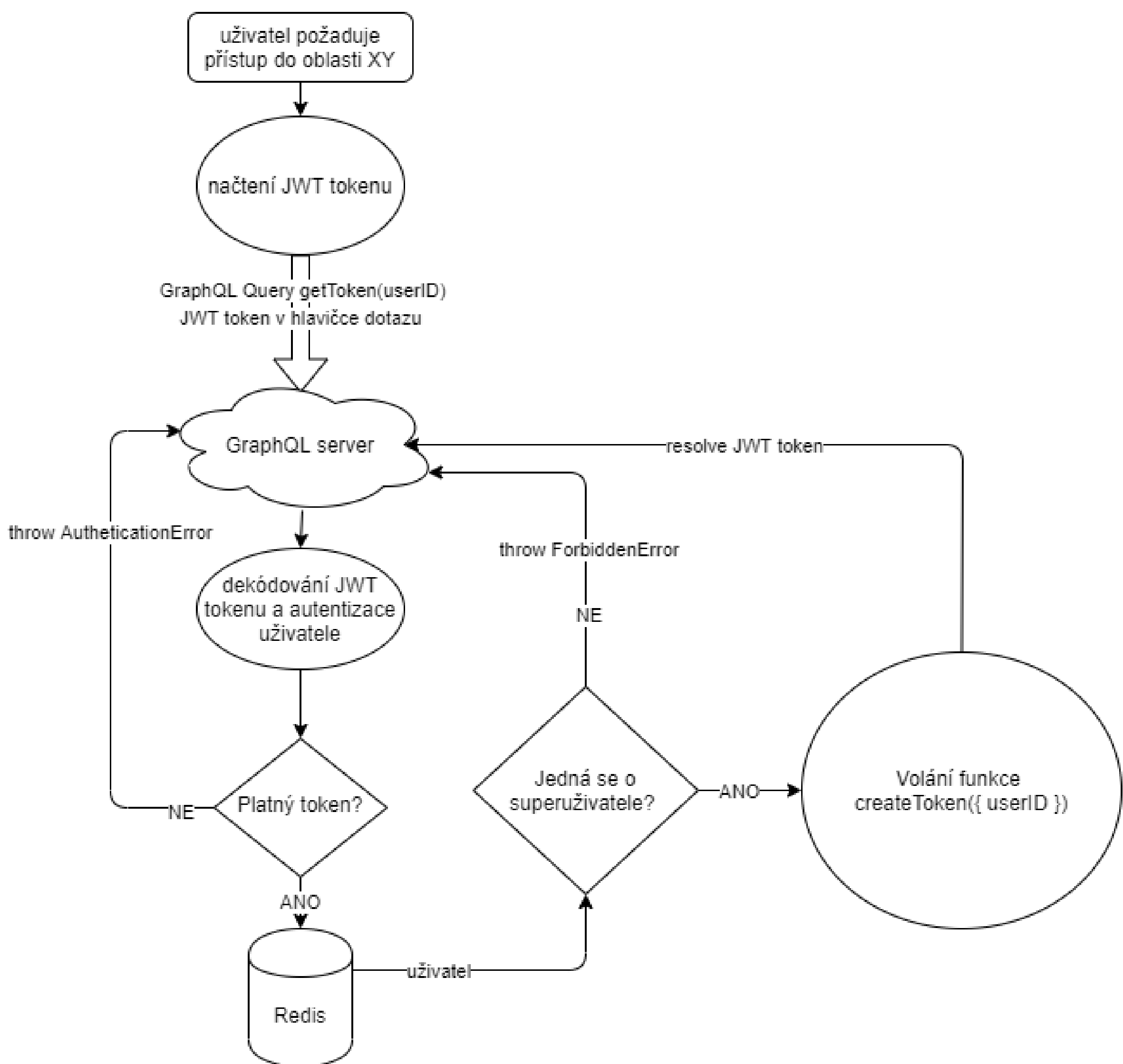
Výsledné pole je vráceno jako návratová hodnota resolveru pomocí funkce resolve(). Ukázka takto zpracovaných dat je zobrazena ve *Výpis 47*.

```
[
  {
    userID: '5f2034519c9263e3d8a8191f',
    name: 'Petr Novák',
    accessLevel: { level: 0, name: 'CEO' }
  },
  {
    userID: '5f21e8792aa21ae0b82a5706',
    name: 'František Kuba',
    accessLevel: { level: 2, name: 'visitor' }
  },
  {
    userID: '5f21e8792e1ba79834aa8a56',
    name: 'Pavel Novotný',
    accessLevel: { level: 1, name: 'fullAccess' }
  }
]
```

Výpis 47. Data zpracována do pole, které je předáno GraphQL jako návratová hodnota.

Resolverová funkce getToken()

Tato funkce slouží pro vygenerování přihlašovacího JWT tokenu pro libovolného uživatele. Jako oprávnění pro tuto funkci je vyžadován level 0 – superuživatel. Blokové schéma tohoto dotazu je znázorněno na *Obr. 19.*



Obr. 19. Blokové schéma resolverové funkce getToken().

Do vygenerovaného JWT je zakódováno userID, čas vystavení a čas expirace. K tomuto účelu je použita funkce createToken, která je importována ze souboru ./src/user/auth.js.

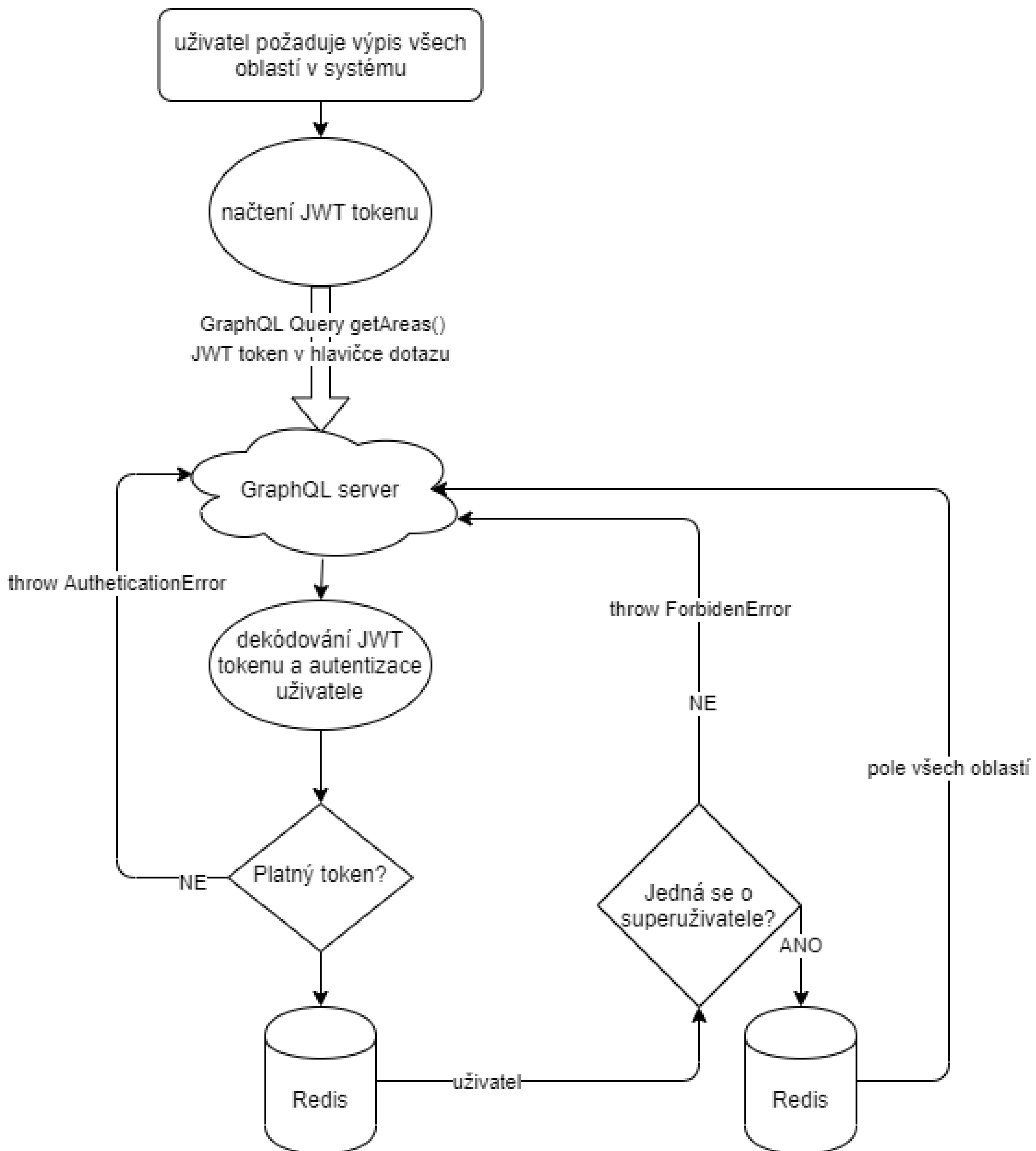
```
  getToken: (parent, { userID }, context) => {
    return new Promise(async (resolve, reject) => {
      // check for privilegies
      if (
        !context.user ||
        !context.user.accessLevel ||
        context.user.accessLevel.level !== 0
      ) {
        reject(
          new ForbiddenError(`User doesn't have
                                sufficient privileges.`),
        );
        return;
      }

      const token = await createToken({ userID });
      resolve(token);
    });
```

Výpis 48. Soubor resolvers.js – vygenerování nového JWT tokenu pomocí API.

Resolverová funkce getAreas()

Tato funkce slouží k získání všech oblastí z databáze. Tyto oblasti musí být navráceny jako návratová hodnota ve formě pole. Blokové schéma tohoto dotazu je znázorněno na *Obr. 20*.



Obr. 20. Blokové schéma resolverové funkce getAreas().

