



**FAKULTA
INFORMAČNÍCH
TECHNOLGIÍ
ČVUT V PRAZE**

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Rozšíření webové aplikace pro projekt „Úspěšný prvňáček“
Student: Bc. Lukáš Rod
Vedoucí: Ing. Stanislav Kuznetsov
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2020/21

Pokyny pro vypracování

1. Proveďte sběr a popis nových požadavků na rozšíření aplikace pro projekt „Úspěšný prvňáček“.
2. Proveďte analýzu nových požadavků na rozšíření stávajícího řešení.
3. Navrhněte aktualizovanou strukturu databáze a další související změny.
4. Implementujte všechny požadavky.
5. Proveďte testování aplikace pomocí nástrojů automatizovaného testování UI a API.
6. Proveďte nasazení výsledné aplikace.
7. Výslednou aplikaci zveřejněte jako open-source spolu s instrukcemi pro spuštění aplikace.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 13. prosince 2019



**FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE**

Diplomová práce

Rozšíření webové aplikace pro projekt „Úspěšný prvňáček“

Bc. Lukáš Rod

Katedra softwarového inženýrství

Vedoucí práce: Ing. Stanislav Kuznetsov

23. května 2020

Poděkování

Děkuji Ing. Stanislavu Kuznetsovi za vedení této práce. Děkuji za spolupráci také své mamce, PaedDr. Janě Rodové, která se mnou jakožto lektorka Úspěšného prvňáčka po celou dobu spolupracovala. Děkuji celé své rodině a přátelům za podporu a trpělivost během celého studia.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 23. května 2020

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2020 Lukáš Rod. 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

Rod, Lukáš. *Rozšíření webové aplikace pro projekt „Úspěšný prvňáček“*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020. Dostupný také z WWW: [⟨https://github.com/rodlukas/masters-thesis⟩](https://github.com/rodlukas/masters-thesis).

Abstrakt

Tato práce si klade za cíl rozšířit webovou aplikaci pro projekt „Úspěšný prvňáček“, který nabízí doučování a kurzy pro budoucí nebo nastupující prvňáčky. Původní aplikace byla vytvořena v rámci předcházející bakalářské práce a slouží k evidování klientů, jejich docházky, skupin, plateb za lekce a zobrazení celé historie klienta. Serverová část původní i nové aplikace je napsána v Pythonu s webovým frameworkem Django, klientská část v Reactu, navzájem spolu komunikují přes REST API díky Django REST Frameworku. Výsledná nová rozšířená aplikace splňuje všechny nové požadavky lektorky. Také je díky zavedení pokročilých nástrojů pro usnadnění vývoje a údržby, vysokému pokrytí automatizovanými testy API i UI (E2E) a zavedení několika prostředí pro nasazování umožněno spolehlivější a rychlejší dodávání nových verzí. Aplikace je nasazena do několika prostředí (včetně produkčního) na Heroku a lektorce umožňuje ještě efektivnější a pohodlnější každodenní práci pokrývající více oblastí díky novým funkcím.

Klíčová slova webová aplikace, Úspěšný prvňáček, Python, Django, React, Django REST framework

Abstract

The goal of this thesis is to extend the web application for the project “Úspěšný prvňáček” which offers an extra education and courses for preschoolers. The original application was created in a bachelor's thesis and offers features for storing information about clients, their attendances, groups, payments for the lectures, and viewing the client's entire history. The server side of the original and also the new application is written in Python with Django web framework, the client side is built with React and communicates via a REST API thanks to Django REST Framework. The final new extended application meets all the new requirements made by the lector. Also, thanks to the integration with advanced tools for easier development and maintenance, high code coverage by automated API and UI (E2E) tests, and configuration of multiple deploy environments, more reliable and faster delivery of new releases is possible. The application is deployed to more environments (including the production one) to Heroku and offers much more efficient and comfortable everyday work covering more areas of functionality thanks to the new features.

Keywords web application, Úspěšný prvňáček, Python, Django, React, Django REST framework, Successful first-grader

Obsah

Úvod	1
1 Cíle práce	3
I Teoretická část	5
2 Aktuální řešení	7
2.1 Implementované požadavky	7
2.1.1 Implementované funkční požadavky	7
2.1.2 Implementované nefunkční požadavky	9
2.2 Použité technologie	9
2.2.1 Serverová část	9
2.2.2 Klientská část	10
2.3 Prostředí, testování a nasazování	11
2.4 Plán rozšíření z bakalářské práce	11
3 Možnosti automatizovaného testování	13
3.1 Srovnání manuálního a automatizovaného testování	13
3.2 Struktura automatizovaných testů	15
3.2.1 Testovací pyramida	15
3.2.2 Alternativní přístupy	17
3.3 Metodiky a nástroje pro automatizované testování	17
3.3.1 TDD	18
3.3.2 BDD	18
3.3.3 Další nástroje	18
4 Nástroje pro usnadnění vývoje a údržby	21
4.1 Monitorování chyb	22
4.2 Správa logů	23

4.3	Statické typování	23
4.3.1	Srovnání s dynamickým typováním	24
4.3.2	Klientská část	25
4.3.3	Serverová část	26
4.4	Statická analýza kódu	27
4.4.1	Průběžná kontrola kvality kódu	28
4.4.2	Lintery	30
4.4.3	Formattery	31
5	Zvolené technologie	33
5.1	Testování	33
5.2	Nástroje pro usnadnění vývoje a údržby	34
5.2.1	Monitorování chyb	34
5.2.2	Správa logů	34
5.2.3	Statické typování	34
5.2.4	Statická analýza kódu	35
II	Praktická část	37
6	Sběr požadavků a analýza	39
6.1	Požadavky	39
6.1.1	Funkční požadavky	39
6.1.2	Nefunkční požadavky	42
6.1.3	Vyřazené požadavky	44
6.2	Detailní analýza některých požadavků	44
6.2.1	F3 - vylepšení předplacených lekcí	44
6.2.2	F13 – automatické předvyplnění údajů lekce	45
6.2.3	F18 – omezení a validace hodnot	45
6.2.4	N4 – revize bezpečnosti	45
6.2.5	N5 – vylepšení použitelnosti	47
7	Návrh	51
7.1	Datový model	51
7.1.1	Předplacené lekce	53
7.1.2	Zájemci o kurzy	53
7.1.3	Vlastnosti stavů účasti	53
7.1.4	Další změny	54
7.2	Komunikační rozhraní	54
7.2.1	Opravy a úpravy stávajícího rozhraní	54
7.2.2	Předplacené lekce	55
7.2.3	Banka	56
7.2.4	Zájemci o kurzy	56
7.2.5	Další rozšíření	56

7.3	Architektura	57
7.4	Konfigurace více prostředí	57
7.5	Uživatelské prostředí	59
7.5.1	Zájemci o kurzy	60
7.5.2	Formulář pro lekce	60
8	Implementace	65
8.1	Funkční požadavky	65
8.1.1	F1 – evidování zájemců o kurz	65
8.1.2	F2 – kontrola časového konfliktu lekcí	66
8.1.3	F3 – vylepšení předplacených lekcí	67
8.1.4	F4 – vyhledávání klientů	67
8.1.5	F5 – přepracování formuláře pro lekce	69
8.1.6	F6 – zavedení aktivních a neaktivních klientů a skupin	69
8.1.7	F7 – nastavitelná délka kurzů	70
8.1.8	F8 – propojení s bankou	70
8.1.9	F9 – změny účastníků skupinových lekcí	72
8.1.10	F10 – automatické přidání předplacené lekce	73
8.1.11	F11 – efektivnější práce v rámci aplikace	73
8.1.12	F12 – evidování barev kurzů	76
8.1.13	F13 – automatické předvyplnění údajů lekce	77
8.1.14	F14 – upozornění na ztrátu dat formulářů	77
8.1.15	F15 – vylepšení chybových hlášení	78
8.1.16	F16 – titulky stránek	80
8.1.17	F17 – nastavitelné vlastnosti stavů účasti	80
8.1.18	F18 – omezení a validace hodnot	81
8.1.19	F19 – automatické zrušení lekce	82
8.1.20	F20 – zobrazení zrušených lekcí	83
8.1.21	F21 – skupinové lekce bez účastníků	83
8.2	Nefunkční požadavky	84
8.2.1	N1 – dokumentace	84
8.2.2	N3 – zavedení nástrojů pro usnadnění vývoje a údržby	86
8.2.2.1	Monitorování chyb	86
8.2.2.2	Správa logů	86
8.2.2.3	Statické typování – serverová část	87
8.2.2.4	Statické typování – klientská část	87
8.2.2.5	Formatter	89
8.2.2.6	Lint	89
8.2.2.7	Statická analýza kódu	90
8.2.3	N4 – revize bezpečnosti	92
8.2.3.1	B1 – aktualizace závislostí	92
8.2.3.2	B2 – sjednocení konfigurací	92
8.2.3.3	B3 – deaktivace DEBUG módu	93
8.2.3.4	B4 – HTTP hlavičky	93

8.2.3.5	B5 – sjednocení práce s tokeny	94
8.2.3.6	B6 – zákaz indexace roboty	95
8.2.4	N5 – vylepšení použitelnosti	96
8.2.5	N6 – optimalizace API	99
8.3	Další řešené problémy	104
8.3.1	Migrace a refaktoring	104
8.3.2	Další opravy chyb a vylepšení	105
8.3.3	Nástroj pro vývoj klientské části	106
9	Testování	109
9.1	Scénáře	109
9.2	Implementace testů	110
9.2.1	Další testování	112
10	Nasazení	113
10.1	Aktualizace prostředí a vylepšení	113
10.2	Testování	114
10.3	Nasazování na více prostředí	114
10.4	Ukázka aplikace	115
11	Zveřejnění jako open-source	117
12	Možná rozšíření	119
	Závěr	121
	Bibliografie	123
A	Seznam použitých zkratk	139
B	Obsah příloženého CD	141

Seznam obrázků

2.1	Logický datový model z bakalářské práce	8
3.1	Srovnání strategie „testovací pyramidy“ a „zmrzlinového kornoutu“	16
3.2	Strategie „testovací trofeje“	17
7.1	Aktualizovaný a rozšířený původní logický datový model	52
7.2	Aktualizovaný diagram nasazení	58
7.3	Návrh zájemců o kurzy	60
7.4	Návrh nového formuláře pro skupinové lekce	61
7.5	Návrh nového formuláře pro lekce jednotlivců	61
7.6	Původní formulář pro lekce skupin	63
8.1	Implementace počítadel předplacených lekcí ve skupinách	68
8.2	Přepracovaný formulář pro lekce	69
8.3	Přepínač aktivních a neaktivních klientů	70
8.4	Komponenta pro informace z banky	72
8.5	Tlačítko pro projevení změn účastníků lekce	73
8.6	Možnost jednoduchého přidání nového klienta z jiného formuláře .	75
8.7	Rychlé přidání lekce – krok 1	76
8.8	Rychlé přidání lekce – krok 2	76
8.9	Komponenta pro výběr barvy kurzu	77
8.10	Konfigurace vlastností stavů účasti v nastavení aplikace	81
8.11	Automatické zrušení lekce, když jsou všichni omluveni	83
8.12	Ukázka z nasazené API dokumentace (Swagger UI)	85
8.13	Upozornění při velmi dlouhém načítání	96
8.14	Komponenta react-select pro výběr kurzů	97
8.15	Analýza SQL požadavku prostřednictvím Django Debug Toolbar .	100
10.1	Snímek obrazovky s týdenním přehledem	116
11.1	Sestavená klientská část ke stažení na GitHubu	118

Seznam ukázek kódu

1	Dotaz pro nalezení časových konfliktů	66
2	Konfigurace vyhledávání klientů ze souboru Main.tsx	68
3	Dekorátor pro zavedení cache	71
4	Upozornění na neuložené změny při zavření formuláře	78
5	Upozornění na neuložené změny při zavření stránky	78
6	Ošetření chyb při mazání entit	79
7	Dokumentace v TS (komponenta PrepaidCounters)	84
8	Dokumentace v Pythonu (metoda pro validaci telefonního čísla)	85
9	Anotace typů v Pythonu	87
10	Anotace typů v TS	89
11	HSTS konfigurace	94
12	Ukázka konfigurace proměnných prostředí	95
13	Kód pro zákaz indexování webu vyhledávači	96
14	Řešení řazení podle české abecedy – 1. část	99
15	Řešení řazení podle české abecedy – 2. část	99
16	Optimalizace SQL dotazů v Django	101
17	Dělení kódu klientské části v Reactu	102
18	Ukázka použití hooků ze souboru Loading.tsx	106
19	Ukázka scénáře pro smazání klienta v souboru clients.feature	110
20	Implementace kroku ověřujícího, že je klient smazaný – ze souboru api/clients.py	111
21	Přístup k elementům v UI nezávislý na konkrétní implementaci UI	111
22	Substituce řetězců na Travisu	114

Úvod

Proces vývoje jakékoliv aplikace nikdy nekončí a jinak tomu není ani v případě současné webové aplikace pro projekt „Úspěšný prvňáček“¹ (dále jen ÚP) vedený speciální pedagožkou PaedDr. Janou Rodovou. Uplynuly přesně dva roky od nasazení první produkční verze v rámci mé bakalářské práce. Tomu předcházela dlouhá cesta příprav, analýz, rešerší, návrhů a poté implementace. Cesta to byla trnitá, i díky volbě mně naprosto cizích nejmodernějších technologií pro serverovou i klientskou část, snaze o jejich propojení, někdy nepříliš přívětivé dokumentaci či odlišným přístupům frameworků. Vše se ale nakonec zdárně podařilo a po akceptačním testování a posledním vyladění lektorka mohla spokojeně začít aplikaci každodenně používat. Existence této aplikace znamenala obrovský skok kupředu vzhledem k tomu, že do té doby byl užíván pro část evidence naprosto nedostačující Microsoft Excel, pro část klasický diář a pro další část různé papíry. To vše díky aplikaci skončilo.

Na závěr bakalářské práce jsem uváděl možná rozšíření v budoucnu, a to nezůstalo jen u slov. O měsíc později po odevzdání bakalářské práce naplno začala druhá etapa vývoje aplikace. O této etapě budu psát v rámci celé této diplomové práce. Během tohoto necelé dva roky trvajícího postupného vývoje bylo třeba řešit mnoho „kostlivců“, mnoho chyb, o kterých jsem v době vytváření neměl ani tušení. Ale také mnoho nových funkcí, protože bylo potřeba jít kupředu vzhledem k rozvoji samotného ÚP. Jak vývoj postupoval, přicházely další a další nové výzvy, požadavky a poznání. Objevilo se také několik slepých uliček. Ale nebudu předbíhat.

U projektu ÚP jsem již od samého počátku, a proto je tato webová aplikace mou srdeční záležitostí. Mojí snahou bylo tedy vyhovět naprosto všem požadavkům lektorky tak, aby s každou novou verzí byla její práce hladší, jednodušší a efektivnější. Další motivací je možnost si v rámci tohoto vývoje vyzkoušet další řadu nových technologií, nástrojů a přístupů a zdokonalit se tak v oblasti tvorby moderních webových aplikací.

¹<https://uspesnyprvnacek.cz/>

V teoretické části nejprve stručně popíši, co nabízí původní verze aplikace vypracovaná v rámci bakalářské práce – jaké má funkce, použité technologie, jak funguje testování a nasazování a do jakých prostředí se nasazuje. Zmíním zde i stručně původní náměty na rozšíření aplikace do budoucna, o kterých bylo uvažováno na závěr bakalářské práce (abych s nimi mohl pracovat v rámci praktické části a odkazovat se na ně). Poté se zaměřím na řešení možností automatizovaného testování webových aplikací a možných nástrojů pro usnadnění vývoje a údržby. Obě tyto části je potřeba dostatečně prozkoumat z důvodu následného zavedení v praktické části na základě požadavků. Na závěr zvolím vhodné technologie a nástroje pro tuto práci a volbu oargumentuji.

V praktické části nejprve provedu sběr a analýzu požadavků a navrhnu úpravy včetně aktualizovaného databázového modelu a upraveného schéma API (Application Programming Interface), také navrhnu způsob řešení konfigurace více prostředí pro nasazování. Poté již popíši postupně všechny kroky v rámci implementace všech požadavků a zavedení nástrojů. Následovat bude podrobný popis vytváření testů, které budou tvořit důležitý podpůrný prvek celého vývoje a popis nového způsobu nasazování aplikace včetně konfigurace více prostředí. Na závěr popíši kroky učiněné pro zveřejnění aplikace jako open-source a také uvedu možná rozšíření aplikace v budoucnu.

Cíle práce

Cílem této práce je navrhnout a implementovat rozšíření webové aplikace, která lektorce ÚP umožní efektivnější a pohodlnější každodenní práci pokrývající ještě více oblastí díky novým funkcím. Rozšíření proběhne na základě sběru a analýzy požadavků a následného návrhu příslušných změn včetně databázového modelu, schéma API a konfigurace více prostředí pro nasazování.

Dalším cílem je na základě rešerše zavedení vhodných pokročilých nástrojů pro usnadnění vývoje a údržby aplikace. Spolu s dalšími dvěma neméně důležitými cíli – zavedení několika prostředí pro nasazování a pokrytí důležitých částí automatizovanými testy API i UI (User Interface, konkrétně testy E2E, tedy End-to-End) – povedou všechny tyto kroky ke spolehlivějšímu a rychlejšímu dodávání nových verzí.

Cílovým bodem je rozšířená aplikace o všechny požadavky a zvolené nástroje nasazená v několika prostředích včetně produkce s automatizovaným testováním před nasazením. Zdrojové kódy také budou spolu s instrukcemi pro spuštění a dalším popisem zveřejněny pod open-source licencí.

Část I

Teoretická část

Aktuální řešení

V této kapitole stručně popíši verzi aplikace, která byla vytvořena v rámci bakalářské práce [1] a kterou budu v praktické části rozšiřovat. Popíši implementované funkční i nefunkční požadavky, použité technologie a také konfiguraci prostředí a způsob nasazování. Na závěr stručně shrnu, jaké možné plány na rozšíření byly v rámci bakalářské práce nastíněny.

2.1 Implementované požadavky

Pro další práci je třeba si vyjasnit, co už stávající aplikace poskytuje za funkcionalitu a jak je řešená – zaměřím se proto nejprve na implementované funkční požadavky a poté na vyřešené nefunkční požadavky. Krátce se vždy pokusím uvést, jak je příslušný požadavek v rámci aplikace řešen či jak souvisí s ostatními.

2.1.1 Implementované funkční požadavky

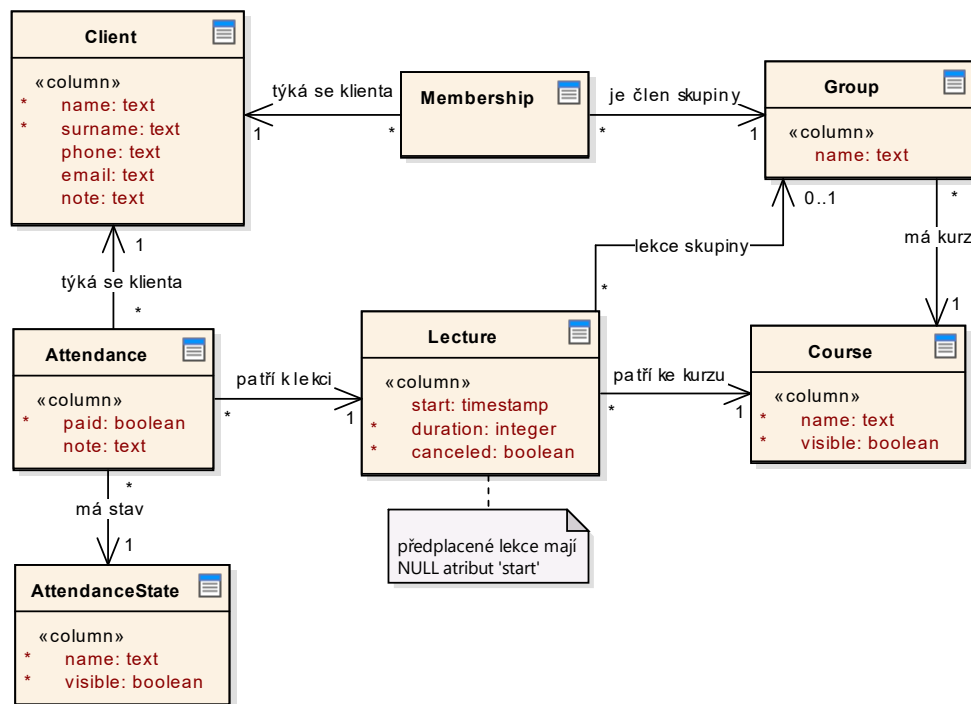
V této podsekci shrnu implementované funkční požadavky. Součástí této podsekce je také původní logický datový model aplikace z bakalářské práce [1] na obrázku 2.1 pro lepší pochopení domény.

Shrnutí implementovaných funkčních požadavků [1]:

- **evidence klientů a skupin:** evidování základních informací o klientovi a skupinách (skupina sdružuje několik klientů dohromady a patří k nějakému kurzu),
- **evidence lekcí klientů:** evidování všech informací o lekcích (včetně stavů účasti všech účastníků a stavu jejich platby za danou lekci), lekce mohou být pro jednotlivce nebo pro skupiny, každá náleží nějakému kurzu,

2. AKTUÁLNÍ ŘEŠENÍ

- **evidence předplacených lekcí:** předplacená lekce je řešená jako lekce, která nemá datum a čas konání,
- **evidence kurzů a stavů účasti:** pro použití při evidenci lekcí,
- **přehled lekcí pro aktuální den:** zobrazení pro dnešní lekce (kromě zrušených),
- **karta klienta a skupiny:** zobrazení všech informací o klientovi/skupině včetně všech lekcí,
- **upozornění na platbu příště:** když už klient nemá žádné předplacené lekce, zobrazí se u poslední placené lekce upozornění na fakt, že má příště zaplatit,
- **pořadové číslo lekce:** u lekce se zobrazí (automaticky vypočítáno), o kolikátou navštívenou lekci v pořadí se jedná,
- **týdenní přehled:** zobrazení lekcí jako v diáři.



Obrázek 2.1: Logický datový model z bakalářské práce [1]

2.1.2 Implementované nefunkční požadavky

Následuje shrnutí implementovaných nefunkčních požadavků [1]:

- **kompatibilita s webovými prohlížeči:** aplikace je plně funkční a kompatibilní s běžnými webovými prohlížeči (Google Chrome, Mozilla Firefox, Microsoft Edge, Apple Safari) v posledních verzích, důraz je především kladen na desktopový prohlížeč Mozilla Firefox, na kterém se aplikace používá primárně,
- **podpora široké škály zařízení:** aplikace je responzivní a korektně se zobrazuje na všech zařízeních používaných lektorkou – iPad s iOS 13.3 a 9,7palcovým displejem, Nokia 5 s Androidem 9.0 a 5,2palcovým displejem, notebook s Windows 10 s rozlišením 1920 × 1080 a 15,6palcovým displejem,
- **připravenost na rozšíření a údržbu:** kód byl tvořen s důrazem na budoucí možná rozšíření (např. použití konstant, respektování principu DRY („Don't repeat yourself“) ad.),
- **bezpečnost:** JWT (JSON Web Token) autentizace a další produkční konfigurace pro zabezpečení aplikace,
- **srozumitelné a jednoduché rozhraní aplikace:** iterativní návrh a implementace UI za neustálé spolupráce s lektorkou pro dosažení co nejlepší použitelnosti.

2.2 Použité technologie

V této sekci shrnu všechny podstatné použité technologie pro serverovou i klientskou část. Součástí bakalářské práce byl přehledný diagram nasazení pro lepší orientaci [1] – vzhledem k tomu, že se jádro architektury nemění, původní diagram zde neuvádím, na obrázku 7.2 z kapitoly návrhu 7 v praktické části je ale k vidění jeho aktualizovaná verze.

Toto shrnutí použitých technologií je také užitečné pro následnou práci, protože je třeba mít podrobný přehled o všech možných používaných nástrojích, frameworkích, knihovnách, doplňcích apod. Taktéž to čtenáři umožní bližší představu o výchozím stavu aplikace, která bude v rámci praktické části rozšiřována.

2.2.1 Serverová část

Serverová část aplikace [1] je napsána v Pythonu 3.6.5 s webovým frameworkem Django 2.0.5. Pro správu závislostí se používá pouze jednoduchý soubor `requirements.txt` obsahující specifikace přesných verzí knihoven (bez povolení jakkoliv malých aktualizací).

Aplikace vystavuje REST API postavené na frameworku Django REST framework 3.8.2. Na produkci se používá webový server Gunicorn 19.8.1 spolu s knihovnou WhiteNoise 3.3.1 pro efektivní servírování zkomprimovaných statických souborů [2].

Odděleně od aplikace běží také SQL (Structured Query Language) databáze PostgreSQL 9, se kterou Django komunikuje přes adaptér psycopg2.

2.2.2 Klientská část

Klientská část aplikace [1] je napsána v JS (JavaScript) ve standardu ECMAScript® 2018 a čistém CSS (Cascading Style Sheets). Pro správu závislostí se používá soubor `package.json`, v němž jsou verze přibližně poloviny knihoven definovány napevno (bez možnosti jakkoliv malé aktualizace), druhá polovina knihoven přijímá malé aktualizace.

Klientskou část aplikace lze klasifikovat jako:

- **SPA** (Single-Page Application), tedy aplikaci běžící přímo u klienta v prohlížeči nevyžadující znovunačítání při přecházení mezi stránkami [3] a
- **CSR** (Client-Side Rendering), tedy aplikaci, která je klientovi doručena jako jednoduchý HTML (Hypertext Markup Language) soubor s odkazy na JS/CSS (Cascading Style Sheets) soubory [4].

Je postavena na knihovně React 16.3 spolu s UI frameworkem Bootstrap 4.1 a související knihovnou Reactstrap 5.0, která umožňuje jednoduché použití Bootstrap komponent v Reactu [5]. Pro asynchronní požadavky na REST API využívá knihovnu axios 0.18.

Konfigurace celé klientské části stojí na nástroji create-react-app 1, který umožňuje vytvářet React aplikace bez počáteční konfigurace [6]. Vzhledem k použití Django bylo ale potřeba pro pohodlný vývoj pomocí příkazu `eject` „vysunout“ celou konfiguraci klientské části a následně pak pomocí knihoven webpack-bundle-tracker a django-webpack-loader propojit Django s nástrojem Webpack [1]. Webpack zde umožňuje mj. spouštět vývojový server pro klientskou část a vytvářet z jednotlivých modulů klientské části balíčky, které lze pak po zadaných transformacích servírovat na produkci pro běžné webové prohlížeče [7].

V případě vývoje na lokálním stroji se používá [1] pro serverovou část vývojový Django server a pro klientskou část webpack-dev-server, oba nabízejí podporu pro „hot reloading“ (tedy okamžité automatické projevení změn v kódu bez kompletního znovunačtení aplikace [8]) a v tomto ohledu bylo vše i díky zmíněným knihovnám zprovozněno.

2.3 Prostředí, testování a nasazování

Aplikace [1] je verzována v privátním repozitáři na serveru GitHub. Při každém nahrání nové revize na server (**push**) se na integračním serveru Travis CI (Continuous Integration) spustí sestavení aplikace, vytvoří se testovací databáze a spustí se základní testy (viz níže). Výsledné pokrytí kódu spočítané pomocí nástroje Coverage.py se poté nahraje na platformu codecov.io pro pokročilé statistiky o testování [9]. Pokud vše na Travisu proběhne v pořádku, začne nasazování na produkční server běžící na Heroku. Nasazení na Heroku probíhá tak, že se přímo na něm spouští celé sestavení aplikace znovu, zmigruje se databáze a aplikace se nasadí. I tento průběh lze sledovat přímo z Travis terminálu.

Spouštěné testy jsou základní a velmi jednoduché, otestují [1]:

- přidání klienta a uživatele do databáze přes Django modely (tedy ne tak, jak to může dělat uživatel, ale na mnohem nižší úrovni),
- zda Django uživateli zobrazí správnou stránku při příchodu do aplikace (neřeší, zda se pak vůbec JS aplikace vyrenderuje),
- základní funkčnost API požadavků – proběhne autorizace (a tedy získání JWT tokenu) a pokus o vytvoření nového klienta přes API.

Jak je vidět, testy byly skutečně pouze velmi povrchní, dalo by se říci, že se jedná o smoke testy. Také je třeba zdůraznit, že provedení **push** na repozitář mělo za následek okamžité nasazení na produkci, pokud sestavení a tyto základní testy prošly. To může být nedozírné následky. I proto se v dalších kapitolách této teoretické i praktické části budu zabývat možnostmi zlepšení.

2.4 Plán rozšíření z bakalářské práce

Na závěr bakalářské práce [1] bylo zmíněno několik možných rozšíření aplikace. Následuje jejich stručný přehled, v kapitole 6 se mimo jiné na některá z nich také dostane, případně bude vysvětleno, proč je daný požadavek vyřazen.

Přehled možných rozšíření [1]:

- **vylepšení předplacených lekcí:** pohodlnější způsob zaznamenávání předplacených lekcí, pro skupiny je to velmi krkolomné a nepohodlné, pro jednotlivce také,
- **kontrola časového konfliktu lekcí:** aby se dvě nezrušené lekce vzhledem k datu, času a délce trvání nijak nepřekrývaly,
- **vyhledávání v aplikaci:** např. vyhledávání klientů,
- **evidování zájemců o kurz:** pro plánování nových lekcí kurzů pro jednotlivce a skupiny,

- **evidence pomůcek a učebnic,**
- **testy:** doplnění dalších testů a vysoké pokrytí kódu,
- **migrace na nový React:** k vydání verze 16.3 došlo na konci vývoje aplikace v rámci bakalářské práce, např. došlo ke změnám v API (životní cyklus komponent) [10],
- **React Context API:** analýza možností využití Context API v rámci nového Reactu [10], zejména by pravděpodobně pomohlo snížit např. počet přístupů do API z klientské části napříč aplikací,
- **offline přístup:** analýza možností řešení offline přístupu, např. automatické ukládání do Google kalendáře, progresivní webové aplikace apod. a s tím související další oblasti jako SSR (Server-Side Rendering) – tedy klient obdrží od serveru HTML dokument připravený k vyrenderování, oproti CSR, kde by klient pro vyrenderování aplikace musel čekat na stažení a spuštění JS souborů [4].

Možnosti automatizovaného testování

Testování je důležitou součástí softwarového vývoje [11]. V této kapitole nejprve srovnám manuální a automatizované testování, poté se zaměřím na strukturu samotných testů – do jakých vrstev se dělí a kolik testů by v těchto vrstvách mělo být. Některé metodiky softwarového vývoje jsou zaměřené na způsob testování, proto také uvedu hlavní principy těchto metodik a s nimi související nástroje, které umožňují tyto metodiky zavést. Uvedu i další nástroje pro testování UI.

3.1 Srovnání manuálního a automatizovaného testování

Dříve, když trval vývojový cyklus několik měsíců až let, se obvykle testovalo převážně pouze manuálně [12]. Tedy na základě scénářů testeré prováděli testy [11]. Dnes, v době agilního vývoje a rychlých dodávek, je potřeba větší část testování provádět automatizovaně [12]. Tedy naprogramovat testy a poté je automaticky spouštět testovacím nástrojem [11]. Díky tomu se týmy mohou dozvědět o chybě v řádu sekund a minut místo dnů a týdnů [13].

Výhodou automatizovaného testování je:

- šetří čas, je rychlejší, umožní rychleji vydávat nové verze (kompletní manuální testování bylo úzkým hrdlem v životním cyklu vývoje softwaru) [12, 14, 15],
- odhalí chyby v dřívějších fázích, tedy snižuje náklady [12],
- dělá testování „zábavnější“ a umožní efektivnější manuální testování – odstraňuje běžnou neustále opakující se rutinu při manuálním testování [12, 16],

3. MOŽNOSTI AUTOMATIZOVANÉHO TESTOVÁNÍ

- dochází ke zpřesnění testů a vyšší spolehlivosti, protože se tester při neustálém opakování na všech operačních systémech a prohlížečích může splést nebo něco přehlédnout [12, 14],
- možnost vyššího pokrytí kódu testy a odhalení více chyb díky jednoduchému zavedení více permutací různých zařízení a operačních systémů [16],
- umožňuje zavést průběžnou integraci a dodávání [17].

