Univerzita Pardubice
Fakulta elektrotechniky a informatiky

REST API framework

Dominik Fűrst

Bakalářská práce
2023

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Dominik Fürst**
Osobní číslo:	**I19082**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**REST API framework**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

V úvodní části bakalářské práce bude proveden úvod do problematiky REST API. Podrobně bude rozebrána architektura REST rozhraní, včetně zhodnocení přínosů a omezení užití tohoto rozhraní a uvedení případných alternativ. Dále bude provedena rešerše současného stavu implementací REST rozhraní ve vybraném programovacím/skriptovacím jazyce a bude provedena kritická komparace těchto implementací.
V aplikační části bakalářské práce bude vytvořena knihovna pro vývojáře webových aplikací pro tvorbu REST rozhraní. Knihovna bude vytvořena tak, aby naplňovala účel jednoduchosti a rychlosti tvorby REST rozhraní v procesu tvorby softwarových aplikací. Užití vytvořené knihovny bude demonstrováno a ověřeno na případové studii. V závěru práce budou zhodnoceny konkrétní přínosy a omezení užití takto vytvořené knihovny.

Rozsah pracovní zprávy: **min. 30 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

JIN Brenda, SAHNI Saurabh a Amir SHEVAT.Designing Web APIs: Building APIs That Developers Love, O'Reilly, 2018, ISBN 978-1492026921

MASSE Mark. REST API Design Rulebook, O'Reilly, 2011, ISBN 978-1449310509

Lubbers, Albers, Salim: Pro HTML 5 Programming: Powerful APIs for Richer Internet Application Development, Apress, 2010, ISBN 978-4302-2790-8

SUBRAMANIAN Harihara a Pethuru RAJ.Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs, Packt Publishing, 2019, ISBN 978-1788992664

Vedoucí bakalářské práce: **Ing. Lukáš Čegan, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **17. prosince 2021**
Termín odevzdání bakalářské práce: **13. května 2022**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2022

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 15. 8. 2022

Dominik Fűrst

PODĚKOVÁNÍ

Rád bych touto formou poděkoval vedoucímu bakalářské práce panu Ing. Lukáši Čeganovi, Ph.D. za celkový dohled, odborné rady a poskytnuté konzultace, které mi v rámci svého volného času ochotně věnoval. Dále bych chtěl poděkovat své rodině za podporu a možnost studovat Univerzitu Pardubice.

ANOTACE

Bakalářská práce se zaměřuje na návrh a vývoj knihovny pro programátory webových aplikací využívajících rozhraní REST. Teoretická část se zabývá analýzou architektury REST a jejími dalšími alternativami. V praktické části je v jazyce PHP vytvořena abstraktní vrstva mezi konkrétním programem a samotným rozhraním REST. Knihovna umožňuje vývojářovi manipulovat s daty v databázi prostřednictvím operací CRUD. K tomuto účelu se využívají HTTP metody GET (pro čtení), POST (pro zápis), PUT (pro úpravu) a DELETE (pro smazání). Programátor může využívat funkce knihovny prostřednictvím konfiguračního souboru, kde pro danou cestu webového serveru uvede jednotlivé metody spolu s potřebnými informacemi pro jeho již konkrétní implementaci. Na základě těchto informacích dojde k provedení jedné z uvedených operací. Aplikace poskytuje prostředky pro práci s databází MySQL, rovněž umožňuje odeslání HTTP požadavku na libovolný cílový bod v síti internet. Výsledky těchto dotazů jsou uživateli zobrazeny společně se stavovým kódem požadavku a zprávou o provedení, které si autor může dále zpracovat dle požadavků vlastní aplikace.

KLÍČOVÁ SLOVA

REST API, architektura rozhraní, webová knihovna, abstraktní vrstva, manipulace s daty, aplikační rozhraní, HTTP požadavky, CRUD operace, vývoj webových aplikací, back-end vývoj

TITLE

REST API framework

ANNOTATION

The bachelor's thesis focuses on the design and development of a library for programmers of web applications using the REST interface. The theoretical part deals with the analysis of the REST architecture and its other alternatives. In the practical part, an abstract layer is created in the PHP language between a specific program and the REST interface itself. The library allows the developer to manipulate data in the database through CRUD operations. For this purpose, the HTTP methods GET (for reading), POST (for writing), PUT (for modification) and DELETE (for deletion) are used. The programmer can use the functions of the library through a configuration file, where he lists individual methods for a given path of the web server together with the necessary information for its already specific implementation. Based on this information, one of the listed operations will be performed. The application provides resources for working with the MySQL database, it also allows sending an HTTP request to any

destination point on the Internet. The results of these queries are displayed to the user together with the status code of the request and the execution report, which the author can further process according to the requirements of his own application.

KEYWORDS

REST API, interface architecture, web library, abstract layer, data manipulation, application interface, HTTP requests, CRUD operations, web application development, back-end development

OBSAH

Seznam obrázků **10**
Seznam tabulek **11**
Seznam příkladů **12**
Seznam ukázek kódů **13**
Seznam zkratek **14**
Úvod **15**
1 REST **16**
1.1 Historie vzniku REST 16
1.2 Architektura REST rozhraní 17
1.3 Zhodnocení přínosů 21
1.4 Nevýhody architektury REST 22
1.5 Porovnání alternativ 23
1.5.1 SOAP 23
1.5.2 GraphQL 23
1.5.3 Falcor 24
1.5.4 gRPC 24
2 Souhrn implementací v programovacím jazyce PHP **26**
2.1 Laravel 26
2.2 Lumen 28
2.3 Guzzle 28
2.4 Epiphany 30
3 Návrh a realizace REST API frameworku **32**
3.1 Popis frameworku 32
3.2 Požadavky na framework 33
3.2.1 Funkční požadavky 33
3.2.2 Nefunkční požadavky 34
3.3 Aplikační logika 34
3.3.1 Konfigurační soubory 34
3.3.2 Soubor index 45
3.3.3 Třída Application 45
3.3.4 Třída Request 46
3.3.5 Třída Response 47
3.3.6 Třída Router 48
3.3.7 Třída Route 51
3.3.8 Třída RestController 51
3.3.9 Správa požadavků 52
3.3.10 Databáze 56
3.3.11 Parsování konfiguračního souboru 56
3.3.12 Autorizace 57
4 Aplikace REST frameworku při tvorbě webové aplikace **59**
4.1 Popis projektu 59

4.2 Struktura projektu 59
4.3 Instalace a implementace REST API framework 60
4.4 Instalace a tvorba projektu ve frameworku React 62
4.5 Výsledný produkt 64
Závěr 68
Použitá literatura 69
Přílohy 72

SEZNAM OBRÁZKŮ

Obrázek 1: Příklad požadavku s využitím SOAP API [2] 16
Obrázek 2 - Schéma komunikace rozhraní REST 21
Obrázek 3: Architektura gRPC [17] 25
Obrázek 4: Struktura požadavku s metodou POST 47
Obrázek 5: Struktura odpovědi při práci s databází 48
Obrázek 6: Struktura autorizačních tabulek 58
Obrázek 7: Struktura databáze pro příklad využití 60
Obrázek 8: Přihlašovací formulář 64
Obrázek 9: Seznam studentů z pohledu učitele 65
Obrázek 10: Přidávání známek studentovi 66
Obrázek 11: Zobrazení známek studenta 66
Obrázek 12: Seznam studentů z pohledu administrátora 67
Obrázek 13: Úprava údajů o studentovi 67

SEZNAM TABULEK

Tabulka 1: Příklad struktury zdrojů pro databázi studentů.....18
Tabulka 2: Příklad metod a operací nad databází studentů 19
Tabulka 3: Komponenty nastavení databáze 36
Tabulka 4: Komponenty nastavení HTTP 42
Tabulka 5: Seznam cest v rámci příkladu využití.....61

SEZNAM PŘÍKLADŮ

Příklad 1: Předání identifikátoru studenta v cestě 20
Příklad 2: Předání rozmezí vrácených výsledků v dotazu 20
Příklad 3: Příkaz pro vytvoření Laravel aplikace [23]....26
Příklad 4: Spuštění vývojového serveru Laravel [23] 26
Příklad 5: Nastavení databáze ve frameworku Laravel [23]....26
Příklad 6: Instalace frameworku Guzzle [26] 29
Příklad 7: Nastavení databáze....35
Příklad 8: Ukázky cest 35
Příklad 9: URL adresa s parametrem....35
Příklad 10: Nastavení REST metod....35
Příklad 11: Nastavení výběru sloupců a spojení tabulek pro databázi 37
Příklad 12: Typy parametrů v URL adrese....38
Příklad 13: Nastavení specifických parametrů databáze 38
Příklad 14: Kompletní nastavení cesty s využitím databáze a parametrem "group by" 39
Příklad 15: Možnosti cest s volitelným parametrem 39
Příklad 16: Kompletní nastavení cesty s využitím databáze a obecným parametrem 40
Příklad 17: Adresa s parametrem v cestě....40
Příklad 18: Odpověď z databáze....40
Příklad 19: Nastavení metody POST pro databázi 41
Příklad 20: Ukázka nastavení HTTP pro převod měn 43
Příklad 21: Odpověď cílového serveru pro převod měn....44
Příklad 22: Nastavení veškerých REST metod při práci s HTTP typem....45
Příklad 23: Příkaz pro stažení frameworku jako závislosti....60
Příklad 24: Nastavení podpory pro systém cURL 60
Příklad 25: Příkaz pro spuštění API serveru....61
Příklad 26: Nastavení výběru známek studenta pro příklad využití 62
Příklad 27: Příkaz pro vytvoření React aplikace 63
Příklad 28: Příkaz pro spuštění React aplikace....63
Příklad 29: Ukázka struktury komponent souboru App.js....63
Příklad 30: Funkce saveMark() pro vrácení seznamu studentů....64

SEZNAM UKÁZEK KÓDŮ

Ukázka kódu 1: Směrování s Regulárním Výrazem v Laravelu [24] 27
Ukázka kódu 2: Ukázka iterace v jazyce Blade [25] 27
Ukázka kódu 3: Použití nástroje Inertia ve frameworku Laravel [25] 28
Ukázka kódu 4: Příklad použití Guzzle [27] 29
Ukázka kódu 5: Nastavení hlaviček ve frameworku Guzzle [28] 29
Ukázka kódu 6: Nastavení cest ve frameworku Epiphany [29] 30
Ukázka kódu 7: Použití tříd a Regulárních výrazů ve frameworku Epiphany [29] 31
Ukázka kódu 8: Obsah souboru index.php 45
Ukázka kódu 9: Konstruktor třídy Application 46
Ukázka kódu 10: Funkce run() třídy Application 46
Ukázka kódu 11: Funkce send() třídy Request 48
Ukázka kódu 12: Funkce resolve() třídy Router 49
Ukázka kódu 13: Funkce resolvePattern() třídy Router 50
Ukázka kódu 14: Funkce GET() třídy RestController 52
Ukázka kódu 15: Funkce buildGetQuery() třídy SQLBuilder 53
Ukázka kódu 16: Funkce addLimitClause() třídy SQLBuilder 54
Ukázka kódu 17: Funkce send() třídy SQLManager 54
Ukázka kódu 18: Funkce buildQueryString() třídy HttpBuilder 55
Ukázka kódu 19: Funkce send() třídy HttpManager 55
Ukázka kódu 20: Připojení k databázi 56
Ukázka kódu 21: Funkce readContent() třídy ConfigParser 57
Ukázka kódu 22: Atributy třídy DatabaseData 57

SEZNAM ZKRATEK

REST	Representational State Transfer
API	Application Interface
HTTP	Hypertext Transfer Protocol
SOAP	Simple Object Access Protocol
PHP	Hypertext Preprocessor
JSON	JavaScript Object Notation
XML	Extensible Markup Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
YAML	YAML Ain’t Markup Language
WSDL	Web Services Description Language
TCP	Transmission Control Protocol
IDE	Integrated Development Environment
RPC	Remote Procedure Call
MVC	Model-view-controller
PDO	PHP Data Objects

ÚVOD

S příchodem interaktivních webových stránek na začátku 21. století se začaly hledat způsoby, jak postupovat při výměně dat při spolupráci mezi klientem a serverem. Stále častěji se tak zmiňovaly pojmy jako API či webové služby. Společně s nimi se do popředí začal dostávat termín REST.

REST je architektonický styl, který prostřednictvím standartních HTTP metod umožňuje přistupovat k datům na určitých místech. Termín REST byl poprvé definován již v roce 2000 v disertační práci Roye Fildinga, který patří k jednomu z tvůrců protokolu HTTP. Rozhraní REST implementuje čtyři základní metody pro práci s daty, které se souhrnně označují jako CRUD operace. Jedná se o metody GET, POST, PUT a DELETE. Pomocí těchto metod lze definovat, jakým způsobem budou cílová data využívána. Důvodem autora ke zvolení tohoto tématu se stal fakt, že je tento styl vysoce využíván při tvorbě profesionálních webových aplikací.

Na úvod této práce je přiblížena architektura REST. Nejprve je popsána samotná architektura REST, poté jsou nastíněny výhody, ale i omezení tohoto rozhraní. V práci jsou následně představeny a zhodnoceny možné alternativy, které lze aplikovat místo rozhraní REST. Dále jsou čtenáři přiblíženy možné implementace REST API v programovacím jazyce PHP. Hlavním cílem práce je však návrh a realizace REST API framework, jehož účelem je poskytnout návrhářům webových aplikací rozhraní pro práci s technologií REST. Na závěr je představena implementace navrženého řešení v rámci jednoduché ukázky.

1 REST

Tato kapitola pojednává o architektonickém stylu „*Representational State Transfer*“, známém též pod zkratkou REST. Na úvod je popsána historie vzniku rozhraní REST, následně je představena jeho architektura, výhody a omezení, které budou porovnány s dalšími alternativami.

1.1 Historie vzniku REST

Integrace tvorby webových API před rokem 2000 obvykle vyžadovala používání protokolů, které byly složité vytváření, manipulaci a ladění. Příkladem je protokol SOAP. [1]

```
<?xml version="1.0"?>
<SOAP:Envelope
    xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema/instance"
    xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
  <SOAP:Body>
      <calculateArea>
        <origin>
           <x xsd:type="float">10</x>
           <y xsd:type="float">20</y>
        </origin>
        <corner>
           <x xsd:type="float">100</x>
           <y xsd:type="float">200</y>
        </corner>
       </calculateArea>
   </SOAP:Body>
</SOAP:Envelope>
```

Obrázek 1: Příklad požadavku s využitím SOAP API [2]

Termín REST poprvé definoval Roy Fielding se svým týmem v roce 2000 ve své disertační práci. Jejich cílem bylo vytvořit standard, který by umožňoval každému serveru možnost komunikovat s kterýmkoliv jiným serverem na světě. Vznikl tak model, který model, který popisoval základní sadu principů, vlastností a omezení, které se nyní nazývají REST. Ačkoliv jsou tyto pravidla striktní, jsou zároveň i univerzální. Díky tomu byl kladen důraz na to, aby API byly jednodušší, což zpomalilo křivku učení vývojářům, kteří se snaží daný software integrovat. [2]

První, kdo se o tento fenomén zajímal, byly technologické společnosti eBay a Amazon, které se zabývaly elektronickým obchodováním. Společnost eBay nabídla své REST API výběru partnerů, což ukázalo, jak lukrativní mohou být nově přístupná API. Vzhledem k tomu, že se trh již neomezoval pouze na návštěvníky jedné webové stránky, ale i na návštěvníky ostatních webových stránek s přístupem k jejich API, došlo tak k vyšší viditelnosti společnosti a tím pádem i k nárustu prodejních příležitostí. Jednoduchost kódu okamžitě svedla i ostatní online

platformy, které tak začaly přemýšlet nad hodnotou svého kódu, nikoliv pouze nad jejich spotřebitelskými produkty. [1]

Své vlastní REST API spustila v srpnu 2004 i stránka Flickr, která se rychle stala hlavní platformou pro obrázky, které mohli blogeři snadno vkládat na své webové stránky. Když později vznikla platforma Facebook a následně Twitter, uživatelé internetu nebyli spokojeni, že platformy nemají veřejně přístupná API. Obě stránky tak ustoupily a v roce 2006 vydaly oficiální verze svých API. [2]

1.2 Architektura REST rozhraní

Architektonický styl REST popisuje šest omezení:

- Jednotné rozhraní – Definuje rozhraní mezi spotřebitelem služby a poskytovatelem služby. [4] Prostředek v systému by měl mít pouze jeden logický URI, který by měl poskytovat způsob, jak lze získat související nebo další data. Reprezentace zdrojů by se rovněž měla řídit specifickými pokyny, jako jsou konvence pojmenování, formáty odkazů nebo formáty dat. Jakmile se vývojář seznámí s daným API, měl by být schopen použít podobný přístup i pro další API. [3]
- Bezstavovost – Zde se Roy Fielding inspiroval protokolem HTTP, odkud je toto pravidlo převzané. Veškeré interakce mezi klientem jsou bez stavů. Server neukládá o příchozím HTTP požadavku klienta nic. Každý požadavek je považován za nový. Není zde tedy žádná relace a ani historie. Pokud klientská aplikace musí být pro koncového uživatele stavová, kde se uživatel jednou přihlásí a následně provádí autorizované operace, pak by každý požadavek od klienta měl obsahovat veškeré potřebné informace, včetně autentizace a autorizace. [3]
- Uložitelnost do mezipaměti – Ukládání dat a odpovědí do mezipaměti je velice důležité pro celkovou použitelnost systému. Ukládání do mezipaměti přináší zlepšení výkonu na straně serveru a z důvodu snížení zátěže i lepší rozsah škálovatelnosti pro server. V architektuře REST se do mezipaměti ukládají zdroje, pokud je to možné. Samotné ukládání do mezipaměti lze implementovat na straně klienta i na straně serveru. Dobře spravované ukládání do mezipaměti by mělo částečně nebo úplně eliminovat některé interakce mezi klientem a serverem. [3]
- Klient-server – Klientské a serverové aplikace se musí být schopny vyvíjet samostatně, aniž by na sobě byly závislé. Klient by měl znát jen a pouze identifikátory

URI prostředků. Klienti a servery se mohou upravovat a vyvíjet samostatně, dokud je rozhraní mezi nimi nezměněno. [3]

- Vrstvený systém – REST umožňuje používat architekturu vrstveného systému, kde lze nasadit rozhraní API na server A, ukládat data na server B a ověřovat požadavky například na serveru C. Klient tedy obvykle nemůže říct, zda je připojen přímo ke koncovému serveru nebo pouze k prostředníkovi na cestě. [3]
- Kód na vyžádání – Zde se jedná pouze o volitelné omezení. Obvykle se skrze API odesílají pouze statická data, typicky ve formátech JSON nebo XML. V případě potřeby lze však vrátit spustitelný kód, který lze využít pro podporu části dané aplikace. [3]

REST API popisuje řadu prostředků a operací, které lze nad těmito prostředky vykonávat. Tyto operace lze vykonávat ze kteréhokoliv klienta HTTP, včetně kódu JavaScript běžícího na straně klienta. [4]

Všechny prostředky REST API jsou definované relativně k jeho základní cestě. Základní cesta může být použita i k zajištění izolace mezi různými rozhraními nebo různými verzemi jednoho REST API. Cesty ke zdroji mohou být hierarchické, přičemž dobře navržená struktura může pomoci uživateli porozumět zdrojům dostupných v rámci REST API. Níže uvedená tabulka ukazuje příklad využití struktury zdrojů pro přístup k databázi obsahující informace o studentech. [4]

Tabulka 1: Příklad struktury zdrojů pro databázi studentů

Zdroje	Popis
/students	Všichni studenti v databázi.
/students/12345	Konkrétní student s identifikátorem 12345.
/students/12345/orders	Všechny objednávky studenta s identifikátorem 12345.
/students/12345/orders/67890	Konkrétní objednávka s identifikátorem 67890 studenta s identifikátorem 12345.