Funkce je dostupná pouze superuživatelům a proto je v prvním kroku při volání funkce provedena kontrola úrovně oprávnění daného uživatele, viz. *Výpis 49*.

```
getAreas: async (parent, args, context) => {
      return new Promise(async (resolve, reject) => {
        // check for privilegies
        if (
          !context.user ||
          !context.user.accessLevel ||
          context.user.accessLevel.level !== 0
        ) {
          reject(
          new ForbiddenError(`User doesn't have
                                    sufficient privileges.`),
          );
          return;
        }
      )
}
```

Výpis 49. Soubor resolvers.js – autorizace resolveru getAreas.

V dalším kroku je provedeno připojení do databáze Redis a získání všech oblastí, viz. *Výpis 50.*

```
        // připojení Redis db
        const client = redis.createClient(redisCnf.port,
                                          redisCnf.url);
        await new Promise((resolve) => {
          client.on('connect', resolve);
        });

        let areas = await new Promise((resolve) => {
          client.hgetall(`area`, function (getError, result) {
            console.log(getError);
            if (getError) throw getError;
            resolve(result);
          });
        });
```

Výpis 50. Soubor resolvers.js – získání seznamu oblastí z databáze.

Získaná data ve formě textových řetězců, *viz. Výpis 51*, je tedy nutné tato data deserializovat a vyparsovat do pole.


```
{
  '5f21efde0a0fee062462d072': '{"id":"5f21efde0a0fee062462d072","name":"Laboratoř","accessLevel":{"level":1,"name":"fullAccess"},"grantedHours":[{"from":"08:00","to":"12:00"},{"from":"13:00","to":"18:00"}]}',
  '5f21efde34c2577b8fca4ebc': '{"id":"5f21efde34c2577b8fca4ebc","name":"Toaleta","accessLevel":{"level":2,"name":"visitor"},"grantedHours":[{"from":"00:00","to":"24:00"}]}',
  '5f21efdede4076505db0166c': '{"id":"5f21efdede4076505db0166c","name":"Jednací místnost","accessLevel":{"level":0,"name":"CEO"},"grantedHours":[{"from":"06:00","to":"22:00"}]}'
}
```


Výpis 51. Syrová data získaná z databáze Redis.

Data ve formě objektu jsou namapována do jednotlivých uživatelů a vyparsována do formátu JSON, viz. *Výpis 52*.

```
        let areasArray = [];
        Object.keys(areas).forEach((key) => {
          areasArray.push(JSON.parse(areas[key]));
        });

        resolve(areasArray);
```

Výpis 52. Soubor resolvers.js – namapování oblastí do pole jakožto návratové hodnoty.

Výsledné pole je vráceno jako návratová hodnota resolveru pomocí funkce resolve(). Ukázka takto zpracovaných dat je zobrazena ve *Výpis 53*.

```
[
  {
    id: '5f21f02d69e488cb84fcd774',
    name: 'Jednací místnost',
    accessLevel: { level: 0, name: 'CEO' },
    grantedHours: [ [Object] ]
  },
  {
    id: '5f21f02d482b9690be1e7078',
    name: 'Toaleta',
    accessLevel: { level: 2, name: 'visitor' },
    grantedHours: [ [Object] ]
  },
  {
    id: '5f21f02d4dd96c02875ede37',
    name: 'Laboratoř',
    accessLevel: { level: 1, name: 'fullAccess' },
    grantedHours: [ [Object], [Object] ]
  }
]
```

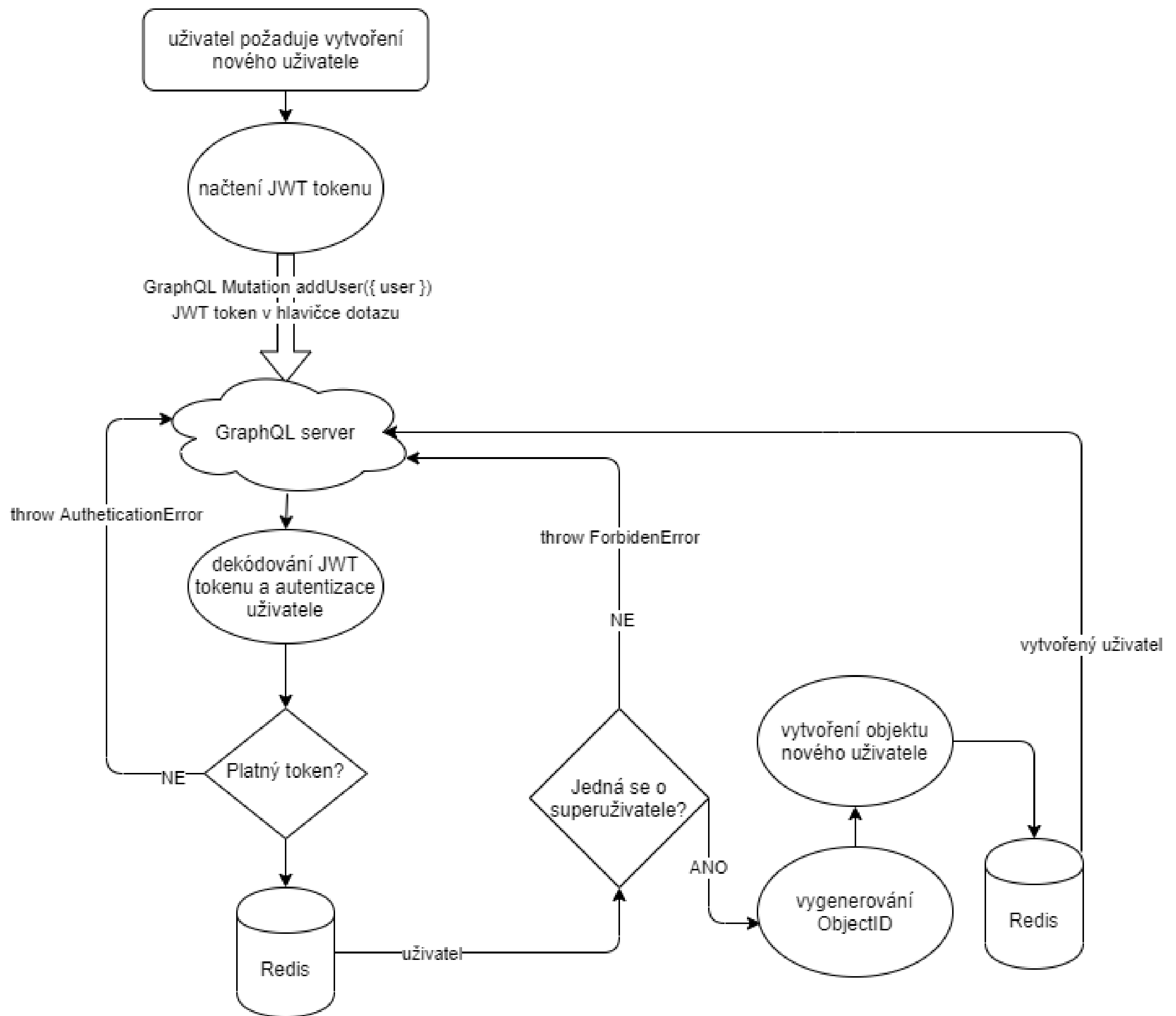
Výpis 53. Data zpracována do pole, které je předáno GraphQL jako návratová hodnota

5.4.3 Resolverové funkce Mutace

Tyto resolverové funkce mají za úkol manipulaci s daty. Jsou exportovány ze souboru resolvers.js v adresáři ./src/resolvers.

Resolverová funkce addUser()

Tato funkce slouží pro přidání nového uživatele. K jejímu vykonání je nutná autorizace na úrovni superuživatele. Blokové schéma této mutace je znázorněno na *Obr. 21.*



Obr. 21. Blokové schéma resolverové funkce addUser().

Funkce je dostupná pouze superuživatelům a proto je v prvním kroku při volání funkce provedena kontrola úrovně oprávnění daného uživatele, viz. *Výpis 54*.

```
addUser: async (parent, args, context) => {
      return new Promise(async (resolve, reject) => {
        // check for privilegies
        if (
          !context.user ||
          !context.user.accessLevel ||
          context.user.accessLevel.level !== 0
        ) {
          reject(
          new ForbiddenError(`User doesn't have
                                sufficient privileges.`),
          );
          return;
        }
      )
}
```

Výpis 54. Soubor resolvers.js – autorizace resolveru getAreas.

V dalším kroku je připojena databáze, vygenerováno ObjectID a vytvořen objekt nového uživatele. Tento krok je znázorněn ve *Výpis 55*.

```
        // připojení Redis db
        const client = redis.createClient(redisCnf.port,
                                          redisCnf.url);
        client.auth(redisCnf.password);
        await new Promise((resolve) => {
          client.on('connect', resolve);
        });

        let newUser = {
          userID: objectID(),
          name: user.name,
          accessLevel: accessLevels[user.accessLevel],
        };
```

Výpis 55. Soubor resolvers.js – vytvoření objektu reprezentujícího nového uživatele.

Nově vytvořený objekt je v dalším kroku převeden do textového řetězce a uložen do databáze Redis, viz. *Výpis 56*. Původní objekt nového uživatele je poté pomocí funkce resolve() vrácen jako návratová hodnota.

```
        client.hmset(
          `user`,
          newUser.userID,
          JSON.stringify(newUser),
          (err, res) => {
            if (err) {
              reject(err);
              return;
            }
            resolve(newUser);
          },
        );
```

Výpis 56. Soubor resolvers.js – uložení nového uživatele ve formě textového řetězce.