Nevýhodou automatizovaných testů je:

- při nesprávném rozhodnutí o oblasti, kterou budeme optimalizovat, můžeme stovky hodin strávit při vytváření testů, které nakonec nebudou vůbec nacházet podstatné chyby [12],
- nevhodně napsané testy mohou dát falešnou naději, že vše funguje, ačkoliv se v aplikaci vyskytují závažné chyby [18],
- nikdy zcela nenahradí lidské pozorování (při manuálním testování) a nemohou garantovat přívětivost k uživateli či pozitivní uživatelský prožitek [14],
- vyšší pravděpodobnost falešně pozitivních výsledků, následná analýza problému je náročná, protože je třeba zjistit, zda se jedná o chybu aplikace či testu [16],
- nutnost neustálé údržby testů – při zavedení změny v aplikaci je třeba vše co nejdříve (ideálně okamžitě) projevit do testů [19].

Jak je vidět ze shrnutí výhod a nevýhod automatizovaného testování, manuální testování má v softwarovém vývoji stále své místo. Zejména kvůli lidskému faktoru a v případech, kdy se konkrétní test nemá spouštět opakovaně kvůli časové náročnosti vytvoření automatizovaného testu [11].

Na možnosti aplikace manuálního a automatizovaného testování lze také nahlížet z hlediska různých testovacích cyklů softwaru [11], tedy manuální testování je vhodné na počátku vývoje/iterace a na závěr při testech použitelnosti, kde je důležitý lidský faktor. Automatizované testování je pak vhodné pro fázi testování výkonu a také pro fázi regresních testů [11], ty se využívají při opětovném testování stávajících funkcí a vlastností aplikace při provádění změn, například rozšiřování jiných oblastí či opravě chyb [20].

Jak uvádí [11], je třeba nalézt pomyslný „rovnovážný bod“, který reprezentuje optimální poměr manuálních a automatizovaných testů vzhledem k ceně jejich vytvoření a počtu běhů testu – cena vytvoření automatizovaného testu je vysoká, pokud poběží jednou, ale rychle se snižuje s počtem opakování, naproti tomu manuální testování je levné, ale s každým během se cena zvyšuje kvůli délce běhu testu.

Díky vytvoření automatizovaných testů lze aplikaci průběžně testovat na integračním serveru při každé revizi [12], to umožní ještě rychlejší zpětnou vazbu, rychlejší vydávání verzí a spokojenost zákazníka [16]. Aplikace je totiž prakticky připravená na nasazení v každé revizi [21].

3.2 Struktura automatizovaných testů

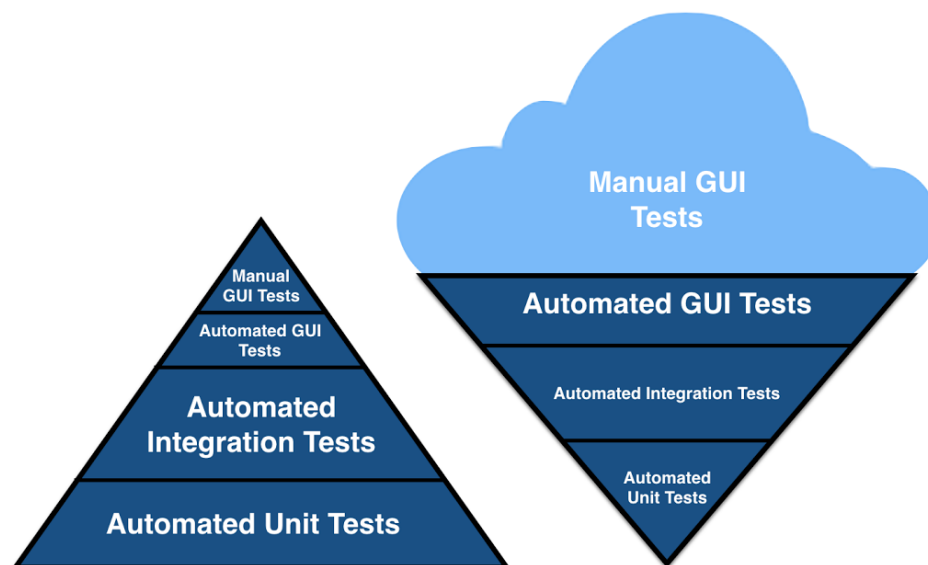
Automatizované testování prostupuje více úrovněmi softwarového projektu [17] a je důležité na počátku tvorby testů stanovit, kde se bude automaticky testovat [12]. Jednou z častých strategií zejména agilních týmů je strategie tzv. „testovací pyramidy“ [22]. V této sekci tuto strategii představím a popíši, jak se dle různých zdrojů aplikuje, zaměřím se také na další alternativní přístupy.

3.2.1 Testovací pyramida

Testovací pyramida rozděluje automatizované testy do několika oblastí [22], viz obrázek 3.1. Popis pyramidy vychází z [17, 22]. Obvykle by měly být jádrem automatizovaných testů unit testy, které ověřují, zda jednotlivé dílčí části kódu odpovídají požadavkům. Těchto testů by měla být většina. Následují testy komponent (např. přihlášení uživatele, tvorba účtu, objednávky) a integrační testy (ověří, že jednotlivé komponenty spolu interagují tak, jak bylo zamýšleno, např. data zákazníka jsou napříč celým procesem objednávky korektně přenášena). Jak je vidět na obrázku 3.1, vrstvy nejsou pevně definované a často některé splynou v jednu [13]. Pod vrcholem pyramidy se nalézají API testy a na vrcholu jsou UI testy (někdy nazývané End-to-End, funkční či E2E), které by měly typicky mít nejmenší podíl ze všech automatizovaných testů vzhledem k jejich náročnosti na tvorbu a křehkosti při změnách UI. Mimo pyramidu, případně na úplný vrchol, lze pak zařadit samotné manuální testy UI.

Konkrétní aplikace této strategie závisí na vlastnostech projektu, týmu a požadavcích. Přesto se pokusím uvést obecná doporučení různých autorů. Čím níže z pohledu test pyramidy se podaří vývojářům dostat, tím lépe pro budoucí práci – lépe začít např. s API, než s UI – případně lze pracovat ve více lidech na více úrovních paralelně [12]. Pokud ale aplikace už běží a postrádá jakékoliv testy, je ideální začít s tvorbou testů na vrcholu pyramidy pro kritické klíčové části byznysu [21]. Pro zbytek je vhodné využít nižší úrovně testů [24].

Mnoho organizací má i přes testovací pyramidu většinu automatizovaných testů postavených nad UI vrstvou [18], výhodou je dohled nad výsledným produktem, který je pak používán, nevýhodou křehkost a nesnadné dohledávání původu chyb zachycených při testech UI, v případě použití AJAX (Asynchronous JavaScript and XML) je také mnohem složitější vytvořit deterministické UI testy [25]. Testovací pyramida je pak obrácená a obvykle se tomuto přístupu vzhledem k tvaru obrácené pyramidy říká „zmrzlinový kornout“ [26], viz opět obrázek 3.1.



Obrázek 3.1: Srovnání strategie „testovací pyramidy“ (vlevo) a „zmrzlinového kornoutu“ (vpravo) [23]

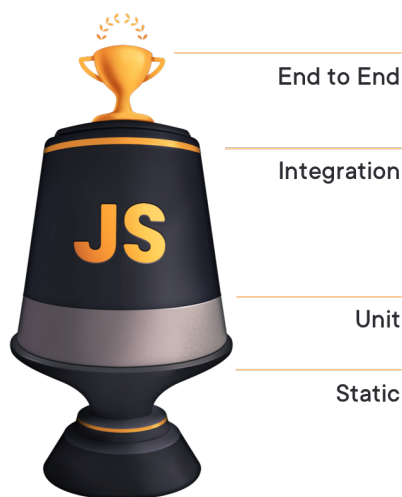
Při volbě správné části k testování je třeba se řídit tím, která část je pro byznys zákazníka důležitá [12], nikoliv pouze například co nejvyšším pokrytím kódu testy [18]. V případě, že se UI testy použijí pro klíčové funkce aplikace, můžeme zajistit, že i přes případné drobnější chyby funguje ta nejdůležitější část aplikace korektně, a to včetně všech vrstev jako např. napojení na API a databázi, to umožní častější dodávání nových verzí zákazníkovi s větší jistotou fungování, to vše je ale třeba dělat s vědomím jednotlivých úrovní testovací pyramidy a důsledků použití UI testů [27]. Je tedy vhodné ve vyšší vrstvě testovat dva typy průchodů – nejhorší a nejideálnější – a pro zbytek hraničních případů využít vrstvy nižší [28].

Strategie testovací pyramidy je často špatně interpretována a může například vyústit v napsání naprosto všech možných unit testů a poté přesunutí do vyšší vrstvy atd., což pro některé týmy může být vhodné, pro jiné ale zbytečně komplexní a drahé [29]. Principem pyramidy je naznačit, že testů na vyšší vrstvě má být méně než na nižší [29]. Ačkoliv se princip testovací pyramidy stává standardem pro agilní týmy, je třeba rozumět principům, se kterými tento přístup přichází, a na základě toho zavést způsoby testování pro konkrétní projekt [29]. Na závěr je třeba říci, že veškeré zmíněné termíny, názvy a definice nejsou rigorózní a často panují různé názory např. na rozdělení testovací pyramidy, definici UI testování (někdy synonymum end-to-end testování, jindy UI testy jednotlivých částí pomocí unit testů) a mnoho dal-

šího, při tvorbě testů v rámci softwarového produktu je tedy vždy třeba jasně vymezit a definovat rozsah testů, sjednotit terminologii a na všem se domluvit [13].

3.2.2 Alternativní přístupy

Alternativním přístupem pro testování klientské části je strategie tzv. „testovací trofeje“ [30], která se skládá ze 4 částí, od té nejnižší: statické testy (založené na statickém typování a linterech, viz sekce 4.4 a 4.3), unit testy (cílené na kritické části aplikace), integrační testy (ověřující, že vše spolu korektně spolupracuje) a end-to-end funkční testy (simulace chování uživatele při důležitých průchodech aplikací prostřednictvím automatizovaného klikání – tedy UI testy z testovací pyramidy). Jak ale uvádí [31], tento přístup opět může pro některé týmy a aplikace být vhodný, pro některé nikoliv a bez dostatečné znalosti konkrétního kódu aplikace a problematiky může následování nejen tohoto vzoru vyústit ve slepou uličku.



Obrázek 3.2: Strategie „testovací trofeje“ [30]

3.3 Metodiky a nástroje pro automatizované testování

Svět testování je ovlivňován několika metodikami vývoje softwaru, především TDD (Test-Driven development) a BDD (Behaviour-Driven development) [32]. S těmito metodikami souvisí také různé nástroje pro automatizované testování podporující příslušný způsob vývoje, o nich se též zmíním. Na závěr uvedu ještě další nástroje pro UI testování.

3.3.1 TDD

Základním principem TDD je napsat nejprve testovací případy (přímo v programovacím jazyce) a až poté implementovat kód, který zařídí splnění těchto případů [33]. Výsledkem je vyšší kvalita a flexibilita kódu (tím pádem lze pak jednoduše provádět např. refaktoring) a také vysoké pokrytí kódu [33]. Nevýhodou TDD je, že změny ve fungování aplikace mohou mít velký dopad na testovací případy, testovacím případům také rozumí pouze lidé se znalostí programovacích jazyků [33]. Vzhledem k zaměření testů na konkrétní implementaci a nikoliv chování je ale snadnější najít v případě TDD testů konkrétní chybu [33]. TDD testy se zaměřují zejména na nejnižší vrstvu v testovací pyramidě, tedy unit testy [32], principy TDD se ale mohou aplikovat napříč všemi vrstvami [28].

Mezi nástroje pro podporu TDD patří xUnit frameworky, např. JUnit, TestNG, NUnit [33]. Pro **JS** se dle [34] a [35] nejvíce používá Jest a Mocha, pro **Python** se používá PyTest nebo unittest [36].

3.3.2 BDD

BDD rozšiřuje TDD a místo psaní testovacích případů se zapisují požadované scénáře chování, později je opět implementován samotný kód aplikace zařizující její předem specifikované chování [33]. Díky zápisu scénářů v jazyce Gherkin, který vychází z přirozeného jazyka, je možná jednoduchá spolupráce mezi vývojáři, testery, analytiky a zákazníkem (i bez znalosti programovacích jazyků) [33, 37], vytvořené scénáře tvoří prakticky základ pro dokumentaci funkcí aplikace [38]. Při vývoji aplikace totiž často dochází k nedorozuměním ohledně výkladu různých požadavků [38]. Výhodou BDD je také zaměření na samotné chování aplikace, nikoliv na přemýšlení o implementaci v kódu – chování aplikace je hlavní prvek, na který se vývojáři a testéři zaměřují, to umožňuje držet se lépe požadavků zákazníka [33]. BDD testy se zaměřují zejména na prostřední vrstvy v testovací pyramidě, tedy API testy [32].

Mezi nástroje pro podporu BDD patří frameworky jako SpecFlow, Cucumber, MSpec [33]. Pro **JS** se používá Cucumber.js, pro **Python** se používá Behave [39]. Ze scénářů lze také vytvářet vždy aktuální dokumentaci a reporty díky nástrojům jako TestComplete [38].

BDD se často používá spolu s TDD, protože dokáže na vyšší úrovni prověřit korektní fungování aplikace a poskytnout tak vyšší důvěru ve výsledný produkt [37] – takové testy tedy prověří nějaké chování aplikace a doplněním o TDD testy na nižších úrovních se dotestují specifické části.

3.3.3 Další nástroje

Jak jsem již zmínil v předchozí podsekci, BDD se zaměřuje na vyšší vrstvy testovací pyramidy, pro UI testování se tedy často používá s nástroji jako Selenium [28]. Selenium je sada nástrojů a knihoven pro automatizované testování

webových aplikací [40]. Prostřednictvím tzv. WebDriveru, který je implementován jednotlivými tvůrci prohlížečů, je možné interagovat s webovou stránkou prostřednictvím běžných prohlížečů přímo z kódu (a to nejen v běžném módu, ale také tzv. „headless“ módu, kde prohlížeč běží bez GUI) [41, 40]. Pro „headless“ mód byl do nedávné doby nejpopulárnější volbou PhantomJS, vzhledem k postupné implementaci této funkce do předních webových prohlížečů byl ale jeho vývoj zastaven a přechází se na běžné prohlížeče, protože právě v těch koncový uživatel pracuje [13, 42].

Selenium je mocný nástroj, nevýhodou je ale poměrně zdlouhavý zápis skriptů pro výběr elementů, práci s výjimkami a časováním, proto se často volí nadstavby zaobalující Selenium, které umožňují jednodušší zápis a práci, např. Selenide [41]. Také je možné využít komplexní testovací frameworky, které nabízí mnohem více nástrojů pro práci s testy, z nichž některé jsou také postavené na Seleniu, z mnohých používaných například Katalon Studio či Robot framework [43, 44].

Selenium se postupem času stalo průmyslovým standardem pro UI testování, a to zejména díky jednoduchosti použití, kompatibilitě (s mnoha prohlížeči, možnost použití s mnoha programovacími jazyky) a popularitě [45]. Jak ale uvádí [46], v případě asynchronních aplikací je Selenium složitější používat vzhledem k principu jeho fungování, protože je třeba vždy explicitně čekat na objevení elementu na stránce, problémem ale je, že nevíme, jak dlouho čekat. I z toho důvodu zde autor nabízí alternativu v podobě novějšího Cypress, který nabízí mnohem lepší práci s asynchronními aplikacemi, problémem ale je možnost použití pouze s JS, malá komunita a popularita a méně návodů [46]. Další nevýhodou Cypress byla podpora výhradně prohlížeče Google Chrome, to se ale v únoru 2020 změnilo [47], naopak výhodou uváděnou např. v [48] byla lepší dokumentace oproti Seleniu, dokumentace Selenia ale byla přepsána a na podzim roku 2019 nasazena [49].

Nástroje pro usnadnění vývoje a údržby

Postupem času se v rámci dalšího vývoje a rozšiřování aplikace ukázalo, že je třeba použít pokročilé nástroje pro usnadnění samotného vývoje a údržby. Cílem této sekce je nastínit způsob jejich výběru. Jedním z hlavních požadavků je, aby vše potřebné bylo zdarma, neuvažuji zde tedy nástroje nenabízející alespoň nějakou formu bezplatného používání na dobu neurčitou (tedy nikoliv jen pro studenty). Při procházení ceníků budu brát také v úvahu fakt, že repozitář s projektem je open-source (to je jeden z cílů této práce).

Nejprve se zaměřím na nástroje pro monitorování chyb a správu logů. Při volbě těchto nástrojů budou případně specifikovány další požadavky. Na základě průzkumu pak uvedu nalezené nástroje, jejich vlastnosti a seřadím je orientačně podle popularity na základě stránky StackShare, která poskytuje [50] možnost sdílet používané nástroje a technologie firem a jednotlivců pro jejich projekty, nástroje a technologie srovnávat, hodnotit, řadit dle popularity apod.

Poté se zaměřím na výhody a nevýhody statického a dynamického typování jazyků a možnosti zavedení anotací typů v projektu ÚP. V poslední kapitole se pak zaměřím na možnosti statické analýzy kódu projektu, rozdělení jednotlivých možných nástrojů a popsání jejich vlastností. Obě tyto kapitoly ukazují další možnosti usnadnění vývoje a údržby aplikace.

Některé řešené oblasti souvisí také s respektováním metodiky s názvem „The Twelve-Factor App“ doporučené (a také vytvořené) v rámci Heroku metodikou. Tato metodika se zabývá stavěním webových aplikací, které nabízejí deklarativní způsob spouštění, jsou maximálně přenosné mezi prostředími, minimalizují rozdíl mezi vývojovým a produkčním prostředím a nasazují se do cloudových platforem [51].

4.1 Monitorování chyb

Pro každou aplikaci je důležité monitorovat chyby, díky tomu (nehledě na to, zda uživatel nahlásí problém a bude jakkoliv konkrétní) je možné pak chyby snadněji reprodukovat a opravit [52]. Nástroje pro monitorování chyb umožňují upozornit vývojáře na výskyt chyby, poskytnout mu kompletní informace o chybě a kontextu, ve kterém nastala [52].

Minimální požadavky na nástroje jsou:

- zdarma pro použití na neomezenou dobu jak na Heroku (aplikace v rámci této práce), tak mimo Heroku (další aplikace např. v rámci ÚP) – tento požadavek je zde kvůli existenci nástrojů jako např. Airbrake, který na Heroku nabízí plán zdarma [53], ale ve svém ceníku jej nenabízí [54], tedy takové nástroje neuvažuji,
- SaaS (Software as a Service) – tedy aplikace hostovaná provozovatelem služby [55],
- možnost použití pro klientskou (JS s Reactem) i serverovou část (Python s Djangoem).

Pro monitorování chyb existuje mnoho nástrojů, podle StackShare [56] volím 4 nejpoužívanější, které splňují všechny požadavky. I přes mnoho dalších funkcionalit těchto nástrojů se zde zaměřuji na primární účel, tedy monitorování chyb – nehledě na to, jaké další integrace a funkce jsou poskytnuty zdarma.

Nejpoužívanější nástroje podle StackShare (řazeno od nejpoužívanějšího) [56] splňující zmíněné požadavky:

1. **Sentry**: zdarma nabízí 5 000 událostí/měsíc pro neomezený počet projektů, 7 dní historie dat [57], podporuje React i Django [58],
2. **Rollbar**: zdarma nabízí 5 000 událostí/měsíc pro neomezený počet projektů, 30 dní historie dat [59], podporuje React i Django [60],
3. **Bugsnag**: zdarma nabízí 7 500 událostí/měsíc pro neomezený počet projektů, 7 dní historie dat [61], podporuje React i Django [62],
4. **Honeybadger**: zdarma nabízí 12 000 událostí/měsíc pro neomezený počet projektů, 15 dní historie dat [63], podporuje React [64] i Django [65].

Jiný přístup k monitorování chyb na klientské části nabízí nástroje jako LogRocket – ten k zachyceným chybám přidá i nahrané video s kroky uživatele vedoucími k dané chybě (ve videu je obrazovka zachycující přesně to, co uživatel viděl) [66]. LogRocket nabízí zdarma 1 000 nahraných sezení uživatelů/měsíc (sezení je jedno kontinuální používání aplikace daným uživatelem) [67], 14 dní historie dat, podporuje React [68].

4.2 Správa logů

Nasazené aplikace generují mnoho logů z různých procesů, v případě Heroku jsou všechny tyto logy agregovány do jednoho kanálu a nabízí možnost uchování posledních 1 500 logů nejdéle 1 týden [69]. V případě, že chceme přístup k většímu počtu logů či ke starším (zde je především problém s limitem pro počet logů, který je nízký), je třeba využít buď možnost napojení logů na některý z doplňků na Heroku, případně logy rovnou přímo přesměrovávat do jiné služby [69]. Tyto nástroje pro správu logů pak umožňují spravovat velké množství agregovaných logů generovaných z mnoha typů zařízení a serverů – nad nimi provádět mj. různé dotazy, vyhledávání, pohledy, případně i analýzy či upozornění na nějaké události [70]. Uvedený způsob řešení práce s logy též odpovídá jednomu z pravidel metodiky „The Twelve-Factor App“ [51].

Minimální požadavky na nástroje jsou:

- zdarma pro použití na neomezenou dobu pro logy z Heroku,
- SaaS – tedy aplikace hostovaná provozovatelem služby [55],
- upozorňování na události e-mailem,
- historie logů alespoň na 7 dnů.

Pro správu logů existuje mnoho nástrojů, podle StackShare [71] volím 2 nejpoužívanější, které splňují všechny požadavky. I přes mnoho dalších funkcionalit těchto nástrojů se zde zaměřuji na primární účel, tedy ukládání logů, vyhledávání a upozorňování – nehledě na to, jaké další funkce jsou poskytnuty zdarma (během hledání se ukázalo, že právě požadavek upozorňování většina nástrojů zdarma neposkytuje, filtrem dle požadavků tedy nakonec prošly pouze 2 nástroje).

Nejpoužívanější nástroje podle StackShare (řazeno od nejpoužívanějšího) [71] splňující zmíněné požadavky:

1. **Papertrail**: zdarma nabízí 50 MB logů/měsíc, 7 dní historie logů (vyhledávání ale jen pro poslední 2 dny), nastavitelná upozornění [72], možnost použít Heroku doplněk pro jednoduché nastavení (nabízí dokonce více uložených logů – 10 MB logů/den) [73],
2. **Logentries**: zdarma nabízí 5 GB logů/měsíc, 7 dní historie logů [74], nastavitelná upozornění [75], možnost použít Heroku doplněk pro jednoduché nastavení [76].

4.3 Statické typování

V programovacích jazycích zajišťuje typový systém, že se v kódu pracuje s očekávanými hodnotami – existují dva typové systémy, dynamický a statický [77].

V následující podsekcí se zaměřím na jejich srovnání a výhody zavedení typových anotací pro jazyky s dynamickým typováním. Poté se v dalších podsekcích zaměřím na možnosti přidání těchto anotací na klientskou a serverovou část ÚP.

4.3.1 Srovnání s dynamickým typováním

Staticky typované jazyky provádějí typovou kontrolu při kompilaci, dynamicky typované až při běhu [77]. Jak uvádí autoři [77], vedou se neustálé debaty o nákladech a přínosech jednoho či druhého typu. Zastánci statického typování argumentují detekováním chyb před spuštěním, rychlejším během, větší srozumitelností kódu, možností pokročilých optimalizací kompilátorem [77] a také údržbu kódu v dlouhodobém horizontu [78]. Dynamicky typované jazyky jsou naopak vyzdvihovány pro svou vhodnost při procesu tvorby prototypů (umožní rychle psát a spustit kód bez vynaloženého úsilí psáním typových anotací), nenutí programátory explicitně omezovat produkováné/konzumované hodnoty výrazů, to usnadňuje psaní flexibilního kódu s využitím dynamického chování (např. reflexe) [77].

V rámci aplikace ÚP se používá JavaScript a Python, oba tyto jazyky jsou dynamicky typované [1]. V rámci následujících podsekcí se zaměřím na možné způsoby zavedení statického typování v obou těchto jazycích, nejprve se zde ale budu věnovat dalším výhodám zavedení typových anotací do dynamicky typovaných jazyků.

JS je v současné době základem mnoha webových projektů a jsou zde tři firmy, které statické považování považovaly za natolik důležité, že se rozhodly investovat do statických typových systémů právě pro JS: nejprve Google vydal Closure, pak Microsoft vydal TypeScript a nakonec Facebook vydává Flow [77].

Na výhody zavedení zmíněných nástrojů se lze dívat z několika stran. Jeden z pohledů nabízí studie „*To Type or Not to Type: Quantifying Detectable Bugs in JavaScript*“ [77], kde se její autoři zaměřují na zodpovězení otázky, kolik procent veřejných chyb v kódech dokáže Flow (verze 0.30) a TypeScript (verze 2.0) odhalit. Pro práci zvolili vzorek opravených chyb z veřejných GitHub repozitářů (na základě statistických metod), každou chybu doplnili o anotace a testovali, zda tyto nástroje chybu odhalí – ta by pak vůbec nemusela být do veřejného repozitáře zanesena. Ukázalo se, že oba nástroje naleznou významné procento veřejných chyb, oba totiž odhalily shodně 15 % chyb (každý nástroj odhalil 60 z 400 chyb, z toho oba odhalily 57 stejných chyb). K tomu je potřeba dodat několik faktů, které sami autoři uvádějí – vzhledem k povaze studie, jež je zaměřená pouze na veřejné chyby v repozitářích, dochází k podcenění některých dopadů těchto nástrojů (kvůli kterým jsou mj. také používány). Mnoho z chyb, které tyto nástroje odhalí, se děje během samotného privátního vývoje, naproti tomu zvolené veřejné chyby plynou převážně z nedorozumění při specifikaci požadavků, což typové systémy ne-

mohou detekovat. Statické typování také vylepšuje srozumitelnost programu, umožňuje lepší navigaci v kódu a inteligentní doplňování kódu, slouží jako dokumentace. Jak autoři sami uvádějí, je třeba brát také ale v úvahu fakt, že anotování si bere svou cenu za práci (v rámci studie se zabývali měřením této ceny a srovnáním obou nástrojů, vzhledem k tomu, že oba nástroje od té doby prošly mnoha změnami, konkrétní výsledky zde nebudou zmíněny). Autoři zde také vyzdvihávají projekt DefinitelyTyped (viz následující podsektce 4.3.2) obsahující definice typů pro mnoho knihoven, ten se využívá pro TypeScript.

Dodání těchto typových anotací, jak již bylo zmíněno, stojí čas a může mírně snížit na počátku produktivitu týmu, jak ale uvádí [78], omezení v podobě typů je z dlouhodobého horizontu zejména pro větší aplikace velmi žádoucí.

4.3.2 Klientská část

ECMAScript, jakožto standard, ze kterého vychází JavaScript, nijak nestandardizuje statické typování, ačkoliv v minulosti bylo několik pokusů o to jej zavést, není ale vyloučeno, že se v budoucnu do standardu typy dostanou [79]. Pro typovou kontrolu v JS se nejvíce používá TypeScript (z dílny Microsoftu) a Flow (Facebook) [80]. Nejedná se ale o stejné typy nástrojů.

Flow je pouze nástroj pro typovou kontrolu, kde se pomocí definované syntaxe typy zapisují do běžného JS a výsledný kód se transpiluje (překládá pomocí nástroje Babel) do čistého JS, TypeScript je přímo jazyk postavený jako rozšíření běžného JS (mj. o statické typování) a pomocí TypeScript kompilátoru (resp. transpileru) se překládá do JS [81, 80].

Výhodou Flow oproti TS je jeho pokročilé odvozování typů („type inference“) pomocí analýzy datových toků („data flow analysis“) – TS sice odvozování nabízí také, ale nikoliv tak pokročilé, ve výsledku tedy u Flow není potřeba uvádět všechny typy a přesto dostaneme korektní chybová hlášení [80, 82]. To znamená, že s menším úsilím získáme v případě Flow větší pokrytí kódu typovou kontrolou [83]. Jak ale autor [82] uvádí, přesto má smysl věnovat čas explicitnímu psaní typů pro striktnější kontrolu tak, aby Flow nějaký problém neminul.

Výhodou TS je rozsáhlá databáze typových definic pro JS knihovny DefinitelyTyped oproti velmi malé databázi flow-typed – tyto definice knihoven třetích stran se používají v případě, že knihovna sama o sobě nepoužívá anotace typů [80].

Protože je Flow z dílny Facebooku, nabízí vestavěnou podporu pro React [80]. V TypeScriptu je naopak napsaný celý populární framework Angular a jedná se o primární jazyk při vývoji v tomto frameworku [84].

Poměrně nedávnou novinkou je možnost transpilovat TS do čistého JS také pomocí nástroje Babel (jako Flow), ale s několika omezeními: všechny anotace typů se smažou a při transpilaci se neprovádí žádná typová kontrola, také

není podporováno několik jazykových konstruktů TS, které se ale dají nahradit alternativami [85]. Babel tedy stejně jako kompilátor TS transpiluje kód do čistého JS (i když bez typové kontroly), ale navíc nabízí obrovské množství doplňků [85]. V případě, že chtěl dříve vývojář Babel použít, bylo možné Babel (nikoliv úplně jednoduše) začlenit do vývojového procesu a výstup TS kompilátoru zaslat do Babelu [85]. Nyní, s novým Babel 7, může využívat během vyvíjení jednoduché a rychlé transpilování do JS (rychlejší než TS kompilátor, který ještě musí provést typové kontroly) a typovou kontrolu pomocí TS kompilátoru vyvolat, až bude chtít [85].

Na závěr je třeba říci, že se v poslední době objevuje mnoho firem, týmů a projektů včetně např. Jest či Yarn přecházejících právě z již zaběhnutého Flow na projektu na TypeScript – zejména kvůli rozsáhlé databázi DefinitelyTyped a naopak malé databázi flow-typed (případně také způsobu použití jednotlivých definic, v případě DefinitelyTyped se instalují jako běžné knihovny pomocí balíčkovacího systému, kdežto u flow-typed se stahují definice přímo do repozitáře jako soubory), pomalému vývoji samotného Flow, pomalé integraci Flow do editoru (a naproti tomu výborné podpoře editorů v případě TS) či obecně problémům se spouštěním a prací s Flow [86, 87, 88]. Jak ale uvádí vývojáři Flow [89], na některé ze zmíněných problémů se v tomto roce 2020 budou zaměřovat.

4.3.3 Serverová část

Situace pro Python je v mnohém odlišná od té v JS. Poměrně nedávno (na konci roku 2016) získal Python 3.6 kompletní nativní podporu pro typové anotace, a to včetně různých pokročilých datových typů díky modulu typing ze standardní knihovny [90]. Není tedy třeba řešit jako v případě JS různé přístupy, způsoby anotace, rozšíření jazyka apod.

Anotace tedy máme, pro samotnou typovou kontrolu je třeba použít jeden z nástrojů pro typovou kontrolu v Pythonu – referenčním nástrojem pro tuto typovou kontrolu je mypy (ještě ale není stabilní, je v beta verzi, ale již několik let používán na produkci např. v Dropboxu [91]) [90]. Tým ze zmíněného Dropboxu (jehož členem je i autor samotného Pythonu Guido van Rossum) stojí za nástrojem mypy a přispívá i do dalších projektů týkajících se statického typování v Pythonu, Dropbox postupně anotuje serverovou část svého kódu v Pythonu čítající dnes přes 4 miliony anotovaných řádků kódu [92]. Alternativou k mypy je např. nástroj Pyre (Facebook) či pytype, liší se např. v rychlosti či podpoře odvozování typů z kódu bez anotací [90, 93]. Další možností je použít vestavěnou typovou kontrolu pro Python v IDE (Integrated Development Environment), např. v PyCharm (nebo Atom pomocí doplňku), jak ale doporučují autoři [90, 94], je vhodné tuto kontrolu doplnit i o některý z již zmíněných nástrojů pro typovou kontrolu – protože se liší v implementaci typové kontroly a mohou tedy zachytit různé problémy.

Vývoj typových anotací v Pythonu pokračuje, přišlo se např. na dva problémy: nemožnost používat v anotacích typy, jejichž definice byla ve zdrojovém kódu později a také fakt, že přítomnost samotných anotací má negativní vliv na dobu spouštění programu, opravy přišly v Pythonu 3.7, chování je ale třeba explicitně povolit, ve výchozím stavu bude povolené až od Pythonu 4.0 [95].