Každý zdroj v REST API obsahuje sadu operací, kterou lze volat prostřednictvím klienta HTTP. Operace v REST API má název a metodu HTTP. Název operace musí být jedinečný v rámci celého rozhraní REST. Kombinace cesty a metody HTTP vytváří požadavek k identifikaci zdroje a provedení dané operace. [4]

REST API využívá následující metody HTTP:

- HTTP GET – Slouží k vrácení informací o zdroji. Protože metoda GET nemění stav zdroje, jedná se o bezpečnou metodu. V případě úspěchu vrátí status kód 200 (OK) a tělo obsahující data o zdroji, obvykle ve formátu JSON nebo XML. [5]
- HTTP POST – Využívá se k vytvoření nového zdroje do kolekce zdrojů. Metoda POST není bezpečná, zavolání dvou identických požadavků bude mít za následek vytvoření dvou zdrojů obsahujících identická data (s výjimkou unikátního identifikátoru). [5]
- HTTP PUT – Slouží k upravení již vytvořeného zdroje. V případě, že zdroj neexistuje, může se API rozhodnout, zdali ho vytvoří či nikoliv. [5]
- HTTP DELETE – Využívá se pro odstranění zdroje identifikovaného v URI požadavku. Pokud je zdroj nenalezen, je vrácen status kód 204 (žádný obsah). [5]
- HTTP PATCH – Slouží k částečnému upravení existujícího zdroje. Rozdíl s metodou PUT je takový, že PATCH by se měl využívat v případech, kdy je vyžadováno upravit pouze část zdroje, zatímco PUT se využívá k nahrazení konkrétního zdroje jiným. Nutno podotknout, že metoda PATCH není v určitých prohlížečích, serverech a webových aplikacích podporovaná. [5]

Tabulka 2: Příklad metod a operací nad databází studentů

HTTP Metoda	Název Operace	Popis
GET	getStudent	Vrátí z databáze informace o studentovi.
PUT	updateStudent	Upraví informace o studentovi v databázi.
DELETE	deleteStudent	Smaže studenta z databáze.

Každá operace v REST API může mít také sadu parametrů, které může klient HTTP použít k předání informací do daných operací. Každý parametr musí být přítomen v definicích REST API s jedinečným názvem a typem. [4]

Rozlišujeme následující typy parametrů:

- Parametry cesty – Lze je použít k identifikaci konkrétního zdroje. Hodnotu parametru klient předá jako proměnnou část adresy URL. Například identifikátor studenta lze předat jako parametr cesty s názvem „studentId“. [4]

/students/{studentId}

Příklad 1: Předání identifikátoru studenta v cestě

- Parametry dotazu – Hodnota parametru dotazu je klientem předána jako pár obsahující klíč a hodnotu na konci adresy URL. To lze například využít k předání minimálního a maximálního počtu vrácených výsledků konkrétní operace. [4]

/students?min=5&max=10

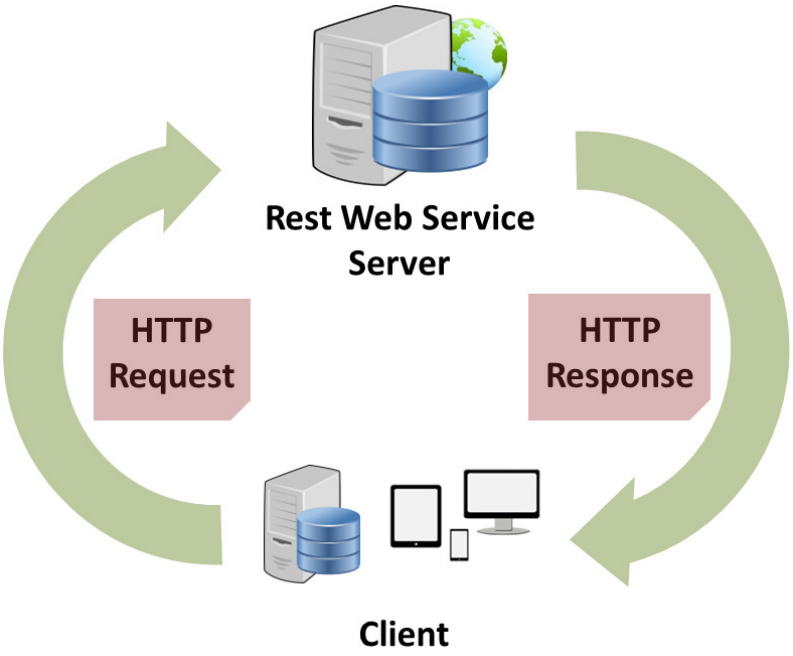
Příklad 2: Předání rozmezí vrácených výsledků v dotazu

- Parametry hlavičky – Klient může předat hodnoty parametrů ve formě HTTP hlavičky požadavku. Příkladem může být předání unikátního identifikátoru, který identifikuje klienta. [4]

Jedna operace může přijmout více parametrů všech tří typů. Dle metody HTTP pak operace může přijímat data od klienta v těle požadavku a odesílat data zpět v těle odpovědi. [4]

Webové aplikace využívající rozhraní REST používají ke sdělování stavů číselné stavové kódy HTTP. Tyto kódy se využívají k detekci chyb a usnadňování procesu monitorování API. Nejčastěji se můžeme setkat s následujícími kódy:

- Stavový kód 200 – Požadavek byl úspěšný.
- Stavový kód 401 – Požadavek nebyl dokončen z důvodu neplatného pověření.
- Stavový kód 404 – Požadovaný zdroj nebyl nalezen.
- Stavový kód 505 – Na serveru došlo k neopravitelné chybě aplikace. [12]



Obrázek 2 - Schéma komunikace rozhraní REST

1.3 Zhodnocení přínosů

REST je jedním z velice populárních architektur používaných při vytváření API. Je tedy zřejmé, že poskytují mnoho výhod. REST API jsou navrženy tak, aby využívaly stávající protokoly. Pro webová rozhraní se obvykle využívá protokol HTTP. Díky tomu vývojáři při implementaci REST API nejsou nuceni instalovat další software nebo knihovny. [6] Zároveň mohou tato rozhraní vytvářet v mnoha programovacích jazycích, jako jsou jazyky Python, Javascript, Node.js, Ruby, C# a mnoho dalších. [8]

Z hlediska vývoje je velikou výhodou, že architektura REST zcela odděluje uživatelské rozhraní od serveru a datového úložiště. Dochází tak ke zlepšování přenositelnosti rozhraní na jiné typy platforem, zvyšování škálovatelnosti a umožnění nezávislého vývoje různých částí aplikace. Každý vývojový tým tak může snadno škálovat produkt bez větších problémů. Mohou migrovat na jiné servery nebo provádět nejrůznější změny v databázi, pokud jsou data z každého požadavku správně odeslána. [13]

Jednou z klíčových výhod REST API je to, že poskytují velkou flexibilitu. Data nejsou nijak svázána se zdroji nebo s metodami, díky čemuž může REST zpracovávat rozličné typy volání a vracet různé formáty dat. Na rozdíl od svého předchůdce SOAP, REST není omezen pouze na formát XML, místo toho může vracet XML, JSON, YAML nebo jakýkoliv jiný formát dat v závislosti na požadavcích klienta. [6] Například formát JSON je kompaktnější a velikostně

menší než formát XML. Díky tomu může být zpracováván rychleji než formát XML. [7] Díky této flexibilitě mohou vývojáři API vyhovět potřebám různorodých zákazníků. [6]

Jak již bylo zmíněno v kapitole popisující architekturu REST, jedním z klíčových principů je bezstavovost. Tedy každý požadavek je zpracováván nezávisle na předchozích. Bezstavové servery udržují prostředky v paměti pouze při jejich zpracování a následně je uvolní. Pokud by se stavy pro jednotlivé přihlášené uživatele ukládaly na straně serveru, mohla by se výsledná data nafouknout a začít zabírat velké množství zdrojů na serveru. S tím souvisí i problém se škálovatelností. Pokud je například vícero serverů disponujících nástrojem pro vyrovnávání zatížení a server z důvodu zatížení přepojí klienta z jednoho serveru na druhý, nebude druhý server znát data své předchozí relace. To je samozřejmě možné vyřešit, ale dojde ke ztížení škálovatelnosti. [7]

Důležitým faktorem škálovatelnosti je možnost ukládání do mezipaměti. Dobře zavedený systém ukládání do mezipaměti může výrazně snížit průměrnou dobu odezvy serveru. REST si klade za cíl ukládání do mezipaměti usnadnit. Vzhledem k tomu, že je server bezstavový, měly by požadavky GET vracet stejnou odpověď bez ohledu na předchozí relaci. Díky tomu lze tyto požadavky snadno uložit do mezipaměti, se kterou mohou interagovat i prohlížeče. Rovněž můžeme prostřednictvím hlaviček Cache-Control a Expires ukládat do mezipaměti požadavky metody POST. [7]

Popularita REST je způsobena rozšířeným použitím v implementacích na straně klienta i serveru. Na straně serveru mohou vývojáři využívat frameworky založené na REST, jako jsou frameworky Restlet nebo Apache CXF. Na straně klienta mohou vývojáři využívat frameworky jako jQuery, Node.js, Angular, EmberJS a mnoho dalších, díky kterým mohou vyvolávat webové služby REST pomocí knihoven zabudovaných v jejich aplikačním rozhraní. [12]

1.4 Nevýhody architektury REST

V předchozí kapitole byla zmíněna bezstavovost jako výhoda ve smyslu škálovatelnosti a lepší správě při ukládání do mezipaměti. Bezstavovost je však možné označit i jako limitaci rozhraní. Většina webových aplikací vyžaduje stavové mechanismy. V případě správy e-shopu je nutné před každým nákupem znát počet položek v nákupním košíku. Veškeré stavy tak zůstávají ukládány na straně klienta, což činí klientskou aplikaci těžkou a obtížně udržovatelnou. [9]

Jednou z nevýhod může být absence jazyka WSDL. Tato zkratka označuje Web Service Description Language, což lze přeložit jako jazyk popisu webových služeb. Tento jazyk se

používá k popisu funkčnosti webové služby založené na SOAP. Soubory WSDL umožňují uživateli nastavovat:

- Zda se prvek nebo atribut může objevit vícekrát.
- Jestli je prvek či atribut povinný nebo volitelný.
- Konkrétní pořadí prvků, pokud je požadováno. [10]

Pomocí WSDL je možné klientovi sdělit, které všechny operace může webová služba provádět. Dokument tak bude obsahovat všechny informace, jako jsou datové typy používané ve zprávách, díky čemuž lze jednoduše provést validaci přijímaných informací. [11]

Vývojáři pracující s REST se často setkávají s omezeními vzhledem k jeho architektuře. Lze mezi ně zařadit multiplexing více požadavků skrze jediné připojení TCP, různé požadavky na zdroje pro každý zdrojový soubor, odesílání požadavků na server a dlouhé hlavičky požadavků HTTP, které způsobují zpoždění při načítání webových stránek. [12]

Vývojáři často nesouhlasí s definováním návrhů založených na architektuře REST. Architektonický styl REST postrádá jasnou referenční implementaci nebo pevně daný standart, pomocí kterého by bylo možné jednoznačně určit, zdali je konkrétní návrh skutečně možné označit jako REST. To vede k nejistotě, zda dané webové API odpovídá principům založených na REST. [12]

Služby REST často vracejí velké množství nepoužitelných dat v kombinaci s relevantními informacemi, což bývá výsledkem vícenásobných serverových dotazů. Díky této neefektivitě dochází k prodlužování doby, kterou klient potřebuje k vrácení požadovaných dat. [12]

1.5 Porovnání alternativ

1.5.1 SOAP

Jak již bylo řečeno v kapitole pojednávající o historii REST, SOAP zde byl již před rokem 2000, je tedy starší než samotný architektonický styl REST. Jedná se o komunikační protokol, který využívá zprávy ve formátu XML k ověřování, autorizaci a spouštění vzdáleného kódu. Snaží se poskytovat rozhraní pro vzdálené provádění metod. Hlavním problémem SOAP je z důvodu závislosti na formátu XML podrobnost zápisu. Z tohoto důvodu se nejedná o nejoptimalizovanější způsob odesílání dat po drátě. [18]

1.5.2 GraphQL

GraphQL je dotazovací jazyk a běhové prostředí na straně serveru pro aplikační programovací rozhraní (API). Slouží pro definování dotazů pomocí systému pro práci s daty. Dotazovací

jazyk není vázán na žádnou konkrétní databázi nebo úložiště, místo toho je podporován stávajícím kódem a daty. [14]

GraphQL byl původně vyvinut společností Facebook v roce 2012, přičemž využití mělo být u mobilních aplikací. V roce 2015 byl projekt zpřístupněn jako open source. Nyní na něj dohlíží GraphQL Foundation. [15]

GraphQL je navržen tak, aby API byla rychlá, flexibilní a přívětivá pro vývojáře. Lze ho dokonce využívat v integrovaném vývojovém prostředí (IDE) známém jako GraphiQL. Jazyk umožňuje vývojářům vytvářet požadavky, které získávají data z více zdrojů dat v jediném volání API. Dále poskytuje správcům rozhraní API flexibilitu při přidávání nebo odstraňování polí bez dopadu na existující dotazy. Vývojáři mohou vytvářet API pomocí libovolně specifikovaných metod, přičemž specifikace GraphQL zajistí, že metody budou klientům fungovat předvídatelným způsobem. [15]

1.5.3 Falcor

Falcor je knihovna JavaScript pro efektivní načítání dat. Umožňuje reprezentovat veškeré vzdálené zdroje jako model jedné domény prostřednictvím virtuálního grafu JSON. Kód vývojáře je tak stejný bez ohledu na to, zda se data nachází v paměti u klienta nebo na skrze síť na serveru. [17]

Falcor je open source projekt od společnosti Netflix, který má tendenci být vnímán jako součást většího zásobníku používaného Netflixem. [16]

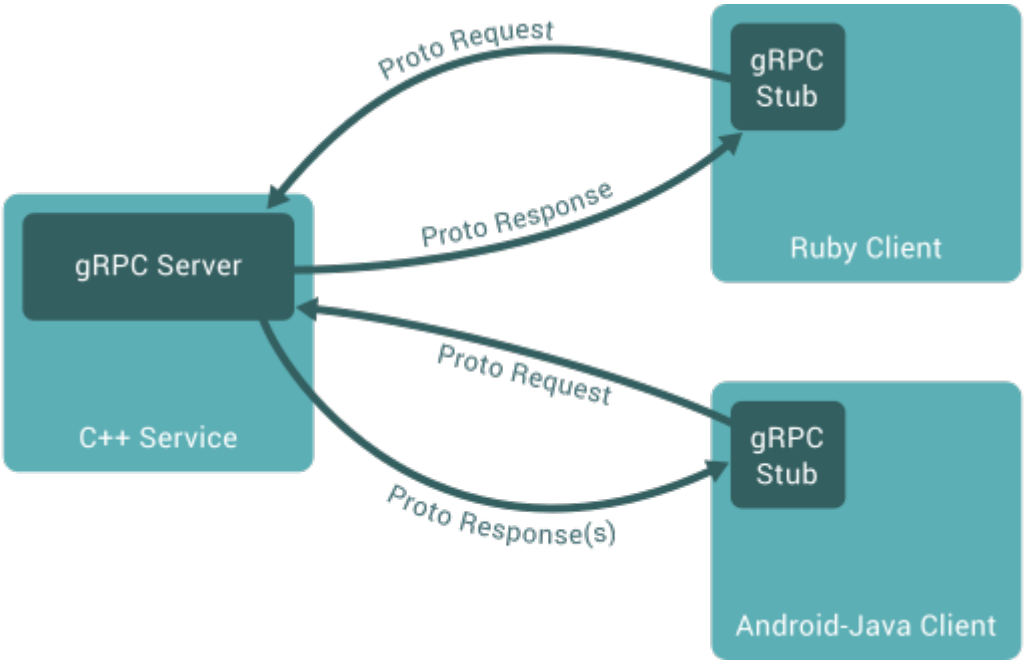
Data jsou prezentována v jednom velkém objektu JSON. Zachovává myšlenku cesty z REST API. Cesta se využívá k procházení objektu JSON s čistším a rychlejším API. Dále zachovává myšlenku referencí z REST API. Vzhledem k tomu, že objekty JSON využívají stromový formát, zobrazuje Falcor graficky objekty s odkazy, čímž v podstatě vytváří cykly. Tyto cesty a reference jsou vhodné pro aplikace fungující v reálném čase, neboť mohou snadno odkazovat a porovnávat odeslaná data. [16]

