VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

OVĚŘENÍ MOŽNOSTÍ MIGRACE Z ARCHITEKTURY REST API DO JAZYKA GRAPHQL

EVALUATION OF MIGRATION FROM REST API ARCHITECTURE TO GRAPHQL LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Pavel Parma
AUTHOR

VEDOUCÍ PRÁCE — Ing. Bohuslav Křena, Ph.D.
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Parma Pavel**
Program: Informační technologie
Název: **Ověření možností migrace z architektury REST API do jazyka GraphQL**
Evaluation of Migration from REST API Architecture to GraphQL Language
Kategorie: Web

Zadání:

1. Prostudujte architekturu REST API a jazyk GraphQL.
2. Navrhněte jednoduchý API systém pro správu úkolů.
3. Tento systém nejdříve implementujte pomocí REST API.
4. Následně implementaci systému převeďte do jazyka GraphQL.
5. Zhodnoťte dosažené zkušenosti s migrací z REST API do GraphQL, přičemž se zaměřte zejména na výhody a nevýhody REST API a GrahpQL.

Literatura:

- Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Disertační práce. University of California, Irvine, 2000.
- Facebook: *GraphQL* [online]. 2018. [cit. 2018-10-17]. Dostupné na URL: https://graphql.org/
- Wittern, E., Cha, A., Laredo J. A.: Generating GraphQL-Wrappers for REST(-like) APIs. In: Mikkonen, T., Klamma, R., Hernández, J., editors. *Web Engineering*. ICWE 2018. LNCS 10845. Springer, Cham, 2018. ISBN 978-3-319-91661-3.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Křena Bohuslav, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Cílem této práce je porovnat dvě technologie používané k implementaci webových služeb a zjistit, zdali je novější technologie připravená k využití a za jakých podmínek. Jedná se o architektonický styl zvaný REST a dotazovací jazyk zvaný GraphQL. Výsledkem je nejen popis těchto dvou technologií a realizace jednoduché referenční služby, ale také autorovo zhodnocení některých aspektů, které mají přímý či nepřímý vliv na udržitelnost a rozšiřitelnost softwaru.

Abstract

The aim of this work is to evaluate technologies used for web service development and find out if the newer technology is ready to be used and under what conditions. It is architecture called REST and query language called GraphQL. Outcome is description of those technologies, simple web service as reference implementation, and author's evaluation of few aspects that have direct or indirect impact on sustainability and extensibility.

Klíčová slova

Webová služba, REST, GraphQL, React, Java

Keywords

Web service, REST, GraphQL, React, Java

Citace

PARMA, Pavel: *Ověření možností migrace z REST API do Jazyka GraphQL*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Bohuslav Křena, Ph.D.

Ověření možností migrace z architektury REST API do jazyka GraphQL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Bohuslava Křeny, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Pavel Parma
1.5.2019

Poděkování

Rád bych poděkoval svému vedoucímu práce za skvělé podněty k zamyšlení a volnost, kterou mi při psaní této práce poskytl.

Obsah

1 ÚVOD ... 6

2 TECHNOLOGIE ... 7

2.1 PROTOKOL HTTP ... 7

2.1.1 Komunikace ... 7

2.1.2 Bezstavovost ... 9

2.1.3 Bezpečnost ... 10

2.1.4 Identifikace zdrojů ... 10

2.1.5 Verze ... 11

2.2 RPC ... 12

2.2.1 SOAP ... 13

2.3 REST ... 13

2.3.1 Principy ... 13

2.3.2 Richardson maturity model ... 15

2.3.3 Pojmenování zdrojů ... 17

2.3.4 Dokumentace ... 17

2.3.5 Verzování ... 17

2.4 GRAPHQL ... 18

2.4.1 Principy ... 18

2.4.2 Zprávy ... 19

2.4.3 Dokumentace ... 19

3 NÁVRH SYSTÉMU ... 21

3.1 FUNKCIONÁLNÍ POŽADAVKY ... 22

3.2 KLIENTSKÁ ČÁST ... 23

3.2.1 Architektura ... 23

3.2.2 Technologie ... 24

3.3 SERVEROVÁ ČÁST ... 26

3.3.1 Architektura ... 26

3.3.2 Technologie ... 31

3.3.3 Databázový model ... 32

4 IMPLEMENTACE ... 34

4.1 KLIENTSKÁ ČÁST ... 34

4.2 SERVEROVÁ ČÁST ... 35

5 ZHODNOCENÍ ... 37

5.1 VÝKONNOST 37
5.2 PROPUSTNOST 38
5.3 FLEXIBILITA 38
5.4 TESTOVATELNOST 39

ZÁVĚR 40

LITERATURA 41

PŘÍLOHY 43

PŘÍLOHA A 43
Obsah CD 43
PŘÍLOHA B 44
Návod na spuštění 44
PŘÍLOHA C 46
Uživatelský manuál 46

1 Úvod

Webové technologie se od svého vzniku značně posunuly. Z uživatelského pohledu od statických webových stránek přes interaktivní až k SPA (Single Page Application) a PWA (Progressive Web Application) technologiím. Z vývojářského pohledu od jednoduchých monolitních systémů až ke komplexním distribuovaným systémům s cloudovou infrastrukturou. Jedním konceptem, který tomuto vývoji pomohl, je koncept webových služeb.

Webová služba je systém, který zprostředkovává nějakou funkcionalitu skrze webové technologie jako třeba HTTP (HyperText Transfer Protocol). Oproti tradičním informačním systémům, kde je komunikace zpravidla typu M2H (Machine To Human), se jedná o systém bez uživatelského rozhraní. U webových služeb se tedy jedná o komunikaci typu M2M (Machine To Machine). Informační systémy se tak mohou vytvořit čistě jako klientské aplikace využívající funkcionalit webových služeb, čímž se zkvalitňuje reakční doba a samotná plynulost aplikace. Také se tím centralizuje logika systému, což umožňuje znovuvyužitelnost pro více klientů, kteří zajišťují prezentaci dat.

S webovými službami se dnes můžeme setkat u mnoha webových systémů, které chtějí podpořit integraci se systémy jinými. Díky tomu lze systémy využívat mezi sebou a rozšiřovat funkcionalitu skrze jednoduché propojení. Pro realizaci webových služeb se využívají různé technologie pro různé věci, přičemž jedním z důležitých aspektů je samotná technologie pro komunikaci.

Tato práce se zaměřuje na dvě technologie, které se pro realizaci webových služeb a komunikaci s nimi aktivně používají. Jedná se o technologie REST (REpresentation State Transfer) a GraphQL. Motivací pro výběr těchto technologií a samotné práce byla má dosavadní zkušenost ve vývoji podnikových informačních systémů, která byla výhradně skrze REST technologii. Pod rukama se mi tak dostaly dobré i špatné návrhy webových služeb, přičemž ty špatné mnohdy pramenily z neznalosti fundamentálních principů a praktik, které tato technologie přináší. Prvním cílem je tedy sepsat principy a přístupy této technologie, které jsou pro správnou realizaci klíčové. Druhým cílem je sepsat a porovnat novou technologii GraphQL, která slibuje vyřešit některé problémy, jež se při používání technologie REST objevují. Přestože je GraphQL v několika službách již v produkčním prostředí, jedná se stále o mladou technologii a konzervativní podniky se jí bojí začít využívat. Při porovnání bych se tedy rád zaměřil na některé praktické otázky, které mohou snížit entropii a zjednodušit rozhodování, zdali je GraphQL tou správnou cestou.

Následující kapitola se zaměřuje na popis uvedených technologií. Další kapitola pak na návrh referenčního systému, následována kapitolou o implementaci a zhodnocení technologií. V závěru je zhodnocení výsledků práce a případné návrhy k dalšímu vývoji a zpracování.

2 Technologie

V této kapitole si představíme technologie, se kterými budeme v této práci dále pracovat. Nejdříve je popsána technologie HTTP (HyperText Transfer Protocol), na které staví zbylé technologie. Dále je popsána technologie RPC (Remote Procedure Call), na kterou již navazují technologie REST (REpresentation State Transfer) a GraphQL. Všechny tři technologie (RPC, REST a GraphQL) slouží k realizaci webových služeb, přičemž každá má svou vlastní formu a přístup.

2.1 Protokol HTTP

HTTP [1] je aplikační protokol pro přenos hypertextových dokumentů. Hypertextový dokument je forma strukturovaného nelineárního textu. Nelineárním je myšleno to, že obsahuje odkazy, čímž dochází k provázání dokumentů.

HTTP leží v rámci TCP/IP modelu na čtvrté, aplikační vrstvě. Jedná se tedy o komunikační protokol mezi dvěma procesy využívajícími pro komunikaci síťový přenos. Pro přenos dat se využívá spolehlivé spojení TCP (Transmission Control Protocol), které zajištuje tzv. best-effort delivery. Vynakládá tedy maximální snahu, ale nezaručuje úspěšnost doručení či kvalitu služby. Standardním portem pro HTTP je TCP/80, v případě šifrovaného spojení HTTPS skrze TLS je to TCP/443.

HTTP je textový protokol (text-based / plain text) [2], což znamená, že reprezentace dat komunikačního protokolu je v lidsky čitelné podobě. Lze tak navázat spojení přímo a vést konverzaci mezi člověkem a počítačem. To je rozdíl oproti binárním protokolům, kde je potřeba zajistit konkrétní způsob přenášení dat.

2.1.1 Komunikace

Komunikace pomocí HTTP protokolu probíhá v textové podobě, jedná se o typ dotaz-odpověď (request-response) v klient-server architektuře. Klient pošle zprávu typu dotaz (request), na kterou server odpoví zprávou typu odpověď (response).

Samotná zpráva pro dotaz začíná dotazovacím řádkem obsahující HTTP metodu, cestu ke zdroji serveru a verzi protokolu. Za dotazovací řádkou se nachází seznam hlaviček, které jsou od sebe oddělené koncem řádku ve windowsu stylu, tedy CRLF (Carriage Return & Line Feed). Hlavičky jsou nepovinné, krom jedné jediné. Tou je hlavička Host, která se používá pro routování ve webových serverech, typicky pomocí virtuálních hostů. Za seznamem hlaviček následuje prázdný řádek, který může následovat obsahem samotného těla zprávy. Na obrázku 2.1 lze vidět jednoduchou zprávu dotazující se na kořenový zdroj webového serveru metodou GET, tedy získání zdroje. Formát obsahu těla zprávy může být jakýkoli, pokud je textově reprezentovatelný. Jelikož často chceme posílat i binární soubory jako obrázky či dokumenty, byl pro tuto potřebu zaveden MIME type (Multipurpose

Internet Mail Extension). Jedná se o standard určený pro specifikaci typu a formátu dokumentu ve formě typ/subtyp. Samotná specifikace předdefinuje základní seznam typů a subtypů, společně s předpisem pro vlastní typ. V případě poslání binárních souborů jako obrázky a dokumenty se tak buď využívá textové překódování jako base64, nebo právě MIME type. Ten se posílá v HTTP hlavičce Content-Type, podle které klient dokáže pracovat s obsahem těla zprávy. MIME type se však nepoužívá pouze pro určení netextových formátů, používá se i pro určení textových serializačních formátů jako xml, json, csv, a jiné.

```
GET / HTTP/1.1
Host: www.example.com
```

Obrázek 2.1 ukázka zprávy typu dotaz [1]

Zpráva pro odpověď se skládá ze statusového řádku obsahující verzi protokolu, status kód a jeho popis. Popis slouží k ladícím účelům, neboť si málokdo pamatuje význam všech statusových kódů, nebo se využívají nestandardizované status kódy. Za statusovým řádkem se nachází seznam hlaviček následované obsahem těla zprávy. Platí zde stejná pravidla jako u zprávy typu dotaz, tedy že jsou hlavičky oddělené koncem řádku typu CRLF a za poslední hlavičkou je před obsahem těla zprávy prázdný řádek. Na obrázku 2.2 je ukázka možné odpovědi na předchozí ukázku dotazu. Zpracování dotazu skončilo úspěchem, což je reprezentováno status kódem 200, který má popis OK. Dále lze vidět hlavičky označující například délku zprávy a typ zprávy. Za hlavičkami následuje obsah těla zprávy, zde v HTML formátu.