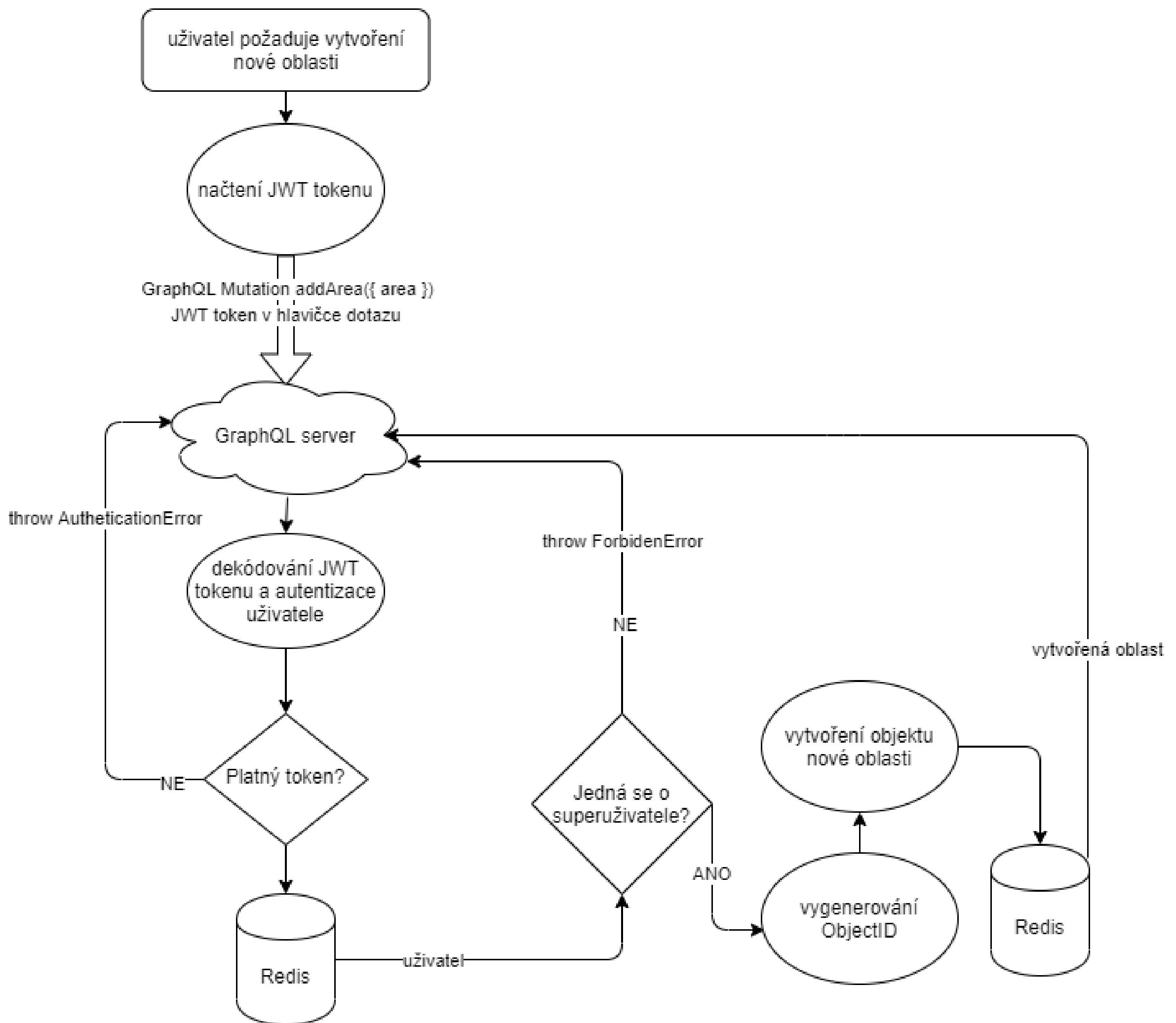
Jako parametr při volání této mutace je nutné použít datatyp UserInput. Příklad tohoto parametru je zobrazen ve *Výpis 57.*

```
{
  "user": {
    "name": "Josef Novák",
    "accessLevel": 0
  }
}
```

Výpis 57. Ukázka vstupní proměnné mutace addUser.

Resolverová funkce addArea()

Tato funkce slouží pro přidání nové oblasti. K jejímu vykonání je nutná autorizace na úrovni superuživatele. Blokové schéma této mutace je znázorněno na *Obr. 22.*



Obr. 22. Blokové schéma resolverové funkce addArea().

Funkce je dostupná pouze superuživatelům a proto je v prvním kroku při volání funkce provedena kontrola úrovně oprávnění daného uživatele, viz. *Výpis 58*.

```
addArea: async (parent, args, context) => {
      return new Promise(async (resolve, reject) => {
        // check for privilegies
        if (
          !context.user ||
          !context.user.accessLevel ||
          context.user.accessLevel.level !== 0
        ) {
          reject(
          new ForbiddenError(`User doesn't have
                                    sufficient privileges.`),
          );
          return;
        }
      )
}
```

Výpis 58. Soubor resolvers.js – autorizace resolveru addArea.

V dalším kroku je připojena databáze, vygenerováno ObjectID a vytvořen objekt nové oblasti pomocí metody zvané destructuring. Tento krok je znázorněn ve *Výpis 59*.

```
        // connect to the Redis db
        const client = redis.createClient(redisCnf.port,
                                          redisCnf.url);
        client.auth(redisCnf.password);
        await new Promise((resolve) => {
          client.on('connect', resolve);
        });

        let newArea = {
          ...area,
          id: objectID(),
        };
```

Výpis 59. Soubor resolvers.js – vytvoření objektu reprezentujícího novou oblast.

Nově vytvořený objekt je v dalším kroku převeden do textového řetězce a uložen do databáze Redis.

Původní objekt nové oblasti je poté pomocí funkce resolve() vrácen jako návratová hodnota, viz. *Výpis 60*.

```
        client.hmset(
          `area`,
          newArea.id,
          JSON.stringify(newArea),
          (err, res) => {
            if (err) {
              reject(err);
              return;
            }
            resolve(newArea);
          },
        );
```

Výpis 60. Soubor resolvers.js – uložení nové oblasti ve formě textového řetězce.

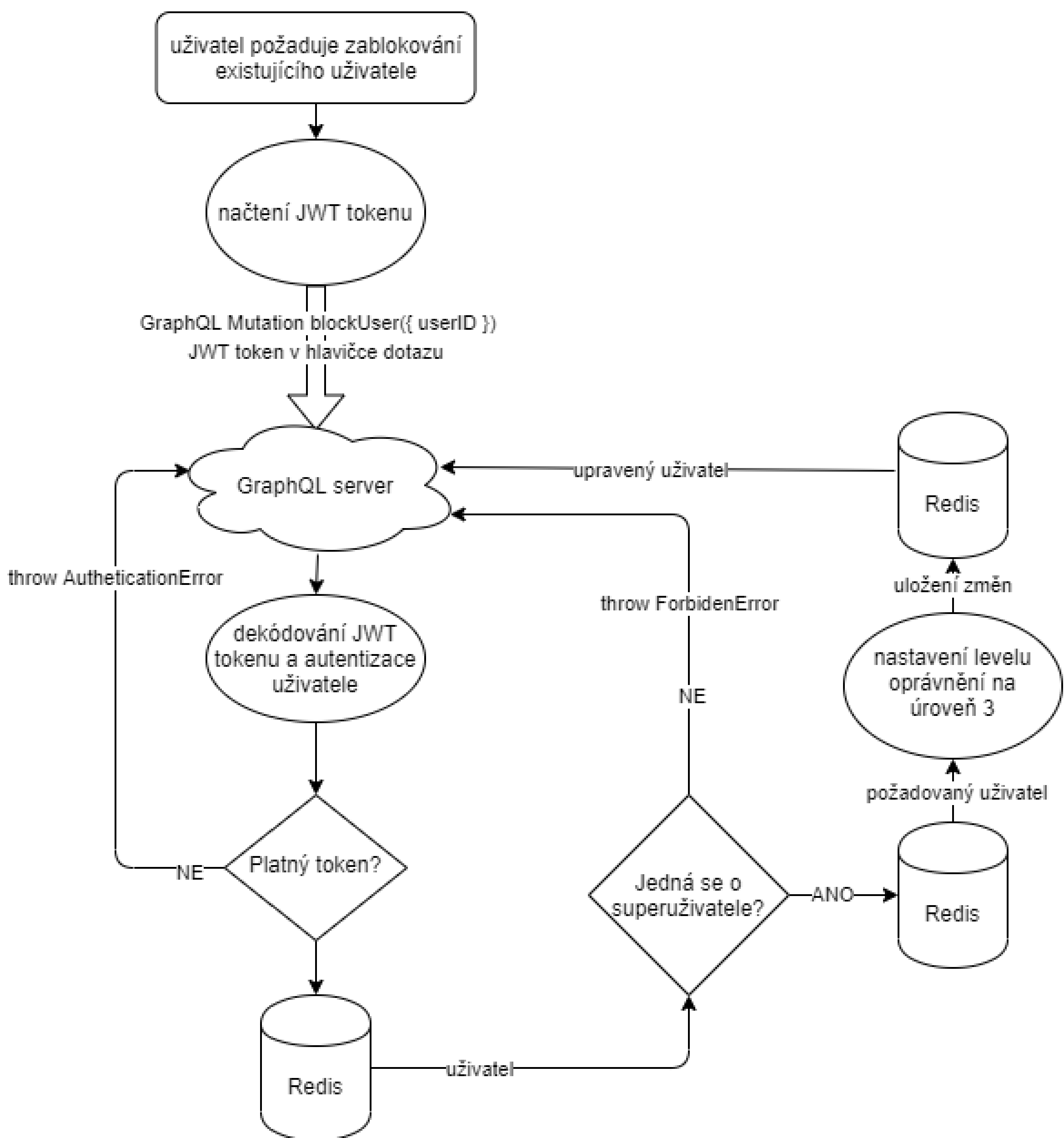
Jako parametr při volání této mutace je nutné použít datatyp AreaInput. Ukázka tohoto parametru je znázorněna ve *Výpis 61*.

```
{
  "area": {
    "name": "Nová místnost",
    "accessLevel": 0,
    "grantedHours": [
      {
        "from": "12:00",
        "to": "23:00"
      },
      {
        "from": "23:10",
        "to": "23:50"
      }
    ]
  }
}
```

Výpis 61. Ukázka vstupní proměnné mutace addArea.

