FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Dráček IV - serverová část
Student:	Martin Kameník
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Dráček je dotyková vzdělávací aplikace pro OS Android pro žáky prvního stupně základní školy. Pro potřeby ukládání dat a návrhů cvičení používá aplikace speciální server.
1) Analyzujte stávající řešení projektu Dráček, zaměřte se především na serverovou část, jádro a komunikační API.
2) Posuzujte aktuálnost funkčních a nefunkčních požadavků z předchozích prací, doplňte je o nové získané od zadavatele.
3) Dle předchozích požadavků navrhněte metodami softwarového inženýrství novou podobu serverové části.
4) Navrhněte administrátorské GUI pro konfiguraci a monitoring serverové část.
5) Implementujte prototyp serverové části.
6) Zaveďte podporu průběžné integrace včetně průběžného testování.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 28. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Dráček IV – serverová část

Martin Kameník

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Chludil

13. května 2018

Poděkování

V prvé řadě bych chtěl poděkovat svému vedoucímu práce panu Ing. Jiří Chludilovi, a to za trpělivost a za veškeré rady, které mi ušetřily spoustu času. Dále bych chtěl poděkovat také svému otci, který mi provedl jazykovou korekturu celé práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 13. května 2018 ……

Odkaz na tuto práci

Kameník, Martin. *Dráček IV – serverová část*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práce se zabývá tvorbou nového serveru pro výukový systém Dráček. Cílem celého projektu Dráček je vytvoření výukové aplikace pro mobilní zařízení pro podporu výuky na základních školách. Pro účel ukládání dat využívá aplikace speciální server. Jelikož dřívější verze serveru z minulých let nevyhovuje současným požadavkům, je cílem této práce vytvořit nový. V práci je analyzováno předchozí řešení a za použití metod softwarového inženýrství je navržena nová podoba serverové části. Dále se práce zabývá tvorbou specifikace nového komunikačního rozhraní, definujícího způsob komunikace mezi serverovou částí a klientskými aplikacemi. Obsahem práce je také návrh grafického uživatelského rozhraní sloužícího pro správu a sledování serveru. Součástí práce je též zavedení průběžné integrace. V příloze lze nalézt zdrojové kódy prototypu vytvořené serverové aplikace.

Klíčová slova výuková aplikace, Dráček IV, serverová část systému, REST, node.js, průběžná integrace

Abstract

The subject of this thesis is the creation of a new server for the educational system Dragon. The goal of the whole project Dragon is creation of educational application for mobile devices concerning support of elementary school education. The application uses a special server for data storing. The aim of this thesis is new server creation because the former server version doesn't meet current requirements. In the thesis there is analysed former solution and a new form of server part is designed using software engineering methods. Further the thesis deals with design of a new communication interface specification which defines the form of communication among the server and clients. The thesis also describes creation of a graphical user interface serving server administration and monitoring. The part of thesis is dedicated to the implementation of the continuous integration. The source code of prototype of server application is included in the thesis attachment.

Keywords educational application, Dragon IV, server part of system, REST, node.js, continuous integration

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vzhledem ke skutečnosti, že počítačová gramotnost žáků základních škol v porovnání s minulostí roste, bylo nasnadě toho využít. Právě proto vzniká projekt Dráček, jehož cílem je vytvořit dotykovou vzdělávací aplikaci pro OS Android pro žáky prvního stupně základní školy. Cílem projektu je výuku inovovat a udělat pro žáky zábavnější. Projekt je vyvíjen již několik let a bylo k němu vypracováno již 14 bakalářských prací.

Projektu jsem se začal věnovat už v předmětu SP1 a následně jsem pokračoval i v SP2. Proto, když mi bylo nabídnuto vypracovat k projektu bakalářskou práci, nabídku jsem přijal.

Pro potřeby ukládání dat a návrhů cvičení používá aplikace speciální server, jenž byl vytvořen v rámci Dráčka II. Jelikož dřívější server nevyhovuje současným požadavkům, je potřeba vytvořit nový, což je cílem mé práce.

V rámci projektu Dráček IV jsem spolupracoval s kolegou Jaroslavem Sivákem, jehož práce se zaobírá uživatelským rozhraním jádra klientské aplikace.

Kapitola 1

Cíl práce

Cílem mé práce je přepracovat původní řešení serverové části projektu Dráček, které nevyhovuje současným požadavkům, obsahuje implementační chyby a díky nedostatečné dokumentaci není možno rozumně rozšiřovat.

Cílem rešeršní části práce je analyzovat stávající řešení. Systém se skládá z více různých částí, jež mají různé účely, proto začnu analýzou celkové architektury projektu. V té se budu zabývat jednotlivými částmi systému, a to jejich vzájemnými vztahy, účely jednotlivých částí a způsoby předávání dat. Vzhledem k tomu, že budu vytvářet novou serverovou část, budu řešit zejména, které části a jak komunikují se serverem. Dalším z hlavních výstupů této části bude zjištění, jaká data je potřeba ukládat na serveru. Budu také posuzovat aktuálnost současných funkčních a nefunkčních požadavků a budu rozebírat nové požadavky.

V rámci mé práce budu analyzovat původní komunikační rozhraní a následně navrhnu nové. Jak jsem již zmínil, obsahem mé práce je vytvořit novou serverovou část, která nahradí v systému dřívější verzi. S touto dřívější verzí již ale komunikují další části systému. Mojí snahou by tedy mělo být to, aby nebylo potřeba příliš měnit v *jádru* části aplikace zajišťující komunikaci se serverem. To samé by mělo ideálně platit pro *učitelské rozhraní*, ale zde budou přidávány nové funkčnosti (například aby učitel mohl přihlásit žáka do školního tabletu), takže jisté změny a rozšíření budou zapotřebí.

Cílem je také serverovou část rozšířit o administrátorské GUI pro konfiguraci a monitoring serverové části. Dalším cílem je zavést pro projekt průběžnou integraci včetně průběžného testování. Implementačním výstupem mé práce by měl být prototyp nové serverové části.

1.1 Rozbor zadání

V této sekci podrobněji rozepíši jednotlivé body ze zadání své práce. Uvedu, jak jsem jednotlivé cíle pochopil a jak očekávám, že je budu řešit.

1. **Analyzujte stávající řešení projektu Dráček, zaměřte se především na serverovou část, jádro a komunikační API.**

   - Nejdřív budu analyzovat projekt jako celek, kde bych měl vysvětlit architekturu systému, zejména jaké místo má v systému server a které další komponenty systému s ním komunikují. Dále pak rozeberu původní komunikační API, ve kterém budu hledat případné problémy.

2. **Posuzujte aktuálnost funkčních a nefunkčních požadavků z předchozích prací, doplňte je o nové získané od zadavatele.**

   - Staré i nové požadavky podrobně popíši a vysvětlím jejich účel.

3. **Dle předchozích požadavků navrhněte metodami softwarového inženýrství novou podobu serverové části.**

   - Provedu návrh nové podoby serverové části. V prvé řadě by výstupem měla být nová specifikace komunikačního API. Dále bych měl popsat architekturu serverové aplikace (vysvětlit jak řeším validaci, autentifikaci, chybové zpávy atd.).

4. **Navrhněte administrátorské GUI pro konfiguraci a monitoring serverové části.**

   - Pro projekt serveru Dráček IV je potřeba zajistit GUI pro monitoring, které bude dostupné online a umožní sledovat využití prostředků serveru (CPU, RAM, disk, síť). Dále pak také vyřeším GUI pro konfiguraci. U toho může být potřeba konfigurovat databázi, serverovou aplikaci a samotný serverový stroj. Pro veškeré zmíněné GUI nepředpokládám, že bych implementoval vlastní řešení. Pravděpodobně využiji nějaké již existující řešení pokrývající danou problematiku.

5. **Implementujte prototyp serverové části.**

   - Jedním z výstupů mé práce by měl být prototyp nové serverové části. Vzhledem k rozsahu projektu není jisté, do jaké míry bude serverová aplikace dokončena – měl bych odvést tolik práce, kolik bude možné, a zprovoznit aplikaci alespoň do takové úrovně, aby byla jasně zřetelná architektura systému. Práci koncipuji tak, že očekávám, že na projektu bude po mě pokračovat další osoba.

6. **Zaveďte podporu průběžné integrace včetně průběžného testování.**

    - Pro serverovou aplikaci zavedu průběžnou integraci. V práci vysvětlím, co průběžná integrace obnáší a jaký je její účel. Budu se také zabývat průběžným testováním a tím, jak souvisí s průběžnou integrací.

Kapitola 2

Analýza

Na začátku této kapitoly se budu velmi stručně věnovat architektuře systému jako celku, a to z důvodu, že je to pro pochopení, jak systém funguje, zcela nezbytné. Dále pak analyzuji existující požadavky na serverovou část a rozebírám nové. Pro správné pochopení funkčnosti systému využiji doménového modelu a několika vybraných procesních modelů. Také budu rozebírat současný stav komunikačního API. Na konci této kapitoly se pak věnuji výběru vhodných technologií pro novou verzi serverové části.

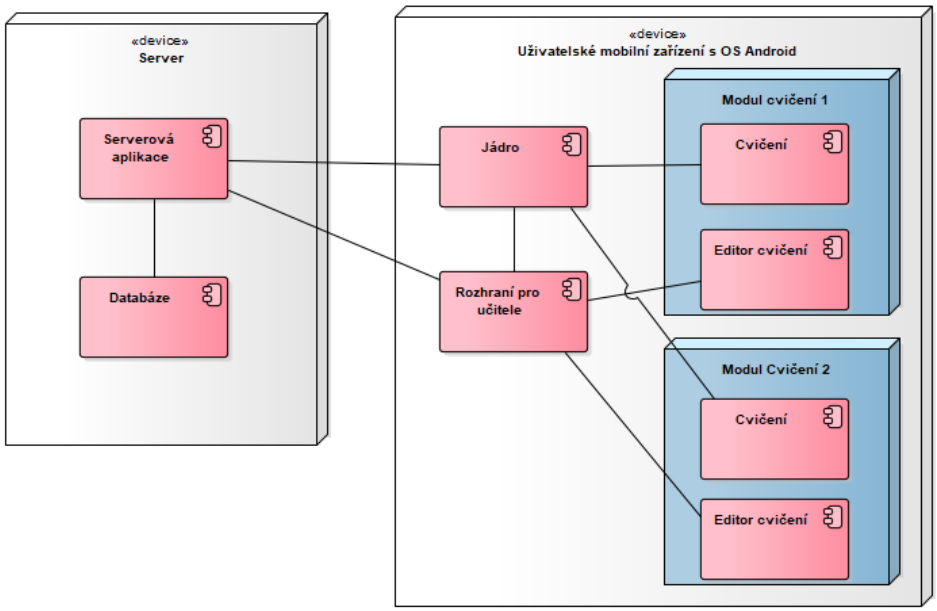
2.1 Analýza stávajícího řešení projektu

Vzhledem k tomu, že Dráček je rozsáhlý projekt skládající se z vícero poměrně samostatných částí, jsem se rozhodl nejdříve vysvětlit architekturu systému jako celku. Zde také uvedu termíny používané pro jednotlivé části systému, které budu dále užívat v celé své práci. Nejdůležitější bude to, jakou roli a místo v celém projektu má serverová část. V druhé časti této sekce velmi stručně rozeberu předchozí verze (iterace) projektu.

2.1.1 Architektura systému

Systém Dráček byl už od počátku navržen tak, aby byl modulární a tudíž aby umožňoval více osobám pracovat společně na stejném projektu. Pro názorné vysvětlení architektury systému jsem se rozhodl vytvořit diagram, který můžeme vidět na obrázku 2.1. V modelu jsem ukázal architekturu systému pro dva moduly cvičení, ovšem ve skutečném případě jich bude mnohem více a dva uvádím jen pro jednoduchost a názornost.

Nyní popíši účel jednotlivých částí systému a jejich vztahy mezi sebou. Budu se zaměřovat především na jejich vztah k serverové části.



Obrázek 2.1: Architektonický model pro dva moduly cvičení

- **Serverová aplikace**
  - Poskytuje komunikační rozhraní, které využívá *jádro* a *rozhraní pro učitele*.
  - Provádí validace veškerých přijatých a odesílaných dat.
  - Svá data dále ukládá do databáze.
- **Databáze**
  - Nachází se na serveru.
  - Zajišťuje perzistentní uložení veškerých dat.
- **Jádro**
  - Jádro je základním prvkem systému.
  - Komunikuje se serverem.
  - Zajišťuje přihlašování všech uživatelů (žáků i učitelů).
  - Pro žáky zajišťuje synchronizaci jejich dosažených výsledků se serverem. Dále také pro žáky stahuje ze serveru jednotlivá zadání do cvičení.
  - Poskytuje GUI, které umožňuje přihlašování uživatelů a spouštění jednotlivých modulů cvičení.