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    <p>Hello World, this is a very simple HTML document.</p>
  </body>
</html>
```

Obrázek 2.2 ukázka zprávy typu odpověď [1]

2.1.2 Bezstavovost

HTTP je protokol bezstavový (stateless), takže si nepamatuje informace z jednotlivých přenosů. Jedná se o jistou formu vstupně/výstupného přenosu, zjednodušeně I/O (input/output). Případná stavovost se tak musí zajistit explicitně. K tomu slouží dva mechanizmy, sezení (sessions) a cookies. Cookies je malý objem dat, který serverová aplikace může uložit na straně klienta. Server zasílá HTTP hlavičku Set-Cookie, kterou klient rozparsuje a uloží. Maximální velikost je většinou omezena, standardně na 4KB. Klient pak tato uložená data posílá zpět serveru při každém dotazu v HTTP hlavičce Cookie, čímž je umožněno zapamatování stavu mezi dotazy. Jelikož však cookies leží na straně klienta, může se jednat o bezpečnostní riziko. K zabezpečení stavu mezi dotazy pak slouží sezení (session). Sezení je kontext klienta na straně serveru, kam se ukládá stav mezi požadavky. Samotné místo uložení záleží na realizaci, může se tak například jednat o souborové, databázové, nebo jen paměťové uložiště. Při vytvoření stavu na straně serveru se vygeneruje jednoznačný identifikátor SESSION ID, který se pošle klientovi jako cookie. Klient pak tento identifikátor zasílá serveru při každém dalším dotazu, čímž se na straně serveru identifikuje klientův kontext a tím umožní přístup ke stavu. SESSION ID je tak velice důležitá a soukromá informace, mnohdy se jedná o identifikace přihlášeného uživatele, skrze který klient komunikuje se systémem jménem uživatele. Zabezpečení tohoto identifikátoru před krádeží má tedy vysokou prioritu, jinak by mohlo docházet k zneužití přístupu, tzv. session hijacking.

2.1.3 Bezpečnost

Jelikož není HTTP protokol šifrovaný, je náchylný na různé kybernetické útoky. To bylo samotnou motivací pro vznik HTTPS, kde se přidává speciální šifrovací vrstva TLS (dříve SSL) využívající asymetrickou kryptografii. Pomocí té je ověřena identita webového serveru a volitelně i klienta. Po ověření identity následuje dohoda na klíči pro symetrické šifrování samotné komunikace. Pro TLS se využívá PKI (Public Key Infrastructure) a certifikáty X.509.

Přestože je přenos s HTTP protokolem bez šifrovaného spojení a užívání tedy nebezpečné, může být značně náročné vynutit užívání HTTPS. Většina serverů to řeší skrze přesměrování, kdy příchozí požadavky protokolem HTTP přesměrují na stejnou adresu s protokolem HTTPS. Webové prohlížeče většinou začínají komunikaci nešifrovaným spojením, pokud se explicitně neurčí schéma. To kvůli zpětné kompatibilitě, neboť webový prohlížeč nemůže dopředu vědět, jestli koncová služba podporuje šifrovaný přenos skrze HTTPS. Toto je však stále velké bezpečnostní riziko kvůli MITM (Man In The Middle), kdy útočník může poslouchat na prvotní nešifrované komunikaci a překládat samotné dotazy, čímž se tváří jako koncová služba. Existuje však mechanismus pro vynucení šifrovaného spojení zvaný HSTS (HTTP Strict Transport Security). Ten funguje tak, že server pošle speciální HTTP hlavičku, kterou informuje klienta o tom, že vždy vyžaduje HTTPS. Pokud by uživatel využívající daného klienta, typicky webový prohlížeč, chtěl přistoupit na stránku skrze nešifrované spojení, klient provede automatické interní přesměrování.

V rámci zabezpečení cookie se přinejmenším nastavují příznaky Secure a Http-only. Secure příznak říká klientovi, že nesmí posílat cookie skrze nešifrované spojení. Tato ochrana je velice důležitá v případě nevyužití HSTS mechanizmu, neboť se tak citlivé informace nedostanou k útočníkovi. Druhý příznak, příznak Http-only, říká klientovi, že je cookie dostupná pouze pro přenos HTTP dotazů. Tím se zabraňuje možnosti krádeže skrze kód na klientské straně, v prohlížečích typicky skrze javascript.

2.1.4 Identifikace zdrojů

K identifikaci zdrojů se využívá URI (Uniform Resource Identifier), tedy sekvence znaků jednoznačně identifikující zdroj v internetu. URI může být dále klasifikována jako lokátor, jméno, nebo obojí. URI tak může nabývat dalších dvou tvarů: URN (Uniform Resource Name) a URL (Uniform Resource Location) [3]. URN je podmnožina všech URI, která identifikuje zdroj podle jména v konkrétním jmenném prostoru. Tento způsob komunikace může být využit k označení zdroje bez vyjádření lokace či způsobu přístupu. Například cloudová služba AWS od Amazonu značně využívá identifikaci zdrojů pomocí jmen, nazývají to ARN (Amazon Resource Name). Identifikují tímto způsobem jednotlivé služby uvnitř AWS infrastrkutry. URL značí podmnožinu všech URI, která kromě identifikace zdroje určuje i způsob přístupu. U statického obsahu se typicky jedná o přímou lokaci souborů ve webovém serveru. U dynamického obsahu se typicky jedná o způsob aplikačního routování a identifikaci příslušné řídicí obsluhy, která dynamický obsah generuje.

Generická syntaxe URI se skládá z hierarchické sekvence pěti komponent: schéma, autorita, cesta, dotaz a fragment.

```
URI = scheme:[//authority]path[?query][#fragment]
```

```
authority = [userinfo@]host[:port]
```

Mezi známá schémata patří například http, https, ftp, mailto, file, data. Autorita se skládá z uživatelské informace, hosta a případně portu. Samotná uživatelská informace může být buď token, nebo třeba přihlašovací jméno a heslo oddělené dvojtečkou. Cesta obsahuje data, běžně organizovaná v hierarchické podobě, která identifikují zdroj v rámci nějaké hierarchické struktury. Jednotlivé části, segmenty, jsou odděleny lomítkem. Volitelná komponenta dotazu předcházená znakem otazníku obsahuje nehierarchickou část dat. Poslední komponentou identifikátoru je fragment, který umožňuje nepřímou identifikaci sekundárního zdroje referovaného zdrojem primárním. V HTML se tak často zaměřují elementy na stránce, díky čemuž prohlížeč ví, kam má automaticky posunout stránku.

2.1.5 Verze

HTTP protokol byl představen v roce 1991 a od té doby prošel dvěma revizemi. První verzí bylo HTTP/1.0, následované verzí HTTP/1.1. Verze HTTP/1.1 přinesla značná vylepšení. Například perzistentní TCP spojení, kde se po první odpovědi neukončuje přenos, jako tomu bylo u předchozí verze. Tím se snižuje režie za navázání spojení, která je způsobena trojcestným handshakingem (three-way handshake). Dotazy se tak mohou zasílat jeden za druhým skrze stejné spojení, tzv. *request pipelining*. Dále nové dotazovací metody včetně OPTIONS metody pro zabezpečení domén a nové návratové status kódy. Poté přišla verze HTTP/2.0, na které se podíleli velcí průmysloví giganti pod vedením společnosti Google. Verze HTTP/2.0 je derivací experimentálního protokolu SPDY, který vyvinul Google. HTTP/2.0 přinesla další významné změny, které redukovaly problémy, kterým vývojáři webových aplikací čelili. Například přechod z textového protokolu na binární. Tím se sice zhoršilo ladění komunikace, ale samotné posílání dat a parsování se značně zrychlilo. Dále *request multiplexing*, který nahrazuje *request pipelining* z předchozí verze. Díky tomu lze vést paralelně několik komunikací přes jedno TCP spojení. Také přibyla komprimace HTTP hlaviček pomocí HPACK specifikace. Klient i server udržují seznam hlaviček používaných z předešlých dotazů. Poslední významnou změnou je push mechanizmus, kdy může server klientovi zaslat dodatečné relevantní zdroje bez obdržení dotazu. To funguje skvěle na zdroje webových stránek jako styly, skripty a obrázky. Prohlížeč zašle dotaz na samotný dokument, který mu server pošle společně s použitými zdroji na stránce. Tím se ušetří komunikace, která by byla stejně potřeba.

2.2 RPC

RPC (Remote Procedure Call) je koncept, jehož cílem je provést aplikační logiku v jiném adresovém prostoru, typicky na jiném stroji. Samotné spouštění aplikační logiky závisí na způsobu komunikace, vždy je ale komunikaci typu dotaz-odpověď. Pro komunikaci se nejčastěji, ale ne výlučně, využívá HTTP. Pro ukázku může být další známý komunikační protokol, WebSocket, který realizuje full-duplex komunikaci s binárním přenosem a nejčastěji se používá pro real-time synchronizaci dat ve webových aplikacích. Jedná se o alternativní přístup oproti opakovanému dotazování, tzv. pollingu, který může zbytečně snižovat propustnost a zahlcovat linku.

V případě této práce se zaměříme na RPC využívající HTTP protokol pro komunikaci. V takovém případě může být určení operace v podobě URI, nebo samotného obsahu těla zprávy. Konkrétní realizací je například XML-RPC, JSON-RPC, či gRPC, lišící se pouze ve formátu dat. Komunikace může vypadat následovně.

Na obrázku 2.3 lze vidět typický dotaz na webovou službu, zde konkrétně s dotazem pro získání názvů státu s parametrem 40 jako 32bitové celé číslo. Význam tohoto parametru není znám, může se jednat o limit délky názvu, o limit délky seznamu, nebo o něco zcela jiného. Na obrázku 2.4 je znázorněna odpověď webové služby, která vrací jeden název jako textový řetězec.

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
        <value><i4>40</i4></value>
    </param>
  </params>
</methodCall>
```

