Zadání bakalářské práce

Název:	Software eKroužek
Student:	Vojtěch Bešťák
Vedoucí:	Ing. Filip Glazar
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

V rámci této práce proveďte návrh a částečnou realizaci systému pro komplexní správu zájmových činností. Mezi hlavní aspekty řešení se řadí jak samotná správa, tak evidence docházky na různé zájmové kroužky. Cílem práce je na základě analýzy a návrhu realizovat serverovou část systému a případně rozpracovat klientskou část. Zaměřte se především na vhodnou volbu technologií, ve které se budou při správném softwarovém návrhu jednotlivé komponenty systému dobře udržovat a dále rozvíjet. Ideálně postupujte dle následujících kroků:

1) Analyzujte danou doménu.
2) Specifikujte hlavní případy užití systému.
3) Proveďte volbu vhodných technologií např. PHP, Python, PostgreSQL, REST API.
4) Navrhněte a implementujte serverovou část systému.
5) Zhodnoťte výsledek předchozích kroků a navrhněte případné úpravy či vylepšení systému.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 6. února 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Software eKroužek

Vojtěch Bešťák

Katedra softwarového inženýrství
Vedoucí práce: Ing. Filip Glazar

11. května 2023

Poděkování

Rád bych poděkoval především vedoucímu mojí práce Ing. Filipovi Glazarovi, který mě podpořil ve výběru tohoto tématu a v průběhu tvorby mi vždy poskytl potřebné rady a svůj čas. Dále bych rád poděkoval Matouši Ettlerovi za to, že mě k tomuto nápadu vůbec přivedl a vždy mi s čímkoliv poradil. Děkuji také Markovi Erbenovi za rady ohledně praktické části této práce. Velký dík patří také Ondřejovi Černému a Petrovi Vocílkovi za skvělou spolupráci při sběru požadavků pro aplikaci. V neposlední řadě bych chtěl poděkovat své rodině a nejbližším přátelům za to, že mě v průběhu studia vždy podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při 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. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na jejímž základě se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

Praha dne 11. května 2023 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Vojtěch Bešťák. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Bešťák, Vojtěch. *Software eKroužek*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Většina lektorů volnočasových aktivit na školách musí v současnosti přihlášky a docházku vyřizovat manuálně. Tato práce se proto zabývá návrhem a realizací serverové aplikace umožňující komplexní správu zájmových činností, která by tuto administrativu ulehčila. Analýza existujících řešení prokazuje, že podobná řešení na trhu neexistují. Na základě podnětů od lektorů školních kroužků jsou definovány funkční a nefunkční požadavky a jsou popsány hlavní případy užití. Dále je provedena volba vhodných technologií, návrh databázového schématu a aplikačního rozhraní. Serverová aplikace je realizována jako REST API prostřednictvím frameworku Symfony v programovacím jazyce PHP. Zdokumentované aplikační rozhraní je následně vhodně otestováno. Výsledná aplikace je připravená k nasazení.

Klíčová slova webová aplikace, backend, školní kroužky, API, Symfony

Abstract

Currently, most lectors of extra-curricular activites in schools have to deal with applications and attendance manually. Therefore, this thesis deals with

the design and implementation of a backend application that enables comprehensive administration of leisure activities to facilitate this administration. An analysis of existing solutions shows that there are no similar solutions on the market. Based on suggestions from school club instructors, functional and non-functional requirements are defined and the main use cases are described. Furthermore, the selection of appropriate technologies, the design of the database schema and the application interface are made. The server application is implemented as a REST API through the Symfony framework in the PHP programming language. The documented application interface is then appropriately tested. The resulting application is ready for deployment.

Keywords web application, backend, school extra-curricular activities, API, Symfony

Obsah

Úvod **1**

Cíle práce **3**

1 Analýza **5**
1.1 Současný stav . . . 5
1.2 Existující řešení . . . 5
1.2.1 Systémy pro sportovní kluby . . . 5
1.2.2 Systémy pro kroužky . . . 6
1.2.3 Systémy pro školy . . . 6
1.3 Analýza požadavků . . . 6
1.3.1 Funkční požadavky . . . 7
1.3.2 Nefunkční požadavky . . . 10
1.4 Analýza případů užití . . . 10
1.4.1 Aktéři . . . 11
1.4.2 Přihlášení a registrace . . . 11
1.4.3 Správa školy . . . 13
1.4.4 Správa pololetí . . . 15
1.4.5 Správa kroužků . . . 17
1.4.6 Správa rozvrhů . . . 17
1.4.7 Zapisování docházky . . . 20
1.4.8 Správa přihlášek . . . 23
1.4.9 Správa uživatelů . . . 26
1.4.10 Export dat . . . 27
1.5 Pokrytí funkčních požadavků . . . 28

2 Návrh **31**
2.1 Aplikační rozhraní . . . 31
2.1.1 Representational state transfer . . . 31

2.1.2 Simple object access protocol 33
2.1.3 GraphQL 33
2.1.4 Shrnutí 33
2.2 Návrh endpointů 34
2.2.1 Správa přihlášek 34
2.3 Architektura 39
2.3.1 Více-vrstvá architektura 39
2.3.2 Architektura prezentační vrstvy 40
2.4 Využité technologie 41
2.4.1 Framework 41
2.4.2 Databáze 45
2.4.3 Objektově relační mapování 46
2.4.4 Docker 48
2.5 Návrh databáze 49
2.5.1 Entity–Relationship model 49
2.5.2 Návrh ER modelu 50

3 Realizace **53**
3.1 Použité nástroje pro vývoj 53
3.1.1 Vývojové prostředí 53
3.1.2 Git a Gitlab 53
3.1.3 Toggl Track 55
3.2 Konfigurace prostředí 56
3.2.1 Instalace Symfony 56
3.2.2 Docker compose 57
3.2.3 Composer 58
3.3 Implementace 59
3.3.1 Entity 59
3.3.2 Služby 60
3.3.3 Controller 61
3.3.4 Formuláře 61
3.3.5 Odchytávání chyb 63
3.3.6 Stránkování, filtrování a řazení 64
3.4 Dokumentace 65
3.4.1 Automatická generace dokumentace 67
3.5 Testování 67
3.5.1 Statické a dynamické testování 68
3.5.2 Metody 69
3.5.3 Typy testů 69
3.5.4 Typy API testování 69
3.5.5 Implementace testů 70
3.6 Výsledky 71

Diskuze **73**

Závěr **75**

Bibliografie **77**

A Kompletní ER model **83**

B API dokumentace **85**

C Existující klientské aplikace **91**

D Seznam zkratek **93**

Obsah přiloženého média **95**

Seznam obrázků

1.1 Případy užití pro FP1 . . . 14
1.2 Případy užití pro FP2 . . . 15
1.3 Případy užití pro FP3 . . . 18
1.4 Případy užití pro FP5 . . . 20
1.5 Případy užití pro FP6 . . . 24
1.6 Případy užití pro FP7 . . . 27

2.1 Vztah frameworku, knihovny a zdrojového kódu . . . 42
2.2 ER model zúžený na přihlášky . . . 51

3.1 Úkoly nahlášené do systému GitLab . . . 54
3.2 Úspěšně provedená pipeline ve službě GitLab CI . . . 55
3.3 Kalendář verzí frameworku Symfony . . . 56
3.4 Dokumentace vizualizovaná pomocí SwaggerUI . . . 68

A.1 Kompletní ER model – část 1. . . . 83
A.2 Kompletní ER model – část 2. . . . 84

B.1 API dokumentace – část 1. . . . 85
B.2 API dokumentace – část 2. . . . 86
B.3 API dokumentace – část 3. . . . 87
B.4 API dokumentace – část 4. . . . 88
B.5 API dokumentace – část 5. . . . 89

C.1 Obrazovka z klientské webové aplikace . . . 91
C.2 Obrazovky z mobilní aplikace pro platformu Android . . . 92
C.3 Mobilní aplikace pro platformu Android umístěná na Google Play 92

Seznam tabulek

1.1 Pokrytí funkčních požadavků případy užití 29

2.1 Popis koncového bodu podání přihlášky 34
2.2 Popis koncového bodu zobrazení přihlášky 36
2.3 Popis koncového bodu zobrazení přihlášek školy 37
2.4 Popis koncového bodu úpravy přihlášky 38
2.5 Popis koncového bodu smazání přihlášky 39

Seznam ukázek zdrojového kódu

2.1 Tělo požadavku podání přihlášky 35
2.2 Tělo úspěšné odpovědi podání přihlášky 35
2.3 Tělo chybné odpovědi podání přihlášky 36
2.4 Tělo úspěšné odpovědi zobrazení přihlášky 37
2.5 Tělo úspěšné odpovědi zobrazení přihlášek školy 38
2.6 Tělo požadavku úpravy přihlášky 38
2.7 Entita v knihovně Doctrine 47
2.8 Příklad dotazu v DQL 48
3.1 Instalace frameworku Symfony 57
3.2 Konfigurace služby pro aplikaci v souboru docker-compose.yml 58
3.3 Doménový model přihlášky implementovaný pomocí Doctrine . 59
3.4 Služba shromažďující logiku týkající se přihlášek 60
3.5 Controller obsahující koncový bod pro podání přihlášky 62
3.6 Formulář pro vytvoření přihlášky 63
3.7 Datový objekt pro vytvoření přihlášky 64
3.8 Implementace stránkovaného, filtrovaného a řazeného koncového bodu 66
3.9 Specifikace dokumentace endpointu ve zdrojovém kódu 67
3.10 Konfigurace modelu přihlášky v knihovně NelmioApiDocBundle 68
3.11 Test koncového bodu v Codeception 71

Úvod

V době, kdy jsou děti čím dál více obklopeny technologiemi, je potřeba, aby si někde mohly odpočinout. Naštěstí existují lidé, kteří pro děti ve školním a předškolním věku pořádají kroužky a volnočasové aktivity. Ačkoliv by se mohlo zdát, že samotné vedení hodin je jediná část jejich pracovní náplně, administrativa spojená s přihláškami, docházkou a osobními údaji rodičů a dětí tuto aktivitu časově velmi předčí.

S nápadem na usnadnění této administrativy mě oslovil můj bývalý spolužák z gymnázia Matouš Ettler. Výsledkem měla být aplikace pro mobilní zařízení pro platformu Android, ze které by lektoři mohli zapisovat docházku. S webovými technologiemi příliš ještě neseznámen, jsem tuto velmi naivní aplikaci vytvořil. Po nástupu na Fakultu informačních technologií Českého vysokého učení technického jsem ihned začal přemýšlet, jakým způsobem by se dal projekt dále rozvinout. Bylo jasné, že aby se projekt mohl do budoucna rozrůstat, bude potřeba kompletně přepsat serverovou část aplikace.

Práce se zaměřuje na návrh a implementaci uceleného serverového systému pro správu přihlášek, kroužků a zápisu docházky. Výsledná aplikace by měla ulehčit lektorům administrativu a rodičům přihlašovací proces.

Nejdříve se budu zabývat sběrem a analýzou požadavků a případů užití. V této části nejdříve teoreticky rozeberu typy požadavků. Následně specifikuji požadavky a případy užití, které jsem během analýzy získal od učitelů a lektorů kroužků.

Ve druhé části představím databázové schéma a popíši do detailu důvody existence vybraných entit. Součástí bude i návrh několika koncových bodů API rozhraní, které bude aplikace poskytovat navenek. Zvolím zde také technologie, které pro vývoj využiji, a architekturu, která bude pro aplikaci optimální.

Třetí část se bude zaměřovat na samotnou realizaci. Použiji zde výstupy z předchozích částí a provedu implementaci celého projektu. Výsledkem by měla být funkční aplikace, která splňuje specifikované požadavky.

Na závěr popíši způsob, jakým bude aplikace testována, a představím fi-

nální výstupy, které vzešly z implementační části. Zmíním také místa v aplikaci, kde by bylo možné pokračovat ve vývoji, a funkcionality, které by bylo vhodné přidat.

Po dokončení tohoto projektu bude možné aplikaci nasadit do produkčního prostředí a zpřístupnit uživatelům.

Cíle práce

Hlavním cílem této práce je návrh a implementace systému umožňující správu zájmových činností na školách. Pro jeho dosažení je třeba definovat následující dílčí úkoly.

Prvním cílem je provedení průzkumu již existujících řešení v oblasti správy zájmových činností na školách. Následně je třeba analyzovat požadavky kladené na aplikaci učiteli a specifikovat hlavní případy užití. Navazujícím cílem je návrh architektury aplikace a vhodné zvolení použitých technologií. Po zpracování předchozích oblastí je třeba realizovat serverovou část systému. Na závěr je potřeba zhodnotit výsledky analýzy, návrhu a realizace.

Výsledky této práce budou po nasazení přínosné především pro školy a jejich pedagogy, neboť existence tohoto systému zjednoduší administrativu spojenou s pořádáním zájmových činností. To se bude týkat jak učitelů, kteří by jinak mohli být odrazeni, tak rodičů, kterým se zjednoduší proces přihlašování dětí.

Kapitola 1

Analýza

1.1 Současný stav

Organizací nebo škol, které pořádají v určité formě kroužky, je mnoho. Přihlášky se v nich pak řeší různě. Někteří přijímají přihlášky v papírové podobě, jiní používají alespoň nějakou elektronickou formu jako například elektronický formulář. Co mají tyto postupy většinou společného je, že po shromáždění přihlášek jsou často informace o dětech, rodičích a stavu plateb uloženy například ve formátu XLSX a dále zpracovávány v kancelářském nástroji Excel. Zapisování docházky pak probíhá formou klasického záznamu do třídní knihy nebo se neřeší vůbec.

Tento postup vede k tomu, že ti učitelé, kteří jsou zodpovědní za přihlášky na kroužky, netráví svůj čas na začátku školního pololetí ničím jiným než přijímáním plateb a kontrolou stavu přihlášek. Pokud docházka zapisována je, probíhá tak do fyzických dokumentů. Pokud tedy rodiče chtějí vědět, zda musí něco zaplatit nebo zda jejich dítě na kroužky chodí, musí se většinou dostavit osobně na některý z kroužků nebo navštívit učitele ve škole, kde ho ani nemusí zastihnout.

1.2 Existující řešení

1.2.1 Systémy pro sportovní kluby

Sportovních klubů existuje veliké množství [1]. Tuto poptávku zaznamenalo stejně tak veliké množství firem, a proto systémů pro správu těchto klubů z pohledu přihlášek, členských příspěvků a docházky existuje již celá řada. Jako příklad lze uvést dvě aplikace:

- *Sportes*,
- *SportsCoach*.

Obě tyto aplikace jsou zaměřeny primárně na sportovní kluby, a proto jsou tomu všechny funkcionality těchto systémů přizpůsobeny. Škola v nich žádným způsobem nefiguruje [2, 3].

1.2.2 Systémy pro kroužky

Dohledatelné systémy zaměřené na kroužky fungují pouze na úrovni databáze existujících kroužků. Mezi existujícími aplikacemi jsou:

- *Databáze táborů a kroužků*,
- *Kroužky & tábory*.

Lze se na nich zaregistrovat jako pořadatel, definovat si vlastní kroužky a následně na ně sbírat prostřednictvím těchto systémů přihlášky. Potenciální rodič si pak může v seznamu všech existujících kroužků vybrat libovolný a na ten svoje dítě přihlásit. Zaměření na docházku je zde až sekundární [4, 5].

1.2.3 Systémy pro školy

V Česku jsou nejvyužívanějšími školními informačními systémy Bakaláři a Škola OnLine [6]. Oba poskytují v určitě míře také správu kroužků.

Bakaláři ji řeší pomocí podvýběru třídní knihy – tedy vytvoření skupiny žáků, pro kterou lze evidovat docházku. Tento způsob nicméně neumožňuje přihlašování žáků rodiči [7]. Školy, které tento informační systém používají, tedy musejí být kreativní a použít modul „Ankety“ pro vytvoření přihlášky na kroužek. Jde pouze o formulář a získaná data je pak potřeba pro další použití manuálně zpracovat. Platby za zvolené kroužky je potřeba řešit také mimo systém [8].

Škola OnLine doporučuje vytvoření školních akcí, u kterých lze omezit termíny přihlašování [9].

Ani jeden z těchto školních informačních systémů v době psaní této práce (leden–květen 2023) nemá modul, který by byl plnohodnotně zaměřen na volnočasové aktivity a kroužky. Naopak obsahují plno funkcí, které škola někdy ani nepotřebuje. Žádná z aplikací zmíněných v předchozích částech dokonce školu jako entitu vůbec neobsahuje. Potvrzuje to tedy výše definovaný cíl práce, že systém eKroužek by měl cílit na školní zařízení, a to konkrétně buď na školy, nebo na školky.

1.3 Analýza požadavků

Jedna z prvních aktivit softwarového procesu je analýza a sběr požadavků. Specifický obor, který se tímto zabývá, se nazývá inženýrství požadavků (v originále Requirements Engineering). Jde o odvětví vývoje softwaru, které zkoumá cíle z reálného světa a snaží se je propojit s výsledným systémem. Výsledkem

tohoto procesu pak je precizní specifikace požadavků, která slouží jako základní dokumentace [10].

Požadavky lze členit na funkční a nefunkční. Funkční požadavky popisují, jak se systém má chovat a co se má implementovat, nicméně současně vymezují i to, co by systém dělat neměl. Nefunkční požadavky naopak nahlížejí na ostatní atributy systému, které nelze jako celek nazvat samostatnou funkcí. Zaměřují se na oblast kvality (bezpečnost, spolehlivost, výkon atd.) nebo určují implementační standardy [10].

Požadavky by měly splňovat základní kritéria, která jsou definována normou *IEEE 830*. Mezi ně patří mj. jednoznačnost, ověřitelnost a trasovatelnost. Tyto vlastnosti zaručují dobře specifikované požadavky a umožňují je použít jako dokumentaci pro další vývoj [11].

Nabízí se otázka, proč tedy psát požadavky a proč se snažit, aby splňovaly výše zmíněné vlastnosti? Sběr a analýza požadavků probíhá na samém začátku procesu tvorby systému. Všechny ostatní části tohto procesu z těchto požadavků budou tedy nutně vycházet. Oprava chyby, která byla detekována až v průběhu vývoje, bude tedy mnohem nákladnější, než kdyby byl proces specifikace požadavků důkladnější [12].

1.3.1 Funkční požadavky

Namísto rozsáhlého textu popisující detaily každého požadavku bude vhodnější charakterizovat každý požadavek několika základními atributy. Později tyto vlastnosti mohou sloužit k jednoduššímu vyhledávání a filtrování [13]. Budou tedy sledovány atributy: název, popis, typ, složitost a priorita požadavku. Atributů existuje větší množství, nicméně zde nebyly použity všechny, protože v rámci této analýzy by přinesly pouze drobné výhody.

Sběr požadavků byl uskutečněn prostřednictvím několika rozhovorů s pedagogy a lektory kroužků na školách. Konkrétně jde o Ondřeje Černého, který pořádá volnočasové aktivity pro pražské školy ZŠ Lyčkovo náměstí, ZŠ Pernerova a ZŠ Petra Strozziho. Druhým pedagogem je Petr Vocílka ze střední školy Gymnázium, Příbram, Legionářů 402. Oba mají společné to, že by rádi poskytli žákům dostupné kroužky. Na základě jejich nápadů a jejich zpětné vazby byly sestaveny následující funkční a nefunkční požadavky.

1.3.1.1 FP1: Přihlášení

Uživatel se může přihlásit do systému pomocí svého e-mailu a svého hesla. Pokud jsou tyto údaje nesprávné, bude na to upozorněn. Nepamatuje-li si uživatel svoje heslo, může si ho obnovit.

Nový uživatel se může do systému zaregistrovat pouze jako administrátor. Registrace dalších jednotlivých uživatelů nebude podporována.

Složitost: vysoká

Priorita: vysoká

1.3.1.2 FP2: Správa školy

Po založení administrátorského účtu si může uživatel vytvořit svoji školu. Stává se tím automaticky administrátorem této školy. O škole musí uvést základní informace, které jsou později prezentovány rodičům.

Ke každé škole se zároveň vážou lokace, na kterých se mohou kroužky konat. Tato místa musí být definovatelná administrátorem.

Pro používání systému musí administrátor vybrat jeden z definovaných tarifů. Tyto tarify budou rozděleny podle počtu dětí, které se mohou na kroužky přihlásit.

Složitost: střední

Priorita: střední

1.3.1.3 FP3: Správa pololetí

Vzhledem k tomu, že každý školní rok mohou být ve škole různé kroužky, musí celý systém stát na definovaných časových intervalech. V tomto případě to budou pololetí. Systém musí umožnit spravovat tato pololetí, na která se budou později navazovat další objekty (kroužky, přihlášky, prázdniny, ...)

Složitost: střední

Priorita: vysoká

1.3.1.4 FP4: Správa kroužků

Systém musí umožňovat spravovat kroužky. Kroužky budou sloužit pouze jako kategorie pro jednotlivé rozvrhy. Budou tedy evidovány pouze základní informace.

Složitost: nízká

Priorita: vysoká

1.3.1.5 FP5: Správa rozvrhů a jejich termínů

Ke každému kroužku musí být možné definovat jednotlivé rozvrhy. Rozvrh je souhrn termínů, které musí být možné k danému rozvrhu specifikovat. Přihláška bude podávána na jednotlivé rozvrhy. Přihlášení na rozvrh znamená přihlášení na všechny přiřazené termíny.

Složitost: nízká

Priorita: střední

1.3.1.6 FP6: Zapisování docházky

Pro definované termíny rozvrhů musí být možné zapsat každý týden docházku. Docházka bude vytvářena pro všechny děti, které na daný rozvrh docházejí. Výstupem pro daný termín bude zapsaná lekce. Lekce bude obsahovat kromě docházky také poznámky, obsah lekce a informaci o tom, kdo zápis provedl.

Zapisování docházky budou provádět uživatelé, kteří mají přiřazenou roli učitel.

Složitost: střední

Priorita: vysoká

1.3.1.7 FP7: Správa přihlášek

Systém musí umožnit podat přihlášku na dané pololetí. Tato přihláška bude obsahovat základní informace o dítěti, rozvrhy, na které se chce dítě přihlásit, a kontaktní informace na jednoho nebo oba rodiče.

Pro rodiče podáním přihlášky vzniknou v systému účty (pokud již neexistují). Zaplatit přihlášku budou moci ihned prostřednictvím platební brány nebo až po přihlášení do účtu.

Složitost: vysoká

Priorita: vysoká

1.3.1.8 FP8: Správa uživatelů

Administrátor školy bude potřebovat mít přehled o všech uživatelích, kteří se v jeho systému vyskytují. Důležitou součástí je možnost přidávat nové učitele a přiřazovat jim kroužky, konkrétně rozvrhy, které budou spravovat.

Učitelé by následně měli mít možnost v rámci lekce jednoduše kontaktovat rodiče daného dítěte.

Složitost: střední

Priorita: střední

1.3.1.9 FP9: Export dat

Systém by měl umožňovat snadno exportovat téměř jakákoliv data. Ať už jde o přihlášené žáky na jednotlivé kroužky, nebo o evidované učitele. Výstup by měl být dostupný ve formátech CSV či XLS.

V ideálním případě budou dostupné i možnosti exportovat data přímo pro konkrétní entitu – jako například ministerstvo školství. Některé školy nebo kluby budou potřebovat podat žádost o dotaci a k ní musí přiložit data o žácích v určitém definovaném formátu.

Složitost: vysoká

Priorita: nízká

1.3.2 Nefunkční požadavky

1.3.2.1 NP1: Autentizace

Autentizace musí využívat zavedený standard oAuth2 [14]. Pro realizaci by bylo vhodné využít již existující řešení poskytované třetí stranou. Zabezpečení je komplexní problém, ve kterém se dá velice snadno udělat chyba s ohromnými následky. Byl již řešen mnohokrát. Nedává tedy smysl vyvíjet vlastní řešení, když lze použít již implementovaný, otestovaný a funkční nástroj.

1.3.2.2 NP2: Platby

Platby uvnitř aplikace musí probíhat skrze platební bránu třetí strany. Varianta napojení přímo na bankovní účet uživatelů je již zastaralá a může působit nezabezpečeně. Použitím existující platební brány lze opět ušetřit čas, který by byl jinak stráven implementací vlastní. Protože se jedná o manipulaci s penězi, bylo by lepší, kdyby nedošlo v tomto místě k implementační chybě. Proto je platební brána pro tento projekt vhodnější volba.