Co se týče typových anotací pro knihovny třetích stran, používá se projekt `typeshed`, kde jsou jak definice typů (nazývané „stubs“) pro standardní knihovny Pythonu, tak i pro některé knihovny třetích stran (pokud je tyto knihovny nemají již rovnou zabudované přímo v sobě), pro knihovny třetích stran může být definice také distribuována zvlášť jako balíček pro instalaci [93, 96]. Definice z `typeshed` jsou automaticky přibaleny jako součást nástrojů pro typovou kontrolu (`mypy`, `pytype` a dokonce i `PyCharm`) [97]. Pro automatickou základní definici typů lze použít nástroje, které zvládnou vygenerovat různě pokročilé definice – nástroje jako `PyAnnotate`, `MonkeyType` či přímo `mypy` (příkazem `stubgen`) – generování probíhá na základě samotného kódu, testů či dokonce přímo z běhu programu [90].

Existují také nástroje jako `pytypes` či `pydantic`, které typy validují za běhu a umožňují tak např. zobrazit uživateli srozumitelnou chybu [90, 98].

4.4 Statická analýza kódu

Statická analýza kódu je efektivní nástroj pro posouzení kvality kódu softwarového projektu (např. konzistence, čitelnost, zranitelnosti, rychlost, pokrytí testy) a předvídání potenciálně vznikajících problémů (tzv. „code smells“) bez spuštění samotného kódu [99, 100]. Samotný kód je analyzován vůči sadě pravidel a standardů [100].

Na základě vlastního průzkumu nástrojů zde zavádím vlastní dělení na nástroje představující vyšší a nižší vrstvu statické analýzy kódu, vysvětlení důvodů vedoucích k tomuto dělení následuje vzápětí.

V první podsekci se zaměřím na rešerši nástrojů na vyšší vrstvě, které umožňují průběžnou kontrolu kvality kódu. Jak je vidět z [101, 102, 99, 103, 104], v této oblasti není úplně jednoznačná terminologie – těmito nástroji se často též říká nástroje pro posuzování kvality kódu, zaměřují se třeba i na pokročilou práci s manuálním posuzováním kódu ad., někdy se též nazývají nástroje pro *automatické* posuzování kvality kódu, což už je k této sekci bližší. V této první podsekci se tedy zaměřím na ty nástroje, které poskytují právě automatickou průběžnou kontrolu kvality kódu a dávají poté zpětnou vazbu pro kód aplikace.

Ve druhé podsekci se zaměřím na nástroje z nižší vrstvy (tzv. linter), na kterých často nástroje z vyšší vrstvy stavějí (proto také dělení na vyšší a nižší vrstvu pro lepší orientaci). Tyto nástroje se používají při samotném vývoji na lokálním zařízení a poté se mohou spouštět např. na integračním serveru.

4.4.1 Průběžná kontrola kvality kódu

Na zmíněné vyšší vrstvě se nalézají nástroje, které jsou provozované jako služby automaticky kontrolující kvalitu kódu. Obvykle jsou to služby typu SaaS, případně On-Premise (tedy opak SaaS [105]) [106]. Tyto nástroje lze do procesu vývoje zapojit a zlepšovat tak kvalitu kódu už od počátku – např. nastavit hranice pro kvalitu kódu vzhledem k pokrytí kódu testy a nulovému počtu nalezených problémů, na základě toho se pak může třeba PR (Pull Request) změně automaticky označit jako úspěšný nebo nikoliv [99]. To pak snižuje množství diskuzí mezi vývojáři, protože je zavedeno objektivní hodnocení kódu [99]. Nejprve uvedu minimální požadavky na nástroje pro zúžení výběru:

- zdarma pro použití na neomezenou dobu,
- integrace s GitHub,
- podpora JS, TS (TypeScript) a Python (ne nutně všech naráz),
- SaaS – tedy aplikace hostovaná provozovatelem služby [55].

Při průzkumu nástrojů na této vyšší vrstvě jsem zjistil, že se prakticky dají dále rozdělit na dva typy podle nabízených možností:

1. Nástroje založené z valné většiny především na integraci mnoha různých nástrojů z nižší vrstvy (lintery, viz podsektce 4.4.2), případně pouze na základních pravidlech pro kontrolu kódu jako např. délka metod, počet řádků souboru, složitost metod, duplikace apod. Jak uvádí [105], některé nástroje upřednostňují použití předvytvořených pravidel z jejich dílny místo integrace nástrojů kvůli lepší rozšiřitelnosti a správě pravidel, ve výsledku pak tedy nabízejí alternativu ke službám integrující všechny standardní nástroje. Možná je také kombinace obou přístupů nebo funkce pro vytváření vlastních jednoduchých pravidel na míru projektu.
2. Nástroje založené na pokročilé hloubkové analýze kódu různými způsoby. Ty mohou být doplněné o možnost vytváření vlastních stejně pokročilých pravidel na míru projektu. Také mohou nabízet volitelnou integraci nástrojů z nižší vrstvy.

Zde je na místě okomentovat, proč je toto rozdělení důležité. Nástroje prvního typu prakticky jen sdružují mnoho různých nástrojů na jednom místě, jejichž výsledky agregují a třídí přehledně na jednom místě. Právě toto sjednocení různých nástrojů do jedné služby je jejich přidaná hodnota oproti tomu, kdy by vývojář tyto nástroje spouštěl jednotlivě (nebo by ani u sebe nic pustit nemohl, protože se jedná pouze o pravidla z dílny dané služby). Nástroje druhého typu nabízejí pokročilou hloubkovou kontrolu kódu způsobem, jakým

to běžné lintery, na kterých stojí nástroje prvního typu, nezvládnou (plyne to ze způsobu jejich fungování naznačeného v 4.4.2) [107]. Díky tomu umožňují vyhledat velmi závažné problémy, chyby či zranitelnosti, které by jinak mohly být opomenuty [107, 108]. To zvládnou díky vlastním pokročilým pravidlům a analýzám založených např. na umělé inteligenci [108]. Jejich výhodou také je, že obvykle najdou reálné problémy místo mnoha nedůležitých, které mohou poskytnout nástroje prvního typu – tuto myšlenku vystihují autoři DeepSource [108]: „*DeepSource is designed to understand the context of your code and filter out the noise from the results. This prevents warning blindness and encourages developers to take action on issues that matter.*“

Nástroje prvního typu splňující minimální požadavky:

- **Codacy:** zdarma [109], podporuje JS, TS i Python [110], GitHub integrace pro PR i commity [111], založeno na integraci mnoha nástrojů do jedné služby [112],
- **Code Climate:** zdarma [113], podporuje JS, TS i Python [114], GitHub integrace pro PR i commity [115], založeno na integraci mnoha nástrojů do jedné služby a dalších základních pravidel [116],
- **CodeFactor:** zdarma [117], podporuje JS, TS i Python [118], GitHub integrace pro PR i commity [118], založeno na integraci mnoha nástrojů do jedné služby a dalších základních pravidel [119],
- **Code Inspector:** zdarma [120], podporuje JS, TS i Python [121], GitHub integrace pro PR i commity [121], založeno pouze na základních pravidlech [121, 122],
- **codebeat:** zdarma [123], podporuje JS, TS i Python [124], GitHub integrace pro PR i commity [124], založeno pouze na základních pravidlech [106],
- **HoundCI:** zdarma [125], podporuje JS, TS i Python [125], GitHub integrace jen pro PR [125], založeno na integraci mnoha nástrojů do jedné služby [126],
- **Sider:** zdarma [127], podporuje JS, TS i Python [128], GitHub integrace jen pro PR [129], založeno na integraci mnoha nástrojů do jedné služby a vytváření vlastních pravidel na míru projektu [128].

Nástroje druhého typu splňující minimální požadavky:

- **DeepSource:** zdarma [130], podporuje jen Python [130], GitHub integrace pro PR i commity [131], založeno na pokročilé hloubkové analýze kódu [108],

- **DeepScan:** zdarma [132], podporuje jen JS a TS (podporuje i React) [133], GitHub integrace pro PR i commity [133], založeno na pokročilé hloubkové analýze kódu [133], možnost integrace ESLint [134],
- **LGTM:** zdarma [135], podporuje JS, TS i Python [136], GitHub integrace pro PR i commity (PR analyzuje vždy okamžitě, commity analyzuje jednou denně) [136], založeno na pokročilé hloubkové analýze kódu, možnost vytváření pokročilých pravidel na míru projektu [135],
- **SonarCloud:** zdarma [137], podporuje JS, TS (podporuje i React [138]) i Python [137], GitHub integrace pro PR i commity [139], založeno na pokročilé hloubkové analýze kódu [140], možnost připojit i vygenerované reporty z externích nástrojů (linterů) [141],
- **DeepCode:** zdarma [142], nezávislé na jazyku [108], GitHub integrace pro PR i commity [142], založeno na pokročilé hloubkové analýze kódu (používá rozsáhlé strojové učení) [143].

4.4.2 Lintery

V předchozí sekci jsem uváděl, že některé nástroje (z vyšší vrstvy) z valné většiny staví na integraci mnoha drobnějších nástrojů (z nižší vrstvy), tzv. linterů. Smyslem této sekce je vysvětlit vlastnosti linterů a jejich fungování a poté prozkoumat možnosti použití pro aplikaci ÚP.

Lintery jsou nástroje provádějící statickou analýzu kódu [144]. V případě dynamicky typovaných jazyků můžeme říci, že z hlediska samotné statické analýzy nahrazují kompilátor staticky typovaných jazyků [144]. Pomáhají vyvarovat se mnoha (samozřejmě ne všech) chyb, které se mohou projevit až při běhu aplikace [144].

Lintery pracují s uživatelem definovanou množinou pravidel a dohlíží na jejich splnění, pravidla jsou jak obecnějšího rázu, tak vytvořená na míru některým frameworkům/knihovnám (např. Reactu) [144]. Některá z pravidel umožňují lintery automaticky opravit bez manuálního zásahu programátora, některá pravidla opravit nelze (např. komplexní problémy jako použití zastaralého API ad.) a programátor musí kód opravit manuálně [144]. Pravidla lze rozdělit do dvou skupin – logická (chyby v kódu, potenciální problémy, špatné vzory v kódu) a stylistická (kód podle konvencí) – lintery pak podporují pravidla z jedné či obou skupin [145].

Nástroje pro statickou analýzu kódu obvykle stojí na konceptu AST (Abstract Syntax Tree) – ten reprezentuje zdrojový kód jako stromovou strukturu, kořen stromu je tvořen samotným souborem, jeho potomci jsou konstrukty z nejvyšší úrovně tohoto souboru atd. V případě linterů je pak AST použit pro kontrolu splnění všech definovaných pravidel napříč uzly stromu (případně pak i pro automatickou opravu přímo v daném místě) [144, 146].

Než uvedu nejběžnější lintery používané pro jazyky v aplikaci ÚP, je třeba říci, že některé lintery prakticky jen zaobalují několik linterů dohromady (např. Flake8) [145].

Pro **JS** je nejpopulárnější linter ESLint, je vysoce konfigurovatelný a nabízí integraci s mnoha různými nástroji, mnoho z pravidel lze automaticky opravit [144]. Mimo jiné nabízí možnost nakonfigurovat používání různých standardů (např. Airbnb) [147] či dalších doplňků pro různé knihovny (např. pro React) [148]. Před vznikem ESLint byl populární také JSHint, který vzešel z JSLint [147].

Pro **TS** byl používán TSLint, ale v roce 2019 bylo oznámeno utlumení jeho vývoje ve prospěch rozvoje podpory TS pod jednou střechou v ESLint [149].

Pro **CSS** je velmi populární stylelint [150, 151], alternativou je CSSLint, který se ale už dle [151] moc nepoužívá a není udržovaný.

Pro **Python** je na výběr více linterů lišících se zaměřením definovaných pravidel – na logická pravidla se zaměřuje Pyflakes a Bandit, na stylistická pravidla se zaměřuje pycodestyle a na obojí se zaměřuje Pylint (ten je používán nejvíce [152]) [145]. Také se používají lintery, které zaobalují několik linterů dohromady – Flake8 (mj. Pyflakes, pycodestyle) a Pylama (mj. pycodestyle, Pyflakes, Pylint) [145].

4.4.3 Formattery

Formattery jsou nástroje, které zajistí konzistentní formátování kódů – to umožní jednodušší čitelnost, srozumitelnost a vyšší efektivitu při práci v týmu [144]. Umožní zaměřit se na to nejdůležitější – co kód dělá, už není třeba řešit konkrétní způsob formátování kódu jednotlivými členy týmu [147].

Vzhledem k tomu, že formattery také pracují nad samotným kódem a provádějí jeho analýzu, lze je též řadit do této kapitoly [144]. Stejně jako lintery pracují nad AST – soubor převedou do AST, ignorují nedůležité informace související s původním formátováním a výsledný upravený AST zapíšou zpět do souboru, konzistentně [144]. Díky tomuto přegenerování AST jsou v oblasti formátování kódu mnohem mocnější než samotné lintery, které, jak již bylo zmíněno v podsekcí 4.4.2, aplikují opravu pouze v daném místě AST, AST nepřegenerovávají celý – proto se vyplatí používat lintery i formattery zároveň a nechat je pracovat v oblastech, kde excelují, tedy formatter pro komplexní formátování kódu a linter pro pravidla týkající se např. syntaxe, problémů a nových vlastností jazyka [146]. Tato kombinace umožní psát v týmu konzistentní kód, tedy kód, kde nelze prakticky poznat, kdo jej psal, nevyskytují se v něm špatné vzory a má méně chyb [147].

Pro **JS**, **TS** a **CSS** je nejpopulárnější formatter Prettier [144].

Pro **Python** se používají 3 formattery: Black, YAPF, autopep8 [153, 145]. Liší se zejména konfigurovatelností a mírou splnění Python standardu PEP 8, Black se dá přirovnat svou ideologií k Prettier – oba dva jsou totiž založeny na

4. NÁSTROJE PRO USNADNĚNÍ VÝVOJE A ÚDRŽBY

co nejmenší míře konfigurovatelnosti (tím pádem se snižuje diskuze ohledně zavedení jednotlivých konfigurací, protože není moc na výběr) [145].

Zvolené technologie

V této kapitole uvedu, které nástroje a technologie zvolím pro použití a mj. je budu zavádět v následující praktické části. Volba bude obsahovat argumenty vycházející z provedených rešersí v rámci celé teoretické části.

5.1 Testování

V souladu se zadáním práce bude implementováno automatizované testování API a UI (v případě UI konkrétně E2E testování). Vzhledem k tomu, že v současné verzi aplikace chybělo testování úplně (resp. bylo zde několik jen velmi základních testů prakticky na smoke úrovni, viz sekce 2.3), byl zvolen přístup orientace API a UI testů na klíčové funkce a průchody v aplikaci, aby bylo zajištěno, že obvykle používané části aplikace budou vždy za jakékoliv situace funkční. Vzhledem ke křehkosti UI testů bude při jejich implementaci kladen důraz na co nejvyšší nezávislost na detailech UI, aby při změnách nebylo třeba do testů zasahovat vůbec, nebo v co nejmenší míře. Výhodou zavedení testů API, které jsou v „testovací pyramidě“ (viz sekce 3.1) na nižší úrovni než UI testy je právě větší stabilita API oproti UI a mnohem větší rychlost samotných testů. Navíc, pokud neprojdou API testy, není se třeba ani zatěžovat s UI testováním (klientská část totiž stojí na používání REST API). Ze strategie „testovací trofeje“ (viz sekce 3.2) bude vycházet použití nástrojů pro statické typování a statickou analýzu kódu (tedy nejnižší úroveň této strategie), jejichž volbu uvedu v následujících sekcích – to umožní i bez tvorby dalších testů na nižších úrovních testovací pyramidy kontrolovat v přiměřené míře problémy a kvalitu kódu.

Manuální testování stále bude mít stále své místo v akceptační části, testech použitelnosti a při samotném vývoji.

Co se týče implementace samotného testování, bude kompletně provedeno v jazyce Python. Zde samozřejmě mohla padnout volba prakticky na jakýkoliv jazyk, smysl ale dává testování psát v jednom z již používaných jazyků

v rámci projektu ÚP, tedy Python či JS. Python byl zvolen především z důvodu možnosti jednoduché integrace s Djangoem a také proto, že celá část implementace testů byla vytvářena jako semestrální práce v rámci předmětu *MI-PYT*. Pro testování bude využit nástroj behave založený na BDD – tedy budou sepsány v prakticky přirozeném jazyce (Gherkin) scénáře, ke kterým budou implementovány testy. Kromě jasně definovaných scénářů tyto soubory díky zápisu v přirozeném jazyce poslouží jako dokumentace, což se hodí jak pro účely samotného projektu ÚP, tak i pro zmíněnou semestrální práci, kde tímto může být jednoduše splněn požadavek dokumentace bez jakékoliv práce navíc a vyučující pak má přehled o splnění úkolu. Pro UI testování bude využito Selenium, a to především díky faktu, že se jedná prakticky o standard v oblasti UI testování, je nejpoužívanější a rozšíří to i mé obzory v této mně dosud neznámé oblasti.

5.2 Nástroje pro usnadnění vývoje a údržby

V této sekci uvedu a oargumentuji zvolené nástroje nejprve pro monitorování chyb, pak správu logů a na závěr pro statické typování a statickou analýzu kódu. V některých případech bude volba ponechána z udaných důvodů až na praktickou část.

5.2.1 Monitorování chyb

Jako nástroj, který bude mít na starost monitorování chyb na serverové i klientské části aplikace jsem zvolil Sentry. Dle zjištěných informací (v sekci 4.1) se jedná o nejpoužívanější nástroj v této oblasti a splňuje všechny vytyčené požadavky. Dále bude prozkoumána možnost použití LogRocket.

5.2.2 Správa logů

Pro správu logů, tedy nástroj, který bude z Heroku přebírat všechny logy, ukládat je a umožní v nich vyhledávat v historii a upozorňovat na zvolené události, byl zvolen nástroj Logentries. Dle zjištěných informací (v sekci 4.2) se sice jedná až o druhý nejpoužívanější nástroj v této oblasti splňující vytyčené požadavky, v porovnání s nejpoužívanějším nástrojem Papertrail ale nabízí více prostoru pro logy a umožní vyhledávat v záznamech starších než 2 dny.

5.2.3 Statické typování

Vzhledem k výhodám typových anotací (viz sekce 4.3) budou zavedeny na serverové části (Python) i na klientské části (JS).

Na serverové části bude využita standardizovaná syntaxe zápisu anotací a pro kontrolu typů bude použit referenční nástroj mypy, protože je oficiální, a také vestavěná typová kontrola IDE PyCharm. Volba nástrojů pro typovou

kontrolu není nevratný krok a v případě zjištěných nedostatků je lze jednoduše nahradit za jiné nalezené alternativy.

Na klientské části napsané v JS, který nenabízí standardizovanou syntaxi pro anotaci typů (viz podsekcce 4.3.2), se nabízí volba mezi Flow a TypeScript. Použití Flow dává smysl z hlediska architektury klientské části, která je postavena na Reactu – oba nástroje pochází z dílny Facebooku – Flow nabízí nativní podporu Reactu a je jednoduché jej začlenit do stávajícího projektu (nevyžaduje např. kompilátor apod.). Použití TypeScriptu naopak dává smysl z hlediska lepší podpory typových definicí knihoven třetích stran, lepší integraci s IDE a časté migraci i větších projektů právě z Flow na TS. I přes poměrně rozsáhlou rešerši ale volbu mezi těmito dvěma nástroji neprovedu rovnou zde v teoretické části a přenechám ji do části praktické, kde provedu zkušební zavedení Flow i TS do projektu a až na základě vlastních zkušeností (a ověření pravdivosti výroků zjištěných v rešerši) a zvážení provedu finální volbu a kompletní migraci.

5.2.4 Statická analýza kódu

V rámci rešerše jsem zjistil, že se nástroje pro statickou analýzu dají rozdělit na několik částí. Lze na ně nahlížet podle vrstev – na nástroje z vyšší vrstvy pro průběžnou kontrolu kvality kódu a z nižší vrstvy, což jsou prakticky lintery.

Nástroje z vyšší vrstvy jsem dále dělil do dvou typů podle dalších nabízených možností – vzhledem k velké podobnosti nástrojů na této vyšší vrstvě budou v praktické části otestovány všechny nalezené nástroje a na základě jejich funkčnosti, nalezených problémů, nabízených funkcí a přívětivosti pak budou zvoleny ty vhodné pro tento projekt.

Linter (nástroj z nižší vrstvy) pro klientskou část bude ESLint, který je v této oblasti nejpopulárnější. Pro CSS se použije též nejpoužívanější stylelint. Vzhledem k tomu, že kód serverové části oproti té klientské není tolik rozsáhlý, nebude prozatím zaváděn linter pro Python, využije se služeb pouze formatteru a nástrojů pro statickou analýzu kódu z vyšší vrstvy.

Formatter pro JS (případně TS) a CSS bude Prettier, který je v této oblasti nejpopulárnější. Pro Python bude použit jeho ekvivalent Black (ekvivalent z hlediska faktu, že oba nabízejí minimální možnost konfigurovatelnosti), protože pokročilá konfigurovatelnost nebude potřeba.

Část II

Praktická část

Sběr požadavků a analýza

V této kapitole přehledně uvedu všechny zjištěné požadavky a podrobně je popíši. Nejprve se zaměřím na funkční požadavky, pak nefunkční a na závěr uvedu i vyřazené požadavky. V případě, že bude požadavek potřeba analyzovat velmi podrobně, budu se mu věnovat v rámci další sekce, toto bude u každého takového požadavku indikováno v jeho stručném popisu.

6.1 Požadavky

Následuje seznam funkčních, nefunkčních a vyřazených požadavků na rozšíření aplikace. Funkční a nefunkční požadavky mají pro následnou práci přiřazeny unikátní identifikátory ve tvaru „F<číslo>“/„N<číslo>“ (kde písmeno značí funkční/nefunkční požadavek). Hvězdičkou (*) jsou dále označeny požadavky, které vychází z plánovaných rozšíření v rámci bakalářské práce uvedených v sekci 2.4. Některé plánované změny z této kapitoly byly vyřazeny a je jim věnována poslední podsekce 6.1.3.

6.1.1 Funkční požadavky

F1 evidování zájemců o kurz *: vzhledem k plnému obsazení všech termínů lekcí během týdne je třeba evidovat zájemce o kurzy – ať už jednotlivce, či zájemce o skupiny – každý klient může mít zájem o daný kurz nejvýše jednou, je třeba také evidovat k tomuto zájmu text formou poznámky, datum přidání zájemce do evidence a kurz, o který má zájem,

F2 kontrola časového konfliktu lekcí *: současná verze aplikace nijak neřeší časové konflikty lekcí, je třeba zakázat možnost jakéhokoliv překryvu lekcí vzhledem k datu, času a jejich délce, toto se netýká zrušených lekcí, které budou pro řešení konfliktů ignorovány (je třeba zavést **F19** pro korektní fungování konfliktů),

- F3 vylepšení předplacených lekcí *:** stávající způsob evidence předplacených lekcí není dostačující – pro jednotlivce je třeba předplacené lekce přidávat po jednom (klienti si ale často předplácí více lekcí dopředu), pro skupiny je evidování ještě horší, protože každý člen obvykle platí jinak a na odlišné časové období (případně vždy jen jednu lekci) a prakticky se nedá tato evidence předplacených lekcí ručně udržovat, je to příliš složité, detailní analýza tohoto požadavku viz podsekcce 6.2.1,
- F4 vyhledávání klientů *:** v aplikaci je mnoho klientů a je časově náročné vždy v seznamu vyhledávat příslušného klienta, je třeba zavést možnost vyhledávání v klientech, a to z jakéhokoli místa aplikace, také je třeba, aby vyhledávání bralo v potaz možné překlapy lektorky v zadávaném výrazu k vyhledávání a také možný překlep ve jménu uloženého klienta, vyhledávalo by se jen mezi aktivními klienty (zavedení aktivních klientů viz požadavek **F6**),
- F5 přepracování formuláře pro lekce:** současný formulář není příliš přehledný, pokud má skupina více než 2 členy, je potřeba neustále posouvat obsahem, protože se nevejde na monitor – formulář musí být kompletně přepracován dle konzultace s lektorkou, v závislosti na návrhu může souviset s vylepšením předplacených lekcí v rámci požadavku **F3**,
- F6 zavedení aktivních a neaktivních klientů a skupin:** v evidenci je mnoho klientů a skupin, kteří aktuálně nechodí, ale např. za rok budou opět chodit, je tedy třeba umožnit skrytí všech klientů/skupin, kteří aktuálně na lekce nedochází a k těmto skrytým (neaktivním) klientům/skupinám umožnit přístup, ve výchozím zobrazení ale ukazovat jen aktivní,
- F7 nastavitelná délka kurzů:** současná verze aplikace automaticky předvyplní dobu trvání lekce v závislosti na tom, zda je skupinová (45 min.) či pro jednotlivce (30 min.) – to není dostačující, protože např. lekce některých kurzů trvají vždy 45 min. nehlédě na počet členů – je tedy třeba umožnit u kurzu evidovat dobu trvání pro jednotlivce, a tu pak ve výchozím stavu jednotlivcům dávat, pro skupiny stále stačí jedna výchozí hodnota (45 min.),
- F8 propojení s bankou:** na hlavní stránce je třeba mimo dnešních lekcí zobrazit aktuální zůstatek na bankovním účtu projektu (Fio banka) a transakce za poslední 3 týdny,
- F9 změny účastníků skupinových lekcí:** někdy se stává, že klient v průběhu kurzu opustí skupinu, v evidenci je již naplánováno několik lekcí dopředu a všech se účastní – je třeba umožnit při úpravě lekce projevít tyto změny členů skupiny do účastníků dané lekce (v současné době nečlen skupiny zůstává účastníkem lekce a lektorka musí každou lekci

ručně smazat a vytvořit znovu bez nečlenů, což je velmi nepohodlné), stejně tak je třeba umožnit do účastníků naopak projevít nové členy skupiny, kteří v dané lekci jako účastníci evidování nejsou,

- F10 automatické přidání (a odebrání) předplacené lekce:** při omluvě klienta/zrušení ze strany lektorky je třeba automaticky u klienta/skupiny zaznamenat, že mají jeden předplacený termín navíc (v případě, že daný klient měl zaplacenou), v případě jednotlivců je třeba také automaticky zaevidovat, za který den je tato náhradní lekce vytvořena, v případě skupin je třeba při přidávání lekce automaticky označit lekci jako place-nou pro účastníky, kteří mají předplacené lekce a při přidání pak odečíst jednu předplacenou lekci, souvisí se zavedením lepší evidence předplacených lekcí v požadavku **F3**,
- F11 efektivnější práce v rámci aplikace:** v současné verzi aplikace musela lektorka pro často prováděné činnosti provádět zbytečně mnoho kroků navíc, což činilo příslušné činnosti náročnějšími a snadno se udělala chyba a ztrácel čas – na základě dalších analýz byly zjištěny problémové oblasti, které vyžadují přepracování: umožnit úpravu lekce (pro jednotlivce i skupiny) z diáře a přehledu, umožnit úpravu klienta/skupiny přímo v kartě klienta/skupiny, umožnit přidání lekce (pro jednotlivce i skupiny) z diáře a přehledu (a to jak s příslušným datem, u kterého bude toto tlačítko, tak i obecně s jakýmkoliv jiným datem), umožnit přidat klienta přímo při přidávání skupiny/zájemce (funkcionalita zájemce implementována v rámci **F1**),
- F12 evidování barev kurzů:** je třeba umožnit přiřazení barvy každému kurzu a tuto barvu použít pro rozlišení kurzů napříč celou aplikací, kdekoliv se vyskytuje název kurzu – lektorka potřebuje okamžitě rozlišit kurzy a mít možnost vidět na první pohled např. v diáři díky barvě, kterého kurzu se lekce týká (kurzy mají v rámci propagačních materiálů své ustálené barvy),
- F13 automatické předvyplnění údajů lekce:** pokud má klient/skupina nějakou historii v evidenci, při přidávání nové lekce je třeba na základě této historie předvyplnit vhodnými hodnotami datum, čas a kurz nové lekce – je tedy třeba vhodně zvolit lekci klienta z jeho historie, podle které budou tyto údaje vypočteny – smyslem je těmito výchozími hodnotami vystihnout co nejvíce případů tak, aby lektorka musela tyto hodnoty co nejméně často upravovat, detailní analýza tohoto požadavku viz podsekcce 6.2.2,
- F14 upozornění na ztrátu dat formulářů:** při vyplňování formulářů lektorka občas omylem formulář/stránku zavře a přijde o úpravy, je třeba zavést ochranu, která tomuto zabrání (a zároveň ale nebude zbytečně upozorňovat na ztrátu dat, když k žádné změně nedošlo),

- F15 vylepšení chybových hlášení:** chybová hlášení jsou někdy málo podrobná, případně nezmiňují možnosti řešení, zobrazují se krátce, případně dojde k neošetřené chybě na serveru a klientská část pak nedokáže korektně uživateli popsat, kde je problém,
- F16 titulky stránek:** každá stránka by měla mít v prohlížeči svůj titulek, v současné době má každá stránka stejný titulek a lektorka tak nemá možnost jednoduše rozlišit, který panel má otevřenou kterou stránku aplikace,
- F17 nastavitelné vlastnosti stavů účasti:** některé stavy účasti mají pro některé výpočty v rámci aplikace speciální význam (např. omluvené lekce se nezapočítávají do počtu absolvovaných lekcí klienta, další stav znamená, že klient dorazí a tento stav je zároveň výchozí), toto svázání stavu účasti s významem je ale založeno na názvu stavu účasti (a pevně nadefinováno v kódu), lektorka si chce ale název příslušných stavů účasti sama měnit a je třeba pro to zavést příslušné možnosti,
- F18 omezení a validace hodnot:** je třeba provést revizi stávajících omezení na jednotlivé hodnoty v rámci aplikace (API a databáze) i klientské části, zavést nová omezení pro nové funkční požadavky a zdokumentovat všechna aktuální omezení a validace prováděná nad celou doménou, detailní analýza tohoto požadavku viz podsekce 6.2.3,
- F19 automatické zrušení lekce:** pokud nikdo z účastníků nemá dorazit (všichni jsou omluveni), lekce má být automaticky zrušena,
- F20 zobrazení zrušených lekcí:** v přehledu na hlavní stránce a v diáři je třeba zobrazit i zrušené lekce, v současné verzi se nezobrazují (to byl původní požadavek, ale nakonec se ukázal jako nesprávný a lektorka zrušené lekce potřebuje vidět nejen v kartě klienta),
- F21 skupinové lekce bez účastníků:** lektorka potřebuje vytvářet lekce pro skupiny bez účastníků (plánuje dopředu termíny a členy přidá až když budou známi) – současná aplikace na toto nebyla připravena, klientská část aplikace dokonce při pokusu o přidání takové lekce spadne (resp. spadne v případě, když skupina nemá žádné členy – nelze zobrazit ani karta skupiny).

6.1.2 Nefunkční požadavky

- N1 dokumentace:** doplnění dokumentace v kódu pro serverovou i klientskou část, dokumentace API,
- N2 testování *:** aplikace prakticky neobsahuje žádné testy (jen několik základních „smoke“ testů), je třeba zavést API a UI (E2E) testy pro klíčové části aplikace a průchody v aplikaci,

- N3 zavedení nástrojů pro usnadnění vývoje a údržby:** je třeba zavést nástroje pro monitorování chyb v aplikaci, správu logů (vyhledávání, ukládání), statické typování a analýzu kódu – k tomu byla provedena v teoretické části rešerše v kapitole 4,
- N4 revize bezpečnosti:** je třeba provést kompletní revizi aplikace z hlediska bezpečnosti a opravit případné slabiny, problémy či zranitelnosti, provést aktualizace závislostí apod., detailní analýza tohoto požadavku viz podsektce 6.2.4,
- N5 vylepšení použitelnosti:** z hlášení lektorky a také z vlastní analýzy vyplývá, že je třeba se zaměřit na opravení problémů s použitelností a její vylepšení, detailní analýza tohoto požadavku viz podsektce 6.2.5,
- N6 optimalizace API *:** pokud klientská část má někde zobrazit více dat, požadavek někdy probíhá enormně dlouho a dokonce může být ze strany Heroku pro dlouhou prodlevu zastaven a lektorka si data nezobrazí – je třeba nalézt úzké hrdlo aplikace, které toto zpomalení v jednotkách případů (když je hodně dat) zpomaluje a toto vylepšit, také je třeba se zaměřit na optimalizaci počtu požadavků na API (zda některé nelze uložit a znovupoužít bez dalšího provádění, viz možné rozšíření o React Context API v sekci 2.4) a případně zjednodušení obsahu odpovědi díky znovupoužívání,
- N7 konfigurace více prostředí:** pokud se aplikace úspěšně sestaví na integračním serveru, nahraje se na produkci [1] – to může v případě neodhalené chyby v aplikaci způsobit okamžitý pád produkce a případně i ztrátu dat, toto souvisí s lepším pokrytím testy (viz požadavek **N2**), ale je třeba se také zaměřit na vhodný návrh více prostředí pro nasazování – tedy nejen na způsob nasazování na produkci, ale také na zavedení dalších prostředí a rozhodnutí, ve kterých fázích se do nich bude nasazovat – a to tak, aby bylo k dispozici jak prostředí naprosto totožné s produkcí (např. pro reprodukování chyb hlášených koncovým uživatelem), tak prostředí opět postavené na tom produkčním, ale s novější nasazenou verzí aplikace – kompletní návrh bude popsán v sekci 7.4, součástí implementace v podsektci 8.2.3 (v rámci požadavku **N4** na revizi bezpečnosti) bude také zavedení jednotného způsobu práce s proměnnými prostředí napříč všemi prostředími (toho bude využito i v požadavku **B5** na sjednocení práce s tokeny),
- N8 zálohy databáze:** je třeba zavést pravidelné automatické zálohování databáze z produkce.