Resolverová funkce blockUser()

Tato funkce slouží pro zablokování existujícího uživatele. Uživatel po této akci nebude mít přístup do žádné oblasti. K jejímu vykonání je nutná autorizace na úrovni superuživatele. Blokové schéma této mutace je znázorněno na *Obr. 23.*



Obr. 23. Blokové schéma resolverové funkce blockUser().

Funkce je dostupná pouze superuživatelům a proto je v prvním kroku při volání funkce provedena kontrola úrovně oprávnění daného uživatele, viz. *Výpis 62*.

```
blockUser: async (parent, args, context) => {
      return new Promise(async (resolve, reject) => {
        // check for privilegies
        if (
          !context.user ||
          !context.user.accessLevel ||
          context.user.accessLevel.level !== 0
        ) {
          reject(
          new ForbiddenError(`User doesn't have
                                    sufficient privileges.`),
          );
          return;
        }
      )
}
```

Výpis 62. Soubor resolvers.js – autorizace resolveru blockUser.

V dalším kroku je připojena databáze a načten požadovaný uživatel, viz. *Výpis 63*.

```
        // connect to the Redis db
        const client = redis.createClient(redisCnf.port,
                                          redisCnf.url);
        client.auth(redisCnf.password);
        await new Promise((resolve) => {
          client.on('connect', resolve);
        });

        let user = await new Promise((resolve) => {
          client.hget(`user`, userID, function (getError, res) {
            if (getError) throw getError;
            resolve(res);
          });
        });
```

Výpis 63. Soubor resolvers.js – vytvoření objektu reprezentujícího novou oblast.

Výsledek čtení z databáze je nutné zkontrolovat a vyhodnotit, zda uživatel s požadovaným ID opravdu existuje, viz. *Výpis 64*. Pokud neexistuje, je vyvolán chybový stav.

```
        if (!user) {
          reject(new Error(`User not found!`));
          return;
        }
```

Výpis 64. Soubor resolvers.js – kontrola existence uživatele.

Data z databáze jsou dále rozparsována do objektu a je nastavena úroveň oprávnění na level 3 – blocked user. Takto upravený objekt je uložen do databáze Redis a pomocí funkce resolve() vrácen jako návratová hodnota, viz. *Výpis 65*.

```
        user = JSON.parse(user);

        user.accessLevel = accessLevels[3];

        client.hmset(`user`, user.userID, JSON.stringify(user),
          (err, res) => {
          if (err) {
            reject(err);
            return;
          }
          resolve(user);
        });
```

Výpis 65. Soubor resolvers.js – uložení nové oblasti ve formě textového retězce.

Jako parametr při volání této mutace je nutné použít datatyp ID. Ukázka tohoto parametru je znázorněna na *Výpis 66*.

```
{
  "userID": "5f2034519c9263e3d8a8191f"
}
```

Výpis 66. Ukázka vstupní proměnné mutace blockUser.

5.4.4 Sestavení https a Apollo serveru

V následujícím kroku je nutné vytvořit instanci express serveru, https serveru a Apollo serveru. Tyto tři moduly budou poté propojeny pomocí middleware.

Vygenerování selfsigned TLS certifikátu

Pro úspěšné sestavení https je nutné poskytnout TLS certifikát a privátní klíč. Tento pár je možné vygenerovat pomocí utility openssl následujícím příkazem:

```
$ openssl req -nodes -new -x509 -keyout server.key -out server.cert
```

Spuštění tohoto příkazu v terminálu vygeneruje selfsigned TLS certifikát ve formátu X509 a privátní klíč, které uloží do souborů server.key a server.cert, viz. *Výpis 67*.


```
-----BEGIN CERTIFICATE-----
MIIDazCCAlOgAwIBAgIUbAr8ZhIAmHsMwaIspCwodWbEDJIwDQYJKoZIhvcNAQEL
BQAwRTELMAkGA1UEBhMCQVUxEzARBgNVBAgMClNvbWUtU3RhdGUxITAfBgNVBAoM
GEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZDAeFw0yMDA3MTYxNTA2MjlaFw0yMDA4
MTUxNTA2MjlaMEUxCzAJBgNVBAYTAkFVMRMwEQYDVQQIDApTb21lLVN0YXRlMSEw
HwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwggEiMA0GCSqGSIb3DQEB
AQUAA4IBDwAwggEKAoIBAQDr60UcMmyntz05bc8nmYjKs5tDx2ISKvNuTuPkgeTa
zK82DD4LEUM4ltH2F3C1VUBLrhRFCdOCVCj6GERYMs2AusuSusm/DoeMvmp+XfZN
KI49Vba45ir77FiueuIs5uaCWsbL0o6IxRBYAEpOpWmI7fuX4FZv1YaXVArKkKrr
iejheMu9cW8HmFZ9Ufm0/W0cHMThkNZiZZvT+YAkGB3nUacwobQtrdSZzF3Plt1b
3xs7Pn9fLz+XDR0TzfEyRQu4MbYeeF668AxfSzbDrvSNz8/UnU80Vjdrapx4JVp9
FvBIPLaOQxt6qti0/FestgXvGyJFWKrW0n//Q0FRzIbdAgMBAAGjUzBRMB0GA1Ud
DgQWBBSrm5wneJpIwNXpdzRKSY7kDlSHhzAfBgNVHSMEGDAWgBSrm5wneJpIwNXp
dzRKSY7kDlSHhzAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3DQEBCwUAA4IBAQCp
qMsPsQpWGfxthK9Tyr6HtFhO4r44WhtYz9orA4QLO2fJvHgNFTjoJryraawvJnee
2YpmnVo5zyy7c9XIWOoYYgyBwRcldFeiwiSOHsSSUJ+qDRJieksb1CtfqI1ye4sd
db+s9aW/zTU1wRdhtR2BN/UiGN1uOL2SF4Ae65dXdUmzUdRDiFryBQAoLe2Kgv6a
Wa399at5QGx10fxfjU8jaa2tHCZqoEmMDpSCk5lb44X8MJZUINyFTBItcwBWEM8c
iDM6b9I6RC3o3eyU/+xKEgE7SN02Hlju5UJWuyv8t/r627VD8xTsZyHP7mAArWuY
u3b5b2Lk4PHxBCtA8q5a
-----END CERTIFICATE-----
```


Výpis 67. Vygenerovaný TLS certifikát.

Certifikát tohoto typu nelze použít veřejně, protože se nejedná o certifikát, který by byl podepsán důvěryhodnou autoritou. Každý veřejně použitý certifikát musí vygenerovat a podepsat uznaná certifikační autorita. Vzhledem ke skutečnosti, že se jedná o M2M komunikaci

a veřejný certifikát lze napevno zasadit do klientské části, se nutně nejedná o handicap. Pokud je tento certifikát v klientské části jako jediný důvěryhodný, klient není schopen navázat spojení s nikým jiným než s držitelem privátního klíče náležícím k danému certifikátu. Tímto je zabráněno možnosti podvržení API pomocí DNS útoků, man-in-the-middle útoku a dalších metod.

Sestavení serveru

Pro účely sestavení serveru slouží soubor server.js v kořenovém adresáři, tento soubor je také hlavním entrypointem celého balíčku. V prvním kroku jsou importovány potřebné knihovny, objekty a funkce. Tento krok je znázorněn ve *Výpis 68.*

```
const { ApolloServer } = require('apollo-server-express');
const { AuthenticationError } = require('apollo-server-express');
const typeDefs = require('./src/schema/schema');
const resolvers = require('./src/resolvers/resolvers');
const { getUser } = require('./src/user/auth');
const https = require('https');
const express = require('express');
const fs = require('fs');
```

Výpis 68. Soubor server.js – import potřebných knihoven, resolverů a schématu.

Jako konfigurace Apollo serveru je použit objekt obsahující cestu k certifikátům a číslo portu na kterém má být server spuštěn, viz. *Výpis 69*.

```
const apolloCnf = {
  port: 443,
  keyPath: './server.key',
  certPath: './server.cert',
};
```

Výpis 69. Soubor server.js – konfigurace serveru.

Nyní následuje samotné vytvoření instance Apollo serveru. Konstruktor ApolloServer má vstupní parametry obsahující importované schéma, resolverové funkce a context.

Koncept contextu:

Context je objekt, nebo funkce, které jsou volány při každém požadavku. Pomocí této funkce lze injectovat kontext, který je dále sdílen napříč všemi resolverovými funkcemi. Tato konstrukce je tedy ideální pro implementaci vrstvy autentizace a autorizace.

Autentizace a autorizace požadavku:

Při procesování každého požadavku je zavolána anonymní funkce, která má jako parametr objekt req. Z tohoto objektu je následně vyparsována hlavička a její parametr authorization v podobě textového řetězce. Tento řetězec je otestován, zda začíná řetězcem "Bearer ". Pokud tomu tak není, je vyvolán chybový stav neautentifikováno, což způsobí zamezení přístupu k resolverům a požadavek není uspokojen. Pokud řetězec splňuje danou podmínku, je z něho daná fráze odstraněna funkcí substring a jako návratová hodnota je vrácen příslib budoucí hodnoty funkce getUser(). Tato funkce vrátí objekt obsahující údaje daného uživatele v případě validního tokenu, viz. *Výpis 70*, nebo vyvolá chybový stav neautentifikováno v případě neplatného, nebo chybějícího JWT tokenu.

```
// vytvoření instance ApolloServeru
const apollo = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    // získání tokenu z HTTP hlavičky
    let token = req.headers.authorization || '';

    if (token.startsWith('Bearer ')) {
      token = token.substring(7, token.length);
      // handle a token
      return getUser({ token });
    } else {
      //Error
      throw new AuthenticationError('Unauthenticated');
    }
  },
});
```

Výpis 70. Soubor server.js – vytvoření instance Apollo serveru pomocí konstruktoru.