Obrázek 2.3 ukázka dotazu v XML-RPC [4]

```xml
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

Obrázek 2.4 ukázka odpovědi v XML-RPC [4]

2.2.1 SOAP

Jednou z derivací protokolu RPC je protokol SOAP (Simple Object Access Protocol), který se dnes stále značně používá. Ten se vyvinul z protokolu XML-RPC, tedy RPC využívající pro komunikaci HTTP s XML formátem pro obsah těla zprávy. SOAP protokol přináší nové funkcionality jako uživatelem definované datové typy, možnost definice adresáta, kontrolu zpracování podle konkrétní zprávy a další. SOAP má větší podporu v podnikových platformách jako .NET a JEE (Java Enterprise Edition) díky specifikaci WSDL (Web Service Description Language). Tato specifikace popisuje způsob popisu webových služeb pomocí XML dokumentu, díky kterému lze vygenerovat integrační části kódu v dané platformě.

2.3 REST

REST je architektonický styl pro distribuované hypermediální systémy. Vznikl jako disertační práce Roye Thomase Fieldinga [5], kde se snažil popsat nástroj pro pochopení architektury webových aplikací. Výsledkem byl vlastní architektonický styl a způsob jeho využití při návrhu a vývoji moderního webu.

2.3.1 Principy

Architektonický styl definuje rodinu systémů ve smyslu vzoru a strukturální organizace a slovník komponent a konektorů s omezeními, jak je lze kombinovat [6]. Roye T. Fielding popsal REST architekturu podle 6 omezení, principů, která byla z jeho úhlu pohledu klíčová.

Client-server

Díky separaci uživatelského rozhraní od datového úložiště a aplikační logiky nad daty dochází ke zvýšení přenositelnosti uživatelského rozhraní na více platforem. Centralizace aplikační a datové logiky zjednodušuje distribuce změn a škálovatelnost serverové části bez nutnosti zásahu do klientských částí.

Rozdělení zodpovědnosti také umožňuje nezávislou evoluci jednotlivých částí, čímž se může značně urychlit samotný vývoj při správné koordinaci.

Stateless

Každý dotaz na server od klienta musí obsahovat veškeré informace potřebné k pochopení dotazu a nesmí využívat výhod uloženého kontextu na straně serveru. Toto omezení přidává vlastnosti jako viditelnost, spolehlivost a škálovatelnost. Viditelnost je zlepšena, jelikož monitorovací systém může pro pochopení podstaty dotazu nahlížet pouze na samotný dotaz. Spolehlivost je zlepšena, jelikož je jednodušší obnova z částečné chyby. Škálovatelnost je zlepšena, jelikož se nemusí ukládat stav mezi dotazy, a serverová část tak může rychleji uvolňovat zdroje.

Toto rozhodnutí s sebou přináší jisté důsledky. Jedním negativním je, že se může snížit výkonnost sítě, jelikož se zvyšuje množství repetitivních dat, která se v dotazech posílají. Také se redukuje kontrola konzistentního aplikačního chování, jelikož aplikace začíná být závislá na správné implementace sémantiky napříč několika klientskými verzemi.

Cacheable

Server může explicitně označit odpověď na dotaz jako kešovatelnou, čímž dává klientovi právo znovuvyužít data odpovědi pro stejný pozdější dotaz. Toto omezení vyžaduje, aby data uvnitř odpovědi na dotaz byla implicitně nebo explicitně označena jako kešovatelná, nebo nekešovatelná. Výhodou je zvýšení výkonnosti skrze snížení latence a serverové zátěže. Negativním důsledkem je snížení spolehlivosti. Data uvnitř cache se mohou značně lišit od dat, která by byla získána při přímém dotazu na server.

Samotné kešování může probíhat buď přímo na klientské části (local cache) nebo na serverové části (proxy cache). Někteří klienti, například webové prohlížeče, kešují odpovědi na dotazy zaslané metodou GET implicitně. Na to je potřeba myslet při nastavování serverové části, aby nedocházelo k nečekaným problémům. Odpovědi na dotazy zaslané většinou zbylých metod jsou nekešovatelné a odpovědi na dotazy zaslané metodou POST jsou kešovatelné jedině s explicitním označením [7].

Uniform interface

Díky využití generalizace skrze rozhraní dochází ke zjednodušení a zpřehlednění interakce. Implementace je oddělena od definice služby, kterou poskytuje, čímž se podporuje nezávislá evoluce. Samotná generalizace je založena na čtyřech aspektech. Prvním je identifikace zdroje. Každý zdroj v dotazu je identifikován pomocí URI (Uniform Resource Identifier), přičemž reprezentace zdroje v odpovědi je odlišná od jeho interní reprezentace v uložišti Druhým aspektem je manipulace zdroje. Kombinací URI a HTTP metod se popisuje operace, kterou má server nad zdrojem provést. Třetím aspektem jsou sebepopisující zprávy. Každá zpráva obsahuje dostatek informací pro její zpracování,

například MIME type. Čtvrtým a posledním aspektem je HATEOS, tedy oddělení klienta od specifické struktury URI pro jednodušší správu struktury.

Layered system

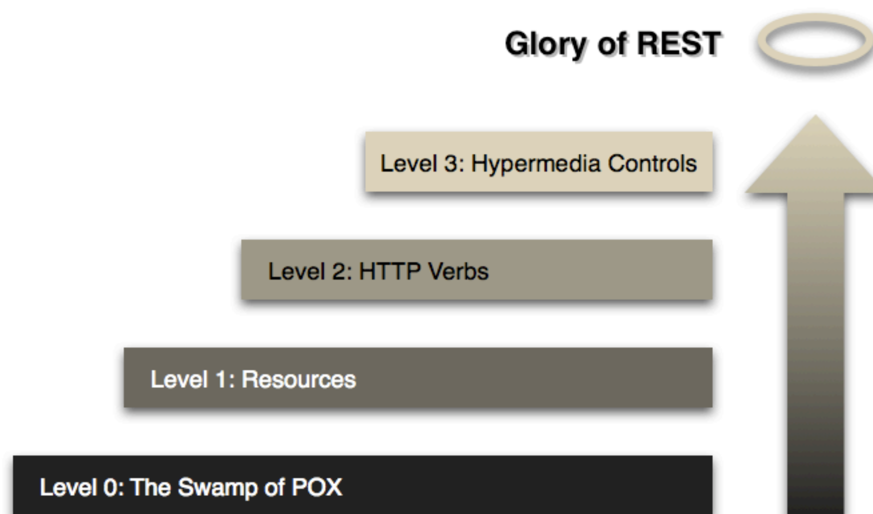
Klient není schopen určit, zdali komunikuje s koncovým serverem nebo nějaký jiným uzlem v infrastruktuře, což také zjednodušuje horizontální škálovatelnost. Do infrastruktury se tak dá zavést load balancing pro řízení provozu podle zátěže.

Code on demand

Ve většině případů vrací server pouze serializovaná data, typicky v XML nebo JSON formátu. V některých případech ale může server vrátit s daty i spustitelný kód pro podporu aplikace.

2.3.2 Richardson maturity model

Jedná se o model popisující vyspělost navrhnuté webové služby. Každá další úroveň přináší nové výhody, které jsou důležité a značně korelují se samotnými principy RESTu [8].



Obrázek 2.5 Richardson maturity model [8]

Level 0

Tato úroveň je jakýmsi výchozím bodem, ve kterém se nachází všechny webové služby využívající HTTP protokol pro přenos dat. Webová služba na této úrovni je dostupná pod jedním URI a využívá jen základní HTTP metody GET a POST.

Tento přístup se dříve značně používal skrze SOAP protokol (Simple Object Access Protocol), případně starší protokol XML-RPC posílající POX (Plain Old XML), XML dokumenty označující, která akce se má stát a s jakými parametry.

Level 1

Webová služba na této úrovni rozděluje funkcionalitu služby na dílčí částí, tzv. zdroje (resources), které mají svou vlastní unikátní URI adresu. Díky tomu je rozhraní webové služby značně čitelnější, neboť lze studovat a testovat jednotlivé zdroje samostatně. Jedná se o jistou formu dekompozice, kdy se celá funkcionalita služby v podobě černé skříňky (black box) rozdělí na několik zdrojů. Tím se zpřehledňuje komunikace, dokumentace, a samotná údržba. Jedná o podobný koncept jako identita objektů v objektovém paradigmatu. Místo volání nějaké funkce a předávání argumentů, voláme metodu na konkrétním objektu a poskytujeme argumenty jako dodatečné informace [8].

Služba stále musí nějak odlišit typ operace, která se se zdrojem má provést. Jelikož se stále používají pouze základní HTTP metody, operace se odlišují skrze obsah zpráv.

Level 2

Webová služba na této úrovni již využívá celý repertoár HTTP metod pro odlišení operací. Jestliže nám URI umožňuje pojmenovat a identifikovat každý zdroj v systému, HTTP metoda nám pak umožňuje odlišit jednotlivé CRUD (Create, Read, Update, Delete) operace nad zdroji. Největším přínosem je odlišení bezpečných operací, které nemění stav systému, od nebezpečných operací, které jej mění.

Toto rozlišování můžeme vidět například v bezpečnostním mechanismu CORS (Cross Origin Resource Sharing) [9], který v prohlížečích kontroluje přístup mezi doménami. CORS rozděluje zprávy na jednoduché, které za určitých podmínek nevynucují explicitního dotazu, tzv. preflight request. Preflight request je speciální dotaz zaslaný HTTP metodou OPTIONS, jejíž odpověď označuje seznam metod a domén, které lze nad zdrojem posílat.

Metody se odlišují také v idempotentnosti, neboli takové vlastnosti operaci, že vícenásobné provádění operace má vždy stejný výsledný efekt.

Level 3

Nejvyšší vrstva představuje webové služby, které využívají HATEOS (Hypertext As The Engine Of Application State) [10], kde server krom dat vrací i odkazy ke zdrojům. Pokud klient nemusí sám skládat URI, ale využívá odkazy, které mu zasílá samotný server, může tak server změnit URI schéma bez nutnosti zásahu do klienta. Přeorganizace zdrojů se sice nestává často, přesto je ale součástí návrhu systému nutné navrhnout takovou abstrakci systému, která bude reflektovat směr evoluce daného systému a redukovat identifikovaná rizika. Stále ale zůstává fakt, že pokud se značně změní pojmenování a struktura zdrojů, je vhodné upravit i strukturu samotných dat, čímž může být zásah do klientů stejně nevyhnutelný.

V případě využití HATEOS principu volá klient kořenový zdroj, na který dostává seznam dalších zdrojů a možných operací nad ním. Při získání konkrétních zdrojů pak dostává odkazy na další zdroje a operace. Značně se reflektuje samotná interakce uživatele s klientskou aplikací.

2.3.3 Pojmenování zdrojů

Při psaní kódu je jedním z důležitých faktorů jeho čitelnost, která určuje jeho udržitelnost a rozšiřitelnost. Faktorů určujících čitelnost kódu je několik, jedním z nejdůležitějších je však vhodné pojmenování. Stejně jako musí být dobře pojmenovány objekty, metody a proměnné v kódu, musí být vhodně pojmenovány entity v databázi a samotné zdroje v end-pointech. Kromě nalezení vhodného slova je ještě otázka ohledně uplatňování jednotného nebo množného čísla. V konečném důsledku jde hlavně o konzistenci, ale na samotném počátku je dobré si zvolit směr, který bude dávat vývojovému týmu větší smysl. U databází je samotná otázka čísla celkem zapeklitá, neboť záleží na úhlu pohledu a způsobu užívání. U REST zdrojů je však tato otázka lehčí, neboť používáme HTTP metody pro určení operace, a celkové složení se až na pár výjimek zdá přirozené. Preferovanou cestou je tak množné číslo u kolekcí a jednotné číslo u dokumentů.

2.3.4 Dokumentace

Dokumentace RESTových služeb je nejčastěji prováděná jazyky OpenAPI, API Blueprint či RAML. Uvedené jazyky mají okolo sebe značně rozšířený ekosystém nástrojů a služeb pro jejich podporu, které usnadňují vývoj. Například služba Apiary, která slouží nejen k dokumentaci webových služeb, ale umožňuje také mockování či inspekci provozu. Případně tvrdý konkurent Swagger. Nevýhoda je však v tom, že dokumentace musí být externí, čímž vzniká vysoké riziko neaktuálnosti.

BDD (behavior driven development) přístup k vývoji přišel s konceptem živé dokumentace (living documentation), kde se nějakým doménovým jazykem (ubiquitous language) popíší business požadavky ve formě testovacích scénářů, které lze spustit. Tím lze zjistit případnou odchylku a následně upravit buď dokumentaci, nebo kód. K tomu slouží nástroje jako cucumber a jazyk gherkin známy jako given-when-then. Pro REST služby existuje nástroj Dredd, který pracuje podobným způsobem přímo s výše popsanými jazyky pro popis RESTových služeb.

2.3.5 Verzování

Verzování je u REST docela rozsáhlé téma, neboť dochází k velkému sporu mezi pragmatičností a dogmatičností. Samotná potřeba verzování je ale běžná, nejčastěji kvůli změně schématu a udržování zpětné kompatibility. Obecně existují 4 základní přístupy [11]:

- URI parametr
- Query parametr
- Vlastní hlavička (například Accept-version)
- Vlastní MIME type a hlavička Accept (například application/vnd.example.v1+json)

Nejběžnějším způsobem verzování je URI parametr, neboť se jedná o nejjednodušší způsob. Otázka vyvstává ohledně zdrojů při povyšování verzí. Zduplikují se všechny zdroje, nebo vzniknou jen přírůstky? Samotná myšlenka duplicity se rozchází s principem Uniform interface, který říká, že každý zdroj by měl mít pouze jeden způsob identifikace. Někteří vývojáři aplikuji evoluční přístup, kdy se schéma nikdy neupravuje, jen rozšiřuje. Je zde potřeba zvážit další aspekty, např. jak dlouho a kolik verzí se zpětně udržuje.

Query parametr se moc nepoužívá, neboť mají webové prohlížeče omezení na maximální délku. Ta se liší prohlížeč od prohlížeče. Až na pár výjimek jako Internet Explorer je to ale okolo 64k znaků.

Vlastní hlavička je způsob, který není tak agresivní a je lehce modifikovatelný, stejně jako vlastní MIME type. U vlastního MIME typu je problém, že může ovlivnit některé knihovny, které nabízí implicitní mechanizmus deserializace. Zde záleží na konfigurovatelnosti knihovny. Stejně tak pro vlastní HTTP hlavičku, jelikož není standardizovaná napříč dodavateli, jedná se o vlastní řešení a značení. Každý nový klient se tak musí modifikovat kvůli integraci, což může narušit znovuvyužitelnost klientského kódu a vyžadovat customizaci. Dříve se vlastní hlavičky označovaly prefixem **X-**. Znamenalo to experimental nebo extension, ale to se později označilo jako zastaralé [12]. Když se totiž standardizovala nějaká nová experimentální hlavička, tak byl značný problém prefix odstranit, protože se musel upravit každý nástroj či knihovna využívající hlavičku.

2.4 GraphQL

GraphQL [13] je dotazovací jazyk vytvořený společností Facebook pro popis možností a požadavků na datové modely v aplikacích s architekturou klient-server. Skrze GraphQL se vytváří jednotné rozhraní nezávislé na konkrétním jazyce či platformě. Motivací pro vznik takového jazyka bylo umožnit klientům specifikovat, jaká data přesně chtějí. Tím nedochází k posílání zbytečných dat, ani k několikanásobnému dotazování na server.

2.4.1 Principy

Stejně jako REST je Graphql založené na několika principech, která usměrňují samotnou specifikaci a tím konkrétní implementace. Tyto principy mají podobný význam, jako omezení u architektonických stylů. Jen nejsou tak rigorózní, spíše orientační a navádějící.

Hierarchical

Většina dnešního vývoje obsahuje vytváření a manipulaci s hierarchiemi pohledů. K dosažení kongruence s takovými aplikacemi je GraphQL dotaz také hierarchicky strukturovaný. Dotaz je tvarovaný jako data, která služba vrací.

Product-centric

GraphQL je řízen požadavky pohledů a front-end vývojáři, kteří je píší. Začíná tedy jejich způsobem myšlení a požadavky a vytváří jazyk a běhové prostředí potřebné k jejich umožnění.

Strong-typing

Každý GraphQL server specifikuje aplikačně specifický typový systém. Dotazy jsou pak zpracovány v kontextu tohoto typového systému. S příchozím dotazem tak nástroje mohou zajistit, že je syntakticky korektní a validní vůči typovému systému, a server tak může poskytnout jistou záruku o tvaru a podstatě odpovědi.

Client-specified queries

Skrze typový systém publikuje GraphQL server možnosti, které mohou klienti využívat. Je pak zodpovědností klienta specifikovat, jak přesně chce využít publikované možnosti.

Introspectivve

Typový systém GraphQL serveru musí být sám dotazovatelný GraphQL jazykem. To vytváří platformu pro vytváření nástrojů a knihoven. Lze se tak například GraphQL zeptat, jaké dotazy podporuje a jak a vypadá typové schéma.

2.4.2 Zprávy

GraphQL rozlišuje zprávy na dotazy a mutace. Dotazy jsou zprávy pro získávání informací bez vedlejších efektů. Mohou být i parametrizované, například při filtrování a stránkování. Mutace jsou zprávy pro změnu stavu systému, které typicky vrací změněné objekty, jejichž strukturu lze opět specifikovat.

2.4.3 Dokumentace

GraphQL poskytuje SDL (Schema Definition Language) [14], vlastní DSL (Domain Specific Langauge) pro definici typů a možností, které služba poskytuje. Tento jazyk slouží jako dokumentace služby, ke které se klient připojuje, Při využití speciálního jazyka pro popis schématu dochází k odstínění od konkrétní platformy, čímž se zvyšuje přenositelnost. Jedná se o stejný koncept jako v případě RESTu, jen je zde lepší integrace s nástroji, které automaticky kontrolují správnost služby vůči schématu.

Tomuto přístupu se říká schema-first [15], kde je schéma bráno jako zdroj pravdy a systém se vůči němu konfiguruje a případně kontroluje. Opačným přístupem je code-first, kde se schéma definuje pomocí kódu v konkrétní platformě. Přestože se může zdát přístup schema-first na první pohled lepší kvůli abstrakci platformy a kompaktnějšímu zápisu, přináší s sebou značné problémy, které mohou u velkých systémů převážit přínosy. Jedním problémem je třeba možná nekonzistence mezi definicí

schématu a resolvery, dalším je pak modularizace schématu a redundance konstrukcí jako třeba paginování.

SDL umožňuje psaní vlastních strukturovaných typů složených buď ze skalárních, nebo jiných vlastních struktur. Mezi skalární typy patří základní známé programátorské typy jako

- Celé číslo (Int), zde konkrétně znaménkové 32bitové číslo
- Desetinné číslo (Float), znaménkové číslo s plovoucí čárkou s double přesností
- Textový řetězec (String), což je sekvence UTF-8 znaků
- Pravdivostní hodnota (Boolean), která nabývá hodnoty true/false (pravda/nepravda)
- ID, unikátní identifikátor, který se používá primární při cechování

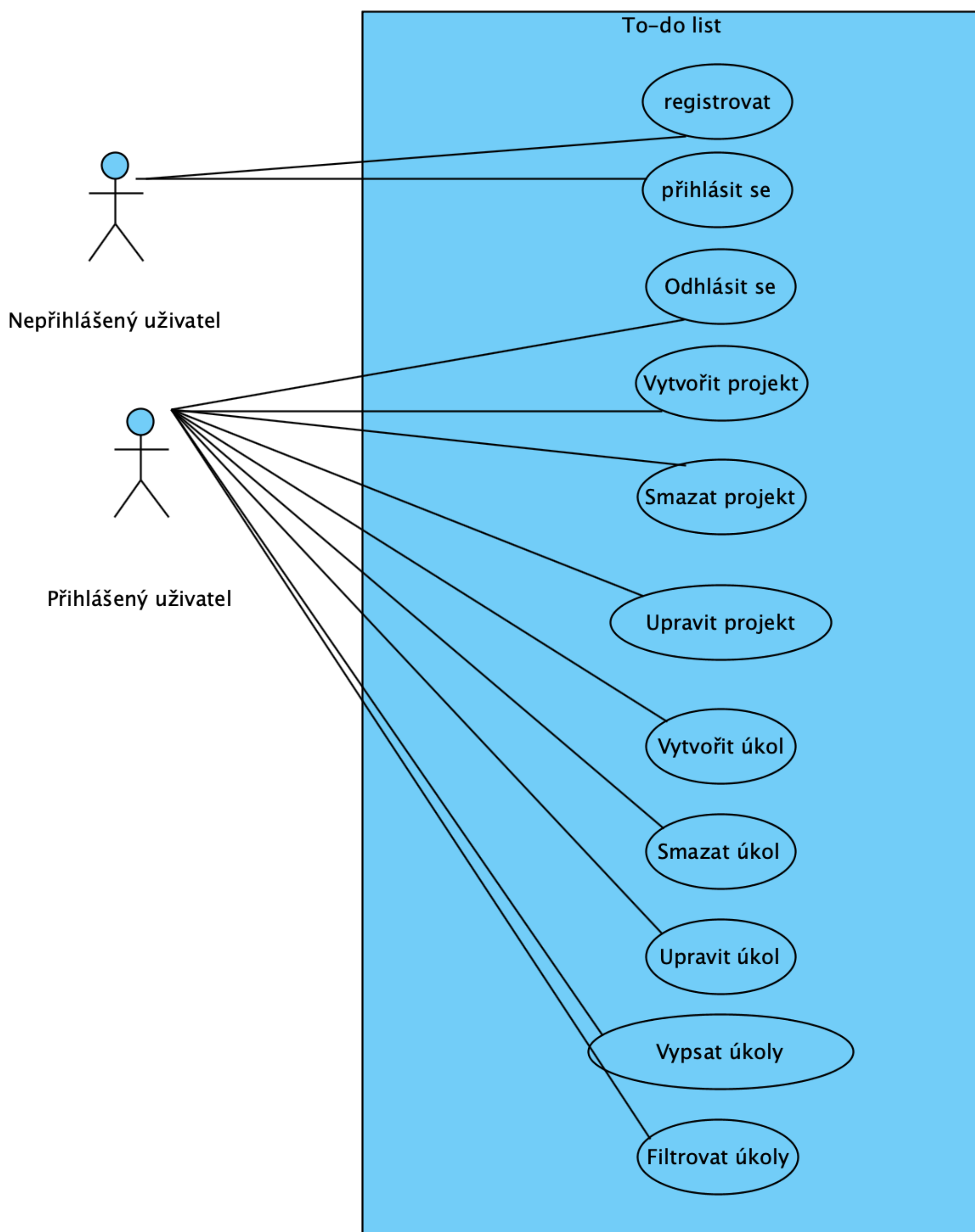
Každý typ, ať už vlastní strukturovaný či skalární, může být označen jako nenulový pomocí vykřičníku. Také je možné zapisovat pole typů skrze hranaté závorky. Je možné i přiložit dodatečné informace v podobě direktiv, jejich interpretace však bývá často závislá na konkrétní implementaci. To se však může hodit například pro mockování či testování.

3 Návrh systému

V této kapitole se zaměříme na návrh systému, který budeme realizovat technologiemi popsanými výše. Návrh systému popisuje jak funkcionální požadavky, tak architekturu a technologie.

Referenčním systémem této práce je To-do list, systém pro evidování a organizaci úkolů. Realizace bude v rozsahu MVP (Minimum Viable Product), jedná se tedy o rozsah značně minimalizovaný na to nejdůležitější. Motivací pro aplikaci je vytvoření digitálního nástroje, kterým lze aplikovat systém zvaný GTD (Getting Things Done). Tento systém se hojně využívá v manažerských rolích, kde je vysoká potřeba po sebeorganizaci.

3.1 Funkcionální požadavky



Obrázek 3.1 Funkční požadavky skrze UML Use Case Diagram

Proces registrace je jednokrokový, bez dodatečného ověření identity uživatele. Komplexnější realizace procesu by neměla přímý vliv na způsob komunikace, pouze na business logiku. V rámci zajištění bezpečnosti dat může uživatel manipulovat pouze se svými úkoly a projekty. Systém neumožňuje kolaboraci více uživatelů. Projekty nelze zanořovat, aplikace umožňuje jen jednu úroveň abstrakce, do

které se zařazují úkoly. Toto omezení je primárně z grafického rozhraní, kde se těžko styluje taková struktura. Zobrazení úkolů je bez stránkování, nepředpokládá se vysoká paměťová náročnost.

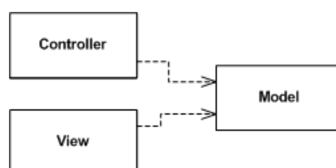
3.2 Klientská část

Uživatelské rozhraní je realizováno jako webová aplikace běžící v prohlížeči, konkrétně jako SPA (Single Page Application) [16]. SPA je typ aplikace, která načte HTML stránku a dynamicky ji upravuje, zatímco uživatel interaguje s aplikací. Oproti tradičnímu stylu MPA (Multi-Page Application) tak nedochází k opakovanému načítání celých stránek ze serveru, Načítají se pouze data, o jejichž vykreslení se stará sama klientská část. Získání SPA je typicky ještě extrahováno do jiného webového serveru, aby se snížila zátěž a zvýšila propustnost webové služby. Jedná se tedy o klienta samostatně žijícího, s vlastním release cyklem, konzumujícího data webové služby.

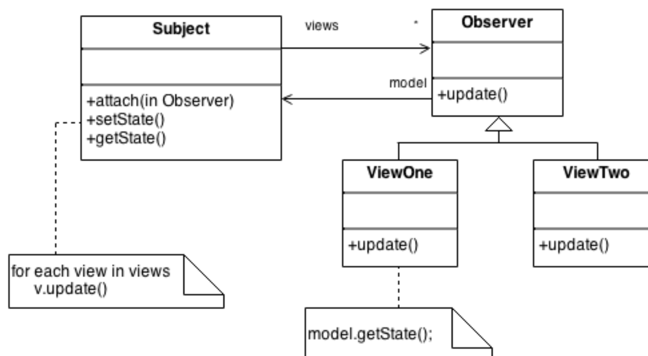
Obrázek 3.2 životní cyklus SPA a MPA [16]

3.2.1 Architektura

Klientská část využívá architekturu MVC (Model View Controller) [17], ve které se odděluje aplikační logika a správa dat od uživatelského rozhraní. Uživatel skrze interakci s uživatelským rozhraním vyvolá událost, na kterou reaguje nějaký kontroler. Ten převezme vstup a zajistí jeho zpracování skrze volání operací na modelech, kde je zapouzdřena business logika a data. Při změně dat model informuje příslušné pohledy, čímž dojde k překreslení. Notifikace o změnách je realizována skrze návrhový vzor observer. V něm se posluchači registrují subjektu, který je následně notifikuje o případných změnách.



Obrázek 3.3 MVC class diagram [17]



Obrázek 3.4 Observer pattern class diagram [18]

3.2.2 Technologie

Klientská část je realizována pomocí **React** technologie, která byla vytvořena společností Facebook právě pro vývoj SPA. Vytváří virtuální DOM (Document Object Model) a stará se o životní cyklus jednotlivých komponent, komunikaci mezi nimi a překreslování skutečné DOM struktury podle změn v DOMu virtuálním. Samotným programovacím jazykem je ES6 (ECMAscript) s rozšířením JSX, které usnadňuje zápis React komponent.

Dále je využita knihovna **Redux**, která zajišťuje správu stavu aplikace. Je realizována architekturou Event Sourcing [19], kde se změny stavu ukládají jako sekvence událostí. Díky tomu se lze nad událostmi dotazovat a rekonstruovat předchozí stav, čímž je jednodušší replikace a investigace nějakého problému. Tato knihovna také realizuje Observer pattern, kdy při změně stavu upozorní jednotlivé grafické komponenty o změně dat a zajistí tak jejich překreslení.

Pro práci se stavem je využit selector pattern, který schovává strukturu stavu za API. Díky tomu je při případné změně struktury stavu nutná úprava na jednom místě. Také je snadnější práce s normalizovaným stavem. Někdy však může docházet ke zbytečnému překreslování komponent kvůli funkcím, které operují nad stavem a vrací nové reference. Redux při změně stavu vyhodnocuje, zdali se změnili objekty, které daná komponenty využívá, aby nevynucoval zbytečně překreslování. Pokud při získání objektů dochází k vytvoření objektů nových skrze nějakou operaci (například filtrování a mapování), může Redux zbytečně vynucovat překreslení. Pro odstranění tohoto problému se používá knihovna reselect, která využívá kompozici a vytváří tzv. memoized functions, které si pamatují stav. K vytvoření nových objektů skrze nějakou operaci tak dochází pouze v případě, že se změnily samotné objekty stavu.

Dále je využita knihovna **immutable-js**, která zajišťuje neměnnost datových struktur. Velkým problémem mnoha systému jsou nepředvídatelnost a vedlejší efekty. Jako prevence se používají