6.1.3 Vyřazené požadavky

V této podsekcí uvedu několik vyřazených požadavků, které nebudou v rámci této práce řešeny a ke každému zmíním i příslušné důvody. Jedná se o oblasti možného rozšíření z bakalářské práce [1], které nebyly vybrány k implementaci, ostatní požadavky byly zakomponovány do předcházejících dvou podsekcí k funkčním a nefunkčním požadavkům.

Seznam vyřazených požadavků:

- **evidence pomůcek a učebnic *:** v rámci projektu ÚP již existuje starší aplikace na míru, která tuto funkcionalitu dostatečně řeší a zatím není potřeba toto řešení integrovat do jednotného řešení,
- **offline přístup a SSR *:** co se týče SSR, načítání aplikace je dostatečně rychlé a není zde tedy potřeba prozatím SSR řešit, řešení offline režimu zatím není ze strany lektorky požadováno (stačí jí stávající řešení).

6.2 Detailní analýza některých požadavků

Některé požadavky nejsou úplně přesně definované a před dalším pokračováním je třeba je dodefinovat.

6.2.1 F3 - vylepšení předplacených lekcí

Detailní analýza požadavku **F3**. V případě jednotlivců je problém s tím, že je předplacené lekce přidávat po jednom – zde bude lektorce vyhovovat ve formuláři pro přidání lekce možnost uvést počet přidávaných předplacených lekcí, tedy bude zakomponováno v rámci **F5**.

Předplacené lekce skupin jsou evidovány v současné verzi stejným způsobem, tedy lekce je označena jako předplacená, jen je možné zvolit, kteří ze členů skupiny ji skutečně mají předplacenou – už zde začíná první problém, kdy lekce je předplacená, ale někteří klienti si ji nepředplatili, což je sice korektně zaznamenáno, ale situace začíná být nepřehledná. Při dalších předplacených lekcích, zejména když si jeden účastník zaplatí celou lekci dopředu, nemají ostatní účastníci evidovanou platbu a platí v jiném časovém horizontu – tato situace je už velmi nepřehledná a může vyústit v chyby lektorky kvůli nepřehlednosti. Zde by lektorce vyhovovaly počítačla předplacených lekcí každého klienta ve skupině, což vyřeší všechny zmíněné problémy. Z počítačel by se při přidávání nové lekce automaticky odečítalo (viz příslušný požadavek **F10**).

Řešení pomocí počítačel vypadá jako ideální řešení i pro jednotlivce, zde by ale byl problém s faktem, že mohou chodit na více kurzů, tedy nelze mít centrální počítačlo pro všechny (na některé kurzy už nechodí, předplácí třeba jen některé, na které chodí apod.), bylo by třeba mít pro každý kurz počítačlo zvlášť, ale v tom případě by pak už nebylo možné evidovat v rámci **F10**,

za který den daná předplacená lekce tvoří náhradu (resp. možné by to bylo, ale značně by to zkomplikovalo jak implementaci, tak i práci lektorky), proto je řešení předplacených lekcí pro jednotlivce a skupiny odlišné.

6.2.2 F13 – automatické předvyplnění údajů lekce

Detailní analýza požadavku **F13**. Smyslem tohoto požadavku je předvyplnit datum, čas a kurz nově přidávané lekce podle minulých lekcí, a to tak, aby muselo být do těchto hodnot co nejméně ze strany lektorky zasahováno. Lekce se nejčastěji konají jednou za týden a obvykle ve stejný čas a patří samozřejmě k téže kurzu. Stačí tedy vhodně zvolit referenční lekci a z ní tyto údaje odvodit. V závislosti na podobě historie klienta/skupiny je třeba pokrýt možné případy výběru lekce.

Případy výběru lekce:

1. V případě, že klient na žádné lekci ještě nebyl, neodvodíme nic a údaje zůstanou nepředvyplněné.

V případě skupiny bez lekcí datum a čas také v tomto případě neodvodíme, kurz je ale jasný z atributů skupiny.

2. V případě, že klient chodí na jeden jediný kurz, vyber tento kurz a datum a čas zvol o týden později oproti poslední lekci tohoto kurzu (poslední lekce může být jen předplacená, tedy z té odvod pouze kurz).

Pro skupiny platí totéž, jen datum a čas bude odvozen vždy, protože předplacené lekce budou v rámci F3 evidovány jinak než u jednotlivců.

3. V případě, že klient chodí/chodil na více kurzů, vyber ten kurz, jehož poslední lekce je nejpozději (a z té odvod všechny údaje), navíc, pokud některý kurz má předplacené lekce, preferuj ten (a tedy odvod jen kurz).

Skupina v této situaci být nemůže, protože má vždy jen lekce k jednomu kurzu.

6.2.3 F18 – omezení a validace hodnot

Detailní analýza požadavku **F18**. V rámci tohoto požadavku bylo třeba vytvořit kompletní seznam všech omezení a validací, které v aplikaci mají být, a to i s ohledem na všechny nové funkční požadavky. Všechna tato omezení byla v rámci analýzy definována. Jejich seznam a rozdělení je vidět v obrázku 7.1 s návrhem aktualizovaného logického datového modelu.

6.2.4 N4 – revize bezpečnosti

Detailní analýza požadavku **N4**. V rámci tohoto požadavku je potřeba zjistit problémové oblasti z hlediska bezpečnosti a případně i konkrétní problémy,

na které se zaměřit. Bylo tedy třeba projít např. konfiguraci aplikace, testovat chování aplikace a také použít nástroj Mozilla Observatory, který skenuje webové stránky a kontroluje jejich zabezpečení [154]. Bylo zjištěno několik problémů níže, mají opět unikátní identifikátor, kde písmeno „B“ značí problém s bezpečností.

Nalezené problémy:

- B1 aktualizace závislostí:** je třeba aktualizovat všechny závislosti (používané knihovny a nástroje) kvůli opravám chyb, zranitelností, ale i novým funkcím – např. knihovny klientské části při použití příkazu `yarn audit` (pro který musela být i upravena verze yarn, protože v požadované verzi v rámci projektu tento příkaz ani není) bylo nalezeno 36 zranitelností, také je třeba zohlednit u knihoven sémantické verzování a povolit zpětně kompatibilní aktualizace,
- B2 sjednocení konfigurací:** v rámci produkční konfigurace je možné některá nastavení sloučit s konfigurací lokální verze – kromě zvýšení bezpečnosti dojde především také ke zvýšení konzistence mezi těmito verzemi aplikace a tedy v budoucnu méně problémy kvůli odlišnostem,
- B3 deaktivace DEBUG módu:** produkční konfigurace pro Django obsahuje nastavenou proměnnou `DEBUG = True`, na produkci toto nastavení ale být nesmí – jedná se jak o bezpečnostní problém (náhled útočníka do metadat aplikace, některých proměnných prostředí ad.), tak výkonostní problém (zbytečné režijní náklady a vysoké nároky na paměť kvůli ukládání všech SQL dotazů) – na produkci je tedy třeba nastavit `DEBUG = False` [155],
- B4 HTTP hlavičky:** díky nástroji Mozilla Observatory bylo zjištěno, že v aplikaci chybí nastavení některých HTTP (Hypertext Transfer Protocol) hlaviček – Referrer Policy, CSP (Content Security Policy), HSTS (HTTP Strict Transport Security) – podrobnému popisu významu se budu věnovat v podsekcí 8.2.3,
- B5 sjednocení práce s tokeny:** je třeba sjednotit práci s citlivými tokeny a dalšími podobnými položkami – těchto citlivých dat bude vzhledem k požadavkům přibývat (např. přístup do banky) a je třeba zavést centrální místo správy, kterým budou proměnné prostředí (nyní jsou některé tokeny zašifrované např. v rámci souboru s konfigurací Travisu, některé kvůli své povaze zašifrované nejsou apod.), nebude tak hrozit žádný únik např. ve verzovacím systému (který již mohl nastat i v případě tohoto projektu, kdy by kvůli bezpečnostnímu problému s FontAwesome PRO [156] došlo k úniku tokenu pro přístup k placeným ikonám prostřednictvím souboru `yarn.lock`, což by znamenalo problém právě při zveřejnění repozitáře, což je jeden z úkolů této práce), zavedení proměnných prostředí umožní také jednoduchou práci s více prostředími

(viz požadavek **N7**) a, jak již bylo zmíněno, umožní samotné zveřejnění repozitáře jako open-source – jak uvádí jedno z pravidel metodiky „The Twelve-Factor App“, ke konfiguraci aplikace je třeba přistupovat tak, aby v jakémkoliv momentu bylo teoreticky možné repozitář zveřejnit a nedošlo by k žádné kompromitaci přístupových údajů [157], to zde bude splněno,

B6 zákaz indexace roboty: roboti mohou přistupovat na stránku s aplikací a stránku indexovat (např. indexovací robot Google) – indexace není potřeba a jediným výsledkem jsou pak zbytečné přístupy z vyhledávačů bez přihlášení, tedy zbytečná spotřeba výpočetního výkonu.

6.2.5 N5 – vylepšení použitelnosti

Detailní analýza požadavku **N5**. Pro zjištění problémů v oblasti použitelnosti a přístupnosti bylo zvoleno několik metod – Nielsenova heuristická analýza, WCAG 2.1 (Web Content Accessibility Guidelines) a uživatelské testování použitelnosti formou pozorování lektorky při každodenní práci v aplikaci.

Nielsenova heuristická analýza obsahuje 10 základních pravidel použitelnosti [158]: viditelnost stavu systému, spojení mezi systémem a reálným světem, uživatelská kontrola a svoboda, konzistence a standardizace, prevence chyb, rozpoznání místo vzpomínání, flexibilní a efektivní použití, estetický a minimalistický design, pomoc uživatelům poznat, pochopit a vzpamatovat se z chyb, nápověda a návody. Tyto body jsou v rámci této heuristiky dále detailněji popsány [158], zde již dále rozepsány ale nebudou, zaměřím se jen na zjištěné problémy.

Doporučení WCAG 2.1 vytvořené v rámci konsorcia W3C (World Wide Web Consortium) je v současnosti nejrozšířenější a celosvětově uznávaná metodika tvorby přístupného webového obsahu [159]. Následování těchto doporučení učiní obsah přístupnější pro větší okruh uživatelů s různým zdravotním postižením [160]. Díky aplikaci doporučení je často webový obsah také více použitelný obecně pro všechny uživatele [160] – a toto je důvod, proč se na tato doporučení zaměřuji v rámci vylepšení použitelnosti aktuální aplikace. Smyslem tedy bude na základě doporučení ověřit a případně napravit problémy s použitelností, které může pocítit i sama lektorka.

Některé zjištěné problémy jsou již v požadavcích definovány, jiné jsou úplně nové. Podle toho problémy rozdělím a navíc ty nové opět opatřím unikátním identifikátorem, kde písmeno „P“ značí problém s **p**oužitelností.

Seznam nových zjištěných problémů:

P1 délka načítání: při delším načítání není uživatel nijak informován, že je vše v pořádku a aplikace stále pracuje – je třeba při delším načítání uživatele upozornit, že je vše v pořádku a v případě, že načítání trvá přespříliš dlouho, nabídnout mu nějaké řešení,

- P2 popis netextových prvků:** některé netextové prvky úplně postrádají popis (např. po najetí myši), případně jej obsahují, ale formou `title`, tedy nelze zobrazit na mobilních zařízeních – je třeba popisy zavést všude a umožnit zobrazení i na mobilních zařízeních,
- P3 favicon:** v záložce prohlížeče není ikona (favicon) – dodat,
- P4 posouvání modálního okna:** v modálním okně se na iOS nedá plynule posouvat – opravit,
- P5 react-select:** některé prvky pro výběr (`select`) jsou řešeny uživatelsky přívětivým `react-select` (výběr členů skupiny), některé (výběr kurzu) ale ne a nedá se v nich tak např. vyhledávat či jednotlivé položky odlišit barvou – všude použít `react-select` (kromě stavu účasti, kde je pohodlnější jednodušší `select`),
- P6 nefunkční label:** na některé popisy formulářových polí (`label`) nelze kliknout pro psaní do pole – opravit,
- P7 povinná pole:** nejsou nijak indikována povinná pole ve formulářích – doplnit indikaci povinných polí,
- P8 kontrola pravopisu:** v polích pro poznámky nefunguje kontrola pravopisu – opravit,
- P9 už. jméno na iOS:** uživatelské jméno na iOS začíná velkým písmenem – opravit a používat malé písmeno,
- P10 autofocus ve formulářích:** některé formuláře automaticky nevyberou první pole pro psaní (`autofocus`) nebo neumožní pohyb pomocí klávesy TAB mezi prvky formuláře – opravit,
- P11 indikace načítání:** zobrazení načítání v mnoha případech nekorresponduje s tím, zda je už skutečně vše načteno a obsah komponent se zobrazí až později (ačkoliv načítání už není zobrazeno), tento fakt by také komplikoval zavedení automatizovaných testů UI (testovací nástroj nepozná, stejně jako uživatel, zda už je načteno), uživatel také není informován po odeslání formuláře, po kliknutí na uložení celá aplikace nic nedělá a až po dokončení požadavku se najednou formulář bez jakéhokoliv upozornění na načítání zavře – uživatel musí být korektně informován o konci načítání až ve chvíli, kdy úplně všechny komponenty získají odpověď z API a vše zpracují, po uložení formuláře je třeba taktéž zobrazit načítání,
- P12 lepší popisy polí:** některá pole nejsou dostatečně popsána, např. chybí jednotky pro příslušnou hodnotu, chybí vysvětlení např. automaticky zaškrtnutých polí při zaškrtnutí jiného, na macOS se pro datum a čas

nezobrazí žádná nápověda formátu (Safari oproti jiným prohlížečům nenabízí uživatelsky přívětivý nativní prvek pro jednoduchý výběr) – doplnit lepší popisy, vysvětlení a nápovědu pro formáty (`placeholder`),

- P13 listování diářem:** pokud lektorka listuje rychle diářem mezi týdny, aplikace je pomalá a tlačítka pro pohyb odskakují podle délky zobrazeného data a mění tak svou pozici, také je nepohodlný pohyb mezi předcházejícím a nadcházejícím týdnem – opravit pozici tlačítek tak, aby nezávisela na počtu znaků data, nastavit prodlevu na požadavky při rychlém procházení diářem (jinak zbytečně probíhá komunikace s API a stahování dat pro každý den, ačkoliv uživatel na daný den už vůbec nekouká, protože rychle proklikl na jiný týden), umožnit procházení diáře šipkami na klávesnici,
- P14 zalamování textů:** některé texty na stránkách se nevhodně zalamují (např. telefonní čísla ad.) – opravit a nezalamovat,
- P15 ESC u select:** při stisku ESC u některých prvků formuláře (`select`, `react-select`) se zavře nečekaně celé modální okno namísto zavření příslušné rozbalovací nabídky – opravit tak, aby se zavřel pouze výběr možností, nikoliv celé modální okno s formulářem,
- P16 responzivita:** některé komponenty v aplikaci při zobrazení na jiné než obvyklé velikosti obrazovky činí použití aplikace a srozumitelnost dat značně náročnější, případně se dokonce některé údaje mohou skrývat – opravit responzivitu napříč všemi různými velikostmi displeje,
- P17 abecední řazení:** řazení podle abecedy nepodporuje znaky s diakritikou, dojde např. ke smíchání příjmení klientů začínajících na „S“ a „Š“ a lektorka se špatně orientuje – opravit podporu pro české znaky,
- P18 výstižné nadpisy:** některé nadpisy v rámci aplikace nejsou úplně výstižné a konzistentní – projít napříč aplikací a opravit,
- P19 obnovení přihlašovacího tokenu:** pokud lektorka zůstane na jedné stránce, kde provádí nějaké změny (např. má otevřenou kartu klienta a zde provádí změny a nepřechází jinam), vyprší mezitím platnost tokenu a následně při přechodu na jinou stránku je z aplikace odhlášena – je potřeba provádět automatickou obnovu platnosti tokenu kdekoli v aplikaci (v současné době se provádí jen při přechodu mezi stránkami),
- P20 react-select mazání:** u komponenty `react-select` pro evidování členů skupiny je zobrazen vedle tlačítka pro rozbalení seznamu členů křížek, který po kliknutí smaže všechny členy – lektorka na něj často omylem kliká při pokusu o rozbalení seznamu klientů – křížek odstranit, funkcionality mazání všech členů není vůbec potřeba.

Seznam problémů, které již jsou součástí požadavků:

- titulky stránek v prohlížeči nejsou odlišné – řeší požadavek **F16**,
- chybová hlášení někdy nezmiňují možnosti řešení a nejsou úplně srozumitelná, jsou dlouhá a lektorka je nestihne přečíst – řeší požadavek **F15**,
- pokud lektorka omylem zavře rozpracovaný formulář, není cesty zpět, to je velmi stresující – řeší požadavek **F14**,
- mnoho kroků v rámci aplikace by šlo dělat efektivněji, pokud by to aplikace umožňovala, zbytečně se ztrácí čas – řeší požadavky **F11**, **F4**, **F3**,
- formulář pro lekce je nepřehledný, nevyužívá nijak ani barev, ani jiných prvků k lepší orientaci, v případě mnoha klientů je moc dlouhý a nevejde se na obrazovku – řeší požadavek **F5**.

Návrh

V této kapitole se budu věnovat návrhu aktualizovaného logického datového modelu aplikace, komunikačního rozhraní, architektury a také navrhnu novou konfiguraci více prostředí pro nasazování. Na závěr se budu věnovat návrhu UI zájemců o kurzy a vylepšením UI v dalších oblastech aplikace na základě požadavků.

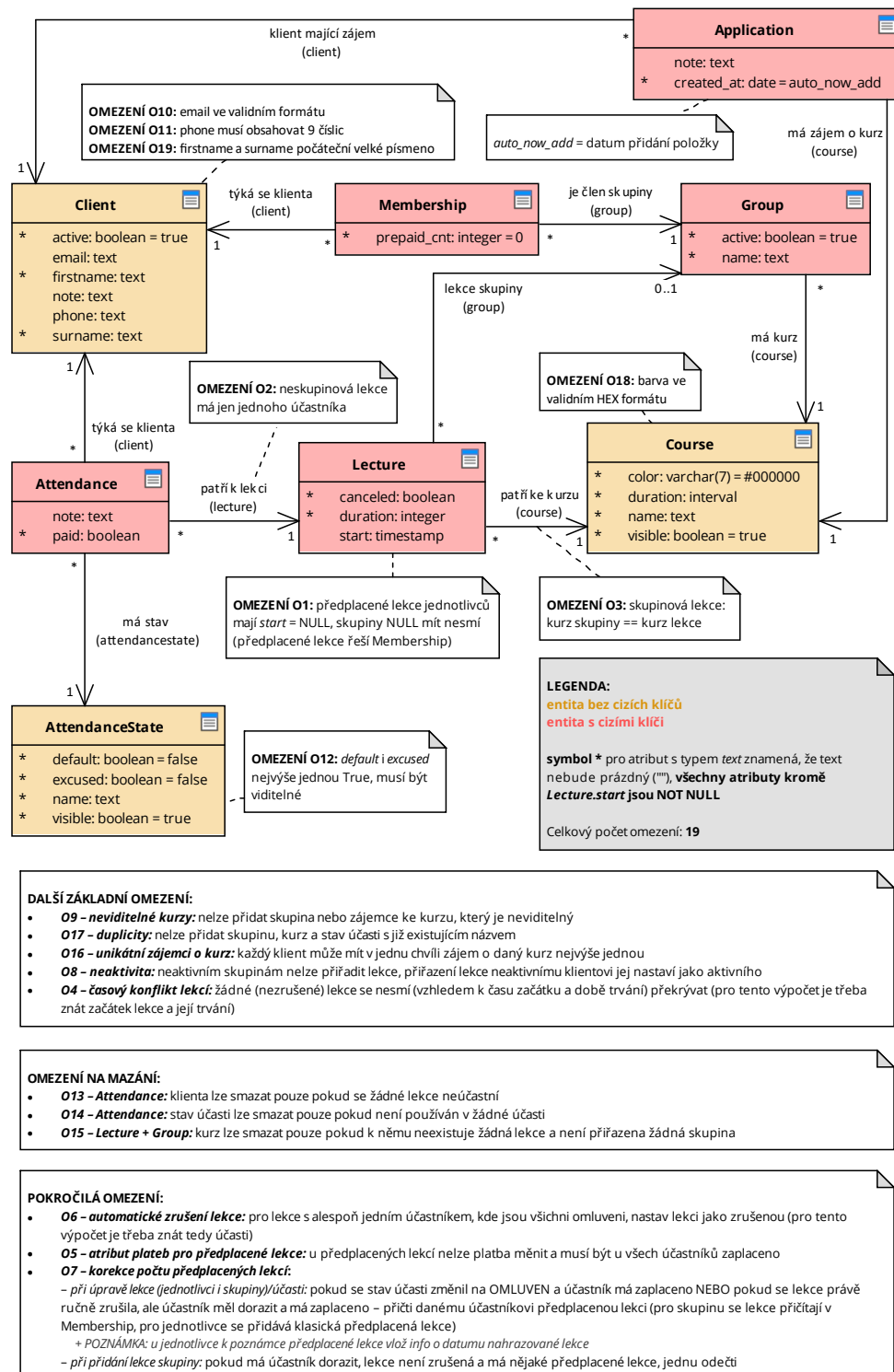
7.1 Datový model

Do datového modelu bylo potřeba projevit všechny nové funkční požadavky. Původní logický datový model je na obrázku 2.1. Finální návrh nového logického datového modelu je na obrázku 7.1.

Nejprve shrnu obecné úpravy a vylepšení tohoto modelu. Během práce se ukázalo velmi vhodné obarvit jednotlivé entity dvěma barvami v závislosti na tom, zda obsahují cizí klíče (oranžová) nebo nikoliv (červená). Popisy vztahů nyní v závorce obsahují i atribut pro přístup k cílové entitě z entity zdrojové, který se fyzicky v rámci aplikace používá. Napříč celým diagramem jsou pak vypsána všechna omezení v rámci domény, která je v kódu třeba řešit – mají unikátní identifikátory, které budou i jako komentáře v kódu, jež tato omezení implementuje, aby byla omezení jednoduše v kódu trasovatelná v případě úprav (ke kterým v průběhu iteračního vývoje docházelo, ale diagram je ve finální verzi). V legendě je též uveden celkový počet omezení, aby se identifikátory nových omezení snadno vytvářely. Atributy jsou pro přehlednost nyní řazeny dle abecedy.

Taktéž se ukázalo jako výhodné dodefinovat význam povinnosti atributu typu `text`, jak totiž uvádí [161], je obvykle nesmyslné povolit na textovém atributu jak prázdný `string`, tak `NULL`, protože pak dvě hodnoty mají stejný význam (žádná data) – proto je v legendě uvedeno, že znak `*` u atributu znamená, že pole nikdy nebude `NULL` a zároveň ani prázdný `string` (prázdný `string` je řešen validací ze strany Django, naproti tomu `NOT NULL` podmínku

7. NÁVRH



Obrázek 7.1: Aktualizovaný a rozšířený původní logický datový model z [1]

řeší přímo databáze). Jedinou výjimkou (též uvedeno v legendě) je atribut **start** lekce, kde **NULL** znamená předplacenou lekci jednotlivce. Tato povinnost atributů byla napříč entitami revidována a doplněna, stejně jako byly u některých atributů doplněny i výchozí hodnoty a řešení duplicit dle požadavků lektorky (např. klient může mít zájem o daný kurz jen jednou).

V následujících podsekcích se zaměřím na konkrétní změny a přidání entit. Součástí uvedených změn je i zavedení mnoha souvisejících omezení, explicitně je ale uvádět nebudu, protože v přehlednější podobě jsou zapsána právě u datového modelu na obrázku 7.1.

7.1.1 Předplacené lekce

V předchozím odstavci jsem se dotkl hodnoty **NULL** pro **start** lekce – zde nastává změna oproti původnímu modelu, protože nyní jsou takto vedeny pouze předplacené lekce jednotlivců (skupin už nikoliv). Díky tomu pak může být u každé takové předplacené lekce, která je vytvořena jako náhrada při omluvě/zrušení lekce, evidován datum nahrazované lekce (viz požadavek **F10**).

Předplacené lekce skupin jsou pro umožnění jednoduché evidence (viz požadavek **F3**) nyní součástí členství klientů ve skupině (Membership), tato entita původně byla dekompozicí vztahu M:N bez dalších atributů, prapůvodní záměr v rámci bakalářské práce byl evidování počátku a konce členství klienta ve skupině, což se pak ukázalo jako zbytečné, ale dekompozice byla pro případné jiné budoucí využití ponechána [1]. Nyní tedy konečně dostala svému řádnému využití a umožní tak evidovat počet předplacených lekcí každého klienta v rámci dané skupiny – když je klient ze skupiny smazán, smaže se samozřejmě i jeho členství a s tím i předplacené skupiny.

7.1.2 Zájemci o kurzy

Dalším funkčním požadavkem **F1**, který je třeba projevit do datového modelu, jsou zájemci o kurzy. Zde se dá s výhodou jednoduše rozšířit původní datový model o entitu navázanou na klienta a kurz – tím docílím evidování zájmu klienta o daný kurz (entita Application). Zájem klienta o kurz obsahuje poznámku a také datum přidání, který bude jednoduše automaticky přidávaný, aby lektorka nemusela nic vyplňovat.

7.1.3 Vlastnosti stavů účasti

Požadavek **F17** uvádí, že je třeba některým stavům přiřadit speciální úlohu – jeden označit jako výchozí (a zároveň ten, který znamená, že uživatel má přijít/přišel) a druhý jako stav s významem „klient omluven“. Původně toto bylo řešeno, jak též uvádí **F17**, pevným zadefinováním v kódu podle názvu stavu účasti, což je ale nedostačující, protože je samozřejmě třeba umožnit upravovat tento název.

Nabízelo se několik možností řešení – zavést nový atribut stavu účasti, který bude označovat typ stavu účasti (jako tomu bývá např. při evidování úrovně oprávnění uživatelů v rámci aplikací). Samotné typy by pak musely být evidovány jako další nová entita. Vzhledem k tomu, že se nepočítá s přidáním dalších typů v budoucnu (stavů účasti také ne, ale ty klidně být přidány mohou, jen nebudou mít speciální typ, protože ten je zde zaveden kvůli dalším speciálním výpočtům jako např. počet absolvovaných lekcí v rámci kurzu), toto řešení nebylo zvoleno z důvodu zbytečné complexity. Byl zvolen jednodušší způsob, kdy stav účasti má dva **boolean** atributy **default** a **excused**, kde serverová část obstará, že právě jedna instance stavu účasti bude mít příslušný atribut aktivní. Toto řešení není tak dobře škálovatelné, ale jak jsem uvedl, není to třeba – je jednoduché a nevyžaduje oproti druhému řešení mnoho změn na úrovni API a klientské části.

7.1.4 Další změny

Aktivita klientů a skupin (viz požadavek **F6**) se vzhledem k návrhu dá vyřešit jednoduchým atributem **active** u obou těchto entit.

V případě kurzu bylo třeba pro evidování délky trvání kurzu (viz požadavek **F7**) pro jednotlivce přidat příslušný atribut **duration**. Dalším přidáním atributem je pak **color**, který umožní u kurzu evidovat jeho barvu (viz požadavek **F12**).

Další změnou, která byla provedena navíc oproti požadavkům bylo přejmenování atributu **name** u klienta na **firstname**. Kromě více vystihujícího názvu (jedná se skutečně o křestní jméno) se zde během vývoje vyskytl problém s nejednoznačností, kde se v rámci klientské části někde pracovalo s **name** jakožto celým jménem klienta, kdežto jinde jako s křestním jménem.

7.2 Komunikační rozhraní

Ukázalo se, že původní komunikační rozhraní (REST API) z bakalářské práce mělo velmi dobrý návrh, bylo třeba provést pouze drobnější úpravy a především začlenit všechny potřebné změny z požadavků a datového modelu. Nejprve se zaměřím na drobnější změny a poté na začlenění změn z požadavků a datového modelu. V rámci této kapitoly nebudu uvádět přesnou novou podobu API včetně původních bodů, ale pouze změny, v rámci požadavku **N1** totiž bude dostupná dokumentace celého API.

7.2.1 Opravy a úpravy stávajícího rozhraní

Pro odpověď na GET požadavek na lekce (GET `lectures/`) byl u každé účasti klienta klíč **count** pro označení pořadového čísla lekce. Zde jsou dva problémy. Prvním je fakt, že název klíče není úplně přesně vypovídající název vzhledem

k dané situaci a při práci v kódu nastávaly nedorozumění, proto došlo k přejmenování na `number`, což lépe odpovídá tomu, že se jedná a pořadové číslo lekce. Druhý problém je, že se z neznámého důvodu tento klíč vyskytoval u každé účasti v rámci lekce, což v případě lekce jednotlivce není důležité, ale v případě skupiny je pak totéž číslo u každé účasti (protože se řeší celkový počet lekcí, nikoliv zda konkrétní klient na nějaké lekci byl) – tedy zbytečně se informace duplikuje a na klientské části se vezme její první výskyt – toto bylo opraveno a pořadové číslo lekce se nyní vyskytuje přímo u lekce, nikoliv u každé účasti.

Součástí odpovědi `GET lectures/`, jak již bylo zmíněno, je také přehled účastí jednotlivých klientů, zde bylo rozhodnuto také o odstranění vnořených informací o stavech účasti jednotlivých klientů – v praxi zde byl poslán vždy název účasti, ID (a nově vzhledem k novému datovému modelu by byly poslány i informace `default` a `excused`), vzhledem k plánovaným změnám v rámci požadavku **N6** (optimalizace API) tyto vnořené informace byly odstraněny a nahrazeny pouze ID stavu účasti, protože si klientská část aplikace bude stavy účasti pamatovat (díky zavedení React Context API, implementace viz podsektce 8.2.5).

U klientů, jak bylo uvedeno v datovém modelu v předchozí sekci 7.1, se přejmenoval klíč pro křestní jméno na `firstname`.

7.2.2 Předplacené lekce

Pro lepší evidenci předplacených lekcí jednotlivce (viz požadavek **F3**) bylo třeba umožnit nějakým způsobem na API zaslat požadavek na přidání daného počtu předplacených lekcí. Nejprve byla zvážena možnost vytváření daného počtu lekcí pomocí zaslání POST požadavku na `lectures/prepaid/` obsahujícím příslušný počet předplacených lekcí, vzhledem k jednodušší implementaci a větší univerzálnosti bylo místo toho umožněno zaslat POST požadavek obsahující více různých lekcí (tedy např. i více předplacených lekcí) na `lectures/`.