1.3.2.3 NP3: Dostupné API rozhraní

Systém by měl poskytovat jasně definované API rozhraní. Výhodou tohoto přístupu oproti vytváření jedné monolitové aplikace je modularita. Místo jedné mohutné aplikace, která by uměla vše, bude možné rozdělit byznysovou logiku a způsob zobrazení na jednotlivé menší aplikace, která každá bude zaměřena pouze na jednu část problému. Po dokončení serverové části se na definované API budou moci jednoduše napojit nové klientské aplikace. Ty budou vyvíjeny samostatně a nezávisle na této serverové. Detailnější popis této problematiky je rozebrán v podkapitole 2.1.

1.4 Analýza případů užití

Potíž s požadavky bez dalšího kontextu je ten, že nepopisují systém příliš detailně. Většinou se totiž u jednoho požadavku jedná o množinu několika situací a nelze z něj vyčíst konkrétní situace.

Tento problém se (alespoň částečně) snaží řešit případy užití. Na rozdíl od obecných požadavků jdou více do hloubky. Umožňují popisovat různé situace za různých podmínek, které systém vykonává pro určité aktéry [15].

1.4.1 Aktéři

Aktérem je taková entita, která interaguje se systémem. Případy užití jsou iniciovány právě těmito aktéry. Proto každý z těchto případů začíná akcí právě daného aktéra. Nemusí jít ale nutně o osobu. Většinou se aktéři definují jako role, ale může se jednat i o neživé věci jako systém nebo čas [16]. V případech užití se pracuje s následujícími aktéry.

1.4.1.1 Uživatel

Obecným uživatelem je jakýkoliv uživatel používající tento systém. Nepotřebuje být ke svým úkonům nijak autorizován a může být ztotožněn kterýmkoliv z ostatních aktérů.

1.4.1.2 Administrátor

Administrátor spravuje školu a školní pololetí. Definuje kroužky, rozvrhy a jejich termíny. Je to osoba, která je zodpovědná za proces přihlašování pro danou školu. Administrátorem se uživatel stane, když se zaregistruje do systému s úmyslem poskytovat kroužky.

1.4.1.3 Učitel

Hlavní role učitele je vyplňování docházky na kroužky, které spravuje. Může zapisovat informace o dané lekci (například co se v hodině dělo, co se má dělat příště), zaznamenat přítomnost či nepřítomnost žáků nebo v případě nutnosti kontaktovat některého z rodičů. Učitelem se může stát pouze uživatel určený administrátorem.

1.4.1.4 Rodič

Rodič především přihlašuje svoje děti na kroužky. Rodičem se stává uživatel po podání alespoň jedné přihlášky. Následně může u svých dětí sledovat stav přihlášek, jejich docházku nebo na jaké přišli lekce.

1.4.1.5 Root administrátor

Hlavní administrátor má přístup ke všem informacím. Může spravovat všechny školy nebo se přihlásit za libovolného jiného uživatele. V případě nutnosti (například problém s platbou) může škole zamezit přístup k celému systému. Tento aktér ztotožňuje správce celého systému.

1.4.2 Přihlášení a registrace

Diagram na obrázku 1.1 zobrazuje případy užití, které se týkají funkčního požadavku FP1 popsaném v podkapitole 1.3.1.1.

1.4.2.1 UC1-1: Přihlášení

Umožňuje uživateli přihlásit se do systému pomocí jeho e-mailu a hesla. Pokud je kombinace údajů nesprávná, je o tom uživatel informován. Pokud přihlášení proběhne bez problémů, je uživatel přesměrován na domovskou stránku aplikace.

Aktéři: všichni

1.4.2.2 UC1-2: Registrace administrátora

Škola nebo osoba, která chce získat přístup k systému eKroužek, se musí zaregistrovat. Po vyplnění základních osobních údajů je uživateli vytvořen účet a automaticky je mu přiřazena role administrátor. Je potřeba provést validaci e-mailu pomocí aktivačního odkazu.

Aktéři: administrátor

Hlavní scénář: Registrace je úspěšná

1. Škola nebo hlavní trenér se chce registrovat.
2. Systém zobrazí registrační formulář.
3. Uživatel vyplní osobní údaje a heslo.
4. Systém údaje zvaliduje, uživatele zaregistruje a přiřadí mu roli administrátor. Následně odešle uživateli e-mail, který obsahuje aktivační odkaz k účtu.
5. Uživatel klikne v e-mailu na aktivační odkaz.
6. Systém aktivuje uživatelův účet.

Alternativní scénář: Duplicitní e-mail

1. Scénář začíná ve 3. kroku hlavního scénáře, pokud se nepodaří ověřit unikátnost e-mailu.
2. Systém informuje uživatele o duplicitě e-mailu.
3. Uživatel opraví e-mail tak, aby již nenastala duplicita.
4. Scénář pokračuje 4. krokem hlavního scénáře.

Alternativní scénář: Nezdařilá aktivace

1. Scénář začíná ve 4. kroku hlavního scénáře, pokud aktivační odkaz již nebude platný.
2. Systém informuje uživatele o neplatnosti aktivačního odkazu.
3. Uživatel se pokusí přihlásit.
4. Systém přihlášení odmítne a nabídne možnost odeslat nový e-mail s aktivačním odkazem na e-mailovou adresu uživatele.
5. Uživatel si vyžádá nový aktivační e-mail.
6. Systém odešle e-mail s novým aktivačním odkazem.
7. Scénář pokračuje 5. krokem hlavního scénáře.

1.4.2.3 UC1-3: Obnova hesla

Umožňuje uživateli obnovit zapomenuté heslo ke svému účtu. Uživatel po zadání svého e-mailu obdrží odkaz, na kterém si může zadat nové heslo. Tento odkaz má pouze omezenou platnost. Po obnově je uživatel o změně hesla upozorněn e-mailem. Pokud uživatel se zadaným e-mailem neexistuje, není žádný e-mail odeslán.

Aktéři: všichni

1.4.3 Správa školy

Diagram na obrázku 1.2 zobrazuje případy užití, které se týkají funkčního požadavku FP2 popsaném v podkapitole 1.3.1.2.

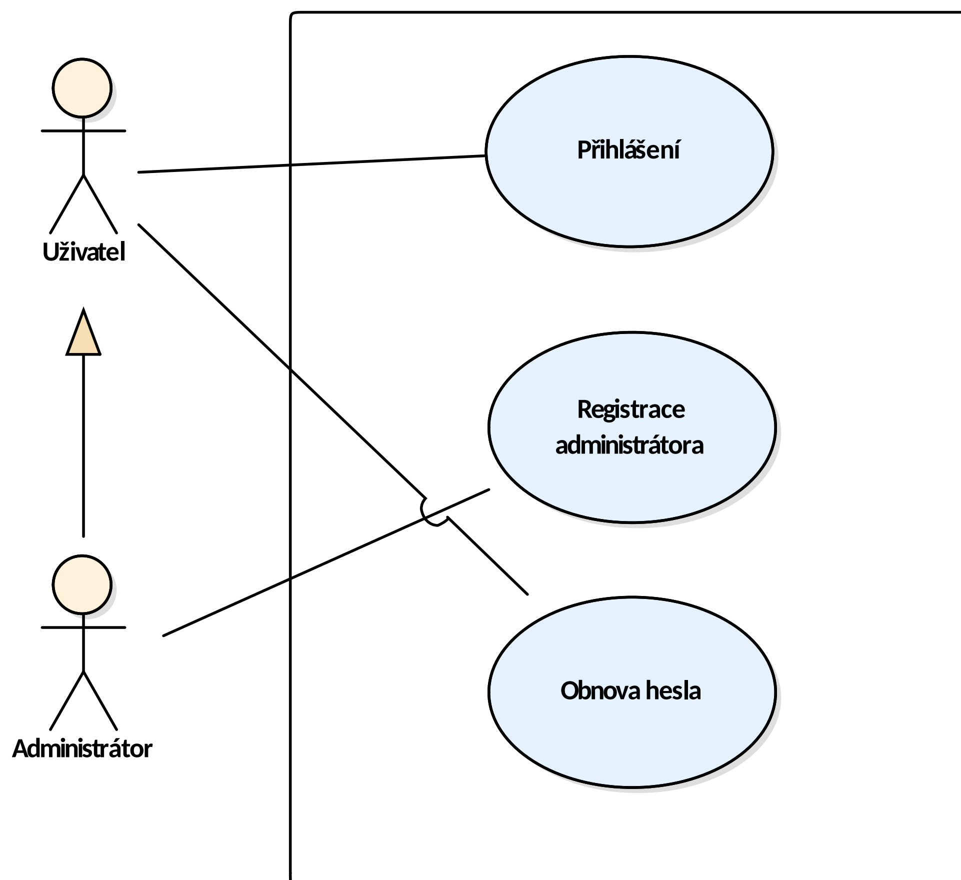
1.4.3.1 UC2-1: Vytvoření školy

Administrátor po prvním přihlášení do systému musí definovat svoji školu. O škole musí vyplnit základní údaje, které budou později k dispozici rodičům při přihlašování dětí.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor poprvé přihlásí do systému.
2. Systém zobrazí uživateli formulář pro vytvoření školy umožňující zadat: jméno, e-mail školy, adresu, bankovní účet pro příchozí platby a další.



Obrázek 1.1: Případy užití pro FP1

3. Uživatel tyto údaje vyplní.
4. Systém údaje zvaliduje, ověří unikátnost školního e-mailu a školu vytvoří.

1.4.3.2 UC2-2: Zobrazení informací o škole

Kromě administrátora si informace o škole mohou zobrazit i rodiče přihlašující svoje děti na kroužky. U vytváření přihlášky tedy uvidí nejen jméno školy, ale i kontaktní údaje a adresu.

Aktéři: všichni

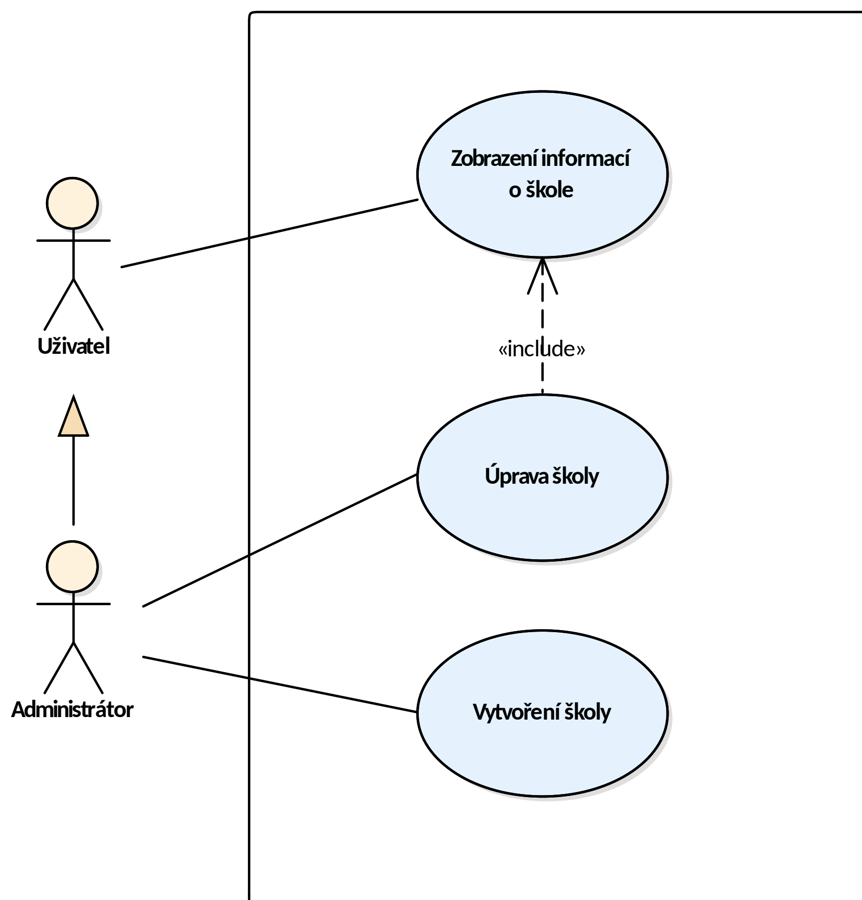
1.4.3.3 UC2-3: Úprava školy

Systém umožní administrátorovi upravit všechny údaje o škole, které při jejím vytvoření zadal.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor rozhodne změnit údaje o škole.
2. Include (zobrazení informací o škole).
3. Systém zkontroluje validitu upravených údajů, ověří unikátnost školního e-mailu a údaje o škole upraví.



Obrázek 1.2: Případy užití pro FP2

1.4.4 Správa pololetí

Diagram na obrázku 1.3 zobrazuje případy užití, které se týkají funkčního požadavku FP3 popsaném v podkapitole 1.3.1.3.

1.4.4.1 UC3-1: Vytvoření pololetí

Umožňuje administrátorovi vytvořit v jeho škole nové časové období. U tohoto období musí uvést počátek a konec.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor rozhodne definovat nové pololetí.
2. Systém zobrazí uživateli formulář pro vytvoření pololetí umožňující zadat název pololetí a následně i datum začátku a datum konce tohoto období. Dále pak datum a čas, kdy se otevírá přihlášování pro toto pololetí, a volitelně i datum a čas, kdy se uzavírá.
3. Uživatel tyto údaje vyplní.
4. Systém zkontroluje, zda zadané rozsahy dat jsou validní a pololetí vytvoří.

1.4.4.2 UC3-2: Zobrazení seznamu pololetí

Před akcemi s jednotlivými pololetími si jedno z nich musí administrátor zvolit. Zobrazí se mu tedy seznam všech pololetí.

Aktéři: administrátor

Hlavní scénář

1. Systém zobrazí seznam pololetí.
2. Uživatel si jedno z pololetí vybere.

1.4.4.3 UC3-3: Úprava pololetí

Systém umožní administrátorovi měnit informace o definovaném pololetí.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor rozhodne změnit údaje o pololetí.
2. Include (zobrazení informací o pololetí).
3. Uživatel zvolí možnost pololetí upravit.

4. Systém zobrazí formulář s předvyplněnými aktuálními informacemi o vybraném pololetí.
5. Uživatel upraví žádané informace.
6. Systém zvaliduje datumové rozsahy a upravené údaje uloží.

1.4.4.4 UC3-4: Smazání pololetí

Pokud si administrátor nepřeje mít definované dané pololetí nebo se při vytváření spletl, systém umožní dané pololetí smazat.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor rozhodne smazat pololetí.
2. Include (zobrazení informací o pololetí).
3. Uživatel zvolí možnost pololetí smazat.
4. Systém požádá uživatele o potvrzení smazání.
5. Uživatel potvrdí smazání vybraného pololetí.
6. Systém zkontroluje, zda k tomuto pololetí již nejsou přiřazeny další objekty (jako například sporty, přihlášky atd.), a pololetí smaže.

1.4.5 Správa kroužků

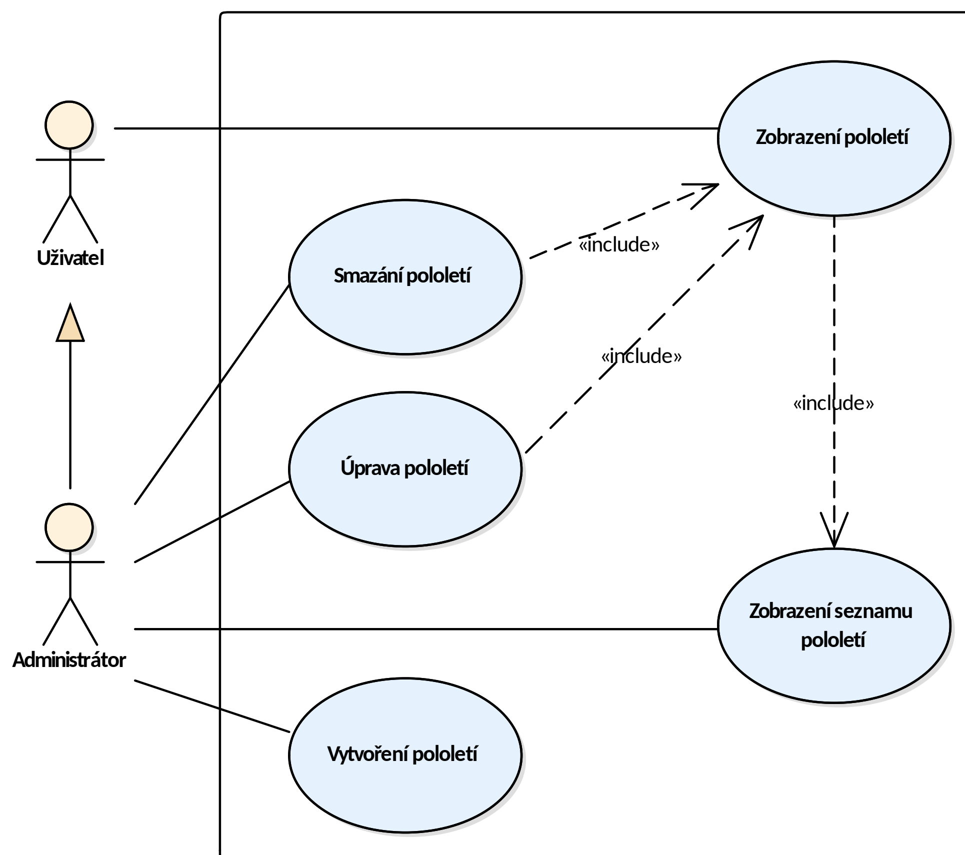
V rámci celé školy mohou být definovány kroužky. Kroužek lze připodobnit v univerzitním prostředí předmětu.

Případy užití by zde byly: vytvoření kroužku, zobrazení kroužku, upravení kroužku a smazání kroužku. Jde tedy pouze o CRUD operace (více v podkapitole 2.1.1). Detailní popis těchto případů je zde proto vynechán z důvodu jejich jednoduchosti.

1.4.6 Správa rozvrhů

Pro každý kroužek musí být možné přidat několik různých rozvrhů. Rozvrh je podobný paralelce v univerzitním prostředí. Kroužek může mít více těchto rozvrhů, stejně jako předmět může mít více paralelek. Po přihlášení žáka na daný rozvrh jsou automaticky přihlášeni na jednotlivé termíny.

Diagram na obrázku 1.4 zobrazuje případy užití, které se týkají funkčního požadavku FP5 popsaném v podkapitole 1.3.1.5.



Obrázek 1.3: Případy užití pro FP3

1.4.6.1 UC5-1: Vytvoření rozvrhu a termínů

Po definici jednotlivých kroužků je možné vytvořit pro daný kroužek konkrétní rozvrh. O každém rozvrhu musí uživatel vyplnit název, popis, cenu, celkovou kapacitu, datum počátku a konce.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor rozhodne definovat nový rozvrh.
2. Systém zobrazí uživateli formulář pro vytvoření rozvrhu umožňující zadat název, popis, cenu, celkovou kapacitu a datum počátku a konce.
3. Uživatel tyto údaje vyplní.
4. Include (Vytvoření termínů).

5. Systém zkontroluje, zda zadané rozsahy dat jsou validní a následně pololetí vytvoří.

1.4.6.2 UC5-2: Vytvoření termínů

V rámci vytváření rozvrhu musí uživatel zadefinovat také alespoň 1 termín. U termínu následně specifikuje den v týdnu, čas začátku, místo konání a datum počátku a konce tohoto termínu.

Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor definuje rozvrh.
2. Systém zobrazí uživateli možnost definovat jednotlivé termíny. O každém termínu musí zadat den v týdnu, čas začátku, místo konání a datum počátku a konce tohoto termínu.
3. Uživatel vytvoří alespoň jeden termín.
4. Systém jednotlivé termíny vytvoří.

1.4.6.3 UC5-3: Zobrazení rozvrhu

O každém rozvrhu si může administrátor zobrazit informace, které zadal při vytvoření – jde tedy jak o základní atributy rozvrhu, tak všechny jeho termíny. Stejná data také uvidí rodiče při vytváření přihlášky.

Aktéři: administrátor

1.4.6.4 UC5-4: Úprava rozvrhu a termínů

Administrátor může kdykoliv během školního roku změnit kterýkoliv z rozvrhů či termínů. Změna se může týkat jakéhokoliv atributu, který definoval při vytváření – většina informací má totiž pouze informační roli. Jediné omezení se týká kapacity, která nesmí být menší než počet přihlášených účastníků.

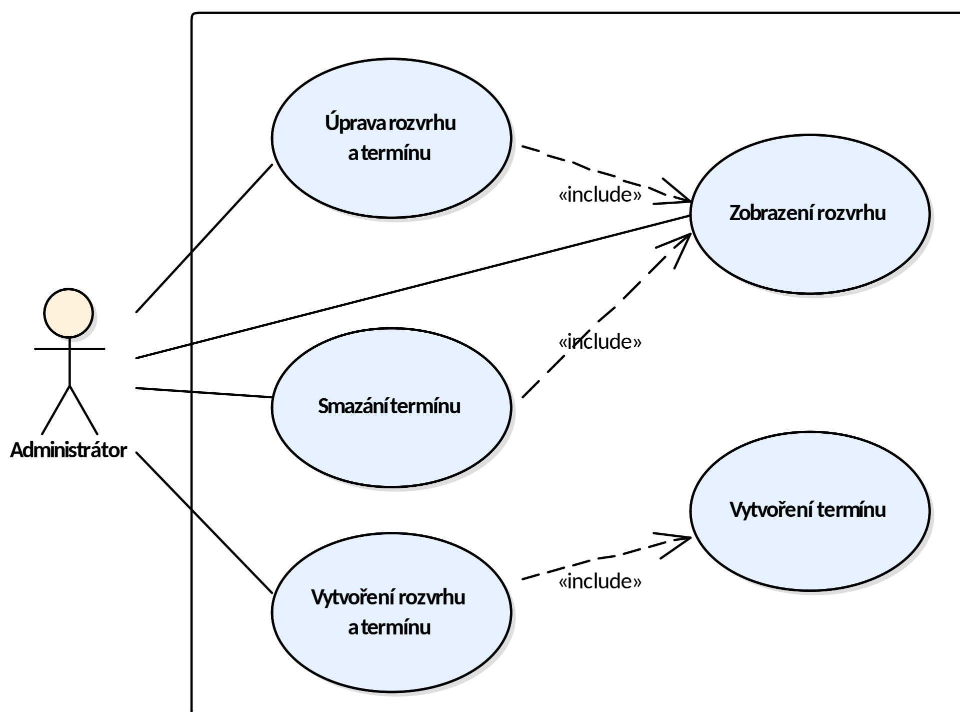
V rámci této úpravy může administrátor přidat nové termíny, upravit stávající nebo některé smazat. Za každé situace nicméně musí existovat pro daný rozvrh alespoň jeden termín.

Aktéři: administrátor

1.4.6.5 UC5-5: Smazání rozvrhu

Za předpokladu, že daný rozvrh není potřeba a některý z jeho termínů nenavštěvuje žádný z účastníků, může administrátor tento rozvrh ze systému smazat. S tímto rozvrhem se při smazání odstraní i termíny, které jsou k němu přiřazeny.

Aktéři: administrátor



Obrázek 1.4: Případy užití pro FP5

1.4.7 Zapisování docházky

Všechny následující případy užití se týkají zapisování docházky učiteli. Daný učitel může tyto operace provádět pouze nad rozvrhy, ke kterým má přístup.

Diagram na obrázku 1.5 zobrazuje případy užití, které se týkají funkčního požadavku FP6 popsaném v podkapitole 1.3.1.6.

1.4.7.1 UC6-1: Zobrazení týdenního přehledu

Při zapisování docházky potřebuje mít učitel jasný přehled o tom, které lekce jsou k zapsání, které lekce již zaznamenal, na které zapomněl a které jsou automaticky zrušeny.

Systém tedy musí tyto údaje přehledně prezentovat a každý týden zobrazit učiteli přehled o jeho lekcích. V tomto přehledu se budou zobrazovat pouze rozvrhy, které daný učitel spravuje.