neměnné struktury, které při změně stavu vrací objekty nové, místo úpravy aktuálních. Redux dokonce vytváření nových objektů při změně vyžaduje, protože používá shallow copy pro zjištění změn stavu. Navíc je také potřeba zachovat historii událostí, aby nedocházelo ke zpětnému přepisování.

Pro zpracování asynchronních volání webové služby je k dispozici několik způsobů. Nejjednodušším řešením je využití **Promise API** společně s **redux-thunk** nebo **redux-promise** knihovnou. Promise API snížilo výskyt callback hell díky fluent API. Dnes se nejvíce používá knihovna **redux-saga**, která umožňuje zápis pomocí javascriptových generátorů. To zpřehledňuje logiku ve funkcích vytvářejících akce (action creators) a zlepšuje testovatelnost. Kvůli jednoduchosti realizovaného systému je však použita knihovna redux-thunk společně s Promise API organizovaných podle konceptu **redux ducks**.

Pro stylování je využit preprocesor **SASS**, který zjednodušuje vývoj a udržování kaskádových stylů (CSS). Výhodou preprocesorů jako SASS nebo LESS je rozšířenější syntaxe a možnosti jazyka. Lze vytvářet proměnné, zanořovat styly, nebo také dědit a mixovat styly. Preprocesor typicky zápis ve svém jazyce transpiluje do jazyka CSS, který má zaručenou podporu v prohlížečích.

Pro správu závislostí a automatizaci je využit CLI (Command Line Interface) nástroj npm, který zajišťuje správu závislostí a vytváření spustitelných příkazů. Npm má vlastní cloudový registr a webové rozhraní, kde lze vyhledávat a procházet publikované balíčky. Npm obsahuje další funkcionality jako třeba audit závislostí, který kontroluje zjištěné zranitelnosti, nebo uzamykání závislostí pro opětovnou instalaci s přesnými verzemi.

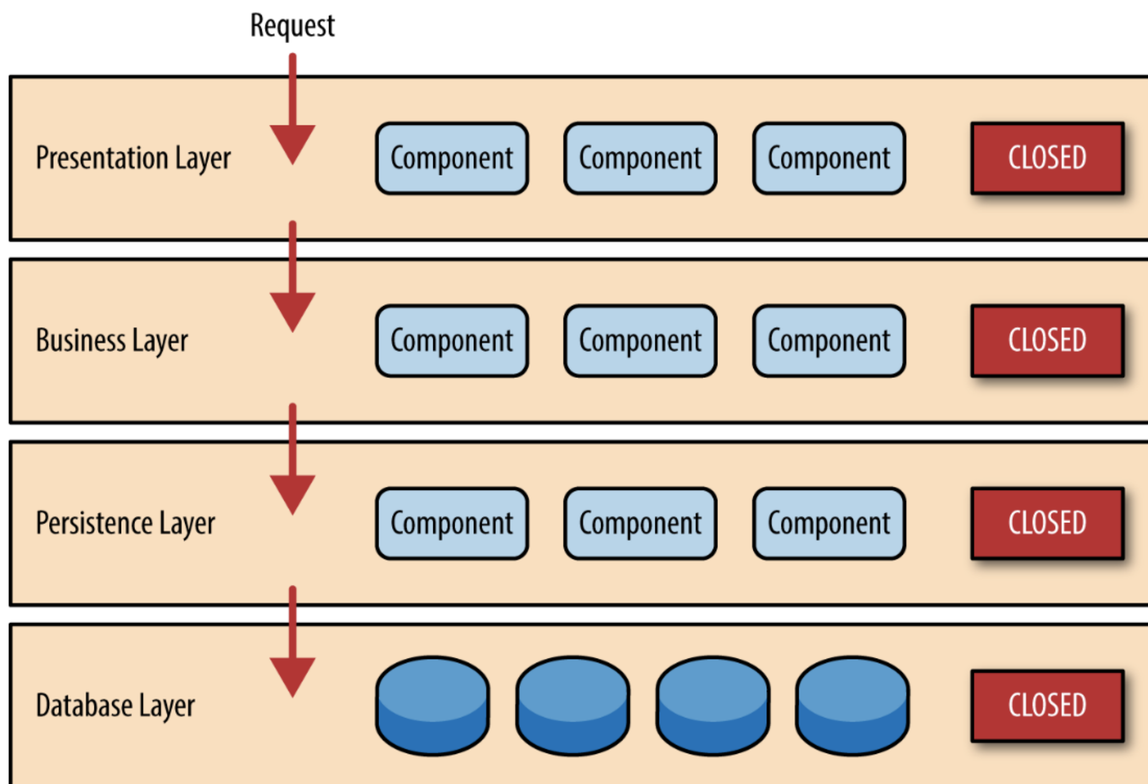
Jako build tool je využit **Webpack**, který nejen značně zjednodušuje a urychluje vývoj, ale také zkvalitňuje složení výsledného kódu. Například řeší separaci modulů a zapouzdření vnitřního stavu, který se dříve musel v javascriptu řeši skrze IIFE (Immediately Invoked Function Expression), čímž nedochází ke kolizím názvů a únikům dat. Dále obsahuje minifikaci a obfuskaci, kdy se zmenšuje velikost a znepřehledňuje kód, aby se ztížilo vykrádání kódu ležícího v prohlížečích uživatelů, tedy všem na očích. Také obsahuje nástroj **Babel**, který zajišťuje transpilaci javascriptových rozšíření do ES standardu. Nebo také dokáže transpilovat novější verzi ES do starší, což se mnohdy hodí kvůli nedostatečné podpoře prohlížečů. Při překládání do starší verze se také využívá tzv. polyfill, který doplňuje funkcionalitu rozšíření novějších verzí jazyka. Kombinace transpilace a polyfillu zajišťuje, že lze aplikaci spustit na více prohlížečích a různých verzích. Webpack také obsahuje jednoduchý vývojářský webový server v technologii NodeJS (javascript pro serverové využití) a frameworku express. Díky tomuto webovému serveru lze využít technologie HMR (Hot Module Replacement), kdy se změna v kódu jednotlivých modulů okamžitě zkompiluje a nahraje do prohlížeče. Také lze tento webový server využít jako proxy, čímž se dají přesměrovávat dotazy na reálný server s webovou službou.

3.3 Serverová část

Služba je realizována jako stateless, tedy bez udržovaného stavu systému jako sezení či obrázky. Pro autentifikaci se využívá JWT (Json Web Token), který se posílá v HTTP hlavičce Authorization typu Bearer. JWT [20] je standard definující kompaktní a soběstačný (self-contained) způsob bezpečného přenosu informací mezi dvěma stranami ve formátu JSON objektu. Kompaktností a soběstačností se myslí to, že poslaná data obsahují veškeré nutné informace. Při využití JWT k autentifikaci tak nemusí být potřeba dodatečného dotazování do databáze, čímž se může zrychlit obsluha a snížit zátěž na databázi. Samotné informaci v tokenu lze věřit, neboť je digitálně podepsaná. K tomu lze využít buď nějakého hesla skrze HMAC algoritmus, nebo veřejného a soukromého klíče skrze RSA či ECDSA algoritmy. Návrh této webové služby předpokládá nesoběstačný JWT token, využívá se dodatečného dotazování do databáze. Toto je mnohdy nutné kvůli zjednodušení synchronizace. Pokud je celý datový model obsažen v tokenech u několika klientů a jeden klient perzistentně změní stav datového modelu, desynchronizují se tak ostatní tokeny. Pro následnou synchronizaci tak lze využít třeba nějakou paměťovou databázi pro označení neaktuálních identifikátorů. Nebo lze v první řadě předejít samotné desynchronizaci tím, že token obsahuje pouze identifikátor, který se v čase nemění, a samotný datový model se vždy získává z databáze. Již při autentifikaci tak lze ověřit trvající existenci či aktivitu uživatele.

3.3.1 Architektura

Samotná webová služba je realizována skrze vrstvenou architekturu, která umožňuje jednoduchou vyměnitelnost jednotlivých vrstev. Vrstvená architektura [21], také známá jako n-tier architecture, je nejběžnější architekturou v monolitických systémech. V Java platformě je to de facto standard. Ve vrstvěné architektuře se jednotlivé komponenty organizují v horizontálních vrstvách, kde každá vrstva vykonává specifickou roli systému (prezentace, perzistence …). Pro účely této práce budou existovat dvě perzistentní vrstvy, které využívají společnou vrstvu s business logikou. Tato architektura má dva způsoby realizace: **striktní** a **nestriktní**. Striktní varianta neumožňuje přeskakovat vrstvy, což může znamenat zbytečné vytváření objektů a metod, které pouze přeposílají zprávy. Tento problém značně redukují mapovací knihovny, které se starají o přelévání dat ze struktur jedné vrstvy do struktur vrstvy druhé. Avšak stále existuje velký počet souborů podobného, ne-li stejného, pojmenování. To může být matoucí, zavádějící a náchylné na chyby. Nestriktní varianta umožňuje přeskakovat vrstvy na úkor izolovanosti a možnostem evoluce, protože dochází k většímu provázání vrstev.

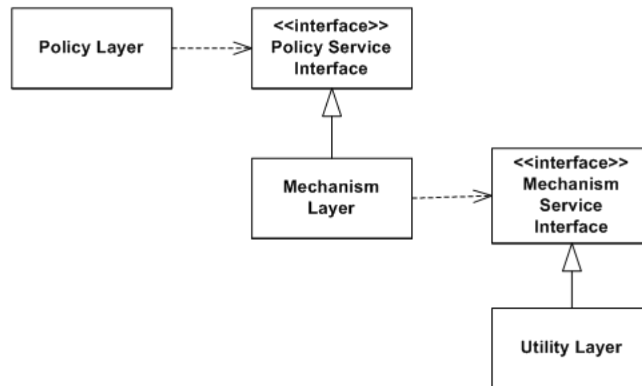


Obrázek 3.5 Striktní vrstvená architektura [21]

Jedná se o běžnou realizaci podnikových systémů skrze modulární monolit. Pro tuto práci je využita striktní varianta, aby se validace centralizovala do business vrstvy přes samotné získání dat.

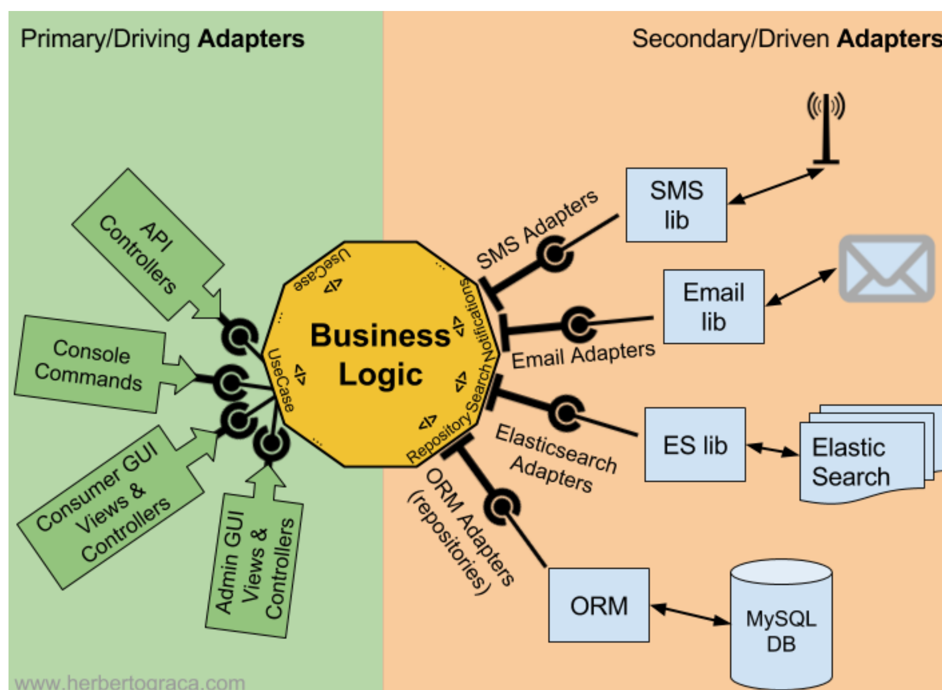
Typicky se tato architektura doplňuje Hexagonal architekturou pro abstrakci externích knihoven a předejití tzv. vendor lock-in situace. Vendor lock-in je problém, ve kterém se závislost na dodavateli stane tak kritickou, že nelze dodavatele vyměnit. Z pohledu kódu to nastává mnohdy pouze v případě přímého provázání kódu dodavatele s kódem vlastním. Hexagonal architektura spočívá v tom, že se vytvoří vlastní interní abstrakce, typicky rozhraní, pro funkcionalitu, kterou zajišťuje dodavatel. Jeho kód se pak integruje v podobě adaptéru (Adapter pattern), který realizuje interní rozhraní a využívá kód dodavatele. Této architektuře se také říká Ports and Adapters. V mnoha platformách již existují různé abstraktní balíčky, které mají za cíl právě oddělit konzumenta od dodavatele skrze abstraktní vrstvu. V PHP platformě existuje PSR (PHP Standards Recomendations), v Java platformě existuje JEE (Java Enterprise Edition).

V rámci OOD (Object Oriented Design) se jedná o DIP (Dependency Inversion Principle) [22], která říká, že vysokoúrovňové moduly by neměly záviset na nízkoúrovňových, ale oba by měly záviset na abstrakci. Abstrakce by tedy neměla záviset na detailech, ale detail by měl záviset na abstrakci. To lze hezky vidět na obrázku 3.6. Vysokoúrovňové je myšleno ve smyslu realizované logiky, kde vyšší úrovní modulů rozumíme ty, které zprostředkovávají funkcionalitu, k jejíž provedení využívají další moduly. Jedná se tedy o pohled na vazbu a závislost.



Obrázek 3.6 DIP skrze UML Class Diagram [23]

Pro zjednodušení však realizace této práce tento problém neřeší do detailu. Vrstvená architektura přináší dostatečné rozdělení zodpovědností a závislostí. Pro účely této práce tedy není potřeba vytvářet další úroveň abstrakce pro externí knihovny skrze Hexagonal architekturu či obecně DIP, jelikož se nepředpokládá evoluce a rozšiřování.



Obrázek 3.7 Hexagonal architektura [24]

V perzistentní vrstvě je použito ORM (Object Relational Mapping), technologie, která zajišťuje transformaci dat z fyzického uložiště do doménových modelů. Jelikož se fyzická reprezentace relační databáze neslučuje s hierarchickou strukturu reprezentace objektové, je potřeba vytvořit jistou logiku pro převádění těchto dvou forem. ORM tak vytváří jistou úroveň abstrakce od perzistentního uložení a reprezentace dat. ORM často na perzistentní vrstvě doprovází návrhové vzory Unit Of Work a Repository / DAO (Data Access Object).

Návrhový vzor Unit Of Work [25] zajišťuje, že se veškeré změny doménového modelu promítnou v perzistentním uložení pouze jednou. Když se data stáhnou z databáze, či jiného perzistentního uložiště, do paměti systému v podobě objektových modelů, je důležité sledovat změny, které se v modelech provedou. Jinak by se změny nepropsaly zpět do databáze. Stejně tak přidání nových objektů, či úplné smazání existujících. Samozřejmě můžeme při každé změně modelu aktualizovat uložiště, to ale může vést k mnoha velice malých volání, která kvůli blokujícímu přístupu zpomalí samotnou business operaci. Také se mohou zpomalit ostatní paralelní operace kvůli transakčnímu zpracování a zamykání uložiště v případě ACID (Atomicity, Consistency, Isolation, Durability) modelu, který je u relačních databází nutný. Unit of Work sleduje provedené změny modelu během business transakce, a při dokončení business operace vyřeší, jaké změny je potřeba provést v perzistentním uložišti.

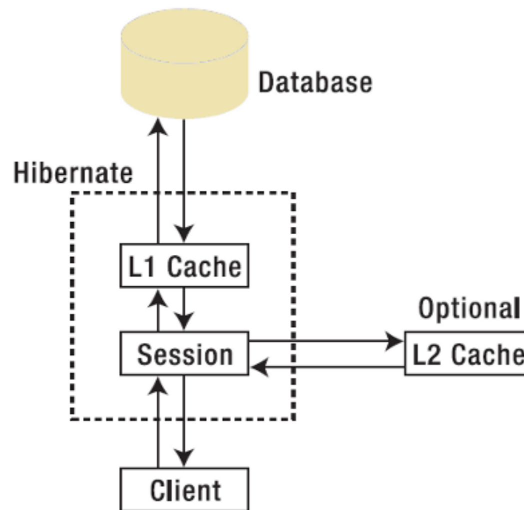
Návrhový vzor Repository [26] přidává ještě další úroveň abstrakce, konkrétně od typu uložiště. Jestliže ORM odděluje způsob fyzické reprezentace a zajišťuje transformaci do doménových modelů, pak Repository odděluje získávání těchto objektů od typu perzistentního uložiště. Díky tomu můžeme k objektům přistupovat způsobem podobným kolekcím, čímž se zjednodušuje vyměnitelnost použitého perzistentního uložiště.

Jelikož je často potřeba využít nějakého dynamického skládání dotazů a nestačí jen zapouzdřit logiku za název funkce, je potřeba parametrizovat logiku, využívá se typicky ještě Query Object Pattern [27]. Ten odstiňuje psaní SQL dotazů skrze objektovou kompozici kritérií, které se ve výsledku překládají do SQL dotazu pro konkrétního databázového dodavatele, případně do daného jazyka konkrétní technologie využité k perzistenci dat. Přestože je totiž SQL standardizovaný jazyk, jednotliví dodavatelé databázových technologií mírně upravují a rozšiřují SQL specifikaci. Jelikož odstiňujeme systém od konkrétní technologie použité pro perzistentní ukládání, potřebujeme odlišit i samotné skládání kritérií pro danou technologii. V Java platformě je toto realizováno skrze Criteria API, které je součástí JPA (Java Persistence API) specifikace.