Pro lepší evidenci předplacených lekcí skupin je na základě předchozí sekce 7.1 zaveden nový bod `memberhips`. V kódu již byl zaveden serializer pro `Membership`, ten ale nebude použit, protože by pak bod umožňoval upravovat i ID klienta, kterému členství náleží, což zde není třeba – bude tedy vytvořen druhý serializer pro `Membership`, který umožní upravit pouze `prepaid_cnt`. Bod pracuje s klíči `id` a `prepaid_cnt` a jeho podoba je následující:

<code>memberhips/:id/</code>	PUT	úprava členství s <code>id</code>
<code>memberhips/:id/</code>	PATCH	částečná úprava členství s <code>id</code>

7.2.3 Banka

Přidání úplně nového bodu nastalo kvůli požadavku **F8** na zobrazení transakcí z banky – zde API bude nově umožňovat **GET** na **bank/**, odpověď bude především obsahovat samotná data z banky, která budou dle potřeby transformována (upravena, doplněna, zjednodušena). Pokud by tento bod na API nebyl a klientská část by do banky přistupovala na přímo, znamenalo by to, že nelze provést žádné transformace, zjednodušení dat, přidání dalších dat a také by součástí klientské části musel být token do banky, což je nepřípustné.

Lektorka pro ÚP používá bankovní účet u Fio banky, která nabízí zdarma možnost zřízení přístupu k datům účtu přes API, pomocí získaného tokenu je možné každých 30 sekund zaslat na API požadavek [162]. Vzhledem k tomuto časovému omezení (kde by se jinak lektorka při dalším načtení stránky dočkala chybové hlášky) bylo rozhodnuto o cachování získaných dat z banky po dobu 60 sekund.

7.2.4 Zájemci o kurzy

Pro evidenci zájemců o klienty byl též vytvořen nový bod, který pracuje s klíči **id**, **note**, **created_at** a dále obsahuje vnořené informace o kurzu (klíč **course**) a klientovi (klíč **client**). Po vzoru ostatních původních bodů se pro úpravy a vytváření zájemců místo vnořených informací zasílá pouze ID a klíč je ve tvaru **klíč_id** – tedy **client_id** a **course_id**.

Podoba bodu je následující:

applications/	GET	vrátí všechny zájemce
applications/	POST	vytvoření nového zájemce
applications/:id/	GET	vrátí zájemce s id
applications/:id/	PUT	úprava zájemce s id
applications/:id/	PATCH	částečná úprava zájemce s id
applications/:id/	DELETE	smazání zájemce s id

7.2.5 Další rozšíření

Kvůli požadavku **F6** pro evidenci aktivních a neaktivních klientů byla pro klienty a lekce zavedena možnost filtrování pomocí „query string“:

- **groups/?active=:boolean**,
- **clients/?active=:boolean**.

V rámci požadavku **N6** pro optimalizaci API bylo také přidáno filtrování pro kurzy dle viditelnosti – **courses/?visible=:boolean**.

Do API bylo také třeba projevit změny z datového modelu v předchozí sekci 7.1. Pro klienty a skupiny přibyl nový klíč **active** znázorňující aktivitu klienta/skupiny. Pro kurzy byl přidán nový klíč **duration** pro evidování délky

trvání kurzu a `color` pro evidování barvy kurzu. Stejně tak zde došlo k projevení změn povinných atributů, povolených hodnot ad., vzhledem k použití Django REST Framework ale není v této oblasti na API provádět v kódu žádné změny, protože se projeví automaticky z datové vrstvy (modelů).

Dále, vzhledem ke zvolenému způsobu řešení evidování vlastností stavů účasti v předchozí sekci 7.1 je třeba v bodu `attendances/` umožnit pracovat nově i s klíči `default` a `excused`, to je velmi jednoduché (proto byl také tento přístup zvolen).

Pro implementaci požadavku **F9**, který má umožnit jednoduchou automatickou změnu účastníků lekce na klienty, kteří jsou členové skupiny (tedy pokud už např. někdo členem není, tak jej z účastníků odebrat, resp. když byl jako člen přidán a účastník nebyl, tak jej jako účastníka přidat), je třeba umožnit na API toto volitelně učinit. Pro toto je nově zaveden pro bod `lectures/` klíč `refresh_clients` (pro operace PUT, POST, PATCH), výchozí hodnota je `false` (obvykle toto projevení změn účastníků požadováno nebude), při opačné hodnotě pak serverová část zařídí projevení požadovaných změn.

7.3 Architektura

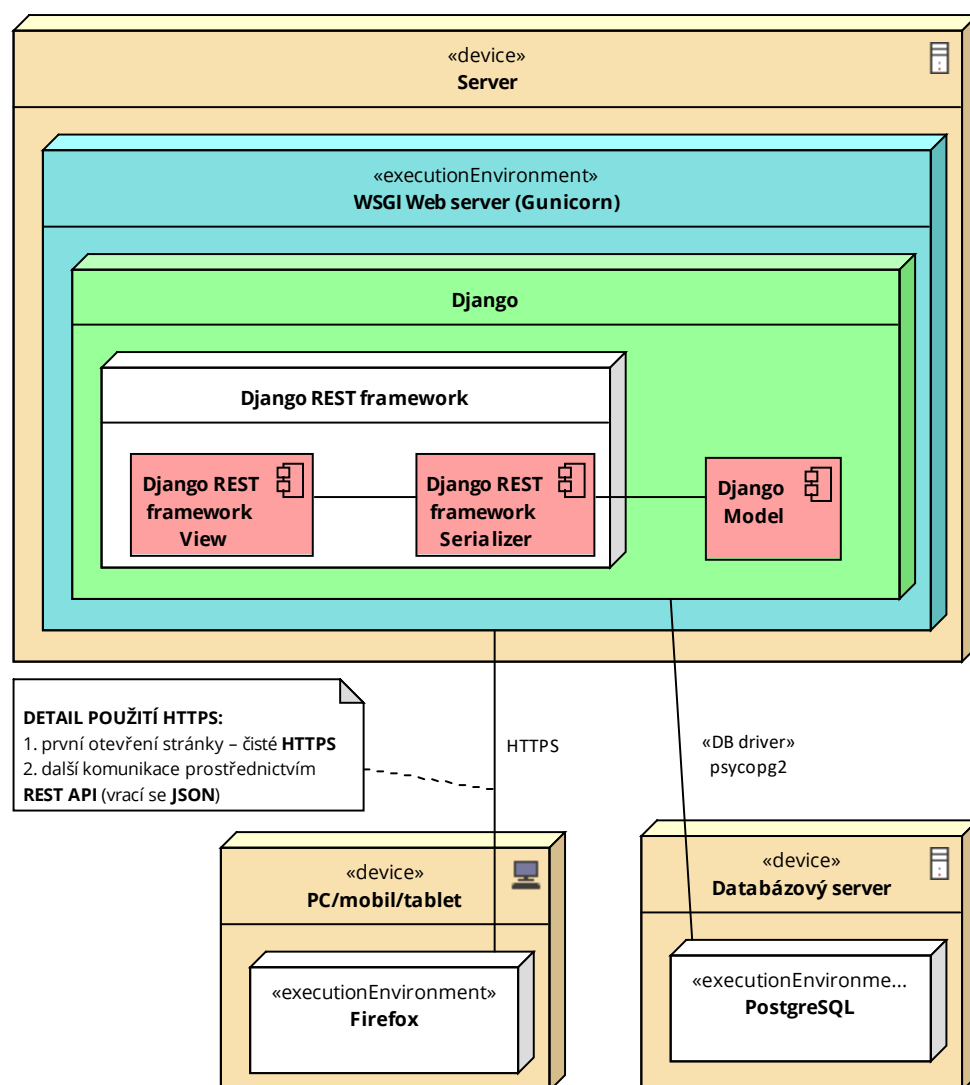
Aktualizovaný diagram nasazení na obrázku 7.2 vychází z původního diagramu [1]. Jádro zůstává stejné a kromě drobnějších vylepšení pro zlepšení přehlednosti je v něm pouze jedna důležitá změna.

V původním diagramu byl jako protokol pro komunikaci mezi serverem a klientem uvedeno HTTP/S. Vzhledem k požadavku na revizi bezpečnosti **N4** a jeho následné detailní analýze 6.2.4 je v problému **B4** zmíněno zavedení HSTS. Podrobnému popisu se budu věnovat v podsekcí 8.2.3, důležitý je ale dopad na návrh architektury, kde díky zavedení HSTS a korektní konfiguraci aplikace všechna komunikace bude probíhat z důvodu bezpečnosti pouze přes HTTPS (Hypertext Transfer Protocol Secure).

7.4 Konfigurace více prostředí

V této sekci se budu zabývat návrhem více prostředí pro nasazování aplikace. Nejprve několik úvodních vět pro uvedení do problému. V současnosti je aplikace nasazena pouze v produkčním prostředí (viz popis aktuálního řešení v sekci 2.3), kam se automaticky nasazuje při každém úspěšném sestavení na CI. V rámci požadavku **N7** je uvedeno, že je třeba prostředí totožné s produkcí (např. pro reprodukci chyb) a prostředí s aplikací sestavenou na základě poslední revize v repozitáři.

Jak uvádí [163], více prostředí může jít ruku v ruce se vznikem příslušných větví v repozitáři, jejichž názvy odpovídají prostředím, do kterých se nasazují, probíhají zde slučování změn z jednotlivých větví do jiných a poté nasazování do příslušných prostředí. Vzhledem k tomu, že se jedná o aplikaci na míru



Obrázek 7.2: Aktualizovaný diagram nasazení, vychází z [1]

vyvíjenou pouze mnou a nejedná se o obrovský projekt, tento, ač často užívaný postup, jsem se rozhodl zde nezavést. Místo něj jsem se zaměřil na vytvoření vhodnějšího přístupu vzhledem ke způsobu práce na projektu.

Obvyklý vývoj v repozitáři probíhá pomocí vytvoření nové větve s danou novou funkcí, otestování a začlenění změn do hlavní vývojové větve. Některé menší změny jsou případně prováděny rovnou ve výchozí větvi. Na tomto pracovním postupu jsem tedy postavil způsob řešení více prostředí, který bude vhodný pro tento projekt na základě požadavků a dosavadních znalostí projektu.

Byl vytvořen návrh více prostředí bez využití příslušných větví:

- **vývojové (lokální):** pro lokální vývoj,
- **testing:** nasazení každé revize (nehledě na větev),
- **staging:** nasazení pro otagované revize, stejná verze aplikace jako produkce,
- **produkce:** nasazení pro otagované revize, aplikace používaná lektorkou.

Jakákoliv nová revize (nezávisle na větvi) v repozitáři bude sestavena a automaticky otestována na integračním serveru a nasazena do prostředí „testing“. Toto prostředí bude velmi podobné produkčnímu, až na verzi aplikace – jakékoliv změny v aplikaci po provedení `git push` budou tedy vidět nasazené v reálném prostředí – bude je zde možné pak testovat manuálně jak mnou, tak lektorkou, řešit další úpravy apod. Zde podotknu, že samozřejmě stále zůstává vývojové (lokální prostředí).

Až bude vše vyladěno a připraveno na nasazení do produkce, vydá se nová verze aplikace pomocí tagu v gitu (resp. release v GitHubu) – pak, když tento otagovaný commit dorazí na CI, proběhnou opět tytéž kroky jako v případě běžného commitu, výsledná aplikace bude ale nasazena nejen do prostředí „testing“, ale také na „staging“ a produkci. Snažím se zde respektovat obecně známé názvy prostředí (jak třeba uvádí [163, 164]). Prostředí „staging“ je přesná kopie produkční verze odlišná pouze svou instancí databáze a slouží např. pro reprodukci problémů hlášených z produkce. Produkční prostředí je verze aplikace používaná lektorkou.

Díky popsanému návrhu se může na produkci nasazovat až v případě, kdy máme vysokou jistotu hladkého běhu na produkci, tedy například po důkladném automatizovaném i manuálním otestování (v závislosti na změnách). Díky zavedení „staging“ prostředí lze snadno reprodukovat problémy mimo produkci, díky „testing“ prostředí lze okamžitě vidět aktuální změny nasazené v reálném prostředí. Navrženému způsobu dodávání nových verzí aplikace se říká průběžné dodávání (CD – „Continuous Delivery“) [165] a jak uvádí [165], pro správné fungování CD je třeba mít dostatečně pokrytou aplikaci testy, což řeší požadavek **N2**. Z hlediska existence více prostředí a jejich co největší podobnosti je v této oblasti vycházeno také z příslušného pravidla v metodice „The Twelve-Factor App“ [166]. Jednoduchou prací s tokeny a dalšími položkami napříč prostředími umožní zavedení různých příslušných proměnných prostředí, viz informace v požadavku **N7**.

7.5 Uživatelské prostředí

Do klientské části bylo třeba navrhnout několik prvků z požadavků. Drobnější změny zde ukázány nebudou, zaměřím se jen na nejdůležitější změny v uživa-

7. NÁVRH

telském rozhraní. Návrhy obvykle probíhaly na papír, ale pro lepší čitelnost je zde uvádím překreslené v aplikaci Pencil.

7.5.1 Zájemci o kurzy

ÚP - Zájemci o kurz				
ÚPadmin	MENU ZÁJEMCI MENU MENU			
Zájemci				Přidat zájemce
Rozvoj grafomotoriky (3 zájemci)				
František Sok	8. 9. 2018	od 11/2018, byl u logopedky, odklad	Upravit	Smazat
Alena Nová	8. 9. 2018	ihned, skupina s P. Novým, středy	Upravit	Smazat
Petr Nový	8. 9. 2018	ihned, skupina s A. Novou, středy, také ADHD	Upravit	Smazat
Předškolák s ADHD (1 zájemce)				
František Sok	8. 9. 2018	může jen úterky/čtvrty	Upravit	Smazat

Obrázek 7.3: Návrh zájemců o kurzy

Nejvýraznější změnou v klientské části je přidání nové stránky se zájemci kurzu (viz požadavek **F1**). Návrh je na obrázku 7.3, jak lze vidět, jsou barevně odlišeny kurzy – zde již počítám se zavedením požadavku **F12** pro evidování barev kurzů (změny budou v implementaci učiněny napříč celou aplikací). Návrh splňuje všechny požadavky lektorky, kromě jednoduše dostupné úpravy je také dostupné tlačítko pro smazání – to je obvykle napříč aplikací dostupné až v modálním okně s úpravou (protože se obvykle stejně nepoužívá a díky tomu, že není přímo na příslušné hlavní stránce, nemůže být ani omylem stisknuto a potvrzeno smazání ve vyskakovacím okně), zde je přímo u každého zájemce, protože oproti ostatním případům zde frekvence mazání bude vysoká vzhledem k postupnému obsluhování všech zájmů o kurzy.

7.5.2 Formulář pro lekce

Na obrázcích 7.4 a 7.5 jsou kompletně přepracované návrhy formulářů pro přidání (a potažmo i úpravu) lekce skupiny, resp. klienta. V rámci příslušného požadavku **F5** bylo třeba postupnými iteracemi dojít k návrhu, který umožní jednoduší práci s tímto nejpoužívanějším formulářem v aplikaci. V detailní analýze požadavku **F3** v podsececi 6.2.1 bylo zjištěno, že je třeba usnadnit při-

ÚP - Karta skupiny

ÚPadmin MENU MENU MENU MENU

Přidání lekce skupiny: **FIE I** X

xx 10. 03. 2018 xx 17:00

☐ Zrušeno Feuersteinova metoda xx 45

[Novák Pavel](#)

xx OK ☒ **Platba** xx půjčena kniha

[Novák Petr](#)

xx omluven ☐ **Platba** xx

Smazání **Smazat lekci**

Storno Uložit

Obrázek 7.4: Návrh nového formuláře pro skupinové lekce

ÚP - Karta klienta

ÚPadmin MENU MENU MENU MENU

Přidání lekce klienta: **Novák Petr** X

☐ Předplaceno 1 xx 10. 03. 2018 xx 17:00

☐ Zrušeno Feuersteinova metoda xx 45

xx OK ☒ **Platba** xx půjčena kniha

Smazání **Smazat lekci**

Storno Uložit

Obrázek 7.5: Návrh nového formuláře pro lekce jednotlivců

dávání více předplacených lekcí pro jednotlivce, tento požadavek v rámci tohoto přepracování formuláře pro lekce byl též začleněn – pro klienty je v levém horním rohu dostupné pole pro zapsání počtu předplacených lekcí. Toto pole bude možné upravit při zaškrtnutí volby „Předplaceno“. Znaky „XX“ u polí naznačují přítomnost ikony vysvětlující význam příslušného pole (místo textů, alternativní text se samozřejmě zobrazí po najetí na ikonu).

Díky větší šířce formuláře bylo možné vměstnat více polí na méně řádků. Místo bylo také ušetřeno použitím ikon a zobrazením účastí klientů ve formě řádků. Platba klientů je navíc odlišena barvou. Všechny tyto změny napomohly přehlednější a jednodušší evidenci lekcí. Jak bylo uvedeno v požadavku **F5**, jedním z hlavních problémů bylo, že ve skupině může být např. 6 dětí a formulář má pak výšku několika obrazovek a je prakticky nepoužitelný – zjednodušený názorný příklad původního formuláře (pouze se dvěma účastníky pro jednoduchost, ale účastníků je obvykle mnohem více) je na obrázku 7.6. Formuláře pro skupiny a jednotlivce (klienty) se liší jak počítadlem pro předplacené lekce (u skupin tato možnost není, protože bude řešena počítadly v rámci karty skupiny, viz požadavek **F3**). Druhou odlišností je pak zobrazení jména účastníka, které v případě jednotlivce je samozřejmě zbytečné (je v horní části formuláře), v případě skupin je toto jméno klienta současně odkazem do jeho karty pro rychlý přechod v případě potřeby.

Úprava lekce skupiny: Kurz Slabika 1

Naplánováno?

☐ Nenaplánováno, ale předplaceno

Příznaky

☐ Zrušeno

Datum

17 . 01 . 2020

Čas

09 : 09

Trvání

45

Kurz

Kurz Slabika

Lukáš Rod

Stav účasti

OK

Platba

☒ Placeno

Poznámka

Pavel Rod

Stav účasti

nepřišel

Platba

☐ Placeno

Poznámka

Smazání

Smazat lekci

Storno

Uložit

Obrázek 7.6: Původní formulář pro lekce skupin

Implementace

V rámci této kapitoly se budu věnovat způsobu samotné implementace rozšíření aplikace. Nejprve se zaměřím na implementaci funkčních požadavků, poté nefunkčních požadavků a na závěr zmíním další zajímavé problémy řešené během implementace.

8.1 Funkční požadavky

V rámci této sekce popíši způsob implementace všech funkčních požadavků. Pokusím se vždy shrnout jednoduše provedené kroky a také tyto kroky zasadit do kontextu ostatních požadavků a problémů.

8.1.1 F1 – evidování zájemců o kurz

Pro zájemce o kurzy bylo třeba přidat do serverové části aplikace nový model `Application` (viz datový model v sekci 7.1), oproti běžným vlastnostem a atributům zde bylo třeba nastavit pomocí parametru `auto_now_add` automatické nastavení data přidání do atributu `created_at`. Na tomto modelu pak staví nově implementovaný bod API pro zájemce (viz komunikační rozhraní v sekci 7.2). Také byly naimplementovány požadované validace a omezení.

Na klientské části jsem vycházel z návrhu uživatelského rozhraní v sekci 7.5. Bylo třeba přidat do aplikace novou (a v této práci jedinou novou) stránku se zájemci o kurzy a dle návrhu zájemce dělit dle kurzů (API poskytne pouze nerozdělený seznam, tedy toto je třeba řešit na klientské části) a také u každého kurzu dopočítat, kolik je o něj zájemců a zobrazit toto číslo u názvu kurzu (opět dle návrhu). Kurz obsahuje v záhlaví na pozadí svou barvu dle požadavku **F12**. Důležitou součástí je také formulář pro práci se zájemcem – ten umožňuje pomocí rozbalovací nabídky `react-select` zvolit jednoduše kurz a klienta (oproti běžnému `select` nabízí možnost vyhledávání a také v případě kurzu zobrazení jeho barvy, viz požadavek **F12** a související problém

s použitelností `react-select` **P5**) a připsat ještě poznámku. Taktéž, dle problému **P7**, jsou zvýrazněny povinné položky. Kromě výběru již existujícího klienta je také možné jedním klikem přidat klienta nového bez opuštění aktuálního formuláře, toto souvisí s požadavkem **F11** na efektivnější práci v aplikaci a zároveň opravdu dává smysl, protože většina přidávaných klientů jsou klienti noví, nikoliv stávající – tedy lektorka by jinak musela formulář zavřít, přejít do klientů, zde klienta vytvořit a poté přejít zpět do zájemců, díky současné implementaci toto ale není třeba a zájemce lze velmi rychle včetně samotného přidání klienta zpracovat v rámci tohoto formuláře pro zájemce.

8.1.2 F2 – kontrola časového konfliktu lekcí

Kontrola časového konfliktu lekcí má lektorku upozornit ve všech možných případech na překryv dvou lekcí. Existuje mnoho variant, jak se lekce mohou navzájem překrývat, jak ale uvádí [167], lze toto ošetřit jednoduchou podmínkou – časový konflikt mezi lekcí „A“ a přidávanou lekcí „B“ znamená, že:

- „A“ začíná před koncem „B“ a zároveň
- „B“ začíná před koncem „A“.

Konec lekce ale v databázi není přímo uložen, známe jen začátek lekce (atribut `start`) a trvání lekce v minutách (atribut `duration`) – z toho lze ale konec lekce vypočítat. Dotaz na časový konflikt bude tedy mírně složitější a pokročilejší, pro zajímavost finální verzi kódu uvádím v ukázce 1 (v reálném kódu je tento dotaz rozpadlý do několika částí, pro přehlednost jej zde ale uvádím dohromady), ukázka si vyžaduje samozřejmě vysvětlení.

```
qs = Lecture.objects.annotate(  
    end_db=ExpressionWrapper(  
        F("start") + (timedelta(minutes=1) * F("duration")),  
        output_field=DateTimeField()  
    )  
)  
)  
.filter(  
    start__lt=  
        data["start"] + timedelta(minutes=data["duration"]),  
    end_db__gt=data["start"],  
    canceled=False,  
)
```

Ukázka kódu 1: Dotaz pro nalezení časových konfliktů

Dotaz v ukázce 1 se skládá ze dvou částí – první část zajišťuje, že se ke všem lekcím dodá atribut `end_db` s dopočítaným koncem lekce, druhá část

pak nad těmito lekcemi provede filtrování a nalezne lekce v konfliktu. Nyní podrobněji k první části – v `annotate` bylo při tvorbě atributu konce lekce `end_db` třeba použít `ExpressionWrapper`, protože typy jednotlivých atributů lekce získaných pomocí výrazu `F()` nejsou stejné, také je třeba číslo uložené v `duration` „přetypovat“ na typ, který zde lze sčítat s typem atributu `start`.

Ve druhé části jsou zakomponovány obě podmínky pro časový konflikt zmíněné výše a navíc se pracuje pouze se zrušenými lekcemi, pro korektní fungování tohoto dotazu je třeba tedy implementovat také požadavek **F19**, který zajistí automatické rušení lekcí, pokud nemá nikdo dorazit. Zde je třeba ještě říci, že lekce, kde jedna končí např. v 15:00 a druhá v 15:00 začíná, nejsou v konfliktu (díky tomu, že se v dotazu neřeší rovnost), což je požadované. Dále je třeba doplnit, že dotaz je dále v reálném kódu ještě ošetřen tak, aby v případě úpravy trvání/startu lekce nebyl hlášen časový konflikt lekce samotné se sebou.

Součástí validace časového konfliktu je také naimplementovaná srozumitelná chybová zpráva pro lektorku obsahující start a dobu trvání lekce a jméno klienta, kterému lekce náleží (případně skupiny)

8.1.3 F3 – vylepšení předplacených lekcí

Pro pohodlnější evidenci předplacených lekcí bylo třeba na serverové části aplikace upravit model `Membership` (viz datový model v sekci 7.1) – doplnit jej o atribut `prepaid_cnt`. Dále bylo třeba tuto změnu projevit i v API (viz komunikační rozhraní v sekci 7.2), stejně jako doplnit do API možnost zaslání více kurzu najednou (pro pohodlnější evidenci více předplacených lekcí jednotlivce), toto řeší v Django REST Framework příslušný `LectureViewSet`, který umožňuje na základě zaslaných dat (jedna lekce/pole lekcí) tyto data zpracovat.

Všechny tyto změny bylo třeba projevit na klientské části – především dodat pole do formuláře pro přidání lekce jednotlivce, které umožní zadat počet předplacených lekcí (viz návrh UI v sekci 7.5), tato změna byla součástí kompletního přepracování tohoto formuláře v rámci požadavku **F5**. Pro evidenci předplacených lekcí pro skupiny, jak bylo uvedeno v analýze tohoto požadavku v podsekcí 6.2.1, byla do karty skupiny implementována počítadla předplacených lekcí pro jednotlivé klienty. Každý klient má zde hodnotu počtu předplacených lekcí, která lze upravit jak ručně, tak je připravena na automatické projevení změn v počtu předplacených lekcí z požadavku **F10**. Toto řešení je vidět na obrázku 8.1. Pokud skupina nemá žádné členy, místo počítadel se zobrazí info „Žádní účastníci“.

8.1.4 F4 – vyhledávání klientů

Součástí požadavku **F4** na implementaci vyhledávání mezi aktivními klienty je požadavek na vyhledávání založené na podobnosti řetězců, nikoliv přesné

Obrázek 8.1: Implementace počítadel předplacených lekcí ve skupinách

shodě. K tomu byla využita knihovna Fuse.js, která nabízí jednoduché fuzzy vyhledávání bez nutnosti řešení vyhledávání na serverové části.

Pro komplexnější vyhledávání bylo rozhodnuto, že kromě jména a příjmení bude vyhledávání fungovat také pro telefonní čísla a e-maily klientů. Konkrétní konfigurace vyhledávání je v ukázce kódu 2 – je zde vidět, že se výsledky vyhledávání řadí dle podobnosti řetězců od nejlepší shody k nejhorší, kde byl testováním vhodně zvolen příslušný práh, kdy ještě dojde k zobrazení příslušné položky ve výsledcích. Zde je třeba říci, že v rámci vývoje aplikace došlo k vydání nové verze knihovny (byla používána verze 3 a poté došlo k vydání verze 5), která výrazně změnila API, zde tedy uvádím popis pro poslední verzi.

Poslední položku v ukázce kódu 2 zde tvoří klíče, podle kterých se vyhledává – oproti běžnému modelu klienta je zde navíc klíč `normalized`, který zde řeší problém s diakritikou. Knihovna totiž nepodporuje ignorování diakritiky, tedy pokud by lektorka vyhledávala bez diakritiky, výsledky by byly tímto ovlivněny a k nalezení klienta by nemuselo vůbec dojít – tedy toto je vyřešeno. Při použití knihovny ve verzi 3 byl součástí tohoto klíče také složený řetězec jména a příjmení, který sloužil pro přesnější vyhledávání z hlediska celého jména, nikoliv jednotlivých částí jména, ve verzi 5 už ale tato konfigurace není k dispozici a tedy tento řetězec dodán již není, výsledky jsou tedy tímto mírně zkresleny, ale bez vážnější újmy a vyhledávání svou hlavní funkci stále plní, pouze se objeví více možných výsledků vyhledávání.

```
const searchOptions: Fuse.IFuseOptions<ClientActiveType> = {
  shouldSort: true,
  threshold: 0.5,
  keys: ["firstname", "surname", "phone", "email",
    "normalized"],
}
```

Ukázka kódu 2: Konfigurace vyhledávání klientů ze souboru Main.tsx

Zavedené vyhledávání je vidět např. na obrázku 10.1 – je dostupné napříč celou aplikací v menu a při napsání prvního písmena do pole dojde k okamži-

tému zobrazení výsledků vyhledávání spolu s informacemi o klientech (jméno, poznámka, e-mail, telefonní číslo), možností rychlé úpravy klienta (viz požadavek **F11** na efektivní práci v aplikaci) a počtem nalezených klientů. Díky použití knihovny Fuse.js je vyhledávání benevolentní jak vůči překlepům ve vyhledávacím dotazu, tak překlepům v uložených údajích klienta (což se lektorce někdy stává).

8.1.5 F5 – přepracování formuláře pro lekce

Přepracovaný formulář pro lekce byl implementován dle návrhů na obrázcích 7.4 a 7.5. Součástí implementace byla také migrace z běžné rozbalovací nabídky `select` na `react-select` pro lepší použitelnost a možnost zobrazení barev kurzů (viz požadavek **F12** a související problém s použitelností `react-select` **P5**). Na obrázku 8.2 je finální implementovaný formulář pro lekce, konkrétně úprava skupinové lekce (od jednotlivce se, jak je též zmíněno v návrhu, liší zobrazením jmen účastníků a naopak nezobrazuje počítadlo předplacených lekcí). Doporučuji srovnat s podobou původního formuláře na obrázku 7.6.

Obrázek 8.2: Přepracovaný formulář pro lekce

8.1.6 F6 – zavedení aktivních a neaktivních klientů a skupin

Pro zavedení aktivních a neaktivních klientů bylo třeba na serverové části aplikace do modelů `Group` a `Client` vložit atribut `active` (viz datový model v sekci 7.1) a taktéž jej zakomponovat do API (viz komunikační rozhraní

v sekci 7.2) – zde bylo využito propojení modelů s Django REST Frameworkem, tedy do API se změny projeví automaticky. Taktéž byly zavedeny omezení související s aktivitou klientů/skupin, např. neaktivní klient se přidáním nové lekce automaticky změní na aktivního (lektorka je o tomto na klientské části předem taktéž informována).

Na klientské části bylo potřeba zvolit, kde se bude z API stahovat seznam všech klientů a kde naopak jen aktivní/neaktivní – např. při výběru klienta pro přidání lekce stačí stahovat jen aktivní (neaktivnímu klientovi lze lekci přidat pouze v jeho kartě). Dalším důležitým bodem bylo zavedení možnosti filtrování aktivních a neaktivních klientů/skupin na stránkách se seznamy klientů a skupin. Dle požadavku bylo třeba ve výchozím stavu zobrazit jen aktivní a na vyžádání přepnout do neaktivních. Toto je řešeno komponentou na obrázku 8.3. Další implementovanou funkcionalitou nad rámec požadavků pro lepší použitelnost je automatické přepnutí na záložku aktivních/neaktivních klientů/skupin podle toho, do které části patří přidávaný klient/skupina (resp. také upravovaný klient/skupina dle aktivity po uložení).



Obrázek 8.3: Přepínač aktivních a neaktivních klientů

8.1.7 F7 – nastavitelná délka kurzů

Aby bylo možné nastavit délku kurzů pro jednotlivce, bylo třeba na serverové části aplikace do modelu `Course` vložit atribut `duration` (viz datový model v sekci 7.1) a opět jej zakomponovat do API (viz komunikační rozhraní v sekci 7.2), což opět automaticky zařídí Django REST Framework.

Na klientské části bylo třeba jak umožnit samotné upravování hodnoty do formuláře s kurzy. Dále bylo ale také potřeba s touto hodnotou vhodně pracovat v rámci formulářů pro vytváření lekce – tato hodnota totiž, jak uvádí požadavek, platí pouze pro lekce jednotlivců, kdežto lekce skupin mají stále jednu fixní hodnotu. Při automatickém vyplňování hodnoty délky trvání lekce v závislosti na zvoleném kurzu je třeba tedy brát v úvahu, zda se jedná o lekci jednotlivce či skupiny. Na základě toho se pak automaticky vloží daná hodnota do pole pro délku trvání lekce.

8.1.8 F8 – propojení s bankou

Fio banka, u které má lektorka účet ÚP, poskytuje velmi jednoduchý přístup k API a zde jednoduchý přístup k transakcím – token se získá v administraci samotného účtu. V datovém modelu aplikace není třeba řešit žádné změny, je třeba ale připravit API pro přístup k transakcím (viz návrh komunikačního

rozhraní v sekci 7.2) – byl vytvořen bod `bank/` umožňující metodou GET získat transakce za poslední 3 týdny. Oproti ostatním bodům, které jsou založeny na datových modelech a používají tak pro vytvoření bodu `ModelViewSet`, zde bylo třeba využít obecnější `APIView`, protože s modely nepracujeme. Příslušná metoda `get` je oánotovaná pomocí dekorátoru v ukázce kódu 3 – díky tomu jsou získané výsledky z banky uloženy do cache na 60 s z důvodu omezeného počtu požadavků na API banky (jednou za 30 s).

```
@method_decorator(cache_page(60))
```




Ukázka kódu 3: Dekorátor pro zavedení cache

Pro použití cache bylo třeba nakonfigurovat Django – to nabízí mnoho možností ukládání a práce s cache, pro jednoduchost ale bylo zvoleno použití `LocMemCache`, tedy ukládání cache do lokální paměti procesu [168]. V případě pokročilejší práce s cache či pokročilejší aplikace běžící ve více procesech by bylo třeba použít sofistikovanějších možností jako např. databáze Memcached [168].