1.5.4 gRPC

Zatímco GraphQL i Falcor pracují s formátem JSON, gRPC přijímá a vrací zprávy prostřednictvím protokolu buffer (protobuf). Na rozdíl od REST, který je úzce svázán s protokolem HTTP, využívá gRPC model přímo z programovacích jazyků. Součástí tak bývají datové struktury jako jsou funkce a metody. Podobně jako Falcor se i gRPC hodí pro aplikace určené ke streamování dat. To je způsobené především z výhod protokolu HTTP/2, nikoliv ze

samotného designu. Rozhraní gRPC je silně typové, zprávy tak musí dodržovat předem definovanou strukturu navrženou ke snížení režie výkonu a rizika chyb. [16]

Klientské aplikace mohou přímo volat metodu serverové aplikace na jiném počítači jako by se jednalo o místní objekt, což usnadňuje vytváření distribuovaných aplikací a služeb. Server implementuje toto rozhraní a spustí gRPC server, který obsluhuje klientská volání. Klient má na své straně stub, který poskytuje stejné metody jako server. [17]



Obrázek 3: Architektura gRPC [17]

2 SOUHRN IMPLEMENTACÍ V PROGRAMOVACÍM JAZYCE PHP

2.1 Laravel

Laravel je open-source API framework, který umožňuje tvorbu Rest API prostřednictvím architektury Model-View-Controller (MVC). Díky této architektuře se programátoři mohou zaměřit pouze na jednu část vývoje v jeden okamžik. Toho lze docílit rozdělením vývoje API na tři části: Model, View a Controller. [21]

Laravel poskytuje mnoho možností pro konfiguraci, což je jedna z jeho hlavních předností. Dále obsahuje mnoho rozšíření, které uživatelé mohou využít za účelem vývoje vlastního API. Webové stránky poskytují četné množství materiálů, které pomohou vývojářům v procesu vytváření API. To ho činí ideální volbou pro začátečníky. [21]

Laravel aplikaci je možné vytvořit prostřednictvím nástroje Composer příkazem: *composer create-project laravel/laravel example-app*

Příklad 3: Příkaz pro vytvoření Laravel aplikace [23]

Po vytvoření projektu lze na adrese localhost:8000 spustit lokální vývojový server Artisan pomocí příkazu serve:

cd example-app
php artisan serve

Příklad 4: Spuštění vývojového serveru Laravel [23]

Pro nastavení databáze lze využít konfiguračního souboru „.env“. Výchozí nastavení určuje připojení skrze databázi MySQL na adrese 127.0.0.1. Lze však i využít databázi SQLite. Toho lze docílit pouhým vytvořením prázdného souboru databáze SQLite, obvykle v adresáři „database“. Následně stačí aktualizovat „.env“konfigurační soubor. [23]

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

Příklad 5: Nastavení databáze ve frameworku Laravel [23]

Laravel poskytuje širokou škálu možností při vytváření směrování mezi stránkami. Samozřejmostí je podpora veškerých REST operací. Nejjednodušší variantou je přímé

zpracování dané cesty anonymní funkcí, rovněž je možné zpracování přenechat *Controlleru* nebo zavolat požadovaný *View*. V neposlední řadě je možné nastavovat parametry prostřednictvím regulárních výrazů. [24]

```
Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

Ukázka kódu 1: Směrování s Regulárním Výrazem v Laravelu [24]

Laravel se řadí mezi full-stack frameworky, neposkytuje tedy pouze nástroje pro tvorbu back-end části, ale usnadňuje vývojářům i tvorbu vzhledu. Laravel poskytuje dva hlavní způsoby, jakým lze řešit tvorbu front-endu. Prvním způsobem je PHP a šablonovacího jazyka Blade. Syntaxe by mohla vypadat následovně. [25]

```
<div>
    @foreach ($users as $user)
        Hello, {{ $user->name }} <br />
    @endforeach
</div>
```

Ukázka kódu 2: Ukázka iterace v jazyce Blade [25]

Druhým způsobem tvorby vzhledu je využití Javascriptu frameworků, jako je React nebo Vue. K tomu je využit nástroj Inertia.js. Po instalaci nástroje Inertia a vytvoření směrování a Controlleru stačí, aby daný Controller místo šablony Blade vracel Inertia stránku. [25]

```
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Inertia\Inertia;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param int $id
     * @return \Inertia\Response
     */
    public function show($id)
    {
        return Inertia::render('Users/Profile', [
            'user' => User::findOrFail($id)
        ]);
```

```
        }
}
```

Ukázka kódu 3: Použití nástroje Inertia ve frameworku Laravel [25]

2.2 Lumen

Lumen je lehký a rychlý PHP framework vytvořený na základě frameworku Laravel. Poskytuje mnoho funkcí, které se zabývají například autentifikací, autorizací, pamětí cache, databází nebo šifrováním. Jedná se o jeden z nejrychlejších micro frameworků. [20]

Tento framework je navržen speciálně pro vytváření REST API. Aby toho docílil, odstranil všechny funkce Laravelu, které omezují bezstavová API. Místo toho se zaměřuje na ty funkce, které jsou užitečné pro REST API. To je také důvod, proč je Lumen tak lehkým frameworkem. Stále však obsahuje mnoho z výhod, které Lumen z Laravelu převzal. Vývojáři tak mohou využít mnoho konfigurací pro vytváření svých aplikačních rozhraní. [21]

Lumen poskytuje funkci pro každou ze základních metod protokolu HTTP:

- POST
- GET
- PUT
- DELETE

Pro používání Lumenu je nutné mít nainstalovaný PHP ve verzi 7.2 nebo novější a nástroj Composer. K užitku při vytváření REST API mohou přijít i rozšíření jako PDO, OpenSSL nebo Mbstring. [21]

2.3 Guzzle

Guzzle je PHP HTTP klient a odlehčený framework pro vytváření webových služeb REST. Poskytuje nástroje, pomocí kterých lze rychle vytvořit klienta webových služeb. Jedná se o výkonné API s jednoduchým rozhraním. Využívá výhody protokolu HTTP/1.1 pro trvalá připojení a paralelní požadavky. Obsahuje systém zásuvných modulů založený na událostech Symfony2, díky kterému lze zcela upravit chování požadavku. Pro testování klientů je možné využít zahrnutý webový server Node.js. [20]

Guzzle lze nainstalovat obdobně jako Laravel či Lumen prostřednictvím nástroje Composer příkazem:

composer require guzzlehttp/guzzle:^7.0

Příklad 6: Instalace frameworku Guzzle [26]

Použití frameworku je následně jednoduché. Pouze vytvoříme a odešleme požadavek skrze Guzzle HTTP klienta, který vrátí odpověď, kterou lze následně zpracovat dle specifických požadavků. Je možné posílat rovněž asynchronní požadavky či požadavky vyžadující synchronizaci. K tomuto účelu lze využít třídu „Promise“. [26]

```
$client = new GuzzleHttp\Client();
$res = $client->request('GET', 'https://api.github.com/user', [
    'auth' => ['user', 'pass']
]);
echo $res->getStatusCode();
// "200"
echo $res->getHeader('content-type')[0];
// 'application/json; charset=utf8'
echo $res->getBody();
// {"type":"User"...'

// Send an asynchronous request.
$request = new \GuzzleHttp\Psr7\Request('GET', 'http://httpbin.org');
$promise = $client->sendAsync($request)->then(function ($response) {
    echo 'I completed! ' . $response->getBody();
});
$promise->wait();
```

Ukázka kódu 4: Příklad použití Guzzle [27]

Uživatel má možnost nastavit celou řadu parametrů požadavku, jako je nastavení hlaviček, parametrů query stringů, timeoutu, těla požadavku a mnoho dalšího. [28] Příkladem může být následující nastavení hlaviček:

```
// Set various headers on a request
$client->request('GET', '/get', [
    'headers' => [
        'User-Agent' => 'testing/1.0',
        'Accept'     => 'application/json',
        'X-Foo'      => ['Bar', 'Baz']
    ]
]);
```

Ukázka kódu 5: Nastavení hlaviček ve frameworku Guzzle [28]

2.4 Epiphany

Jedná se o jednoduchý PHP framework pro vývoj webových služeb. Epiphany je rychlý, jednoduchý a využívá REST. [20] Hodí se pro vývojáře, kteří chtějí jednoduchý a jasný proces vytváření API. [21] Dokumentace poskytuje několik konvencí, které dle tvůrců vedou k dobře napsanému kódu, umožňuje však použití libovolného stylu a nikdy nediktuje, jak by měli vývojáři psát nebo strukturovat jejich aplikaci. [20]

Webové služby Epiphany obsahují následující moduly:

- Route – Směrovací knihovna pro mapování cest k funkcím.
- Api – Pomocný modul pro vytváření soukromých a veřejných rozhraní API.
- Session – Knihovna relací, která podporuje nativní relace PHP, APC a Memcached.
- Database – Jednoduché rozhraní k ovladači MySQL PDO.
- Cache – Knihovna pro snadné ukládání do mezipaměti podporující APC a Memcached.
- Config – Konfigurační knihovna založená na ini podporující přetížení. [20]

Cesty se jako u většiny frameworků nedefinují fyzicky, nýbrž programově. Nejprve je nutné inicializovat modul cesty, to lze provést funkcí „Epi::init“. Následně stačí funkcí „getRoute“ vytvořit námi požadované REST metody pro konkrétní cesty a k tomu definovat funkce, které volané požadavky obslouží. Jako poslední je nutné zavolat funkci „run“, která zahájí zpracování požadavku. [29]

```
Epi::init('route');
getRoute()->get('/', 'home');
getRoute()->get('/contactus', 'contactUs');
getRoute()->post('/contactus', 'contactUsPost');
getRoute()->run();
```

Ukázka kódu 6: Nastavení cest ve frameworku Epiphany [29]

Framework dále umožňuje přesměrování na určitou adresu, například při nastání specifického HTTP kódu. Dále je možné načítat cesty z externích souborů či nastavovat cesty prostřednictvím regulárních výrazů. Aby bylo možné při větším počtu cesty vhodně rozdělit, umožňuje framework volání tříd místo funkcí. [29]

```
getRoute()->get('/profile/(\d+)', array('Site', 'profile'));
getRoute()->get('/profile/(\d+)/photos', array('Site', 'photos'));
class Site {
    public static function profile($userId) {
        // logic for profile page with access to user id from URL
    }
```

```
    public static function photos($userId) {
        // logic for photo page with access to user id from URL
    }
}
```

Ukázka kódu 7: Použití tříd a Regulárních výrazů ve frameworku Epiphany [29]

3 NÁVRH A REALIZACE REST API FRAMEWORKU

Tato kapitola se zabývá vývojem frameworku využívajícího rozhraní REST. Ne úvod je tvořený framework popsán, následně jsou představeny požadavky, které by měl projekt splňovat. Poté je detailně, včetně ukázek nastavení a zdrojových kódů, popsána aplikační logika. Na závěr je pro daný framework vytvořen projekt, na kterém je ukázáno jeho použití pro praktické účely.

3.1 Popis frameworku

Cílem tohoto projektu bylo vytvoření frameworku v programovacím jazyce PHP, který by uživateli umožnil správu back-endové části webové aplikace. Ke komunikaci frameworku je využito rozhraní REST, které je v současné době hojně rozšířené, díky čemuž je mnohem jednodušší framework implementovat do webové aplikace. Velký důraz je kladen na jednoduchost a efektivitu, aby framework mohl využít i programátor, který nedisponuje pokročilými znalostmi, ať už ve tvorbě back-endové části webové aplikace, nebo v práci s programovacím jazykem PHP. Z tohoto důvodu veškerá uživatelská interakce probíhá prostřednictvím konfiguračního souboru, ve kterém uživatel jednoduše definuje cesty spolu s informacemi, kterou REST metodu si přeje použít, z jakého zdroje a jaká data si přeje získat. Díky tomuto přístupu není nutné vytvářet vlastní datové struktury pro každou REST metodu každé cesty, místo toho je využita abstraktní vrstva, která veškerou komunikaci obstarává na základě nastavení v konfiguračním souboru.

Framework umožňuje vytvářet dotazy na dva základní typy umístění dat. Prvním typem je dotaz na databázi MySQL, kde má vývojář možnost využít velkého množství nástrojů jazyka SQL. V konfiguračním souboru je možné nejen vytvářet jednoduché dotazy, ale rovněž spojovat tabulky nebo definovat sloupce včetně aliasů. Samozřejmostí je i využití funkcí, jako jsou například funkce SUM, AVG a mnoho dalších. Pro upřesnění výsledků jsou použity klauzule. Framework poskytuje podporu pro klauzule WHERE, GROUP BY (včetně klauzule HAVING), ORDER BY a LIMIT. Veškeré klauzule se dají podrobně upravovat a přiřazovat k parametrům zadaných uživatelem, což poskytuje komplexní nástroj pro manipulaci s daty v databázi.

Framework dále umožňuje posílat HTTP požadavky na libovolný cílový bod v síti internet. Samozřejmostí je i možnost nastavení hlaviček požadavku. Aby bylo zaručeno korektní zpracování odpovědí z cílového serveru, má uživatel možnost nastavení datového formátu, přičemž podporovány jsou formáty JSON, XML nebo klasický text.

Konfigurační soubor umožňuje komplexní správu parametrů. Uživatel má možnost parametry posílat přímo v *adrese cesty* nebo prostřednictvím *query stringu*. Rovněž je možné určit volitelnost parametrů nebo výchozí hodnotu. Samozřejmostí je i typové kontrola, kdy je možné parametru specifikovat, jakého datového typu má být. Kontrola se pak provádí automaticky v rámci zpracování požadavku.

Aby bylo možné zaručit exkluzivní přístup pro určité cesty, využívá framework vlastní autorizační systém využívající autorizaci HTTP. Správce má možnost vytvoření libovolného množství uživatelů a rolí, které jsou ukládány do databáze MySQL.

3.2 Požadavky na framework

Tato kapitola pojednává o požadavcích na framework. Nejprve jsou definovány funkční požadavky. Poté je pozornost věnována rovněž nefunkčním požadavkům frameworku.

3.2.1 Funkční požadavky

- Framework umí komunikovat s databází MySQL.
- V rámci databáze je možné definovat značnou část jazyka MySQL, tj. spojování tabulek, výběr sloupců, využívání funkcí, nastavování aliasů či použití klauzulí.
- Framework dokáže rozesílat HTTP dotazy v rámci sítě internet.
- Pro použití stačí modifikovat konfigurační soubor skeleton aplikace.
- Konfigurace frameworku umožňuje vytvořit více REST metod pro jednu cestu.
- V rámci konfigurace je možné definovat parametry požadavku prostřednictvím adresy URL či query stringu.
- Konfigurační soubor umožňuje definovat oprávnění pro získání dat prostřednictvím HTTP autorizace.
- Datové typy parametru je možné automaticky kontrolovat.
- Parametry mohou být volitelné s možností nastavení výchozí hodnoty.
- U parametru lze nastavit, zda ho uživatel může upravit.
- V rámci HTTP dotazů lze nastavit hlavičky požadavku.
- HTTP dotazy umožňují dle nastavení zpracovávat formáty JSON, XML nebo klasický text.
- Framework umožňuje přidávat, zobrazovat, odebírat, upravovat či přiřazovat autorizační role.
- Framework umožňuje přidávat, zobrazovat, odebírat či upravovat uživatelské účty využité pro autorizaci.

- Konfigurační soubor umožňuje nastavení vlastních zpráv při úspěšném zpracování požadavku.

3.2.2 Nefunkční požadavky

- V rámci implementace frameworku bude možné instalovat moduly třetích stran.
- Framework bude získávat data prostřednictvím rozhraní REST.
- Odezva zobrazení výsledné odpovědi bude kratší než deset sekund.
- Konfigurační soubor bude obsahovat příklady pro snadné testování.
- Framework musí být schopni používat uživatelé bez znalosti PHP.

3.3 Aplikační logika

Kapitola pojednává o aplikační logice, na jejímž principu celý framework funguje. Postupně jsou představeny veškeré důležité komponenty, díky kterým může být framework funkční. Prvky jsou představovány v postupném pořadí dle využití.

3.3.1 Konfigurační soubory

Konfigurační soubory aplikace je nutné vytvořit v adresáři „config“ v domovském adresáři projektu. V případě použití skeleton aplikace jsou oba konfigurační soubory včetně adresáře vytvořeny, pro úspěšné použití aplikace je pouze vyžadováno tyto soubory modifikovat. Konfigurační soubory jsou ve formátu YAML, který je vhodný pro nastavování aplikací různého druhu.

Prvním konfigurační soubor nese název „database.yaml“ a slouží pro nastavení připojení k databázi. V současné době aplikace podporuje pouze databázi MySQL, v budoucnu však lze podporu rozšířit i o další typy databází. To bude možné provést pouhou úpravou hodnoty *adapter*, která určuje typ databáze. Následuje nastavení potřebných parametrů připojení k danému typu databáze. Pro databázi MySQL se jedná o *jméno databáze*, *adresu databázového serveru*, *přihlašovací údaje* a *znakovou sadu*. Databáze je rovněž využívaná pro autorizaci, neboť se do ní ukládají uživatelské účty a role oprávnění. Z tohoto důvodu nelze aplikaci využívat bez připojení k databázi. Poslední hodnotou k nastavení je prefix tabulek pro autorizaci.