Apollo server defaultně neimplementuje TLS, protože se předpokládá provoz za proxy serverem v rámci lokální sítě. K implementaci TLS je nutné použít mezivrstvy v podobě Express serveru a knihovny https, viz. *Výpis 71*.

```
// vytvoření instance https server
const server = https.createServer(
  {
    key: fs.readFileSync(apolloCnf.keyPath),
    cert: fs.readFileSync(apolloCnf.certPath),
  },
  app,
);

// aplikování express middleware na apollo
apollo.applyMiddleware({ app });
```

Výpis 71. Soubor server.js - implementace TLS pomocí https a Express.

Jako poslední krok je nutné spustit server pomocí zavolání metody listen().

```
// `listen` metoda spouští webserver
server.listen({ port: apolloCnf.port }, async () => {
  console.log(
    '🚀 Server ready at',
    `https://localhost:${apolloCnf.port}${apollo.graphqlPath}`,
  );
});
```

Výpis 72. Soubor server.js – spuštění serveru na daném portu.

Nyní je možné celý server spustit pomocí terminálu následujícím příkazem.

```
$ node server.js
🚀 Server ready at https://localhost:443/graphql
redis is connected!
```

Výpis 73. Spuštění serveru v terminálu.

6 TESTOVÁNÍ

Testování lze provádět ručně, nebo automaticky v rámci CI/CD. V první části této sekce bude provedeno ruční testování, které se používá během fáze vývoje v rámci ladění. V druhé fázi poté budou navrženy automatické testy ve frameworku Mocha.

6.1 Insomnia

Program Insomnia je vyvíjen společností Kong Inc. a v základní verzi je zdarma (Insomnia je zdarma k použití navždy, ale lze ji vylepšit tak, aby vyhovovala potřebám vás nebo vašeho týmu.). Jedná se o komplexní program pro testování API jehož heslo je „Navrhněte a odlaďujte API jako člověk, ne jako robot."

Mezi jeho hlavní funkce patří:

1. GraphQL support
2. OAuth 1.0 and 2.0 auth
3. Multipart form builder
4. Query parameter builder
5. Plugin System
6. SSL client certificates
7. JSONPath and XPath
8. Response history
9. Data import/export
10. Image and SVG preview
11. AWS authentication
12. Configurable proxy
13. Color themes
14. Cloud sync and sharing
15. Mac, Windows, Linux
16. JWT Bearer token

6.1.1 Nastavení autentizace

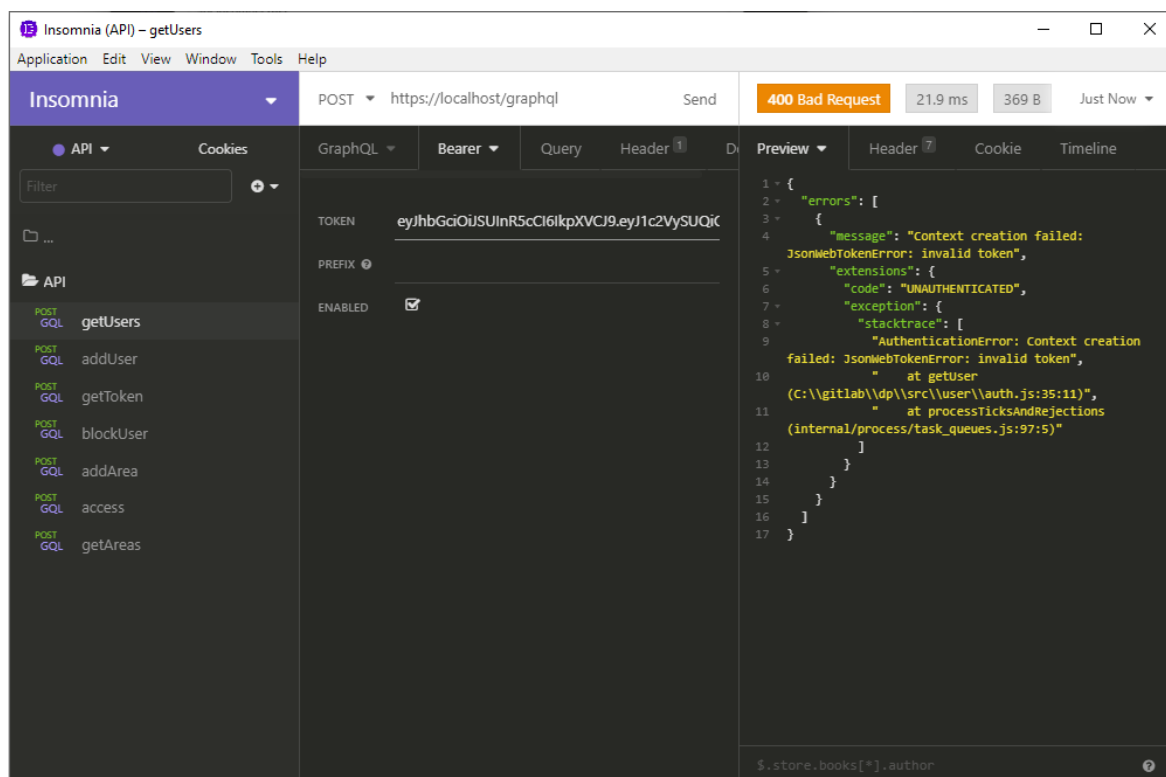
V každém dotazu pomocí programu Insomnia je nutné nastavit autentizaci pomocí Bearer tokenu.

6.1.2 Nastavení TLS

Nastavení TLS probíhá automaticky a je vynuceno pomocí volání API na portu 443 (https).

6.1.3 Otestování Query bez platného JWT tokenu

Bez platného tokenu by neměla být dostupná žádná funkce implementovaného API.

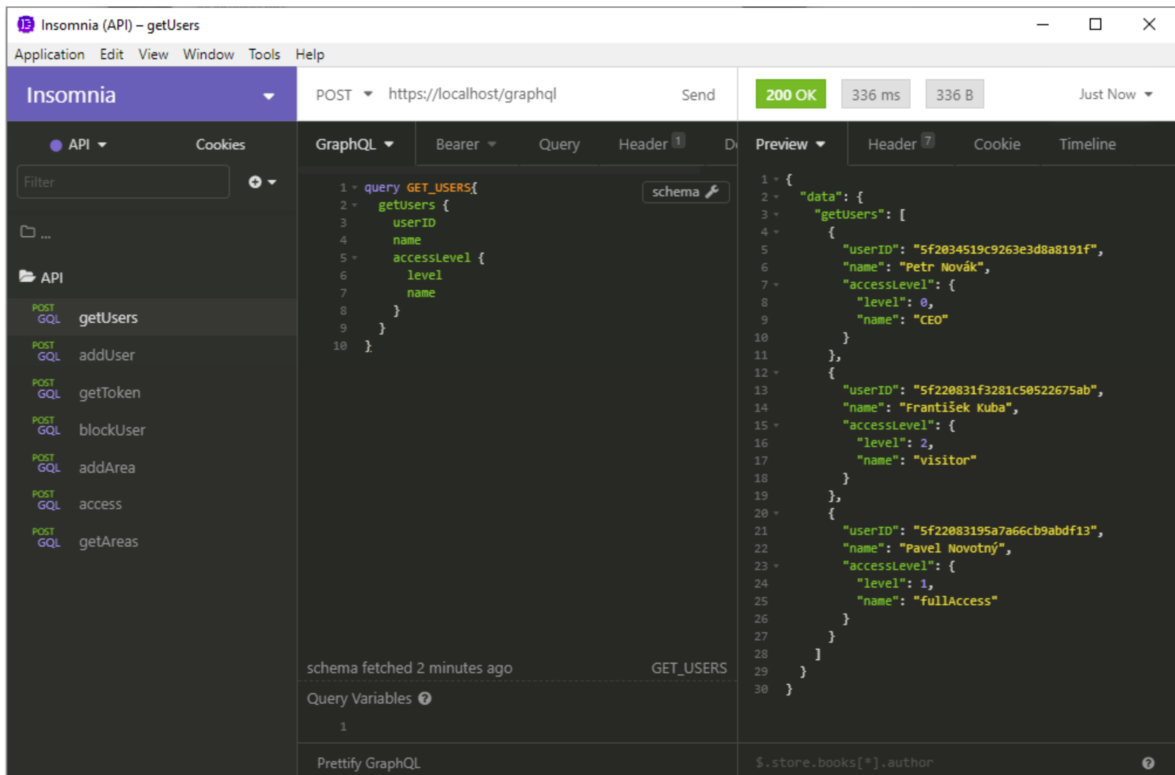


Obr. 24. Volání API s nesprávným JWT tokenem.

Dle předpokladu byla vrácena chyba UNAUTHENTICATED a dotaz nebyl vykonán, viz. *Obr. 24*.

6.1.4 Otestování Query getUsers

Tento dotaz je volán bez proměnných a vrací seznam všech uživatelů.



Obr. 25. Otestování query getUsers s platným JWT tokenem.

Dle předpokladu bylo vráceno pole obsahující seznam všech uživatelů, viz. *Obr. 25.*

6.1.5 Otestování Query getToken

Tento dotaz je volán s proměnnou obsahující userID a vrací nově vygenerovaný JWT token.

Obr. 26. Otestování query getToken s platným JWT tokenem.

Dle předpokladu byl vrácen vygenerovaný JWT token, viz. *Obr. 26*.