Logika práce s bankovním API byla oddělena do samostatného souboru `services.py`, získaná data z banky zde projdou transformací, zjednodušením a doplněním – odstraní se nepotřebné položky o účtu, doplní se klíč `fetch_timestamp` (timestamp stažených dat z banky, aby bylo jasné, ze kdy získaná data z banky pocházejí), klíč `rent_price` (výše nájmu v Kč), seřadí se transakce od nejnovějších po nejstarší a hodnota zůstatku na účtu se sníží o 100 Kč (minimální zůstatek na Fio účtu, tedy nelze toto započítat do reálně použitelných prostředků, ačkoliv API banky vrátí hodnotu zůstatku včetně této hodnoty). Celá logika je připravena na jakékoliv výskyty chyb ze strany banky, ošetřuje své chování a korektně vrací stavové kódy HTTP. V případě stavových kódů zde bylo otázkou, jak řešit vrácení kódu pro požadavky, které zkolabují ze strany API banky, ale ze strany API aplikace proběhnou prakticky korektně, jen neobsahují získaná data – kód 200 sice dává smysl z hlediska toho, že samotné naše API nemá problém, ale prakticky došlo k interní chybě na serveru, tedy proto zde dojde k vrácení kódu 500 a součástí odpovědi je i přesný popis problému, který nastal.

Na klientské části byla dle požadavku vytvořena komponenta **Bank** pro napojení na banku umístěná přímo na hlavní stránku do přehledu (zde se tedy nyní kromě přehledu lekcí pro dnešní den nachází přehled transakcí). Lektorka tak už nemusí chodit často do bankovníctví a kontrolovat účet a platby. Součástí této komponenty je i tlačítko umožňující jednoduché znovunačtení dat z banky a také možnost rychle přejít do samotného bankovníctví, pokud je třeba např. zaslat platbu, pokročile vyhledávat či přistoupit k transakcím starších 3 týdnů. Výše nájmu je zde využita k tomu, aby mohla být lektorka případně upozorněna přímo na hlavní stránce na nedostatek prostředků pro

uhrazení nájmu na účtu. Zobrazují se dle požadavku transakce za poslední 3 týdny a také aktuální zůstatek na účtu. Součástí přehledu každé transakce je srozumitelný datum (pro blízké dny se také zobrazí např. „včera“, „dnes“), částka, barevné rozlišení příchozí a odchozí platby (kromě odlišení znakem „-“), poznámka k platbě a zpráva pro příjemce. Z analýzy seznamu transakcí banky vyplývá, že v poznámce většiny transakcí se nachází jméno vlastníka účtu, z toho důvodu bylo rozhodnuto v rámci klientské části údaje o transakcích zjednodušit uvedenou formou, tedy zobrazit pouze poznámku (a v případě, že chybí, tak vlastníka účtu) a případně zprávu pro příjemce, je-li uvedena. Transakce z dnešního dne jsou žlutě zvýrazněny dle následného požadavku lektorky. Výsledná komponenta je na obrázku 8.4.

Bankovní účet			
Aktuální stav: 1234,56 Kč			
Čas výpisu: 19:30:54  Bankovníctví 			
Poznámka	Zpráva pro příjemce	Datum	Suma
Rod Lukáš	Jan Lukáš - předškoláček 1h + Elko	dnes	320 Kč
Petr Novák	David Novák	včera	1 200 Kč
inkaso sazkamobil (soukromé číslo)	---	včera	-80 Kč
Mařík Josef	Josef Mařík - kurz Slabika	čt 14. 11. 2019	100 Kč
Gregor Lukáš	Milan Gregor	po 11. 11. 2019	444 Kč
Jan Novotný	GMT Novotný	po 4. 11. 2019	353 Kč
 Transakce starší než 3 týdny lze zobrazit pouze v bankovníctví .			

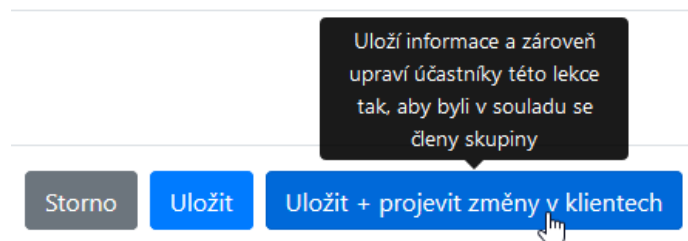
Obrázek 8.4: Komponenta pro informace z banky

8.1.9 F9 – změny účastníků skupinových lekcí

Jak uvádí požadavek **F9**, je třeba umožnit lektorce jednoduché automatické projevení změn účastníků lekce v závislosti na aktuálních členech. Pro toto bylo třeba dle návrhu komunikačního rozhraní v sekci 7.2 zavést nový klíč `refresh_clients` pro operace PUT, PATCH, POST na lekcích.

Tento klíč bylo poté třeba využít na klientské části takovým způsobem, aby si lektorka mohla vybrat, zda chce úpravy/přidání lekce provést bez změn v účastnících, nebo změny automaticky během ukládání provést. Pro toto bylo zvoleno vytvoření druhého tlačítka pro uložení formuláře s popisem „Uložit + projevit změny v klientech“, po najetí se také zobrazí více vysvětlující popis.

Po kliknutí na toto tlačítko dojde jak k uložení změn v lekci, tak k projevení změn v účastnících (po kliknutí na běžné tlačítko „Uložit“ dojde jen k uložení změn).



Obrázek 8.5: Tlačítko pro projevení změn účastníků lekce

8.1.10 F10 – automatické přidání předplacené lekce

Implementace tohoto požadavku souvisí s požadavkem **F3** na lepší evidenci předplacených lekcí. Na základě implementace zmíněného požadavku bylo pak možné kód rozšířit a upravit o automatické přidání předplacených lekcí.

Pro jednotlivce bylo naimplementováno, aby se při omluvě klienta/zrušení lekce automaticky klientovi přidala předplacená lekce s poznámkou např. „Náhrada lekce (7. 4. 2020)“. V případě lekcí skupin bylo využito implementace požadavku **F3** a členovi skupiny se předplacená lekce přidá do jeho členství (přičte se k `prepaid_cnt`). V obou případech musely být v kódu přesně stanoveny podmínky, aby došlo k přičtení předplacené lekce právě jednou a nikoliv vícekrát – například při úpravě již zrušené lekce tedy nesmí být další lekce přičtena.

Součástí požadavku je ale také automatické odebírání předplacených lekcí pro skupiny, což je třeba řešit jak na serverové části, kde dojde k samotnému odečtení předplacených lekcí při přidávání klientů, tak na klientské části, kde se při přidávání lekce automaticky u účastníků označí lekce jako zaplacená v případě, že mají nějaké předplacené lekce. Lektorka tak vůbec nemusí řešit, který klient má danou přidávanou lekci zaplacenou, protože to za ni označí aplikace a po uložení se automaticky odečtou předplacené lekce (pokud klient nějaké má).

8.1.11 F11 – efektivnější práce v rámci aplikace

V rámci analýzy požadavku **F11** byl uveden podrobný výčet oblastí, kde je třeba zefektivnit práci v aplikaci – obecně řečeno se jedná o oblasti, ve kterých je jasné, že je třeba provést určitý úkon a zde pak také vidět výsledky po uložení, ale aplikace toto nenabízí a příslušnou oblast je třeba opustit, provést úkon jinde a manuálně se zpět vrátit do původní oblasti. Problémem je, že původní oblast navíc nemusí být přímo přístupná, nemusí se totiž např. jednat

o pouhou stránku s diářem, ale o nějaký formulář, ve kterém je třeba vybrat např. klienta, ale klient zatím není vytvořený – nejen, že je třeba vše zrušit a přejít do klientů a zde přidat klienta, ale pak je třeba ještě jít zpět na původní formulář, což je skutečně mnoho kroků navíc.

Původní verze aplikace na toto přeužívání formulářů nebyla připravena. Bylo tedy třeba vymyslet takový způsob přeužívání formulářů, že je bude možné využít napříč aplikací bez jakékoliv duplikace kódu a co možná nejjednodušeji.

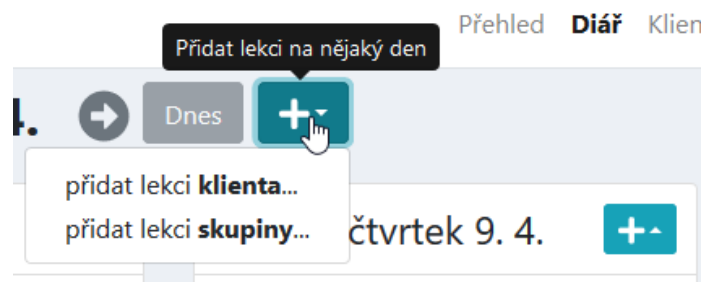
Řešení přiblížím na konkrétním příkladu – dle požadavku je třeba umožnit úpravu klienta nejen ze seznamu klientů, ale také z karty klienta. Podívejme se nyní na aktuální situaci v seznamu klientů – je zde tlačítko pro úpravu klienta, kde po kliknutí lze prostřednictvím formuláře v nově otevřeném modálním okně provádět úpravy. V kódu této stránky je jak příslušné tlačítko, tak i kód obstarávající zobrazení modálního okna, potomkem tohoto modálního okna je uveden pak samotný formulář pro práci s klientem. Jako ideální řešení pro všechny možné situace zadané v požadavcích (tedy i tuto) se ukázalo kompletní odstranění všeho kódu souvisejícího s modálním oknem a formulářem včetně samotných tlačítek. Tento kód byl vždy vyčleněn do speciální komponenty (např. `ModalClients`), která na základě toho, zda obdrží klienta, či ne, pak zobrazí tlačítko pro přidání, resp. úpravu klienta, které pak otevře formulář pro přidání, resp. úpravu klienta. Na stránce s klienty tedy pouze vložíme komponentu `ModalClients`, které předáme několik potřebných parametrů (tzv. „props“) a vše ostatní necháme na ní. Tutéž komponentu pak můžeme využít kdekoli napříč aplikací, stačí ji nainportovat a dát jí potřebné parametry. Všechna logika je obsažena v ní, na jednom centrálním místě. Zmíněný přístup byl použit také pro úpravu skupin v kartě skupiny (původně bylo taktéž možné skupinu upravit jen ze seznamu skupin) a také pro úpravu lekcí jednotlivců i skupin z diáře a přehledu (lekce šly původně upravit pouze v kartě klienta/skupiny).

Dále bylo třeba vyřešit možnost přidávání klientů např. při vytváření zájemce – tedy aby lektorka nemusela přidávání zájemce zrušit, přejít do klientů, vytvořit zde klienta, přejít zpět do zájemců a znovu vytvořit zájemce z již založeného klienta. Implementace tohoto je opět založena na komponentách a logice výše, jen ji bylo třeba rozšířit o možnost získat z otevřeného formuláře uložená data (např. informace o nově přidaném klientovi), a ty poskytnout jiné komponentě s formulářem (např. formuláři pro zájemce o kurz) – zde je totiž seznam aktuálních klientů, který my po přidání nového klienta aktualizujeme a rovnou vybereme jako zájemce příslušného nově přidaného klienta. Pro ilustraci je na obrázku 8.6 vidět, jak je tlačítko pro přidání řešeno – lektorka je navedena k tomu, že buď zvolí existujícího klienta, nebo vytvoří nového (pokud ještě klient neexistuje), ten nový se pak, jak již bylo zmíněno, také po přidání automaticky vybere jako zvolený klient ze seznamu již existujících klientů. Totéž řešení je implementováno pro tvorbu skupiny – tedy lektorka může v klidu vytvářet skupinu a do ní vkládat klienty, které

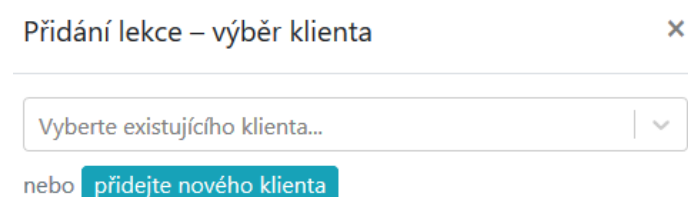
si jednoduše během práce v tomto formuláři se skupinou vytvoří a nemusí jej nikdy opustit.

Obrázek 8.6: Možnost jednoduchého přidání nového klienta z jiného formuláře

Nejtěžším krokem bylo zefektivnění práce s přidáváním lekcí. Je totiž třeba umožnit v diáři i přehledu přidat lekci buď pro jednotlivce nebo pro skupinu. Dále je třeba umožnit rychlé přidání lekce pro daný zobrazený den, ale také pro úplně jiný den, bez jakéhokoliv přecházení na zobrazení tohoto dne (na obrázku 8.8 je toto vidět – je otevřená nabídka pro přidání lekce bez určení data, ale vpravo lze vidět možnost přidání lekce pro konkrétní datum). K řešení těchto mnoha cest, kterými se lektorka může chtít vydat, byla vytvořena jednoduchá komponenta `ModalLecturesWizard` (s poměrně složitější implementací), která tvoří prostředníka mezi jakoukoliv stránkou (diář/přehled) a zmíněnými komponentami jako `ModalClients` či `ModalGroups`, které mají na starost práci s formulářem kdekoli v aplikaci. Tato komponenta může obdržet informaci o dni, pro který se lekce vkládá (když lektorka chce vložit lekci v příslušném dni), ale nemusí (když lektorka chce vložit lekci v jiném dni, který si vybere až ve formuláři pro lekci). Dále, po kliknutí na tlačítko „+“, které tato komponenta vykresluje, si lektorka zvolí, zda chce přidat lekci pro skupinu nebo jednotlivce (viz obrázek 8.7). Poté si (viz obrázek 8.8) buď z rozbalovací nabídky zvolí existujícího klienta/skupinu, pro kterou přidává lekci, nebo přidá nového klienta/skupinu (jednoduše bez jakékoliv práce navíc díky napojení na komponenty zmíněné výše, např. `ModalClients`), v obou případech, jakmile dojde k výběru, automaticky se otevře formulář pro přidání lekce skupiny/jednotlivce dle situace s předvyplněným údajem o datu (pokud byl předán komponentě) a mj. také předvyplněnými dalšími informacemi dle požadavku **F13** (zde je třeba ošetřit případ, abychom např. nepřepsali datum lekce odhadnutým datem následující lekce z implementace tohoto požadavku, protože lektorka již dala jasně najevo, že chce přidat lekci na daný den, nikoliv něco odhadovat). Po uložení lekce dojde k automatickému obnovení původní stránky (diář/přehled) a lektorka vidí všechny změny, aniž by musela za celou dobu někde vůbec přecházet – vše se dělo v rámci jedné stránky.



Obrázek 8.7: Rychlé přidání lekce – krok 1



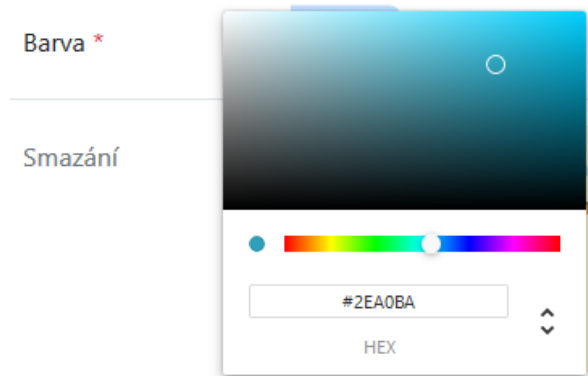
Obrázek 8.8: Rychlé přidání lekce – krok 2

8.1.12 F12 – evidování barev kurzů

Bylo třeba implementovat uživatelsky přívětivý způsob výběru barvy kurzů, není samozřejmě možné pouze připravit pole pro zapsání kódu barvy. Zde bylo využito knihovny `React Color`, která nabízí množství různých komponent pro jednoduchý výběr barvy – konkrétní komponenta již integrovaná do aplikace je vidět na obrázku 8.9. Ve formuláři lektorka jednoduše klikne do pole pro změnu barvy a v této komponentě si barvu vybere. Pro integraci této komponenty bylo třeba vymyslet vhodnou formu otevření a zavření komponenty, v rámci drobného testování použitelnosti na lektorce pak byl finální návrh přijat a nasazen.

Vzhledem k pravidlům použitelnosti a přístupnosti [160] je třeba dbát na to, aby volba konkrétní barvy nezpůsobila nečitelnost jednotlivých komponent v aplikaci – součástí požadavku je totiž nejen samotná volba barvy kurzu, ale také začlenění této barvy do komponent napříč aplikací pro zvýšení přehlednosti. Zde bylo nejprve třeba napříč aplikací tuto barvu vhodně začlenit všude, kde se zmiňuje kurz – tedy do přehledu lekcí a diáře, do rozbalovacích nabídek ve formulářích (díky tomu, že jsou řešeny komponentou `react-select` je toto možné), do karty klienta, k seznamu skupin, jichž je klient členem, do seznamu všech skupin. Na základě začlenění barvy do aplikace bylo jasné, že se vždy bude vyskytovat barva kurzu s bílým textem, tedy bylo třeba vyřešit, jak lektorku upozornit, že zvolená barva není příliš kontrastní – byla vytvořena validační funkce, která díky použití knihovny `chroma.js` umožní zjistit, na kolik jsou dvě barvy vůči sobě kontrastní a toto porovnat vůči vhodně zvo-

lené konstantě (experimenty bylo zjištěno, že je zbytečně restriktivní vycházet z doporučení WCAG a lze jej mírně zrelaxovat pro možnost volby více barev bez zásadního vlivu na čitelnost). Výsledné začlenění barev napříč aplikací je vidět například na obrázku 10.1.



Obrázek 8.9: Komponenta pro výběr barvy kurzu

8.1.13 F13 – automatické předvyplnění údajů lekce

Dle detailní analýzy 6.2.2 byly implementovány funkce s daným algoritmem. Funkce jsou oddělené od implementace samotného formuláře i z důvodu požadavku **F11** pro efektivnější práci v rámci aplikace, kde se počítá s více různými možnostmi přidávání lekcí. Pokud je tedy možné z historie klienta odvodit nové hodnoty pro lekci, při přidávání nové lekce odkudkoliv napříč aplikací se lekce příslušného kurzu (posledního navštíveného) automaticky vytváří o týden později (včetně času) oproti poslední lekci – tento datum a čas i kurz samozřejmě může lektorka před uložením upravit v případě odlišností, ale ve většině případů bude toto ponecháno bez dalších úprav, protože takto kurzy v mnoha případech fungují.

8.1.14 F14 – upozornění na ztrátu dat formulářů

Pro všechny formuláře napříč aplikací bylo třeba implementovat funkcionalitu upozornění na ztrátu nově zadaných dat do formuláře při pokusu o jeho zavření bez uložení. Vzhledem k principu DRY bylo třeba tuto implementaci provést tak, aby byla pouze na jednom místě. K tomu byly využity React Hooks (budu se jim věnovat v sekci 8.3), tedy hook pro modální okno (který poskytuje nejen požadovanou funkcionalitu, ale také sjednocuje práci s modálními okny napříč aplikací) obsahuje indikátor „dirty“, který při první úpravě lektorkou ve formuláři nabude hodnoty „true“, v opačném případě (lektorka změny neprovede) nikoliv. Při zavírání jakoukoliv formou (zavření modálního okna či celého okna) se pak provede kontrola indikátoru a případně je lektorka

formou vyskakovacího okna upozorněna na ztrátu dat při opuštění, zde může buď jít zpět do formuláře, nebo potvrdit, že o ztrátě ví.

Zde je třeba říci, že z hlediska implementace je odlišné, zda lektorka zavírá jen samotné modální okno s formulářem, nebo rovnou celé okno/panel s aplikací. V případě modálního okna s formulářem se pro zobrazení vyskakovacího okna použije kód v ukázce 4 – zde se lektorce jednoduše zobrazí okno s textem v kódu. V případě zavření celé stránky se použije kód v ukázce 5 – zde se přidává naslouchávání na danou událost, kde se při výskytu zavolá daná funkce, ale jak uvádí [169], prohlížeče vyžadují různou podobu této funkce a její chování a také obvykle již nepodporují možnost zobrazit vlastní text zprávy – funkce byla tedy implementována s důrazem na co nejlepší podporu prohlížečů a lektorce se místo textu v ukázce kódu 4 zobrazí generická zpráva příslušného prohlížeče o možné ztrátě dat.

```
window.confirm(  
  "Opravdu chcete zavřít formulář bez uložení změn?"
```

Ukázka kódu 4: Upozornění na neuložené změny při zavření formuláře

```
window.addEventListener("beforeunload", beforeUnload)
```

Ukázka kódu 5: Upozornění na neuložené změny při zavření stránky

8.1.15 F15 – vylepšení chybových hlášení

Požadavek **F15** definuje několik částí, na které je třeba se zaměřit. V první řadě byl zvýšen časový limit zobrazení chybové notifikace na 15 s (původně nebyl definován, tedy aplikovala se výchozí hodnota knihovny React-Toastify 5 s, což nestačilo na pochopení chyby). Všechny chybové hlášky napříč původní aplikací i novými implementovanými požadavky byly revidovány a rozšířeny o více podrobností, možné způsoby řešení – např. při vzniku časového konfliktu mezi lekcemi je součástí notifikace podrobné info o lekci (datum, čas, klient/skupina, doba trvání) a také info, že je třeba upravit datum a čas lekce. Notifikace také překrývaly menu aplikace, což znesnadňovalo přechod na jiné stránky (bylo třeba notifikací ručně zavřít nebo čekat na automatické zavření) – notifikace byly posunuty tak, aby menu nepřekrývaly. Notifikace také díky použití nové komponenty `Notification` mají sjednocený vzhled – obsahují nadpis, ikony symbolizující ne/úspěch a přehledný popis.

Dále bylo třeba se věnovat parsování chyb na klientské části z API. Zde bylo několik problémů – často se stávalo, že nastala na serverové části chyba,

kterou klientská část neuměla rozpoznat a nezobrazila chybu žádnou, lektorka tedy nechápala, proč aplikace „nic nedělá“ (chyba se zobrazila jen v konzoli prohlížeče). Parsování chyb bylo přepracováno a vylepšeno a v případě neznámé chyby se zobrazí také notifikace, stejně tak se lépe a srozumitelněji chyby parsují a je pokryto větší spektrum chyb. Součástí tohoto bylo třeba opravit související problém, kdy při pokusu o smazání entity, která závisí na jiných a je zakázáno ji smazat pokud nějaké tyto závislosti existují (např. smazání klienta s lekcemi), taktéž nedošlo k zobrazení jakékoliv chyby – zde se ale nehodí obecná chyba a na straně serverové části bylo tedy třeba pomoci kódu v ukázce 6 toto ošetřit na všech příslušných místech (jednalo se o chybu `ProtectedError`) a dát lektorce srozumitelné vysvětlení, proč se nepodařilo danou entitu smazat.

```
def destroy(self, request, *args, **kwargs):
    try:
        result = super().destroy(request, *args, **kwargs)
    except ProtectedError:
        result = super().get_result(
            "Klienta lze smazat jen pokud nemá žádné lekce.")
    return result
```

Ukázka kódu 6: Ošetření chyb při mazání entit

Jedním z dalších problémů, který nastával, byl pád aplikace na klientské části, který mohl nastat. Takový pád způsobí, že celá aplikace přestane fungovat, zmizí a je třeba celou stránku načíst znovu (toto chování je výchozí od Reactu 16 [170], na kterém je tato aplikace postavená). Samozřejmě nejlepší cestou je vše ošetřit, což bylo také učiněno, problémy se ale stále mohou přihodit a je třeba zvolit takové záchranné řešení, aby celá aplikace nespadla a nabídla lektorce možné řešení. React nabízí od verze 16 možnost toto řešit pomocí komponent zvaných „Error Boundaries“ – je to běžná komponenta, která ale obsahuje navíc jednu (nebo obě) metody `static getDerivedStateFromError` a `componentDidCatch`, tyto komponenty odchyťávají všechny chyby ve svých potomcích, logují je a umožňují zobrazit záložní UI [170]. Je mnoho možností, kam a kolik komponent umístit, od zabalení individuálních komponent až po zabalení nejvyšších komponent ve stromu komponent. Zde jsem zvolil přístup zabalení tagu `main` (který obsahuje všechny komponenty v rámci aplikace pod menu) do jedné komponenty zvané přímočaře `ErrorBoundary`. Díky tomu se při výskytu jakékoliv chyby v těchto potomcích místo pádu a zmizení celé aplikace zobrazí informace o chybě a stránka je díky zobrazenému menu stále nadále použitelná. Jak bude uvedeno v podsekcí 8.2.2, do zmíněné metody `componentDidCatch` bude zakomponováno logování do Sentry a možnost zpětné vazby lektorky přiřazené k danému problému.

8.1.16 F16 – titulky stránek

Každá stránka by měla mít svůj unikátní titulek (zobrazený např. v panelu prohlížeče), v aktuální verzi toto řešeno není a napříč aplikací je jen jeden titulek. Unikátní titulky tedy bylo třeba zakomponovat do klientské části, konkrétní titulek závisí na aktuální stránce, přičemž vzhledem ke SPA architektuře aktuální stránku řeší React Router. Byla tedy implementována komponenta **Page**, kterou React Router vykreslí, a ta zároveň zaobaluje samotnou stránku – jen jí navíc přidá titulek, titulky k jednotlivým stránkám byly přidány formou rozšíření struktury, která doposavad držela pouze URL adresy stránek napříč aplikací.

Toto prvotní řešení, které bylo nasazeno, mělo ovšem jeden nedostatek, který už může být z popisu implementace patrný – každá stránka (tedy např. karta klienta) má svůj titulek, ale právě jeden. Tedy různí klienti mají stejný titulek na své kartě, tedy „Karta klienta – ÚPadmin“. Toto ale není žádoucí, protože při otevření více klientů pak není jasné, který je zobrazený. Totéž platí pro karty skupin a zobrazený týden v diáři. Implementované řešení tedy bylo rozšířeno o možnost manipulovat s titulkem i ze samotné stránky, která je potomkem **Page** – a to tak, že na základě informací z předka buď má titulek na starost přímo **Page**, nebo jej řeší až její potomek (např. **Card** – karta klienta) a do titulků vloží např. i jméno klienta, resp. skupiny, resp. pro diář zobrazený týden. Toto je třeba, protože rodičovské komponenty pouze ví, která stránka se bude zobrazovat, neví ale např. jméno klienta (v URL adrese je jeho ID, k němuž si informace získá až potomek **Page**).

Díky tomuto řešení má nyní aplikace např. titulky:

- „Novák Dominik – Karta klienta – ÚPadmin“,
- „Diář (6. 4. – 10. 4.) – ÚPadmin“,
- „Zájemci – ÚPadmin“.

Jak je vidět, informace v titulcích jsou řazeny podle důležitosti, tedy v případě prvním uživatel okamžitě vidí celé jméno klienta. Pokud by informace byly naopak, viděl by všude nejprve název aplikace a zbytek by neviděl vůbec, nebo pouze částečně.

8.1.17 F17 – nastavitelné vlastnosti stavů účasti

Vzhledem ke zvolenému způsobu řešení evidence vlastností stavů účasti v návrhu datového modelu (sekce 7.1) pomocí atributů bylo třeba dodat do modelu **AttendanceState** atributy **default** a **excused**. Pro oba atributy platí, že hodnotu **true** smí mít vždy nejvýše jeden řádek v databázi. Bylo tedy toto třeba explicitně ošetřit a pro lepší použitelnost byla zakomponována také funkce automatického odznačení příslušného atributu u starého stavu účasti při aktualizaci tohoto atributu – tedy pokud někdo označí nějaký stav účasti jako

stav `excused`, původní takto označený stav účasti je automaticky odznačen. Co se týče komunikačního rozhraní, bylo dle návrhu v sekci 7.2 třeba do bodu `attendancesstates/` přidat možnost práce s klíči `default` a `excused`.

Na klientské části bylo třeba zvolit vhodný způsob zobrazení informací o tom, které stavy účasti mají jaké označení. Byl zvolen přístup, který jednoduše umožní příslušný stav účasti také změnit (a využije tedy zmíněného automatického odznačení starého stavu) – v nastavení jsou dvě rozbalovací nabídky, jedna pro stav účasti „omluven“, jedna pro stav účasti s významem „dorazil/dorazí“ (ten je také výchozí), toto je vidět na obrázku 8.10. Lektorka zde může snadno změnit příslušný stav, v případě, že je aplikace nasazena do prázdné databáze, jsou zde také zobrazena upozornění ohledně nenastavených vlastností stavů účasti – protože v případě nenastavení atributů nemůže aplikace korektně fungovat (resp. funguje, ale např. není schopná korektně spočítat pořadové číslo lekce, automaticky lekce rušit).

Obrázek 8.10: Konfigurace vlastností stavů účasti v nastavení aplikace

8.1.18 F18 – omezení a validace hodnot

V rámci všech požadavků a také na základě revize stávající domény, jejích entit a omezení byla vytvořena a sepsána v rámci návrhu na obrázku 7.1 všechna omezení a validace. Na základě tohoto návrhu bylo třeba mnoho z omezení doimplementovat – o některých jsem se již zmínil při implementaci jednotlivých funkčních požadavků. Zde uvedu ještě několik dalších komentářů k implementaci.

Vzhledem k tomu, že počet omezení od původní aplikace poměrně narostl a v čase se i měnil (zde uvádím finální verze návrhu a implementace), bylo třeba, aby byla omezení trasovatelná napříč aplikací a bylo jasné, který kód řeší které omezení – proto bylo zavedeno značení pomocí identifikátorů (např. „O13“). Toto označení je pak součástí dokumentace v kódu a implementace je tak prakticky propojená s návrhem, toto řešení tedy kromě trasovatelnosti umožňuje jistou míru konzistentnosti.

Konkrétní implementaci omezení a validace zde popisovat nebudu, jednak je často zmíněna v textech o implementaci příslušných souvisejících poža-

avků, jednak se obvykle nejedná o složité konstrukty. Pro zajímavost zde uvádím, že bylo původně požadováno omezení pro neaktivní klienty, kterým nebylo možné přiřadit lekci – po delší době existence tohoto omezení nakonec ale vzešel požadavek na zjemnění tohoto omezení tak, že neaktivnímu klientovi sice nelze lekci přiřadit, ale pokud mu bude lekce přiřazována, automaticky se změní na aktivního. Taktéž bylo na základě následného požadavku zrušeno omezení, které zakazovalo přidání neaktivního klienta do skupiny, nyní je to již možné.

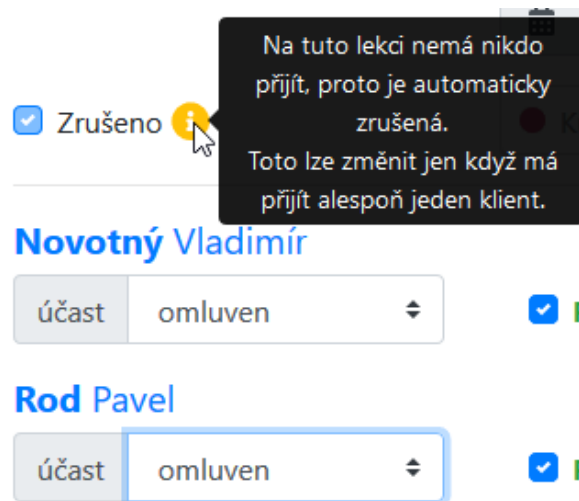
Mnoho z validačních metod bylo přesunuto do zvláštního souboru s názvem `serializers_helpers.py`, aby byl původní soubor `serializers.py` zachován co možná nejjednodušší. Součástí implementace tohoto požadavku bylo také mnoho ošetření napříč celou aplikací, zejména na serverové části, kde bylo zjištěno mnoho případů, kdy při využití holého API (nikoliv prostřednictvím klientské části) by v datech vznikly nekonzistence a při následném zobrazení na klientské části by mohly být z toho důvodu způsobeny různé problémy. Byly tedy opraveny všechny problémy na API tak, aby pracovalo v souladu s očekáváním a nepovolovalo ukládat do aplikace z pohledu omezení nevalidní data. Mnoho ze zjištěných problémů nebylo odhaleno dříve z důvodu, že klientská část API používá určitým způsobem a mnoha omezeními na klientské části se zamezí už samotnému zaslání nevalidních dat na API – kdežto při použití čistého API toto někdy hlídáno nebylo.