- **Rozhraní (modul) pro učitele**
  - Také komunikuje se serverem.
  - Může ho používat pouze uživatel s rolí učitel.
  - Umožňuje prohlížet dosažené výsledky všech žáků.
  - Umožňuje administraci žákovských účtů (vytvářet, upravovat i smazat účty).
  - Z rozhraní pro učitele lze spouštět editory cvičení.
- **Modul cvičení**
  - Je většinou zaměřen na určitou probíranou látku nebo předmět.
  - Skládá se z *cvičení* a z *editoru cvičení*, jednotlivě popsáno níže.
- **Cvičení**
  - Cvičení vyplňují žáci.
  - Výsledky předává *jádru*, které je následně odesílá na server.
- **Editor cvičení**
  - Umožňuje učiteli vytvářet do jednotlivých cvičení nová zadání a upravovat stávající.

2.1.2 Předchozí verze (iterace) projektu

Projekt Dráček je vyvíjen již několik let a verze vyvíjená v současnosti nese již číslo IV. K projektu už bylo vypracováno 14 bakalářských prací. Pro potřeby pochopení projektu a toho, jak byl vyvíjen, jsem se rozhodl zde stručně shrnout, co a jak bylo řešeno v jednotlivých iteracích (verzích) projektu Dráček a jak se toto dotýká mé práce. Zmíním také vybrané předchozí bakalářské práce, z nichž jsem při vytváření Dráčka IV vycházel.

Dráček I

- Dráček I byl implementován jako desktopová aplikace se zaměřením na výuku primárně dětí se specifickými poruchami učení.
- K verzi Dráček I bylo vypracováno 5 bakalářských prací.
- Z této verze nebylo přejato do současné verze nic a ve své práci se jí již vůbec nezabývám.

Dráček II

- Ve verzi Dráček II bylo uděláno zásadní rozhodnutí o zaměření projektu, a to nově na mobilní zařízení s operačním systém Android.

- Z této verze pochází původní *jádro*, *modul s rozhraním pro učitele* a server, které jsou upravovány či znova vytvářeny v současné verzi Dráček IV.
- Pro verzi Dráček II vzniklo 5 bakalářských prací. Pro moji práci je nejdůležitější předchozí práce [1] zabývající se serverovou částí systému. Také využívám poznatků z práce [2] zabývající se problematikou gamifikace a personalizace.

Dráček III

- Dráček III se zabýval pouze vytvářením zásuvných modulů.
- Cílem bylo vyvinout již prakticky použitelné moduly, které budou vhodně demonstrovat a využívat možnosti celého systému Dráček.
- Nová verze Dráček IV musí zůstat kompatibilní s těmito moduly.
- Vznikly 3 bakalářské práce: [3], [4] a [5].

K projektu Dráček vznikla ještě jedna práce, která přímo nesouvisí s žádnou verzí. Je to bakalářská práce [6] zabývající se tvorbou animovaného avatara.

2.2 Analýza existujících a nových požadavků

Vzhledem k tomu, že navazuji na předcházející práce, začnu analýzou existujících požadavků. V rámci toho budu posuzovat jejich aktuálnost pro projekt Dráček IV. Dále pak rozeberu nové požadavky na serverovou část.

2.2.1 Existující požadavky na serverovou část

Při analýze existujících požadavků na serverovou část vycházím z požadavků ze starších prací, zejména pak z bakalářské práce [1], na niž navazuji.

Funkční požadavky

- **FP0.1 - Správa dosažených výsledků**
  - Na server žáci odesílají své dosažené výsledky.
  - Žák může ze serveru získat pouze své výsledky.
  - Učitel může získat výsledky všech žáků.
- **FP0.2 - Správa dostupných modulů cvičení**
  - Na serveru jsou evidované jednotlivé moduly cvičení, které pak obsahují jednotlivá zadání cvičení.

  - Učitel může přiřadit jednotlivým žákům přístup pro jednotlivé moduly (možnost stáhnout si a nainstalovat do svého zařízení a následně spouštět z *jádra*).

- **FP0.3 - Správa zadání do existujících cvičení**

  - Učitelé mohou vytvářet nová zadání do jednotlivých cvičení. Ty pak mohou nahrát na server, odkud jsou staženy do žákovských zařízení.

- **FP0.4 - Správa tříd**

  - Studenti mohou být seskupováni do tříd. Server toto musí evidovat.
  - K tomuto seskupování má přístup jen učitel. Nejedná se sice o nijak citlivá data, ale nenašel jsem důvod, proč by žák potřeboval vědět, kdo je v jaké třídě.

- **FP0.5 - Správa uživatelů**

  - V systému existují dva různé druhy uživatelů: *učitel* a *žák*.
  - Učitel může vytvářet účty žákům.
  - Učitel může smazat účet žáka, ale z důvodu, aby se zabránilo nechtěnému smazání, je v takovém případě po učiteli vyžadováno znovu zadat své přihlašovací údaje.
  - Učitelské účty vytváří administrátor.

- **FP0.6 - Správa úkolů**

  - Učitel může zadat žákům úkol k vypracování s datem, do kdy ho mají vypracovat, a minimální úspěšnost, které žák musí dosáhnout.

- **FP0.7 - Podpora offline režimu klienta**

  - Žáci mohou pracovat i bez připojení k internetu. V takovém případě jsou data synchronizována ve chvíli, kdy je připojení k internetu dostupné. Server toto musí brát v potaz a musí být schopen přijímat i data nastřádaná během offline období.

Všech sedm výše zmíněných funkčních požadavků zůstává aktuální i v nové verzi. Důvodem pro to je skutečnost, že se týkají základní architektury projektu, která se v nové verzi projektu nemění (zůstává stejná od Dráčka II).

Nefunkční požadavky

- **NP0.1 - Podpora více uživatelů najednou**
  - Na server bude přicházet více požadavků zároveň, a to jak od jednoho uživatele, tak od více různých uživatelů. Server musí být schopný obsluhovat tyto požadavky paralelně.
  - Vzhledem k tomu, že nová verze serverové části by měla být v ideálním případě použita reálně pro větší počet uživatelů, stává se tento požadavek ještě důležitějším než dříve.
- **NP0.2 - Pouze učitel může zobrazit výsledky jiných uživatelů (žáků)**
  - Žáci by neměli vidět výsledky jiných žáků, a to jak z důvodu, že se jedná o potenciálně citlivé údaje, tak i protože by to mohlo mít nežádoucí účinky na jejich snažení. (Např. kdyby žák viděl, že je v řešení úloh napřed, mohl by lenivět a přestat se tolik snažit. Nebo naopak kdyby viděl, že je pozadu, mohl by být frustrován.)
  - Toto bylo rozepsáno už u jednotlivých funkčních požadavků výše.
  - Je stále aktuální v nové verzi.
- **NP0.3 - Poskytuje REST rozhraní pro komunikaci**
  - Tento způsob komunikace byl vybrán již v předchozích pracích. *Jádro* aplikace a učitelské rozhraní ho již používají, proto musí být tento způsob komunikace zachován. Navíc není ani důvod, proč by měl být použit jiný způsob.
- **NP0.4 - Zdarma dostupné technologie**
  - Je žádoucí, aby pro užívání systému nebylo potřeba dokupovat další licence.
  - Zůstává aktuální už od Dráčka I.

2.2.2 Nové požadavky na serverovou část

Od doby Dráčka II přibyly následující nové požadavky:

Funkční požadavky

- **FP1.1 - Učitel může přihlásit libovolného žáka**
  - Vzhledem k problematičnosti přihlašování u ještě negramotných žáků bylo v Dráčku IV rozhodnuto, že bude umožněno učiteli přihlásit libovolného žáka. Toto má smysl zejména u školních tabletů,

které jsou uloženy ve třídě a jsou rozdávány žákům na jednotlivé vyučovací hodiny. V takovém případě se učitel na žákovském tabletu přihlásí svými údaji a v učitelském rozhraní vybere konkrétního žáka a vybere volbu k přehlášení na tohoto žáka. Tím je sám odhlášen a následně tablet může předat danému žákovi, který je už přihlášen na svůj vlastní účet.

  - Pro server toto znamená, že musí podporovat kromě standardního způsobu přihlášení žáka za použití jeho přihlašovacích údajů také alternativní způsob přihlášení žáka na žádost již přihlášeného učitele.

- **FP1.2 - GUI pro konfiguraci a monitoring serverové části**

  - Pro nový server by mělo vzniknout GUI umožňující sledovat stav serveru. Mezi sledované veličiny by mělo patřit využití CPU, RAM, disku a sítě.
  - Také je vhodné, aby existovalo GUI pro konfiguraci serveru.

Nefunkční požadavky

- **NP1.1 - Podpora průběžné integrace včetně průběžného testování**

  - Bylo rozhodnuto, že pro usnadnění vývoje bude zavedena průběžná integrace, jejíž součástí by mělo být co nejvíce testů, které budou následně automaticky spouštěny při každém sestavení. Díky tomuto by měla být zajištěna kontrola, že případné změny „nerozbijí“ jiné již existující a správně pracující funkcionality systému.

- **NP1.2 - Verzovatelnost, více verzí paralelně**

  - Musí být možné snadno vytvořit novou verzi systému.
  - Jelikož někteří uživatelé mohou mít starší verze programu, tak je vyžadováno, aby mohlo být **v provozu více verzí v jeden okamžik**. Navíc v případě, že systém bude používán v ostrém provozu, je vhodné, aby nové verze byly před uvolněním pro všechny uživatele testovány menší skupinou vybraných uživatelů.

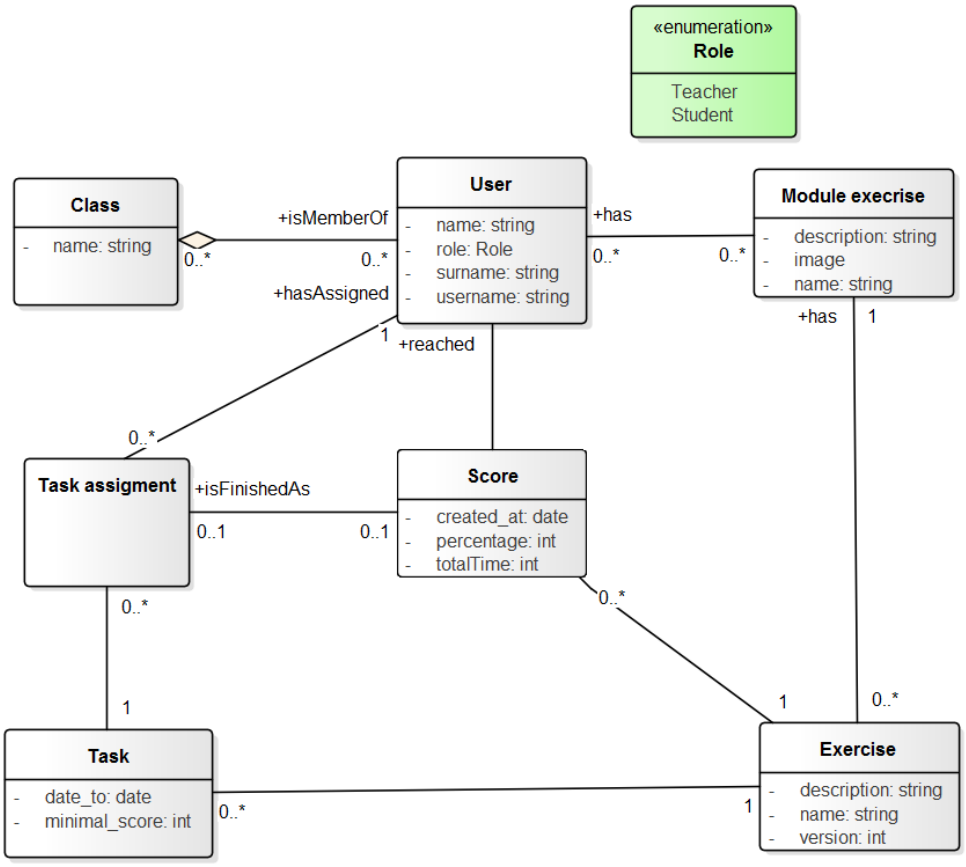
- **NP1.3 - Kvalitní dokumentace**

  - Jedním z hlavních problémů předchozího řešení serverové části projektu Dráček byla nedostatečná dokumentace, proto v nové verzi bude kladen důraz na její kvalitu a praktickou použitelnost. Dokumentace musí umožnit, aby na projektu mohla v budoucnu pokračovat jiná osoba.
  - Mělo by se to týkat hlavně specifikace komunikačního API.