```
adapter: mysql
db_name: 'rest_students'
host: 'localhost'
username: 'furst'
password: 'Password1*'
```

```
charset: 'utf8'
auth_prefix: 'auth_'
```

Příklad 7: Nastavení databáze

Druhá a nejdůležitější konfigurace se nachází v souboru „config.yaml“. Zde má uživatel možnost nastavit veškeré chování frameworku. Díky tomuto přístupu nemusí programátor vytvářet žádný kód a framework je tak rychle a jednoduše použitelný.

Záznam v konfiguračním souboru začíná vždy začíná lomítkem a cestou, pro kterou chceme nastavit funkcionalitu. Aplikace podporuje dva základní typy psaní cesty:

/students
/student/{id}

Příklad 8: Ukázky cest

Prvním přístupem je pouze název cesty. V případě druhého typu je možné definovat jeden či více parametrů. Parametr se vždy definuje ve složených závorkách a jeho konkrétní nastavení lze specifikovat v pozdější části. Lze tak například definovat datový typ, kterého musí atribut být. Pokud je server PHP spuštěn na lokální úložišti s portem 8080 a atribut „id“ je typu integer, bude tento zápis splňovat například adresa:

http://localhost:8080/student/5

Příklad 9: URL adresa s parametrem

Po nastavení cesty je nutné specifikovat, které metody REST pro zvolenou cestu budou aktivní. Jednotlivé metody je nutné psát velkým písmenem. Jelikož je možné stanovit více REST metod pro jednu cestu, je nutné tyto hodnoty definovat jako list. Ten se ve formátu YAML definuje znakem pomlčky. Nastavení tak může vypadat následovně:

```
/student/{id}:
  - GET:
      # Další nastavení
  - PUT:
      # Další nastavení
  - DELETE:
      # Další nastavení
```

Příklad 10: Nastavení REST metod

Po předchozím nastavení je pro každou zvolenou metodu REST nutné nastavit její funkcionalitu. Základním údajem k nastavení je „type“, kde předpokládané hodnoty jsou „DB“ nebo „HTTP“. Hodnota „DB“ označuje přístup k databázi. Naopak „HTTP“ označuje odeslání

dotazu na libovolnou adresu v síti internet. Zvolení této hodnoty má výrazný dopad na následující nastavení.

Nastavení databáze

V případě zvolení databáze lze nastavovat následující hodnoty:

Tabulka 3: Komponenty nastavení databáze

Název	Název v konfiguračním souboru	Povinné	Popis
Název tabulky	table	Ano	Označuje jméno tabulky v databázi, pro kterou chceme provést transakci
Výběr sloupců	columns	Ano (pro metodu GET)	Definuje seznam a nastavení sloupců, které lze zobrazit v rámci metody GET.
Spojení tabulek	joins	Ne	Umožní spojit více tabulek při výběru dat metodou GET.
Parametry	parameters	Ano (při použití parametru v cestě)	Lze použít pro nastavení uživatelských parametrů včetně lokace, volitelnosti, výchozích hodnot nebo datového typu. Při přístupu k databázi tyto parametry odpovídají SQL klauzulím WHERE, GROUP BY, ORDER BY a LIMIT.
Tělo požadavku	content	Ano (pro metody POST, PUT a PATCH)	Představuje hodnoty, které uživatel musí zaslat v těle požadavku. Tyto údaje obvykle představují pole hodnot, které chceme přidat či upravit.
Úspěšná zpráva	response: message	Ne	V případě úspěšného provedení transakce lze nastavit specifickou zprávu.
Autorizace	authorization	Ne	Umožní stanovit seznam rolí, které mohou daný požadavek odeslat.

Výběr sloupců v metodě GET umožňuje definovat, které sloupce ze zadané tabulky chce uživatel zobrazit. V případě spojení více tabulek lze definovat sloupce z obou tabulek. Pokud však sloupce obou tabulek mají stejné jméno, je nutné sloupce odlišit přidáním názvu tabulek a tečky před název sloupce. Aplikace rovněž umožňuje výběr všech sloupců z tabulky či tabulek. To lze provést vložením „*“ jako hodnoty prvního sloupce. Každý sloupec má rovněž možnost podrobnějšího nastavení. Kromě samotného názvu lze definovat i alias a funkci. Alias specifikuje, jak se má jmenovat sloupec vrácený z dotazu. Hodnota funkce odpovídá funkcím jazyka SQL. Je tedy možné pro konkrétní sloupec nastavit například funkci „count“, která vrátí počet daného sloupce ve výsledném zobrazení.

```
/students/marks/average:
  - GET:
      type: DB
      table: students
      columns:
        - name
        - surname:
```

```
        alias: lastname
    - mark:
        function: avg
        alias: Study average
joins:
  - marks:
        type: join
        column: students.student_id
        pk: marks.student_id
```

Příklad 11: Nastavení výběru sloupců a spojení tabulek pro databázi

Následující nastavení vrátí na adrese cesty */students/marks/average* z databázové tabulky *students* spojené s tabulkou *marks* sloupce *name*, *surname* a *mark*, přičemž *surname* bude vypsáno jako *lastname* a na sloupec *mark* bude aplikována funkce *avg*. Výsledek bude vypsán jako „*Stude average*“. Toto nastavení tedy vrátí pro každého studenta jeho jméno, příjmení a studijní průměr.

Spojení tabulek je možné provést přidáním parametru „joins“ a definováním seznamu tabulek s udáním typu spojení a sloupci, podle kterých se má spojení vykonat. Jako typ spojení lze zadat v případě databáze MySQL hodnoty „join“, „left join“ či „right join“. Hodnota parametru „column“ označuje název sloupce první tabulky a hodnota parametru „pk“ označuje název sloupce spojované tabulky. Zde se obvykle bude jednat o primární klíč. Pokud jsou názvy sloupců obou tabulek identické, je opět nutné je odlišit specifikováním názvu tabulky před sloupcem.

Nastavování parametrů poskytuje komplexní nástroj pro práci s MySQL databází. Seznam parametrů je možné specifikovat přidáním „parameters“ a nastavením potřebného nastavení. Každý parametr musí mít nastavenou hodnotu „editable“. Tato hodnota určuje, zda má uživatel možnost tento parametr měnit. Požadované hodnoty jsou „true“ či „false“. Pokud je hodnota nastavena na false, uživatel nemůže tento parametr žádným způsobem měnit a aplikace použije hodnoty nastavené v rámci konfiguračního souboru. To může být výhodné, pokud chceme výsledky například řadit určitým způsobem a nechceme, aby toto pořadí nemohl uživatel měnit. V takovém případě je nutné nastavit parametr „default“, který určuje výchozí a v tomto případě jedinou hodnotu, která se má použít. Pokud je však hodnota „editable“na „true“, je nutné dále specifikovat, z jakého místa může uživatel parametr měnit. To je možné upřesnit přidáním „location“, přičemž očekávané hodnoty jsou „query“ nebo „path“. Aby bylo možné parametry identifikovat, je nutné, aby název parametru v query stringu odpovídal názvu v konfiguračního souboru. V případě parametru v cestě je nutné pouze dodržet pořadí.

http://localhost:8080/student/5

http://localhost:8080/students?id=5

Příklad 12: Typy parametrů v URL adrese

Pokud je hodnota „editable“ „true“, je rovněž nutné nastavit, zda je tento parametr povinný. V případě povinného parametru se požadavek neakceptuje, pokud daný parametr chybí. Avšak pokud je parametr volitelný, je možné dále nastavit výchozí hodnotu, která se použije, pokud parametr není zadán. Aby bylo možné efektivně kontrolovat zadané parametry, poskytuje konfigurační soubor rovněž možnost pro nastavení datového typu. To lze provést nastavením hodnoty type. Aplikace podporuje následující datové typy:

- boolean nebo bool
- integer nebo int
- double
- string

Pokud hodnota zadaná uživatelem neodpovídá zadanému typu, je požadavek zamítnut a uživateli vrácena chyba. Pro práci s databází MySQL jsou definované čtyři základní typy parametrů. Jedná se o parametry „limit“, „order by“, „group by“ a podmínkové parametry. Parametr „limit“ určuje počet vrácených výsledků. Nastavení „order by“ pak definuje řazení, kdy je možné nastavit buďto řazení vzestupné (ASC) nebo sestupné (DESC). V rámci „default“ hodnoty se stanoví seznam sloupců, podle kterých se má aplikovat řazení.

```
parameters:
  - limit:
      editable: true
      location: query
      required: false
      default: 200
  - order by:
      editable: false
      order: DESC
      default:
        - surname
        - name
```

Příklad 13: Nastavení specifických parametrů databáze

Nastavení „group by“ určuje seskupování, obvykle při použití jedné z databázových funkcí. Podobně jako v jazyce SQL je možné nastavit klauzuli „having“, která určuje, za jakých podmínek jsou hodnoty řazeny. V následujícím případě jsou hodnoty řazeny, pokud je průměrný průměr známek menší než dva, přičemž tuto hodnotu může uživatel upravit

parametrem „avg“ v query stringu. Nastavení parametru „having“ odpovídá nastavení obecného parametru, který bude popsán následovně.

```
/students/marks/average:
  - GET:
      type: DB
      table: students
      columns:
        - name
        - surname:
            alias: lastname
        - mark:
            function: avg
            alias: Study average
      joins:
        - marks:
            type: join
            column: students.student_id
            pk: marks.student_id
      parameters:
        - group by:
            editable: false
            default: students.student_id
            having:
              - avg:
                  editable: true
                  type: integer
                  location: query
                  required: false
                  function: avg
                  represents: mark
                  operator: <
                  default: 2
```

Příklad 14: Kompletní nastavení cesty s využitím databáze a parametrem "group by"

/students/marks/average?avg=4
/students/marks/average

Příklad 15: Možnosti cest s volitelným parametrem

Obecné nastavení parametru odpovídá při použití databáze MySQL klauzuli WHERE a je definované libovolným názvem parametru. Kromě obecných hodnot „editable“, „type“, „location“, „required“ a „default“ lze rovněž nastavit hodnoty „represents“ a „operator“. Hodnota „represents“ určuje, na který sloupec má být aplikovaný zadaný operátor. Nejčastějším použitím je operátor „=“ s povinným parametrem, který říká, že hodnota daného sloupce musí odpovídat zadanému parametru. V následujícím příkladu jsou zobrazeny zadané sloupce z tabulky „students“ a „cities“ pro studenta, jehož identifikátor „student_id“ odpovídá hodnotě parametru „id“ zadaného v cestě požadavku.

```
/students/{id}:
  - GET:
```

```
      type: DB
      table: students
      columns:
        - students.name
        - surname:
            alias: lastname
        - cities.name:
            alias: city
      joins:
        - cities:
            type: join
            column: students.city_id
            pk: cities.city_id
      parameters:
        - id:
            editable: true
            type: integer
            location: path
            required: true
            represents: student_id
            operator: =
```

Příklad 16: Kompletní nastavení cesty s využitím databáze a obecným parametrem

```
localhost:8080/students/5
```

Příklad 17: Adresa s parametrem v cestě

```
{
  "status_code": 200,
  "status_message": "OK",
  "data": [
    {
      "name": "David",
      "lastname": "Kral",
      "city": "Praha"
    }
  ]
}
```

Příklad 18: Odpověď z databáze

Aby bylo možné efektivně využívat metody POST, PUT a PATCH, je nutné umožnit uživateli nastavení těla požadavku. V rámci tohoto těla lze zaslat na server požadované informace, na základě kterých bude provedeno vložení či modifikace existujícího záznamu. Každý záznam tak odpovídá jednomu sloupci cílové tabulky. V konfiguračním souboru je tělo požadavku nastavováno parametrem „content“. Následuje seznam hodnot, jejichž název odpovídá názvům v těle požadavku. Pro každý záznam je nutné nastavit, zda je nebo není povinný. To je nutné provést přidáním položky „required“. Podobně jako u definování parametrů se zde nachází automatická validace datového typu. Framework předpokládá, že název identifikátoru odpovídá názvu sloupce u dané tabulky. Je zde však možnost identifikátor libovolně přejmenovat. V takovém případě je nutné explicitně definovat, o který sloupec v databázi se jedná. To lze

provést přidáním hodnoty „represents“ s názvem sloupce v databázové tabulce. Na libovolný záznam je rovněž možné aplikovat hashovací funkci. To je užitečné převážně při zasílání hesla, neboť jinak by uživatel musel odesílat již zahashované hodnoty. Hashovací algoritmus je nutné zvolit takový, který je podporován v PHP funkcí „hash“. Framework předpokládá, že ne všechny hodnoty v těle požadavku musí odpovídat sloupcům pouze v jedné tabulce. Často bude nutné měnit údaje na základě hodnoty z jiné tabulky spojené pomocí cizího klíče. Pro tento účel je možné pro kterýkoliv záznam přidat parametr „refers“, který obdobným způsobem, jaký je použit při spojování tabulky, provede SQL podpříkaz, který na základě zadané hodnoty získá primární klíč. Původní hodnota v hlavním příkazu pak bude nahrazena získaným identifikátorem.

```
/users:
  - POST:
      type: DB
      table: auth_users
      content:
        - username:
            type: string
            required: true
        - password:
            type: string
            required: true
            hash: sha256
        - role:
            type: string
            required: false
            represents: role_id
            refers:
              table: auth_roles
              column: name
              pk: role_id
      authorization:
        - Admin
      response:
        message: User was added successfully
```

Příklad 19: Nastavení metody POST pro databázi

Pro snadnější správu požadavků a odpovědí lze definovat zprávu, která se použije ve třídě Response v okamžiku, kdy je transakce úspěšně dokončena. Pomocí této zprávy je pak jednodušší odpovědi rozpoznávat a logovat. Zároveň se tím eliminuje nutnost vytvářet vlastní zprávu, která by byla ve front-endové aplikaci odeslána uživateli. Tuto zprávu je možné nastavit v parametru „response“ s vnitřním parametrem „message“.

Posledním nastavením v rámci komunikace s databází je možnost definovat autorizaci. Pomocí autorizace lze definovat seznam rolí, které mohou daný požadavek odeslat a zobrazit si tak odpověď. Role je kontrolována prostřednictvím HTTP autorizace, kdy uživatel zadá do

požadavku jméno a heslo, pomocí kterých je zkontrolováno, že daný uživatel existuje a vlastní jednu z požadovaných rolí. V konfiguračním souboru je možné roli přidat v sekci „authorization“.

Nastavení HTTP

Následující kapitola se zabývá nastavováním konfiguračního souboru pro posílání HTTP požadavků. Pro metodu HTTP lze nastavovat následující hodnoty:

Tabulka 4: Komponenty nastavení HTTP

Název	Název v konfiguračním souboru	Povinné	Popis
URL adresa	url	Ano	Adresa URL určující cílový bod, na který je zaslán požadavek.
Hlavičky	headers	Ne	Umožní definovat seznam hlaviček HTTP požadavku.
Parametry	parameters	Ano (při použití parametru v cestě)	Lze použít pro nastavení uživatelských parametrů včetně lokace, volitelnosti, výchozích hodnot nebo datového typu.
Tělo požadavku	content	Ano (pro metody POST, PUT a PATCH)	Představuje hodnoty, které uživatel musí zaslat v těle požadavku. Tyto údaje obvykle představují pole hodnot, které chceme přidat či upravit.
Formát odpovědi	response: format	Ano	Definuje formát, v jakém jsou zaslána data v odpovědi z cílového serveru.
Úspěšná zpráva	response: message	Ne	V případě úspěšného provedení transakce lze nastavit specifickou zprávu.
Autorizace	authorization	Ne	Umožní stanovit seznam rolí, které mohou daný požadavek odeslat.