Aktéři: učitel

1.4.7.2 UC6-2: Zapsání lekce

Když probíhá daný kroužek, učitel musí mít možnost na začátku hodiny zaznamenat základní údaje o dané lekci. Kromě docházky jednotlivých žáků může uvést, co se na dané hodině bude dělat, a případně si něco poznamenat na příští lekci.

O žácích bude zaznamenávat docházku jedním ze 3 stavů: přítomen, omluven, nebo nepřítomen.

Aktéři: učitel

Hlavní scénář: Obyčejné zapsání lekce

1. Scénář začíná, když se učitel rozhodne zapsat lekci.
2. Include (zobrazení týdenního přehledu).
3. Uživatel zvolí lekci k zapsání.
4. Systém zobrazí uživateli formulář k zápisu lekce.
5. Uživatel vyplní údaje o tom, co se bude na dané lekci dělat, může doplnit další poznámky a vyplní docházku všem účastníkům daného kroužku.
6. Systém zkontroluje, zda byla docházka vyplněna pro všechny žáky a následně lekci uloží.

Alternativní scénář: Zrušení lekce

1. Scénář začíná po 4. kroku hlavního scénáře, jestliže se uživatel rozhodne lekci zrušit.
2. Systém zobrazí uživateli formulář ke zrušení lekce.
3. Uživatel může volitelně vyplnit důvod ke zrušení dané lekce.
4. Systém uloží lekci jako zrušenou se zadaným důvodem. Pokud byl důvod ponechán prázdný, lekce se bude zobrazovat jako zrušená bez důvodu.

1.4.7.3 UC6-3: Zobrazení historie lekcí

Umožní uživateli zobrazit si zapsané lekce pro daný rozvrh. Každý učitel bude vidět historii pouze u těch rozvrhů, ke kterým je přiřazen.

Aktéři: učitel

1.4.7.4 UC6-4: Zobrazení zapsané lekce

Systém musí umožnit učiteli zobrazit si již zapsanou lekci. V detailu učitel uvidí, kdo lekci zapsal, kdy byla zapsána, co bylo obsahem hodiny, poznámky a samozřejmě docházku.

Aktéři: učitel

Hlavní scénář: Zobrazení z přehledu

1. Scénář začíná, když se učitel rozhodne zobrazit si detail zapsané lekce.
2. Include (zobrazení týdenního přehledu).
3. Uživatel si zobrazí detail po vybrání jedné z již zapsaných lekcí probíhajících daný týden.

Alternativní scénář: Zobrazení z historie

1. Scénář pokračuje po 1. kroku hlavního scénáře.
2. Include (zobrazení historie lekcí).
3. Uživatel si zobrazí detail po vybrání jedné z lekcí z historie.

1.4.7.5 UC6-5: Úprava zápisu lekce

Protože v průběhu dané lekce může dojít ke změnám v docházce nebo v obsahu hodiny, musí mít učitel možnost zápis změnit.

Alternativně může učitel dodatečně již zapsanou lekci zrušit – podobně jako při vytváření.

Hlavní scénář

1. Scénář začíná, když se učitel rozhodne upravit zápis lekce.
2. Include (zobrazení zapsané lekce).
3. Uživatel se rozhodne vybranou lekci upravit.
4. Systém zobrazí formulář s předvyplněnými údaji o vybrané lekci.

5. Uživatel upraví libovolné údaje z těch, které zadával při vytváření lekce.
6. Systém uloží provedené změny.

Aktéři: učitel

1.4.7.6 UC6-6: Smazání zápisu lekce

Umožní učiteli smazat zápis k již zapsané lekci. Po smazání zápisu se lekce bude objevovat v přehledu opět jako nezapsaná.

Aktéři: učitel

Hlavní scénář

1. Scénář začíná, když se učitel rozhodne smazat zapsanou lekci.
2. Include (zobrazení zapsané lekce).
3. Uživatel se rozhodne vybranou lekci smazat.
4. Systém požádá uživatele o potvrzení smazání.
5. Uživatel potvrdí smazání vybrané lekce.
6. Systém danou lekci smaže.

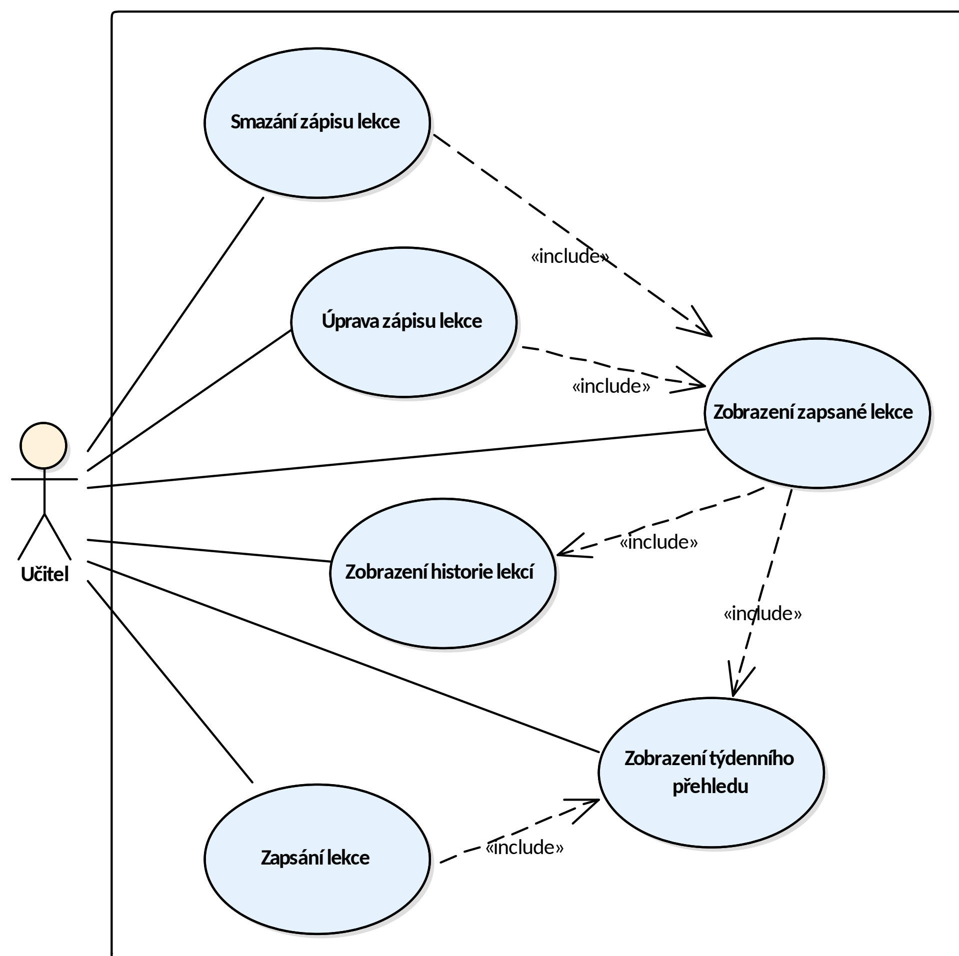
1.4.8 Správa přihlášek

Diagram na obrázku 1.6 zobrazuje případy užití, které se týkají funkčního požadavku FP7 popsaném v podkapitole 1.3.1.7.

1.4.8.1 UC7-1: Podání přihlášky

Když chce rodič přihlásit svoje dítě na některý z kroužků, musí podat elektronickou přihlášku. Pokud ještě nemá v systému účet, je mu současně s přihláškou vytvořen.

Aktéři: rodič



Obrázek 1.5: Případy užití pro FP6

Hlavní scénář

1. Scénář začíná, když se rodič rozhodne zapsat svoje dítě na některý z kroužků.
2. Systém zobrazí uživateli formulář s dostupnými kroužky, na kterých je ještě volné místo.
3. Uživatel vybere jeden nebo více kroužků, kam chce svoje dítě přihlásit.
4. Systém zobrazí formulář, kam lze zadat základní informace o dítěti, jako je jméno, příjmení, datum narození a rodné číslo.
5. Uživatel vyplní základní informace o dítěti.
6. Systém zobrazí formulář umožňující zadat informace o jednom nebo o dvou zákonných zástupcích.

7. Uživatel vyplní svoje jméno, příjmení, e-mail a telefonní číslo.
8. Systém zobrazí přehled přihlášky, ve kterém jsou všechny zadané údaje z předchozích kroků.
9. Uživatel informace zkontroluje, potvrdí souhlas s podmínkami a závazně přihlášku odešle.
10. Systém přihlášku uloží, pro všechny rodiče vygeneruje nový účet (pokud již neexistují). O dokončené přihlášce informuje systém rodiče e-mailem.
11. Extends (zaplacení přihlášky).

1.4.8.2 UC7-2: Zobrazení přihlášky

O odeslané přihlášce si může rodič zobrazit všechny dostupné informace jako: datum odeslání, datum splatnosti, celková částka k úhradě a jednotlivé rozvrhy, které jsou součástí přihlášky.

Aktéři: rodič, administrátor

1.4.8.3 UC7-3: Zaplacení přihlášky

Pro usnadnění procesu jsou platby prováděny skrze platební bránu. Rodič tedy pro dokončení přihlašování musí přihlášku zaplatit.

Aktéři: rodič

Hlavní scénář

1. Scénář začíná, když se rodič rozhodne zaplatit za již podanou přihlášku.
2. Systém přesměruje uživatele do platební brány, kde je předvyplněna částka k úhradě. Ta se rovná součtu cen jednotlivých kroužků.
3. Uživatel uvnitř prostředí platební brány platbu odešle.
4. Systém přesměruje uživatele zpět na stránky tohoto systému, úspěšnou platbu zaznamená a přesune přihlášku do stavu „Aktivní“. O obdržení platby systém informuje rodiče e-mailem.

1.4.8.4 UC7-4: Automatické zrušení přihlášky

Umožní automaticky zrušit všechny přihlášky, které nebyly do určitého počtu dní od podání zaplaceny. Tento počet dní bude nastavitelný pro každou školu zvlášť.

Aktéři: čas

Hlavní scénář

1. Scénář začíná, když se přihláška po n dnech nachází stále ve stavu „Čeká na platbu“.
2. Systém přesune přihlášku do stavu „Zrušená“. Současně odešle e-mail všem rodičům přiřazeným k dítěti informující o automatickém zrušení.

1.4.8.5 UC7-5: Manuální změna stavu přihlášky

Může nastat situace, kdy musí administrátor stav přihlášky upravit manuálně. Tato situace může nastat, když rodič provede platbu v hotovosti, když chce administrátor manuálně odhlásit dítě z kroužku aj.

Administrátor tedy musí mít možnost na přihlášce manuálně měnit její stav. Ostatní úpravy přihlášky (tedy především změna rozvrhů na přihlášce) nejsou žádoucí a zbytečně by komplikovaly logiku.

Aktéři: administrátor

1.4.8.6 UC7-6: Smazání přihlášky

Někdo z rodičů může udělat při přihlašování chybu nebo podat přihlášku omylem. Administrátor by měl mít možnost takovou přihlášku smazat.

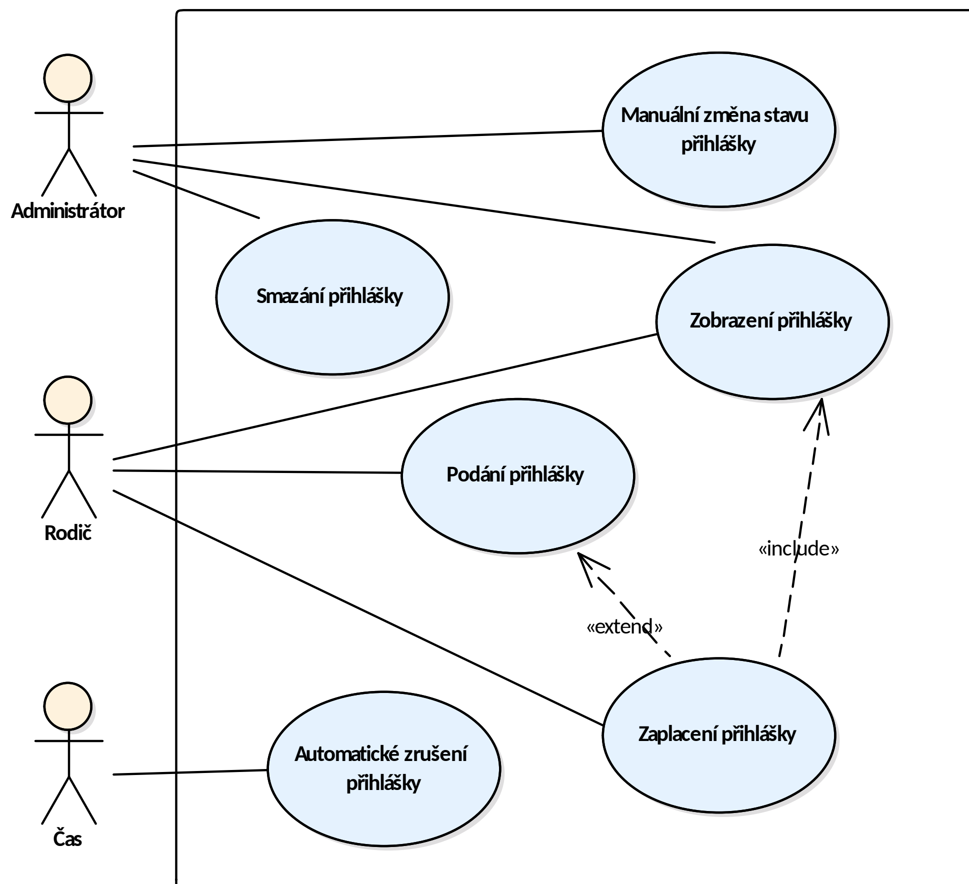
Aktéři: administrátor

Hlavní scénář

1. Scénář začíná, když se administrátor rozhodne smazat přihlášku.
2. Include (zobrazení přihlášky).
3. Uživatel zvolí možnost vybranou přihlášku smazat.
4. Systém si vyžádá potvrzení této akce.
5. Uživatel potvrdí smazání vybrané přihlášky.
6. Systém přihlášku smaže. Jak rodičovské účty, tak entity dětí zůstanou v systému zachovány, a i když nemají přiřazené žádné kroužky, smazány nebudou.

1.4.9 Správa uživatelů

Přestože je správa uživatelů návrhově relativně náročná oblast, nejsou zde odpovídající případy užití detailně rozepsány. Je to především z toho důvodu, že se nijak zásadně neodlišují od jakéhokoliv jiného informačního systému, který pracuje s uživateli. Uživatele bude možné manuálně vytvořit, upravit a smazat. Přiřazené role uživatele pak v rámci úprav bude dovoleno měnit.



Obrázek 1.6: Případy užití pro FP7

1.4.10 Export dat

1.4.10.1 UC9-1: Export zobrazované tabulky

Systém by měl dovolovat export jakékoliv tabulky, která je uživateli zobrazována. Ve většině případů bude tato možnost dostačující. Může se sice někdy stát, že uživatel bude mít na stahovaná data jiné požadavky, nicméně by to mělo nastávat velmi zřídka.

Na výběr by měl dostat z formátů uvedených u funkčního požadavku FP9. Ten je popsán v podkapitole 1.3.1.9.

Aktéři: uživatel

Hlavní scénář

1. Scénář začíná, když si uživatel chce exportovat některá data ze systému.

2. Systém zobrazí formulář pro uložení zobrazených dat. Lze zde zvolit, které sloupečky mají být do souboru zahrnuty, a do jakého formátu mají být data uložena.
3. Uživatel zvolí sloupečky a formát souboru.
4. Systém vytvoří požadovaný soubor a započne jeho stahování.

1.5 Pokrytí funkčních požadavků

V tabulce 1.1 je zobrazeno pokrytí funkčních požadavků případy užití. Tato tabulka je vhodná pro kontrolu toho, zda všechny funkční požadavky byly splněny – tedy zda je každý pokryt alespoň jedním případem užití [17]. Jak lze z této tabulky vyčíst, zde tomu tak je.

	Požadavky								
Případy užití	FP1	FP2	FP3	FP4	FP5	FP6	FP7	FP8	FP9
UC1-1	X								
UC1-2	X								
UC1-3	X								
UC2-1		X							
UC2-2		X							
UC2-3		X							
UC3-1			X						
UC3-2			X						
UC3-3			X						
UC3-4			X						
UC4-*				X					
UC5-1					X				
UC5-2					X				
UC5-3					X				
UC5-4					X				
UC5-5					X				
UC6-1						X			
UC6-2						X			
UC6-3						X			
UC6-4						X			
UC6-5						X			
UC6-6						X			
UC7-1							X		
UC7-2							X		
UC7-3							X		
UC7-4							X		
UC7-5							X		
UC7-6							X		
UC8-*								X	
UC9-1									X

Tabulka 1.1: Pokrytí funkčních požadavků případy užití, zápis s hvězdičkou vyjadřuje všechny detailně nerozepsané případy užití v dané kategorii

Kapitola 2

Návrh

2.1 Aplikační rozhraní

Aplikační rozhraní (dále API) je způsob, jakým spolu mohou komunikovat různé systémy. Na rozdíl od uživatelského rozhraní se API používá pro komunikaci aplikací. Nepoužívají ho tedy koncoví uživatelé, ale vývojáři.

API poskytují jasnou a jednoznačnou definici toho, jakým způsobem se má s daným systémem komunikovat. Když se někdo rozhodne tohoto rozhraní využít, může se na něj jednoduše napojit. Umožňuje to tedy nezávislý vývoj těchto systémů, protože jediné místo, kde probíhá komunikace, je definováno tímto rozhraním.

Důvodů, proč použít API, je několik. Tím nejzřetelnějším ve vývoji webových systémů je možnost oddělit od sebe vývoj serverové a klientské aplikace. Pokud by se tento projekt v budoucnu rozrůstal a byla by potřeba spolupracovat s novými partnery, možnost propojit tyto systémy přes API by ušetřila velké množství zdrojů [18].

V průběhu rozvoje oboru webových aplikací se používalo a používá několik architektur pro návrh API.

2.1.1 Representational state transfer (REST)

Jde o architektonický styl, který je svázán určitými podmínkami a omezeními. Pokud jsou tato omezení splněna, lze o daném systému prohlásit, že je tzv. „RESTful“ – tedy že naplňuje REST. Patří mezi ně:

1. Klient–server: Oddělením uživatelského rozhraní a byznysové logiky je docíleno jednoduššího škálování celého systému. Každá z aplikací se pak může rozvíjet svým vlastním tempem nezávisle na té druhé.

2. Bezstavovost: Každý požadavek na server by měl být nezávislý na všech ostatních. Musí tedy obsahovat všechny informace k tomu, aby se dal nezávisle vyhodnotit.

3. Mezipaměť: Na základě dostupných konceptů by mělo být možné odpovědi ukládat a ušetřit tak komunikaci se serverem.

4. Vícevrstvé systémy: Nemělo by být vyžadováno, aby klient znal implementační detaily serveru.

5. Kód na vyžádání (volitelné): Server by měl dovolit klientům předávát spustitelné skripty (např. v JavaScriptu). Cílem je snížení počtu funkcionalit, které server musí poskytovat [19].

Lze také tvrdit, že REST je abstrakce architektury hypertextových systémů – s největším zástupcem World Wide Web. REST byl tedy vyvinut tak, aby se dal snadno použít se standardem HTTP 1.1.

Hlavním pojmem pro styl REST je zdroj. Zdroj je jakákoliv informace, kterou lze určitým způsobem pojmenovat. Jedná se tedy například o soubory, objekty z reálného života nebo kolekce jiných zdrojů. Tento zdroj by měl být serializovatelný – tedy reprezentovatelný například ve formátu JSON [19].

Pro přístup k těmto zdrojům se využívají uniformní identifikátory zdrojů (zkráceně URI). Jde o koncept ze světa HTTP a jedná se o adresu daného zdroje, která ho identifikuje. Nad tímto konstruktem dále staví REST a základní operace mapuje na HTTP metody. Těch může být nad daným zdrojem vytvořeno hned několik. Většinou odpovídají konceptu CRUD – tedy vytvoření, čtení, upravení a smazání. Kombinace cesty zdroje a operace se pak nazývá koncový bod (také „endpoint“) [18]. Mapování těchto operací na HTTP metody je následující [20]:

- vytvoření zdroje $\iff$ POST,
- čtení zdroje $\iff$ GET,
- úprava zdroje $\iff$ PATCH [21],
- náhrada zdroje $\iff$ PUT,
- smazání zdroje $\iff$ DELETE.

Kromě zmíněných HTTP metod by měl REST používat i HTTP návratové hodnoty k signalizování typu úspěchu či neúspěchu [18]. Standard HTTP definuje následující možné návratové hodnoty v rozsahu *1xx–5xx*. Název a význam nemá přiřazeno všech 500 hodnot, nýbrž pouze několik desítek [22]. Nevyplněný prostor mezi nimi je rezervován pro možná budoucí rozšíření. Takto je lze rozdělit podle zmíněných řádů:

1. *1xx*: Informace: Zpráva o přijetí požadavku a následném vyhodnocení.

2. *2xx*: Úspěch: Operace byla úspěšně vykonána.

3. *3xx*: Přesměrování: Zdroj se nachází na jiném místě a je potřeba se k němu dostat jiným způsobem.

4. *4xx*: Chyba u klienta: Pochybil zde klient – například špatným formátem dotazu nebo kvůli nevykonatelnosti požadavku.

5. *5xx*: Chyba na serveru: Informace o tom, že server pochybil a je si vědom této anomálie [20].

2.1.2 Simple object access protocol (SOAP)

Jde o styl, který je úzce založen na stylu RPC. Ten využívá ke komunikaci s druhou stranou předem definované procedury. Je realizován prostřednictvím serializace celého dotazu do daného formátu a následné deserializace na vzdáleném zařízení. Odpověď je uskutečněna obdobně.

SOAP na něm tedy staví a pro komunikaci používá formát XML. Zpráva musí být vložena do „SOAP obálky“, která navíc zprávě přidává další metadata. Následně je třeba uvnitř definovat volané procedury a jejich parametry. Odpovědí je nová SOAP zpráva strukturovaná totožně, tentokrát ovšem již s výsledkem operace [23].

Služby založené na tomto architektonickém stylu bývají považovány za komplexní. Je to především kvůli velikosti a složitosti jednotlivých požadavků. Práce s nimi je tedy o něco náročnější. Nicméně pro velké podnikové systémy to může být stále vhodná volba [18].

2.1.3 GraphQL

GraphQL nabízí – na rozdíl od předchozích zmíněných – klientovi možnost definovat si, jakým způsobem má dotaz vypadat. Místo toho, aby byly specifikovány konkrétní endpointy, se předává kontrola o získaných datech tomu, kdo o ně žádá.

Pomocí strukturovaného deklarativního jazyka je umožněno specifikovat strukturu dat, která budou součástí odpovědi ze serveru. Nespecifikuje se zde, jak se má dotaz vykonat. Pouze se určuje, co má být výsledkem. Jsou podporovány jak čtecí, tak zapisovací operace [18].

V projektech, kde se velmi rapidně mění požadavky, je použití této technologie velmi vhodné. Zde se nezávislost systémů projeví o to více. Pokud se totiž změní požadavky na klientskou část aplikace a je například nutno zobrazit nový parametr, stačí změnit definici dotazu. Serverová část tedy nemusí v tomto případě projít žádnou změnou.

2.1.4 Shrnutí

Z výše analyzovaných stylů vychází, že nejvhodnější volbou bude použít REST architektonický styl. Přestože by SOAP byl také vhodné volba, hodí se spíše

do velkých podnikových aplikací. REST je jednoduše použitelná a velmi odlehčená technologie. Napříč IT odvětvím je zdaleka nejrozšířenější [18]. Mimo jiné díky RESTu by také bylo možné v budoucnu projekt jednoduše rozšířit i na další platformy.

2.2 Návrh endpointů

Na základě analýzy API byl již výše zvolen styl REST (více v podkapitole 2.1). REST definuje komunikaci serveru s okolním světem prostřednictvím endpointů. U každého z nich je třeba sledovat lokaci jeho zdroje, typ dostupné HTTP metody, strukturu odpovědi, možné návratové hodnoty a případně strukturu těla požadavku.