6.1.6 Otestování Query getAccess

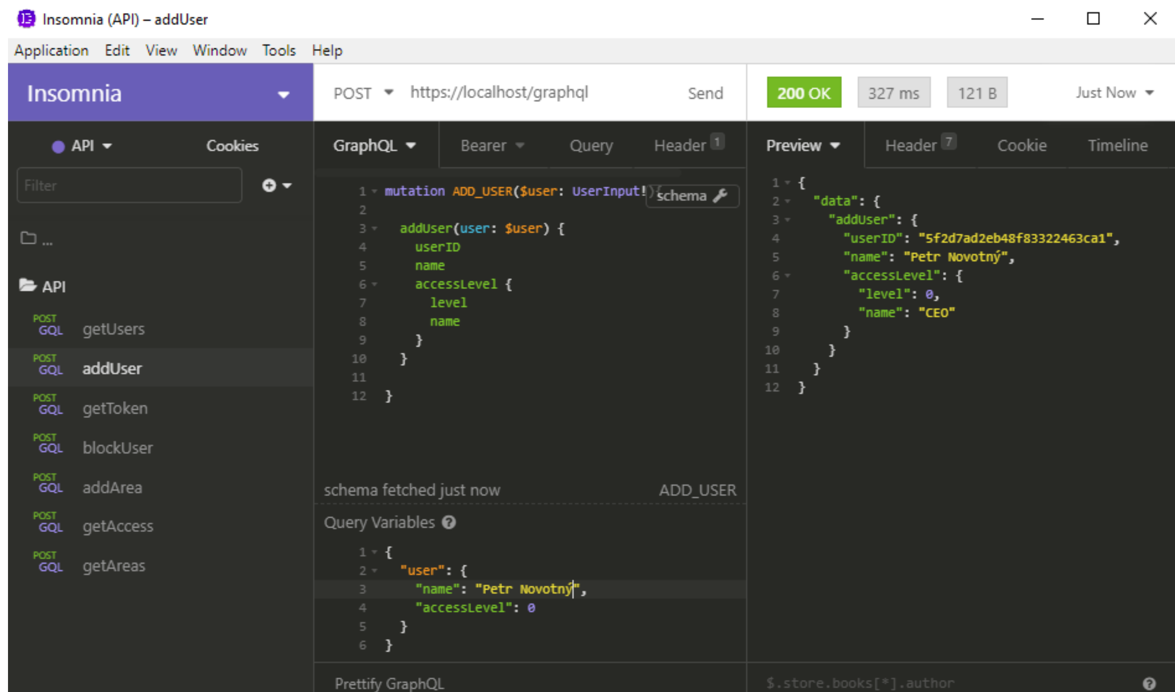
Tento dotaz je volán s proměnnou obsahující areaID a vrací boolean hodnotu reprezentující informaci, zda je přístup povolen, nebo zamítnut.

Obr. 27. Otestování query getAccess s platným JWT tokenem.

Dle předpokladu byla navrácena boolean hodnota reprezentující povolení ke vstupu do oblasti, viz. *Obr. 27.*

6.1.7 Otestování Mutace addUser

Tato mutace je volána s proměnnou obsahující objekt nově vytvářeného uživatele a vrací objekt vytvořeného uživatele, nebo chybu.

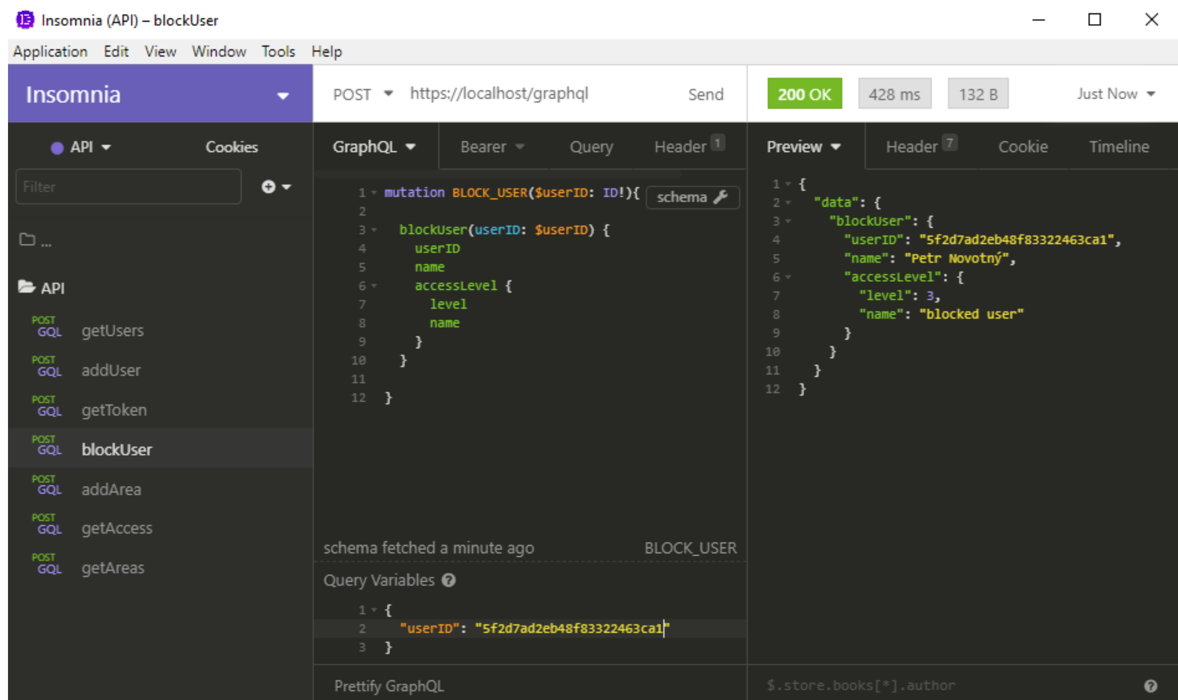


Obr. 28. Otestování mutace addUser s platným JWT tokenem.

Dle předpokladu byl vrácen nový uživatel s vygenerovaným userID, viz. *Obr. 28.*

6.1.8 Otestování Mutace blockUser

Tato mutace je volána s proměnnou obsahující userID a vrací objekt uživatele na kterém je provedena požadovaná akce.

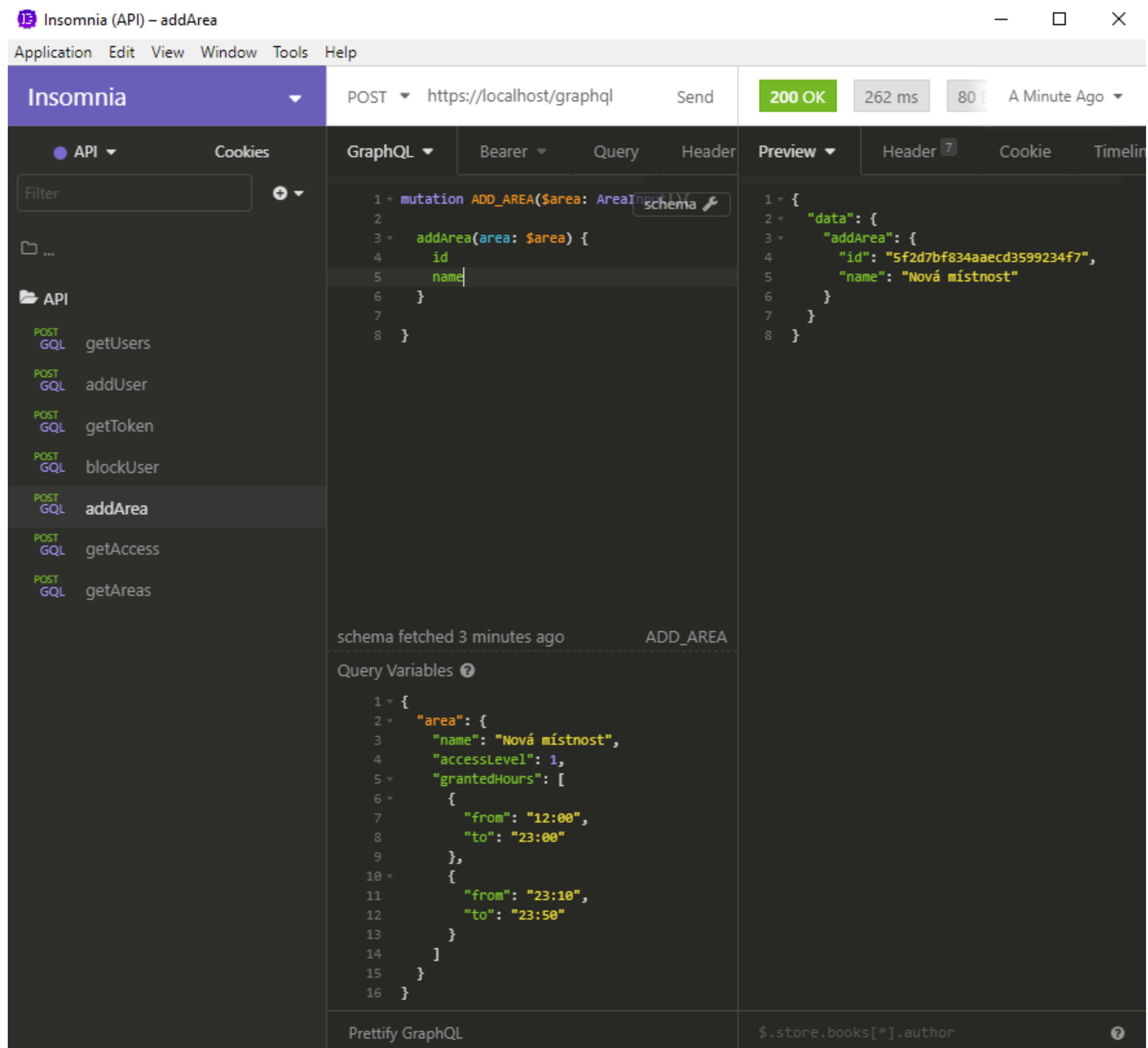


Obr. 29. Otestování mutace blockUser s platným JWT tokenem.

Dle předpokladu byl vrácen zeditovaný uživatel s úrovní oprávnění „blocked user“, viz. *Obr. 29.*

6.1.9 Otestování Mutace addArea

Tato mutace je volána s proměnnou obsahující objekt nově vytvářené oblasti a vrací objekt vytvořené oblasti, viz. *Obr. 30*.



Obr. 30. Otestování mutace addArea s platným JWT tokenem.

Dle předpokladu byl vrácen objekt nově vytvořené oblasti, viz. *Obr. 30*.