- **NP1.4 - Rozšiřitelnost**
  - Dá se očekávat, že se v budoucnu objeví nové požadavky na funkcionality serverové části. Systém proto musí být snadno rozšiřitelný.
  - Základním předpokladem pro toto je **NP1.2 - Verzovatelnost** a **NP1.3 - Kvalitní dokumentace**.

2.3 Doménový model

V rámci předchozí bakalářské práce [1] byl vypracován doménový model. Tento model jsem shledal stále aktuálním, proto nebudu vypracovávat vlastní. Model je dle mého názoru vypracován zcela správně. Model přejatý z [1] můžete vidět na obrázku 2.2.



Obrázek 2.2: Doménový model

Myslím si, že tento model je dostatečně názorný a není potřeba ho detailně

rozebírat. Chtěl bych jen upozornit na následující skutečnosti, které by někomu nemusely být jasné:

- *Exercise* v modelu vyjadřuje zadání do modulu cvičení.
- *Task* vyjadřuje domácí úkol, který může učitel zadávat studentům. Úkol má přiřazené konkrétní zadání a má specifikováno datum, do kdy musí být dokončen s definovaným minimálním dosaženým výsledkem.
- *Class* vyjadřuje třídu, kterou žák navštěvuje (např. „4.B“).
- *Score* je výsledek (hodnocení) studenta. Může vzniknout buď samostatně, nebo jako součást domácího úkolu.

Podrobný popis lze dohledat v [1], ale pro zjednodušení jsem vybrané stránky týkající se doménového modelu přidal do přílohy mé práce.

2.4 Model serverových procesů

Podobně jako u doménového modelu i zde jsem se rozhodl použít modely z předchozí bakalářské práce [1]. Analogicky jsem podrobný popis procesních modelů z [1] přidal do přílohy. V diagramech jsem nalezl a opravil několik drobných chyb. Do této práce jsem umístil pouze jeden procesní model, a to 2.3 – Vytvoření zadání. Ostatní opravené diagramy jsou také samostatně v příloze. Chtěl bych upozornit, že ve vybrané části z [1] umístěné v příloze jsou diagramy v původním neopraveném stavu.

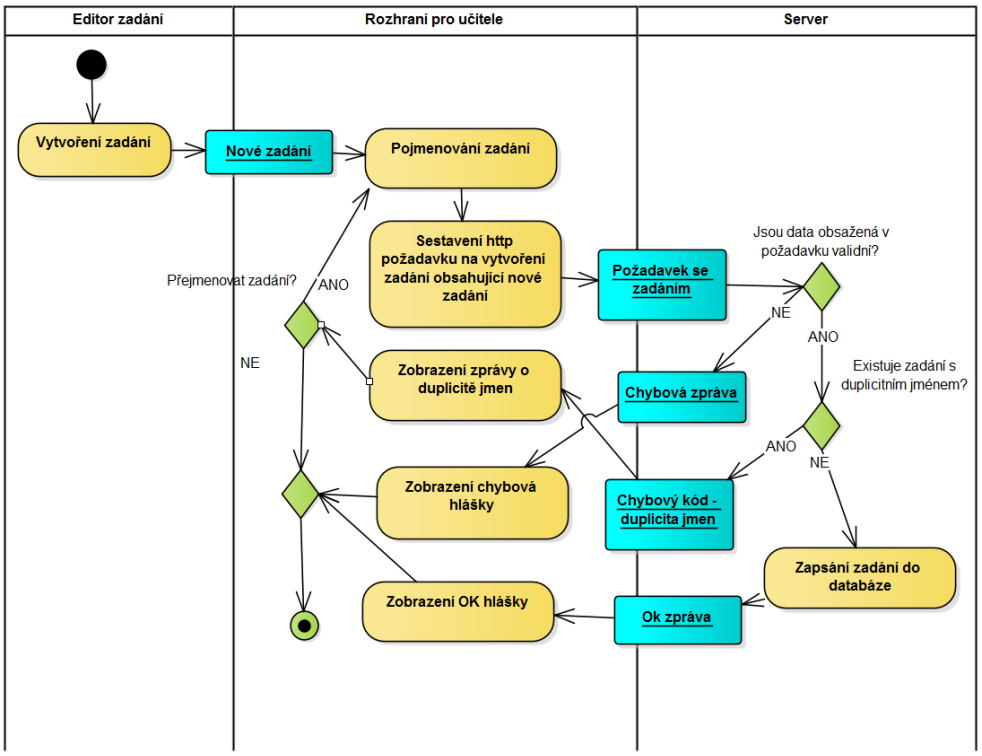
2.5 Webová služba pro komunikaci – REST

V předchozí práci byl vybrán způsob komunikace za požití REST rozhraní. S tímto rozhodnutím souhlasím a nebudu ho měnit. Případná změna by navíc vyžadovala další úpravy v *jádru* a v *rozhraní pro učitele*, tudíž je silně nežádoucí.

Nyní stručně vysvětlím, co to REST vlastně je. Při psaní tohoto shrnutí jsem čerpal ze zdrojů: [7] a [8].

REST *(Representational State Transfer)* je architektura rozhraní umožňující přístup k datům na serveru. REST definuje čtyři základní operace (metody) pro přístup k datům. Jedná se o CRUD operace. Nyní operace vyjmenuji a stručně popíši.

- **GET (Retrieve / Read)**
  - Slouží pro získání dat ze serveru. Často jsou také v rámci požadavku posílány další údaje sloužící pro filtrování vracených dat.



Obrázek 2.3: Diagram aktivit - Vytvoření zadání

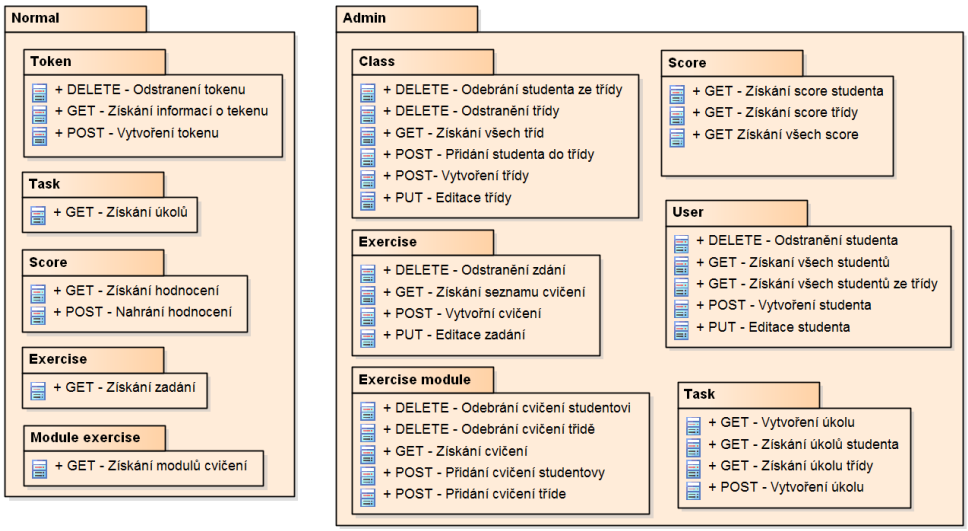
- **POST (Create)**
  - Operace POST slouží pro vytvoření dat. V požadavku by měly být obsaženy údaje potřebné pro vytváření objektu. Někdy lze operaci použít i jinak než pro vytváření objektů, a to pro zadání příkazu pro serverovou aplikaci.
- **PUT (Update)**
  - Standardně slouží pro úpravu již existujících dat.
- **DELETE**
  - Umožňuje odstranění již existujících dat.

Operace jsou volány nad zdroji. Zdroj je identifikován pomocí URI (např. „/dracek/v1/task“). Údaje mohou být od klienta na server předávány více způsoby, mezi nejběžnější a v mé práci používané způsoby patří:

- v rámci cesty (path) – např. /dracek/v1/task/{taskId}
- jako parametr (query) – např. /dracek/v1/task?taskId=8d15ac15d67d5
- v hlavičce požadavku (header)
- v těle požadavku (body)

2.6 Analýza současného stavu komunikačního API

V rámci předchozí bakalářské práce [1] bylo vytvořeno jisté komunikační rozhraní. Toto rozhraní již využívá současná verze *jádra* a *modulu pro učitele*, proto je potřeba co nejlépe zachovat současný formát tohoto API. Ke komunikačnímu API byla v rámci zmíněné bakalářské práce vypracována i specifikace. Vzhledem k tomu, že nepřejímám z předchozí serverové části žádný software, bude tato specifikace to, z čeho budu vycházet při vytváření nového komunikačního API. K tomuto rozhraní si z předchozí bakalářské práce [1] vypůjčím schéma, které můžete vidět na obrázku 2.4. Materiály týkající se komunikačního API jsou bohužel problematické a navzájem se neshodují. Schéma 2.4, které zde uvádím, pochází z textu bakalářské práce [1]. Chtěl bych upozornit na skutečnost, kterou jsem zjistil během analýzy, a to, že toto schéma se neshoduje s tím, co je uvedeno ve specifikaci tohoto rozhraní, která se nachází v příloze zmíněné práce. Tyto nesrovnalosti se týkají jak pojmenování jednotlivých endpointů, tak i jejich počtu a účelu.



Obrázek 2.4: Schéma komunikačního API

Dále jsem pak analyzoval současný stav specifikace komunikačního API a odhalil jsem následující nedostatky:

- **Chybějící identifikátory**
  - Není jasné, jak mají být chybějící identifikátory předávány. Pravděpodobně by měly být součástí cesty, jak je tomu u jiných podobných případů uvedených ve specifikaci. Ovšem vyskytují se i případy, kdy je identifikátor předáván jinak, například v hlavičce metody.
- **Nejasný účel některých metod**
  - Účel některých metod není v podobě, v jaké jsou uváděny ve specifikaci, zcela jasný. Je možné, že buď chybějí nějaké předávané údaje, nebo byly tyto metody vytvořeny dříve a nějaké změna způsobila, že jsou v tuto chvíli nesmyslné.
- **Nejednotnost označení předávaných údajů**
  - V práci jsem narazil na různě pojmenované parametry, které ale u konkrétní skupiny metod mají stejný význam, tudíž by bylo vhodné je pojmenovat stejně.
- **Kompletní absence odpovědí pro chybové případy**
  - Ve specifikaci zcela chybí informace, jak má server odpovídat v chybových případech, jako například když neexistuje žádný žák s daným id apod.
  - Toto je dle mého názoru nejvýznamnější problém současné specifikace komunikačního API.

Chtěl bych zmínit, že tento výčet problémů neobsahuje veškeré problémy a při návrhu nové verze se dá očekávat nalezení dalších problémů.

V rámci mé práce bude také potřeba rozšířit komunikační API, a to zejména o části umožňující učiteli přihlásit libovolného žáka, jak uvádím ve funkčním požadavku FP1.1. Také se dá očekávat, že se během vývoje objeví nové požadavky na to, jaká data server bude uchovávat. Potřeba ukládat nějaké další údaje se pak samozřejmě promítne i do komunikačního API.

Jednotlivým částem komunikačního API se budu později podrobně věnovat v rámci návrhu, kde budu vytvářet novou verzi API, která by měla odstranit výše zmíněné nedostatky.

2.7 Analýza použitých technologií

Je potřeba vybrat, jaké technologie budou použity v nové verzi serverové části. Při výběru jsem bral ohled zejména na jednoduchost použití a na skutečnost, že v mé práci bude později pravděpodobně pokračovat někdo jiný.

2.7.1 Formát přenášených dat

Verze Dráček II využívala pro přenos dat formát **JSON**. Podobně jako u bodu "Webová služba pro komunikaci" i zde musí být tento způsob přenosu dat v nové verzi zachován, protože jinak by byly nutné rozsáhlé změny v *jádře* a v *rozhraní pro učitele*, což je silně nežádoucí.