Pro ukázku jsou zde představeny pouze endpointy týkající se funkčního požadavku FP7. Ten je detailněji popsán v podkapitole 1.3.1.7. Kompletní seznam všech navrhovaných endpointů je součástí přílohy B. Dokumentace jednotlivých endpointů je pak dostupná na přiloženém médiu.

2.2.1 Správa přihlášek

2.2.1.1 Podání přihlášky

Http metoda	Cesta zdroje
POST	/schools/{schoolId}/applications

Tabulka 2.1: Popis koncového bodu podání přihlášky

Endpoint v tabulce 2.1 bude sloužit k vytvoření nové přihlášky. Každá škola má definované aktuální pololetí, a proto k identifikaci období, ke kterému se má přihláška navázat, stačí pouze identifikátor školy. Tato akce vytváří v databázi nové entity, takže by metoda tohoto endpointu měla být POST.

Součástí musí být informace o účastníkovi, jeho rodičích a na jaké konkrétní rozvrhy je přihlašován. Tato data budou předána prostřednictvím těla požadavku. Jeho struktura je zobrazena v ukázce 2.1.

```
1  {
2    "child": {
3      "name": "Petr",
4      "surname": "Novak",
5      "date_of_birth": "2021-08-12",
6      "birth_number": "000101/9656",
7      "foreigner": false,
8      "address": {
9        ...
```

```
10      }
11    },
12    "parents": [
13      {
14        "name": "Jan",
15        "surname": "Novak",
16        "email": "novak@example.com",
17        "phone": "+420 123 456 789"
18      }
19    ],
20    "note": "alergie na pyl",
21    "child_timetables": [
22      42
23    ],
24    "consent_given": true
25  }
```

Ukázka 2.1: Tělo požadavku podání přihlášky

Po podání dotazu bude možné obdržet čtyři různé odpovědi. Buď přihláška bude úspěšně vytvořena a budou o ní zobrazeny další informace, nebo dojde ke špatné specifikaci dat. V tom případě bude zobrazeno, z jakého důvodu k chybě došlo. Za předpokladu, že uživatel není autentifikovaný, tento požadavek bude zamítnut a ke zpracovaní nedojde. Poslední možností je, že požadovaná škola nebude existovat. Tyto možné odpovědi mohou vypadat takto:

1. **201 Created**: Přihláška byla úspěšně vytvořena. Tělo odpovědi je zobrazeno v ukázce 2.2.

```
1  {
2    "id": 42,
3    "datetime": "2022-01-01",
4    "due_date": "2022-01-14",
5    "application_status": "WAITING_FOR_PAYMENT",
6    "total_price": 1000
7  }
```

Ukázka 2.2: Tělo úspěšné odpovědi podání přihlášky

2. **400 Bad Request**: Přihlášku se nepodařilo vytvořit z důvodu zadaných dat klientem. Chyba bude opět vrácena jako součást těla odpovědi, jak je definováno v ukázce 2.3.

```
{
  "code": "400",
  "message": "Nelze vytvorit novou prihlasku",
  "errors": {
    "name": "Musi byt specifikovan alespon 1 rodic."
  }
}
```

Ukázka 2.3: Tělo chybné odpovědi podání přihlášky

3. **403 Access Denied**: Nebylo možné ověřit identitu odesílatele. Z toho důvodu byl přístup k tomuto endpointu zamítnut.
4. **404 Not Found**: Požadovaná škola nebyla nalezena.

2.2.1.2 Zobrazení přihlášky

Http metoda	Cesta zdroje
GET	/applications/{applicationId}

Tabulka 2.2: Popis koncového bodu zobrazení přihlášky

Získání jedné podané přihlášky bude dostupné na cestě zobrazené v tabulce 2.2. Tento endpoint v databázi nic neupravuje ani nemaže, ale pouze zobrazuje data. Je zde proto vhodné zvolit metodu GET. Součástí cesty bude jeden parametr, a to identifikátor požadované přihlášky. Tělo požadavku není potřeba, a proto bude tedy ponecháno prázdné. Možné návratové hodnoty pro zobrazení přihlášky budou:

1. **200 Ok**: Požadovaná přihláška, která bude dostupná v těle odpovědi definovaném v ukázce 2.4.

```
{
  "id": 42,
  "datetime": "1628769600",
  "due_date": "1628726400",
  "note": "alergie na pyl",
  "total_price": 1000,
  "child": {
    "id": 42,
    "name": "John",
```

```
10      "surname": "Doe",
11      ...
12    },
13    "timetable_contracts": [
14      {
15        "id": 42,
16        ...
17      }
18    ]
19  }
```

Ukázka 2.4: Tělo úspěšné odpovědi zobrazení přihlášky

2. **403 Access Denied**: Přístup k tomuto zdroji byl zamítnut.
3. **404 Not Found**: Požadovaná přihláška nebyla nalezena.

2.2.1.3 Zobrazení všech přihlášek školy

Http metoda	Cesta zdroje
GET	/schools/{schoolId}/applications

Tabulka 2.3: Popis koncového bodu zobrazení přihlášek školy

Pro zobrazení všech přihlášek, které byly podány na danou školu, bude sloužit endpoint popsaný v tabulce 2.3. Opět jde pouze o získávání dat, a proto je zde vhodná metoda GET.

Přihlášek, které se týkají jedné školy, by mohlo být relativně hodně. Nejde sice o získání všech záznamů dané tabulky, nicméně i tak by to mohlo být výkonnostně náročné. Vhodnější strategií je použít stránkování, které je detailněji popsáno v podkapitole 3.3.6.1 [24]. V okamžiku, kdy je třeba stránkování, nabízí se poskytnout i možnost výsledek filtrovat a řadit. To je více popsáno v podkapitole 3.3.6.2.

Parametry pro stránkování bude možné předat přímo v cestě pomocí query parametrů. Budou to parametry: *page* – požadovaná stránka a *per_page* – požadovaný počet položek na jedné stránce. Kromě toho budou dostupné další 2 parametry na filtrování a řazení: *filter* a *sort*. Možné odpovědi na tento požadavek budou:

1. **200 Ok**: Seznam přihlášek dané školy, které budou dostupné v těle odpovědi zobrazeném v ukázce 2.5. Struktura jednotlivých přihlášek bude totožná s předchozím endpointem.

```
{
  "_pagination": {
    "total": 42,
    "page": 2,
    "per_page": 10
  },
  "items": [
    {
      "id": 1,
      ...
    }
  ]
}
```

Ukázka 2.5: Tělo úspěšné odpovědi zobrazení přihlášek školy

2. **403 Access Denied**: Přístup k tomuto zdroji byl zamítnut.

2.2.1.4 Úprava přihlášky

Http metoda	Cesta zdroje
PUT	/applications/{applicationId}

Tabulka 2.4: Popis koncového bodu úpravy přihlášky

Na základě případu užití popsaném v podkapitole 1.4.8.5 je třeba prostřednictvím API poskytnout možnost změnit aktuální stav přihlášky. K tomu bude sloužit endpoint popsaný v tabulce 2.4. Stav je pouze jeden z údajů, které jsou o přihlášce uloženy. Bude tedy vhodnější použít metodu PATCH. Ta totiž oproti metodě PUT při požadavku nenahradí celý zdroj, ale pouze aktualizuje požadované atributy [21].

Tělo požadavku zobrazené v ukázce 2.6 bude obsahovat pouze jediný atribut, a to nový stav přihlášky.

```
{
  "application_status": "ACTIVE"
}
```

Ukázka 2.6: Tělo požadavku úpravy přihlášky

Po vykonání tohoto požadavku může opět nastat několik situací. Jako odpověď může totiž přijít několik různých návratových hodnot:

1. **200 Ok**: Přihláška byla úspěšně upravena. Odpověď bude obsahovat upravenou přihlášku a bude mít stejnou strukturu jako v ukázce 2.4.
2. **403 Access Denied**: Přístup k tomuto zdroji byl zamítnut.
3. **404 Not Found**: Požadovaná přihláška nebyla nalezena.

2.2.1.5 Smazání přihlášky

Http metoda	Cesta zdroje
DELETE	/applications/{applicationId}

Tabulka 2.5: Popis koncového bodu smazání přihlášky

Endpoint zobrazený v tabulce 2.5 bude sloužit ke smazání dané přihlášky. Protože dojde k odstranění některých záznamů z databáze, byla zde zvolena metoda DELETE. Cesta ke zdroji obsahuje identifikátor přihlášky, která má být smazána. V těle požadavku nebudou posílána žádná data. Výsledkem může být jeden z následujících stavových kódů:

1. **204 No Content**: Smazání přihlášky proběhlo úspěšně.
2. **403 Access Denied**: Přístup k tomuto zdroji byl zamítnut.
3. **404 Not Found**: Požadovaná přihláška nebyla nalezena.

2.3 Architektura

2.3.1 Více-vrstvá architektura

Při vytváření softwarové aplikace je vhodné zvolit pro systém architekturu. Architektura je způsob, jakým lze dělit zdrojový kód do daných celků, které spolu tematicky souvisí. Toto dělení je v tomto případě čistě logické a nijak nesouvisí s fyzickým umístění zdrojového kódu na serveru [25].

Mezi nejčastější dělení patří architektury: dvou-vrstvá, tří-vrstvá a více-vrstvá. První zmíněná není příliš vhodná pro vývoj rozsáhlejších systémů s více klientskými aplikacemi. Poslední je naopak pro tento projekt příliš složitá [25, 26]. Následující podkapitola se tedy bude zabývat především tří-vrstvou architekturou.

2.3.1.1 Tří-vrstvá architektura

Tří-vrstvá architektura specifikuje dělení zdrojového kódu na následující 3 části:

- prezentační vrstva,

- byznysová vrstva,
- datová vrstva.

V datové vrstvě jsou uchovávána a spravována veškerá aplikační data. Jejím hlavním úkolem je vyhovovat požadavkům na data a předávat je do vyšších vrstev. Byznysová vrstva se stará o logiku, která je specifická dané aplikaci. Toto chování je mimo jiné utvářeno funkčními požadavky. Stejně jako datová, musí tato vrstva reagovat na dotazování těch vyšších. V prezentační vrstvě jsou již předzpracovaná data zobrazena uživateli. V případě této serverové aplikace bude uživatelským rozhraním pouze API a uživatelem klientská aplikace.

Závislosti uvnitř této architektury musí být striktně jednosměrné. Vyšší vrstvy mohou komunikovat pouze s těmi nižšími. Žádná vrstva také nesmí být při toku dat přeskočena – především byznysová. Nesmí tedy být například žádáno z datové vrstvy o informace z prezentační či z prezentační komunikováno napřímo s datovou [26].

Dodržení této architektury může být pro projekt výhodné v několika oblastech. Striktním dělením zdrojového kódu do jednotlivých vrstev lze kteroukoliv vrstvu kdykoliv vyměnit. Pokud uživatelské rozhraní nebude obsahovat žádnou doménovou logiku, bude jednodušší vyvinout nové klientské aplikace. V neposlední řadě je zde jednodušší možnost projekt škálovat. Zavedením této standardizace má zdrojový kód v projektu vždy jasně definované místo a budoucí rozvoj projektu to pouze usnadní [25].

2.3.2 Architektura prezentační vrstvy

Pro komunikaci aplikace s uživatelem byl již určen jako prostředek API. Také zde je třeba uváženě zvolit vhodný návrhový vzor. Stejně jako rozdělení na jednotlivé vrstvy, specifikace tohoto vzoru pro uživatelské rozhraní může pomoci dosáhnout lepší udržitelnosti celého projektu.

Mezi nejpoužívanější vzory se řadí Model–View–Controller (MVC) a dále pak Model–View–Presenter (MVP).

2.3.2.1 Model–View–Controller

Tento vzor rozděluje zdrojový kód na tři téměř nezávislé části – Model, View a Controller. Model zahrnuje veškerou byznysovou logiku a procesy. Dochází v něm ke komunikaci s databází a k přípravě prezentovaných dat.

Na View by se dalo nahlížet jako na uživatelské rozhraní – prezentuje totiž data uživateli. Tato data získává dotazováním modelu, který ji relevantní předzpracované informace dodává. Současně dostává od modelu upozornění o nových změnách.

Controller koordinuje všechny tyto části dohromady. Reaguje na uživatelské akce, o kterých je informován z View. Na základě těchto požadavků požádá

Model o odpovídající změnu stavu. Controller je také zodpovědný za zvolení korektního View [27].

V serverových aplikacích má uživatelské rozhraní trochu jinou podobu. Tímto rozhraním je výše zmíněné API (popsáno v podkapitole 2.1). Model v tomto případě zůstane nezměněn. Controller bude definovat endpointy, které budou dotazovat Model. Výstupem bude text v libovolném formátu, který je možné namapovat v tomto vzoru na View.

2.3.2.2 Model–View–Presenter

V porovnání s MVC se kromě přejmenování Controlleru na Presenter liší MVP především v komunikaci těchto komponent mezi sebou.

View je totiž nezávislé na Modelu a svá data získává přímo z Presenteru. Komunikuje s ním napřímo udržováním reference na něj. Presenter je zodpovědný za zpřístupňování a upravování doménových informací Modelu. Ty následně předává konkrétnímu View k zobrazení. Aby nevznikla cyklická reference, k View je odtud přistupováno prostřednictvím rozhraní. Role Modelu je totožná svému protějšku v MVC [27].

2.4 Využité technologie

Pro webovou aplikaci je v první řadě potřeba zvolit vhodné technologie pro vývoj. Volba těchto technologií ovlivní budoucnost a rozšiřitelnost tohoto projektu.

Jedno z kritérií pro výběr by měla být používanost dané technologie. Pokud bude velká řada vývojářů již znát technologie, které systém využívá, nebude problém najmout nové programátory. Bude tedy možné vyhnout se nákladným školením nebo zaučování dalších členů týmu.

2.4.1 Framework

Některé problémy vývojáři řeší pokaždé, když vytváří webovou aplikaci. Téměř každý projekt bude v určité formě řešit zabezpečení, práci s databází nebo volání endpointů. Aby se tyto funkcionality nemusely neustále vyvíjet znovu, vznikla řešení, která se této duplikaci snaží předejít – tzv. „frameworky“. Jde o soubory několika různých knihoven, které zároveň poskytují určitou architekturu pro vývoj. Současně obsahují řešení výše zmíněných obvyklých problémů webových aplikací.

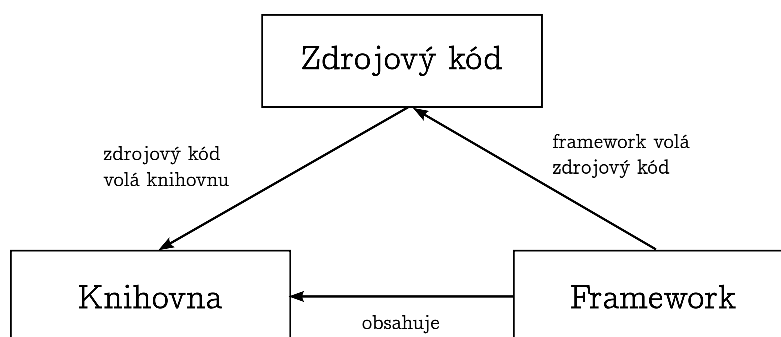
Dodaná architektura poskytuje vývojářům možnost soustředit se na tvoření byznysové logiky místo zaobírání se již vyřešenými problémy. Není třeba znovu objevovat kolo, když už ho někdo vynalezl. Využití těchto již existujících řešení vede k rychlejšímu dodání finálního produktu. Modularita těchto nástrojů současně umožňuje, při dodržení smluvených postupů, jednodušší paralelizaci práce mezi různé členy týmu.

Odlišnost frameworku od knihovny popisují Porebski a kol. následovně:

„The main difference between a library and a framework is that:“

- *„libraries are called from your code“*
- *„frameworks call your code“ [28]*

Rozdíl je jasně pozorovatelný na následujících příkladech. Jako knihovní funkci si lze představit třídění nebo hledání nad polem. Ve zdrojovém kódu je třeba funkci specificky zavolat a ona vrátí výsledek. Naopak například při definování endpointů ve frameworku je často vytvořena metoda a samotný framework ji v určitém momentu zavolá. Tento vztah je reprezentován na obrázku 2.1.



Obrázek 2.1: Vztah frameworku, knihovny a zdrojového kódu, založeno na `https://www.programcreek.com/2011/09/what-is-the-difference-between-a-java-library-and-a-framework/`

Na tomto místě je třeba také uvést termín „inverze kontroly“. Jedná se o princip, kdy komponenty systému umístěné ve vyšších vrstvách architektury předají jejich subsystémům kontrolu nad tím, co mají vykonávat. Samy se tím stanou mnohem méně závislé na nižších částech systému. Tomuto ve frameworcích odpovídá možnost programátorů definovat části zdrojového kódu, které sami nikde nevolají. Framework však za pomoci automatické generace zdrojového kódu přesto zajistí jejich spuštění.

Ne vždy se vyplatí sáhnout právě po frameworku. Existují situace, kde může být použití tohoto řešení naopak kontraproduktivní. Jsou to případy, kdy je potřeba pouze statický web, jsou vysoké nároky na efektivitu vyhodnocení nebo se pracuje s velmi experimentálními požadavky. Nic z toho ovšem není případ tohoto projektu. Tato aplikace je standardní webovou aplikací s dynamickým obsahem a klasickými funkcemi, jako je zabezpečení přístupu a práce s databází [28].

Z předchozích odstavců lze tedy vyvodit to, že framework se pro použití v tomto projektu hodí. Je nyní nutné zvolit, který je nejvhodnější. Výběr správného frameworku může rozhodnout o úspěchu nebo zániku naší aplikace. V následujících podkapitolách jsou tedy popsány nejpoužívanější nástroje tohoto

typu z různých programovacích jazyků. Na základě počtu hvězdiček u repozitářů na úložišti GitHub v roce 2022 vycházejí nejlépe Laravel, Django, a Flask [29]. Laravel sám je založen na frameworku Symfony, a proto zde bude také zmíněn [30].

2.4.1.1 Laravel

Laravel je jedním z nejpopulárnějších frameworků v jazyce PHP. Jako ostatní výše zmíněné frameworky je distribuován s otevřeným zdrojovým kódem pod licencí MIT. Je založen na architektuře Model–View–Controller (detailněji popsáno v podkapitole 2.3.2.1) [31].

Rozhraní frameworku umožňuje modulární konfiguraci závislostí, které jsou spravovány prostřednictvím manažeru závislostí Composer.

Podporuje objektově relační mapování (detailněji popsáno v podkapitole 2.4.3) pomocí interní knihovny Eloquent. Ta je implementována pomocí konceptu Active Record. Entitu, která reprezentuje relační tabulku, tedy stačí rozšířit o třídu `Model`, která zajistí propojení s databází [32].

Laravel využívá pro implementaci některých funkcionalit tzv. „magické“ metody. Je jich dohromady 15 a jsou součástí samotného PHP. Jde o metody, které nejsou volány programátorem, ale PHP je zavolá na pozadí samo. Laravel je v nadprůměrném množství využívá například pro ukládání atributů entit. Nejsou totiž definovány jako členské proměnné, ale jako pole atributů. Při zavolání atributu na entitě (například `entity->name`) dojde ve skutečnosti k zavolání magické metody `__get()`. Interní implementace Laravelu následně (kromě dalších operací) získá skutečnou hodnotu z pole atributů [33].

Jako šablonovací engine používá Laravel knihovnu Blade, kterou lze ovšem použít i samostatně. Blade se využívá pro renderování dynamického obsahu a umí základní direktivy pro řízení rozhodování (podmínky, cykly, ...).

Co je často vytýkáno, je rychlost vývoje. Tento framework je určen spíše pro větší projekty, které jsou rapidně vyvíjeny (tento přístup je nazýván Rapid Application Development). Pokud je tedy časový termín pro projekt tohoto typu velmi blízko, je Laravel vhodná volba [34]. U dlouhodobějších projektů bude ale naopak složitější na údržbu.

2.4.1.2 Django

Django je jeden z největších frameworků pro vývoj webových aplikací v programovacím jazyku Python. Stejně jako Laravel, jde o otevřený software a drží se architektury MVC.

Poskytuje ORM a používá pro jeho realizaci princip Active Record. Pro dotazování nad databází lze využívat vestavěné metody frameworku, nebo lze sestoupit do nižších vrstev a psát v SQL.

Co se týče šablon, lze využít vestavěnou knihovnu Django template language. Současně je zde podpora i knihovny třetích stran – jako například Jinja. Lze si dokonce definovat vlastní jazyk pro parsování šablon.

Z pohledu bezpečnosti umožňuje používat základní systém autentizace, který spravuje uživatele a jejich hesla. V případně potřeby je možné tento systém nevyužít a zakomponovat řešení třetí strany [35].

Frameworku je vytýkáno málo ustálených konvencí, a v důsledku toho se nemusí hodit pro větší projekty. Pro zprovoznění funkčnosti je totiž potřeba napsat více zdrojového kódu než u konkurenčních frameworků [32].

2.4.1.3 Flask

Flask je mikro-framework určený pro programovací jazyk Python. Je nazýván mikro z toho důvodu, že se jedná o opravdu nejmenší možný soubor knihoven, aby se jako framework dal identifikovat. Neposkytuje žádnou možnost práce s databází, nenabízí posílání a validaci formulářů, ani veliké možnosti zabezpečení. Není to však jeho cílem. Flask se snaží být takovým frameworkem, aby byl použitelný pro co nejvíce různých aplikací. Zahrnuje proto pouze funkcionality, které využije téměř 100 % projektů – tedy například šablonovací engine. Díky tomu je velmi flexibilní. To, co neposkytuje, je možné přidat dodatečnou knihovnou. Tímto způsobem tedy projekt obsahuje pouze ty závislosti, které skutečně potřebuje [36].

Nehodí se již z definice pro větší projekty. Jeho použití by bylo pouze časově náročné a zdrojový kód by byl kvůli různým knihovnám méně čitelný. V tomto ostatní frameworky, které poskytují širší funkcionalitu, vynikají [32].

2.4.1.4 Symfony

Symfony je jeden z dalších PHP frameworků. Stejně jako Laravel, je distribuován s otevřeným zdrojovým kódem a je založen na architektuře MVC. Pro správu závislostí je používán Composer. Mezi podporované funkcionality tohoto frameworku patří zabezpečení, šablony, logování, testování a internacionalizace.

Pro objektově relační mapování používá Symfony knihovnu Doctrine, která je založená na vzoru Data Mapper (detailněji popsáno v podkapitole 2.4.3.1). Entity a atributy se označují prostřednictvím anotací. Symfony pak samo zajistí namapování na relační databázi. Relační schéma databáze je udržováno pomocí tzv. „migrací“. Jsou v nich vytvářeny relační objekty a zajišťují plynulý přechod mezi verzemi schématu.

Šablony jsou realizovány prostřednictvím knihovny Twig. Lze ji využít pro vykreslování klientské aplikace nebo pro návrh obsahu e-mailů [37].

V některých ohledech může být Symfony složitější. Protože byl ale navrhnut pro větší dlouhodobé projekty, tato složitost je zastíněna jednodušší udržitelností vývoje [34].

2.4.1.5 Laravel vs. Symfony

Oba frameworky jsou určeny pro jazyk PHP. Jsou postaveny na architektuře MVC a oba poskytují základní funkcionality, které jsou od moderních frameworků očekávány, jako je zabezpečení, šablony nebo internacionalizace.

Laravel je oproti Symfony populárnější mezi vývojáři. Při ověření četnosti hledání pojmů ve vyhledávači Google pomocí Google Trends lze pozorovat, že je až 9krát hledanější [34]. Na platformě GitHub má pak zhruba o 40 tisíc více hvězdiček [29]. V počtu přispěvatelů do kódové základny se oba frameworky pohybují okolo 2,5 tisíce [34].

Symfony na rozdíl od Laravelu nevyužívá magické metody zdaleka v takovém množství. Zdrojový kód je tedy o něco čitelnější, protože vývojář nemusí dohledávat význam některých volání, která magické metody využívají.

2.4.1.6 Závěr

Na základě analýzy jednotlivých výše uvedených frameworků jsem se rozhodl vyřadit Flask, protože nejméně naplňuje požadavky na rozsáhlejší webovou aplikaci. Dále jsem zúžil výběr na Laravel a Symfony, protože obsahují funkce, které jsou potřebné pro splnění funkčních a nefunkčních požadavků.