```
/currency:
  - GET:
      type: HTTP
      url:
https://api.apilayer.com/fixer/convert?to={to}&from={from}&amount={amount}
      headers:
        - apikey: TbXD1h4jFUuZJactGvmIm8twt2kxwOxE
      parameters:
        - to:
            editable: true
            type: string
            location: query
            required: true
        - from:
            editable: true
            type: string
            location: query
            required: true
        - amount:
            editable: true
            type: integer
            location: query
            required: false
            default: 1
```

```
    response:
      format: JSON
```

Příklad 20: Ukázka nastavení HTTP pro převod měn

Nastavování *úspěšné zprávy* a *autorizace* zůstalo při práci s HTTP požadavky stejné. V konfiguračním souboru se tak s nimi pracuje obdobně jako při databáze. Z tohoto důvodu nebudou tyto parametry opakovaně probírány.

Nejdůležitější hodnotou nastavovanou při práci s HTTP požadavky je URL adresa. Tato adresa specifikuje adresu cílového serveru, na který se má požadavek odeslat. V této adrese je možné definovat parametry, které má uživatel možnost měnit. Tyto parametry jsou definované uzavřením uvnitř složených závorek. Aby bylo možné tyto parametry úspěšně použít, je nutné je později nastavit v sekci „parameters“.

Framework volitelně umožňuje nastavení hlaviček. Editace probíhá přidáním názvu hlavičky a hodnoty do seznamu v sekci headers. Použití hlaviček může být vyžadováno cílovým serverem. Obvykle se tímto způsobem posílají například autorizační klíče, pomocí kterých si cílový server ověří práva, na základě kterých zašle příslušnou odpověď.

Nastavování parametrů probíhá velice podobně jako při práci s databází. Jediným rozdílem je, že byly odebrány specifické databázové klauzule. Nastavování tak probíhá obdobně jako u obecného parametru. Jedinou změnou oproti nastavování obecného parametru je odebrání možnosti „represents“, která přiřazuje název parametru k databázovému sloupci a která by zde neměla využití.

Důležitým prvkem k nastavení je formát odpovědi. Vzhledem k tomu, že lze požadavky zasílat na libovolné servery, je nutné předpokládat, že ne všechny servery budou odpovědi zasílat ve formátu JSON. Editace probíhá stanovením hodnoty „format“ v sekci „response“. Podporovány jsou formáty „JSON“, „XML“. Pokud server zasílá odpovědi v klasickém textovém formátu, lze tuto možnost zpracovat nastavením hodnoty formátu na „TEXT“. V případě, že od serveru neočekáváme žádnou odpověď, lze rovněž zadat hodnotu „NONE“. Veškeré typy formátu budou v rámci přijetí odpovědi od cílového serveru detekovány a převedeny na formát JSON, který bude zobrazen uživateli v sekci „data“.

```
{
  "status_code": 200,
  "status_message": "OK",
  "data": {
    "success": true,
    "query": {
      "from": "USD",
```

```
      "to": "CZK",
      "amount": 50
    },
    "info": {
      "timestamp": 1657899723,
      "rate": 24.266015
    },
    "date": "2022-07-15",
    "result": 1213.30075
  }
}
```

Příklad 21: Odpověď cílového serveru pro převod měn

Velice podobně jako v případě použití databáze probíhá i nastavování těla požadavku. Obdobně lze tak v sekci „content“ nastavovat jednotlivé hodnoty včetně datových typů či volitelnosti. Stejně jako v případě parametrů byla odebrána možnost „represents“ a rovněž sekce „refers“, neboť se zde nekomunikuje s databází a není tak nutné spojovat tabulky.

```
/http:
  - GET:
      type: HTTP
      url: https://httpbin.org/get
      response:
        format: JSON
        message: Get successful
      authorization:
        - Admin
  - POST:
      type: HTTP
      url: https://httpbin.org/post
      content:
        - something:
            type: string
            required: true
      response:
        format: JSON
        message: Post successful
  - PUT:
      type: HTTP
      url: https://httpbin.org/put
      content:
        - something:
            type: string
            required: true
      response:
        format: JSON
        message: Put successful
  - DELETE:
      type: HTTP
      url: https://httpbin.org/delete
      response:
```

```
format: JSON
message: Delete successful
```

Příklad 22: Nastavení veškerých REST metod při práci s HTTP typem

3.3.2 Soubor index

Index je jediný veřejný soubor, na který směřují veškeré dotazy. V případě použití skeleton aplikace je tento soubor již vytvořen a nastaven. Nachází se ve složce public. V rámci tohoto souboru dojde k vytvoření instance hlavní třídy frameworku, která se jmenuje „Application“. Aby bylo možné předat řízení zpracování dotazu frameworku, je nutné zavolat funkci „run()“. Po dokončení zpracování dotazu je vhodné uživateli zobrazit odpověď, ať už ve formě požadovaných dat, zprávy či chybové hlášky. To lze provést voláním funkce „send()“ ze třídy „Response“. Soubor indexu se nachází ve složce public měl by být jako jediný viditelný pro běžné uživatele. Toho lze docílit například konfigurací webového serveru.

```php
<?php

use app\Application;

require_once __DIR__ . '/../vendor/autoload.php';

$app = new Application();

$app->run();
$app->getResponse()->send();
```

Ukázka kódu 8: Obsah souboru index.php

3.3.3 Třída Application

Jedná se o hlavní třídu frameworku sdružující veškeré potřebné atributy pro úspěšné zpracování požadavku. Zároveň se jedná o jedinou třídu, u které je možné přistupovat k její instanci v rámci celého frameworku. To může být výhodně pro rychlý přístup k atributům v rámci ostatních tříd. To je zajištěno statickým atributem *app* odkazujícím sám na sebe. Třída dále obsahuje atributy *request*, *response* a *database* a jejich gettery. Všechny tyto atributy jsou inicializovány v konstruktoru. Posledním a pravděpodobně nejdůležitějším atributem je pak *router*. Ten se stará jednak o parsování dat z konfiguračního souboru, ale hlavně kontroluje a porovnává cesty a parametry mezi konfiguračním souborem a přijatým požadavkem. Aby bylo možné snadněji data kontrolovat, přijímá konstruktor námi vytvořené instance tříd *request* a *response*.

```php
public function __construct()
{
    self::$app = $this;
    $this->response = new Response();
    $this->request = new Request();
    $this->database = new Database();
```

```
    $this->router = new Router($this->request, $this->response);
}
```

Ukázka kódu 9: Konstruktor třídy Application

Abychom mohli předat další zpracování požadavku routeru, nachází se v rámci této třídě funkce „run()“, která je volaná z index souboru. Tato funkce nejprve zkontroluje, zda již nedošlo k určité chybě a pokud ne, je zavolána funkce routeru „resolve“, která obstará kontrolu cest. Jednou z možných chyb, které mohou nastat v okamžiku kontroly funkcí *run()* je chyba při parsování z konfiguračního souboru. K tomu dochází již při inicializaci routeru. Pokud tedy nedojde k úspěšné kontrole dat z konfiguračního souboru, bude nastaven příslušný HTTP status kód označující chybu a podmínka pro pokračování zpracování požadavku nebude splněna. Následným krokem tak bude vypsání uživateli zprávu o chybě prostřednictvím funkce *send()* v index souboru. Samotná kontrola chyby probíhá porovnáním status kódu třídy response s kódem 200, který označuje, že nedošlo k chybě.

```
public function run(): void
{
    if($this->response->getStatusCode() == 200) {
        $this->router->resolve();
    }
}
```

Ukázka kódu 10: Funkce run() třídy Application

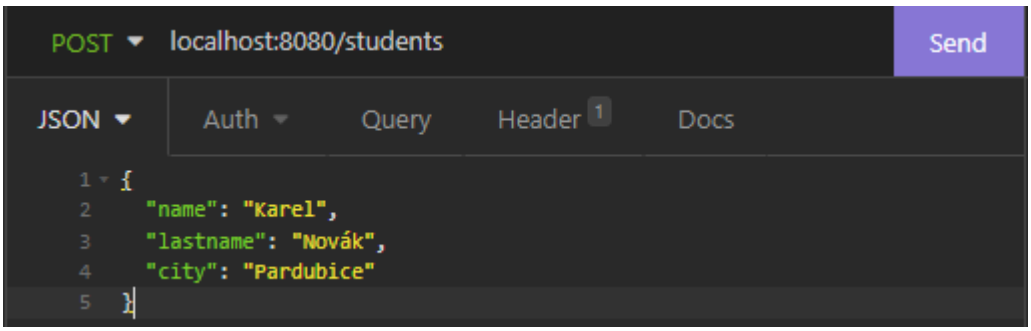
3.3.4 Třída Request

Request představuje příchozí požadavek na server a obsahuje veškeré potřebné informace, jako jsou parametry zadané uživatelem, dále REST metoda, kterou požadavek přišel nebo tělo požadavku, které je využíváno především při metodách POST a PUT. Tyto data jsou kontrolována a zpracovávána dále v aplikaci.

V konstruktoru třídy se nejprve zkontroluje, zda přijatá REST metoda odpovídá jedné z podporovaných metod frameworku. Pokud by tomu tak nebylo, nastavil by se chybový kód „400“ s příslušnou chybovou hláškou a požadavek by dále nebyl zpracováván. Následně jsou uloženy ostatní atributy.

V rámci třídy je možné pomocí get funkce získat pole parametrů, ať už předávaných v rámci query stringu či přímo v cestě. Rovněž je zde funkce „getParameter()“, která je využita při parsování konfiguračního souboru. Tato funkce na základě dat z konfiguračního souboru vyhledá příslušný hledaný parametr. Pokud to nastavení konfigurace umožňuje a parametr není

nalezen, lze využít výchozí hodnotu nastavenou uživatelem. Při nenalezení parametru je vrácena hodnota „null“.



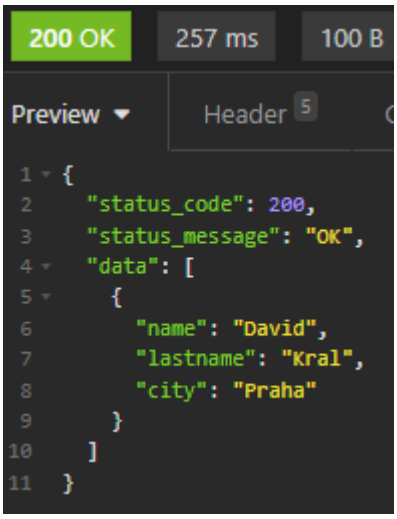
Obrázek 4: Struktura požadavku s metodou POST

3.3.5 Třída Response

Třída Response reprezentuje odpověď, kterou dostane uživatel po zaslání požadavku. V případě, že v průběhu zpracování dotazu dojde k chybě, ať už k chybě v rámci parsování konfiguračního souboru, k nenalezení cesty v routeru či k chybě při zasílání požadavku na server, aplikace chybu zachytí a upraví atributy třídy Response tak, aby uživatel dostal potřebné informace a mohl tak problém opravit.

Třída obsahuje tři atributy. Prvním atributem je „content“, který může být string či pole. Tento atribut uchovává data, která byla vrácena z databáze či z HTTP dotazu. V případě použití metodou GET se jedná o požadovaná data. Druhým atributem je „statusCode“. Jedná se o HTTP kód udávající status zasílaného požadavku. Pokud byl požadavek úspěšně zaslán, bude hodnota statusCode „200“. Pokud však během zpracování dojde k chybě, číslo tohoto kódu se bude lišit. Díky této hodnotě lze jednoduše kontrolovat, zda došlo k chybě či nikoliv. Právě tato hodnota je kontrolována ve funkci „run()“ ve třídě „Application“, pomocí které se rozhoduje, zda bude zpracování pokračovat do routeru nebo skončí a bude vypsána chyba. K tomuto účelu se ve třídě nachází setter, pomocí kterého lze tuto hodnotu jednoduše měnit.

Posledním atributem je „statusText“. Jedná se o jednoduchý textový řetězec, který má za cíl uživateli popsat stav zaslaného požadavku. Výchozí hodnota atributu „statusText“ je „OK“. Pokud dojde k chybě, bude hodnota přepsána. Výchozí hodnotu může uživatel měnit pomocí setteru buď globálně v index souboru, nebo v rámci konfiguračního souboru pro konkrétní REST metodu a cestu.



Obrázek 5: Struktura odpovědi při práci s databází

Třída rovněž obsahuje funkci pro výpis. Jedná se o jedinou funkci zajišťující výpis v rámci celého frameworku. Tato funkce je standartně volána na konci index souboru pouze jednou. Cílem je zobrazit uživateli veškeré potřebné informace o stavu aplikace. Výpis je ve formátu JSON. Při každém požadavku je uživateli zobrazen statusCode a statusText. Pokud je však dostupný context, který označuje, že ze serveru byla přijata data, jsou tato data přidána do výpisu rovněž.

```
public function send(): void
{
    $message = array(
        "status_code" => $this->statusCode,
        "status_message" => $this->statusText);
    if(isset($this->content)) {
        $message['data'] = $this->content;
    }
    echo json_encode($message);
}
```

Ukázka kódu 11: Funkce send() třídy Request

3.3.6 Třída Router

Router je jedna z nejdůležitějších částí frameworku. Jeho hlavní úlohou je kontrola možných cest, jejich porovnání s momentální cestou a v případě shody i volání příslušné funkce abstraktního REST controlleru.

Aby však bylo možné porovnat cesty z konfiguračního souboru, je nejprve nutné konfigurační soubor parsovat, tedy přeložit jeho obsah použitelné datové struktury. K tomuto účelu slouží funkce „loadFromConfig()“ a třída „ConfigParser“. Ve funkci loadFromConfig(), která je volána z konstruktoru, nejprve dojde k načtení celého konfiguračního souboru do formátu

JSON. Pro tento účel je využita třída „YamlConfig“, která pracuje s knihovnou „Yaml“, jež je součástí frameworku Symfony. Pokud je soubor nalezen a načten do formátu JSON, jsou cyklem projety veškeré cesty a jejich příslušné REST metody, přičemž pro každou REST metodu jsou třídou ConfigParser načtena veškerá data z konfiguračního souboru. Tato data, včetně REST metod, cesty, rolí a dalších údajů, vytváří strukturu třídy „Route“ a pole těchto objektů tvoří samotný Router.

Nejdůležitější funkcionalita se nachází ve funkci „resolve()“. Tato funkce je volána ze třídy Application v případě, že nedošlo k chybě při parsování konfiguračního souboru. Cílem této funkce je porovnat cesty, zkontrolovat oprávnění a v případě shody předat řízení třídě RestController.

```
public function resolve(): void
{
    foreach ($this->getRoutes() as $route) {
        if($this->matchExactly($route) || $this->resolvePattern($route)) {
            if($route->getSuccessMessage() != null) {
                $this->response->setStatusText($route->getSuccessMessage());
            }

            $callback = $this->getCallback($route);
            $authManager = new AuthorizationManager($route);
            if(!$callback || !$authManager->checkAuthorization()) {
                return;
            }

            call_user_func($callback, $this->request, $this->response, $route);
            return;
        }
    }

    $this->response->setStatusCode(404);
    $this->response->setStatusText($this->notFoundMessage);
}
```

Ukázka kódu 12: Funkce resolve() třídy Router

Porovnávání probíhá prostřednictvím cyklu, kdy je každá cesta konfiguračního souboru porovnána se skutečnou cestou požadavku. Funkce „matchExactly(Route $route)“ nejprve zkontroluje, zda se shodují cesty i REST metody. Pokud se však neshodují, je nutné zkontrolovat, zda cesta obsahuje jeden z předpokládaných parametrů. Tato kontrola se provádí funkcí „resolvePattern(Route $route)“.

Funkce resolvePattern nejprve obě cesty rozdělí na jednotlivé argumenty cesty, které jsou oddělené lomítkem. Poté jsou kontrolovány metody REST a délky argumentů, neboť v případě neshody lze s jistotou určit, že cesty neodpovídají. Následně jsou postupně procházeny a porovnávány jednotlivé argumenty. Pokud argument obsahuje dle konfiguračního souboru parametr (tedy hodnota se nachází ve složených závorkách) a parametr odpovídá lokací i

datovým typem, lze s jistotou určit shodu. Pokud všechny parametry odpovídají vzoru a ostatní argumenty se shodují, vrací tato funkce hodnotu „true“.

```
private function resolvePattern(Route $route): bool {
    $requestURL = trim($this->request->getUrl(), '/');
    $splRequestURL = explode('/', $requestURL);

    $routeURL = trim($route->getPath(), '/');
    $splRouteURL = explode('/', $routeURL);

    $routeParams = [];
    if($this->request->getRestMethod() == $route->getMethod()
        && sizeof($splRequestURL) == sizeof($splRouteURL)) {
        for($i = 0; $i < sizeof($splRequestURL); $i++) {
            if($splRequestURL[$i] != $splRouteURL[$i]) {
                $paramName = trim($splRouteURL[$i], '{}');
                $paramData = $this->getParameterData($paramName, $route);
                $paramValue = $splRequestURL[$i];
                if(!preg_match('/^\{.*}$/', $splRouteURL[$i])
                    || !$paramData
                    || $paramData->getLocation() != "path"
                    || !settype($splRequestURL[$i], $paramData->getType())
                    || $paramValue != $splRequestURL[$i]) {
                    break;
                }
                $routeParams[$paramName] = $splRequestURL[$i];
            }

            if($i == sizeof($splRequestURL) - 1) {
                $this->request->setRouteParams($routeParams);
                return true;
            }
        }
    }
    return false;
}
```

Ukázka kódu 13: Funkce resolvePattern() třídy Router

Pokud některá z cest odpovídá konfiguračnímu souboru, je v případě existence nastavena nová úspěšná zpráva ve třídě Response. Následně je vytvořen tzv. „callback“. Callback je pole, jehož první hodnotou je instance třídy a druhou hodnotou je název funkce, která se má zavolat. Zde se předává řízení třídě RestController, konkrétně pak funkci, která odpovídá názvu použité REST metody (GET, POST, PATCH, PUT nebo DELETE). Před předáním řízení je nutné zkontrolovat, zda přístup není omezen pouze pro určitou roli a pokud ano, zda uživatel touto rolí disponuje. K tomuto účelu je využita třída „AutorizationManager“. Pokud je vše v pořádku, je funkcí „call_user_func(callable $callback, mixed …$args)“ zavolán předaný callback. Do této funkce jsou vloženy parametry volané funkce, což je $request, $response a použitá $route.

V případě, že ani jedna cesta z konfiguračního souboru neodpovídá skutečné cestě, je uživateli vrácen status kód „404“ označující nenalezenou stránku, čímž je zpracování požadavku ukončeno.

3.3.7 Třída Route

Route představuje jednu cestu z konfiguračního souboru. Kromě samotné cesty se dále eviduje *REST metoda*, *typ požadavku*, samotná *databázová či HTTP data*, *výchozí zpráva*, *seznam autorizačních rolí* a *abstraktní controller*. Typ požadavku značí, zda se jedná o požadavek na databázi nebo na HTTP cílový bod. Data lze pak podle typu požadavku rozdělit na *databázová* a *HTTP*. Databázová data nesou informace o *adaptéru*, *tabulkách*, *sloupcích*, *parametrech* a *těle*. U HTTP dat se místo databázových informacích uchovává adresa *URL*, *formát odpovědi* a *pole hlaviček*. Abstraktní controller pak nese informaci o tom, kterou třídu a funkci v případě shody v routeru zavolat. Všechny tyto informace jsou pro zbytek aplikace dostupné prostřednictvím funkcí get.

3.3.8 Třída RestController

Jestliže router detekuje shodu cest, volá se funkce třídy RestController pojmenovaná podle použité metody REST. Třída RestController tak obsahuje funkce GET, POST, PUT, PATCH a DELETE. Veškeré funkce třídy RestController si jsou podobné. Každá funkce obsahuje podmínku, zda se jedná o dotaz na databázi nebo na HTTP bod. Podle této podmínky se provedou různé operace a odešle se požadavek. Jedinou výjimku představuje funkce PATCH, která pouze zavolá funkci PUT, neboť v rámci aplikační logiky dojde ke stejným operacím.

V případě dotazu na databázi je využita instance třídy „SQLBuilder“, jejíž účelem je sestavení SQL dotazu. Podle typu REST metody dojde k sestavení správného SQL dotazu. Pro metodu GET tak bude SQL příkaz začínat klauzulí „SELECT“, naopak pro metodu POST bude využita klauzule „INSERT“. Pro samotné odeslání příkazu na databázi slouží funkce „send()“ pomocné třídy „SQLManager“.

Pokud je požadavek odesílán ve formě HTTP, není třeba vytvářet dotaz SQL. Místo toho je nutné zkontrolovat, zda URL adresa z konfiguračního souboru neobsahuje parametr, který by bylo nutné doplnit do konečné URL adresy. K tomuto účelu je využita třída „HttpBuilder“. Obdobně, jako v případě použití databáze, se následně volá funkce „send()“, zde však ze třídy „HttpManager“. Tato třída využívá knihovnu „cURL“, která umožňuje odesílat a přijímat data prostřednictvím internetové sítě.