2.7.2 Serverová aplikace

Úkolem serverové aplikace je zprostředkovávat přístup do databáze a poskytovat REST rozhraní pro komunikaci. Vzhledem ke skutečnosti, že ze serverové části předchozí verze Dráček II nepřejímám implementační stránku serverové aplikace, není nutné použít stejné technologie. Uvažoval jsem o následujících třech možných technologiích:

- **PHP + Nette**
  - Bylo použito v serverové části předchozí verze Dráček II.
  - Nemám s touto technologií téměř žádné zkušenosti.
- **Java EE**
  - Vhodné pro vývoj rozsáhlých podnikových aplikací. V našem případě zbytečně složité.
  - S touto technologií mám základní zkušenosti z předmětu BI-TJV. K technologii mám jisté výhrady, ale neměl bych problém ji použít.
- **Node.js**
  - Použití Node.js mi bylo doporučeno vedoucím práce.
  - Vedoucí práce mi doporučil použít jako ukázkový prototyp jinou serverovou aplikaci, která vznikla v rámci diplomové práce [9].
  - Jelikož Node.js je v současnosti široce rozšířenou technologií, je pravděpodobnější, že můj případný následovník bude s touto technologií umět.
  - S technologií jsem měl před touto prací jen minimální zkušenosti.
  - Nativně využívá formát JSON, což je, vzhledem k přenášení dat ve formátu JSON v rámci projektu Dráček, velmi užitečné.

Po zhodnocení výhod a nevýhod jednotlivých technologií **jsem si vybral Node.js**.

Chtěl bych ještě podotknout, že existuje i celá řada dalších možností, ale zvolená technologie danému účelu plně vyhovuje, takže nemám potřebu hledat další možnosti.

2.7.3 Databáze

V předchozí verzi byla použita databázová technologie MySQL. Nejsem si ovšem jistý, zda tato databáze je vhodná i pro novou verzi, která, jak jsem již rozhodl v 2.7.2, bude psaná v Node.js. Proto provedu podrobnější analýzu dostupných technologií.

Hodnotící kritéria

Při posuzování vhodnosti jednotlivých databází se zaměřím především na následující vlastnosti:

- vhodnost pro ukládání dat přímo ve formátu JSON
- kompatibilita s Node.js
- dostupnost zdarma

Jako další kritérium jsem původně chtěl zvolit i podporu administrativních nástrojů, ovšem už po letmé analýze dostupných nástrojů jsem došel k názoru, že jich je obrovské množství a jejich analýza by byla velmi časově náročná. Navíc vzhledem k tomu, že existuje i celá řada nástrojů nezávislá na konkrétní databázové technologii, by toto kritérium stejně nehrálo pravděpodobně velkou roli při výběru konkrétní databázové technologii.

Pro hodnocení *vhodnosti ukládání dat přímo ve formátu JSON* jsem čerpal zejména ze zdrojů [10] a [11].

Způsob hodnocení

Způsob hodnocení jednotlivých kritérií jsem zvolil následovně:

- vhodnost pro ukládání dat přímo ve formátu JSON – Budu hodnotit vždy jedním z následujících stupňů: NEDOSTATEČNÉ, DOSTATEČNÉ, DOBRÉ, VÝBORNÉ.
- kompatibilita s Node.js a dostupnost zdarma – Budu hodnotit pouze SPLŇUJE nebo NESPLŇUJE. Hodnocení NESPLŇUJE bude automaticky znamenat nevhodnost dané technologie pro náš účel.

Volba kandidátů

Databázových technologií existuje velké množství, a proto budu vybírat jen z několika kandidátů. Tyto kandidáty jsem zvolil zejména kvůli jejich všeobecné známosti a skutečnosti, že na první pohled vypadají, že dokáží pokrýt problematiku této práce. Nakonec jsem jako kandidáty vybral: MySQL, MongoDB, PostrgeSQL, SQLite. Všichni tito kandidáti jsou běžně používané technologie, tudíž by neměl být pro nikoho problém se s těmito technologiemi snadno naučit, pokud bude někdo v mé práci pokračovat.

MySQL

Jedná se o relační databázi. MySQL bylo použito v serverové části předchozí verze.

- **Vhodnost pro ukládání dat přímo ve formátu JSON** – Poskytuje speciální datový typ pro ukládání JSON dokumentů. Umožňuje automatickou validaci, zda string odpovídá JSON formátu. Poskytuje celou řadu speciálních funkcí pro práci s formátem JSON. Hodnotím jako **DOBRÉ**.

- **kompatibilita s Node.js** – Pro Node.js existuje modul *„mysql“*, který umožňuje jednoduše komunikovat s MySQL databází. Hodnotím **SPLŇUJE**.

- **dostupnost zdarma** – MySQL je nabízeno ve dvou edicích: zdarma – *„Community Server“* a placené – *„Enterprise Server“*. Pro náš účel by měla bohatě stačit zdarma dostupná edice. Hodnotím **SPLŇUJE**.

SQLite

Jde se o relační databázi. Na rozdíl od většiny ostatních databází se nejedná o samostatný program, ale jde pouze o knihovnu, která se přidává k aplikaci.

- **Vhodnost pro ukládání dat přímo ve formátu JSON** – Pro SQLite existuje speciální rozšíření s názvem *„JSON1“* [12]. Toto rozšíření umožňuje manipulaci s JSON dokumenty. JSON je ukládán jako obyčejný text. Neobsahuje speciální datový typ pro JSON. Bez tohoto rozšíření je SQLite pro ukládání ve formátu JSON zcela nevhodné. Hodnotím jako **DOSTATEČNÉ**.

- **kompatibilita s Node.js** – Podobně jako u MySQL i zde existuje pro Node.js modul pro práci s SQLite databází. Jedná se o modul *„sqlite3“*. Hodnotím **SPLŇUJE**.

- **dostupnost zdarma** – Jedná se o zdarma dostupnou technologii šířenou pod licencí *public domain*. Hodnotím **SPLŇUJE**.

MongoDB

Jedná se o dokumentově orientovanou databázi.

- **Vhodnost pro ukládání dat přímo ve formátu JSON** – Všechna data jsou v MongoDB nativně ukládána ve formátu JSON, přesněji v jeho binární variantě BSON. MongoDB je přímo určeno k ukládání dat ve formátu JSON. Hodnotím jako **VÝBORNÉ**.

- **kompatibilita s Node.js** – Pro komunikaci s MongoDB se používá Node.js modul *„mongodb“*. Vzhledem k nativní podpoře formátu JSON v MongoDB je komunikace s databází z Node.js aplikace velmi jednoduchá. Hodnotím **SPLŇUJE**.

- **dostupnost zdarma** – Pro MongoDB jsou nabízeny jak zdarma, tak placené edice. Co se týče zdarma dostupného, lze použít takzvaný *„Community Server“*, který je vhodný pro náš účel. Dále také lze omezeně zdarma využít službu *„Atlas“*, kde je databáze uložena na jejich speciálním cloudu (na jejich serveru dostupném přes internet), což ale pro tento projekt není příliš vhodné. Hodnotím **SPLŇUJE**.

PostrgeSQL

Jedná se o objektově-relační databázi.

- **Vhodnost pro ukládání dat přímo ve formátu JSON** – V PostrgeSQL existuje speciální datový typ pro ukládání dat ve formátu JSON. Dále také implementuje operátory pro práci s formátem JSON. Hodnotím jako **DOBRÉ**.

- **kompatibilita s Node.js** – Jako ve všech předchozích případech i pro PostrgeSQL existuje Node.js modul. Tento modul najdeme pod krátkým jménem *„pg“*. Hodnotím **SPLŇUJE**.

- **dostupnost zdarma** – PostrgeSQL je dostupné zcela zdarma. Jedná se o open-source technologii. Hodnotím **SPLŇUJE**.

Závěr

Nyní shrnu předchozí údaje do následující tabulky 2.1:

Tabulka 2.1: Výběr databázové technologie

technologie	podpora formátu JSON	kompat. s Node.js	dostupnost zdarma
MySQL	DOBRÉ	SPLŇUJE	SPLŇUJE
SQLite	DOSTATEČNÉ	SPLŇUJE	SPLŇUJE
MongoDB	VÝBORNÉ	SPLŇUJE	SPLŇUJE
PostrgeSQL	DOBRÉ	SPLŇUJE	SPLŇUJE

Jak je vidět v tabulce 2.1, všechny potenciální technologie splňují požadavky na kompatibilitu s Node.js a na dostupnost zdarma. Tudíž rozhodujícím kritériem se stává vhodnost pro ukládání dat přímo ve formátu JSON. Zde zvítězila databáze MongoDB s hodnocením VÝBORNÉ. Proto **použiji ve své práci právě databázovou technologii MongoDB**.

Nástroj pro správu databáze

Jako nástroj pro správu MongoDB databáze použiji nástroj **Robo 3T** [13] (známý i pod starším názvem *Robomongo*). Nástroj jsem si nainstaloval, zkusil použít a byl jsem spokojen. Jedná se o jednoduchý a intuitivní nástroj, který po připojení k databázi umožňuje prohlížet a upravovat data uložená v databázi. Pro účely mé práce je zcela postačující.

Kapitola 3

Návrh

V této kapitole budu provádět návrh nové podoby serverové části. Jako první popíši strukturu dat ukládaných v databázi. Poté navrhnu a zdokumentuji nové komunikační API. Následně se budu zaobírat serverovou aplikací, kde na názorném příkladu jedné operace vysvětlím architekturu serverové aplikace. V rámci toho také vysvětlím způsob autentifikace uživatelů. Dále budu zavádět průběžnou integraci spolu s průběžným testováním. Nakonec budu řešit problematiku GUI pro monitoring serveru a GUI pro konfiguraci serveru.

3.1 Databáze

Jak jsem již rozhodl, jako databázi použiji dokumentově orientovanou databázi MongoDB, která se řadí mezi takzvané NoSQL databáze. Na rozdíl od standardních relačních databází nemá MongoDB pevně definovanou strukturu ukládaných dat. Data jsou ukládána ve formátu BSON (binární podoba formátu JSON). Jednotlivé termíny, používané pro data v rámci MongoDB společně s jejich ekvivalenty z relačních databází, uvádím v následující tabulce 3.1. Velkým rozdílem oproti relačním databázím je pak skutečnost, že není vyžadováno, aby měly veškeré dokumenty v rámci jedné kolekce stejná všechna pole (angl. fields). Také považuji za důležité možnost ukládat jako součást dokumentu další vnořené objekty, případně pole (kolekci) objektů. Pro toto shrnutí vlastností MongoDB jsem čerpal z [14] a [15].

Tabulka 3.1: Terminologie MongoDB

relační databáze	**MongoDB**
tabulka	kolekce
řádek	dokument / záznam
sloupec	pole (angl. field)

Nyní popíši jednotlivé kolekce ukládané v databázi a údaje, které evidují.

Uvádím ve formátu JSON.

Uživatel – User

Jedná se o záznam o uživateli (učitel nebo student). Údaje jsou smyšlené a slouží pouze pro názornost. Datum narození je nepovinný údaj. Pole `password` bude pravděpodobně obsahovat hash hesla, přesný systém ovšem závisí na implementaci *jádra* a *rozhraní pro učitele*. (Server pouze porovnává údaj obdržený při přihlašování s údajem uloženým v databázi a nestará se o to, zda se jedná o hash nebo o otevřený text hesla.)

```
{
  "id": "5ae0ba96654ca25660c70b8e",
  "name": "Petr",
  "surname": "Novák",
  "role": "student",
  "birthday": "2010-02-24",
  "username": "PetrNovak31",
  "password": "abcd1234"
}
```

Module exercise – Modul cvičení

Na rozdíl od Dráčka II je nově evidována i současná verze a také poslední podporovaná verze (modul se starší verzí už *jádro* nedovolí spustit).

```
{
  "id": "5ae0ba96654ca25660c70665",
  "name": "name of module",
  "packageNameExcercise": "cz.something.something",
  "packageNameEditor": "cz.something.something",
  "description": "description of module",
  "image": "file",
  "googlePlayUrl": "url"
  "version": 5,
  "lastSupportedVersion": 3,
}
```

Exercise – Zadání cvičení

```
{
  "id": "5ae0ba96654ca25660c703c3",
  "moduleExcerciseId": "5ae0ba96654ca25660c70665",
  "name": "name of excercise",
  "description": "description of excercise",
  "file": "file",
  "depends": 2,
  "done": true,
  "task": true
}
```

Score – Výsledek (Hodnocení)


```
{
  "id": "5ae0ba96654ca25660c3f7b2",
  "moduleExcerciseId": "5ae0ba96654ca25660c70665",
  "excerciseId": "5ae0ba96654ca25660c703c3",
  "percentage": 20,
  "totalTime": 20,
  "file": "file",
  "finished": true,
  "finishedAsTask": false
}
```


Task – Úkol


```
{
  "id": "5ae0ba96654ca25660c70cd8",
  "moduleExcerciseId": "5ae0ba96654ca25660c70665",
  "excerciseId": "5ae0ba96654ca25660c703c3",
  "userId": "5ae0ba96654ca25660c70b8e",
  "endingDate": 0,
  "minimalScore": 20,
  "active": 1,
  "loaded": 1,
  "branchId": "5ae0ba96654ca25660c74f52",
  "version": 4,
  "deleted": false
}
```


Class – Třída

Jedná se o záznam o třídě (myšleno jako školní třída, kterou navštěvují žáci).