Všechny tyto frameworky jsou velice podobné a rozdíly jsou pouze v drobnostech. Laravel ovšem oproti Symfony používá mnohem více magických metod. U větších projektů pak hrozí nebezpečí, že čitelnost zdrojového kódu bude již natolik ovlivněna, že vývoj aplikace bude neudržitelný. Přestože se Laravel těší větší popularitě a bude tedy zákonitě jednodušší nabírat vývojáře na tyto pozice, Symfony má stále dost velikou komunitní základnu a podporu, a proto tento fakt nehraje tak velikou roli. Laravel se ale hodí více pro RAD projekty, což tento systém není.

Kvůli výše uvedeným důvodům, a také kvůli své dřívější zkušenosti ze studia s PHP a se Symfony, jsem se rozhodl použít ve zbytku práce tento framework.

2.4.2 Databáze

Tato aplikace bude pracovat ve velké míře s daty tak jako většina aplikací tohoto typu. Nejvhodnějším řešením pro systémy tohoto rozměru je použít databázi. Volba konkrétního typu není jednoduchá a rozhoduje o ní několik aspektů.

2.4.2.1 Architektura databází

Databáze lze dělit na relační, NoSQL a objektově relační[1].

[1]Různé zdroje uvádí různá dělení, a proto jsou zde uvedeny pouze typy nejčastěji zmiňované.

Relační databáze ukládá data jako řádky do tabulek. Každá tabulka má své atributy a každý řádek v tabulce je jeden záznam. Pomocí odkazování se na jiné záznamy mohou vznikat relace.

NoSQL databáze (někdy také „Ne pouze SQL“) se vyznačují tím, že data neukládají pouze použitím SQL. Do této kategorie spadá ukládání dat pomocí párů *klíč–hodnota*, hierarchie dokumentů nebo grafová databáze.

Objektově relační databáze rozšiřují relační databáze tím, že umožňují pracovat s objektovými typy. Je také podporováno zapouzdření interních implementací metod a dědičnost atributů. Aktuálně ale není tento způsob příliš rozšířen [38].

2.4.2.2 PostgreSQL

PostgreSQL je objektově relační databáze. Byla vyvinuta jako projekt na Univerzitě Berkeley v Kalifornii v roce 1986 a od té doby kolem sebe vytvořila širokou komunitu vývojářů. Díky tomu může být distribuována na základě licence s otevřeným zdrojovým kódem. Licence *PostgreSQL License* je velmi podobná známějším BSD nebo MIT a je udržována organizací Open source Initiative.

Oproti konkurenčním databázím, jako je například Oracle nebo MySQL, je díky zmíněné licenci nabízena se všemi funkcemi zdarma. Je také vysoce rozšiřitelná. Nabízí možnost vytvářet si vlastní datové typy nebo dodávat vlastní funkcionalitu přímo do databáze. Nejen díky tomu je tedy vhodnou volbou pro webové aplikace [39].

2.4.3 Objektově relační mapování

Objektově relační mapování (dále ORM) je způsob, jakým lze propojit dva spolu složitě komunikující systémy – konkrétně objektový a relační. Jde o provázání doménových modelů a perzistentní relační databáze. Toho je docíleno mapováním a převodem datových typů z jednoho do druhého systému.

Hlavní výhodou ORM je vůbec možnost provést automatické mapování relačních záznamů na objekty. Celkově se tím snižuje náročnost celého vývoje, protože jednoduché dotazy není ani třeba psát v jazyku SQL. Díky tomu je vývoj celého systému mnohem rychlejší. Většina ORM nástrojů má v sobě také podporu cachování. Jde o udržování si již načtených objektů v paměti aplikace, a tím zlepšení rychlosti provádění jednotlivých požadavků [40].

2.4.3.1 Vzory mapování dat

Pro mapování dat existuje několik návrhových vzorů. Mezi ty, které frameworky většinou používají, patří především Active Record a Database Mapper.

Kromě atributů daného objektu přidává Active Record do entitní třídy metody obsluhující komunikaci s databází. Každý záznam je tedy zodpovědný za ukládání a načítání dat. Byznysová logika se do této třídy přidává také.

```php
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue
     */
    private int|null $id = null;
    /**
     * @ORM\Column(type="string")
     */
    private string $name;
}
```

Ukázka 2.7: Entita v knihovně Doctrine

Data Mapper místo toho extrahuje databázové operace mimo samotnou entitu a ponechává v ní pouze atributy a metody byznysové logiky. Tento objekt, starající se o mapování mezi relačním a objektovým světem, má na starost kromě přenosu dat také jejich oddělení. Ani jedna z vrstev nemusí o té druhé nic vědět. Dochází tak k lepšímu rozdělení zodpovědností [25].

2.4.3.2 Doctrine

Doctrine je objektovně relační mapovač, který byl vyvinut pro programovací jazyk PHP. Tento nástroj je přímo doporučen ke zvolenému frameworku Symfony. V jádru používá výše zmíněný vzor Data Mapper. Pracuje s konceptem entit, které jsou identifikovatelné podle unikátního identifikátoru. Každá entita je určena danou třídou, která může entitě kromě atributů doplňovat také chování ve formě metod.

Pro správně propojení objektového a relačního světa je možné v Doctrine využít systém anotací. Pomocí nich lze označit, které třídy jsou entity, který z atributů je primární klíč a jaké kardinality mají vazby na jiné entity [41]. Příklad entity v tomto systému je zobrazen na ukázce 2.7.

2.4.3.3 Doctrine query language

Jedna z dalších funkcí, které Doctrine poskytuje, je Doctrine query language (dále DQL). Jde o objektový dotazovací jazyk, který umožňuje abstrakci nad samotnou databází. Díky tomuto způsobu dotazování dokáže Doctrine podporovat různé typy databází bez toho, aby se jakýmkoliv způsobem musel měnit zdrojový kód. Každá databáze má totiž mírně odlišnou syntaxi a pokud by byl dotaz napsán přímo v jazyku SQL, mohlo by při výměně databáze dojít k chybám. Pokud je dotaz položen v DQL, dokáže knihovna sama na základě typu databáze vygenerovat odpovídající dotaz v nativním jazyku.

Nejedná se ovšem o typ jazyka SQL, přestože nelze popřít jistou podobnost. Místo názvů sloupců je třeba dotazovat se přímo nad doménovými objekty za pomocí jejich atributů. Jsou podporovány standardní dotazy jako *SELECT*, *UPDATE* nebo *DELETE*. Existují zde také agregační funkce a aritmetické operace [42]. Jako příklad lze v ukázce 2.8 uvést základní dotaz pro vybrání uživatelů s věkem vyšším než 20.

```
1  $query = $em->createQuery('SELECT u FROM User u WHERE u.age >
   ↪  20');
2  $users = $query->getResult();
```

Ukázka 2.8: Příklad dotazu v DQL

2.4.4 Docker

Distribuovat tuto webovou aplikaci budeme jako kontejner – konkrétně za pomoci platformy Docker. Kontejner je izolovaná výpočetní jednotka, která dává dohromady zdrojový kód a jeho závislosti. Docker byl jeden z prvních nástrojů, který byl ke kontejnerizaci vyvinut, a započal revoluci v tomto odvětví. Dle průzkumu z roku 2019 je jednou z nejrozšířenějších aplikací tohoto typu [43].

Hlavní výhoda kontejnerů oproti konvenčnímu vývoji aplikací je izolovanost a opakovatelnost prostředí. Pokud na daném projektu pracuje více vývojářů najednou, je důležité, aby všichni měli stejně nakonfigurované prostředí – tedy stáhnuté stejné knihovny nebo nastavené všechny služby. Kontejnery přesně tuto funkcionalitu poskytují. Aplikace je spuštěna uvnitř kontejneru vždy za stejných podmínek se stejnou konfigurací. Mimo jiné lze aplikaci ve stejném režimu následně spustit přímo na produkčním serveru.

Kontejnery beží na rozdíl od virtuálního stroje (dále VM) v aplikační vrstvě. VM funguje v porovnání na fyzické vrstvě počítače. Kvůli tomu VM obsahují spoustu věcí, které aplikace nutně ke svému běhu nepotřebuje. V porovnání s nimi jsou kontejnery více odlehčené. Nevyžadují ke svému běhu totiž samostatný operační systém, ale sdílí ho s hostujícím strojem a ostatními kon-

tejnery. Tímto způsobem navíc ušetří znatelné množství místa v paměti, které by jinak kvůli duplikaci bylo obsazeno [44].

Pro jednoduší aplikace stačí použít jeden kontejner, který je definovaný konfiguračním souborem `Dockerfile`. Zde se definuje, ze kterého základního „obrazu" se má kontejner postavit. To může být například distribuce Linuxu nebo databáze. Následně lze provést základní nastavení tohoto obrazu. V tento moment je také možné provést vystavení tohoto kontejneru na určitý port hostujícího stroje.

2.4.4.1 Docker compose

Docker compose je nástroj pro správu aplikací obsahujících více kontejnerů. Nejčastějším důvodem pro použití je případ, kdy je potřeba mít více různých služeb, které spolu nějakým způsobem spolupracují. Každá z těchto služeb je poté izolována, nicméně všechny společně běží na stejném hostiteli.

Konfigurace je spravována jedním hlavním souborem ve formátu YAML `docker-compose.yaml`, ve kterém se definují jednotlivé služby. Pro každou službu lze definovat obraz, ze kterého se spustí. To lze učinit buď pomocí samostatného `Dockerfile`, nebo přímo uvedením základního obrazu. Každá služba následně může být vystavena na určitém portu a provázána s portem nebo rozsahem portů hostitele [45].

Častá konfigurace webových aplikací bude obsahovat webový server, databázi a samotnou aplikaci.

2.5 Návrh databáze

Jedna z metod při návrhu databáze je použití grafické reprezentace navrhovaného schématu. Vizualizací entit a jejich vztahů je dosaženo jednoduššího propojení těchto konceptů s reálným světem [46].

2.5.1 Entity–Relationship model

ER schéma je jeden ze způsobů, jak zachytit základní informace důležité pro projekt nebo organizaci. Jeho cílem je poskytnutí těchto informací během následného vývoje [47]. Tvorba ER modelu probíhá před samotným návrhem databáze. ER model je totiž jednodušší než relační model, nezabíhá do konkrétních detailů a může být použit při komunikaci s klientem. Díky jeho veliké nezávislosti na hardwaru nejsou návrháři nijak omezeni konkrétní implementací. Třemi základními kameny v tomto schématu jsou entity, atributy a vztahy [46, 47].

Konkrétních specifikací notace pro ER modely existuje veliké množství. Pro ER diagramy v této práci byla použita Barkerova notace popsána v jeho knižní sérii *CASE method* [47].

2.5.1.1 Důležité prvky diagramu

Nejdůležitějším objektem v ER modelu je entita. Jedná se o jakýkoliv důležitý objekt, o kterém je potřeba udržovat informace. Je určen svým jménem a atributy. Atributy jsou vlastnosti daného objektu a mohou mít několik příznaků (primární klíč, povinný, nepovinný, ...).

Mezi dvěma entitami jsou definovány vztahy. Ty mají dle tohoto typu notace dvě vlastnosti. První je volitelnost, která určuje, jestli se daná entita účastní vztahu povinně nebo nikoliv. V diagramu je volitelnost reprezentována buď plnou čárou pro povinnost, nebo přerušovanou pro nepovinnost. Druhá vlastnost je kardinalita, která stanovuje u každé entity počet účastníků v rámci tohoto vztahu. Může být definována jako `1:1`, `1:N` nebo `N:N`. V modelu je toto zobrazeno u spoje mezi entitou a vazbou buď jednoduchou čárou pro jednoho účastníka, nebo vidličkou pro N účastníků [47].

2.5.2 Návrh ER modelu

Vzhledem k velikosti navrženého ER modelu je zde prezentována pouze jeho část. Jako názorná ukázka byla zvolena oblast přihlašování na kroužky zobrazená na obrázku 2.2. U všech následujících entit budou kromě níže zmíněných atributů evidovány také umělé identifikátory, které budou sloužit jako primární klíče. Kompletní diagram lze nalézt v příloze A.

2.5.2.1 Přihláška

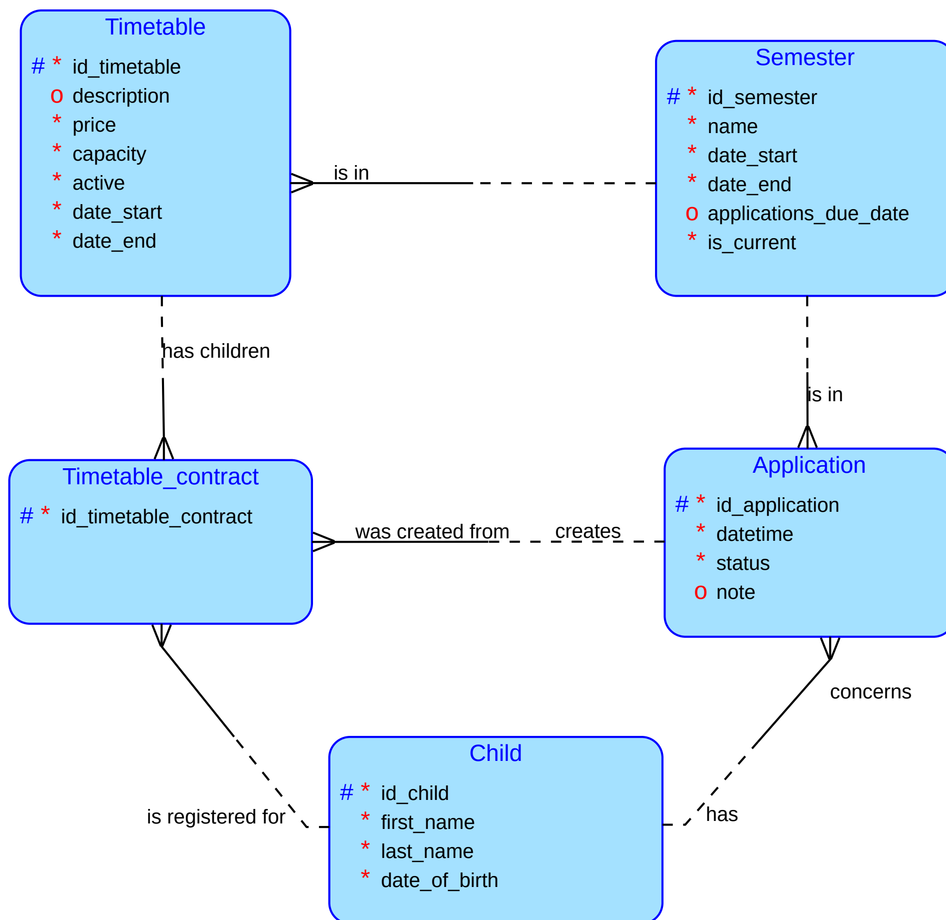
Přihlášku bude reprezentovat entita `Application`. Je potřeba o ni znát především datum, kdy byla podána, a její aktuální stav (čeká na zaplacení, aktivní, zrušená). Každá přihláška patří povinně do určitého semestru a může jich být více. Současně na základě přihlášky vznikne vztah mezi dítětem a rozvrhem, který symbolizuje samotné přihlášení.

2.5.2.2 Semestr

Pro zachycení časového úseku, na kterém jsou definované kroužky, bude zavedena entita semestr – `Semester`. Ta bude udržovat informace jako pojmenování, datum začátku a datum konce tohoto období. Dále také datum, do kdy v časovém úseku bude třeba podat přihlášky. V případě, že nebude vyplněno, bude možné přihlášky podávat bez časového omezení. Posledním atributem bude příznak toho, zda daný semestr aktuálně probíhá nebo ne. Ovšem pouze jeden semestr v dané škole může být ten aktuální.

2.5.2.3 Dítě

Účastníka kroužku `Child` není třeba modelovat jako uživatele, neboť se k systému přihlašovat nebude. Stačí o něm evidovat základní informace, jako je jméno, příjmení, datum narození a jeho rodné číslo.



Obrázek 2.2: ER model zúžený na přihlášky

2.5.2.4 Rozvrh

Realizací kroužku, na kterou se půjde přihlásit, bude rozvrh – `Timetable`. Bude definován v daném semestru. V jednom semestru jich může existovat libovolné množství. O rozvrhu je třeba znát jeho cenu, kapacitu a zda je aktivní – tedy jestli je nabízen k přihlašování. Libovolně k němu bude možné přiřadit i popis.

2.5.2.5 Přiřazení dítěte k rozvrhu

Jelikož mezi rozvrhem a dítětem vznikla vazba N:N, a o tomto vztahu chceme ukládat dodatečné atributy, bude potřeba ho dekomponovat na dva vztahy 1:N a novou entitu. Vzniklá vztahová entita bude silná – bude určená svým umělým identifikátorem [47]. Ten bude potřeba především kvůli entitě docházky, která bude sama o sobě vztahovou entitou (ta nicméně není v této ukázce

zobrazena). Vznikne tedy entita `ChildToTimetable`, ve které zmíněným dodatečným atributem bude mj. přihláška, ze které byl tento vztah odvozen.

Kapitola 3

Realizace

3.1 Použité nástroje pro vývoj

3.1.1 Vývojové prostředí

Jako integrované vývojové prostředí (dále IDE) byla zvolena aplikace PhpStorm. Umožňuje současně pracovat na několika různých typech souborů (php, html, css, js) a všechny je zvládá syntakticky a sémanticky analyzovat. Při vývoji pak zobrazuje odpovídající nápovědy. K dispozici je také tzv. „refaktorování“. Jde o možnost například přejmenovat třídu nebo změnit předpis metody globálně napříč celou kódovou základnou. Kromě samotné úpravy zdrojového kódu je také možné k IDE připojit databázi. Při napojení vývojové databáze lze pak velmi snadno manipulovat s testovacími daty [48].

Pro vývoj aplikace ve frameworku Symfony je možné rozšířit IDE pomocí pluginu Symfony Support. Okamžitě po instalaci je dostupných hned několik funkcionalit:

- Při psaní dotazů prostřednictvím Doctrine je zobrazována nápověda na názvy atributů entit.
- Je přidána nápověda pro automatické dokončování překladových klíčů.
- Umožněno je také okamžité zobrazení služeb, které jsou definovány hluboko uvnitř konfiguračních souborů [49].

Všechny výše zmíněné funkce podstatně zrychlují a zjednodušují vývoj v tomto frameworku.

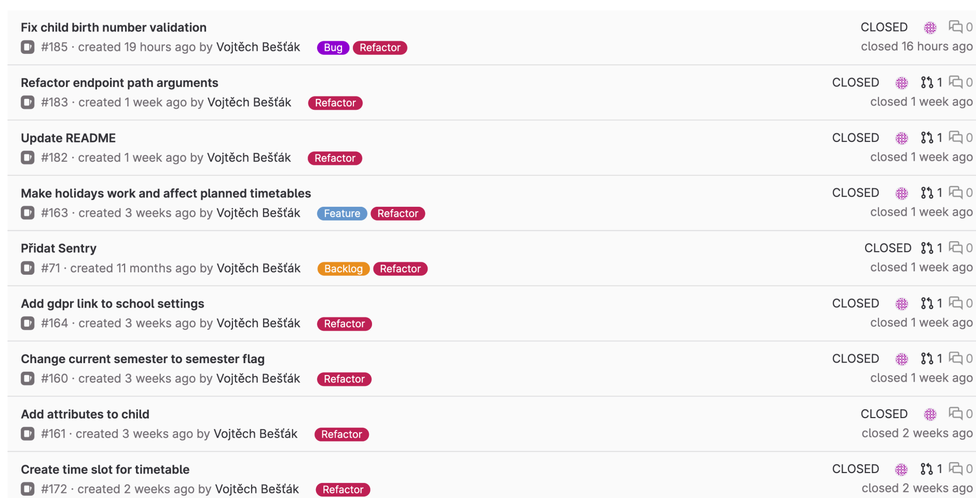
3.1.2 Git a Gitlab

Pro udržování historie projektu byl použit systém pro správu verzí Git. Jde o nástroj, který umožňuje nejen zálohovat zdrojový kód, ale současně usnadňuje paralelní práci více vývojářů na stejném projektu. Ty jsou v prostředí

Gitu označovány jako repozitáře. Oddělení práce je dosaženo prostřednictvím různých verzí – tzv. větví. Dále na rozdíl od dřívějších VCS není Git centralizovaný, nýbrž distribuovaný, a každý uživatel tak má lokální kopii celého zdrojového kódu [50].

V tomto projektu byl praktikován pracovní postup, který by byl využitelný i ve větším týmu programátorů. V hlavní větvi `main` je zdrojový kód, který je určený pro produkci. Větev `dev` slouží k uchování verze, na které se aktuálně pracuje a která obsahuje nejnovější dokončené funkcionality. Následně je pro každou novou funkcionalitu nebo opravu vytvořena samostatná větev z `dev`, která je po dokončení zapojena zpět. Tímto je docíleno požadovaného výsledku oddělení práce jednotlivých členů týmu.

Aby byl zdrojový kód dostupný online, je třeba mít své lokální repozitáře provázané s těmi vzdálenými. Jedním z poskytovatelů těchto vzdálených repozitářů založených na Gitu je GitHub nebo GitLab [50, 51]. Kromě kontroly verzí poskytuje mj. možnost spravovat úkoly a vykonávat automatické testy [51]. Obě tyto funkcionality byly při vývoji často využívány. Na obrázku 3.1 je snímek obrazovky zachycený v průběhu vývoje, který zobrazuje již vyřešené úkoly uvnitř tohoto systému.



Obrázek 3.1: Úkoly nahlášené do systému GitLab

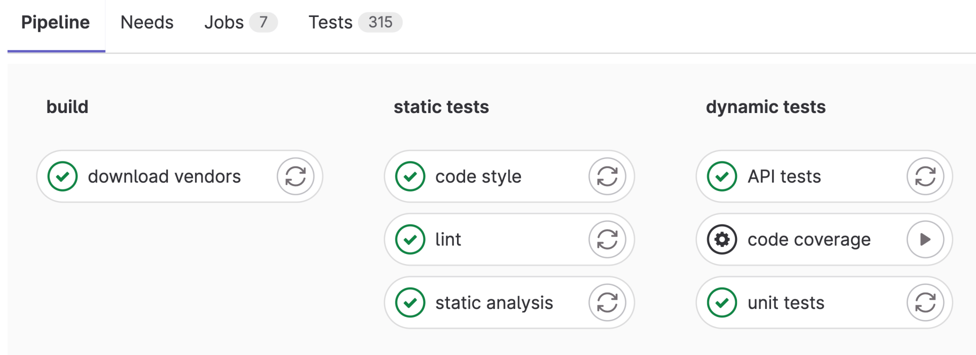
3.1.2.1 Kontinuální integrace a nasazení

Kontinuální integrace (zkráceně CI) je způsob vývoje, kde vývojáři mohou delegovat manuální, opakující se a časově náročné procesy výpočetnímu serveru. Ten buď pravidelně, nebo v systému Git po každém commitu, spustí předem definovanou sérii skriptů. Ta má za úkol ověřit funkčnost a správnost nahraného zdrojového kódu. Sekvence může obsahovat například sestavení projektu, spuštění statické analýzy nebo provedení dynamických testů.

Kontinuální nasazení (zkráceně CD) pracuje na stejné myšlence jako CI, ale navíc definuje možnost automatického nasazení. Při vydání nové verze softwaru tedy CD automaticky zařídí všechny potřebné úkony pro její nasazení. Hlavním cílem při vytvoření tohoto postupu byla eliminace lidské chyby a možná rychlost celého procesu.

V tomto projektu byl jako výpočetní server použit GitLab CI, který umožňuje definovat sérii úkonů pomocí jednotlivých rour (v angličtině „pipeline"). Pipeline definuje několik skriptů, které musí být provedeny. Mohou být navázané například na jednotlivé větve v Gitu. Při nahrání zdrojového kódu na GitLab se na základě těchto podmínek vyhodnotí, která pipeline se má spustit. Pokud ovšem jakákoliv z částí skončí chybou, celá pipeline selže [52].