V této práci je využito jak návrhového vzoru ORM, tak návrhového vzoru Repository. Unit of Work je otázkou konkrétní technologie využité k perzistentnímu ukládání, není pro potřeby této práce výlučně potřebný. Návrhový vzor Query Object Patter je otázkou potřeby parametrizovného dotazování, takže jeho využití záleží na realizaci systému.

V případě použití návrhového vzoru ORM je často možné načítat reference líně (lazy fetching), čímž může docházet k N+1 problému [28]. Pří líném načítání se využívá Proxy pattern, který při prvním přístupu k referenci danou referenci načte a vytvoří objekt. Tím se šetří paměť, neboť k přístupu k referenci nemusí v dané operaci dojít. Naopak ale kvůli tomu může dojít ke snížení výkonnosti, protože se nevyužívá optimalizovaných databázových dotazů se spojením tabulek. Když se načte seznam objektů, který se proiteruje s přístupem na referenci, celkově vznikne 1 (původní) + N (počet referencí) databázových dotazů. Samozřejmě nemusí dojít k volání databáze pro všechny reference, neboť ORM nástroje mívají interní cache. Například Hibernate má L1 pro cachování sezení a sdílenou L2 cache.

Zdali se využije cachování na úrovni samotného ORM nástroje, databáze, či nějakého replikování a balancování dotazů nad samotnou databází je irelevantní, neboť ve finále stejně dochází k N+1 volání a vyhodnocování dotazů na potenciálně několika úrovních.



Obrázek 3.8 Hibernate cache

Při návrhu datové vrstvy je tak vždy dilema. Buď se budou entity načítat rychle, ale brát více paměti. Nebo se budou načítat pomaleji, ale budou brát tolik paměti, kolik potřebují. Bohužel na toto neexistuje univerzální odpověď a je potřeba zvážit více aspektů. Běžnou praxí je zvolit první variantu a dále řešit v případě performance problémů, které stejně ze začátku většinou nebývají na datové úrovni, ale špatné práci s objekty v business logice.

N+1 problém je větší problém u GraphQL než u REST přístupu, neboť se u GraphQL předpokládá hierarchická struktura. U REST přístupu se často používá právě plochá a normalizovaná struktura, kde se posílají pouze identifikátory referencí a případně odkazy při využití HATEOS. Díky tomu lze u REST přístupu lépe odhadnout práci s objekty a správně nastavit způsob načítání. U GraphQL je potřeba monitorovat reálné klienty a vědět, jakým způsobem se dané objekty načítají.

3.3.1.1 CQRS

Jedním ze způsobů redukce N+1 problému je využití architektury CQRS (Command Query Responsibility Segregation) [29]. Jedná se o architektonický styl, který odlišuje modely pro modifikaci (Commands) a modely pro dotazování (Queries).

Obrázek 3.9 CQRS architektura

Lze tak vytvořit specializované end-pointy využívající rozdílné modely. Například jeden model s plochou strukturu a druhý model se zanořenou strukturou. První model by načítal reference buď lazy způsobem, případně by obsahoval pouze samotný referenční klíč. Druhý model by se však načítal okamžitě skrze optimalizovaný dotaz se spojením tabulek.

Díky tomuto konceptu lze také třeba optimalizovat načítání dat z databáze skrze replikaci typu Master-Slave a rozdělení uložišť podle typu modelů. Modely pro dotazování tak směrovat na instance typu Slave, typicky několik, a modely pro modifikace směrovat na instanci typu Master, která je vždy jedna.

3.3.2 Technologie

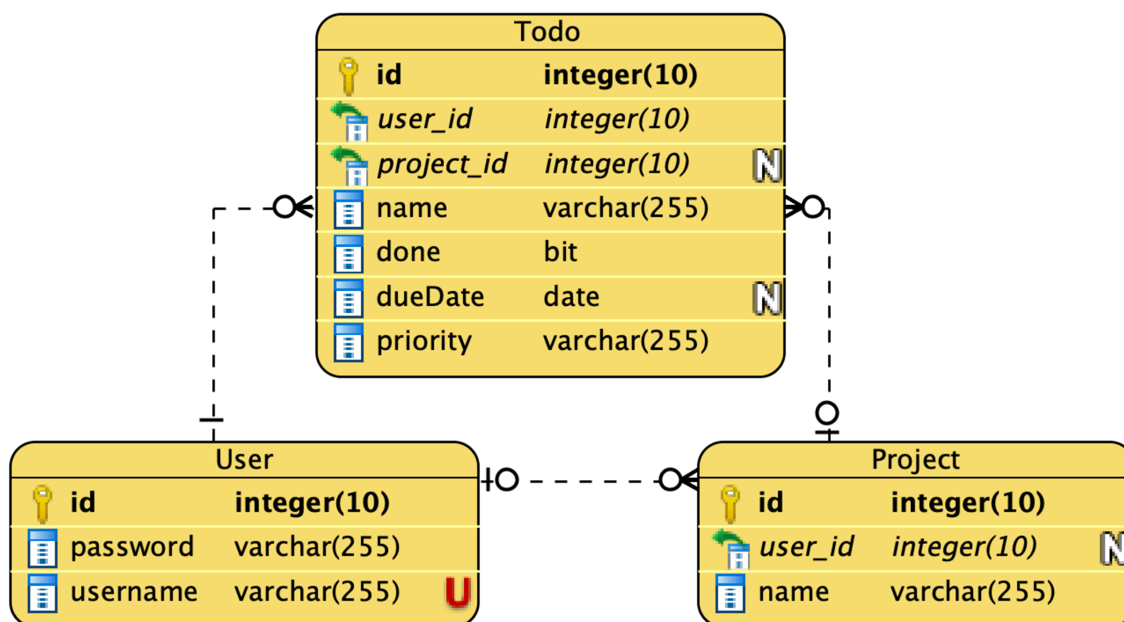
Serverová část je realizována v Java platformě, konkrétně JDK verzi 1.8. Dále je využit **Spring boot** framework, který poskytuje prostředí pro rapidní vývoj. Jeho cílem je zrychlit a zjednodušit vývoj aplikací s využitím modularizovaných komponent samotného frameworku Spring. Nabízí například vestavěný webový server, díky čemuž není potřeba rozcházet a spouštět samostatný proces s aplikačním či servlet kontejnerem. Nabízí také starter balíčky, které seskupují nutné závislosti pro nějakou konkrétní funkcionalitu a zjednodušují tak správu závislostí.

Pro ORM (Object Relational Mapping) je využita knihovna **Hibernate**, která realizuje JEE JPA (Java Persistence API) specifikaci. ORM je doplněna návrhovým vzorem Repository skrze Spring modul **spring-data-jpa** obsahující **Spring Data Repositories**. Tím je zajištěn přístup k doménovým objektům jako ke kolekci, čímž se zvyšuje abstrakce databázové vrstvy a zjednodušuje vyměnitelnost. Spring data repositories dokonce redukuje množství kódu nutného pro implementaci přístupu k datům pro různé dodavatele. Repozitáře se definují pouze jako rozhraní, ke kterým Spring generuje Proxy objekty. Samotný přístup k datům pro jednotlivé metody rozhraní je realizován buď konvenčně podle signatur (návratová hodnota, název a parametry), případně podle specifické anotace definující explicitně dotaz.

Jako datové uložiště je využita **H2** databáze ve vestavěném módu. Tím tedy dochází ke ztrátě dat při vypnutí systému, avšak pro účely práce není potřeba zvyšovat komplexnost infrastruktury a technologického stacku. Perzistentní vrstva odděluje způsob získání dat od jeho fyzického uložiště, čímž umožňuje jednoduchou vyměnitelnost. Ve vestavěném módu je přístup k databázi pouze z virtuální mašiny, která jej spustila. Nelze tedy využít nějakého externího klienta, ale je potřeba využít rozhraní poskytnuté přímo ve webové službě, které je dostupné pouze pro vývojářské účely.

Jako build tool je využit **Gradle**, který zajišťuje správu závislostí a sestavení aplikace. Ten oproti konkurenčním nástrojů jako Maven či starší Ant využívá DSL (Domain Specific Language) využívající syntaxy programovacích jazyků Groovy a Kotlin, čímž zjednodušuje konfiguraci a rozšiřuje možnosti automatizace. Závislosti jsou stahovány z veřejného repozitáře Maven Central, o který se stará Sonatype.

3.3.3 Databázový model



Obrázek 3.10 ERD s Crow's foot notací

Databázové entity nejsou nikterak složité. Nutno znovu podotknout, že cílem práce je analyzovat a porovnat technologie pro komunikaci, k čemuž není potřeba navrhnout zbytečně komplexní systém. V systému se vyskytuje entita uživatele, která obsahuje pouze uživatelské jméno a heslo. Délka jména a hesla nehraje roli, proto je zvolen maximální rozsah pro textový řetězec s proměnlivou délko, jméno ale musí být unikátní. Jako primární klíč je zvolen umělý číselný identifikátor, který poskytuje rychlejší číselné porovnání a má menší paměťový otisk. Entita projektu obsahuje název a referenci uživatele v podobě cizího klíče. Entita úkolu reference na uživatele a projekt, přičemž nemusí spadat do žádného projektu. Úkol nespadající do žádného projektu tak spadá do implicitního projektu Inbox, který nelze upravovat. Dále obsahuje název, příznak, zdali je úkol hotový, volitelné datum dokončení a prioritu.

Priorita je přitom reprezentována jako textový řetězec, aby se nemusel modelovat číselník, jelikož není potřeba lokalizace.

4 Implementace

V této kapitole je popsána implementace navrženého systému z předešlé kapitoly. Kapitola se zaměřuje primárně na vývojové aspekty jako rozdělení kódu, dodatečné knihovny nad rámec návrhu, či samotná realizace některé logiky. Popis je rozdělený na klientskou a serverovou část.

4.1 Klientská část

Klientská část využívá grafické rozhraní z předmětu ITU (Tvorba uživatelských rozhraní). To umožnilo soustředit se více na jádro práce, tedy samotnou komunikaci mezi klientem a webovou službou, místo zpracování grafického uživatelského rozhraní. Původní implementace využívala lokální uložiště k perzistenci stavu, což je pro účely této práce změněno na integraci webové služba skrze REST a GraphQL. Původní realizace nebyla bezchybná, což jsem se pokusil v rámci integraci s webovou službou doladit. Opravena byla například responzivita pro menší rozlišení a reakce na některé chybové situace.

Klientská část je, jak se již psalo v kapitole s návrhem systému, realizována pomocí technologie React. Jedná se tedy o SPA (Single Page Application), která se sama stará o překreslování aplikace, a z webové služby získává pouze data. Pro rozchození a vývoj byla využita knihovna **create-react-app**, která zajišťuje prvotní inicializaci projektu. Jedná se o knihovnu udržovanou vývojáři Facebooku a komunitou. Knihovna také obsahuje globální konfiguraci nástroje webpack, čímž se značně zjednodušuje prvotní spuštění a údržba, dokud není potřeba nějaké vlastní komplexnější konfigurace. Adresářová struktura je rozdělena podle typu.

Ve složce *src/models* leží modely, které se v aplikací využívají. Modely jsou reprezentovány třídou Record z immutable-js knihovny, která zajišťuje neměnnost struktury objektů. Modely slouží primárně k deserializaci dat obdržených ze služby. Explicitním vyjádřením a pojmenováním struktury modelu místo dynamického prototypování (duck-typing) se zpřehledňuje kód a redukují defekty kvůli špatné práci s objekty a neaktualizované typové kontrole v komponentách vyjádřené tvarem.

Ve složce *src/ducks* leží redux moduly podle konceptu redux ducks. Každý modul tak obsahuje actions, reducer a action creators. Jelikož je pro asynchronní zpracování využita knihovna redux-thunk a Promise API, leží logika tohoto zpracování přímo v modulech, konkrétně v action creators.

Ve složce *src/pages* leží samotné grafické komponenty dále členěné do podsložek podle stránek. Každá stránka má vždy Container soubor zastřešující celou stránku a složku *components*, kde leží dílčí komponenty stránky.

Ve složce *src/assets* leží SASS soubory organizované konceptem SMACSS (Scalable and Modular Architecture for CSS), který zjednodušuje údržbu a rozšiřování. Ten rozděluje styly do 5

skupin: base, layout, module, state a theme. Base slouží ke stylování základních HTML elementů, neměl by obsahovat žádné třídy či identifikátory. Layout slouží k rozdělení stránky jako celku do jednotlivých sekcí. Zastřešuje tak moduly do větších celků jako třeba hlavička, patička, sidebar, a další. Module slouží ke stylování znovuvyužitelných, modulárních částí designu. Jedná se o dílčí komponenty stránky jako formuláře, tlačítka, tabulky, a další. State slouží k popisu stavu komponent a theme pak případnému grafickému stylování v případě, že je potřeba podporovat více vzhledů.

Ve složce *src/layout* leží dekompozice layoutu a routování. Pro tuto aplikaci existují pouze dvě stránky, a to stránka pro přihlašování a stránka pro samotnou práci s úkoly.

Ve složce *src/core* leží interní knihovny zastřešující nějakou obecnou funkcionalitu jako HTTP dotazování či perzistenci autentifikačního tokenu.

4.2 Serverová část

Klientská část je, jak se již psalo v kapitole s návrhem systému, realizována pomocí frameworku Spring boot. Adresářová struktura je stejně jako v klientské části rozdělena podle typu. V balíčku *cz.sapiente.bachelor_thesis.dto* leží třídy sloužící k definici struktury vstupních a výstupních dat. DTO (Data Transfer Object) je koncept modelování objektů sloužících k serializaci a deserializaci při síťovém přenosu dat. Jedná se o anemické modely. To je takový model, které neobsahují žádnou logiku, a pouze zapouzdřují data. Jelikož Java neobsahuje konstrukt pro definici takových struktur a z hlediska zapouzdření není vhodné definovat atributy jako *public*, je využita knihovna **Lombok**. Ta využívá annotation preprocessing a dogenerovává některé instrukce při kompilaci. Jednodušeji se tak zapisují například field accessors (getters & setters).