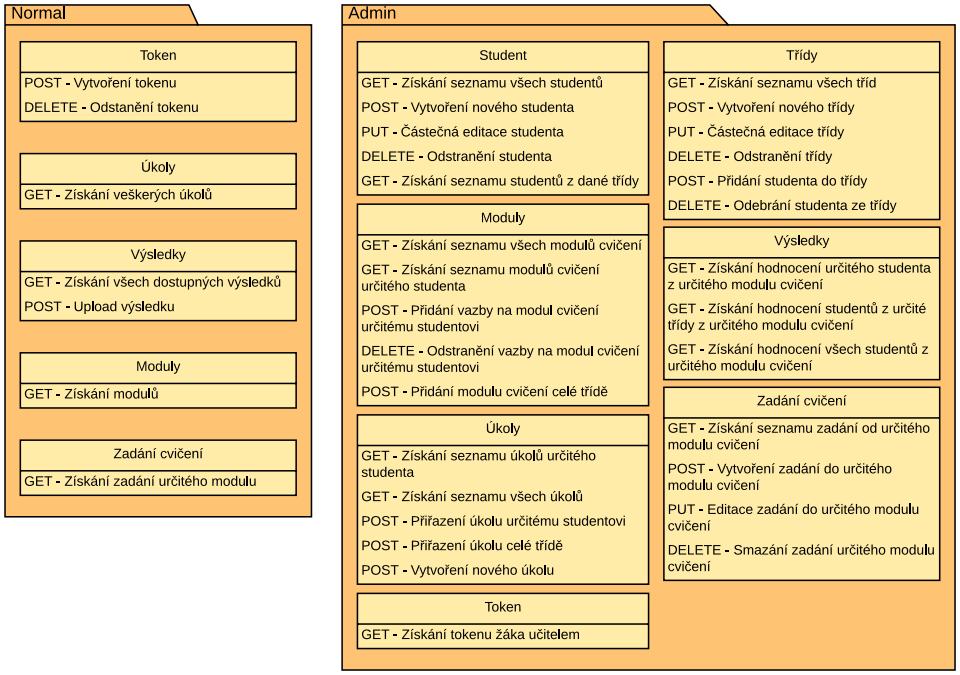

```
{
  "id": "5ae0ba96654ca25660b391b4",
  "name": "4.A"
}
```


V databázi se také nacházejí další pomocné dokumenty, ty ale popisovat nebudu.

3.2 Komunikační API

Při vytváření komunikačního rozhraní jsem odstranil všechny chyby odhalené v rámci analýzy (viz 2.6). Dále jsem také rozšířil API o operaci umožňující přihlášení žáka učitelem (určenou v rámci funkčního požadavku FP1.1). Upozorňuji na možná nepříliš vhodné pojmenování „admin“ části API týkající se operací, které může provádět jen učitel. Dle mého názoru by bylo vhodnější označení „teacher“, ale jelikož je prefix součástí všech endpointů učitelské části, jež jsou již implementovány v *rozhraní pro učitele*, nebudu prefix měnit, protože jak jsem si ověřil, nachází se v zdrojovém kódu *rozhraní pro učitele* na více místech a přidělal bych tak další práci.

Pro názornost jsem vytvořil následující dva diagramy. Diagram 3.1 znázorňuje rozdělení operací do skupin dle účelu (tagy). Diagram 3.2 znázorňuje hierarchii endpointů.



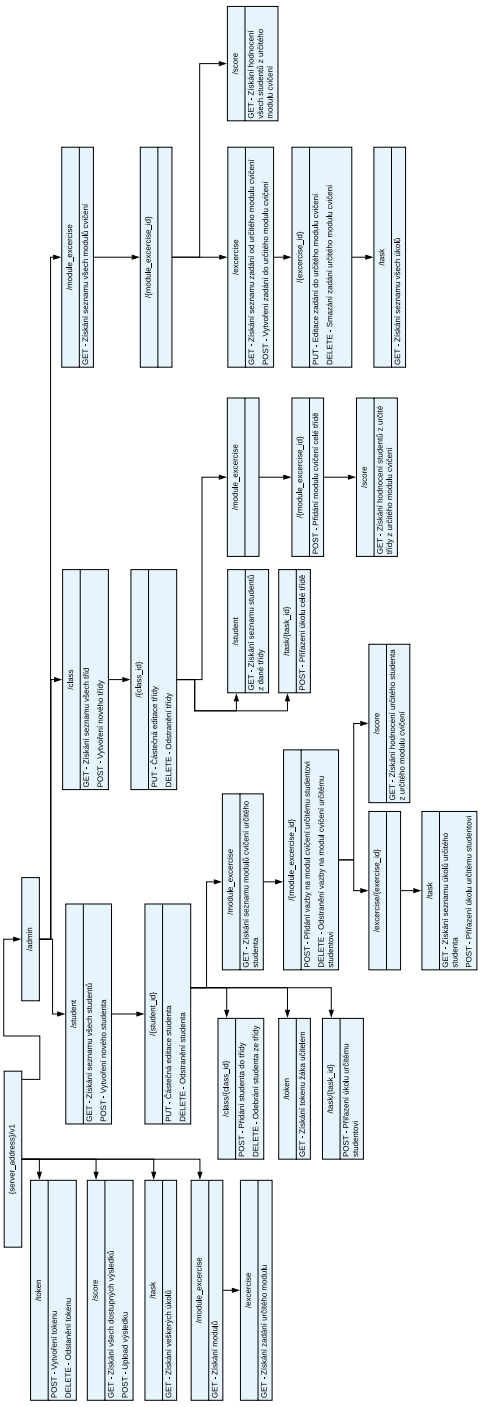
Obrázek 3.1: Diagram rozdělení API dle tagů

3.2.1 Dokumentační nástroj Swagger

Pro zdokumentování plné podoby specifikace komunikačního rozhraní jsem se rozhodl použít nástroj Swagger [16]. Jedná se online dostupný nástroj určený přímo pro popis RESTful API. V nástroji jsem sepsal kompletní specifikaci komunikačního API pro server projektu Dráček IV. Nástroj nyní stručně zhodnotím dle zkušeností, které jsem získal po zdokumentování celého API.

Klady

- Podporuje autokompilaci (kód se automaticky průběžně kompiluje během psaní).
- Případné chyby při kompilaci dokumentace jsou zobrazovány přehledně a intuitivně.
- Umožňuje třídění operací do tagů (třídění dle účelu).
- Použitý jazyk jsem shledal intuitivním.



Obrázek 3.2: Diagram hierarchie endpointů

- Umožňuje i jiné formáty než JSON (můžou fungovat i paralelně). Ve své práci tohoto v tuto chvíli nevyužívám, ale jedná se o zajímavou možnost pro případné rozšiřování serverové aplikace.

Zápory

- Při finální délce kódu dokumentace přes 1700 řádek se stává kód značně nepřehledným. Nepodařilo se mi najít uspokojivou možnost, jak kód rozdělit do více souborů.

- S delším kódem se zpomaluje autokompilace a systém obecně reaguje pomaleji. Ovšem připouštím možnost, že příčina tohoto zpomalení leží jinde v mém počítači. Neprováděl jsem v tomto ohledu žádný další podrobnější výzkum.

Nová specifikace RESTful API, kterou jsem vytvořil, je online dostupná na serveru Dráčka IV, a to na adrese [17]. Pokud by byla specifikace na uvedené adrese nedostupná, lze ji také nalézt ve formátu YAML a ve formátu JSON v příloze mé bakalářské práce, pro její zobrazení je potřeba použít swagger editor dostupný na adrese: `https://editor.swagger.io/` (doporučuji online editor) a zvolit možnost importovat specifikaci ve formátu YAML nebo JSON.

Snímek obrazovky z dokumentace můžete vidět na obrázku 3.3. Popisuje operaci *„PUT - Částečná editace studenta“*. Zde v práci umísťuji jen ilustrační kus. Celou (3 strannou) dokumentaci zmíněné operace můžete najít v příloze.

3.2.2 Chybové zprávy

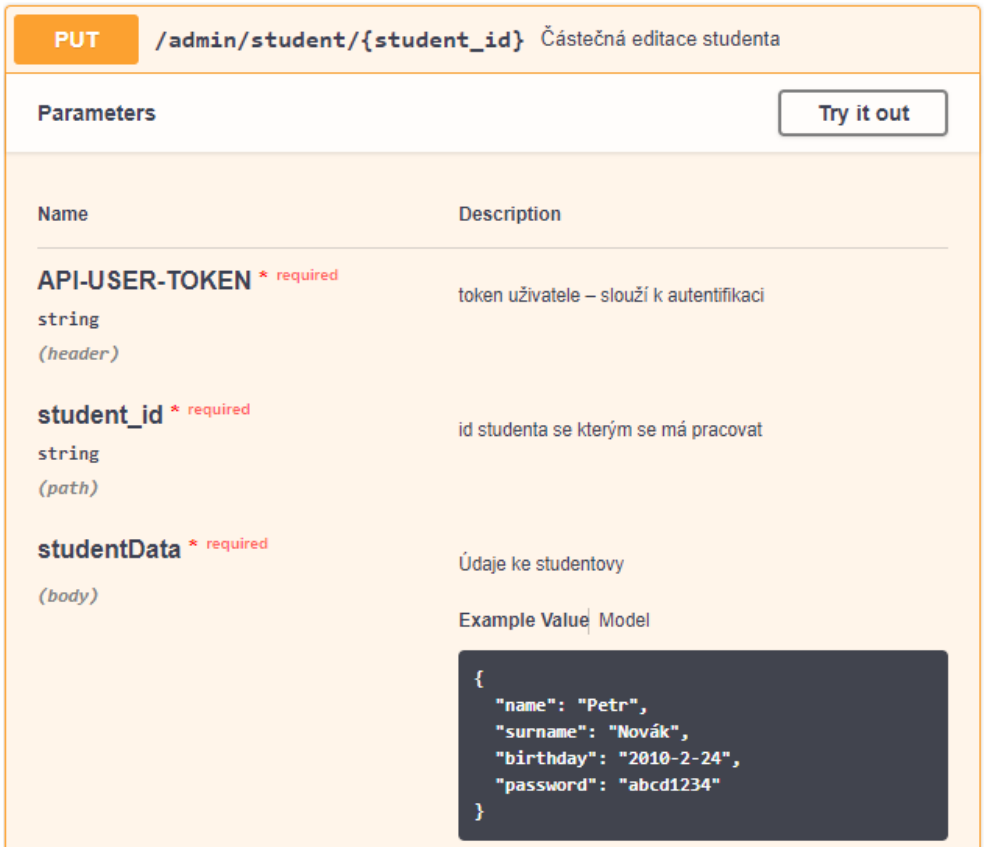
Během komunikace může nastat celá řada chybových scénářů, jako například při zadání špatných přihlašovacích údajů, nebo při snaze o vytvoření třídy s duplicitním jménem.

Pro vytváření chybových hlášek využívám knihovnu (modul) Boom. Formát chybových hlášek ukážu na příkladu pro chybu způsobenou zasláním požadavku s id neexistujícího žáka.

```
{
  "statusCode":404,
  "error":"Not Found",
  "message":"Student not found"
}
```

Položka `statusCode` obsahuje kopii čísla „response code“ a v položce `error` je odpovídající textové pojmenování k „response code“. Položka `message` obsahuje textový popis chyby vhodný pro zobrazení člověkem (programátorem nebo i uživatelem, pokud není řešeno jinak).

V rozhraní z Dráčka II nejsou vůbec uváděny odpovědi metod na chybové scénáře. Jedním z mých hlavních úkolů při tvorbě nové specifikace komunikačního rozhraní bylo doplnit metody o správné řešení chybových scénářů.



Obrázek 3.3: Snímek obrazovky z dokumentace API v nástroji Swagger [16] – operace Částečná editace studenta

Uvedu příklad několika možných chybových zpráv (přesněji obsah položky `message`):

- Bad username or password
- USER TOKEN expired or not found
- Student not found
- Missing class name in method body
- You can edit only students

3.2.3 Autentifikace – Token

V rámci Dráčka II byl vymyšlen princip autentifikace uživatelů. Tento systém autentifikace byl dle mého názoru zvolen správně a proto ho použiji i v nové

verzi. Nyní princip autentifikace popíši tak, jak bude použit v nové verzi.