Do API byly také začleněny některé další transformace dat umožňující konzistentně uložená data, ale s možností zaslání jiného formátu, který je automaticky převeden na požadovaný formát. Příkladem budiž různé délky hexadecimálního zápisu barev, mezery v rámci telefonních čísel klientů či velká počáteční písmena křestních jmen a příjmení klientů – toto řešila klientská část pro lepší použitelnost již při práci v rámci formulářů, pro konzistentnost dat je ale následná validace a transformace kontrolována především také na API. Dalším opraveným problémem byla nefunkčnost některých PATCH metod, které klientská část nepoužívá (ale mohla by).

8.1.19 F19 – automatické zrušení lekce

Primární místo, kde se bude řešit automatické zrušení lekce, bude serverová část aplikace – zde bylo třeba vytvořit funkci, která zjistí dle počtu omluvených účastníků (fakt, že stav účasti znamená omluven lze díky implementaci požadavku **F17** snadno zjistit) a počtu všech účastníků, zda je třeba lekci zrušit. Funkcionalita byla do funkce oddělena také z toho důvodu, že se zrušení lekce bude řešit nejen při práci s lekcí, ale také při práci se samotnou účastí (**Attendance**) – např. při rychlé úpravě stavu účasti klienta na lekci bez otevření formuláře se pošle PATCH požadavek na `attendances/:id`, tedy API bod lekcí se zpracováním tohoto požadavku nemá nic společného – je třeba kontrolu provádět i zde.

Bylo rozhodnuto, že je vhodné, aby se na klientské části (ve formuláři pro lekce) v případě, že jsou všichni klienti lekce omluveni, dávalo najevo, že lekce bude označena jako zrušená. Toto bylo vyřešeno tak, že se pak automaticky zaškrtně „Zrušeno“, hodnota nelze upravit a vedle ní se zobrazí symbol s vysvětlením, jak je vidět na obrázku 8.11.



Obrázek 8.11: Automatické zrušení lekce, když jsou všichni omluveni

Po nasazení bylo zjištěno, že zde došlo k problému zvoleného řešení tohoto požadavku v souvislosti s požadavkem **F21** na vedení skupinových lekcí bez účastníků, kdy algoritmus označil jako zrušenou lekci bez účastníků, toto bylo opraveno (viz implementace tohoto požadavku v podsekcí 8.1.21).

8.1.20 F20 – zobrazení zrušených lekcí

Vzhledem ke způsobu řešení původního požadavku z bakalářské práce na skrytí zrušených lekcí (bylo řešeno jednoduchým filtrem na API) je snadné toto filtrování zrušit a zobrazit tak zrušené lekce i v přehledu a diáři (tyto stránky právě užívají API bod s filtrováním). Filtrování zrušených lekcí bylo tedy v kódu jednoduchou úpravou jednoho řádku zrušeno a zrušené lekce se nyní zobrazují nejen v kartě klientů, ale také se korektně včetně příslušného zvýraznění barvou a symboly zobrazují i v přehledu a diáři.

8.1.21 F21 – skupinové lekce bez účastníků

V rámci požadavku **F21** je zmíněno, že původní aplikace nezvládá evidovat lekce bez účastníků – serverová část s tímto nepočítá (ale při zaslání lekce bez účastníků neprotestuje), klientská část dokonce při práci se skupinou bez členů a lekcí bez účastníků spadne.

V rámci klientské části bylo třeba opravit chyby způsobující zmíněné pády, dále bylo třeba upravit kód serverové části tak, aby byla na evidování lekci bez účastníků připravena.

Po opravě chyb se objevil problém, kdy vzhledem k implementaci požadavku **F19** na automatické zrušení lekce, když nemá nikdo z účastníků dorazit, byly lekce bez účastníků označovány jako zrušené. Toto bylo způsobeno řešením algoritmu, který byl postaven na procházení klientů a srovnávání počtu omluvených s celkovým počtem klientů – bylo dodáno ošetření, aby lekce bez účastníků nebyly v tomto algoritmu označovány jako zrušené.

8.2 Nefunkční požadavky

V této sekci se zaměřím na implementaci většiny nefunkčních požadavků, některé nefunkční požadavky (**N2**, **N7**, **N8**) budou ale řešeny až v následujících kapitolách.

8.2.1 N1 – dokumentace

Na klientské části (v jazyce TypeScript zavedeném v rámci požadavku **N3** v podsekcí 8.2.2) byla do kódu doplněna dokumentace pro všechny exportované funkce a také komponenty (a jejich „props“ a „state“). V ukázce kódu 7 je vidět ukázka dokumentace „props“ komponenty `PrepaidCounters`.

```
type Props = {  
  /** Pole se členstvími všech klientů. */  
  memberships: Array<MembershipType>  
  /** Funkce, která se zavolá po aktualizaci  
    počtu předplacených lekci. */  
  funcRefreshPrepaidCnt: (  
    id: MembershipType["id"],  
    prepaidCnt: MembershipType["prepaid_cnt"]  
  ) => void  
  /** Skupina je aktivní (true). */  
  isGroupActive: boolean  
}
```

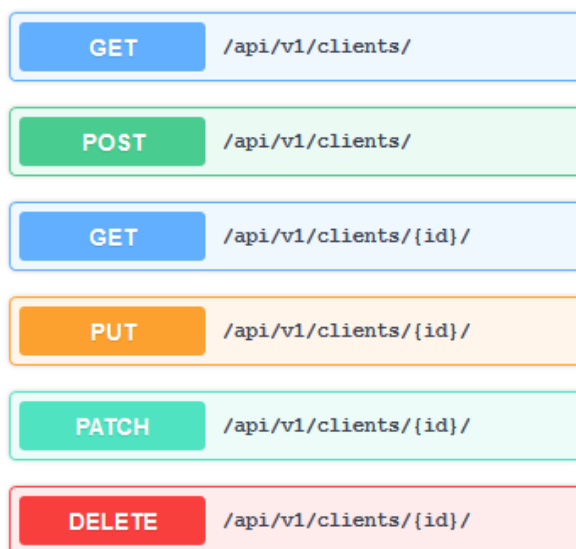
Ukázka kódu 7: Dokumentace v TS (komponenta `PrepaidCounters`)

Na serverové části (Python) byla dokumentace dodána ke všem třídám a metodám, mj. obsahuje i popis konkrétních omezení, pokud tato metoda nějaká implementuje – kvůli trasovatelnosti omezení definovaných v podsekcí 8.1.18. Na ukázce kódu 8 je dokumentace metody pro validování telefonního čísla včetně označení implementovaného omezení.

```
def validate_phone(phone: str) -> str:
    """
    OMEZENÍ 011: Ověří, že je telefonní číslo ve správném
    formátu, jinak vyhodí výjimku.
    """
```

Ukázka kódu 8: Dokumentace v Pythonu (metoda pro validaci telefonního čísla)

Pro dokumentaci API (která v rámci bakalářské práce vůbec nebyla a jediná forma dokumentace byl jednoduchý popis API v tabulkách přímo v textu práce [1]) bylo využito možností Django REST Frameworku – ten nově nabízí podporu pro automatické generování OpenAPI schémat (standard pro jazykově nezávislý popis REST API [171]), toto vygenerované schéma je pak možné pomocí dalších nástrojů jednoduše zobrazit uživateli srozumitelnou interaktivní formou [172]. Díky automatickému generování není třeba nijak aktualizovat dokumentaci API, protože se generuje dynamicky na základě kódu API, pro zobrazení schématu uživateli byl využit nástroj Swagger UI, díky tomu je možné dokumentaci interaktivně procházet a dokonce zasílat na API přímo z UI dotazy (vše je možné pouze po přihlášení). Na obrázku 8.12 je vidět ukázka z této dokumentace, jednotlivé body lze rozkliknout a zobrazit atributy, jejich popisy, zaslat jednoduše na daný bod požadavek apod.



Obrázek 8.12: Ukázka z nasazené API dokumentace (Swagger UI)

8.2.2 N3 – zavedení nástrojů pro usnadnění vývoje a údržby

V této podsekcí budu psát o způsobu zavedení nástrojů pro usnadnění vývoje a údržby. V rámci kapitoly 4 byla provedena rešerše možných nástrojů a následně v kapitole 5 bylo zvoleno, které nástroje budou použity, případně vyzkoušeny. Nejprve se zaměřím na zavedení nástrojů pro monitorování chyb a správu logů. Poté popíši způsob zavedení statického typování do serverové části (Python) i klientské části (JS). V poslední části pak popíši způsob zavedení linterů, formatterů a dalších nástrojů pro statickou analýzu kódu.

8.2.2.1 Monitorování chyb

Pro monitorování chyb bylo třeba zavést nástroj Sentry. Aby bylo možné monitorovat chyby jak ze serverové, tak klientské části, bylo potřeba obě tyto části se Sentry propojit zvlášť. Při výskytu jakékoliv chyby na serverové části se v Sentry objeví podrobné informace včetně kontextu. Totéž při výskytu problému na klientské části, kde navíc byla integrována do komponenty `ErrorBoundary` implementované v podsekcí 8.1.15 možnost zaslání zpětné vazby lektorkou – tedy při výskytu chyby je možné prostřednictvím formuláře napsat slovy, co se dělo, a tyto informace jsou pak přiřazeny k danému problému v Sentry. Používaný formulář nabízí Sentry, bylo pouze třeba prostřednictvím konfigurace celý formulář přeložit do českého jazyka. Sentry je propojené s Heroku a GitHubem a díky pokročilé konfiguraci v rámci projektu je možné u problémů vidět, ve kterém nasazeném prostředí problém nastal, jaká zde byla verze aplikace, s jakým commitem chyba může souviset apod.

Sentry bylo na projektu zavedeno v červenci 2018 a od té doby se ukázalo jako velmi užitečný pomocník. Díky němu (a možnosti zobrazit kontext chyby, logy z konzole prohlížeče apod.) bylo zachyceno a jednoduše reprodukováno mnoho chyb. Formulář pro zaslání zpětné vazby lektorka několikrát využila a i díky němu bylo ještě snadnější chyby reprodukovat.

Byla analyzována možnost využití nástroje Logentries pro zaznamenávání videa s kroky uživatele, toto zavedeno nebylo. Za prvé díky podrobným informacím ze Sentry a Logentries byl za celou dobu používání vždy k dispozici dostatek informací. Za druhé je zde problém s počtem nahraných sezení, kdy se nahrávají všechna sezení, nikoliv pouze ta vedoucí k chybě, tedy snadno se vyčerpá limit 1 000 nahrávek a až dojde k chybě, nahrávka už nebude.

8.2.2.2 Správa logů

Pro správu logů bylo třeba zavést nástroj Logentries. Bylo využito možnosti jednoduchého napojení Logentries díky doplňku na Heroku, logy z Heroku jsou tak dostupné v rámci Logentries, je k dispozici historie a vyhledávání. Využito je též upozorňování na některé události, aplikace nabízí ve výchozím stavu mnoho předdefinovaných upozornění. Logentries je na projektu nasazeno od září 2018 a od té doby bylo několikrát úspěšně využito např. pro zjištění

historie práce lektorky v rámci aplikace (v rámci reprodukování chyby), díky e-mailovým upozorněním byly včas identifikovány problémy ohledně pomalých operací na API (souvisí s optimalizací API v podsekcí 8.2.5) a také byl díky upozorněním odhalen výpadek API Fio banky (kde kvůli nevhodnému ošetření možného výpadku se aplikace nechovala korektně). Prostor pro ukládání logů je dostačující, dle statistik za celou dobu používání bylo využito maximálně 4 MB/měsíc prostoru.

8.2.2.3 Statické typování – serverová část

Napříč celou serverovou částí bylo v Pythonu zavedeno statické typování, v ukázce kódu 9 je vidět konkrétní implementace statického typování na kusu kódu. Pro kontrolu typů byla nejprve využita vestavěná typová kontrola v IDE (Pycharm), později byla zavedena i kontrola pomocí nástroje mypy. Díky oběma těmto nástrojům bylo nalezeno několik potenciálně problematických míst v kódu, které mohly způsobit různé problémy právě v souvislosti s typy, všechny byly opraveny. Oba nástroje nemají stejný výstup, po zavedení mypy se objevilo mnoho chyb, na které IDE neupozornilo, to odpovídá obdobným tvrzením zjištěným v rešerši v podsekcí 4.3.3.

Díky zavedení typů je tak kromě nalezení několika problémů k dispozici kompletně oannotovaná základna, což tvoří výbornou dokumentaci kódu. Pro definice typů Django a Django REST Frameworku jsou využity „stubs“ `django-rest-framework-stubs` a `django-stubs`, po zavedení se ale ukázalo, že definice nejsou úplně kompletní a stoprocentně funkční, bylo tedy třeba na několika místech zavést explicitní ignoraci problémů, protože oba balíčky jsou stále ve vývoji.

```
def process_error(self, status_code: int) ->
    Tuple[Dict[str, str], int]:
```

Ukázka kódu 9: Anotace typů v Pythonu

8.2.2.4 Statické typování – klientská část

Pro klientskou část nebylo v rámci kapitoly 5 zvoleno, zda se vydat cestou Flow či TypeScriptu – volba byla ponechána do praktické části po odzkoušení. Vzhledem k tomu, že Flow nabízí vestavěnou integraci pro React (je také z dílny Facebooku), rozhodl jsem se zavést jej jako první a pouze pro část aplikace a na základě následných zkušeností jej buď zavést napříč celou klientskou částí, nebo vše zmigrovat na TS.

Během zavádění Flow jsem narazil na mnoho problémů, z nichž většina byla oficiálně nahlášena, ale nijak neřešená. Příkladem budiž naprosto ne-

funkční integrace s jakýmkoliv IDE/editorem (typová kontrola buď nefunguje, nebo způsobuje zamrzání celého editoru/IDE) a mnoho dalších problémů, které se objeví v průběhu postupného oánotovávání kódu. Dalším problémem v průběhu anotování se ukázalo málo typových anotací v databázi flow-typed, pokud definice dostupné byly, často byly chybné a způsobovaly mnoho falešně pozitivních hlášení, které nešly nijak vyřešit, mnoho z nich bylo dlouho hlášených. Potvrdila se tedy zjištění z rešerše v podsekci 4.3.2 a vzhledem k tomu, že tedy bylo prakticky nemožné Flow použít kvůli špatné integraci s editory a IDE a mnoha neřešeným problémům, bylo rozhodnuto provést kompletní migraci na TS (jako zmíněné firmy a projekty v rešerši).

Vzhledem k tomu, že je v projektu kvůli použití knihovny react-hot-loader potřeba Babel, bylo dle rešerše v podsekci 4.3.2 přistoupeno k začlenění TS do projektu pomocí Babelu – tedy Babel transpiluje bez typové kontroly kód do čistého JS a typovou kontrolu má na starost kompilátor TS propojený s IDE. Integrace s editorem je velmi svižná a nejsou zde žádné problémy jako s Flow. V rámci zavádění TS a anotací se nevyskytl žádný problém, který by byl neřešitelný a třeba jako v případě Flow i hlášený a neřešený. Díky databázi typů DefinitelyTyped je k dispozici mnohem více definic pro více použitých knihoven v rámci projektu oproti flow-typed a jejich kvalita je vyšší (nedošlo zde k obdobným problémům, jako u Flow – falešně pozitivní chyby s typy a neřešené problémy). Při anotování jsem narazil na drobné problémy s definicí typů vestavěnou v knihovně react-fontawesome, kde nebyl akceptován pro komponentu s ID atribut `id`, ačkoliv v reálu tento atribut použít lze, problém jsem nahlásil a formou PR² opravil a je již začleněn v nové verzi.

Vzhledem k rozsáhlosti kódu (doplnění typů probíhalo až po implementaci všech ostatních požadavků) zabralo kompletní oánotování typy poměrně dlouhou dobu – často muselo být totiž zasahováno do kódu, protože byly nalezeny různé problémy právě související s typy a potenciálními chybami. Bylo také třeba důkladně promýšlet a několikrát měnit způsob řešení definic vlastních typů – při přípravě a odesílání požadavku na API se například struktura daného typu liší od typu, který z API přijmeme metodou GET, toto všechno muselo být řešeno, definováno, a to tak, aby byly typy snadno znovupoužitelné napříč celou aplikací, rozšiřitelné a srozumitelné. Tyto definované typy entit v rámci klientské části jsou uvedeny v souboru `types/models.ts`, řeší všechny možné případy práce s API (pomocí různých metod a příslušných dat v odpovědích).

Postupným zaváděním typů se přicházelo na více a více drobných i větších problémů ve stávajícím kódu, které byly právě díky explicitním typovým anotacím nalezeny a vyřešeny. Výsledkem práce je kompletní pokrytí celé klientské části anotacemi, převedení všech souborů do TS a zavedení všech souvisejících nástrojů pro TS. Typové anotace jsou maximálně vyčerpávající tak, aby sebemenší chyba v kódu znamenala okamžité hlášení problému

²<https://github.com/FortAwesome/react-fontawesome/issues/324>

ze strany TS kompilátoru. Díky zavedení těchto anotací je také nyní velmi snadné psát kód, protože IDE napovídá vzhledem k typovým anotacím. Příklad části kódu v TS včetně anotací je v ukázce kódu 10 – využívá již zmíněné vlastní typy pro běžné entity v aplikaci, konkrétně zde říkáme, že např. parametr `prepaidCnt` odpovídá typu dostupnému pod klíčem `prepaid_cnt` v typu s názvem `MembershipType` (po najetí kurzorem v IDE se hned dozvíme, že to je `number`).

```
funcRefreshPrepaidCnt = (
  id: MembershipType["id"],
  prepaidCnt: MembershipType["prepaid_cnt"]
): void => {...}
```

Ukázka kódu 10: Anotace typů v TS

8.2.2.5 Formattery

Co se týče formatterů, tedy nástrojů pro formátování kódu, na klientské části byl zaveden a nakonfigurován Prettier. Kromě JS, TS a TSX souborů má na starost v tomto projektu také formátování souborů CSS, HTML, JSON, YAML, TOML a MD. Pro Python byl úspěšně zaveden a nakonfigurován Black.

Oba nástroje jsou propojeny s IDE a k formátování kódu tak dochází automaticky během práce. Díky využití formatterů je nyní formátování kódu konzistentní za jakýchkoliv okolností, kód je tak přehlednější a při psaní kódu je možné se zaměřit na samotný význam kódu, nikoliv například konzistentní odsazení.

8.2.2.6 Lintery

Co se týče linterů (v řešerši v podsekcí 4.4.1 bylo zmíněno, že reprezentují prakticky nižší vrstvu nástrojů pro statickou analýzu kódu), byl zaveden a nakonfigurován pro TS nástroj ESLint. Jeho konfigurace v souboru `eslinttrc.js` je poměrně rozsáhlá – kromě konfigurace v souvislosti se samotnými vlastnostmi kódu jej bylo třeba korektně nakonfigurovat pro použití s Reactem, využít pravidla pro kontrolu kódu pro různé oblasti Reactu, dále korektně nakonfigurovat pravidla pro TS, obecná doporučená pravidla ESLint, pravidla, která umožní fungování s formatterem Prettier a mnoho dalších pravidel. V souvislosti se zavedením všech těchto pravidel bylo samozřejmě třeba učinit mnoho oprav a vylepšení v kódu, aby splňoval všechna pravidla.

Díky zavedení ESLint je kód klientské části mnohem srozumitelnější, méně náchylný k chybám, přehlednější, konzistentní a bylo opraveno i mnoho pro-

blémů (např. překrývání oblastí platnosti proměnných kvůli stejnému názvu, problémy s React Hooky, které mohly způsobit chyby na produkci).

Pro CSS byl zaveden a nakonfigurován stylelint, díky jeho zavedení jsou CSS soubory mnohem konzistentnější a přehlednější (např. vlastnosti se řadí dle abecedy) a bylo opraveno také množství různých problémů vycházejících např. z nevhodně definovaných CSS pravidel, duplikace selektorů apod. Pro oba nástroje platí, že mnoho z chyb bylo možné opravit automaticky, čehož bylo také hojně využíváno.

8.2.2.7 Statická analýza kódu

V kapitole 5 o zvolených technologiích bylo řečeno, že nalezené nástroje z vyšší vrstvy pro statickou analýzu kódu budou otestovány a následně budou zavedeny ty, které se osvědčí. Vzhledem ke způsobu práce na projektu se ukázalo jako důležité, aby nástroj uměl analyzovat repozitář i bez provedení PR, a to buď přímo v závislosti na commitech, nebo alespoň třeba jednou denně. Proto zde nejsou uvažovány nástroje **HoundCI** a **Sider**.

Výsledky aplikace nástrojů založených na integraci více linterů (nástroje 1. typu z podsekcce 4.4.1):

- **Codacy**: oproti CodeFactor mnohem pokročilejší rozhraní s více funkcemi, stále ale srozumitelné a přehledné, po aktivaci ESLint se ale ukázalo, že kvůli neaktualizovanému Prettier (na verzi 2) produkuje mnoho falešných chyb (protože v rámci ÚP je již používána tato nová verze),
- **Code Climate**: velmi podobné možnosti jako Codacy, použité nástroje jsou někdy až o mnoho majoritních verzí pozadu, mohou se tedy objevovat nerelevantní chyby,
- **CodeFactor**: srozumitelné jednoduché rozhraní, pro ESLint ale nepodporuje soubor s konfigurací ve formátu „js“ (to je pro projekt potřeba), pro stylelint nepodporuje některé závislosti použité jako doplňky (které jsou opět potřeba) – tedy prakticky neprovádí potřebné kontroly a nelze jej z toho důvodu použít,
- **Code Inspector**: velmi nepřívětivé rozhraní působící dojmem nedodělků, často nepřehledné a s chybami (vše je rozpadlé, nefunguje ani tlačítko zpět), nepodporuje ani korektní parsování kódu, protože prakticky každý řádek kódu označí s chybou, nefungují ani importy – tedy vůbec nefunguje a nelze využít,
- **codebeat**: velmi odlišný nástroj, který nenabízí integraci s linterem, pouze základní kontrolu – nabízí tedy základní upozornění na problémy s kódem jako délka metod, počet argumentů ad., toto zatím v rámci projektu není potřeba.

Jak je vidět z výsledků výše, výsledky nejsou nikterak oslňující, kromě základního nástroje codebeat (který ale není třeba) nelze v praxi především kvůli používání starých verzí nástrojů použít ani jeden z testovaných nástrojů.

Výsledky aplikace nástrojů, které provádějí pokročilou hloubkovou analýzu pomocí vlastních způsobů a pravidel (nástroje 2. typu z podsekcce 4.4.1):

- **DeepSource:** přehledné a jednoduché rozhraní, zjištěné problémy jsou ovšem spíše základnějšího rázu (zvládl by je podchytit i běžný linter, jak ale bylo uvedeno v rámci kapitoly 5, linter pro Python zatím není potřeba),
- **DeepScan:** velmi jednoduché rozhraní, výhodou tohoto nástroje je úzké zaměření na React a tedy možné odhalení problémů, které jiné nástroje nenaleznou, protože se přímo Reactem nezabývají, v reálu bylo nalezeno jedno možné vylepšení kódu (zjednodušení podmínky), nalezených problémů by pravděpodobně mohlo být více, ale nástroj byl zaveden až po opravení chyb ze všech ostatních nástrojů,
- **LGTM:** tento nástroj byl zaveden na projektu jako první a velmi se osvědčil, od nasazení bylo za dobu 10 měsíců vývoje nalezeno necelých 50 problémů a některé z nich byly poměrně pokročilé a mohly v reálné aplikaci skutečně způsobit problémy (např. špatná práce se stavem v rámci React komponent mohla vzhledem ke svému asynchronnímu principu ve vzácných případech způsobit problémy),
- **SonarCloud:** tento nástroj byl na projektu testován jako poslední po všech úpravách hlášených jinými nástroji, přesto v některých souborech odhalil místa pro zlepšení (nikoliv z hlediska přímo problémového kódu, ale zbytečně složitých zápisů, překrývání oblastí platnosti proměnných, duplikace kódu, jiných názvů souboru oproti exportované komponentě ad.), také nahlásil několik míst v kódu, které bylo třeba zkontrolovat z hlediska bezpečnosti (protože sám nemůže určit, zda se jedná o problém nebo nikoliv), připomínky byly relevantní, místa byla zkontrolována a nebylo třeba učinit žádné změny (vše bylo v pořádku), nástroj je velmi komplexní a obsahuje obrovské množství pravidel pro různé jazyky a maximální možnou míru konfigurovatelnosti, výhodou je také možnost přímého napojení IDE (Pycharm) a sledování nalezených problémů v reálném čase při psaní kódu, což ostatní nástroje nenabízejí,
- **DeepCode:** nebyly nalezeny žádné chyby ani varování, nelze tedy vyhodnotit, zda to je kvůli slabým schopnostem nebo dobré kvalitě kódu.

Nástroje z této oblasti se oproti předcházejícím ukázaly velmi užitečné a díky jejich zavedení bylo nalezeno mnoho různých problémů ať z hlediska srozumitelnosti a udržitelnosti kódu, tak případných potenciálních chyb na

produkci. Zejména se zde osvědčily nástroje LGTM a SonarCloud, navzájem velmi odlišné, ale z hlediska počtu a povahy nahlášených problémů oba velmi užitečné. Pro trvalé zavedení v projektu byly tedy zvoleny LGTM a SonarCloud a dále ještě DeepScan díky svému úzkému zaměření na React. Tyto nástroje trvale sledují změny v repozitáři a upozorní na případné problémy.

8.2.3 N4 – revize bezpečnosti

V rámci detailní analýzy tohoto požadavku v podsekcí 6.2.4 bylo vytyčeno celkem 6 oblastí, na které je třeba se zaměřit a vylepšit je. Byly detailně specifikovány problémy, které je třeba opravit. Zde se postupně všem těmto vytyčeným bodům budu věnovat.

8.2.3.1 B1 – aktualizace závislostí

Problém **B1** popisuje zastaralé knihovny, nerespektování alespoň drobných oprav v rámci sémantického verzování závislostí, nalezené zranitelnosti v závislostech. Nejprve se zaměřím na serverovou část – zde byly závislosti v původní aplikaci v souboru `requirements.txt`. Pro pokročilou práci se závislostmi, virtuálními prostředími [173] byl do projektu zaveden nástroj Pipenv a s ním příslušný soubor `Pipfile` obsahující seznam knihoven, na kterých projekt závisí. Všechny verze knihoven jsou zde definovány pomocí `~` a tedy nově umožňují drobné aktualizace, všechny knihovny byly aktualizovány na své poslední verze, do kódu a konfigurace aplikace byly projeveny všechny s tím související potřebné změny.

Na klientské části se pro správu závislostí používá nástroj Yarn, který byl zachován, bylo ale třeba upravit závislosti v souboru `package.json` tak, aby všechny respektovaly alespoň drobné aktualizace (formou `~verze`). Opět bylo třeba všechny knihovny zaktualizovat na poslední verze a příslušné změny projevit do kódu. Pro klientskou i serverovou část platí, že mnoho z knihoven prošlo poměrně velkými změnami a bylo tedy třeba do kódu samotné aplikace více zasahovat. S aktualizací klientské části souvisí přechod na jiný nástroj pro práci s klientskou částí, o tom bude řeč v sekci 8.3.

8.2.3.2 B2 – sjednocení konfigurací

Problém **B2** se sjednocením konfigurací Django aplikace a problém **B3** ohledně deaktivace `DEBUG` módu Django aplikace spolu souvisejí. V původní aplikaci byla produkční konfigurace vytvořena tak, že pouze rozšiřovala konfiguraci lokální (vše z ní importovala). Bylo tedy nejprve potřeba kompletně projít lokální a produkční konfiguraci a kde to bylo možné, přesunout konfiguraci z produkční na lokální verzi. Tím bylo dosaženo toho, že je lokální konfigurace co nejpodobnější té produkční a vyvarujeme se tedy možných chyb, které by se projevy až na produkci. Taktéž bylo upraveno celé pojetí těchto konfigurací tak, že existuje základní konfigurace v souboru `up/settings/base.py`,

která obsahuje společný základ pro všechny konfigurace, další konfigurace (lokální v `local.py` a produkční v `production.py`) pak rozšiřují tuto základní konfiguraci. Toto umožňuje ještě přehlednější konfiguraci a taktéž umožňuje například jednoduché dodání CSP v jednom z následujících problémů **B4** (jinak by to bylo značně komplikované a bylo by třeba duplikovat kód).

Aby bylo možné na lokálním stroji testovat přímo produkční konfiguraci (tedy např. běh bez `webpack-dev-server` pouze se statickými soubory), byl v rámci konfigurace aplikace (prostřednictvím proměnné prostředí) zaveden režim manuální produkce (`MANUAL_PRODUCTION`), kde lze spustit aplikaci v prakticky totožném prostředí a produkční konfiguraci, jako po nasazení (až na několik drobných výjimek), díky tomu lze ještě přesněji testovat fungování před nasazením.

8.2.3.3 B3 – deaktivace DEBUG módu

V původní verzi aplikace, jak bylo zmíněno v rámci problému **B3**, byl aktivovaný režim `DEBUG`, jak bylo v rámci analýzy zmíněno, jedná se jak o bezpečnostní, tak výkonnostní problém. V rámci inspekce bylo zjištěno, že po deaktivování tohoto režimu přestane aplikace na Heroku fungovat, tento problém se do vydání verze pro bakalářskou práci nepodařilo vyřešit. Nyní už byl ale problém zjištěn – na Heroku se používalo nastavení proměnné pomocí `export`, což ale Heroku nepodporovalo a tedy k nastavení proměnné nedošlo, na tuto proměnnou ale spoléhal následující příkaz `collectstatic`, který sice i tak posbíral všechny statické soubory, ale nikoliv dle produkční, ale dle lokální konfigurace, samotná nasazená aplikace pak běžela v produkční konfiguraci (to bylo v kódu napevno) a soubory nenašla tam, kde měly být. Dotyčná proměnná byla tedy nastavena přímo v Heroku, jak je požadováno, a následně již vše fungovalo v pořádku a `DEBUG` režim se používá jen na lokálním stroji.

8.2.3.4 B4 – HTTP hlavičky

Problém **B4** podrobně popisuje, které konkrétní HTTP hlavičky v aplikaci v rámci bezpečnosti scházejí – Referrer Policy, CSP a HSTS. Hlavička s názvem `Referrer-Policy` slouží k omezení hodnoty hlavičky `Referer`, která obsahuje informace o předchozí stránce [174]. Je například nežádoucí, aby se mohly cizí stránky dozvědět, kolik klientů máme v aplikaci (díky ID klientů v URL), naproti tomu je ale žádoucí, aby byla někdy známá alespoň doména (např. kvůli povolení zasílání chyb do Sentry v závislosti na jejich původu). Z toho důvodu je v rámci Django konfigurace nastavena tato hlavička na hodnotu `strict-origin-when-cross-origin`, která umožňuje napříč naší aplikací zasílat úplnou adresu (hodí se např. pro Google Analytics, viz zavedení v sekci 8.3), mimo doménu se zašle pouze název původní domény a pokud cílová doména nemá HTTPS, nepošle se nic (proto `strict`).

Související problematikou jsou pak odkazy napříč aplikací, které se otevírají v novém panelu (`target="_blank"`) – zde se může vyskytnout útok na původní stránku, odkud došlo k otevření [175], proto byl do takovýchto odkazů dodán atribut `rel="noopener noreferrer"` – dokonce v některých případech (bankovníctví Fio) se bez tohoto atributu nedá v nově otevřeném panelu zobrazit přihlašovací formulář, stránka místo toho zahlásí, že došlo k úspěšnému odhlášení.

Hlavička **Strict-Transport-Security** umožňuje, aby prohlížeč komunikoval se serverem už od počátku přes HTTPS, jinak jsou první komunikace jen v HTTP, což umožňuje útok typu „man-in-the-middle“ [176]. Díky možnosti registrace stránky do autoritativní databáze³, která je distribuována do jednotlivých prohlížečů, lze i při úplně první komunikaci s webem (a pak i dalších prvních komunikacích) fungovat přímo přes HTTPS [177]. Django konfigurace opět umožňuje nastavit HSTS, konkrétní nastavení je v ukázce kódu 11.

```
SECURE_HSTS_SECONDS = 63072000 # 2 roky
SECURE_HSTS_PRELOAD = True
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
```

Ukázka kódu 11: HSTS konfigurace

Hlavička **Content-Security-Policy** určuje prohlížeči, odkud může které zdroje načítat – díky tomu nabízí prevenci např. vůči XSS (Cross-Site Scripting) útokům [178]. Django nenabízí nativní podporu CSP [178], k tomu byla využita knihovna `django-csp`. Obsah této hlavičky může být poměrně pokročilý a rozsáhlý vzhledem k počtu nabízených možností [178] – proto zde její finální verzi nebudu ani uvádět. Při její tvorbě bylo třeba upravit práci s některými knihovnami a nástroji tak, aby nekládaly JS/CSS kód přímo do stránky, ale zvláště do souborů, dále bylo třeba umožnit aplikaci komunikovat se Sentry, Google Analytics, Google Fonts a dalšími servery a umožnit správné fungování všech částí aplikace, protože v původním stavu by aplikace nefungovala vůbec. Na základě dalších doporučení např. z [179] či přímo nástroje Mozilla Observatory (který na chybějící CSP hlavičku v 6.2.4 upozornil) byla hlavička rozšířena o další hodnoty až do své finální verze, kdy díky všem hlavičkám nástroj Mozilla Observatory hodnotí aplikaci nejvyšším stupněm „A+“ se skóre 115/100.