Jako součást CI bylo v tomto projektu definováno automatické sestavení, kontrola stylů, statické testy, API a jednotkové testy. Všechny tyto úkony se tedy provedou při každém nahrání na server. Nabízí se zde možnost v budoucnu pomocí GitLab CI definovat i CD. Na obrázku 3.2 je vidět úspěšně provedená pipeline po nahrání jedné z verzí zdrojového kódu v průběhu vývoje na GitLab.



Obrázek 3.2: Úspěšně provedená pipeline ve službě GitLab CI

3.1.3 Toggl Track

Pro zpětnou analýzu pracnosti jednotlivých částí projektu jsem se rozhodl všechny činnosti časovat pomocí nástroje Toggl Track. Ten umožňuje na daném projektu zaznamenávat strávený čas s popisem řešeného problému a dovoluje také přidávat štítky, které dále specifikují danou aktivitu. Následně pak poskytuje přehledy o vykázaném čase prostřednictvím webového rozhraní. Vytvořená data je možné kdykoliv exportovat například ve formátu CSV.

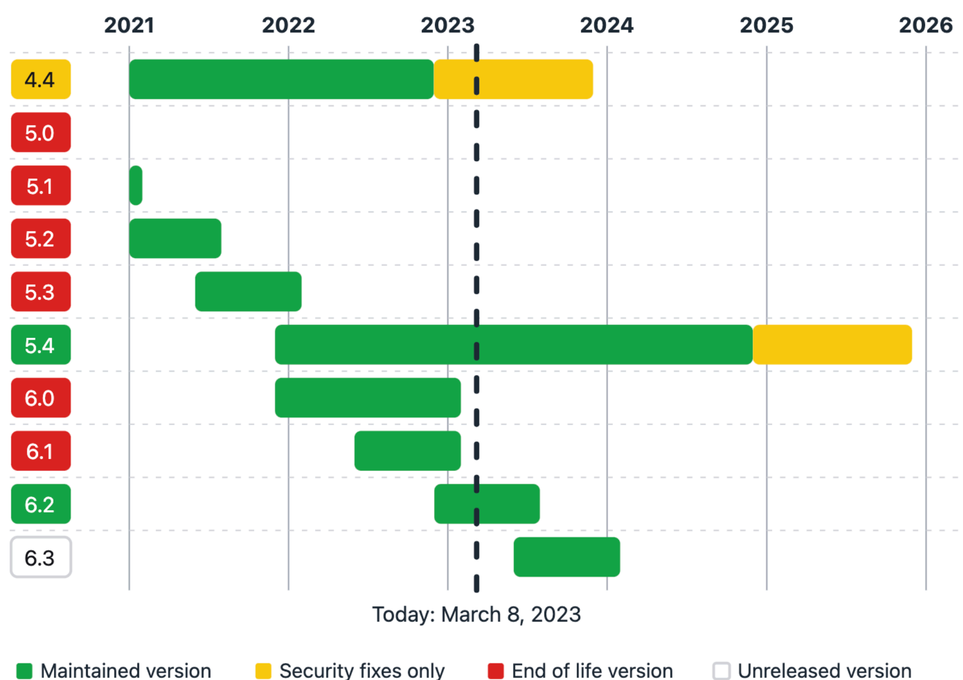
Celý tento pracovní postup může být výhodný zejména při vývoji softwaru. K názvu časového záznamu jsem v případě implementační aktivity zařadil i číselný odkaz na právě řešený úkol ve službě GitLab. Na základě takto posbíraných dat bude následně možné zpřesnit odhady na dalších podobných

projektech. Pokud by byl tento projekt vyvíjen pro klienta nebo v rámci zaměstnání, mohla by být data využita k vykázání odvedené práce [53].

3.2 Konfigurace prostředí

3.2.1 Instalace Symfony

Jak bylo zvoleno výše v podkapitole 2.4.1.6, jako framework bude použito Symfony. Před instalací je nicméně třeba zvolit verzi, kterou použijeme. Jak lze vyčíst z kalendáře verzí na obrázku 3.3, verze s dlouhodobou podporou je `5.4`. Bude po následující téměř dva roky plně udržovaná a následný rok bude dostávat bezpečnostní záplaty [54]. Sice nebude možné využít nové funkcionality frameworku, nicméně z hlediska udržitelnosti a stability je o něco starší verze bezpečnější volba.



Obrázek 3.3: Kalendář verzí frameworku Symfony, získáno z `https://symfony.com/releases`

Pro instalaci Symfony na s touto verzí stačí pak použít příkaz v ukázce 3.1. Ke správnému provedení je třeba mít již v instalačním prostředí připravené PHP a správce závislostí Composer. Příkaz vytvoří základní strukturu projektu a současně stáhne základní knihovny potřebné pro běh aplikace [55].

```
1 symfony new my_project_directory --version="5.4.*"
```

Ukázka 3.1: Instalace frameworku Symfony

3.2.2 Docker compose

Pro distribuci aplikace byl v podkapitole 2.4.4.1 zvolen Docker Compose. Framework Symfony nabízí jednoduchou integraci s touto službou. [56] Při prvotní konfiguraci je třeba pouze specifikovat, které služby budou pro běh systému potřeba.

3.2.2.1 Vývojová databáze

Nejdříve definujme vývojovou databázi. Ta bude mít samostatnou službu. Jako základní obraz zde bude použit již dříve zvolený PostgreSQL. Navíc databázi nastavíme znakovou lokalizaci na češtinu a časovou zónu na Prahu.

Je zde uvedeno, že se jedná pouze o vývojovou databázi. Je to kvůli tomu, že není pro nasazení příliš vhodná. Po zániku kontejneru, kde tato databáze běží, zaniknou i všechna data v ní uložená. Tomu by se dalo předejít, pokud by data byla ukládána mimo kontejner. V ten moment by ale přestalo být uskupení kontejnerů bezstavové. Pokud by tedy při nečekané události došlo k pozastavení stavového kontejneru, mohlo by dojít ke ztrátě dat. Databáze běžící přímo na serveru (tedy mimo kontejner), kde bude aplikace nasazená, umožní jednodušší perzistenci dat bez další režie a současně zachová bezstavovost kontejnerů [57].

3.2.2.2 Webový server

Přijmutí HTTP požadavků, jeho zpracování, zvolení správné služby, která požadavek vyřídí, a následné odeslání odpovědi zařizuje webový server [58]. Volba toho správného není příliš kritická a velmi snadno se dá případně později změnit. Pro jednoduchost konfigurace jsem proto použil server Nginx [59]. Opět jej definujeme jako samostatnou službu.

3.2.2.3 Aplikace

Aplikace jako taková poběží uvnitř služby, jejíž základem bude obraz obsahující PHP. Jak tato služba v konfiguračním souboru vypadá, je uvedeno v ukázce 3.2. Vytvoření obrazu je delegováno do samostatného `Dockerfile`. Je zde provedeno mapování portů a definice propojených souborů s hostitelským operačním systémem. Současně je specifikováno, na jakých jiných službách aplikace závisí.

```
version: "3.6"

services:
    php:
        build:
            context: .docker/php-fpm
            dockerfile: Dockerfile
        container_name: "backend_php"
        hostname: "backend_php"
        networks:
            - internal
        ports:
            - "9000:9000"
        volumes:
            - .:/var/www/app
            - .docker/php-fpm/php.ini:/usr/local/etc/php/php.ini
            - .docker/php-fpm/xdebug.ini:/usr/local/etc/php/conf.d/xdebug.ini
        depends_on:
            - postgres_dev
```

Ukázka 3.2: Konfigurace služby pro aplikaci v souboru docker-compose.yml

3.2.3 Composer

Framework Symfony využívá pro správu knihoven a balíčků správce Composer. Pro každý projekt řídí na základě konfiguračního souboru `composer.json` všechny jeho závislosti. Ty je možné definovat pomocí jejich jména a verze. Specifikaci verze lze provést několika způsoby. Kromě uvedení konkrétní verze (`1.0.0`) existuje několik různých omezení, která dovolují určitou volnost. Ať už jde o jasné omezení verzí (`>=1.0 <2.0`), tak o použití divokých karet (`1.0.*`). Díky tomu je možné pomocí jediného příkazu `composer update` všechny závislosti aktualizovat.

Aby bylo možné udržovat přehled o aktuálně používaných verzích, generuje Composer automaticky soubor `composer.lock`. Uvnitř jsou uvedeny všechny použité knihovny společně s jejich konkrétními nainstalovanými verzemi. Závislosti definované v tomto souboru je možné jednoduše aplikovat pomocí příkazu `composer install` [60].

3.3 Implementace

Opět vzhledem k rozsáhlosti implementační části projektu bude pohled v této sekci zúžen pouze na přihlášky.

3.3.1 Entity

Aby bylo možné implementovat byznysovou logiku, je nejdříve potřeba vytvořit doménové modely, které v ní budou využity. Jak bylo zmíněno v sekci 2.4.3.2, Symfony pro objektově relační mapování používá knihovnu Doctrine. Pro přihlášku tedy bude existovat třída `Application.php`, která bude mít jako atributy identifikátor, datum a čas vzniku, volitelnou poznámku, aktuální stav a vazby na další entity – konkrétně semestr, do kterého byla podána, dítě, kterého se týká, a rozvrhy, na které byla podána. Ukázka 3.3 zobrazuje několik prvních atributů společně s jejich anotacemi. Tyto anotace zajišťují mapování modelu na databázové schéma.

```
1  /**
2   * @ORM\Entity(repositoryClass=ApplicationRepository::class)
3   */
4  class Application
5  {
6      /**
7       * @ORM\Id
8       * @ORM\GeneratedValue(strategy="SEQUENCE")
9       * @ORM\SequenceGenerator(⌋
        ↪  sequenceName="application__id_application_seq")
10      * @ORM\Column(type="integer", name="id_application")
11      */
12     private int $id;
13     /**
14      * @ORM\Column(type="datetime")
15      */
16     private Carbon $createdAt;
17     /**
18      * @ORM\ManyToOne(targetEntity=Semester::class,
        ↪  inversedBy="applications")
19      * @ORM\JoinColumn(name="id_semester",
        ↪  referencedColumnName="id_semester", nullable=false)
20      */
21     private Semester $semester;
```

Ukázka 3.3: Doménový model přihlášky implementovaný pomocí Doctrine

3.3.2 Služby

Pro práci s touto entitou je potřeba vytvořit službu, která bude shromažďovat byznysovou logiku. Ukázka 3.4 tuto službu `ApplicationService` zobrazuje.

Nejdříve je vytvořena nová instance přihlášky, do které jsou postupně nastavena data dodaná v těle požadavku (jak s nimi pracuje Symfony je popsáno v podkapitole 3.3.4). Nicméně to pro uložení dat do databáze nestačí. To zajišťuje až třída `EntityManagerInterface`.

Nejdříve je třeba sdělit frameworku, že je v plánu uložit instanci přihlášky do databáze pomocí metody `persist`. V ten moment začne být tento objekt sledován ze strany Doctrine. Dosud ovšem nejsou vykonány žádné dotazy na databázi. To se stane až v okamžiku, kdy je zavolána metoda `flush`. Ta teprve všechny připravené dotazy odešle k vyhodnocení [61].

Pro složitější sekvence práce s databází je možné využít explicitní specifikaci transakcí. Ty jsou uvnitř tohoto frameworku opět realizovány na zmíněném rozhraní. Pomocí metody `beginTransaction` lze transakci započít a následně provádět nad databází potřebné změny. Pokud v určitém bodu dojde k nějaké chybě a je třeba všechny změny vrátit, stačí zavolat `rollback`. Pokud vše proběhne dle plánu a transakci je možné zachovat, lze ji ukončit pomocí metody `commit` [62].

```
1  class ApplicationService extends AbstractService
2  {
3      public function create(
4          CreateApplicationRequest $request,
5          School $school
6      ): Application {
7          $this->entityManager->beginTransaction();
8
9          $application = new Application();
10         $application->setDatetime(Carbon::now());
11         $application->setNote($request->note);
12         $application->setApplicationStatus(⌟
           ↪  ApplicationStatus::WaitingForPayment);
13         ...
14         $this->entityManager->persist($application);
15         $this->entityManager->flush();
16         $this->entityManager->commit();
17         return $application;
18     }
19     ...
```

Ukázka 3.4: Služba shromažďující logiku týkající se přihlášek

3.3.3 Controller

Protože koncové body jsou v aplikaci poskytující REST API uživatelským rozhraním, budou dle architektury frameworku umístěné v Controllerech (detailněji popsáno v podkapitole 2.3.2.1). Do těchto tříd budou rozděleny na základě toho, čeho se daný koncový bod týká. Každá metoda uvnitř controlleru reprezentuje jeden koncový bod. Skupina navrhnutých endpointů v podkapitole 2.2 se zaměřuje na přihlášky, a proto se třída v ukázce 3.5 jmenuje `ApplicationController`.

Vyobrazena je implementace koncového bodu, který byl navrhnut v podkapitole 2.2.1.1. Prostřednictvím anotace je definována HTTP metoda a cesta, na které bude možné tento bod zavolat. Framework pak automaticky zajistí zavolání této funkce s předanými parametry.

Uvnitř samotné metody je nejdříve ověřena existence požadované školy, ve které by přihláška měla vzniknout. Pokud zdroj neexistuje, bude jako odpověď na požadavek odeslána chyba společně s krátkou zprávou. Samotný systém pro zpracování chyb je detailněji popsán v sekci 3.3.5.

Součástí požadavku přijde i jeho tělo. Použitím Symfony formulářů (které jsou popsány v sekci 3.3.4) jsou z požadavku získána data a je ověřena jejich validita.

Pomocí služby, která shromažďuje byznysovou logiku, je vytvořena nová přihláška. Pokud proces proběhne bez problémů, jako výsledek je obdržena vytvořená a uložená přihláška. Ta je použitím pomocné třídy převedena do pole a jako odpověď je vrácena i s odkazem na koncový bod, na kterém je možné si ji zobrazit. Framework už sám zajistí, že vrácené pole bude převedeno do formátu JSON.

3.3.4 Formuláře

Pro zpracování těla požadavku a jeho validaci je doporučené v Symfony použít formuláře. Jde o třídy definující atributy, které mohou být použity v požadavku. O každém atributu je možné specifikovat, jak má být pojmenovaný a jakého má být typu. Tímto způsobem dojde k oddělení logiky a k možnosti znovu použít formulář na více místech [63]. Příklad formuláře je zobrazen v ukázce 3.6.

Formulář je tedy použit jako předpis pro naplnění požadovaného objektu daty z požadavku. Tímto objektem může být buď entita definovaná Doctrine, nebo separátní třída, jejíž jediný úkol je předávat a validovat tato data.

Pokud by byly použity entity, došlo by k narušení rozdělení definované třívrstvou architekturou, která byla popsána v podkapitole 2.3.1.1. Prezentační třídy by totiž závisely na datové vrstvě. Může také dojít k tomu, že se změní požadavky na aplikaci, a součástí požadavku by měl být nový atribut, který ale v databázi být nemá. Proto pokud hned od začátku budou správně použity

```
class ApplicationController extends AbstractController
{
    /**
     * @Rest\Post("/api/schools/{schoolId}/applications",
     *     requirements={"schoolId" = "\d+"},
     *     name="application_application_post")
     */
    public function createApplication(
        Request $request,
        int $schoolId
    ): View {
        $school = $this->schoolService->getById($schoolId);
        if ($school === null) {
            $this->sendNotFound(
                'school.read.failure',
                $schoolId
            );
        }

        $formRequest = new CreateApplicationRequest();
        $form = $this->createForm(
            CreateApplicationForm::class, $formRequest
        );
        $form->submit($request->request->all());
        if (!$form->isValid()) {
            $this->sendFormErrors($form);
        }

        $application = $this->applicationService->create(
            $formRequest, $school
        );
        return $this->sendCreated(
            $this->applicationService->serialize($application),
            locationRoute: 'application_application_get',
            locationParams: [
                'applicationId' => $application->getId()
            ]
        );
    }
...
}
```

Ukázka 3.5: Controller obsahující koncový bod pro podání přihlášky

```php
class CreateApplicationForm extends AbstractForm
{
    public function buildForm(
        FormBuilderInterface $builder,
        array $options
    ): void
    {
        $builder->add(
            'child',
            CreateApplicationChildForm::class
        );
        $builder->add('note', TextType::class);
        ...
    }

    protected function getDataClassName(): string
    {
        return CreateApplicationRequest::class;
    }
}
```

Ukázka 3.6: Formulář pro vytvoření přihlášky

oddělené datové třídy, bude aplikace jednodušeji reagovat na změny [64]. Navíc lze při složitějším požadavku tyto datové třídy do sebe vrstvit.

Pro validaci jednotlivých atributů je možné použít anotace, které jsou opět součástí Symfony. Omezení mohou mít například podobu jednoduchého vynucení vyplněnosti `\NotBlank` nebo `\NotNull`, u čísel vyžádání kladnosti `\Positive` nebo u dat definice porovnání `\LessThan(...)` [65]. Příklad použití je ukázán na datovém objektu pro vytvoření přihlášky v ukázce 3.7.

3.3.5 Odchytávání chyb

Oddělení úspěšné a chybové cesty ve zdrojovém kódu lze jednoduše dosáhnout využitím výjimek. Prozatím si projekt vystačí s použitím výjimek, které jsou poskytnuté z frameworku Symfony. Mezi tyto výjimky patří například `NotFoundHttpException` nebo `BadRequestHttpException`. Obecná výjimka `HttpException` je rodičem obou zmíněných.

Aby framework zobrazoval chyby jednotným způsobem, je třeba zavést službu, která bude výjimky odchytávat a modifikovat výstup zobrazovaný uživateli. Při obdržení výjimky je pouze zkontrolováno, zda ji lze úspěšně zpracovat. Následně je vytvořena zpráva ve formátu JSON, která obsahuje kód chyby, krátkou zprávu popisující problém a případně další volitelné atributy.

```
class CreateApplicationRequest
{

    /**
     * @Assert\NotNull
     * @Assert\Valid
     */
    public CreateApplicationChildRequest $child;

    public ?string $note = null;

    ...
}
```

Ukázka 3.7: Datový objekt pro vytvoření přihlášky

Je tím tedy splněn požadavek na neúspěšnou odpověď popsanou v návrhu koncového bodu v ukázce 2.3.

3.3.6 Stránkování, filtrování a řazení

Pro stránkování, filtrování a řazení jsem se rozhodl použít knihovnu, kterou jsem vytvořil v rámci jednoho z předmětů v průběhu studia. Jedná se o balíček, který je určený specificky pro framework Symfony a ke svému fungování využívá knihovnu Doctrine [66].

3.3.6.1 Stránkování

Stránkování slouží k rozdělení velikého souboru dat na menší části – stránky. Zmenšením požadovaných dat se zrychlí vyhodnocení celého dotazu [24]. Pokud by byla data stránkována až v klientské aplikaci, nedalo by se využít rychlosti databázového stroje. Ten totiž dokáže vrátit pouze požadovaný rozsah dat a nebude plýtvat výpočetním časem.

Uživateli by mělo být umožněno, aby si definoval, jak veliká má tato stránka být. V této knihovně je možné předat stránkovací detaily přímo v cestě požadavků pomocí „query“ parametrů:

- `page`: číslo požadované stránky,
- `per_page`: vyžadovaný počet položek na jedné stránce.

Před vyhodnocením samotného dotazu jsou do něj tyto parametry nejdříve zakomponovány a následně společně vykonány [66].

3.3.6.2 Filtrování a řazení

Filtrování dat přímo v serverové části aplikace má podobné výhody jako stránkování. Místo toho, aby byla v klientské aplikaci složitě implementována logika pro filtrování, lze využít možností databázového stroje, který tyto funkcionality má již v základu a provede výpočet mnohem efektivněji.

Značnou část knihovny proto tvoří logika pro filtrování a řazení výsledků. Prvním krokem je získání struktury filtru a nastavení řazení. Stejně jako u stránkování je využito následujících „query“ parametrů:

- `filter`: požadovaný filtr, kde jeden výraz je ve tvaru *<metoda>:<atribut>:<hodnota>*,
- `sort`: požadované řazení ve tvaru *<field>:[asc,desc]*.

Aby bylo možné specifikovat komplexnější filtry, podporuje knihovna v tomto parametru pravdivostní logiku pomocí logických operátoru AND a OR. Pro oddělení jednotlivých výrazů je také možné využít závorek. Algoritmus následně musí tento řetězec zpracovat a sestavit dotaz zohledňující požadované omezení. Popis tohoto procesu je však nad rozsah této práce, a proto ho zde vynechám. Dotaz je možné seřadit podle zadaného parametru, a to buď sestupně, nebo vzestupně [66].

Jak filtrování, tak řazení je přiřazeno k dotazu těsně před jeho vyhodnocením, čímž je opět docíleno větší efektivity provedení příkazu.

3.3.6.3 Použití knihovny

Použití této knihovny je velmi přímočaré. Zobrazeno je na ukázce 3.8. Stačí vytvořit instanci objektu `PaginationHandler`. Tomu je třeba definovat všechny atributy, pomocí kterých bude možné tento koncový bod filtrovat. Zde musí dojít k mapování názvů atributů přijímaných z požadavků na jeho název v DQL dotazu.

Předáním základního dotazu tomuto objektu dojde k celkovému vyhodnocení a výsledná data jsou navrácena zpět. O aktuálně získaných datech bude uživatel informován prostřednictvím odpovědi, která bude mít speciální hlavičku obsahující číslo aktuální stránky, počet položek na stránce a celkový počet existujících dat vyhovujících tomuto dotazu [66].

3.4 Dokumentace

Dokumentace pro aplikační rozhraní je jedna z nejdůležitějších částí návrhu serverové aplikace. Je to totiž zdroj pravdy pro všechny vývojáře implementující toto rozhraní. Hlavní součástí dokumentace je popis jednotlivých koncových bodů. O každém z nich je sledována jeho cesta, HTTP metoda, krátký

```
$paginationHandler = new PaginationHandler($paramFetcher);
$paginationHandler->createQueryFilter()
    ->addNumberField("id", "a.id")
    ->addDatetimeField("datetime", "a.datetime")
    ->addTextField("status", "a.applicationStatus")
    ->addDefaultSort("datetime", SortField::DESC);
$query = $this->courseService
    ->getAllBySemesterFilterQuery($semester);
$applications = $paginationHandler->getPaginatedData($query);

$result =
↪   $this->applicationService->serializeMany($applications);
$paginatedResult = $paginationHandler
    ->sendPaginatedResponse($result);
```

Ukázka 3.8: Implementace stránkovaného, filtrovaného a řazeného koncového bodu

popis, formát volitelného těla požadavku a možné návratové hodnoty. Ty mohou následně specifikovat popis a formát těla odpovědi.

Webová aplikační rozhraní bývají často popisována ve standardu OpenAPI. Jde o formát, který byl vyvinut přímo pro specifikaci REST aplikačních rozhraní. Jeho hlavním účelem je umožnit vývojářům implementaci tohoto rozhraní bez znalosti implementačních detailů aplikace.

Swagger je ekosystém služeb, které mají za cíl zjednodušit vývojářům práci na aplikačních rozhraních. Pro zobrazení a rychlé úpravy lze použít službu Swagger Editor. Ta umožňuje navrhovat specifikaci v OpenAPI a současně sledovat generovaný výsledek, který je uživatelsky přívětivější. Pro drobné úpravy specifikace je tato služba nejvhodnější.

Další užitečnou službou, která je využita i v tomto projektu, je Swagger UI. Umožňuje vizualizovat aplikační rozhraní pouze na základě dodané specifikace a okamžitě s ním interagovat (například je zde možnost autorizovat se a odesílat požadavky).