V balíčku *cz.sapiente.bachelor_thesis.entity* leží modely reprezentující databázové entity. Modely jsou označeny JPA anotacemi, které zpracovává Hibernate. Ten zajišťuje vytvoření databázového schématu v nastavené databázi při spuštění služby.

V balíčku *cz.sapiente.bachelor_thesis.repository* leží repozitáře, definující metody pro přístup k databázi. Repozitáře se definují jako rozhraní, pro které Spring vygeneruje dynamické proxy objekty. Samotné metody mají jistou konvenci pojmenování, podle které Spring vytváří databázový dotaz.

Ve balíčku *cz.sapiente.bachelor_thesis.service* leží třídy, které zprostředkovávají business logiku.

Jak bylo popsáno výše, tak databáze je ve vestavěném módu a přístup je umožněn pouze skrze stejnou virtuální mašinu, nelze využít externího klienta. Pro vývojářské účely je k dispozici jednoduché rozhraní pod URL adresou **/h2-console** s uživatelským jménem **sa** bez hesla. Pro vývojářské účely také systém obsahuje předpřipravená data pod uživatelem **ibp** s heslem **ibp**, která se nahrají při startu služby.

K realizaci REST end-pointů je využita **spring-web** knihovna. Ačkoli JEE nabízí abstrakci **jax-rs**, nebylo pro tuto práci potřeba řešit vyměnitelnost technologií na prezentační vrstvě. Spring navíc nabízí jednodušší konfiguraci pomocí anotací, díky čemuž se mírně zmenšuje a zpřehledňuje kód. Autentifikace je za pomoci **spring-security** a **spring-security-config** skrze vlastní servlet filtr, který

kontroluje přítomnost autorizační HTTP hlavičky. Pokud je hlavička přítomna, tak filtr ověří podpis, expiraci a v případě úspěchu nastaví bezpečnostní kontext na autentifikovaného uživatele. Konfigurace REST prezentační vrstvy není nikterak složitá, neboť se využívá Java anotací. Logika prezentační vrstvy je také velice prostá, zprostředkovává pouze transformaci vstupů na volání služby a následnou transformaci výstupu služby. Dodatečně zde může být ještě validace a logování. Validace je zde přenesena do business vrstvy, aby se nemusela duplikovat v následné GraphQL realizaci, a logování je zde vynecháno, jelikož se nejedná o produkční verzi.

K realizaci GraphQL end-pointu jsou využity knihovny **graphql-java-boot-starter** a **graphql-java-tools**, které poskytují jak jednoduchou integraci do Spring boot frameworku, tak konfiguraci GraphQL end-pointů. Jedná se o schema-first přístup, kde není potřeba manuálně konfigurovat schéma skrze objekty. End-point je dostupný pod URL adresou **/graphql** metodou POST. End-pointy jsou realizovány v balíčku *cz.sapiente.bachelor_thesis.graphql*. Jedná se o třídy realizující rozhraní *com.coxautodev.graphql.tools.GraphQLQueryResolver*
a *com.coxautodev.graphql.tools.GraphQLMutationResolver*.

5 Zhodnocení

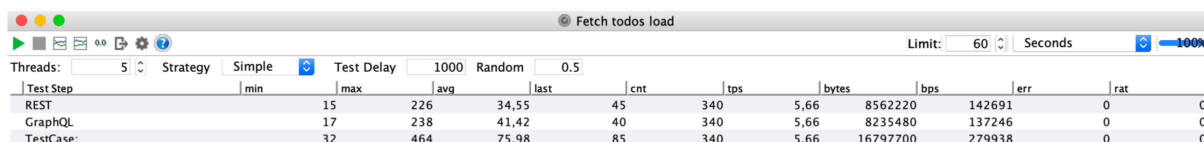
Tato kapitola se věnuje zhodnocení technologií REST a GraphQL s ohledem na realizovaný systém a případné další aspekty, které byly z realizace pro zachování jednoduchosti vynechány. Cílem této kapitoly je naznačit rozdíly a případné výhody a nevýhody, který z rozdílů plynou.

5.1 Výkonnost

Výkonnost aplikace je jedním z měřitelných ukazatelů kvality. Pokud je interakce s uživatelským rozhraním pomalá a uživatel po každé akci vidí animaci s načítáním, bude jej užívání aplikace frustrovat. Například průměrný zákazník nakupující online očekává, že se stránka načte do 2 vteřin, a po 3 vteřinách až 40 % uživatelů stránku opouští [30].

Při měření výkonnosti se využívají výkonnostní testy, kam řadíme: Load testing, Stress testing, Spike testing a další. Cílem výkonnostního testování je odhalit výkonnostní problémy jako dlouhá načítací doba, špatná doba odezvy, špatná škálovatelnost, úzká hrdla a další [31]. V případě aplikací využívajících webové služby můžeme měřit end-to-end, kde se simuluje pohyb uživatele v uživatelském rozhraní, nebo samostatné služby poskytující data.

Pro účely této práce bylo zvoleno testování webových služeb s cílem porovnat načítací dobu. K testování byl použit nástroj SoapUI a Load testy. Testovaly se dva scénáře. Prvním bylo načítání úkolů, kde se načítaly i identifikátory projektů, kde se může projevit tzv. N+1 problém (Obrázek 5.1). Druhým bylo načítání projektů s plochou strukturou (Obrázek 5.2).

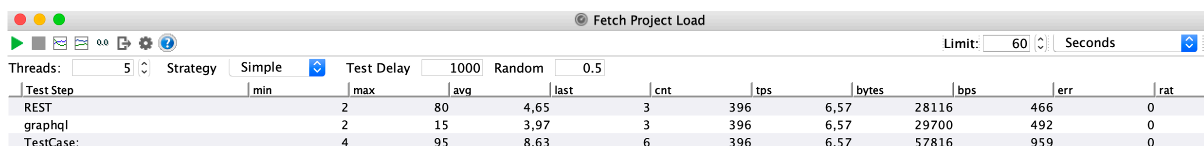

Fetch todos load

Limit: 60 Seconds 100%

Threads: 5 Strategy Simple Test Delay 1000 Random 0.5

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
REST	15	226	34,55	45	340	5,66	8562220	142691	0	0
GraphQL	17	238	41,42	40	340	5,66	8235480	137246	0	0
TestCase:	32	464	75,98	85	340	5,66	16797700	279938	0	0

Obrázek 5.1 načítání úkolů


Fetch Project Load

Limit: 60 Seconds

Threads: 5 Strategy Simple Test Delay 1000 Random 0.5

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
REST	2	80	4,65	3	396	6,57	28116	466	0	
graphql	2	15	3,97	3	396	6,57	29700	492	0	
TestCase:	4	95	8,63	6	396	6,57	57816	959	0	

Obrázek 5.2 načítání projektů

Podle naměřených výsledků nelze přímo určit, která technologie je z pohledu výkonnosti lepší. V prvním případě je rychlejší REST o ~7 ms, ale v druhém případě je rychlejší GraphQL o ~0,6 ms. To je bohužel zapříčiněno nedostatečnou hloubkou hierarchie dat, který vychází z referenční aplikace a neposkytuje správné podmínky.

5.2 Propustnost

Jedním z negativních aspektů webových služeb je plýtvání síťového přenosu kvůli over-fetching a under-fetching problémům. Over-fetching znamená načítání velkého množství dat, která klient ani nepotřebuje. Under-fetching znamená načítání nedostatečných dat, což nutí klienta vytvořit a zaslat nový dotaz pro data chybějící. Primární motivací GraphQL je odstranění těchto dvou problémů a tím snížení plýtvání síťového přenosu. Platební modely cloudových služeb často uvažují nejen nároky na provoz služby, ale také velikost přenesených dat v rámci infrastruktury. Pro správný návrh systému tak nestačí jen řešit rychlost a zátěž služby, ale také způsob přenosu a velikost přenášených dat.

Tento aspekt se také odráží ve webových službách placených podle počtu volání, jako je OneDrive, Google Drive a mnoho dalších. Pokud nám platební model integrované služby omezuje množství volání, a struktura odpovědí nás nutí volat službu vícekrát pro realizace jedné operace, pak dochází ke značnému plýtvání dostupných volání a zvýšení investice do integrace. Z pohledu služby se přitom nemusí jednat o černou praktiku pro zvýšení výnosů. Může se jednat o pouhý aspekt udržitelnosti. Pokud se do REST end-pointů začnou přidávat nové zdroje jen kvůli optimalizacím, dříve či později začne být struktura služby nepřehledná a náročná k údržbě, čímž se omezí rozšiřitelnost a vitalita produktu na kompetitivním trhu.

Zde právě přichází k záchraně GraphQL, dotazovací jazyk s hierarchickou strukturou, díky které jsme schopni omezit množství volání webové služby, a snížit množství přenášených dat.

5.3 Flexibilita

REST přístup postrádá jistou formu flexibility na nestandardizované aspekty. Vezměme například filtrování nad zdrojem uživatelů. Pro filtrování uživatelů podle jména můžeme jednoduše navrhnout end-point jako *GET /users/?name=Pavel*. Pokud bychom chtěli filtrovat podle dvou jmen, museli bychom zavést nějaký separátor. Výsledný end-point by mohl vypadat takto: *GET /users?name=Pavel|Petr*. Zde již začíná být situace více komplikovaná, neboť musíme zajistit syntaktickou správnost a samotnou analýzu. Co prakticky vytváříme, je vlastní dotazovací jazyk, který REST přístup sám o sobě nezavádí. Nemluvě o tom, že *GET* metoda má v prohlížečích omezenou délku. Pokud bychom chtěli zavést strukturovanější dotazování, museli bychom využít metodu *POST*, která není ovšem podle specifikace idempotentní, a odporuje záměru operace. GraphQL tento problém řeší již v samotném návrhu, neboť se jedná o hierarchicky strukturované a typované zprávy oddělené podle záměru na dotazy a mutace.

Dalším důkazem jisté neflexibility je pojmenování. Například vyhledávání pro konkrétní produkty v konkrétních obchodech. Jedná se o zdroj vyhledávání, zdroj obchodů, nebo zdroj produktů? Nemluvě o přihlašování uživatelů, o jaký zdroj se jedná? Typickými přístupy jsou POST /*login* a *POST /auth*, které sice svým jménem vystihují záměr, ale nezapadají do samotného konceptu zdrojů.

5.4 Testovatelnost

Testování softwaru je nedílnou součástí procesu vývoje z pohledu zaručení kvality. V části zhodnocení výkonnosti byly již popsány metody pro testování výkonnosti. Z hlediska testovatelnosti se dá však testovat i správnost systému z pohledu sémantiky a struktury. V případě testování služby jako black box, tedy testování bez znalosti vnitřní realizace, takové testování nazýváme end-point testing.

Testování webových služeb můžeme provádět s různým záměrem, který určuje míru složitosti testů. Můžeme například testovat samotnou strukturu odpovědí, čímž společně s automatizací získáme možnost podchytit změnu struktury včas. Nebo můžeme testovat samotnou sémantiku dat skrze testovací scénáře, kde sekvencí kroků ověřujeme správnost a korektnost chování služby.

Pro testování lze využít známé nástroje jako **Katalon** či **Postman**, které se zaměřují na obecné testování služeb dostupných přes HTTP, neomezují tedy ve formátu dat. Tyto nástroje pouze usnadňují konfiguraci HTTP dotazů a organizací testovacích scénářů.

Pro REST přístup existuje dokonce nástroj **Dredd**, který umožňuje automatizované testování podle dokumentace. Skrze hooky se dají doplnit některé typy kontrol, avšak samotné schéma odpovědí se dá testovat OOTB (Out Of The Box).

GraphQL lze také testovat skrze nástroj Dredd, ale dokumentace schématu by se musela přepsat do nějakého Dreddem podporovaného jazyka jako OpenAPI či API Blueprint. Pro Graphql existuje docker image **symm/graphql-contract-test**, který testuje samotné schéma vůči webové službě. Jinak kromě těchto dvou nástrojů existuje spousta knihoven, které ulehčují psaní testů pro komunikaci. Jen se již musí psát v konkrétním programovacím jazyce.

Závěr

Z vytyčených cílů práce se podařilo jednotlivé technologie popsat a vytvořit referenční aplikaci a webovou službu. Posledním cílem bylo snížit entropii a zjednodušit rozhodování ohledně zavedení GraphQL. Z praktických zkušeností i průzkumu vyplývá, že jedním z klíčových aspektů úspěšného zavádění nových technologií je přítomnost doménové expertízy. REST ani GraphQL nejsou výjimkou, proto je samotná otázka preference spíše otázkou kompetence a zkušenosti, kterou je potřeba získat a dále kultivovat. Co se týká přechodu z REST do GraphQL u produkčně běžících služeb, tak je proces jednodušší, pokud je systém realizován modulárně, ideálně skrze vrstvenou či eventovou architekturu. Lze tak postupně a inkrementálně přidávat nové a předělávat staré end-pointy, neboť lze mít obě verze paralelně vedle sebe. Samotná otázka přechodu pak závisí na rozpočtu a míře motivace, neboť GraphQL přináší konkrétní výhody a zlepšení.