Vzhledem ke skutečnosti, že komunikace s RESTful API probíhá bezstavově (server si mezi jednotlivými požadavky nic nepamatuje), bylo by pro ověřování identity uživatelů potřeba posílat s každým požadavkem na server i uživatelovy přihlašovací údaje. To by ale bylo potenciálně nebezpečné, neboť přihlašovací údaje by vzhledem k jejich častému výskytu bylo snadnější odposlechnout. Proto je použit princip takzvaného „tokenu“. Ten funguje následovně:

Pokud chce subjekt komunikovat se serverem, nejdříve odešle na server své přihlašovací údaje a zpět dostane token a informaci o jeho době expirace. Samotný token je řetězec znaků jednoznačně identifikující uživatele. Je náhodně generovaný na serveru. Server si ukládá seznam těchto tokenů společně s id uživatelů, kterým daný token patří. Subjekt dále při každém požadavku posílá token jako parametr `API-USER-TOKEN` předávaný v headru požadavku a server na jeho základě identifikuje uživatele. Subjekt při odhlašování požádá o odstranění tokenu, ale kdyby to neudělal, nic se nestane, neboť tokenu samovolně vyprší doba platnosti. V případě, že server nenalezne obdržený token (pravděpodobně vypršela doba platnosti a byl smazán), vrací následující chybovou hlášku:

```
{
  "statusCode":401,
  "error":"Unauthorized",
  "message":"USER TOKEN expired or not found"
}
```

Tokeny jsou ukládány v databázi. Bylo by také možné tokeny ukládat pouze v serverové aplikaci, jelikož není potřeba mít zajištěné perzistentní uložení. Uložení pouze v serverové aplikaci by bylo pravděpodobně výkonově výhodnější, ale nakonec jsem tuto možnost nezvolil. Důvodem je, že tokeny je potřeba po expiraci doby platnosti odstraňovat, což ve spojení s potřebou parelelního přístupu způsobuje obtíže. Databáze automaticky takovéto problémy řeší. Další výhodou ukládání v databázi je, skutečnost, že MongoDB pro jednotlivé záznamy automaticky generuje unikátní identifikátor, který tak mohu použít jako token. Neočekávám navíc, že by se jednalo o významný výkonnostní problém.

Dalším bezpečnostním prvkem je takzvaný *API key*. Je posílán v headru požadavku na vytvoření tokenu pod názvem `API-API-KEY`. Jedná se o řetězec znaků, který je známý pouze serveru a jednotlivým dalším subjektům, které smějí komunikovat se serverem. V tuto chvíli existují dva různé *API key*, a to jeden pro *jádro* a jeden pro *modul pro učitele*.

3.3 Serverová aplikace

Pro serverovou aplikaci mi vedoucí práce nabídl využít jako prototyp jinou již vypracovanou serverovou aplikaci poskytující RESTful API a využívající

MongoDB. Jedná se o aplikaci, která vznikla v rámci diplomové práce [9]. Aplikaci jsem analyzoval a došel jsem k názoru, že po architektonické stránce je velmi dobře navržena. Proto nakonec pro svou aplikaci využiji stejné architektury, neboť si nemyslím, že bych dokázal vytvořit lepší. Navíc vzhledem k tomu, že na této architektuře již byla vytvořena zmiňovaná funkční aplikace, je možné tvrdit, že řešení je ověřeno jako funkční.

Největším problémem při snaze využít zmiňované aplikace jako prototypu byla skutečnost, že se nejednalo o standardní RESTful API aplikaci. Aplikace provádí nad daty poměrně složitou aplikační logiku. V aplikaci se tak naneštěstí nikde nevyskytoval žádný „obyčejně se chovající“ endpoint. Mojí prací proto zpočátku bylo zjednodušit a vybrat potřebné části pro vytvoření „běžného“ endointu. Za běžný endpoint pak považuji ten, který umožňuje nad daty provádět CRUD operace s tím, že na datech je při ukládání provedena nějaká jednoduchá logika.

Pro názorné vysvětlení architektury aplikace a způsobu, jak ji využívám ve své aplikaci, jsem se rozhodl, že jako příklad využiji endpoint „student“. Na něm názorně vysvětlím, co vše obnáší vytvoření jednoho endpointu. Endpoint „student“ jsem si vybral, protože se nejedná ani o nejjednodušší, ani o nejsložitější endpoint. Je tedy dostatečně názorný, ale zároveň není příliš obsáhlý, takže ho mohu v této práci ukázat. Ostatní endointy jsou pak vytvořeny obdobným způsobem. Jako ukázkovou operaci jsem vybral *„PUT - Částečná editace studenta“*. Operace umožňuje editaci všech údajů studenta s výjimkou id a uživatelského jména, které jsou v systému neměnné.

3.3.1 Definice endpointu a operací

Základní definice endpointu a operací se společně s definicemi všech ostatních endpointů nachází v souboru *server.js*. Pro vytvoření REST serveru je využita knihovna (modul) *Hapi*. Následuje ukázka z kódu pro operaci *„PUT - Částečná editace studenta“*.

```
server.route({
      method: 'PUT',
      path: prefix+'student/{id}',
      handler: require('./lib/controllers/student.js').put,
      config: {
        tags: ['api'],
        validate: {
          query: false,
          params: require('./lib/validators/student.js').params,
          payload: require('./lib/validators/student.js').payload,
          header: require('./lib/validators/token.js').commonHeader,
        },
        response: {
          schema: require('./lib/validators/student.js').response,
          modify: false
        }
      }
    });
```

V ukázce je vidět, že v dalších samostatných souborech leží controller (zajišťující aplikační logiku) a validace. Pro validaci hlavičky požadavku využívám společnou metodu, protože všechny požadavky (kromě požadavku na vytvoření tokenu) v hlavičce obsahují token uživatele.

3.3.2 Aplikační logika (Controller)

v souboru *lib/controllers/student.js* se nachází controller pro endpoint student. Následuje ukázka kódu pro operaci *„PUT - Částečná editace studenta“*.

```
put: function(request, reply) { //editace studenta
  reply(mongoose.model('Student').findById(request.params.id).exec()
      .then(function (stud) {
    if (tokenController.getUserId(request.headers) == null)
    {
        throw Boom.unauthorized('Token expired or not found');
    }
    else if (stud == null) {
        throw Boom.notFound('Student not found');
    }
    else {
      if(stud.role != 'student') {
        throw Boom.forbidden('You can edit only students');
      }
      if(request.payload.name != null) {
        stud.name = request.payload.name;
      }
      if(request.payload.surname != null) {
        stud.surname = request.payload.surname;
      }
      if(request.payload.password != null) {
        stud.password = request.payload.password;
      }
      if(request.payload.birthday != null) {
        stud.birthday = request.payload.birthday;
      }
    }
    stud.save();
    return {
      id:stud._id,
      name:stud.name,
      surname:stud.surname,
      username:stud.username,
      role:stud.role,
      password:stud.password,
      birthday:stud.birthday};
  }));
},
```

3.3.3 Validace

Při validaci využívám knihovnu (modul) *Joi*. Validaci provádím jak pro údaje v požadavku, tak pro odpověď.

Validace příchozích údajů slouží k ověření, že na server jsou odesílané v rámci požadavku veškeré potřebné údaje, a že jsou ve správném formátu. Ověřuje tak vlastně chování klientské aplikace, nikoliv samotného serveru.

Validace odchozích odpovědí pak slouží k ověření, zda data obsažená v odpovědi odpovídají očekávanému formátu a jedná se tak o ověření správného chování samotného serveru. Navíc u některých operací využívám možnosti, že v rámci validace mohou být údaje přejmenovány či z odpovědi odstraňovány. U operací, kde tohoto nevyužívám, by v případě, že by se někdy v budoucnu zvýšily nároky na výkonnost serverové aplikace, bylo možno nastavit, aby se kontroly odpovědí prováděly jen na určitém náhodném vzorku odpovědí (např. jen pro 10% všech odpovědí).

Jako ukázku použiji validaci údajů obsažených v těle požadavku *„PUT - Částečná editace studenta“* uvedenou v souboru *lib/validators/student.js*.

```
exports.payload = Joi.object().keys({
  name: Joi.string().regex(/^[a-zA-Z]+$/).required(),
  surname: Joi.string().regex(/^[a-zA-Z]+$/).required(),
  password: Joi.string().required(),
  birthday: Joi.string().allow("").regex(generic.dateRegex),
});
```

Na heslo nejsou v tuto chvíli žádné požadavky. Jako heslo na server bude pravděpodobně odesílán hash SHA-256, ale na serveru to nevaliduji a umožňuji tak, aby to mohlo být na straně *jádra* a *rozhraní pro učitele* případně změněno.

Datum narození má být ve formátu „YYYY-MM-DD“. To je ověřováno regulárním výrazem umístěným v souboru *lib/validators/generic.js*.

3.3.4 Model

V rámci databáze je student pouze druh uživatele, s polem role obsahujícím hodnotu „student“. Model pro data uživatele se nachází v souboru *lib/model/user.js*. Ukázku zde neuvádím, protože se jedná pouze o definici názvů a datových typů jednotlivých polí záznamu v databázi a není tudíž příliš zajímavá.

3.4 Průběžná integrace

Pro serverovou aplikaci jsem úspěšně zavedl průběžnou integraci, použil jsem při tom části kódu (sloužícího pro průběžnou integraci) z již zmíněného prototypu z [9]. Pro vysvětlení, co je účelem průběžné integrace, jsem se rozhodl citovat následující definici. Myslím si, že dobře a stručně vystihuje podstatu problematiky.

„Continuous integration je volně řečeno souhrn praktik a nástrojů, při kterých vývojáři integrují (commitují) své změny často (typicky alespoň jednou za den). Každá integrace je automaticky ověřena testy a případně může vést až k automatickému nasazení nové verze aplikace (continuous delivery), v případě neúspěchu k okamžitému reportování problému. Přínosy continuous integration jsou především zrychlení vývoje, rychlejší dodávání nových verzí a snížení

chybovosti – to vše díky automatizaci co nejvíce úkonů, které je třeba dělat při vývoji, testování a nasazovaní aplikace.“ [18]

Jak je vidět z předchozí definice, pojem průběžná integrace je poměrně široký. Nyní popíši postup, jak průběžná integrace probíhá v rámci této práce. Je potřeba, aby veškeré zdrojové kódy byly umístěné v systému pro správu verzí Git.

1. Vývojář provede změny v repozitáři. (commit, push)

2. Automaticky se zahájí proces průběžné integrace dle souboru *.gitlab-ci.yml*.

3. Jsou spuštěny automatické testy – probíhá následovně:

   a) Je vytvořen nový dočasný obraz (virtuální systém), na kterém budou prováděny testy.

   b) Před každým testem nebo skupinou testů jsou do databáze vložena připravená testovací data nacházející se v souboru *config/db/test.js*. Následně jsou prováděny jednotlivé testy. Veškeré prováděné testy se nachází ve složce *tests*. Výsledky jednotlivých testů lze sledovat v konzoli v Gitu. Příklad jednoho testu uvádím v 3.5.

   c) Po dokončení všech testů je dočasný obraz odstraněn.

4. Pokud byly všechny testy úspěšné, následuje **nasazení** – probíhá následovně:

   a) Dle scriptu *makedeb.sh* je vytvořen instalační balíček pro aplikaci.

   b) Instalační balíček je uložen v repozitáři. K repozitáři má z bezpečnostních důvodů přístup pouze vedoucí práce, který prováděl jeho nastavení.

   c) Git se pomocí protokolu SSH přihlásí na cílový server, kde má být nasazena aplikace.

   d) Aplikace je nainstalována pomocí příkazů *apt-get update* a *apt-get install*.

   e) Po dokončení instalace je spuštěn script *debian/postinst*, který spouští serverovou aplikaci jako službu. Script navíc kontroluje, zda je spuštěna služba *mongod* obsluhující MongoDB databázi a v případě, že není, ji spustí.

5. V případě, že testy nebo nasazení selže, může Git dle nastavení provést další akce (jako např. odeslat správci varovný email).

3.5 Průběžné testování

Před testy jsou vždy do části databáze určené k testování vloženy připravené testovací údaje.