Díky tomu, že je standard dobře čitelný jak pro lidi, tak pro stroje, je možné na základě této dokumentace generovat pomocí služby Swagger Codegen automaticky zdrojový kód pro klientské aplikace. Vygenerovaný zdrojový kód poskytuje připravené doménové objekty a dostupná volání na definovanou doménu prostřednictvím HTTP komunikace na základě platformy [24]. Mezi podporované programovací jazyky patří například Java, Kotlin, Swift nebo JavaScript [67].

```
use OpenApi\Annotations as OA;
/**
* @OA\Post(operationId="application_application_post")
* @OA\RequestBody(ref="#/components/requestBodies/CreateApplicationRequest")
* @OA\Response(response=201, description="Application created successfully", @OA\JsonContent(ref="#/components/schemas/Application"))
* @OA\Response(response=400, description="Application could not be created", @OA\JsonContent(ref="#/components/schemas/BadRequest"))
* @OA\Response(response=403, description="Access denied")
* @OA\Response(response=404, description="School not found")
*/
public function createApplication(Request $request, int $schoolId)
```

Ukázka 3.9: Specifikace dokumentace endpointu ve zdrojovém kódu

3.4.1 Automatická generace dokumentace

Pro framework Symfony je dostupná knihovna NelmioApiDocBundle. Ta umožňuje automaticky generovat dokumentaci na základě napsaného zdrojového kódu. Aby bylo možné tento převod uskutečnit, každý koncový bod musí pomocí anotací specifikovat základní informace. Mezi ty patří název, krátký popis, možné návratové hodnoty a struktura těl požadavků a odpovědí. Na ukázce 3.9 je zobrazena dokumentace pomocí těchto anotací k endpointu z ukázky 3.5.

Definice modelů, zmíněných v možných odpovědích u endpointu, je prováděna prostřednictvím konfiguračního souboru ve formátu YAML, jehož struktura je velmi podobná standardu OpenAPI. Jeden z těchto modelů je zobrazen v ukázce 3.10.

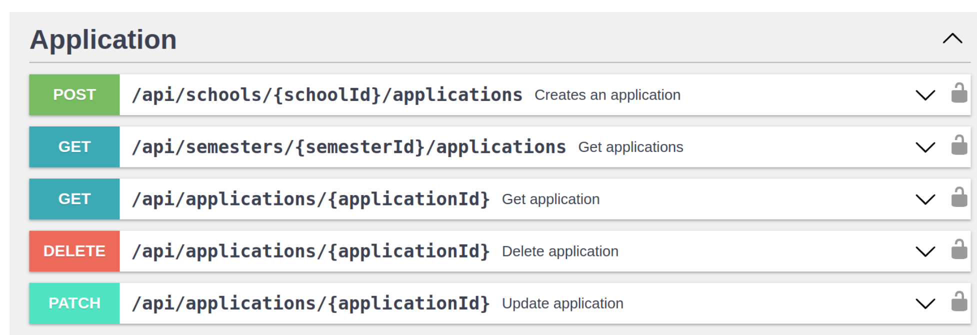
Pro vystavení této dokumentace přímo v aplikaci lze určit cesty, na kterých bude OpenAPI specifikace dostupná. Knihovna také podporuje vizualizaci dokumentace pomocí Swagger UI, které bylo více popsáno v předchozí podkapitole 3.4 [68]. Vygenerovaný výsledek pro soubor endpointů pro přihlášky je vidět na obrázku 3.4.

3.5 Testování

Testování softwaru je proces určený určený k identifikaci chyb a defektů v daném systému. Má sloužit k posouzení aktuálního stavu a kvality daného pro-

```
Application:
    description: Application response object
    type: object
    properties:
        id:
            type: integer
            example: 42
            description: Application identifier
        due_date:
            type: string
            example: 2021-08-12
            description: The date when the payment for the
            ↪  application is due.
```

Ukázka 3.10: Konfigurace modelu přihlášky v knihovně NelmioApiDocBundle



Obrázek 3.4: Dokumentace vizualizovaná pomocí SwaggerUI

duktu. Na základě těchto dat mohou být vyhodnocena rizika a splnitelnost případných termínů [69].

Cílem testování je především ověření splněnosti požadavků kladených na produkt. Dalším důležitým záměrem testování je nalezení chyb a závad ještě před nasazením systému do produkčního prostředí. Oprava chyby během vývoje je mnohonásobně levnější a rychlejší, než kdyby byl tento problém nalezen až později [70]. Pro samotné vývojáře pak testy mohou sloužit k tomu, že se nemusejí obávat úprav zdrojového kódu. Je-li totiž kód pokrytý testy, mohou si být jisti, že jejich případný zásah do tohoto zdrojového kódu nezpůsobí chybu a nebude mít neočekávaný efekt.

3.5.1 Statické a dynamické testování

Testování se dá rozdělit na statické a dynamické. Hlavním rozdílem mezi nimi je to, že pro vyhodnocení dynamických testů je zdrojový kód potřeba skutečně

vyhodnotit. Na základě vstupních hodnot je tento kód spuštěn a jeho výsledky jsou následně kontrolovány.

Oproti tomu statické testování žádný zdrojový kód nevyhodnocuje. Kontrola se provádí buď manuálně, nebo s použitím různých nástrojů, a může mít několik různých podob. První je možnost před spuštěním zdrojového kódu provést statickou analýzu, která pomocí předdefinovaných pravidel odhalí například nedosažitelný zdrojový kód nebo problematickou logiku. Druhá forma statického testování, která se nabízí především ve větších týmech, je strukturované skupinové přezkoumání [70]. Jedna z možných podob je technická revize zdrojového kódu. Jde o aktivitu prováděnou s cílem ohodnotit danou část softwaru [69].

3.5.2 Metody

Pro testování se většinou používá buď přístup Black box, nebo White box. Jedná se o metody definované na základě toho, jak moc informací má tester o samotné implementaci.

Při použití přístupu Black box nejsou testerovi známy žádné implementační detaily zdrojového kódu, který je testován. Jediné, co tester má k dispozici, jsou informace o tom, co má testovaný subjekt dělat. Specifikace tohoto chování vychází z analýzy a byznysových požadavků. Tyto testy tedy mohou být psány i testery, kteří nemají hlubší znalost programování.

Oproti tomu White box testy přímo využívají znalost implementace. Na základě zdrojového kódu a použitého algoritmu lze ověřit chování, které by jinak bylo skryto. Protože jdou tyto testy více do implementačních detailů, jsou časově více náročné na tvorbu. Tato metoda se tedy kvůli tomu více využívá k testování menších komponent [69].

3.5.3 Typy testů

Testy lze dále dělit na základě úrovně detailu, do které je v nich zacházeno. Nejníže se nacházejí testy jednotkové. Ty testují jednotlivé komponenty v izolovaném prostředí. Navazují na ně testy integrační, které se věnují více komponentám současně a kontrolují jejich vzájemnou spolupráci. Systémové testy jsou postaveny nejvýše a mají za úkol ověřit funkčnost systému jako celku. Poslední úrovní jsou akceptační testy a předání uživatelům. Ty mají za cíl ověřit chování tak, jak si ho představuje klient, se kterým jsou tyto testy prováděny v blízké spolupráci [69].

3.5.4 Typy API testování

Automatizované API testování se dělí na několik různých typů. Prvním z nich je testování kontraktu definovaného aplikačním rozhraním. Kontrakt je v tomto případě úmluva o struktuře komunikace, kterou mezi sebou udržují serverová aplikace a kterákoliv jiná klientská. API je také kontrakt a kontraktové

testování má za cíl zkontrolovat, že API specifikace odpovídá poskytovaným datům. Ve zdrojovém kódu je toto testování realizováno kontrolou toho, zdali data získaná zavoláním endpointu strukturálně odpovídají specifikaci. Při testech jsou pak kontrolovány názvy atributů a datové typy. Kromě úspěšných cest lze v rámci těchto testů kontrolovat také návratové hodnoty, hlavičky a další vlastnosti získané odpovědi.

Jednotlivé endpointy lze testovat dalším typem testů, které umožňují jemnější kontrolu nad jednotlivými koncovými body. Testy endpointů na základě vstupních dat kontrolují výsledná data. Tentokrát se však nejedná pouze o jejich strukturu, ale také o vlastní hodnoty.

Kontrolu provedení určitého celku operací umožňují testovat integrační testy. Většinou se v rámci tohoto testu volá několik endpointů a cílem je otestovat jejich vzájemnou kooperaci [71].

3.5.5 Implementace testů

Jako testovací framework bylo zvoleno Codeception. Podporuje totiž rovnou bez instalace dalších knihoven jednotkové, integrační i akceptační testy. Pro vyhodnocování jednotkových testů je na pozadí využívána knihovna PHPUnit. Zároveň – a to je pro tento projekt důležité – je možné použít Codeception k testování aplikačního rozhraní. To je podporováno jak pro REST, tak SOAP API [72]. Sada testů je reprezentována třídou a každá její metoda přestavuje jeden test [73].

3.5.5.1 API testy

V tomto projektu je realizováno především kontraktové API testování popsané v podkapitole 3.5.4. Je pro něj vyhrazena samostatná databáze, do které jsou při prvotním spuštění testů nahrána testovací data. Integrací se Symfony a Doctrine dokáže Codeception podporovat i testování aplikačního rozhraní. Aby bylo docíleno izolace testu od ostatních, je každý test vložen do samostatné transakce. Jakmile je test dokončen, úspěšně či neúspěšně, je transakce vrácena zpět [74]. Každý test tedy pracuje se stejnými daty, přestože některé z nich mohou tato data ovlivnit.

Na ukázce 3.11 je zobrazen test, který získá přihlášku s identifikátorem 1 a ověří strukturu obdržené odpovědi. Nejdříve je specifikováno, jako jaký uživatel bude test vykonáván. Použitím pomocné funkce je k požadavku doplněn odpovídající autorizační token. Specifikací HTTP metody a cesty je odeslán požadavek. Verifikační fáze testu zkontroluje shodu návratové hodnoty, formát odpovědi a to, zda struktura odpovídá té očekávané. Pokud žádná z těchto validačních metod neoznámí chybu, je test prohlášen za úspěšný.

```
public function testGetApplicationSuccess(ApiTester $I): void
{
    $I->wantToTest('get application successfully');
    $I->amLoggedInAsTeacherAdmin1();
    $I->sendGet('/api/applications/1');

    $I->seeResponseCodeIs(HttpCode::OK);
    $I->seeResponseIsJson();
    $I->seeResponseMatchesJsonType([
        'id' => 'integer',
        'application_status' => 'string',
        'due_date' => 'string|null',
        'child' => 'array',
        ...
    ]);
}
```

Ukázka 3.11: Test koncového bodu v Codeception

3.6 Výsledky

Výsledkem této práce je serverová aplikace, která vznikla na základě definovaných požadavků a která poskytuje jasně definované aplikační rozhraní. Byla vytvořena za využití frameworku Symfony a všechny její součástí jsou spustitelné v Dockeru. V rámci aplikace vzniklo 77 endpointů, přičemž většina z nich je otestována pomocí API testů.

Z definovaných funkčních požadavků byly splněny všechny kromě FP9 definovaném v podkapitole 1.3.1.9 (především kvůli své složitosti a nižší prioritě) a částečně FP7 z podkapitoly 1.3.1.7, protože nebyla implementována platební brána a místo toho je pouze poskytnut bankovní účet, na který je platbu za přihlášku potřeba zaslat. S tím úzce souvisí nesplnění nefunkčního požadavku NP2 z podkapitoly 1.3.2.2. Opět tak bylo učiněno z důvodu velké časové náročnosti. Ostatní nefunkční požadavky byly splněny.

Výše zmíněné nedostatky nicméně nijak zásadně neovlivňují použitelnost této aplikace. Systém je funkční a umožňuje provádět všechny důležité operace definované funkčními požadavky a případy užití.

V současné době k systému existují také dvě klientské aplikace, které jsem vytvořil k předchozí verzi systému. První z nich je webová aplikace, která poskytuje uživatelské rozhraní k administrativní sekci. Druhá je pak klientská mobilní aplikace pro platformu Android, která slouží především k zápisu docházky učiteli. Tato volba – tedy vyvinout separátní mobilní aplikaci – byla učiněna z důvodu složitosti přístupu k počítači během zápisu docházky napří-

klad v tělocvičně nebo mimo školu. Obě tyto aplikace se budou muset upravit, aby byly kompatibilní se serverovou aplikací, která vznikla v této práci. Obrazovky z obou těchto klientských aplikací jsou zobrazeny v příloze C.

Diskuze

Tato práce se zaměřila na doposud neúplně řešené téma správy volnočasových aktivit ve školních zařízeních. Výsledkem návrhu a implementace je funkční aplikace, která tuto problematiku řeší.

Na rozdíl od již existujících řešení volí tento projekt cestu zaměření na školní zařízení. Primární cílovou skupinou nejsou sportovní kluby, přestože by se pro ně tento systém dal také využít. Již zavedené systémy pro sportovní kluby řeší problematiku aktuálně více do hloubky než tento projekt. Pro tyto subjekty tedy aktuálně nedává smysl použít aplikaci popsanou v této práci, a to především z důvodu chybějících funkcionalit. Systémy zaměřené primárně na školy (jako například Bakaláři) neposkytují stejnou funkcionalitu, která je dodána aplikací vytvořenou v této práci. Bude proto jednodušší nalézt školy, které budou chtít tento software využívat.

Nevýhodou navržené a implementované aplikace by mohla být její specifičnost. Analýza požadavků vycházela pouze ze dvou různých škol. Každá škola může mít ale drobně odlišné požadavky a mohlo by dojít k tomu, že aplikace jim nebude přesně odpovídat. Vzájemným dialogem by se nicméně dalo dojít k řešení, ve kterém by požadovaná chybějící funkcionalita mohla být integrována do celého systému.

Vzhledem k prezentovaným výsledkům je vidět, že ne všechny funkcionality navržené ve funkčních a nefunkčních požadavcích byly v době dokončení této práce (květen 2023) splněny. Nebyla dokončena implementace plateb prostřednictvím platební brány za jednotlivé kroužky. Je tedy nutné, aby administrátor manuálně měnil stav přihlášky na základě obdržených plateb. Současně nebyla implementována možnost generických tiskových sestav. Aplikace byla testována API testy kontraktů. Bylo by vhodné testování rozšířit dodáním integračních testů, a zajistit tak ověření funkčnosti hlavních scénářů.

Aby byla tato serverová aplikace využitelná reálnými uživateli, bude třeba dokončit zmíněné nedostatky. Teprve potom bude možné nasadit tuto aplikaci do produkčního prostředí a otestovat ji na skupině reálných uživatelů. Vzhledem k tomu, že je tato aplikace serverová s jasně definovaným rozhra-

ním, mohou nové klientské aplikace vznikat velmi jednoduše. Dalším krokem bude zamyšlení se nad byznysovým modelem a zpoplatněním, aby byl projekt udržitelný a životaschopný. V ideálním případě by pak následovalo rozšíření počtu škol, na kterých je software eKroužek nasazen.

Závěr

Hlavní myšlenkou této práce bylo usnadnit náročnou administrativu týkající se pořádání kroužků ve školách. Bylo jí alespoň částečně dosaženo analýzou, návrhem a následnou realizací aplikace, která organizátorům a lektorům s touto problematikou pomůže. Nebudou tedy muset tolik času trávit spravováním kroužků a budou se moci více věnovat přihlášeným dětem.

Čeho ale bylo dosaženo kompletně, je splnění cílů, které byly pro tuto práci vytyčeny – od analýzy domény přes návrh řešení až po samotnou realizaci. Vytvořená aplikace umožní lektorům jednoduše komunikovat dostupné volnočasové aktivity na jejich škole. Rodiče dětí budou moci podat elektronickou přihlášku a tímto je registrovat na vybrané kroužky. Evidence docházky nebude muset být prováděna ručně, ale lektoři ji budou moci zaznamenávat elektronicky přímo v aplikaci. Rodiče pak budou mít okamžitý přehled o docházce svých dětí.

Na základě prvotní analýzy a návrhu se povedlo zrealizovat celou serverovou aplikaci, která je téměř připravena k nasazení do produkce. V práci na projektu chci rozhodně pokračovat a těším se na to, jak se bude rozvíjet.

Bibliografie

1. ČESKÁ UNIE SPORTU. *Sportovní Kluby - ISCUS.CZ* [online]. © 2013-2023. [cit. 2023-05-08]. Dostupné z: https://iscus.cz/web/sportovni-subjekty.
2. JML GROUP S. R. O. *Administrativní a evidenční Systém Pro Sportovní Kluby* [online]. Brabec Media, 2020 [cit. 2023-01-18]. Dostupné z: https://www.sportes.cz/.
3. SPORTSCOACH S. R. O. *Sportovní systém SportsCoach* [online]. 2018. [cit. 2023-01-18]. Dostupné z: https://sportscoach.cz/.
4. OMNIVEDA SCHOOL S. R. O. *Kroužky & Tábory* [online]. Clubs & Camps s. r. o., 2016 [cit. 2023-01-18]. Dostupné z: https://www.krouzkyatabory.cz/.
5. KIDSJOY Z.S. *Celorepubliková databáze táborů a kroužků* [online]. Web7master s. r. o., 2020 [cit. 2023-01-18]. Dostupné z: https://www.tabory-krouzky.cz/.
6. DOSTÁL, J. *Školní informační systémy.* Univerzita Palackého v Olomouci, Pedagogická fakulta, 2011. Studijní opory. ISBN 9788024427843. Dostupné také z: https://books.google.cz/books?id=DVyX%5C_MtxT4gC.
7. BAKALÁŘI SOFTWARE. *Podvýběry třídní knihy, předměty OV* [online]. 1988. [cit. 2023-01-18]. Dostupné z: https://napoveda.bakalari.cz/tk_data_data_filter.htm.
8. ZŠ LIBEREC, UL. 5. KVĚTNA 64/49, PŘÍSP. ORG. *Zájmové kroužky 2021/2022* [online]. 2021. [cit. 2023-01-18]. Dostupné z: https://www.zs5kveten.cz/clanek/zajmove-krouzky-20212022.
9. BAKALÁŘI SOFTWARE. *Zápisy na školní akce* [online]. 2008. [cit. 2023-01-18]. Dostupné z: https://www.skolaonline.cz/Moduly/Z%C3%A1pisyna%C5%A1koln%C3%ADakce.aspx.

10. LAPLANTE, P.A. *Requirements Engineering for Software and Systems*. CRC Press, 2017. Applied Software Engineering Series. ISBN 9781315303703. Dostupné také z: https://books.google.cz/books?id=dkQPEAAAQBAJ.
11. IEEE Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998*. 1998, s. 1–40. Dostupné z DOI: 10.1109/IEEESTD.1998.88286.
12. BRAY, I.K. *An Introduction to Requirements Engineering*. Addison-Wesley, 2002. ISBN 9780201767926. Dostupné také z: https://books.google.cz/books?id=7QmqTtd8ycUC.
13. HULL, E.; JACKSON, K.; DICK, J. *Requirements Engineering*. Springer London, 2010. ISBN 9781849964050. Dostupné také z: https://books.google.cz/books?id=5xREIrqnDQEC.
14. GROUP, IETF OAuth Working. *OAuth 2.0* [online]. [B.r.]. [cit. 2023-03-30]. Dostupné z: https://oauth.net/2/.
15. BITTNER, K.; SPENCE, I.; JACOBSON, I. *Use Case Modeling*. Addison Wesley, 2003. Addison-Wesley object technology series. ISBN 9780201709131. Dostupné také z: https://books.google.cz/books?id=zvxfXvEcQjUC.
16. KULAK, D.; GUINEY, E. *Use Cases: Requirements in Context*. Pearson Education, 2012. ISBN 9780133085150. Dostupné také z: https://books.google.cz/books?id=cQ-QmAnu0HUC.
17. SPARX SYSTEMS PTY LTD. *Relationship Matrix* [online]. © 2000-2023. [cit. 2023-04-13]. Dostupné z: https://sparxsystems.com/enterprise_architect_user_guide/15.2/guidebooks/tools_ba_relationship_matrix.html.
18. BIEHL, M. *RESTful API Design*. CreateSpace Independent Publishing Platform, 2016. API-University Series. ISBN 9781514735169. Dostupné také z: https://books.google.cz/books?id=DYC3DwAAQBAJ.
19. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]. 2000. [cit. 2023-02-09]. Doctoral dissertation. University of California, Irvine.
20. FIELDING, Roy T.; NOTTINGHAM, Mark; RESCHKE, Julian. *HTTP Semantics* [RFC 9110]. RFC Editor, 2022 [cit. 2023-02-11]. Request for Comments, č. 9110. Dostupné z DOI: 10.17487/RFC9110.
21. DUSSEAULT, Lisa M.; SNELL, James M. *PATCH Method for HTTP* [RFC 5789]. RFC Editor, 2010 [cit. 2023-03-29]. Request for Comments, č. 5789. Dostupné z DOI: 10.17487/RFC5789.

22. IANA: INTERNET ASSIGNED NUMBERS AUTHORITY. *Hypertext Transfer Protocol (HTTP) Status Code Registry* [online]. 2022. [cit. 2023-04-16]. Dostupné z: https://www.iana.org/assignments/http-status-codes/http-status-codes.xml.
23. GUDGIN, M.; HADLEY, M.; MENDELSOHN, N.; MOREAU, J.; NIELSEN, H.; KARMARKAR, A.; LAFON, Y. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)* [online]. 2007. [cit. 2023-02-11]. Dostupné z: https://www.w3.org/TR/soap12/.
24. ARAÚJO, B.M. *Hands-On RESTful Web Services with TypeScript 3: Design and develop scalable RESTful APIs for your applications*. Packt Publishing, 2019. ISBN 9781789955019. Dostupné také z: https://books.google.cz/books?id=AjqPDwAAQBAJ.
25. FOWLER, M. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Pearson Education, 2012. Addison-Wesley Signature Series (Fowler). ISBN 9780133065213. Dostupné také z: https://books.google.cz/books?id=vqTfNFDzzdIC.
26. DARIE, C.; BALANESCU, E. *Beginning PHP and MySQL E-Commerce: From Novice to Professional*. Apress, 2008. Apresspod Series. ISBN 9781590598641. Dostupné také z: https://books.google.cz/books?id=YHSQM2URXzoC.
27. QURESHI, M. Rizwan Jameel; SABIR, Fatima. A comparison of model view controller and model view presenter. 2014. Dostupné z DOI: 10.48550/ARXIV.1408.5786.
28. POREBSKI, B.; PRZYSTALSKI, K.; NOWAK, L. *Building PHP Applications with Symfony, CakePHP, and Zend Framework*. Wiley, 2011. IT Pro. ISBN 9781118067925. Dostupné také z: https://books.google.cz/books?id=gTEl2mWGNtAC.
29. STATISTICS & DATA. *Most popular backend frameworks – 2012/2022* [online]. 2022. [cit. 2023-02-01]. Dostupné z: https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/.
30. SYMFONY. *Laravel: A framework project using Symfony components* [online]. [B.r.]. [cit. 2023-04-01]. Dostupné z: https://symfony.com/projects/laravel.
31. LARAVEL LLC. *The PHP framework for web artisans* [online]. [B.r.]. [cit. 2023-02-01]. Dostupné z: https://laravel.com/docs/9.x.
32. CLARK, Jessica. *Top 10 backend frameworks: Which is the best option for you?* [Online]. 2022. [cit. 2023-02-01]. Dostupné z: https://blog.back4app.com/backend-frameworks.
33. YUNG, Steven. *Laravel Greatest Trick revealed: Magic methods* [online]. DEV Community, 2019 [cit. 2023-03-29]. Dostupné z: https://dev.to/nvio/laravel-greatest-trick-revealed-magic-methods-31om.