```
public function GET(Request $request, Response $response, Route $route): void
{
    if($route->getRequestType() == RequestType::DATABASE) {
        if($sql = $this->sqlBuilder->buildGetQuery($request, $response, $route)) {
            $sqlManager = new SQLManager($sql, $response, $route->getMethod());
            $sqlManager->send();
        }
```

```
    } else if($route->getRequestType() == RequestType::HTTP
            && $this->httpBuilder->buildQueryString($request, $response, $route)) {
        $httpManager = new HttpManager($route->getData(), $request, $response);
        $httpManager->send();
    }
}
```

Ukázka kódu 14: Funkce GET() třídy RestController

3.3.9 Správa požadavků

Pod pojmem „správa požadavků“ lze označit tvorbu a odeslání požadavku včetně zachycení a zpracování odpovědi. Pro oba typy požadavku je nejprve nutné sestavit dotaz. V případě odesílání požadavku na databázový server je nutné sestavit SQL dotaz. K tomu slouží třída „SQLBuilder“. O následné odeslání požadavku, přijetí dat a nastavení odpovědi se stará třída „SQLManager“. Podobně je to i v případě odesílání dat prostřednictvím metody HTTP. Zde se třídou „HttpBuilder“ nejprve vytvoří finální URL adresa požadavku, který třída HttpManager odešle a nastaví odpověď.

SQLBuilder

Jelikož framework poskytuje velké množství nastavení pro databázovou transakci, je nutné vytvořit systém, který dle uživatelského nastavení konfiguračního souboru provede sestavení SQL dotazu, který bude následně odeslán na databázový server.

Třída SQLBuilder poskytuje čtyři funkce s veřejným modifikátorem přístupu. Jedná se o funkce:

- buildGetQuery
- buildPostQuery
- buildPutQuery
- buildDeleteQuery

Každá z funkcí odpovídá jedné metodě REST a jejím účelem je sestavení daného SQL dotazu. Každá tato funkce přebírá Request, Response a Route. Instance třídy Route poskytuje informace o dané cestě, především pak nastavení dané cesty z konfiguračního souboru. Pomocí Request lze přistupovat k parametrům přijatého požadavku a Response slouží k nastavení odpovědi v případě nastání chyby.

```
public function buildGetQuery(Request $request, Response $response, Route $route):
string|false {
    $sql = "SELECT ";
    $databaseData = $route->getData();
    if($sql = $this->addColumns($sql, $databaseData->getColumns())) {
        $sql .= " FROM " . $databaseData->getTable() . " ";
        $sql = $this->addJoins($sql, $databaseData->getJoins());
        if($sql = $this->addParameters($sql, $databaseData->getParameters(),
```

```
$databaseData->getAdapter(), $request, $response)) {
            return $sql;
        }
    }
    return false;
}
```

Ukázka kódu 15: Funkce buildGetQuery() třídy SQLBuilder

Aby byl kód vhodně modulární, jsou jednotlivé části tvorby SQL dotazu vhodně rozděleny na privátní funkce, z nichž každá sestaví odpovídající část SQL dotazu. Jedná se o funkce:

- addInsertColumns – Přidá seznam sloupců, které jsou explicitně nastavené při vkládání záznamu.
- addInsertValues – K odpovídajícím sloupcům přidá jejich hodnoty.
- addColumns – Přidá seznam sloupců, které mají být vráceny v případě klauzule „SELECT“. Rovněž do dotazu nastaví alias či funkci použitou na sloupec, jsou-li nastavené.
- addJoins – Sestaví dotaz s klauzulí „JOIN“ označující práci s více tabulkami.
- addParameters – Přidá k dotazu postupně klauzule „WHERE“, „GROUP BY“, „ORDER BY“ a „LIMIT“.
- addWhereClauses – Nastaví k SQL dotazu podmínku s klauzulí „WHERE“.
- addGroupByClause – Připojí k SQL dotazu klauzuli „GROUP BY“, je-li použita.
- addOrderByClause – Přidá řazení na základě zadaných sloupců pomocí klauzule „ORDER BY“.
- addLimitClause – Umožní nastavení maximálního množství záznamů vrácených z databáze pomocí klauzule „LIMIT“. Tato funkce rovněž ukazuje, jakým způsobem by bylo možné sestavit dotaz na základě různých typů databází.

```
private function addLimitClause(DatabaseAdapter $adapter, ?LimitClause
$limitClause, Request $request): string|int {
    $sql = "";
    if($limitClause != null) {
        if($param = $this->getParameter($limitClause, $request)) {
            switch ($adapter) {
                case DatabaseAdapter::MYSQL:
                    $sql .= " LIMIT " . $param;
                    break;
                case DatabaseAdapter::ORACLE:
                    $sql .= " FETCH NEXT " . $param . " ROWS ONLY";
                    break;
                case DatabaseAdapter::POSTGRESQL:
                    $sql .= " LIMIT " . $param . " OFFSET 0";
                    break;
            }
        } else {
            return -1;
```

```
        }
    }
    return $sql;
}
```

Ukázka kódu 16: Funkce addLimitClause() třídy SQLBuilder

SQLManager

Jakmile je SQL dotaz sestaven, je nutné ho pouze odeslat. K tomuto účelu slouží třída „SQLManager“, která přebírá SQL dotaz, Response a metodu REST. V konstruktoru si ze třídy Application rovněž uloží připojení k databázi MySQL.

Třída obsahuje pouze funkci „send()“, která s pomocí knihovny „mysqli“ odešle sestavený dotaz na databázový server. Pokud se jedná o metodu GET, dojde taktéž k nastavení přijatých dat do třídy Response. Nastane-li při komunikaci s databází chyba, dojde rovněž k jejímu nastavení.

```
public function send(): void
{
    try {
        if($this->method == RestMethod::GET) {
            $result = $this->connection->query($this->query)
                 ->fetch_all(MYSQLI_ASSOC);
            if (sizeof($result) > 0) {
                $this->response->setContent($result);
                return;
            }
            $this->response->setStatusCode(404);
            $this->response->setStatusText("No data found");
        } else {
            $stmt = $this->connection->prepare($this->query);
            $stmt->execute();
        }

    } catch (mysqli_sql_exception $e) {
        $this->response->setStatusCode(400);
        $this->response->setStatusText($e->getMessage());
    }
}
```

Ukázka kódu 17: Funkce send() třídy SQLManager

HttpBuilder

Třída „HttpBuilder“ obsahuje pouze jednu veřejnou metodu „buildQueryString“, která v případě existence nahradí vzor parametru jeho skutečnou hodnotou. Pokud však požadovaný parametr neexistuje nebo zda datový typ neodpovídá požadovanému typu, vrátí funkce hodnotu „false“ a zpracování požadavku bude přerušeno.

```
public function buildQueryString(Request $request, Response $response, Route
$route): bool {
    $data = $route->getData();
    $uri = $data->getUri();
    foreach ($data->getParameters() as $parameter) {
        $name = $parameter->getName();
```

```
            if($param = $this->getParameter($parameter, $request)) {
                if(!$this->checkType($parameter, $response, $param)) {
                    return false;
                }
                $uri = str_replace("{" . $name . "}", rawurlencode($param), $uri);
            } else {
                return false;
            }
        }
        $data->setUri($uri);
        return true;
    }
```

Ukázka kódu 18: Funkce buildQueryString() třídy HttpBuilder

HttpManager

Třída „HttpManager“ využívá k odesílání HTTP dotazů knihovnu cURL. Ve funkci „send()“ dojde dle použité metody REST k nastavení požadovaných informací, jako jsou hlavičky, tělo požadavku či použitá metoda.

```
public function send(): void
{
    curl_setopt($this->ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($this->ch, CURLOPT_HTTPHEADER, $this->data->getHeaders());

    $content = $this->checkContent();
    if($content != -1) {
        switch ($this->data->getMethod()) {
            case RestMethod::POST:
                curl_setopt($this->ch, CURLOPT_POST, true);
                curl_setopt($this->ch, CURLOPT_POSTFIELDS, $content);
                break;
            case RestMethod::PUT:
                curl_setopt($this->ch, CURLOPT_CUSTOMREQUEST, "PUT");
                curl_setopt($this->ch, CURLOPT_POSTFIELDS, $content);
                break;
            case RestMethod::DELETE:
                curl_setopt($this->ch, CURLOPT_CUSTOMREQUEST, "DELETE");
                break;
            default:
                break;
        }

        $result = curl_exec($this->ch);
        $this->setResponse($result);
        curl_close($this->ch);
    }
}
```

Ukázka kódu 19: Funkce send() třídy HttpManager

Po odeslání a přijetí odpovědi dojde ke zpracování odpovědi funkcí „setResponse()“. Zde nejprve dojde ke kontrole status kódu a následně se dle požadovaného formátu převedou přijatá data na formát JSON, který se vloží do třídy Response.

3.3.10 Databáze

V rámci celého frameworku je využita databáze MySQL. Tato databáze je reprezentována třídou „Database“. Database je vytvářena v rané fázi zpracování, konkrétně při vytváření instance třídy Application. V konstruktoru třídy dojde k načtení údajů z konfiguračního souboru „database.yaml“, pomocí kterých je ustanoveno připojení k databázi. Pokud by byla přidána podpora různých typů databází, bylo by toto připojení nutné rozlišit na základě atributu „adapter“, který označuje typ databázového serveru.

Jelikož databáze představuje prostředí pro ukládání autorizačních dat, tj. uživatelských účtů a rolí, jsou po připojení vytvořeny autorizační tabulky. K tomuto účelu slouží funkce „createAuthTables()“, která provede SQL dotaz s cílem vytvoření požadovaných tabulek, pokud již vytvořeny nejsou.

```
try {
    $connection = new mysqli ($host, $username, $password, $db_name);
    $connection->set_charset($charset);
    $this->connection = $connection;
    $this->createAuthTables($config['auth_prefix'] ?? null);
} catch(mysqli_sql_exception $e) {
    Application::$app->getResponse()->setStatusCode(400);
    Application::$app->getResponse()->setStatusText($e->getMessage());
    return;
}
```

Ukázka kódu 20: Připojení k databázi

3.3.11 Parsování konfiguračního souboru

Aby bylo možné kontrolovat jednotlivé cesty, je nejprve nutné získat z konfiguračního souboru veškeré povolené cesty a jejich dodatečné informace. Z tohoto důvodu se při vytváření instance třídy „Router“ volá privátní funkce „loadFromConfig()“, která načte konfigurační soubor a s využitím iteračního procházení postupně načte veškeré cesty a jejich data, které bude moci kontrolovat router.

Pro načítání veškerých dodatečných dat, jako jsou například sloupce z databáze či adresa URL z HTTP metody, je využita pomocná třída „ConfigParser“. Tato třída obsahuje pouze jednu veřejnou funkci „readData(array $val, RestMethod $method)“, která na základě typu přístupu zavolá příslušné privátní funkce na načítání databázových či HTTP dat.

Parsování dat z konfiguračního souboru probíhá na kaskádovém přístupu. Systém volá funkci zajišťující čtení některého z možných nastavení a funkce kontroluje, zda se v konfiguračním souboru nachází slovo označující některé z požadovaných nastavení. Pokud ano, funkce nastavení přečte a data vloží do struktury třídy, která je vrácena.

```
private function readContent(array $val): array|false
{
    $contents = [];

    if(isset($val['content'])) {
        foreach($val['content'] as $content) {
            $name = key($content);
            $content = $content[$name];
            $type = $content['type'];
            $required = $content['required'];

            if(!isset($name) || !isset($type) || !isset($required)) {
                Application::$app->getResponse()->setStatusCode(400);
                Application::$app->getResponse()->setStatusText("Wrong content
                        structure in config.yaml");
                return false;
            }

            $contents[] = new Content($name, $type, $required, $content['hash']
                                ?? null);
        }
    }

    return $contents;
}
```

Ukázka kódu 21: Funkce readContent() třídy ConfigParser

Následující ukázka představuje čtení těla požadavku. Nejprve je zkontrolováno, zda se v konfiguračním souboru nachází nastavení začínajícím slovem „content“. Pokud ano, funkce očekává seznam atributů těla, kdy každý atribut je definován jménem, kdy pro každé jméno je nutné mít definovaný datový typ a zda je atribut povinný či volitelný. Pokud některá z hodnot není použita, je nastavena chyba a vrácena hodnota „false“. V opačném případě je vráceno pole třídy „Content“ obsahující kontrolované atributy.

Podobným přístupem jsou postupně přečtena a zkontrolována veškerá data, která na konci procesu vytváří hierarchickou strukturu třídy „DatabaseData“ či „HttpData“. Tyto třídy představují jeden z atributů třídy „Route“.