Uvedu zde jako ukázku jednodušší test na požadavek textit„GET - Získání seznamu všech studentů“. V databázi ve chvíli spuštění testu je jeden ukázkový záznam o studentovi. Je testováno, zda jsou v odpovědi správně uvedeny všechny požadované údaje.

```
lab.test('can get default test student', function (done) {
  common.getStudents()
  .then(function(students) {
    console.log(students);
    expect(students).to.be.instanceof(Array);
    expect(students.length).to.equal(1);
    let idx = students.findIndex(hv => hv.username === 'TallJohn254');
    expect(students[idx].name).to.equal('John');
    expect(students[idx].surname).to.equal('Tall');
    expect(students[idx].password).to.equal('test1');
    expect(students[idx].role).to.equal('student');
    done();
  }).catch(common.e);
});
```

3.6 GUI pro konfiguraci a monitoring serverové části

V této části práce se budu nejdříve věnovat problematice monitorování. Následně pak budu řešit problematiku konfigurace. Pro tu jsem nalezl tři oblasti, které má potencionálně smysl konfigurovat, a to jsou: databáze, serverová aplikace a samotný serverový stroj.

3.6.1 GUI pro monitoring serveru

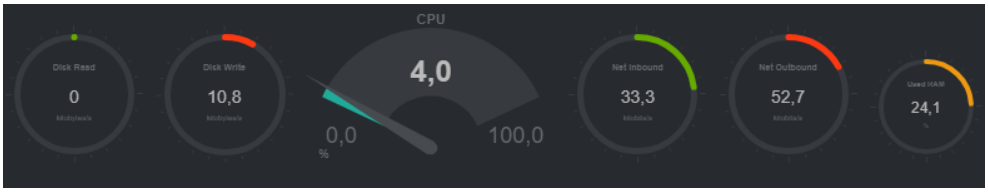
GUI by mělo umět sledovat využití CPU, RAM, disku a sítě. Případná podpora sledování nějakých dalších veličin by rozhodně také nebyla na škodu. GUI by mělo být dostupné online z libovolného počítače. Došel jsem k názoru, že implementovat vlastní GUI pro tento účel by bylo zbytečné, jelikož již existuje celá řada nástrojů plně pokrývajících danou problematiku. Nejdříve jsem se pokusil využít nástroj Zabbix [19], který mi byl doporučen vedoucím práce. Nástroj sice danou problematiku plně pokrývá, ovšem nakonec jsem tento nástroj nepoužil. Důvod vysvětlím v následujícím odstavci.

Nástroj funguje principem, že na sledovaném stroji je spuštěn pouze takzvaný Agent, který sbírá data a odesílá je na server. Tento server by měl být ideálně umístěn na jiném stroji. Systém tak umožňuje sledovat více strojů najednou, což je ale pro tento projekt zbytečné. Nástroj by sice pravděpodobně bylo možně celý nainstalovat na jeden stroj, ovšem v takovém případě ztrácí

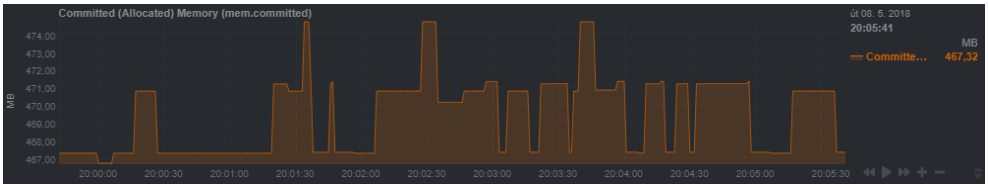
značná část nástroje smysl a zůstává tak jen potřeba složité postupné instalace a konfigurace více různých komponent. Z tohoto důvodu jsem se rozhodl vyhledat jiný nástroj.

Nakonec jsem zvolil nástroj NetData [20]. Na rozdíl od nástroje Zabbix je instalace tohoto nástroje mnohem jednodušší a není vyžadována žádná složitá konfigurace. Nástroj shledávám přehledným a zároveň velmi obsáhlým. Nástroj umožňuje sledovat nejen požadované veličiny, ale poskytuje i další podrobné statistiky o využití systému. Na obrázku 3.4 můžeme vidět ukázku z GUI zobrazující základní přehled současného využití sledovaných veličin. Dále na obrázku 3.5 můžeme vidět podrobnější statistiku využití paměti. Další snímky obrazovky lze najít v příloze.

Nástroj jsem úspěšně nasadil na server Dráčka IV a GUI pro monitoring je dostupné na adrese serveru s portem 19999: `http://185.88.73.25:19999/`.



Obrázek 3.4: Ukázka z GUI pro monitoring zobrazující základní přehled současného využití sledovaných veličin (nástroj NetData [20])



Obrázek 3.5: Ukázka z GUI pro monitoring zobrazující podrobnější statistiku využití paměti (nástroj NetData [20])

3.6.2 GUI pro konfiguraci databáze

Základní potřebnou konfiguraci provádí serverová aplikace automaticky v rámci nasazení. Mezi tu patří:

- Kontrola, zda je obslužný program databáze spuštěn – pokud ne, pak ho aplikace spustí.
- Pokud neexistuje databáze, je vytvořena a naplněna výchozími daty.

Pro další konfiguraci je možné využít nástroje Robo 3T [13], který poskytuje intuitivní GUI pro editaci obsahu databáze. Do této konfigurace bych zařadil:

- vytvoření, editace a odstranění učitelských účtů
- vytvoření, editace a odstranění modulů cvičení

Zmíněné úkony není potřeba provádět příliš často, proto tento způsob shledávám dostačujícím.

3.6.3 GUI pro konfiguraci serverové aplikace

Serverovou aplikaci není v zásadě potřeba konfigurovat. Jediná záležitost, kterou dříve bylo potřeba konfigurovat, byla IP adresa, na které serverová aplikace naslouchá. V původním prototypu z [9] bylo nutné přepsat v konfiguračím souboru IP adresu na skutečnou adresu serveru. Tento problém jsem ovšem odstranil tím, že serverová aplikace nyní automaticky naslouchá na všech IP adresách.

3.6.4 GUI pro konfiguraci serverového stroje

Došel jsem k názoru, že veškeré potřebné konfigurace serverového stroje lze provádět v GUI systému Big Cloud [21], pod kterým je vytvořen virtuální stroj serveru Dráček IV. V systému lze měnit množství přidělených prostředků, mezi které patří:

- počet jader procesoru
- množství paměti
- disky
- *a další*

Pokud tedy v GUI pro monitoring zjistíme, že je nedostatek nějakého prostředku, můžeme zvýšit jeho příděl snadno v systému Big Cloud. Tento přístup je výhodný, jelikož zvýšení přidělených prostředků vede ke zvýšení denní ceny účtované za provoz serveru, tudíž je vhodné, aby nebylo serveru přiděleno zbytečně mnoho prostředků.

Kapitola 4

Realizace

Na začátku této kapitoly nejdříve zmíním pár problémů, které jsem řešil při implementaci serverové aplikace. Dále v této kapitole naleznete uživatelskou příručku popisující způsoby, jak odesílat na server požadavky. Důraz kladu na nástroj cURL [22], který jsem shledal užitečným a hojně jsem využíval pro testování správné funkce serverové aplikace. Také pak v této kapitole najdete instalační příručku popisující, jak systém nasadit na nový stroj.

4.1 Řešené problémy při implementaci serverové aplikace

Jak jsem již dříve zmiňoval, prototyp, z něhož vycházím, sloužil pro nestandardní API. Neobsahoval tak žádný „obyčejný“ endpoint. V první řadě bylo potřeba zjednodušit stávající endpointy a na jejich základě vytvořit nový „obyčejný“ endpoint, což mi zabralo více času, než jsem původně předpokládal. Z dalších problémů, jež se vyskytly, bych chtěl zmínit následují:

V rámci API jsem potřeboval využít takzvané agregace (obdoba operace JOIN relačních databází). Ve využívaném prototypu jsem její použití ovšem nikde nenašel, musel jsem si tedy zjistit sám, jak ji použít. To by běžně neměl být problém, ovšem i přes veškerou snahu se mi nedařilo agregaci zprovoznit. Nakonec jsem problém odhalil. Ten byl způsoben skutečností, že pro serverovou část jsem použil nejnovější verzi databáze MongoDB, zatímco v serverové aplikaci byla starší verze modulu *mongoose* komunikující s databází. V databázi byly v rámci některé z aktualizací mírně změněny požadavky na operaci agregace, které starší verze modulu nebyla schopna obsloužit. Naneštěstí ovšem modul nebyl schopen tento problém nahlásit, a proto mi trvalo docela dlouho, než jsem tuto příčinu odhalil.

4.2 Uživatelská příručka – návod na odesílání požadavků na server

Rozhodl jsem se, že napíši návod, jak komunikovat se serverem, neboli jak posílat REST požadavky. Ukážu zde tři způsoby odesílání požadavků.

4.2.1 Pomocí webového prohlížeče

Jedná se o nejjednodušší možnost, ovšem silně omezenou. Tímto způsobem lze odesílat pouze GET požadavky, a navíc bez hlavičky a těla. Stačí zadat cestu k serverové aplikaci společně s endpointem a parametry do webového prohlížeče. Pro vyzkoušení otevřete v prohlížeči následující URL: `http://185.88.73.25:25785/v1/admin/student/`. Jedná se o požadavek sloužící k získání seznamu všech studentů. Vzhledem k tomu, že v rámci požadavku chybí hlavička obsahující token, přijde jako odpověď chybová hláška. Jelikož všechny požadavky serveru Dráček IV vyžadují nějaká data v hlavičce, není možno získat pomocí webového prohlížeče jinou než chybovou odpověď.

4.2.2 Nástroj cURL

Nástroj cURL [22] umožňuje posílat REST požadavky přímo z příkazové řádky. Jeho použití je rychlé a jednoduché. Po nainstalování nástroje stačí do příkazové řádky napsat vhodně upravený následující příkaz, který jsem připravil jako ukázku:

```
1 curl -i -X POST -w "\n" \
2 -H 'Content-Type: application/json' \
3 -H 'API-API-KEY: abcde12345' \
4 -d '{"username": "abc", "password": "abc321"}' \
5 http://185.88.73.25:25785/v1/token
```

Tento příkaz slouží pro odeslání požadavku na získání uživatelského tokenu. Pokud příkaz použijete přímo ve výše uvedené formě, měli byste od serveru obdržet jako odpověď chybovou hlášku. Nyní vysvětlím význam jednotlivých částí příkazu. První řádek kódu definuje, že se jedná o operaci POST, a navíc říká, že má být za výstup celého příkazu přidán nový řádek (cURL to jinak neudělá). Na druhém řádku je v hlavičce předávána informace, že komunikace probíhá ve formátu JSON. Na třetím řádku je v hlavičce předáván *API key*. Pro správné fungování požadavku je ho potřeba nahradit skutečným klíčem, který zde ovšem z bezpečnostních důvodů neuvádím. Na čtvrtém řádku je definován obsah těla požadavku. V něm je obsaženo uživatelské jméno a heslo. Pro použití změňte na hodnoty skutečného uživatele. Jako heslo by měl být posílán hash SHA-256 skutečného hesla. Na pátém řádku je adresa serveru spolu s endpointem.

Pokud změníte zmíněné hodnoty na platné hodnoty, měli byste od serveru obdržet jako odpověď token společně s dobou jeho platnosti a také vaše

uživatelské údaje. Přesný formát všech možných odpovědí můžete nalézt ve specifikaci API.

4.2.3 Knihovny pro programovací jazyky

Vzhledem k tomu, že technologie REST je běžně využívána, existují pro odesílání REST požadavků (neboli fungování jako REST Client) knihovny pro většinu běžných programovacích jazyků. Například pro jazyk Java bych doporučil článek [23], názorně popisující dva způsoby, jak odesílat REST požadavky.

4.3 Instalační příručka

V této sekci popíši jak nainstalovat celý systém serverové části na nový stroj. Jako operační systém doporučuji operační systém Ubuntu. Na serveru, který jsem využíval já, byl systém ve verzi 14.04. Následují návody, jak nainstalovat jednotlivé potřebné komponenty systému. Uvádím je ve stejném pořadí, jak jsem je instaloval na server já.

4.3.1 Node.js

Instalaci jsem prováděl dle návodu z [24], a to následujícími příkazy:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Pokud se vyskytnou problémy, hledejte řešení nebo jiný způsob instalace na oficiálních stránkách (`nodejs.org`).