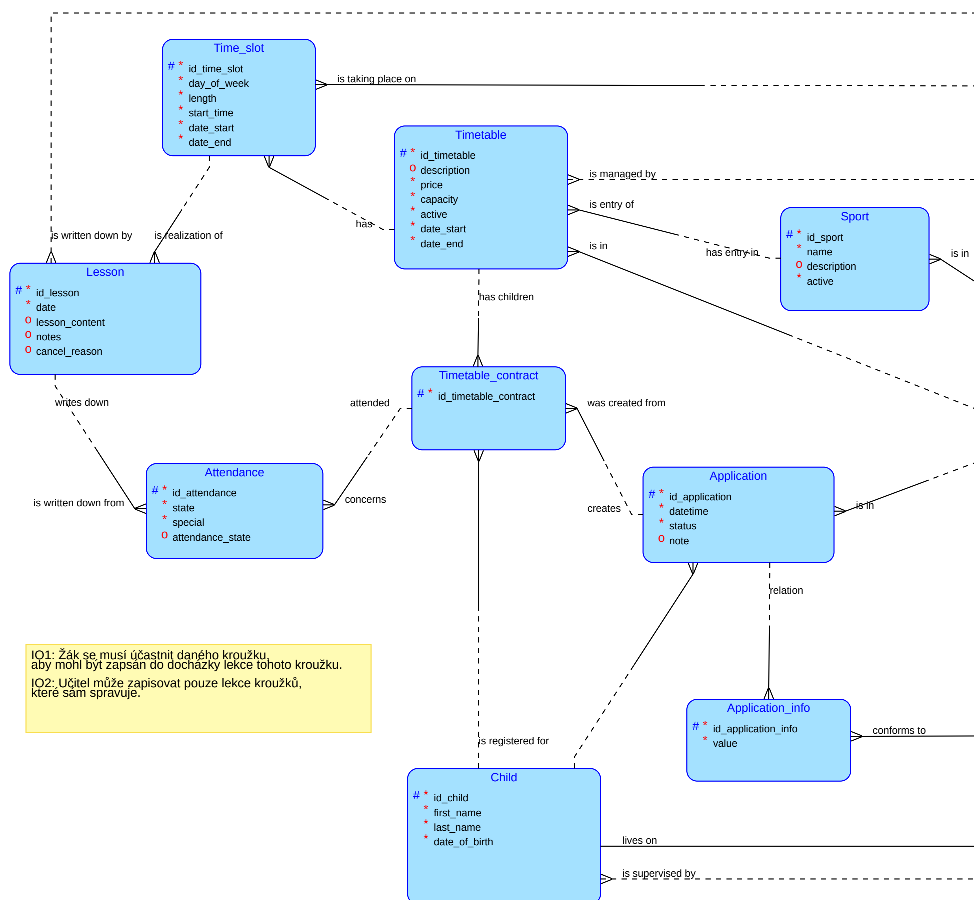
34. CHOJRIN, Mauro. *Laravel vs. Symfony: A side by side comparison* [online]. Adeva, 2021 [cit. 2023-02-01]. Dostupné z: https://adevait.com/laravel/laravel-vs-symfony-comparison.
35. DJANGO SOFTWARE FOUNDATION. *Django documentation* [online]. [B.r.]. [cit. 2023-02-05]. Dostupné z: https://docs.djangoproject.com/en/4.1/.
36. FLASK. *Design decisions in flask* [online]. [B.r.]. [cit. 2023-02-01]. Dostupné z: https://flask.palletsprojects.com/en/2.2.x/design/#.
37. SYMFONY. *Symfony Documentation* [online]. [B.r.]. [cit. 2023-04-01]. Dostupné z: https://symfony.com/doc/current/index.html.
38. MEIER, A.; KAUFMANN, M. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*. Springer Fachmedien Wiesbaden, 2019. ISBN 9783658245498. Dostupné také z: https://books.google.cz/books?id=XOCgDwAAQBAJ.
39. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *What is PostgreSQL?* [Online]. 2022. [cit. 2023-01-29]. Dostupné z: https://www.postgresql.org/about/.
40. MEHTA, V.P. *Pro LINQ Object Relational Mapping in C# 2008*. Apress, 2008. Books for professionals by professionals. ISBN 9781430205975. Dostupné také z: https://books.google.cz/books?id=ZbUYAAAAQBAJ.
41. DOCTRINE PROJECT. *Getting Started with Doctrine* [online]. [B.r.]. [cit. 2023-01-29]. Dostupné z: https://www.doctrine-project.org/projects/doctrine-orm/en/current/tutorials/getting-started.html.
42. DOCTRINE PROJECT. *Doctrine query language* [online]. [B.r.]. [cit. 2023-01-30]. Dostupné z: https://www.doctrine-project.org/projects/doctrine-orm/en/2.14/reference/dql-doctrine-query-language.html.
43. BAYERN, Macy. *The 10 most popular container tools for businesses* [online]. TechRepublic, 2019 [cit. 2023-01-29]. Dostupné z: https://www.techrepublic.com/article/the-10-most-popular-container-tools-for-businesses/.
44. DOCKER INC. *What is a container?* [Online]. © 2023. [cit. 2023-01-30]. Dostupné z: https://www.docker.com/resources/what-container/.
45. DOCKER INC. *Docker Compose overview* [online]. © 2013-2023. [cit. 2023-01-30]. Dostupné z: https://docs.docker.com/compose/.
46. SUMATHI, S.; ESAKKIRAJAN, S. *Fundamentals of Relational Database Management Systems*. Springer Berlin Heidelberg, 2007. Studies in Computational Intelligence. ISBN 9783540483977. Dostupné také z: https://books.google.cz/books?id=RjnNA0GW0wsC.

47. BARKER, R. *CASE Method: Entity Relationship Modelling*. Addison-Wesley, 1990. CASE method, č. sv. 1. ISBN 9780201416961. Dostupné také z: https://books.google.cz/books?id=yNNQAAAAMAAJ.
48. JETBRAINS S. R. O. *PhpStorm: Php Ide and code editor from JetBrains* [online]. [B.r.]. [cit. 2023-03-11]. Dostupné z: https://www.jetbrains.com/phpstorm/.
49. ESPENDILLER, Daniel. *Symfony Support* [online]. JetBrains s. r. o., © 2000-2023 [cit. 2023-04-13]. Dostupné z: https://plugins.jetbrains.com/plugin/7219-symfony-support.
50. UZAYR, S. *Mastering Git: A Beginner's Guide*. CRC Press, 2022. Mastering Computer Science. ISBN 9781000552911. Dostupné také z: https://books.google.cz/books?id=cqheEAAAQBAJ.
51. GITLAB B.V. *GitLab Docs* [online]. 2023. [cit. 2023-03-16]. Dostupné z: https://docs.gitlab.com.
52. GITLAB B.V. *CI/CD concepts* [online]. 2023. [cit. 2023-03-16]. Dostupné z: https://docs.gitlab.com/ee/ci/introduction/index.html.
53. HENTZEN, W. *The Software Developer's Guide*. Hentzenwerke Publishing, 2002. Hentzenwerke Series. ISBN 9781930919006. Dostupné také z: https://books.google.cz/books?id=ClZvrmABkrMC.
54. SYMFONY. *Symfony releases, notifications and release Checker* [online]. [B.r.]. [cit. 2023-03-08]. Dostupné z: https://symfony.com/releases.
55. SYMFONY. *Installing & Setting up the Symfony Framework* [online]. [B.r.]. [cit. 2023-03-09]. Dostupné z: https://symfony.com/doc/current/setup.html.
56. SYMFONY. *Using Docker with Symfony* [online]. [B.r.]. [cit. 2023-03-09]. Dostupné z: https://symfony.com/doc/current/setup/docker.html.
57. DAVIS, A. *Bootstrapping Microservices with Docker, Kubernetes, and Terraform: A project-based guide*. Manning Publications, 2021. ISBN 9781617297212. Dostupné také z: https://books.google.cz/books?id=QKQbEAAAQBAJ.
58. YEAGER, N.J.; MCGRATH, R.E. *Web Server Technology*. Elsevier Science, 1996. ISBN 9781558603769. Dostupné také z: https://books.google.cz/books?id=0jExRH3%5C_-hQC.
59. F5, INC. *Advanced load balancer, web server, & reverse proxy* [online]. [B.r.]. [cit. 2023-03-30]. Dostupné z: https://www.nginx.com/.
60. ADERMANN, N.; BOGGIANO, J. *Composer: A Dependency Manager for PHP* [online]. [B.r.]. [cit. 2023-03-09]. Dostupné z: https://getcomposer.org/.

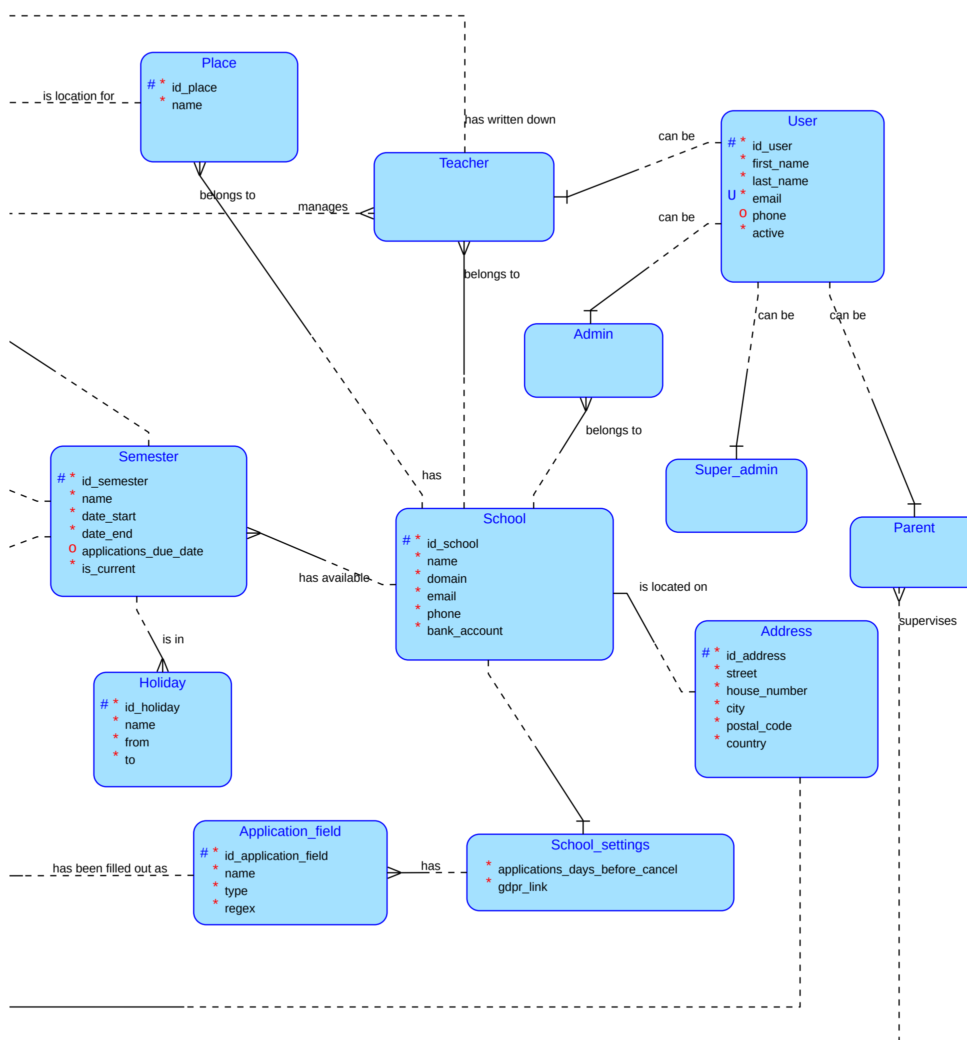
61. SYMFONY. *Databases and the Doctrine ORM* [online]. [B.r.]. [cit. 2023-03-16]. Dostupné z: https://symfony.com/doc/current/doctrine.html.
62. DOCTRINE PROJECT. *Transactions and Concurrency* [online]. [B.r.]. [cit. 2023-03-16]. Dostupné z: https://www.doctrine-project.org/projects/doctrine-orm/en/2.14/reference/transactions-and-concurrency.html.
63. SYMFONY. *Forms* [online]. [B.r.]. [cit. 2023-03-16]. Dostupné z: https://symfony.com/doc/current/forms.html.
64. HUJER, Martin. *Don't Use Entities in Symfony Forms. Use Custom Data Objects Instead* [online]. 2017. [cit. 2023-03-16]. Dostupné z: https://blog.martinhujer.cz/symfony-forms-with-request-objects/.
65. SYMFONY. *Validation* [online]. [B.r.]. [cit. 2023-03-16]. Dostupné z: https://symfony.com/doc/current/validation.html.
66. BEŠŤÁK, Vojtěch. *Pagination and filters bundle* [online]. GitHub, 2023 [cit. 2023-04-10]. Dostupné z: https://github.com/ekrouzek/pagination-filters-bundle.
67. SMARTBEAR SOFTWARE. *Swagger Codegen* [online]. 2023. [cit. 2023-03-30]. Dostupné z: https://github.com/swagger-api/swagger-codegen.
68. SYMFONY. *NelmioApiDocBundle* [online]. [B.r.]. [cit. 2023-03-16]. Dostupné z: https://symfony.com/bundles/NelmioApiDocBundle/current/index.html.
69. BURNSTEIN, I. *Practical Software Testing: A Process-Oriented Approach*. Springer New York, 2003. Springer Professional Computing. ISBN 9780387951317. Dostupné také z: https://books.google.cz/books?id=0v6HSeqA0OoC.
70. DESAI, S.; SRIVASTAVA, A. *SOFTWARE TESTING : A Practical Approach*. Phi Learning, 2016. ISBN 9788120352261. Dostupné také z: https://books.google.cz/books?id=B4sQDAAAQBAJ.
71. WESTERVELD, D. *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing, 2021. ISBN 9781800565739. Dostupné také z: https://books.google.cz/books?id=93QqEAAAQBAJ.
72. CODECEPTION. *API testing* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: https://codeception.com/docs/APITesting.
73. CODECEPTION. *Introduction* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: https://codeception.com/docs/Introduction.
74. CODECEPTION. *Doctrine2* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: https://codeception.com/docs/modules/Doctrine2.

Příloha A

Kompletní ER model



Obrázek A.1: Kompletní ER model – část 1.



Obrázek A.2: Kompletní ER model – část 2.

Příloha B

API dokumentace

Application

Method	Endpoint	Description
POST	/api/schools/{schoolId}/applications	Creates an application.
GET	/api/semesters/{semesterId}/applications	Get applications.
GET	/api/applications/my	Get my applications.
GET	/api/applications/{applicationId}	Get application
DELETE	/api/applications/{applicationId}	Delete application
PATCH	/api/applications/{applicationId}	Update application

Child

Method	Endpoint	Description
GET	/api/schools/{schoolId}/children	Get children.
GET	/api/children/{childId}	Get child
DELETE	/api/children/{childId}	Delete child
PATCH	/api/children/{childId}	Update child

Obrázek B.1: API dokumentace – část 1.

Enumeration

- GET `/api/enum/application-statuses` Get application statuses
- GET `/api/enum/countries` Get countries
- GET `/api/enum/roles` Get roles

Export

- GET `/api/export/timetable-contracts` Get applications CSV

Holiday

- GET `/api/semesters/{semesterId}/holidays` Get holidays.
- POST `/api/semesters/{semesterId}/holidays` Create holiday
- GET `/api/holidays/{holidayId}` Get holiday
- DELETE `/api/holidays/{holidayId}` Delete holiday
- PATCH `/api/holidays/{holidayId}` Update holiday
- GET `/api/semesters/{semesterId}/holidays/current` Get ongoing holiday

Lesson

- GET `/api/lessons/{lessonId}` Get lesson
- DELETE `/api/lessons/{lessonId}` Delete lesson
- PATCH `/api/lessons/{lessonId}` Update lesson
- POST `/api/time-slots/{timeSlotId}/lessons` Create lesson
- GET `/api/lessons/{lessonId}/attendances` Get lesson attendance
- GET `/api/timetables/{timetableId}/lessons` Get lessons by timetable
- GET `/api/sports/{sportId}/lessons` Get lessons by sport

Obrázek B.2: API dokumentace – část 2.

Place

GET `/api/schools/{schoolId}/places` Get places.

POST `/api/schools/{schoolId}/places` Create place

DELETE `/api/places/{placeId}` Delete place

PATCH `/api/places/{placeId}` Update place

School

GET `/api/schools` Get schools

POST `/api/schools` Create school

GET `/api/schools/{schoolId}` Get school

PATCH `/api/schools/{schoolId}` Update school

GET `/api/schools/by-slug/{schoolSlug}` Get school

GET `/api/schools/by-slug/{schoolSlug}/exists` School with slug exists

GET `/api/schools/{schoolId}/setting` Get school's setting

PATCH `/api/schools/{schoolId}/setting` Update school setting

Semester

GET `/api/schools/{schoolId}/semesters` Get semesters.

POST `/api/schools/{schoolId}/semesters` Create semester

GET `/api/semesters/{semesterId}` Get semester

DELETE `/api/semesters/{semesterId}` Delete semester

PATCH `/api/semesters/{semesterId}` Update semester

Security

POST `/api/security/register` Register

Obrázek B.3: API dokumentace – část 3.

Sport

Method	Endpoint	Description
GET	/api/semesters/{semesterId}/sports	Get sports.
POST	/api/semesters/{semesterId}/sports	Create sport
GET	/api/schools/{schoolId}/sports/active	Get sports from current semester .
GET	/api/sports/{sportId}	Get sport
DELETE	/api/sports/{sportId}	Delete sport
PATCH	/api/sports/{sportId}	Update sport
GET	/api/semesters/{semesterId}/sports/teacher	Get teachers sports.

Time slot

Method	Endpoint	Description
GET	/api/time-slots/{timeSlotId}	Get timetable
DELETE	/api/time-slots/{timeSlotId}	Delete time slot
PATCH	/api/time-slots/{timeSlotId}	Update time slot
GET	/api/semesters/{semesterId}/time-slots	Get time slots.
GET	/api/semesters/{semesterId}/time-slots/planned	Get planned time slots

Timetable

Method	Endpoint	Description
GET	/api/timetables/{timetableId}	Get timetable
DELETE	/api/timetables/{timetableId}	Delete timetable
PATCH	/api/timetables/{timetableId}	Update timetable
GET	/api/timetables/{timetableId}/children	Get children attending a timetable.
GET	/api/timetables/{timetableId}/parents	Get parents of children attending a timetable.
GET	/api/semesters/{semesterId}/timetables/teacher	Get my timetables
GET	/api/semesters/{semesterId}/timetables	Get all timetables

Obrázek B.4: API dokumentace – část 4.

GET `/api/sports/{sportId}/timetables` Get sport timetables

POST `/api/sports/{sportId}/timetables` Create timetable

User

GET `/api/users/{userId}/parent/children` Get parent's children

GET `/api/users/teachers` Get school teachers

GET `/api/users` Get all users.

POST `/api/users` Create user

GET `/api/schools/{schoolId}/users` Get school users.

GET `/api/users/self` Get logged-in user

GET `/api/users/{userId}` Get user

PUT `/api/users/{userId}` Update user

DELETE `/api/users/{userId}` Delete user

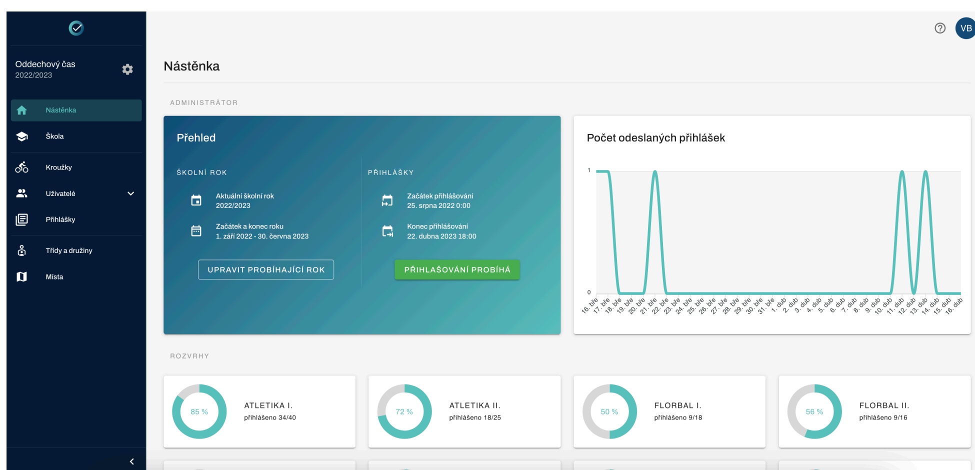
GET `/api/users/by-email/{email}/exists` User with e-mail exists

PATCH `/api/users/me` Update my profile

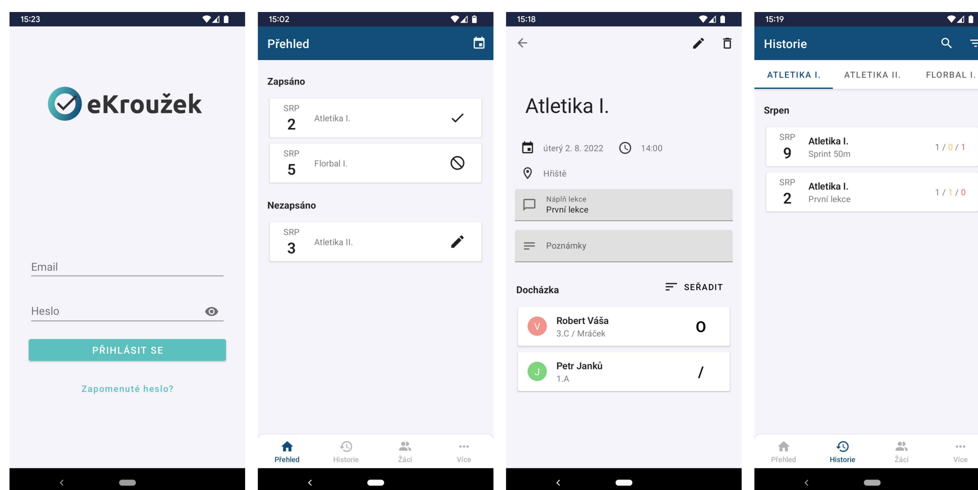
Obrázek B.5: API dokumentace – část 5.

Příloha C

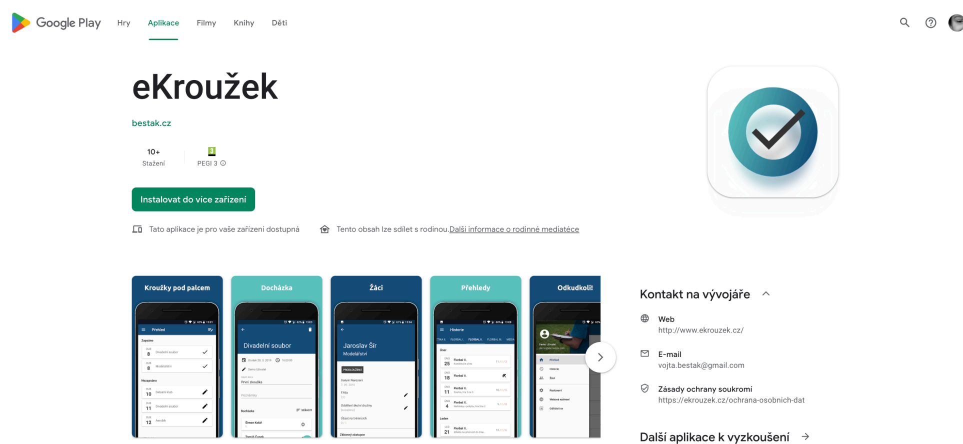
Existující klientské aplikace



Obrázek C.1: Obrazovka z klientské webové aplikace, získáno z `https://app.ekrouzek.cz`



Obrázek C.2: Obrazovky z mobilní aplikace pro platformu Android



Obrázek C.3: Mobilní aplikace pro platformu Android umístěná na Google Play, získáno z `https://play.google.com/store/apps/details?id=cz.bestak.ekrouzek`

Příloha D

Seznam zkratek

API Application programing interface

CD Continuous deployment

CI Continuous integration

CRUD Create, read, update, delete

DQL Doctrine query language

ER Entity relationship

HTTP Hypertext Transfer Protocol

IDE Integrated development environment

IT Information technology

JSON JavaScript Object Notation

MVC Model-View-Controller

MVP Model-View-Presenter

ORM Objektově relační mapování

REST Representational state transfer

RPC Remote procedure call

SOAP Simple object access protocol

URI Uniform resource identifier

VCS Version control system

VM Virtual machine

XML Extensible markup language

YAML YAML Ain't Markup Language

Obsah přiloženého média

- `README.md` .... stručný popis obsahu média
- `other`
  - `er-diagram.pdf` .... kompletní ER model
  - `api-docs.json` .... OpenAPI dokumentace
- `src` .... zdrojové kódy implementace
  - `README.md` .... popis implementace a příručka ke spuštění
- `text` .... text práce
  - `thesis.pdf` .... text práce ve formátu PDF
  - `thesis` .... zdrojová forma práce ve formátu LaTeX