```
private DatabaseAdapter $adapter;
private string $table;
private ?array $columns;
private ?array $joins;
private ?DatabaseParameters $parameters;
private ?array $content;
```

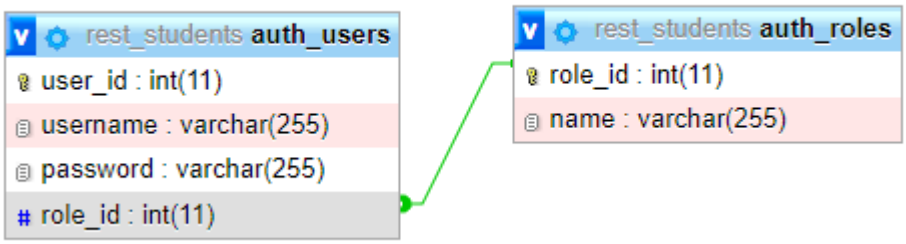
Ukázka kódu 22: Atributy třídy DatabaseData

3.3.12 Autorizace

Autorizace umožňuje definování rolí a uživatelů, kteří smějí posílat požadavek na danou cestu s konkrétní metodou REST. Pokud má být přístup omezen, je nutné v konfiguračním souboru stanovit seznam rolí, které mohou daný požadavek odeslat. Uživatel se v takovém případě musí

identifikovat prostřednictvím uživatelského jména a hesla zaslaného uvnitř požadavku formou HTTP autorizace.

Uživatelské účty a role jsou uchovávány uvnitř databáze MySQL. Při vytváření spojení dojde k vygenerování autorizačních tabulek, pokud již vygenerovány nejsou. Jedná se o tabulku „auth_roles“, která tvoří výčet rolí. Dále se zde nachází tabulka „auth_users“, která uchovává uživatelské účty. Kromě uživatelského jména a hesla je pro účet definovaný cizí klíč odkazující na roli, kterou uživatel zastává. Na heslo je využitý hashovací algoritmus „sha256“.



Obrázek 6: Struktura autorizačních tabulek

Pro správu autorizací slouží třída „AuthorizationManager“, která je využita v metodě při kontrole cest uvnitř routeru. Tato třída přebíhá parametr Route a funkcí „checkAuthorization()“ dojde ke kontrole oprávnění. Systém zkontroluje, zda se na danou cestu vztahuje autorizační omezení. Pokud ano, třída odešle SQL dotaz na server, který vrátí počet záznamů z tabulek uživatelů a rolí, kde uživatelské jméno a heslo odpovídá údajům v požadavku a uživatel zároveň vlastní jednu z povolených rolí. Pokud bude vrácena hodnota „0“, uživatel zadal špatné přihlašovací údaje nebo nevlastní jednu z požadovaných rolí. V takovém případě je přístup zamítnut.

4 APLIKACE REST FRAMEWORKU PŘI TVORBĚ WEBOVÉ APLIKACE

V rámci následujícího příkladu bude představen malý projekt, na kterém bude demonstrováno použití REST API frameworku. Současně bude popsána kompletní instalace a struktura projektu. Samozřejmostí je i ukázka tvorby konfiguračního souboru. Výsledný produkt bude představen v rámci praktické ukázky, kde bude ukázáno, jakým způsobem lze s webovou aplikací pracovat.

4.1 Popis projektu

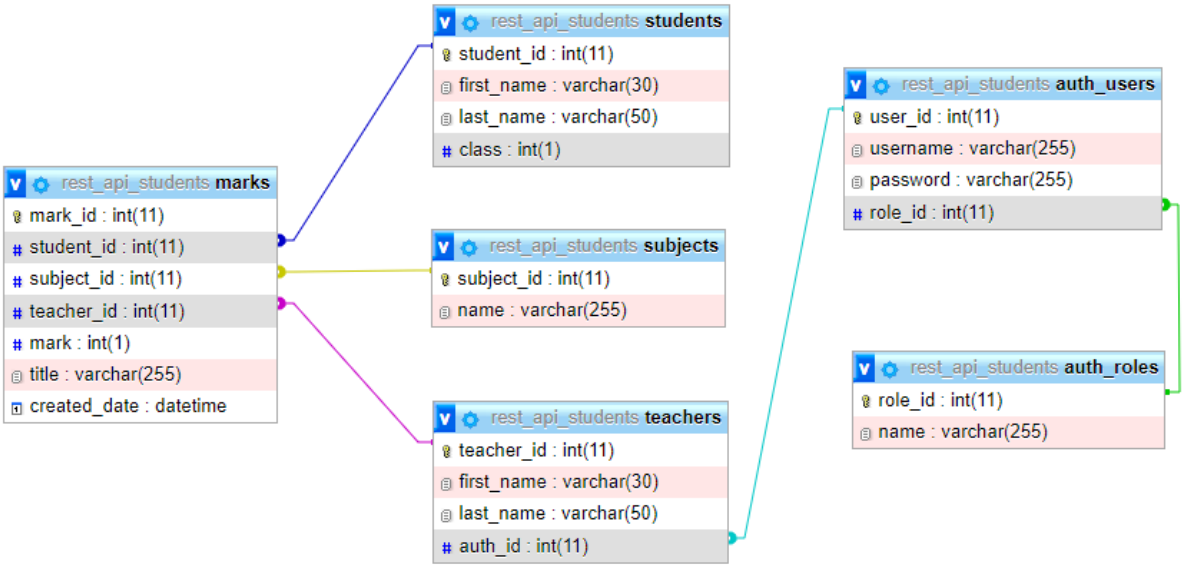
Projekt bude představovat administrační systém pro základní školy. Přístup k této webové aplikaci budou mít pouze učitelé a administrátoři. Administrátor bude moci přidávat nové studenty, upravovat jejich informace a případně je i mazat. Aplikace bude umožňovat zobrazení studentů a jejich filtrování dle tříd. Učitel bude moci každému studentovi přidat známku či si zobrazit jeho známky pro daný předmět včetně studijního průměru.

4.2 Struktura projektu

Projekt bude koncipován na dvě základní části, podle kterých jsou vytvořeny složky v kořenové složce projektu. První částí je front-end, který je reprezentován složkou „client“. V rámci této části je zpracováván vzhled a struktura stránek aplikace. Pro tvorbu této části je použit programovací jazyk JavaScript a jeho knihovna React.

Druhá část představuje back-end a bude se nacházet ve složce „server“. Zde bude využit framework, který byl vytvořen v rámci této práce. Pomocí něj jsou v konfiguračním souboru vytvořeny cesty, na které je možné zasílat požadavky z front-end části, čímž je zajištěna komunikace mezi oběma částmi projektu.

Všechna potřebná data jsou ukládána v databázi MySQL. Komunikace s databází bude probíhat výhradně prostřednictvím PHP frameworku. Databáze má definovaný přihlašovací účet s dostatečným stupněm oprávnění. Struktura databáze odpovídá obrázku níže.



Obrázek 7: Struktura databáze pro příklad využití

4.3 Instalace a implementace REST API framework

Instalaci frameworku lze provést více způsoby. Nejjednodušším způsobem je stažení skeleton aplikace, jejíž obsah stačí přesunout do složky „server“. Tento projekt již obsahuje vytvořené konfigurační soubory a soubor „index.php“. Druhým způsobem instalace je vytvoření nového projektu „composer“ a instalace frameworku příkazem:

composer require furstd/furst-d_rest-api-framework

Příklad 23: Příkaz pro stažení frameworku jako závislosti

Zde je však nejprve nutné vytvořit konfigurační soubory a soubor „index.php“. Jelikož se v rámci frameworku využívá systém cURL, je před spuštěním aplikace nutné povolit jeho spuštění přidáním hodnoty či odstraněním komentáře v konfiguračním souboru „php.ini“:

extension=curl

Příklad 24: Nastavení podpory pro systém cURL

Před samotným spuštěním serveru je nutné nastavit hodnoty pro připojení k databázi v souboru „database.yaml“. Poté stačí již samotný server spustit. To lze provést zadáním příkazu ze složky „public“ s definováním požadovaného portu, na kterém bude framework naslouchat:

php -S localhost:4000

Příklad 25: Příkaz pro spuštění API serveru

Nyní je server spuštěný a připravený pro zpracování zaslaných požadavků z klientské aplikace. Pro potřeby projektu je rovněž nutné modifikovat konfigurační soubor přepsáním či přidáním požadovaných cest a jejich funkcí.

Tabulka 5: Seznam cest v rámci příkladu využití

Cesta	Metoda REST	Potřebné oprávnění	Popis
/students	GET	ADMIN, TEACHER	Zobrazí veškeré studenty z databáze. Pro každého studenta bude zaslán jeho identifikátor, jméno, příjmení a třída.
/students	POST	ADMIN	Umožní administrátorovi přidat nového studenta.
/subjects	GET	ADMIN, TEACHER	Zobrazí seznam předmětů.
/students/{id}	GET	ADMIN, TEACHER	Zobrazí jméno, příjmení a třídu konkrétního uživatele, jehož identifikátor odpovídá parametru „id“.
/students/{id}	PUT	ADMIN	Umožní administrátorovi upravit údaje o studentovi, jehož identifikátor odpovídá parametru „id“.
/students/{id}	DELETE	ADMIN	Umožní administrátorovi smazat studenta, jehož identifikátor odpovídá parametru „id“.
/students/marks	POST	TEACHER	Slouží k zapsání nové známky. Tělo této metody obsahuje informace o známce, studentovi, předmětu a učitelovi.
/students/{id}/{subject_id}	GET	TEACHER	Zobrazí známky studenta pro daný předmět, jehož identifikátor odpovídá parametru „id“ a hledaný předmět odpovídá parametru „subject_id“.
/students/{id}/{subject_id} /average	GET	TEACHER	Zobrazí studijní průměr daného žáka v zadaném předmětu.
/validate-teacher	GET	Žádné	Zobrazí údaje o učiteli, pokud zadané uživatelské jméno a heslo odpovídá záznamu v databázi. Slouží jako kontrola přihlášení učitelů.
/validate-admin	GET	Žádné	Zobrazí údaje o administrátorovi, pokud zadané uživatelské jméno a heslo odpovídá záznamu v databázi. Slouží jako kontrola přihlášení administrátorů.

Pro každou cestu je dle popisu nastavena příslušná funkcionalita. Ve všech případech se jedná o komunikaci s databází. Níže je uvedeno nastavení cesty pro výběr známek konkrétního studenta a předmětu, jejichž parametry jsou zadány ve formě atributů v cestě. Podrobné nastavení konfiguračního souboru bylo popsáno v kapitole pojednávající o aplikační logice frameworku.

```
/students/{id}/{subject_id}:
  - GET:
      type: DB
      table: marks
      columns:
        - mark_id:
            - alias: id
        - mark
        - title
        - created_date
        - teachers.first_name:
            alias: firstName
        - teachers.last_name:
            alias: lastName
      joins:
        - students:
            type: join
            column: marks.student_id
            pk: students.student_id
        - subjects:
            type: join
            column: marks.subject_id
            pk: subjects.subject_id
        - teachers:
            type: join
            column: marks.teacher_id
            pk: teachers.teacher_id
      parameters:
        - id:
            editable: true
            type: integer
            location: path
            required: true
            operator: =
            represents: students.student_id
        - subject_id:
            editable: true
            type: integer
            location: path
            required: true
            operator: =
            represents: subjects.subject_id
      authorization:
        - TEACHER
```

Příklad 26: Nastavení výběru známek studenta pro příklad využití

4.4 Instalace a tvorba projektu ve frameworku React

Pro instalaci frameworku je nutné mít nainstalovaný balíčkovací systém „Npm“, který je standartní součástí prostředí „Node.js“. Front-end aplikaci je možné nainstalovat příkazem:

npx create-react-app client

Příklad 27: Příkaz pro vytvoření React aplikace

Příkaz nainstaluje veškeré potřebné soubory a balíčky závislostí do složky „client“. Pro potřeby aplikace je ještě nutné doinstalovat balíček „react-router-dom“, který umožní definovat stránky, ke kterým lze přistupovat. Pro spuštění aplikace je nutné ve složce „client“ spustit příkaz:

npm start

Příklad 28: Příkaz pro spuštění React aplikace

Aplikace se standartně spustí na portu 3000, pokud není obsazen. Základní bod tvorby struktury elementů představuje soubor „App.js“. Zde je definovaná komponenta „AuthManager“, která má za cíl ověřit, zda je uživatel přihlášen. Ověřování probíhá kontrolou, zda se v lokálním úložišti nachází údaje o uživateli. V případě, že se tyto informace v lokálním úložišti nenachází, je uživatel přesměrován na komponentu přihlašovacího formuláře.

```
<AuthManager>
    <Routes>
        <Route path="/" element={<Students />} />
        <Route path="/:id" element={<StudentDetail />} />
    </Routes>
</AuthManager>
```

Příklad 29: Ukázka struktury komponent souboru App.js

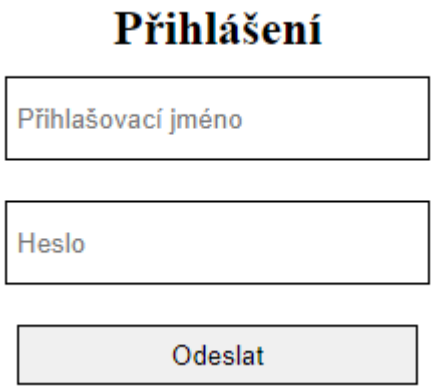
Autentizace probíhá prostřednictvím přihlašovacího jména a hesla. Kontrola zadaných údajů probíhá zasláním požadavku na adresu REST frameworku. Nejprve je požadavek odeslán na adresu „http://localhost:4000/validate-teacher“, která má dle konfiguračního souboru vrátit identifikátor, jméno a příjmení učitele s odpovídajícími přihlašovacími údaji. Pokud je však vrácen status kód „404:, jež značí, že nebyla vrácena žádná data, je odeslán druhý požadavek na stejnou adresu, pouze s jiným cílovým bodem „/validate-admin“. Zde je uživatel porovnán s údaji v tabulce „auth_users“. Oba dotazy je nutné odlišit, neboť se administrátor, na rozdíl od učitele, nachází pouze v jedné tabulce. U učitele je kromě identifikátoru vráceno ještě jméno a příjmení, aby bylo možné jednoduše zobrazit jméno učitele, který zapsal známku.

Pokud je přihlášení úspěšné, bude uživateli zobrazen obsah komponenty „Routes“ dle strany, na které se nachází. Pro účely projektu jsou zobrazovány pouze dvě strany. První strana slouží k zobrazení seznamu studentů. Druhá strana je využita při zobrazování známek jednotlivých studentů. Ostatní funkce, jako je úprava informací o studentovi či přidání známky, jsou zpracovány prostřednictvím modálního okna. Pro veškerou komunikaci s frameworkem je využita funkce „fetch“, pomocí které je možné zasílat požadavky pro veškeré metody REST a zároveň přijímat a zpracovávat odpovědi.