4.3.2 MongoDB

Jako MongoDB databázi budeme instalovat zdarma dostupnou verzi, která je na oficiálních stránkách nazývána *Community server*. Při instalaci jsem následoval návod [25], ze kterého pocházejí následující příkazy, které by měly provést instalaci softwaru:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
--recv 2930ADAE8CAF5059EE73BB4B58712A2291FA4AD5
echo "deb [ arch=amd64 ] \
https://repo.mongodb.org/apt/ubuntu \
trusty/mongodb-org/3.6 multiverse" | \
sudo tee /etc/apt/sources.list.d/mongodb-org-3.6.list
sudo apt-get update
sudo apt-get install -y mongodb-org
```

V případě, že provádíte instalaci bez využití průběžné integrace, je potřeba databázi nastartovat, a to následujícím příkazem.

```
sudo service mongod start
```

Pokud využíváte průběžné integrace, databázi nastartuje script automaticky spouštěný po nasazení.

4.3.3 Serverová aplikace při využití průběžné integrace

Pokud jste již nainstalovali Node.js a MongoDB, můžete přistoupit k instalaci samotné serverové aplikace. Nejdříve popíši, jak provést instalaci s využitím průběžné integrace, poté napíši, jak bez ní.

S využitím průběžné integrace

Aby Git mohl provést nasazení na cílový server, budete potřebovat na serveru mít uživatele s pravomocemi na použití příkazu *apt-get*. Pokud takového uživatele nemáte, můžete ho vytvořit následujícím postupem:

1. Standartním způsobem vytvořte v systému nového uživatele. V ukázce dále ho mám pojmenovaného „usergit“.
2. Otevřete soubor definující práva uživatelů. K tomu použijte následující příkaz: `sudo visudo` spouštějící speciální textový editor *Visudo*. Tento editor na rozdíl od standardního textového editoru zajistí skutečnost, že v případě, že do souboru zapíšete neplatná data, nový soubor nebude použit a zůstane původní nezměněný soubor.
3. Do souboru připište následující řádek („usergit“ je název uživatele):
   ```
   usergit ALL=NOPASSWD: /usr/bin/apt-get, /usr/bin/aptitude
   ```
4. Soubor uložte a zavřete.

Pokud již tedy máte vhodného uživatele, je potřeba nastavit v Git proměnné týkající se nasazení. Jsou to následující položky:

- DEPLOY_SSH_HOST – adresa repozitáře
- DEPLOY_SSH_ID – fingerprint
- DEPLOY_SSH_KEY – klíč
- DEPLOY_SSH_TARGET – adresa cíle ve formátu: <ip>@<username>

Nyní by už měl být systém připraven pro nasazení. Pro vyvolání nasazení je dle mého názoru nejjednodušší provést bezvýznamný commit a push, který následně vyvolá nasazení.

Bez využití průběžné integrace

Pokud nevyužíváte průběžné integrace, je potřeba nejdříve připravit (či vyčistit) databázi, proto ve složce se serverovou aplikací zavolejte:

```
node cleanDb.js
```

Následně můžete již spustit samotnou serverovou aplikaci. To provedete následujícím způsobem:

```
node index.js
```

4.3.4 NEPOVINNÉ – Zpřístupnění specifikace API

Pokud chcete vystavit specifikaci komunikačního API na serveru, postupujte následovně: Nejdříve vytvořte novou složku obsahující pouze soubor se specifikací ve formátu json, poté zavolejte ve složce následující příkaz:

```
nohup http-server --cors > /dev/null 2>&1 &
```

Tímto spustíte server zpřístupňující soubor na IP adrese serveru. Pro přístup k souboru jděte na níže uvedenou adresu s tím, že nezapomeňte v adrese nahradit „IP“ adresou serveru a „PORT“ použitým portem (výchozí je 8080).

```
http://petstore.swagger.io/?url=http://IP:PORT/dracekAPI.json#/
```

4.3.5 GUI pro monitoring

Pro instalaci nástroje NetData poskytujícího GUI jsem využil návod [26]. Pro běžnou instalaci by měl stačit příkaz:

```
bash <(curl -Ss https://my-netdata.io/kickstart.sh)
```

Příkaz automaticky stáhne instalační script. Příkaz můžete spouštět bez privilegovaných práv. Script o ně zažádá sám, pokud to bude potřeba. Během instalace budete několikrát vyzváni k potvrzení instalace jednotlivých komponent, všechny potvrďte. Pokud narazíte při instalaci na potíže, využijte alternativních způsobů instalace popsaných v [26].

Závěr

V závěru stručně shrnu obsah celé své práce a vysvětlím, proč v systému Dráček nebyl původní server nahrazen mnou vytvořeným novým serverem.

V kapitole Analýza jsem nejdříve rozebíral funkční a nefunkční požadavky, dále jsem pak analyzoval problémy v původní verzi komunikačního rozhraní. Nakonec jsem pak vybíral vhodné technologie pro serverovou aplikaci a pro databázi.

Po analýze jsem přistoupil k návrhu nové podoby serverové části. Jako první jsem popsal strukturu dat ukládaných v databázi. Poté jsem navrhl nové komunikační API, které jsem následně zdokumentoval pomocí nástroje Swagger [16]. Tato dokumentace je prvním větším výstupem mé práce a měla by sloužit dalším osobám, které budou pravděpodobně na projektu pokračovat po mně. Měla by být dostupná online na [**swagger_dracek**]. Následně jsem se zaobíral serverovou aplikací, kde jsem pro její vytvoření využil prototypu z práce [9]. Na názorném příkladu jedné operace jsem vysvětlil architekturu aplikace. V rámci toho jsem také řešil způsob autentifikace uživatelů. Zdrojové kódy serverové aplikace můžete najít v příloze. Pro serverovou část projektu Dráček jsem zavedl průběžnou integraci spolu s průběžným testováním. Řešil jsem problematiku GUI pro monitoring, kterou jsem nakonec vyřešil využitím nástroje NetData [20]. Toto GUI pro server Dráček IV by mělo být dostupné online na adrese: `http://185.88.73.25:19999/`. Co se týče GUI pro konfiguraci, došel jsem k názoru, že úkony spojené s administrací samotného stroje lze provádět v GUI systému Big Cloud [21], na kterém je umístěn server Dráčka IV. Dále jsem došel k názoru, že serverová aplikace nevyžaduje speciální konfiguraci a sama také provádí většinu konfigurace databáze. Pro případnou další potřebnou konfiguraci databáze lze využít nástroje Robo 3T [13] poskytujícího vhodné GUI.

V kapitole Implementace jsem zmínil několik problémů, které jsem při implementaci prototypu serverové aplikace řešil. V kapitole pak můžete také nalézt uživatelskou příručku popisující způsoby, jak lze na server poslat požadavek, zejména pak nástroj cURL [22]. Ke konci své práce jsem vytvořil

instalační příručku, která popisuje postup, jak systém nasadit na nový stroj.

Vytvořil jsem prototyp serverové aplikace. Nejedná se ovšem o plně dokončenou aplikaci (například chybí endpointy týkající se správy úkolů). Aplikaci jsem nestihl dokončit, důvodem byla potřeba přizpůsobovat převzatý prototyp [9], jenž jsem se rozhodl využít. Zmíněný prototyp totiž sloužil pro nestandartní API, což jsem zpočátku nevěděl. S tím byla spojená potřeba analyzovat stávající kód a provést úpravy, které bylo někdy složité zakomponovat do existující architektury. Zpětně bych tedy zhodnotil rozhodnutí využít zmíněného prototypu jako možná ne úplně vhodné. Ovšem v případě, že někdo bude na mé práci jednou pokračovat a dokončí serverovou aplikaci, se bude jednat o kvalitně vytvořený systém. Mohu tak prohlásit, že jsem položil základy pro solidní systém, a doufám, že nepřijde má práce vniveč. Jelikož v cílech mé práce bylo vytvořit prototyp, což jsem dle mého názoru splnil, mohu prohlásit cíle mé práce za splněné.

Bibliografie

1. FILIP, Ondřej. *Výuková aplikace Dráček II – Serverová část a rozhraní pro učitele*. 2016. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií.
2. KOVAŘOVIC, Karel. *Výuková aplikace Dráček II – gamifikace a personalizace*. 2016. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií.
3. ŠTĚPÁN, Jaroslav. *Zásuvné moduly aplikace Dráček III - výuka fyziky*. 2017. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií.
4. RYBA, Jaroslav. *Zásuvné moduly aplikace Dráček III - výuka matematiky*. 2017. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií.
5. SLABÝ, Ondřej. *Zásuvné moduly aplikace Dráček III - výuka základů algoritmizace*. 2017. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií.
6. PODANÝ, Pavel. *Animovaný avatar pro výukovou aplikaci Dráček*. 2016. Bakalářská práce. ČVUT v Praze, Fakulta informačních technologií.
7. MALÝ, Martin. REST: architektura pro webové API. *Zdroják, o tvorbě webových stránek a aplikací* [online]. 2009 [cit. 2018-04-28]. ISSN 1803-5620. Dostupné z: `https://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/`.
8. ROUSE, Margaret. RESTful API. *Microservices architecture information, news and tips* [online]. 2016 [cit. 2018-04-28]. Dostupné z: `https://searchmicroservices.techtarget.com/definition/RESTful-API`.
9. GREGOR, Petr. *BigCloud - backend pro veřejný cloudový systém*. 2015. Diplomová práce. ČVUT v Praze, Fakulta informačních technologií.

10. SCHWARTZ, Baron. JSON Support in PostgreSQL, MySQL, MongoDB, and SQL Server [online]. 2017 [cit. 2018-05-10]. Dostupné z: https://www.vividcortex.com/blog/2015/06/02/json-support-postgres-mysql-mongodb-sql-server/.
11. List of JSON Databases. *Quackit* [online] [cit. 2018-05-10]. Dostupné z: https://www.quackit.com/json/tutorial/list_of_json_databases.cfm.
12. *The JSON1 Extension* [software] [cit. 2018-05-10]. Dostupné z: https://www.sqlite.org/json1.html.
13. 3T SOFTWARE LABS. *Robo 3T* [software] [cit. 2018-05-10]. Dostupné z: https://robomongo.org/.
14. KOROTYA, Eugeniya. MongoDB vs MySQL Comparison: Which Database is Better? *Hacker Noon* [online]. 2017 [cit. 2018-04-28]. Dostupné z: https://hackernoon.com/mongodb-vs-mysql-comparison-which-database-is-better-e714b699c38b.
15. MONGODB, INC. MongoDB and MySQL Compared [online] [cit. 2018-04-28]. Dostupné z: https://www.mongodb.com/compare/mongodb-mysql.
16. SMARTBEAR SOFTWARE. *Swagger* [software] [cit. 2018-05-10]. Dostupné z: https://swagger.io/.
17. KAMENÍK, Martin. *Dráček IV specifikace komunikačního API* [online]. 2018 [cit. 2018-05-10]. Dostupné z: http://petstore.swagger.io/?url=http://185.88.73.25:8080/dracekAPI.json#/.
18. AUGUSTÝN, Michal. Průběžná integrace [online]. 2011 [cit. 2018-04-28]. Dostupné z: https://www.augi.cz/programovani/prubezna-integrace/.
19. ZABBIX LLC. *Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution* [software] [cit. 2018-05-10]. Dostupné z: https://www.zabbix.com/.
20. *NetData* [software] [cit. 2018-05-10]. Dostupné z: https://my-netdata.io/.
21. S.I.C. SPOL. S.R.O. *Big Cloud* [software] [cit. 2018-05-10]. Dostupné z: https://www2.bigcloud.cz/.
22. *cURL* [software]. 2018 [cit. 2018-05-07]. Dostupné z: https://curl.haxx.se/.
23. RAVAL, Harsh. Simple REST client in Java. *Java Code Geeks* [online]. 2012 [cit. 2018-05-06]. Dostupné z: https://www.javacodegeeks.com/2012/09/simple-rest-client-in-java.html.
24. *Installing Node.js via package manager* [online] [cit. 2018-05-07]. Dostupné z: https://nodejs.org/en/download/package-manager/.

25. *Install MongoDB Community Edition on Ubuntu* [online] [cit. 2018-05-07]. Dostupné z: `https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/`.

26. *Installation – firehol/netdata Wiki* [online] [cit. 2018-05-10]. Dostupné z: `https://github.com/firehol/netdata/wiki/Installation`.

Dodatek A

Seznam použitých zkratek

GUI Graphical user interface

JSON javaScript Object Notation

REST Representational State Transfer

API Application Programming Interface

URL Uniform Resource Locator

URI Uniform Resource Identifier

CRUD Create, read, update and delete

YAML Ain't Markup Language

Dodatek B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `APIspecification` ... zdrojové kódy dokumentace komunikačního API
- `text` ..... text práce včetně příloh
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `annexes` ..... přílohy práce