6.2 Mocha

Mocha je testovací framework JavaScriptu bohatý na funkce běžící na Node.js a v prohlížeči, což zjednodušuje asynchronní testování. Testy Mocha probíhají sériově, což umožňuje flexibilní a přesné podávání zpráv. Mocha je poskytována jako OpenSource a je hostována na GitHubu [14]. Je distribuována pomocí balíčkovacího systému NPM.

6.2.1 Testy pomocí Mocha

Tyto testy jsou integrační a provádí se zcela automaticky při každé změně. Výhodou je rychlé odhalení různých bočních efektů a zamezení průsaku nově zavedených bugů do produkčního prostředí z developerských verzí. V této části budou automatickými testy pokryty funkce pro práci s JWT tokeny.

Přidání testů do package.json

Testovací příkaz je do package.json vložen následovně:

```
  "scripts": {
    "test": "mocha"
  },
```

Výpis 74. Soubor package.json – přidání nástroje mocha.

Vytvoření testů

Vytvoření testů je realizováno pomocí souboru index.js v adresáři ./test/. Tento soubor obsahuje funkci potřebné importy a je v něm volána funkce desribe(), ve které je definována sada testů, viz. *Výpis 75.*

```
var assert = require('assert');
const chai = require('chai');
const expect = chai.expect;
chai.use(require('chai-as-promised'));
const { createToken, getuserID } = require('../src/user/jwt');

describe('jwt functions', () => {
  let generatedToken = '';
});
```

Výpis 75. Soubor index.js – definování bloku automatických testů.

Otestování vytvoření JWT tokenu

Tento test má za cíl vygenerovat a ověřit JWT token. Test tedy garantuje správnou funkcionalitu funkcí zajišťujících práci s JWT tokeny, viz. *Výpis 76*. Je také testováno, zda dojde k detekci chyby při chybějícím userID.

```
let generatedToken = '';
it('vytvoření tokenu', async function () {
  return createToken({ userID: objectId })
    .then((token) => {
      generatedToken = token;
    })
    .catch((err) => {
      assert.fail(err);
    });
});

it('vytvoření tokenu s prázdným userID', async function () {
  await expect(createToken({ userID: '' })).to.be.rejectedWith(
    'Délka userID je nekorektní!',
  );
});
```

Výpis 76. Soubor index.js – test funkce generující JWT tokeny.

Otestování dekódování a ověření JWT tokenu

Tento test otestuje, zda je možné ověřovat JWT tokeny a zda dochází k detekci poškozených tokenů, viz. *Výpis 77*.

```
  it('dekódování a ověření tokenu', async function () {
    return getuserID({ token: generatedToken })
      .then((userID) => {
        assert.equal(userID === objectId, true,
                     'userIDs není shodné');
      })
      .catch((err) => {
        assert.fail(err);
      });
  });

  it('dekódování a ověření poškozeného tokenu', async function ()
{
    await expect(
      // damage the token by slicing
      getuserID({ token: generatedToken.slice(0, 10) }),
    ).to.be.rejectedWith('jwt poškozen');
  });
```

Výpis 77. Soubor index.js – test funkce generující JWT tokeny.

Spuštění testů

Testy je možné spustit následujícím příkazem:

```
$ npm test
```

Po spuštění jsou všechny testy automaticky vykonány s následujícím výstupem do konzole:

```
> graphql-server@1.0.0 test C:\gitlab\dp
> mocha



  jwt functions
    √ vytvoření tokenu
    √ vytvoření tokenu s prázdným userID
    √ dekódování a ověření tokenu
    √ dekódování a ověření poškozeného tokenu


  4 passing
```

Obr. 31. Výstup do konzole z nástroje Mocha.

Výstup na *Obr. 31* potvrzuje správné vykonání napsaných testů a správnou funkcionalitu testovaných funkcí včetně otestování očekávaných chybových stavů.

7 ZHODNOCENÍ PŘÍNOSŮ

Dle výsledků testování se podařilo navrhnout a implementovat moderní API komunikující pomocí end-to-end šifrování realizovaného pomocí TLS a autentizace s autorizací realizovaných pomocí JWT tokenů.

Vybrané programovací technologie a knihovny poskytují značnou míru abstrakce programování a díky tomuto přístupu je možné implementovat velmi pokročilé zabezpečovací mechanismy bez podrobných znalostí nízkoúrovňových programovacích jazyků. Tato abstrakce také zamezí mnoha chybám při implementaci, protože nízkoúrovňová vrstva (v případě zvoleného Node.JS se jedná o programy v jazyku C a C++) je velmi dobře testována a na jejím vytváření a ladění se podílí velké týmy programátorů z celého světa.

Abstrakce je v současné době rozšiřována také do serverového sektoru, kde ji reprezentuje trend cloudových služeb. Na příkladu databáze Redis bylo předvedeno, jak rychlé a efektivní může být používání cloudových služeb. Protože většina renomovaných cloudových poskytovatelů dodržuje vysoké standardy bezpečnosti a zálohování dat, v některých případech lze toto řešení považovat za výhodnější než vlastní serverové řešení.

Použití těchto technologií, metod a přístupů je možné zvážit v průmyslu komerční bezpečnosti. Tento průmysl je tradičně založen na práci s nízkoúrovňovými programovacími jazyky a vlastním serverovým řešením, což nemusí být vždy nejlepší řešení.

ZÁVĚR

V rámci diplomové práce byla zpracována literární rešerše z oblasti technologií aplikačních rozhraní a možností jejich zabezpečení pomocí autentizace a autorizace. Jako vhodná technologie pro zabezpečení uvažovaného API ACCESS systému byla vybrána technologie JWT tokenů z následujících důvodů:

- JWT je vhodná pro M2M autentizaci (turnikety, zámky dveří, závory a další prvky ACCESS systémů) a s ohledem na fakt, že ve zvažovaném API budou tyto tokeny vydávány centrálně, není nutné použít velmi složité postupy oAuth vhodné spíše pro mezi-systémové přihlašování.
- V rámci JWT lze jednoduše implementovat silné zabezpečení tokenů pomocí kryptografie.
- Do JWT tokenů lze uložit užitečné informace – payload.
- JWT tokeny jsou bezestavové, lze je použít všude, kde je dostupný veřejný klíč vydavatele tokenu.

Pro účely uložení dat na straně serveru byla vybrána databáze Redis z těchto důvodů:

- Velmi rychlá In-Memory databáze.
- Možné nastavení persistence dat.
- Možný provoz v cloudu.
- Dostupné klientské knihovny pro většinu programovacích jazyků.

Dále byla provedena analýza konvenčního REST API, kde byly rozebrány jeho nedostatky a složitost. Jako nástupce této technologie je považována technologie GraphQL a byla tedy zvolena pro řešení uvažovaného API ACCESS systému.

Toto GraphQL rozhraní bylo implementováno pomocí skriptovacího jazyka ECMA Script 2017 a runtime Node.js.

Po úspěšné implementaci bylo rozhraní otestováno a to jak manuálně, tak automaticky.

SEZNAM POUŽITÉ LITERATURY

[1] REDDY, Martin. *API design for C++*. Burlington, MA: Morgan Kaufmann, c2011. ISBN 0123850037.

[2] TODOROV, Dobromir. *Mechanics of user identification and authentication: fundamentals of identity management*. Boca Raton: Auerbach Publications, c2007. ISBN 1420052195.

[3] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <https://www.rfc-editor.org/info/rfc2617>.

[4] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[5] Hammer-Lahav, E., Ed., "The OAuth 1.0 Protocol", RFC 5849, DOI 10.17487/RFC5849, April 2010, <https://www.rfc-editor.org/info/rfc5849>.

[6] TILBORG, Henk C. A. van a Sushil JAJODIA. *Encyclopedia of cryptography and security*. 2nd ed. New York: Springer, c2011. Springer reference. ISBN 978-1-4419-5906-5.

[7] About Node.js®. *Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.* [online]. USA: OpenJS Foundation, 2020 [cit. 2020-05-10]. Dostupné z: https://nodejs.org/en/about/

[8] OPPLIGER, Rolf. *SSL and TLS: theory and practice*. Second edition. Boston: Artech House, [2016]. ISBN 978-1608079988.

[9] VIEGA, John, Matt MESSIER a Pravir CHANDRA. *Network security with OpenSSL*. Sebastopol, CA: O'Reilly, c2002. ISBN 059600270x.

[10] ÖZKAN, Akif. Implementation of a Lightweight Trusted Platform Module. Nürnberg, 2014. Thesis. Friedrich-Alexander-University of Erlangen.

[11] JWT [online]. USA: Auth0, 2020 [cit. 2020-03-16]. Dostupné z: https://jwt.io/

[12] JACKSON, Wallace. JSON quick syntax reference. Lompoc, CA: Apress, [2016]. Expert's voice in Web development. ISBN 978-1484218624.

[13] MASSE, Mark. REST API Design Rulebook. 1st ed. OReilly Media; 2011. ISBN 978-1449310509.

[14] *Mocha - the fun, simple, flexible JavaScript test framework* [online]. USA: OpenJS Foundation, 2020 [cit. 2020-07-14]. Dostupné z: https://mochajs.org/

[15] *GraphQL: A query language for APIs.* [online]. USA: GraphQL Foundation, 2020 [cit. 2020-07-17]. Dostupné z: https://graphql.org/

[16] Get started with Apollo Server. *Apollo GraphQL | Apollo Data Graph Platform— unify APIs, microservices, and databases into a data graph that you can query with GraphQL* [online]. USA: Meteor Development Group, 2020 [cit. 2020-07-18]. Dostupné z: https://www.apollographql.com/docs/apollo-server/getting-started/

[17] *Redis* [online]. USA: Redis Labs, 2020 [cit. 2020-07-12]. Dostupné z: https://redis.io/

[18] SINGH, Prabhat Kumar. *RSA: data encryption and data decryption*. Saarbrücken: Lambert academic publishing, [2016]. ISBN 978-3-659-86297-7.