Z pohledu zvýšení síťové propustnosti a lepších podmínek zpoplatněných služeb má GraphQL velký potenciál. Cenou za větší flexibilitu a propustnost je menší čitelnost z pohledu přenosu, ohledně které se snažil REST přístup poučit od RPC, a ke které se GraphQL opět vrací. Čitelnost je však také otázkou doménové znalosti. Pokud se vývojový tým naučí správné praktiky ladění s danou technologií, redukuje se případný negativní vliv na efektivitu.

Referenční aplikace bohužel neposkytuje takové podmínky, které by umožnily využít potenciál technologie GraphQL a prokázat svou přednost v přenosu dat. Porovnání a zhodnocení bylo kvůli tomu omezeno bez možnosti přímého doložení tvrzení. To poskytuje příležitost pro navázání a rozšíření porovnávaných kritérií a jejich zhodnocení. Jako další krok bych tak navrhoval vytvořit větší a hlubší strukturu dat, na které by se daly uvedené aspekty lépe testovat a prokázat.

Literatura

[1] W. contributors, *Hypertext Transfer Protocol.* [Online]. [cit. 2019-04-01]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=896381679.

[2] W. contributors, *Text-based protocol*. [Online]. [cit. 2019-04-01]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Text-based_protocol&oldid=863471352.

[3] T. Berners-Lee, R. Fielding a L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax.* [Online]. 2005 [cit. 2019-04-01]. Dostupné z: https://tools.ietf.org/html/rfc3986.

[4] W. contributors, *XML-RPC*. [Online]. [cit. 2019-04-02]. Dostupné z: https://en.wikipedia.org/w/index.php?title=XML-RPC&oldid=885216579.

[5] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. [Online]. 2000 [cit. 2019-04-02]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[6] M. Shaw a G. David, *Software architecture: perspectives on an emerging discipline.* Pearson, 1996, ISBN 978-0131829572.

[7] D. Wessels, Web Caching: Reducing Network Traffic. O'Reilly Media, 2001, ISBN 978-1565925366.

[8] M. Fowler, *Richardson Maturity Model*. [Online]. 2010 [cit. 2019-01-02]. Dostupné z: https://martinfowler.com/articles/richardsonMaturityModel.html.

[9] Mozilla, *Cross-Origin Resource Sharing (CORS)*. [Online]. [cit. 2019-04-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[10] *HATEOS Driven REST APIs*. [Online]. [cit. 2019-01-02]. Dostupné z: https://restfulapi.net/hateoas/.

[11] M. Abid, *Four REST API Versioning Strategies*. [Online]. [cit. 2019-05-10]. Dostupné z: https://www.xmatters.com/integrations-blog/blog-four-rest-api-versioning-strategies/.

[12] P. Saint-Andre, D. Crocker a M. Nottingham, *Deprecating the "X-" Prefix and Similar Constructs in Application Protocol*. [Online]. 2012 [cit. 2019-04-02]. Dostupné z: https://tools.ietf.org/html/rfc6648.

[13] I. Facebook, *GraphQL*. [Online]. 2018 [cit. 2019-04-10]. Dostupné z: https://graphql.github.io/graphql-spec/June2018/.

[14] Facebook, *Schemas and Types*. [Online]. [cit. 2019-04-10]. Dostupné z: https://graphql.org/learn/schema/.

[15] N. Burk, *The Problems of "Schema-First" GraphQL Server Development*. [Online]. 2019 [cit. 2019-04-10]. Dostupné z: https://www.prisma.io/blog/the-problems-of-schema-first-graphql-development-x1mn4cb0tyl3.

[16] W. Ezell, *What is a Single Page Application? (And Should You Use One?)*. [Online]. 2018 [cit. 2019-04-12]. Dostupné z: https://dotcms.com/blog/post/what-is-a-single-page-application-and-should-you-use-one-.

[17] M. Fowler, *GUI Architectures*. [Online]. [cit. 2019-04-12]. Dostupné z: https://martinfowler.com/eaaDev/uiArchs.html.

[18] SourceMaking, *Observer Design Pattern*. [Online]. [cit. 2019-04-12]. Dostupné z: https://sourcemaking.com/design_patterns/observer.

[19] M. Fowler, *Event Sourcing*. [Online]. [cit. 2019-04-12]. Dostupné z: https://www.martinfowler.com/eaaDev/EventSourcing.html.

[20] M. Jones, J. Bradley a N. Sakimura, *JSON Web Token (JWT)*. [Online]. 2015 [cit. 2019-04-12]. Dostupné z: https://tools.ietf.org/html/rfc7519.

[21] M. Richards, *Software Architecture Patterns.* O'Reilly Media, Inc., 2015, ISBN 978-1491971437.

[22] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices.* Pearson, 2002, ISBN 978-0135974445.

[23] W. contributors, *Dependency inversion principle*. [Online]. [cit. 2019-05-01]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Dependency_inversion_principle&oldid=895302070.

[24] H. Graca, *Ports & Adapters Architecture*. [Online]. [cit. 2019-04-13]. Dostupné z:: https://herbertograca.com/2017/09/14/ports-adapters-architecture/.

[25] M. Fowler, *Unit of Work.* [Online]. [cit. 2019-04-13]. Dostupné z: https://martinfowler.com/eaaCatalog/unitOfWork.html.

[26] M. Fowler, *Repository*. [Online]. [cit. 2019-04-13]. Dostupné z: https://martinfowler.com/eaaCatalog/repository.html.

[27] M. Fowler, *Query Object*. [Online]. [cit. 2019-04-13]. Dostupné z: https://martinfowler.com/eaaCatalog/queryObject.html.

[28] M. Ajith, *The (Silver) Bullet for the N+1 Problem*. [Online]. [cit. 2019-04-13]. Dostupné z: https://www.sitepoint.com/silver-bullet-n1-problem/.

[29] M. Fowler, *CQRS*. [Online]. [cit. 2019-04-13]. Dostupné z: https://martinfowler.com/bliki/CQRS.html.

[30] L. C. Hogan, *Performance is User Experience*. [Online]. [cit. 2019-04-13]. Dostupné z: http://designingforperformance.com/performance-is-ux/.

[31] Guru99, *Performance Testing Tutorial: What is, Types, Metrics & Example*. [Online]. [cit. 2019-04-13]. Dostupné z: https://www.guru99.com/performance-testing.html.

Přílohy

Příloha A

Obsah CD

- *src/* – Zdrojové soubory
    - *api/* – Webová služba
    - *client-rest/* – Klientská aplikace využívající REST technologii
    - *client-graphql/* – Klientská aplikace využívající GraphQL technologii
- *docs/* – Text práce

Příloha B

Návod na spuštění

Serverová část

Pro spuštění i sestevaní je potřeba mít nainstalovanou správnou verzi JDK. Testované jsou Oracle JDK 1.8 a Amazon Corretto 11. Níže zmíněné příkazy předpokládají, že aktuální pracovním adresářem je složka *src/api* z odevzdaného obsahu.

Pro spuštění i sestavení lze využít docker prostředí příkazem *docker run --rm -p 8080:8080 -v $(pwd):/app -w /app amazoncorretto:11 <command>*, kde *<command>* nahradíme celým příkazem pro spuštění nebo sestavení.

Pro spuštění spustíme příkaz *java -jar build/libs/bachelor-thesis-0.0.1-SNAPSHOT.jar*. Na standardním výstupu se objeví URL adresa, pod kterou je webová služba dostupná (typicky *localhost:8080*). Odevzdaný obsah obsahuje již sestavenou aplikaci, není ji tak třeba sestavovat.

Pro sestavení spustíme příkaz *./gradlew -g gradle-cache build*, čímž se aplikace sestaví do složky *build/libs* a lze ji spustit podle odstavce výše.

Klientská část

Pro spuštění i sestavení je potřeba mít nainstalovanou správnou verzi Node.js a npm. Testovaná je Node.js verze 8.14 a npm verze 6.4.1. Níže zmíněné příkazy předpokládají, že aktuální pracovním adresářem je složka *src/client-rest* či *src/client-graphql* z odevzdaného obsahu. Klientskou část lze buď spustit s vývojářským serverem, nebo sestavit do statických souborů, které je následně potřeba buď umístit na spuštěný webový server, nebo otevřít z prohlížeče. Důležitá je správná konfigurace proměnných prostředí v souborech *.env.local* a *.env.production.local*. Tyto soubory obsahují klíče

- REACT_APP_API_URL: nastavení url adresy pro webový server (například *htttp://localhost:8080*)
- PUBLIC_URL: nastavení cesty k systému, jedná se o URL prefix, který se užívá při generování odkazů ke zdrojům
- REACT_APP_BASE_PATH: nastavení bázové části cesty k systému, slouží jako prefix kvůli správné funkčnosti routování v React technologii

Pro spuštění i sestavení lze využít docker prostředí příkazem *docker run --rm -p 3000:3000 -v $(pwd):/app -w /app node:8.14-jessie <command>*, kde *<command>* nahradíme celým příkazem pro spuštění nebo sestavení.

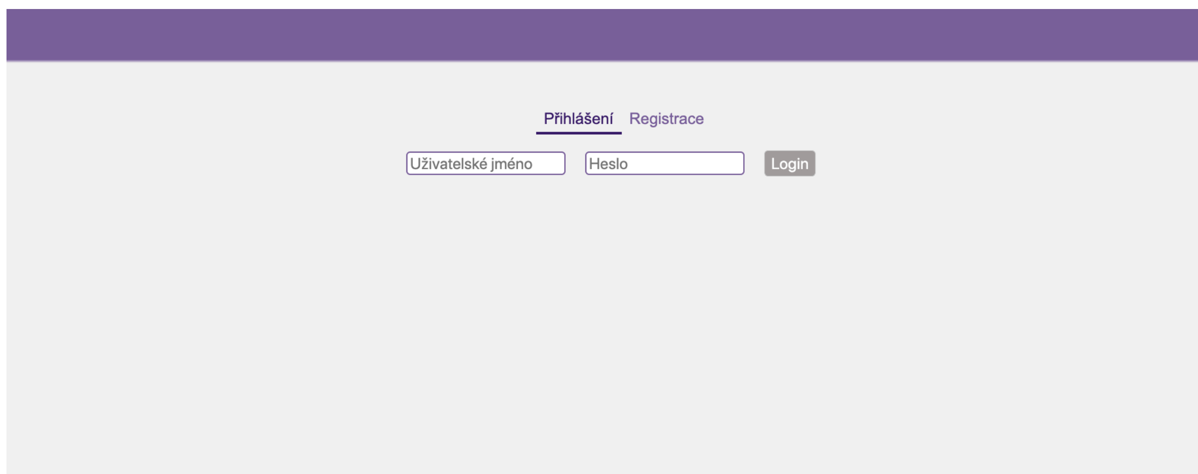
Pro spuštění vývojářského serveru spustíme příkaz *npm ci && npm run start*, který na standardním výstupu vypíše dostupnou URL adresu a port (typicky *localhost:3000*).

Pro sestavení statických souborů spustíme příkaz *npm ci && npm run build*, který do složky *build* vygeneruje statické soubory, které lze následně zpřístupnit skrze webový server či zobrazit přes prohlížeč. Je zde potřeba dát pozor na konfiguraci proměnných prostředí.

Příloha C

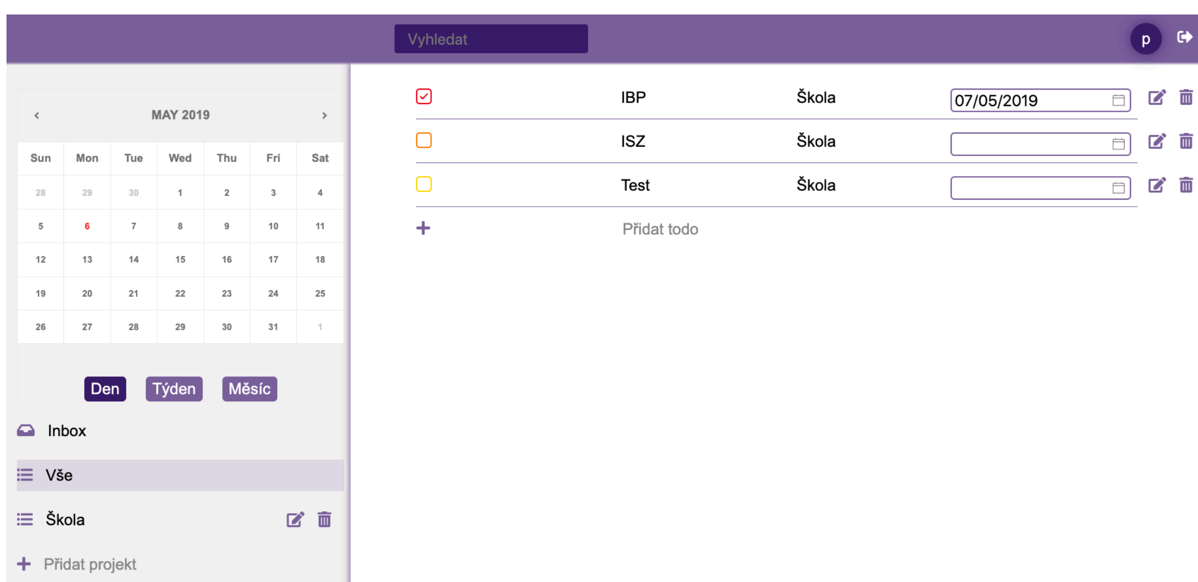
Uživatelský manuál

Při načtení výchozí stránky se zobrazí stránka pro přihlášení. Tlačítko *Login* se zpřístupní po vyplnění vstupních polí. Kromě přihlášení již registrovaného uživatele lze také registrovat uživatele nového bez jakýchkoli omezení. Stačí kliknout na horní záložku *Registrace*, vyplnit a odeslat údaje. Následně se lze přihlásit na záložce *Přihlášení* pod nově vytvořeným uživatelem.



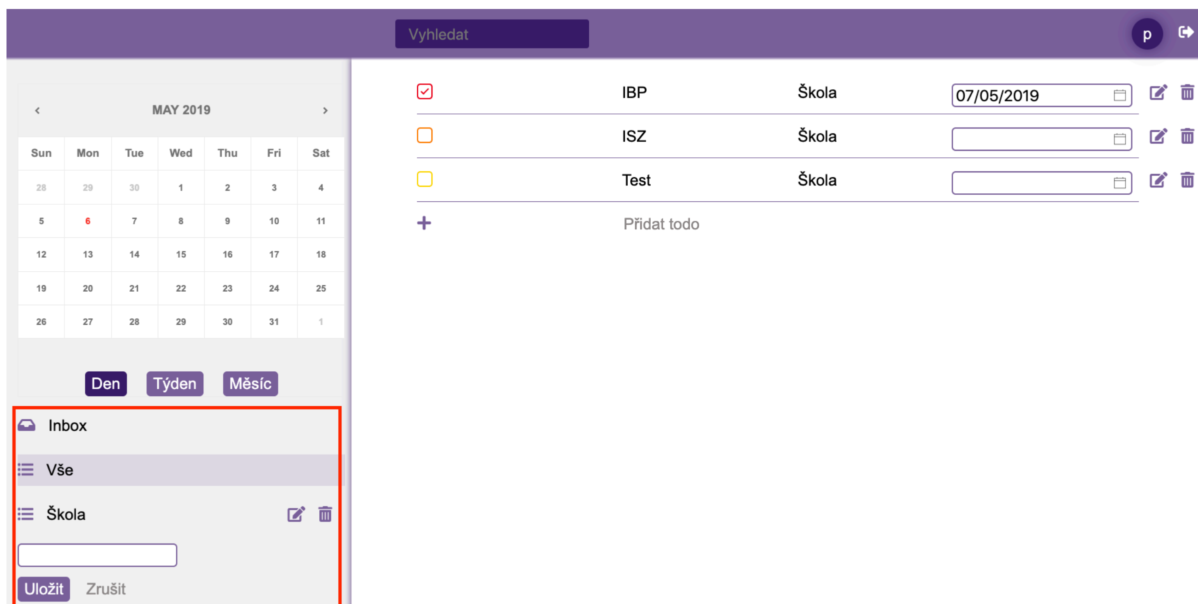
Obrázek C.1 Stránka s přihlášením

Po přihlášení se zobrazí stránka, na které lze vidět projekty na levé straně a úkoly na straně pravé. V menu leží vstupní pole pro vyhledávání a tlačítko pro odhlášení. Nad projekty se také nachází kalendář, který slouží k filtrování úkolů podle zadaného intervalu. Interval lze zadat na konkrétní den, týden, nebo měsíc.



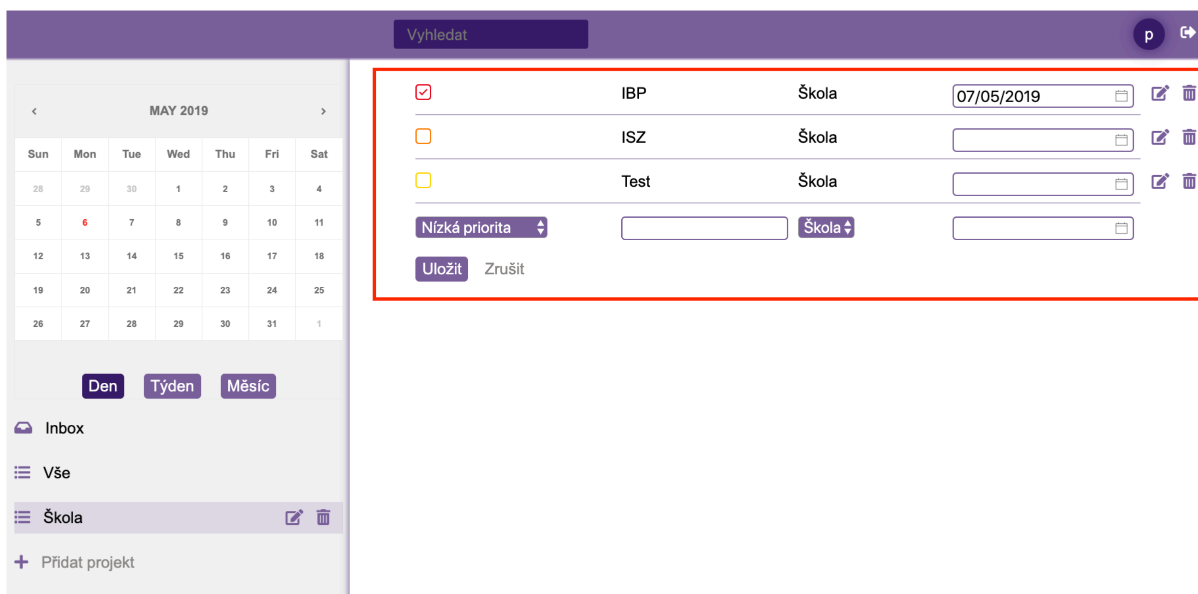
Obrázek C.2 Stránka s úkoly a projekty

Na levé straně s projekty lze kliknout na tlačítko *Přidat projekt*, čímž se nám zobrazí vstupní pole pro název. Stejným způsobem lze i upravit existující projekt, stačí najet myší na název a kliknout na editační ikonu. Při kliknutí na projekt dochází k filtrování úkolů podle projektu. V seznamu projektu tak leží neměnitelné položky *Vše* a *Inbox*. Vše slouží k zobrazení úkolů nezávisle na projektu, *Inbox* pak k zobrazení úkolů bez přiřazeného projektu.



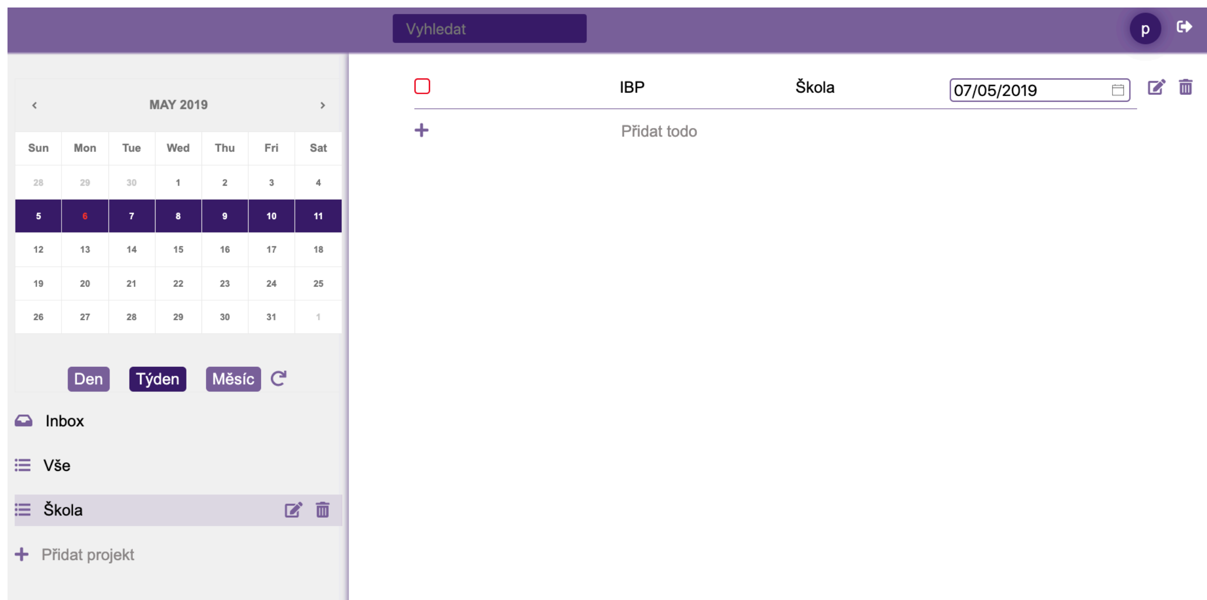
Obrázek C.3 Stránka při přidávání projektu

Na pravé straně s úkoly lze klinout na tlačítko *Přidat todo*, čímž se nám zobrazí vstupní pole pro přidání. Stejná pole se zobrazí i při editaci, kterou umožníme kliknutím na editační tlačítko na řádku úkolu. Úkol lze také odškrtnout jako splněný/nesplněný kliknutím na čtvereček ležící na levé straně. V případě splněného úkolu se ve čtverečku objeví fajfka. Barva tohoto čtverečku se určuje podle priority úkolu. Také lze kliknout na ikonu kalendáře, čímž lze změnit datum splnění bez nutnosti editace celého úkolu. To slouží k zjednodušení a urychlení manipulace.



Obrázek C.4 Stránka při přidávání úkolu

Při kliknutí na kalendář se nám zvýrazní vybraný interval, přičemž červená barva čísla značí aktuální den. Vedle výběru typu intervalu leží ikona pro smazání výběru, čímž se zruší časový filtr.



Obrázek C.5 Stránka při výběru intervalu v kalendáři