```
const saveMark = () => {
    fetch(`http://localhost:4000/students/marks`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': 'Basic ' +
                window.btoa(`${localStorage.getItem("username")}
                :${localStorage.getItem("password")}`)
        },
        body: JSON.stringify({
            student: id,
            subject: subjectId,
            teacher: localStorage.getItem("id"),
            mark: mark,
            title: title
        })
    })
        .then(response => {
            return response.json()
        })
        .then(json => {
            toast.success("Známka byla uložena", {
                position: toast.POSITION.BOTTOM_RIGHT
            });
            onClose();
        })
}
```

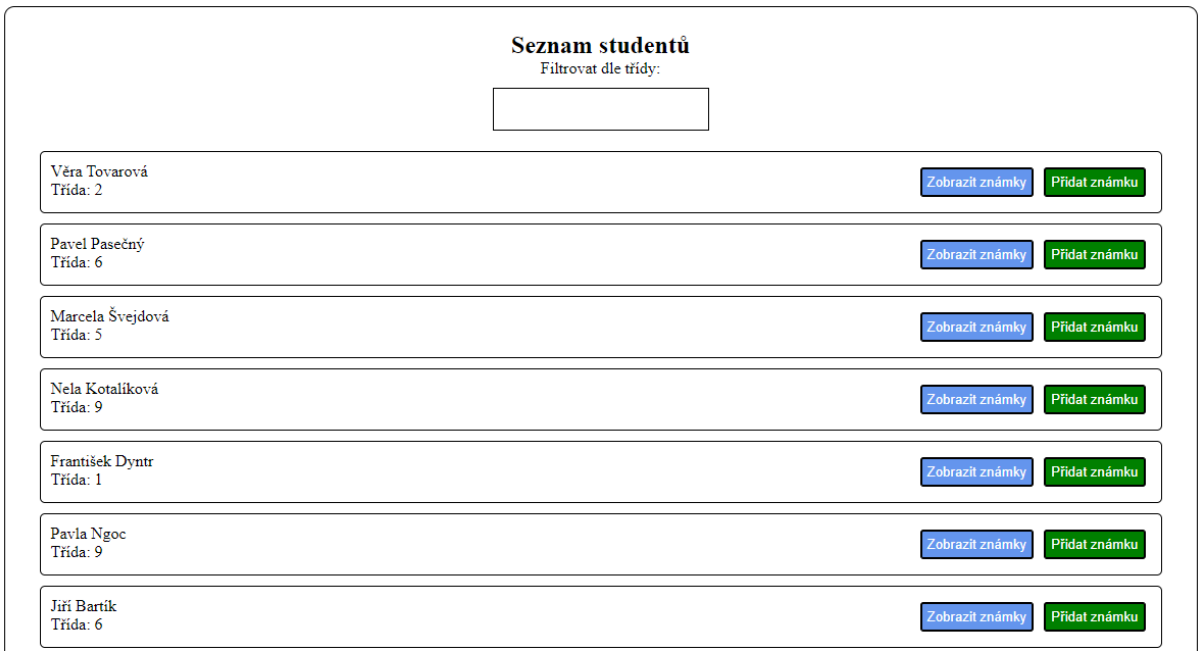
Příklad 30: Funkce saveMark() pro vrácení seznamu studentů

4.5 Výsledný produkt



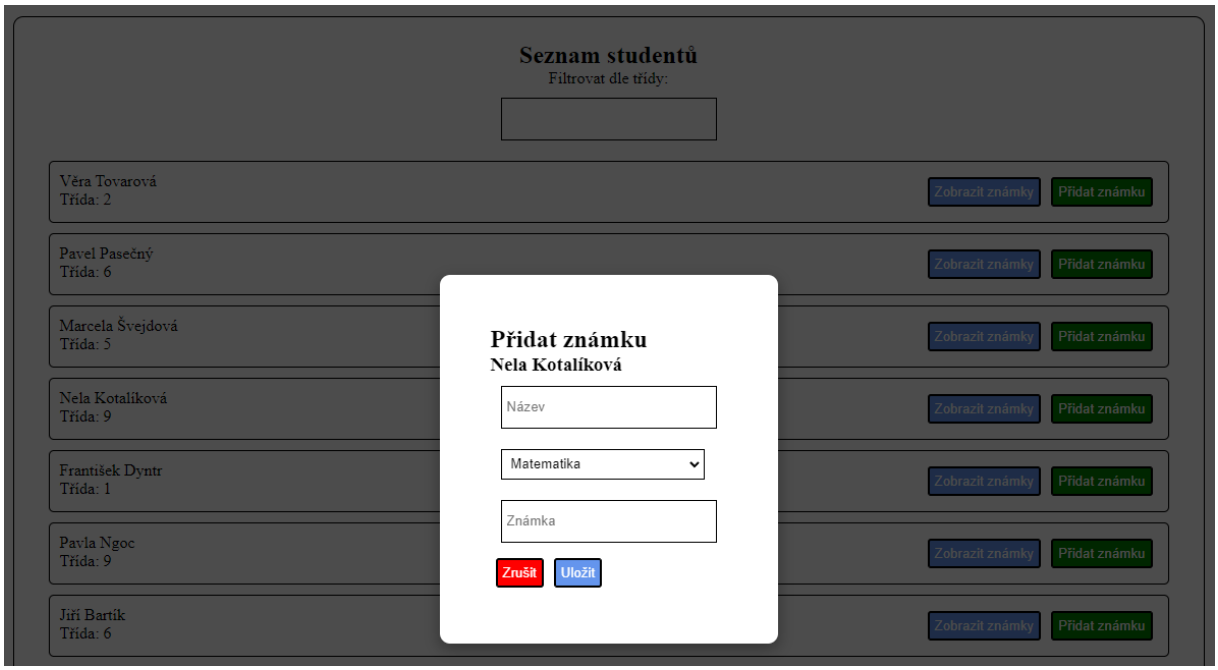
Obrázek 8: Přihlašovací formulář

Při zobrazení libovolné stránky aplikace je uživateli zobrazen přihlašovací formulář, pokud již nebyl přihlášen dříve. Po přihlášení je uživatel přesměrován na hlavní stánku aplikace, kde mu je zobrazen seznam studentů. Na straně se rovněž nachází vstupní okno, pomocí kterého je možné vyfiltrovat studenty dle jejich tříd. Dle role jsou uživateli zobrazeny příslušná tlačítka. Učitel má možnost zobrazit studentovy známky nebo přidat novou známku.



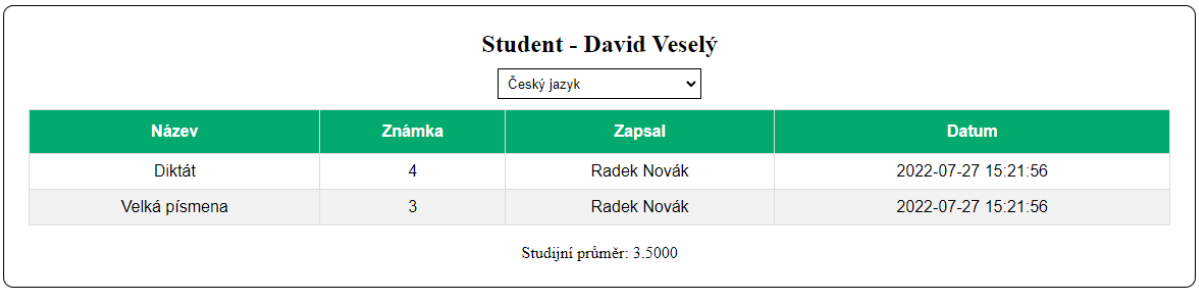
Obrázek 9: Seznam studentů z pohledu učitele

Přidání nové známky probíhá prostřednictvím tlačítka „Přidat známku“, po které se zobrazí modální okno, ve kterém musí učitel zadat název látky, samotnou známku a předmět. Pro vybírání předmětu je využit tag „select“, který je při otevření okna naplněn seznamem předmětů, který je v bezprostředním okamžiku získán prostřednictvím dotazu na API.



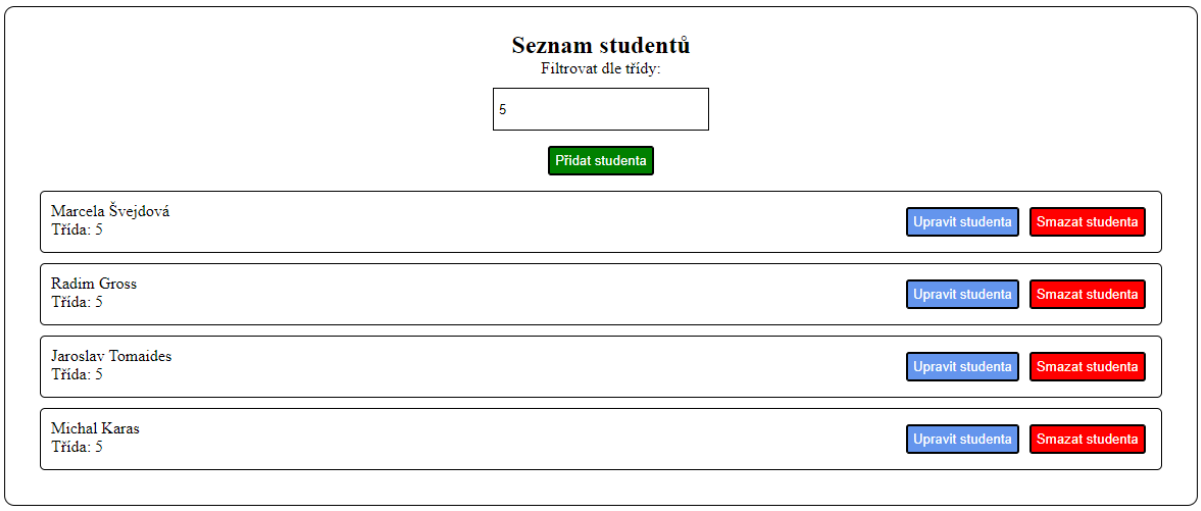
Obrázek 10: Přidávání známek studentovi

Uživatel má rovněž možnost zobrazit si známky vybraného studenta. K tomuto účelu slouží tlačítko „Zobrazit známky“. Po kliknutí na tlačítko je uživatel přesměrován na detailní stránku studenta. Ta je definovaná identifikátorem studenta v cestě. Na této stránce má učitel možnost zobrazení známek studenta dle vybraného předmětu. Zde je kromě názvu a číselné známky rovněž zobrazen učitel, který známku zapsal a datum, kdy se tak stalo. Pro zobrazení studijního průměru je odeslán samostatný požadavek.

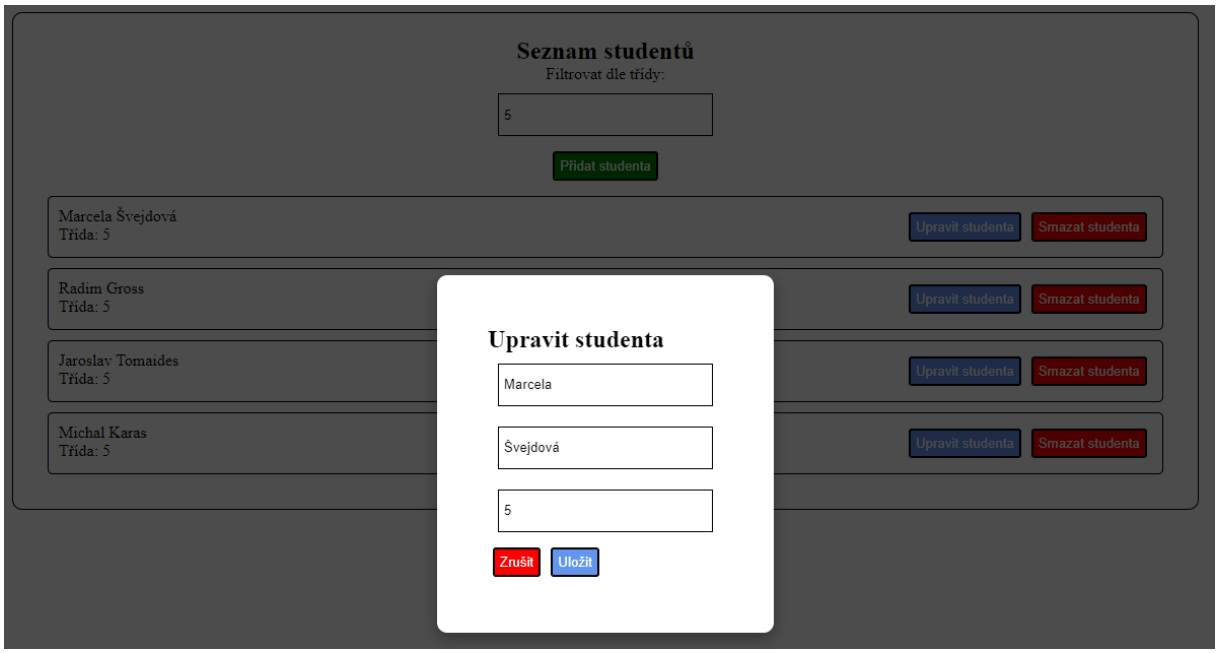


Obrázek 11: Zobrazení známek studenta

Pokud je uživatel přihlášen jako administrátor, má místo správy známek možnost upravovat studenty samotné. Upravit lze jejich jméno, příjmení či třídu, kterou navštěvují. Administrátor má rovněž možnost přidat nového studenta či ho z databáze permanentně odstranit. Pro odhlášení z aplikace lze využít tlačítko „Odhlásit se“ v pravém horním rohu stránky.



Obrázek 12: Seznam studentů z pohledu administrátora



Obrázek 13: Úprava údajů o studentovi

ZÁVĚR

Cílem bakalářské práce bylo představit problematiku REST API, definovat rozhraní REST a vyvinout knihovnu určenou pro vývojáře webových aplikací, s jejíž pomocí lze vytvářet implementovat architekturu REST.

Teoretická část představila architekturu REST. Kromě základní charakteristiky a historie byla podrobně popsána rovněž architektura rozhraní. Součástí teoretické části dále bylo zhodnocení přínosů, ale i nevýhod rozhraní REST. Posledním bodem bylo představení čtyřech různých implementací rozhraní REST určených pro programovací jazyk PHP. Zde byly kromě popisu rovněž demonstrovány praktické příklady použití daného frameworku.

V rámci praktické části a hlavním bodem bakalářské práce bylo vytvoření vlastní knihovny aplikačního rozhraní pracujícího s daty a vytvořeného na architektuře REST. Na úvod byly popsány funkční a nefunkční požadavky na framework. Stěžejní částí se stalo navržení konfiguračního souboru, který obstarával veškerou interakci mezi vývojářem a frameworkem. Struktura konfiguračního souboru byla vytvořena tak, aby umožňovala jednoduchou, avšak komplexní správu back-endové části webové aplikace, především při komunikaci s databázovým serverem MySQL. Následně byly velice detailně představeny hlavní komponenty v rámci aplikační logiky knihovny.

Na závěr byl framework aplikován na příkladu použití ve formě jednoduché webové aplikace pro správu studentů pro základní školy. Zde bylo představeno jedno z možných nastavení cest a operací REST uvnitř konfiguračního souboru. Z front-endové části aplikace bylo nastíněno, jakým způsobem lze zasílat požadavky na framework a zpracovávat přijaté odpovědi. K tomuto účelu byla využita knihovna React využívající programovací jazyk JavaScript. Výslednou aplikaci je možné rozšířit o další funkcionality.

Z důvodu snadné konfigurace bez nutnosti znalosti programovacího jazyka PHP je framework vhodný především pro začínající vývojáře webových aplikací. Výhodou je rovněž rychlejší tvorba cílových bodů, což lze uplatnit především u rozsáhlejších projektů obsahujících četné množství cest, na které lze zasílat požadavky.

Projekt by bylo možné rozšířit o možnost využití více databázových systémů, jako je například OracleDB, MSSQL nebo PostgreSQL. Z hlediska optimalizace by bylo vhodné lépe kontrolovat jednotlivé požadavky a neprovádět nepotřebné operace, čímž by se zrychlila celková doba odezvy.

POUŽITÁ LITERATURA

[1] SCHULTHESS, Coline. History of REST APIs. *Mobapi* [online]. 2017-01-26 [cit. 2022-02-09]. Dostupné z: https://mobapi.com/history-of-rest-apis/

[2] The History of REST APIs. *Readme BLOG* [online]. 2016-11-15 [cit. 2022-02-09]. Dostupné z: https://blog.readme.com/the-history-of-rest-apis/

[3] REST Architectural Constraints. *REST API Tutorial* [online]. 2021-09-27 [cit. 2022-02-11]. Dostupné z: https://restfulapi.net/rest-architectural-constraints/

[4] SURWASE, Vijay. REST API Modeling Languages -A Developer ' s Perspective. *IJSTE - International Journal of Science Technology & Engineering* [online]. 2016, **2**(10), 634-637 [cit. 2022-02-11]. ISSN 2349-784X. Dostupné z: https://www.academia.edu/27064725/REST_API_Modeling_Languages_A_Developers_Perspective

[5] HTTP Methods. *REST API Tutorial* [online]. 2021-12-11 [cit. 2022-02-11]. Dostupné z: https://restfulapi.net/http-methods/

[6] What is a RESTful API?. *MuleSoft* [online]. © 2022 [cit. 2022-02-18]. Dostupné z: https://www.mulesoft.com/resources/api/restful-api

[7] KEMAL ERINÇ, Yiğit. The Benefits of Going RESTful – What is REST and Why You Should Learn About It. *FreeCodeCamp* [online]. 2020-12-31 [cit. 2022-02-18]. Dostupné z: https://www.freecodecamp.org/news/benefits-of-rest/

[8] HAYS, Natalie. What Is a RESTful API, How It Works, Advantages, and Examples. *Mailgun* [online]. 2021-12-09 [cit. 2022-02-18]. Dostupné z: https://www.mailgun.com/blog/restful-api/

[9] Advantages and Disadvantages of REST API. *Krify* [online]. © 2022 [cit. 2022-02-19]. Dostupné z: https://krify.co/advantages-and-disadvantages-of-rest-api/

[10] WALKER, Alyssa. SOAP Vs. REST: Difference between Web API Services. Guru99 [online]. 2022-06-25 [cit. 2022-07-05]. Dostupné z: https://www.guru99.com/comparison-between-web-services.html

[11] Working with WSDLs. SoapUI [online]. © 2022 [cit. 2022-07-05]. Dostupné z: https://www.soapui.org/docs/soap-and-wsdl/working-with-wsdls/

[12] DOYLE, Kerry, Kevin FERGUSON a Cameron MCKENZIE. REST (REpresentational State Transfer). *TechTarget* [online]. 2021-01 [cit. 2022-02-19]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/REST-REpresentational-State-Transfer

[13] REST API: What is it, and what are its advantages in project development?. *BBVA API_Market* [online]. 2016-03-23 [cit. 2022-02-19]. Dostupné z: https://www.bbvaapimarket.com/en/api-world/rest-api-what-it-and-what-are-its-advantages-project-development/

[14] Introduction to GraphQL. *GraphQL* [online]. © 2022 [cit. 2022-02-20]. Dostupné z: https://graphql.org/learn/

[15] What is GraphQL?. *Red Hat* [online]. 2019-01-08 [cit. 2022-02-20]. Dostupné z: https://www.redhat.com/en/topics/api/what-is-graphql

[16] POPULI, Nous. 5 Powerful Alternatives to REST APIs (2018 Update). *LEAPGRAPH* [online]. © 2018 [cit. 2022-02-20]. Dostupné z: https://leapgraph.com/rest-api-alternatives/

[17] Falcor: One Model Everywhere. *Falcor* [online]. © 2021 [cit. 2022-02-20]. Dostupné z: https://netflix.github.io/falcor/

[18] Introduction to gRPC. *GRPC* [online]. 2021-08-11 [cit. 2022-02-20]. Dostupné z: https://grpc.io/docs/what-is-grpc/introduction/

[19] DOGLIO, Fernando. Not All Microservices Need to Be REST — 3 Alternatives to the Classic. *Bits Pieces* [online]. 2021-06-30 [cit. 2022-02-20]. Dostupné z: https://blog.bitsrc.io/not-all-microservices-need-to-be-rest-3-alternatives-to-the-classic-41cedbf1a907

[20] Top 8 RESTful API Frameworks for PHP in 2022. *Phpflow.com* [online]. 2021-12-31 [cit. 2022-02-28]. Dostupné z: https://www.phpflow.com/php/restful-api-frameworks-for-php/

[21] SAHEED GANIYU, Isola. Best PHP REST API Frameworks For 2022. *Hevo Data* [online]. 2021-10-28 [cit. 2022-02-28]. Dostupné z: https://hevodata.com/learn/php-rest-api/

[22] GUPTA, Lokesh. REST Architectural Constraints. *REST API Tutorial* [online]. 2022-03-09 [cit. 2022-07-05]. Dostupné z: https://restfulapi.net/rest-architectural-constraints/#uniform-interface

[23] Installation. *Laravel* [online]. © 2022 [cit. 2022-07-07]. Dostupné z: https://laravel.com/docs/9.x/installation#why-laravel

[24] Routing. *Laravel* [online]. © 2022 [cit. 2022-07-07]. Dostupné z: https://laravel.com/docs/9.x/routing

[25] Frontend. *Laravel* [online]. © 2022 [cit. 2022-07-07]. Dostupné z: https://laravel.com/docs/9.x/frontend

[26] DOWLING, Michael. Overview. *Guzzle 7* [online]. © 2015 [cit. 2022-07-07]. Dostupné z: https://docs.guzzlephp.org/en/stable/overview.html

[27] DOWLING, Michael. Guzzle Documentation. *Guzzle 7* [online]. © 2015 [cit. 2022-07-07]. Dostupné z: https://docs.guzzlephp.org/en/stable/index.html

[28] DOWLING, Michael. Request Options. *Guzzle 7* [online]. © 2015 [cit. 2022-07-07]. Dostupné z: https://docs.guzzlephp.org/en/stable/request-options.html

[29] MATHAI, Jaisen a Alexandr MARCHENKO, FERREIRA, Lourenzo, ed. Route. GitHub [online]. 2012-09-21 [cit. 2022-07-07]. Dostupné z: https://github.com/jmathai/epiphany/blob/master/docs/Route.markdown

PŘÍLOHY

Příloha A – REST API framework 73
Příloha B – Skeleton aplikace 74
Příloha C – Příklad aplikace frameworku 75

PŘÍLOHA A – REST API FRAMEWORK

Příloha obsahuje kompletní zdrojové kódy REST API Frameworku psaného v programovacím jazyce PHP.

PŘÍLOHA B – SKELETON APLIKACE

Příloha obsahuje zdrojové kódy projektu využívajícího REST API framework. Tento projekt obsahuje vytvořené konfigurační soubory a index soubor aplikace, díky čemuž je možné webovou aplikaci spustit bez nutnosti modifikace struktury projektu.

PŘÍLOHA C – PŘÍKLAD APLIKACE FRAMEWORKU

Příloha obsahuje zdrojové kódy webové aplikace pro správu studentů základní školy. Součástí je nejen serverová část, využívající REST API framework, ale i klientská část psaná v knihovně React.