[19] Node Redis. *Npm | build amazing things* [online]. USA: npm, 2020 [cit. 2020-07-20]. Dostupné z: https://www.npmjs.com/package/redis

[20] Get started with Apollo Server. *Apollo GraphQL | Apollo Data Graph Platform— unify APIs, microservices, and databases into a data graph that you can query with GraphQL* [online]. USA: Meteor Development Group, 2020 [cit. 2020-07-18]. Dostupné z: https://www.apollographql.com/docs/apollo-server/getting-started/

[21] *Simple Object Access Protocol (SOAP) 1.1* [online]. USA: W3C, 2000 [cit. 2020-05-10]. Dostupné z: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[22] FIELDING, Roy Thomas. *Architectural styles and the design of network-based software architectures*. USA, 2000. Disertační práce. University of California, Irvine.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

2FA	Dvoufaktorová autentizace
API	Aplikační programovatelné rozhraní
B2B	Business to Business
CI/CD	Continuous Integration and Continuous Deployment
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IMAP	Internet Message Access Protocol
JSON	JavaScript Object Notation Data Interchange Format
M2M	Machine to machine
POSIX	Portable Operating System Interface
REST	Representational state transfer
RS256	RSA Signature with SHA-256
SOAP	Simple Object Access Protocol
SPA	Single Page Application
SSH	Secure Shell
SSL	Secure Sockets Layer
Sniffing	Odposlech komunikace
TLS	Transport Layer Security
URI	Uniform Resource Identifier

SEZNAM OBRÁZKŮ

Obr. 1. Funkční schéma API. 14
Obr. 2. Funkční diagram autentizace a 16
Obr. 3. Schéma protokolu SSL a (pod)vrstev [8]. 18
Obr. 4. Schéma získání kódu HMAC [10]. 21
Obr. 5. Schéma fungování OAuth 1.0. 27
Obr. 6. Schéma metody autorizace pomocí ověřovacího kódu. 31
Obr. 7. JSON zašifrovaný do JWT tokenu [11]. 33
Obr. 8. Schéma URI [13]. 38
Obr. 9. Znázornění funkce GraphQL [15]. 42
Obr. 10. Logo runtime Node.js [7]. 44
Obr. 11. Schéma fungování Apollo GraphQL serveru [16]. 46
Obr. 12. Logo databáze Redis. [18] 48
Obr. 13. Diagram aplikačního rozhraní. 50
Obr. 14. Kontrola tokenu pomocí jwt.io a veřejného klíče. 61
Obr. 15. Vytvoření databáze pomocí služby Redis Labs 63
Obr. 16. Datové a cenové plány služby Redis Enterprice. 64
Obr. 17. Blokové schéma resolverové funkce getAccess() 72
Obr. 18. Blokové schéma resolverové funkce getUsers(). 76
Obr. 19. Blokové schéma resolverové funkce getToken(). 80
Obr. 20. Blokové schéma resolverové funkce getAreas(). 82
Obr. 21. Blokové schéma resolverové funkce addUser(). 86
Obr. 22. Blokové schéma resolverové funkce addArea(). 89
Obr. 23. Blokové schéma resolverové funkce blockUser(). 92
Obr. 24. Volání API s nesprávným JWT tokenem. 100
Obr. 25. Otestování query getUsers s platným JWT tokenem. 101
Obr. 26. Otestování query getToken s platným JWT tokenem. 102
Obr. 27. Otestování query getAccess s platným JWT tokenem. 102
Obr. 28. Otestování mutace addUser s platným JWT tokenem. 103
Obr. 29. Otestování mutace blockUser s platným JWT tokenem. 104
Obr. 30. Otestování mutace addArea s platným JWT tokenem. 105
Obr. 31. Výstup do konzole z nástroje Mocha. 109

SEZNAM TABULEK

Tab. 1. Datové typy JSON..34

SEZNAM VÝPISŮ

Výpis 1. Ukázka TLS certifikátu.....19
Výpis 2. Ukázka výstupu HMAC-SHA1.....21
Výpis 3. Příklad hlavičky JWT tokenu [11].....35
Výpis 4. Příklad užitečných dat JWT tokenu [11].....35
Výpis 5. Následný výpočet podpisu JWT tokenu [11].....36
Výpis 6. Ukázka HTTP požadavku SOAP [21].....37
Výpis 7. Ukázka resolverové funkce v implementaci GraphQL pomocí knihovny Apollo Server. [16]....43
Výpis 8. Ukázka sestavení webového serveru pomocí Node.js [7].....45
Výpis 9. Sestavení GraphQL serveru pomocí knihovny apollo-server [16].....47
Výpis 10. Úrovně oprávnění uživatelů.....51
Výpis 11. Sada oblastí ACCESS systému.....51
Výpis 12. Sada testovacích uživatelů.....52
Výpis 13. Soubor package.json nově založeného projektu.....53
Výpis 14. Generování RSA klíčů pomocí utility ssh-keygen.....55
Výpis 15. Část privátního RSA klíče.....56
Výpis 16. Odvození veřejného klíče PEM z privátního klíče RSA.....56
Výpis 17. Vygenerovaný veřejný RSA klíč.....57
Výpis 18. Soubor jwt.js - import knihovny jsonwebtoken a načtení privátního klíče z disku.....57
Výpis 19. Soubor jwt.js – funkce pro vytváření JWT tokenu.....58
Výpis 20. Otestování funkčnosti vytváření tokenů.....59
Výpis 21. Vygenerovaný JWT token, který je podepsaný pomocí privátního klíče.....60
Výpis 22. Payload vygenerovaného tokenu.....61
Výpis 23. Soubor jwt.js – autentizace ověřením tokenu.....62
Výpis 24. Export funkcí pro práci s JWT.....62
Výpis 25. Soubor config.js obsahující konfiguraci databáze Redis.....64
Výpis 26. Ukázka použití knihovny Redis [19].....65
Výpis 27. Soubor auth.js – připojení Redis klienta a importy potřebných funkcí.....65
Výpis 28. Soubor auth.js - získání úrovně oprávnění z databáze Redis.....66
Výpis 29. Soubor schema.js - datové schéma GraphQL serveru.....67
Výpis 30. Soubor schema.js – datový typ vstupu nového uživatele.....68

Výpis 31. Soubor schema.js – datový typ vstupu nové oblasti...................................*68*

Výpis 32. Soubor schema.js – datový typ vstupu nového časového úsek....................*69*

Výpis 33. Soubor schema.js – datový typ oblasti...*69*

Výpis 34. Soubor schema.js – datový typ časového úseku....*69*

Výpis 35. Soubor schema.js – datový typ úrovně oprávnění. *70*

Výpis 36. Soubor config.js – možné hodnoty datatypu oprávnění..............................*70*

Výpis 37. Soubor schema.js – datový typ reprezentující uživatele. *70*

Výpis 38. Sada demonstračních uživatelů....*71*

Výpis 39. Soubor resolvers.js – autorizace resolveru getAccess................................*73*

Výpis 40. Soubor resolvers.js – načtení požadované oblasti z databáze Redis...........*73*

Výpis 41. Soubor resolvers.js – testování oprávněnosti akce....................................*74*

Výpis 42. Soubor resolvers.js – testování časových úseků. *74*

Výpis 43. Soubor resolvers.js – autorizace resolveru getUsers. *77*

Výpis 44. Soubor resolvers.js – získání seznamu uživatelů z databáze. *78*

Výpis 45. Syrová data získaná z databáze Redis. *78*

Výpis 46. Soubor resolvers.js – namapování uživatelů do pole jakožto návratové hodnoty...*79*

Výpis 47. Data zpracována do pole, které je předáno GraphQL jako návratová hodnota...*79*

Výpis 48. Soubor resolvers.js – vygenerování nového JWT tokenu pomocí API........*81*

Výpis 49. Soubor resolvers.js – autorizace resolveru getAreas. *83*

Výpis 50. Soubor resolvers.js – získání seznamu oblastí z databáze..........................*83*

Výpis 51. Syrová data získaná z databáze Redis. *84*

Výpis 52. Soubor resolvers.js – namapování oblastí do pole jakožto návratové hodnoty. *84*

Výpis 53. Data zpracována do pole, které je předáno GraphQL jako návratová hodnota..*85*

Výpis 54. Soubor resolvers.js – autorizace resolveru getAreas. *87*

Výpis 55. Soubor resolvers.js – vytvoření objektu reprezentujícího nového uživatele. *87*

Výpis 56. Soubor resolvers.js – uložení nového uživatele ve formě textového řetězce. *88*

Výpis 57. Ukázka vstupní proměnné mutace addUser....*88*

Výpis 58. Soubor resolvers.js – autorizace resolveru addArea. 90
Výpis 59. Soubor resolvers.js – vytvoření objektu reprezentujícího novou oblast. 90
Výpis 60. Soubor resolvers.js – uložení nové oblasti ve formě textového řetězce. 91
Výpis 61. Ukázka vstupní proměnné mutace addArea. 91
Výpis 62. Soubor resolvers.js – autorizace resolveru blockUser. 93
Výpis 63. Soubor resolvers.js – vytvoření objektu reprezentujícího novou oblast. 93
Výpis 64. Soubor resolvers.js – kontrola existence uživatele. 94
Výpis 65. Soubor resolvers.js – uložení nové oblasti ve formě textového retězce. 94
Výpis 66. Ukázka vstupní proměnné mutace blockUser. 94
Výpis 67. Vygenerovaný TLS certifikát. 95
Výpis 68. Soubor server.js – import potřebných knihoven, resolverů a schématu. 96
Výpis 69. Soubor server.js – konfigurace serveru. 96
Výpis 70. Soubor server.js – vytvoření instance Apollo serveru pomocí konstruktoru. 97
Výpis 71. Soubor server.js - implementace TLS pomocí https a Express. 98
Výpis 72. Soubor server.js – spuštění serveru na daném portu. 98
Výpis 73. Spuštění serveru v terminálu. 98
Výpis 74. Soubor package.json – přidání nástroje mocha. 106
Výpis 75. Soubor index.js – definování bloku automatických testů. 106
Výpis 76. Soubor index.js – test funkce generující JWT tokeny. 107
Výpis 77. Soubor index.js – test funkce generující JWT tokeny. 108