8.2.3.5 B5 – sjednocení práce s tokeny

V rámci problému **B5** je třeba zavést jednotnou práci s proměnnými prostředím napříč aplikací. K tomu byla použita knihovna `Django-environ`, která

³<https://hstspreload.org/>

umožňuje proměnné prostředí převést na odpovídající typy pro použití v Pythonu (např. řetězec `True` se převede na boolean s hodnotou `True`) a také umožňuje proměnné prostředí naplnit hodnotami ze souboru `.env` [180]. Díky tomu umožňuje jednoduchou konfiguraci aplikací pomocí proměnných prostředí přesně dle požadavků metodiky „The Twelve-Factor App“ [157, 180]. Těchto funkcí bylo využito tak, že na lokálním stroji jsou hodnoty proměnných prostředí uloženy ve zmíněném souboru, při spuštění aplikace jsou načteny do proměnných prostředí a mohou být využívány. Při nasazení se pak již nevyužívá tohoto souboru, ale proměnné prostředí jsou definovány přímo na Heroku/Travisu prostřednictvím GUI/CLI aplikace (Command Line Interface) a jejich rozparsované hodnoty se pak taktéž používají v rámci aplikace. Další výhodou tohoto řešení je možnost nastavit výchozí hodnoty pro různé proměnné prostředí. Celkem bylo takto zavedeno 12 proměnných prostředí, mj. sloužících pro uložení údajů pro přístup do databáze, dalších tokenů a také proměnných indikujících určité stavy aplikace (např. zda je aktivováno bankovníctví). Díky tomuto přístupu také bude jednoduché zprovoznit v rámci požadavku **N7** více různých prostředí pro nasazení a také umožnit jednoduché zprovoznění aplikace u kohokoliv (což je výhodné v souvislosti s plánem vystavení aplikace jako open-source). Ukázka konfigurace proměnných prostředí v rámci knihovny Django-environ (v konfiguraci Django) je vidět v ukázce kódu 12 (proměnné nejsou uvedeny všechny).

```
env = environ.Env(
    DATABASE_URL=str, # url pouzivane DB
    SECRET_KEY=str, # tajny klic pro Django
    FIO_API_KEY=(str, ""), # token pro pristup do Fia
    BANK_ACTIVE=(bool, True), # aktivace propojeni s bankou
    ...
)
```

Ukázka kódu 12: Ukázka konfigurace proměnných prostředí

8.2.3.6 B6 – zákaz indexace roboty

Problém **B6** popisuje požadavek na zakázání indexování všem robotům. Zde je volba mezi několika způsoby – zákazem procházení prostřednictvím souboru `robots.txt` a zákazem indexování pomocí `meta` tagu (případně HTTP hlavičky `X-Robots-Tag`) [181]. Soubor `robots.txt` sice zakáže procházení stránky roboty, stránka ale stále může být zaindexována z jiné stránky, z toho důvodu je zvolena druhá možnost, která nezakáže procházení, ale zakáže indexování ve všech případech, řešení je v ukázce kódu 13 – důležitý je zde především parametr `noindex`, který toto řeší.

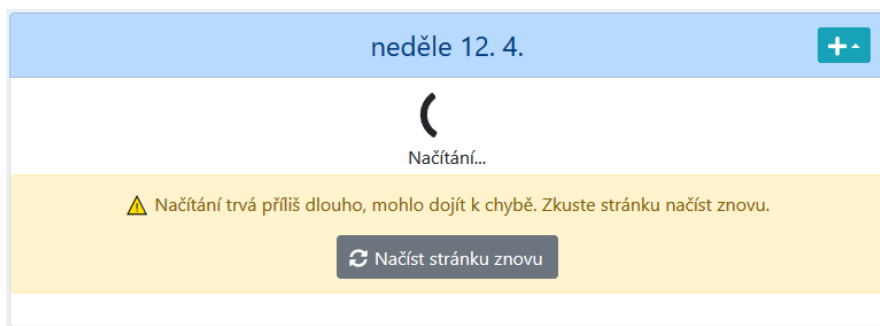
```
<meta name="robots" content="noindex, nofollow"/>
```

Ukázka kódu 13: Kód pro zákaz indexování webu vyhledávači

8.2.4 N5 – vylepšení použitelnosti

V rámci detailní analýzy požadavku **N5** v podsekcí 6.2.5 bylo zjištěno celkem 20 problémů s použitelností. V této sekci popíši některá řešení těchto problémů – především ta zajímavější. Řešení některých problémů zde naopak popisovat nebudu, protože není příliš zajímavé (ale všechny problémy byly vyřešeny) – řešit zde nebudu problémy **P3**, **P4**, **P6**, **P8**, **P9**, **P10**, **P12**, **P14**, **P16**, **P18** a **P20**.

P1: Lektorka nebyla nijak informována o délce načítání. Proto byla rozšířena komponenta **Loading** mající na starost zobrazení informace o načítání – o informování vzhledem k času, tedy po 5 sekundách se lektorce navíc zobrazí informace, že aplikace stále pracuje a načítá, po 25 s (což je dostatečně dlouhá doba na to, aby se případně i probudila aplikace na Heroku, pokud spí v rámci programu zdarma) se zobrazí výstražnější upozornění, že pravděpodobně došlo k nějaké chybě a je třeba aplikaci načíst znovu (včetně tlačítka pro načtení znovu), toto je vidět na obrázku 8.13.

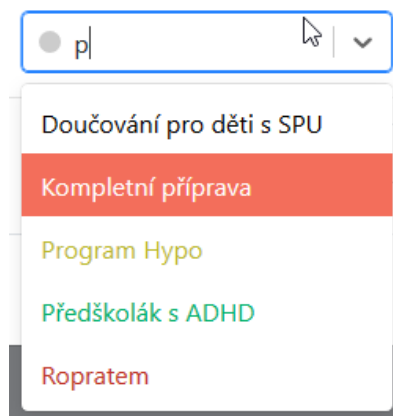


Obrázek 8.13: Upozornění při velmi dlouhém načítání

P2: Chyběly popisy u některých netextových prvků, případně byly vytvořeny formou **title**, kterou nelze zobrazit na dotykových zařízeních. Toto bylo vyřešeno jak dodáním a vylepšením samotných popisů, tak využitím komponenty **Tooltip** z knihovny **Reactstrap**, která umožňuje zobrazit lépe čitelné popisy s možností otevření i na dotykových zařízeních. Zde byl při implementaci objeven v knihovně **Reactstrap** problém, který na iOS způsoboval, že se popis zobrazil vždy až po druhém kliknutí, problém byl nahlášen a poté jsem jej formou PR do tohoto repozitáře⁴ opravil, oprava je již vydána v nové verzi.

⁴<https://github.com/reactstrap/reactstrap/issues/1676>

P5: Některé prvky pro rozbalovací seznamy nebyly řešeny uživatelsky přívětivým `react-select` – kde tedy bylo třeba dle analýzy, byly tyto seznamy nahrazeny `react-select`, na obrázku 8.14 je vidět, jak tyto komponenty mohou práci usnadnit – umožňují např. vyhledávat, dodat barvy ke kurzům (a v případě členů skupiny umožnit výběr více položek, tedy klientů, naráz).



Obrázek 8.14: Komponenta `react-select` pro výběr kurzů

P7: Napříč aplikací nebylo poznat před odesláním formuláře, která pole jsou povinná. Toto bylo napraveno dodáním `*` k popisům jednotlivých polí (jednoduše pomocí CSS), jak je to vidět např. na obrázku 8.6. K tomuto znaku je třeba obvykle pro aplikace používané širším okruhem uživatelů v rámci použitelnosti dodat popis, zde se ale jedná o interní aplikaci pro jednoho konkrétního uživatele srozuměného s významem, tedy zbytečně vysvětlení nezabírá prostor ve formulářích.

P11: V aplikaci se často načítání zobrazovalo jen v závislosti na dokončení jednoho, nikoliv všech požadavků v rámci stránky. Tedy často se stávalo, že se zobrazila neúplná data a poté docházelo k překreslení, lektorka toto ale nemohla vědět, tedy mohly nastávat komplikované situace, kdy např. chyběl obsah rozbalovacího seznamu se stavy účasti. V rámci použitelnosti je třeba, aby bylo vždy jasné, že dochází k načítání. Tento problém je navíc o to vážnější, že v rámci automatizovaného testování v kapitole 9 se objeví tentýž problém – vzhledem k asynchronní práci aplikace nevíme, kdy už můžeme testovat UI, protože stále nemusí být načteno vše potřebné, což může vést k selhání testů. Všechny části aplikace musely být zrevidovány a přepsány tak, aby skutečně až po načtení úplně všech částí na dané stránce načítací animace (prostřednictvím již zmíněné komponenty `Loading`) zmizela.

P13: Pohyb v diáři komplikuje neustálé načítání dní v týdnu, odskakování tlačítek dle délky zobrazeného data a nemožnost ovládat diář klávesnicí. Odsakování tlačítek bylo opraveno v CSS, do diáře byla přidána možnost přesunu mezi jednotlivými týdny pomocí šipek vlevo a vpravo na klávesnici – to

značně zvyšuje pohodlnost používání diáře. Posledním zmíněným problémem je, že při pohybu mezi týdny se okamžitě začínají načítat data jednotlivých dnů, ačkoliv lektorka daný týden jen přeskóčí a jde do jiného týdne – zde bylo dle požadavku implementováno zpoždění požadavku o 700 ms, tedy pokud lektorka na stránce zůstane, po uplynutí tohoto intervalu (s počátkem ve chvíli přechodu na jiný týden) se data lekcí jednotlivých dnů v daném týdnu začnou teprve načítat – ulehčí se tak jak klientské, tak serverové části. Zde je třeba zdůraznit, že bylo myšleno také na zbytečné prodlevy, jelikož se toto čekání děje v komponentě `DashboardDay`, která je pětkrát použita v rámci diáře a jedenkrát v rámci přehledu na hlavní stránce. Bylo tedy třeba vyčlenit případy, kdy se lekce ve dnech načtou bez prodlevy – v přehledu kdykoliv nehledě na situaci, v diáři se prodleva neuplatní při prvním načtení diáře a také při znovunačítání všech dní v týdnu při úpravě v nějakém dni přímo v diáři.

P15: Při otevření rozbalovacích seznamů (`select` i `react-select`) a stisku klávesy ESC dojde jak k zavření seznamu, tak k zavření celého modálního okna. Toto chování je silně závislé na konkrétní implementaci modálních oken v Bootstrapu, zde bylo podrobnou inspekci kódu zjištěno, že se zde používá chování `onKeyUp`, uvolnění ESC tedy modální okno zavře. Toho bylo využito při vytváření obalující komponenty pro `select` z knihovny Reactstrap s názvem `CustomInputWrapper` – po otevření seznamu (`onClick`) si do stavu komponenty uložíme příznak, že je seznam otevřený, `onKeyUp` pak při stisku ESC a příznaku otevřeného seznamu příznak obrátí a zastaví propagaci události `onKeyUp`, aby nedošlo k uzavření modálního okna. Situace pro obalující komponentu pro `react-select` je mírně složitější, protože zde po podrobné inspekci bylo zjištěno, že zavření samotného seznamu je řešeno přes `onKeyDown` a API `react-select` neumožňuje pracovat s `onKeyUp` a tedy zastavit propagaci této události při zavření seznamu. Naproti tomu API umožňuje vytvořit si vlastní součásti celé komponenty `react-select`, čehož bylo využito a byl vytvořen vlastní `Input`, který umožňuje v rámci `onKeyUp` pro klávesu ESC v případě, že skutečně předtím (na základě zjištění z interního stavu komponenty) došlo k zavření seznamu prostřednictvím `onKeyDown`, propagaci `onKeyUp` zastavit (a modální okno se tak nezavře).

P17: Problém s nefunkčním abecedním řazením dat na API při použití diakritiky se ukázal jako poměrně složitý na řešení. PostgreSQL nabízí možnost toto řazení přizpůsobit pomocí proměnné `LC_COLLATE`, jak ale uvádí Heroku dokumentace [182], toto není možné na Heroku upravit – dá se ale na úrovni tabulek řešit jinak, příklad pro řešení je v ukázce kódu 14. Takto byla připravena konkrétní Django migrace s pomocí příkazu `migrations.RunSQL`. Posledním chybějícím dílem do skládačky je konkrétní kód jazyka, zde `cz` – problém je, že na každém operačním systému mají kódy jiné názvy, bylo tedy třeba toto řešit v rámci migrace, jak je uvedeno v ukázce kódu 15 – na základě systému zvolit vhodný kód jazyka. Dále bylo třeba na Travis připravit skript (dostupný v `scripts/postgresql_cs.sh`), který stáhne chybějící balíčky (které jsou jinde dostupné) i zde.

```
ALTER TABLE admin_group
  ALTER COLUMN name TYPE VARCHAR COLLATE cz;
```

Ukázka kódu 14: Řešení řazení podle české abecedy – 1. část

```
if platform.system() == "Windows":
    collation = migrations.RunSQL(
        'CREATE COLLATION cz (locale = "cs-CZ-x-icu")')
else:
    collation = migrations.RunSQL(
        'CREATE COLLATION cz (locale = "cs_CZ.utf8")')
```

Ukázka kódu 15: Řešení řazení podle české abecedy – 2. část

P19: Problém s neobnovováním tokenu v určitých situacích bylo třeba vyřešit přepracováním kódu pro práci s přihlášením. Konkrétně bylo třeba nejprve vytvořit komponentu **AuthChecking**, která provádí dotaz na platnost tokenu napříč aplikací (kde se používá autentizace, tedy ne ve veřejných částech), a to neohledně na to, zda se přechází mezi stránkami nebo nikoliv – jedenkrát za 3,5 h. Při požadavcích na API se taktéž řeší platnost tokenu a pro oba zmíněné případy platí, že pokud má token menší (nebo rovnu) platnost 65 min, obnoví se. Zavedené metody pro kontrolu platnosti jsou součástí objektu kontextu **AuthContext** zavedeného v rámci následujícího požadavku 8.2.5, díky zavedení tohoto kontextu bylo kompletně přepracováno přihlašování na klientské části a vyřešeny i další drobné chyby a problémy s tím související (např. už nelze otevřít přihlašovací stránku když je uživatel už přihlášen, během přihlašování se zobrazuje načítání, je sloučena veškerá logika přihlašování, na mobilu už se nezobrazuje ikonka hamburger menu pro nepřihlášené uživatele).

8.2.5 N6 – optimalizace API

Práci na tomto požadavku lze rozdělit do dvou částí – optimalizace samotného API na serverové části a optimalizace využívání API z klientské části. Jedním problémem je totiž samotná pomalost API v určitých případech, druhým pak fakt, že na klientské části jsou dokola opakovány požadavky, které by bylo možné si na poději zapamatovat a neprovádět je znovu.

Nejprve se zaměřím na serverovou část. Bylo třeba analyzovat, jaké konkrétní požadavky na databázi jsou prováděny v rámci nějakého požadavku na API – zde byla do projektu integrována knihovna Django Debug Toolbar, která toto umožňuje. Vzhledem k tomu, že požadavky ale probíhají asynchronně, bylo třeba dodat do tohoto nástroje doplněk `django-debug-toolbar-request-`

8. IMPLEMENTACE

history, který umožňuje analyzovat i asynchronní požadavky. Na obrázku 8.15 je vidět výstup analýzy požadavku na lekce skupiny.

Dotaz	Časová osa	Čas (ms)	Akce
<code>SELECT ... FROM "admin_group" WHERE "admin_group".id = 1 LIMIT 21</code>		0,97	Sel Expl
<code>SELECT ... FROM "admin_lecture" INNER JOIN "admin_group" ON ("admin_lecture".group_id = "admin_group".id) INNER JOIN "admin_course" ON ("admin_lecture".course_id = "admin_course".id) INNER JOIN "admin_course" T4 ON ("admin_group".course_id = T4.id) WHERE "admin_lecture".group_id = 1 ORDER BY "admin_lecture".start DESC</code>		2,01	Sel Expl
<code>SELECT ... FROM "admin_attendance" INNER JOIN "admin_client" ON ("admin_attendance".client_id = "admin_client".id) WHERE "admin_attendance".lecture_id IN (11, 12, 13) ORDER BY "admin_client".surname ASC, "admin_client".firstname ASC</code>		2,30	Sel Expl
<code>SELECT ... FROM "admin_membership" INNER JOIN "admin_client" ON ("admin_membership".client_id = "admin_client".id) WHERE "admin_membership".group_id IN (1) ORDER BY "admin_client".surname ASC, "admin_client".firstname ASC</code>		17,22	Sel Expl
<code>SELECT ... FROM "admin_membership" WHERE ("admin_membership".group_id = 1 AND "admin_membership".client_id = 3) LIMIT 21</code>		1,01	Sel Expl

6 similar queries. Duplicated 3 times.

Obrázek 8.15: Analýza SQL požadavku prostřednictvím Django Debug Toolbar

Díky těmto analýzám bylo zjištěno, že se zbytečně provádí mnoho požadavků – obvykle se jednalo o dohledávání informací o cizích klíčích, kde namísto toho, aby proběhl JOIN a všechna data byla ihned k dispozici, docházelo k neustálému opakovanému dohledávání též informací o entitách. Pro znázornění uvedu modelový příklad. Nejprve shrnu nastavení aplikace před měřením – v DEBUG režimu Django aplikace na lokálním stroji bylo Django napojeno na vzdálenou databázi na Heroku a analyzováno prostřednictvím Django Debug Toolbar – vzhledem k běhu na lokálním stroji, DEBUG režimu a vzdálené databázi došlo k mnohonásobnému zpomalení aplikace, tedy následující čísla zdaleka nejsou taková, jaká by byla na produkci (to by činilo aplikaci značně nepoužitelnou), důležité ale jsou poměry čísel při srovnání před a po optimalizaci, které samozřejmě zůstávají stejné. Pro určitou poměrně rozsáhlou kartu skupiny se začínalo na 637 SQL dotazech, které zabraly 35,4 s, postupnými úpravami toto bylo sníženo na 113 dotazů a dobu vyřízení 7 s, mimo měření pak ještě došlo k dalším drobnějším úpravám, které by tato čísla ještě snížily a pro určité případy tedy došlo k více než čtyřnásobnému zrychlení.

Hlavními kroky k této optimalizaci na straně Django bylo využití metod `select_related`, `prefetch_related` a `Prefetch` – všechny tyto metody řeší

zavedení určitého způsobu spojování dat pomocí JOIN, které Django ve výchozím stavu neprovádí [183], rozdíl je, že `select_related` provádí JOIN přímo v SQL (ale funguje jen pro jednoduché vztahy – cizí klíče, 1:1), naproti tomu `prefetch_related` (a více konfigurovatelný `Prefetch`) provádí JOIN přímo v Pythonu (a podporuje i složitější vztahy, např. 1:N, M:N). Například dotaz na lekce, který je již optimalizovaný pomocí těchto metod je v ukázce kódu 16. Dále vzhledem k úpravám bylo na některých místech úplně upuštěno od JOIN a zasílá se z API pouze ID, protože informace o daném ID si již pamatuje samotná klientská část. Taktéž bylo identifikováno několik oblastí API, kde docházelo ke zbytečné komunikaci s databází, přestože data již byla dostupná v paměti – toto bylo též opraveno. Django ve výchozím stavu po každém požadavku ukončí spojení s databází, spojení se pak vždy musí znovu vytvářet, což představuje zbytečný režijní náklady navíc [184] – v konfiguraci bylo prostřednictvím parametru `CONN_MAX_AGE` nastaveno znovupoužívání spojení po dobu 10 minut. Další optimalizací pak bylo, jak je uvedeno v sekci 7.2, zavedení více filtrů na API (např. viditelné kurzy apod.).

```

queryset = (
    Lecture.objects.order_by("-start")
    .select_related("group__course", "course")
    .prefetch_related(
        Prefetch("attendances",
            queryset=Attendance.objects
                .select_related("client")),
        Prefetch("group__memberships",
            queryset=Membership.objects
                .select_related("client")),
    ))

```

Ukázka kódu 16: Optimalizace SQL dotazů v Django

V rámci analýzy požadavků také bylo zjištěno, že použitá knihovna pro JWT autentizaci na serverové části (`django-rest-framework-jwt`) nefunguje dle očekávání – jedním z principů JWT má být, jak bylo uvedeno v rámci bakalářské práce [1], že není třeba zatěžovat databázi opakujícími se dotazy, protože všechny potřebné informace jsou součástí objektu tokenu. Knihovna ale při každém přístupu do API stejně kvůli uživateli do databáze přistupovala, tento problém byl hlášen mnoha uživateli [185], ale knihovna už není oficiálně udržovaná [186]. Dle doporučení autora této knihovny byla knihovna nahrazena knihovnou `django-rest-framework-simplejwt`, byly provedeny příslušné změny pro zmigrování, protože knihovny nemají stejný základ. Díky této migraci už se do databáze opakovaně nepřistupuje. Vzhledem k tomu, že tato nová knihovna neposkytuje český překlad, některé notifikace v aplikaci

byly v angličtině, proto jsem do projektu přispěl formou PR⁵ a kompletní český překlad je již součástí hlavní větve projektu.

Na klientské části bylo identifikováno několik možných vylepšení – zrychlení vykreslování, rozdělení kódu aplikace na více částí a snížení počtu požadavků na API díky zavedení React Context API.

```
// lazy nactani pro jednotlivé stránky
...
const Diary = React.lazy(
  () => lazySafe(() => import("./pages/Diary")))
...
const Main: React.FC = () => {
  ...
  return (
    ...
    <React.Suspense fallback={<Loading />}>
      ...
      <PrivateRoute
        path={`\${APP_URLS.diar.url}/:year?/:month?/:day?`}
        component={Diary}
      />
      ...
    </React.Suspense>
  )}
}
```

Ukázka kódu 17: Dělení kódu klientské části v Reactu

V původní verzi aplikace se při otevření aplikace v prohlížeči načtou skripty pro celou aplikaci, to je zbytečné, stačí načíst jen ty potřebné pro zobrazení dotyčné stránky, další skripty pak lze načíst až při přechodu na jinou stránku. Toto chování nově umožňuje React 16.6 pomocí funkce `React.lazy()` a komponenty `Suspense` (ta umožní počkat na lazy načtení importu a mezitím uživateli ukázat zvolenou formu načítání) [187], použití je vidět v ukázce kódu 17. Tato ukázka obsahuje nejdůležitější části kódu z aplikace, tedy použití lazy načtené komponenty `Diary` a zobrazení načítání pomocí `Suspense`. Jak je vidět v kódu, v `React.lazy` se navíc volá ještě funkce `lazySafe`, toto není knihovní funkce, ale přímo mnou implementovaná funkce řešící problém, na který se přišlo až několik měsíců po nasazení lazy načítání. Pro pochopení je důležité vědět, že si prohlížeče ukládají do cache kódy aplikace, včetně souboru `manifest.json`, který označuje jednotlivé části kódu aplikace („chunky“), jejichž jména jsou hashe dle obsahu souboru. Při nasazení nové verze aplikace se

⁵<https://github.com/SimpleJWT/django-rest-framework-simplejwt/pull/188>

může stát, že lektorka má u sebe starou verzi tohoto souboru a v případě, že došlo ke změně názvů souborů (změna hashe), dojde k chybě „Error: Loading chunk <číslo_chunku> failed.“. Toto React prozatím nijak neošetřuje a bylo třeba toto ošetřit tedy přímo ručně – funkce `lazySafe` monitoruje výskyt zmíněné chyby a pokud nastane, pokusí se znovu načíst celou aplikaci (aby se odstranily staré hashe v cache), ale pouze jednou, aby nedošlo k zacyklení. Díky tomuto ošetření už lektorka neskončí na chybové stránce, ale aplikace se sama zotaví.

Dalším důležitým vylepšením je zavedení React Context API pro snížení počtu požadavků na API. Context přišel do Reactu v rámci verze 16.3, která vyšla v době dokončení bakalářské práce, umožňuje sdílet data mezi komponentami bez nutnosti explicitního předávání těchto dat skrze „props“ – tedy dat, která jsou v jistém smyslu globální pro určitý strom komponent [188]. Vzhledem k délce kódu souvisejících i se základnějším použitím Context v rámci aplikace uvedu spíše jen základní popis stavebních kamenů Context API – jsou 3 [188]:

- **React.createContext:** funkce vytvářející objekt kontextu,
- **Context.Provider:** komponenta (součást objektu kontextu) umožňující konzumujícím komponentám v daném stromu komponent přihlásit se k odběru změn v kontextu,
- **Context.Consumer:** komponenta (součást objektu kontextu) přihlašující se ke změnám v kontextu.

V rámci aplikace byl Context zaveden pro mnoho částí aplikace:

- přihlašování/odhlašování (`AuthContext`),
- aktivní klienti (`ClientsActiveContext`),
- viditelné kurzy (`CoursesVisibleContext`),
- aktivní skupiny (`GroupsActiveContext`),
- stavy účasti (`AttendanceStatesContext`).

Součástí objektů těchto kontextů jsou kromě samotných příslušných dat i metody pro aktualizaci dat, zjištění, zda se data zrovna načítají, zda jsou načtená. Díky tomu lze napříč aplikací získat například stavy účasti bez jakéhokoliv požadavku na API – data jsou do kontextu načtena při prvním požadavku na data a při úpravě dat (např. přidání stavu účasti) dojde k aktualizaci dat i v kontextu, tedy data jsou stále aktuální. Díky tomu také během přechodů v aplikaci není třeba na nic čekat a data jsou okamžitě bez načítání zobrazena (některá data, např. neaktivní klienti, v kontextu nejsou, protože je to zbytečné, tedy se stále načítají pokaždé čerstvě). Aplikace je tak svižnější,

lektorka nemusí často čekat, API je méně vytížené a tento jednotný přístup umožňuje také lépe kontrolovat, zda se data načítají či nikoliv (to bylo potřeba v rámci problému s použitelností **P11**). Časté požadavky na API plynuly právě z toho, že se přechází např. mezi stránkami v aplikaci a zde nebyla možnost sdílet již načtené stavy účasti, protože se jedná o jiný strom komponent, tedy byly načítány znovu, znovu a znovu.

Dalším drobným vylepšením z hlediska výkonu klientské části při vykreslování a překreslování aplikace bylo odstranění definice komponent z metod `render`, protože komponenty (zejména ty vnořené) se pak musejí při každém vykreslení překreslit všechny, což je zbytečné.

8.3 Další řešení problémy

V rámci iteračního vývoje byly řešeny i další různé problémy a vylepšení, které nebyly uvedeny v původních požadavcích. Příkladem budiž implementace drobných změn mimo požadavky uvedených v návrhu datového modelu v sekci 7.1 a komunikačního rozhraní v sekci 7.2. Dále byl zaveden nástroj Google Analytics pro analýzu průchodů aplikací díky knihovně `react-ga`. V rámci požadavku **N5** na vylepšení použitelnosti bylo třeba aplikaci při vývoji testovat přímo v mobilním zařízení/tabletu, bylo tedy třeba zprovoznit tuto možnost testování v rámci lokální sítě z jakéhokoli zařízení (to nebylo v původní aplikaci možné a bylo třeba vždy aplikaci nasadit).

V následujících podsekcích se zaměřím na podobnější popis dalších oblastí, ve kterých došlo k významnějším změnám.

8.3.1 Migrace a refaktoring

V rámci migrace na API nového Reactu 16.3 a dalších ještě novějších verzí (která byla plánována již v bakalářské práci [1], také toto souvisí s požadavkem na aktualizaci závislostí **B1**) bylo třeba napříč aplikací provést příslušné změny. Kromě samotného zavedení používání Context API pro optimalizaci práce s API (viz podsekcce 8.2.5) bylo třeba přejít na nový životní cyklus komponent Reactu.

Při této migraci bylo zjištěno, že se na mnoho místech naprosto zbytečně komplikovaly komponenty metodami jako `getDerivedStateFromProps`, které ale nebyly potřeba (nebylo třeba kopírovat „props“ do „state“, stačilo číst přímo „props“), to mohlo také způsobit nečekané problémy a chyby v aplikaci – při vydání Reactu 16.4 dokonce vychází článek [189] upozorňující právě na mnohdy zbytečné používání těchto metod a možný výskyt chyb. Vzhledem k tomu bylo třeba projít celou klientskou část aplikace a vyhledat všechny tyto problémy – byly způsobené neznalostí v této oblasti v době tvorby původní aplikace. Příkladem reálného důsledku této chyby je problém s komponentou zobrazující stav účasti klienta na lekci (formou rozbalovacího seznamu) `SelectAttendanceState` – při změnách architektury Reactu na nástroj `nwb`

(viz podsekce 8.3.3) a především zavedení „lazy“ načítání v rámci aplikace (v podsekcí 8.2.5) se začal projevovat problém se zobrazováním špatného stavu účasti v závislosti na pořadí obdržení odpovědi ze serveru (z API).

Se zmíněnou migrací souvisí i kompletní refaktoring celé klientské i serverové části, zjednodušení a zefektivnění kódu, struktury projektu a souborů, rozdělení do více metod, funkcí, odstranění podobných kódů či nepoužívaných metod (např. díky nástroji *vulture* pro Python). Na klientské části opravy nekonzistentních stavů (množina možných hodnot ukládaných do stavu byla širší, než měla být), rozdělení CSS stylů ke komponentám, rozdělení do více komponent, opravy nevalidního HTML a CSS (díky tomu drobné opravy UI), zavedení minifikace HTML, zjednodušení rozhraní komponent. Zmíněný refaktoring měl velký dopad na serverovou část s API, kde byl kód velmi nepřehledný, dlouhý. Tyto velké změny ale bylo samozřejmě možné provádět až při zavedení všech testů pro zásadní části aplikace (viz následující kapitola 9), aby bylo zajištěno, že tyto změny způsobí co možná nejméně problémů.

V Reactu 16.8 přibyla také možnost používat takzvané „hooks“ – funkce, které umožňují používat mnoho vlastností Reactu bez tříd a také přepoužívat jednoduše stavovou logiku (bez změny hierarchie komponent, což doteď nebylo možné) [190]. Jak též oficiální dokumentace [190] uvádí, není třeba hromadně migrovat všechny třídy na klasické funkcionální komponenty s hooky, ale postupně je v dalších iteracích začleňovat při práci na aplikaci – toho jsem se držel a hooky tak jsou zavedeny pro všechna modální okna (hook `useModal`), také je vytvořen hook pro odchyťování stisku klávesy (`useKeyPress`) a práci s daty formuláře (`useForm`). Díky tomu bylo možné v rámci jednoduchých komponent umožnit používání stavu a také dodat (a znovupoužívat) mnoho logiky ke stavu navíc – v případě modálních oken má totiž hook na starost i zobrazování upozornění na ztrátu dat formuláře či třeba předávání dat rodičovským komponentám po odeslání formuláře. Použití jednoduchého hooku pro přidání stavu do funkcionální komponenty je v ukázce kódu 18 – přidává se stav pro indikaci délky probíhajícího načítání, který se pak prostřednictvím `useEffect` po určité době například změnil na jinou hodnotu.

8.3.2 Další opravy chyb a vylepšení

Kromě všech již zmíněných chyb bylo nalezeno množství dalších, pro zajímavost jich několik uvedu v tomto odstavci. V diáři se špatně pracovalo s datem a pokud se do URL zadal datum s 31. dnem v měsíci, došlo k „přesunu do minulosti“ (číslo dne se změnilo na „1“) – v důsledku toho se v aplikaci nedalo dostat do roku 2019, protože 31. 12 2018 bylo pondělí a další týden se přepnul na listopad. Také docházelo k nekorektnímu zobrazení lekci v předchozím dnu – pokud by lekce byla v 1 h ráno (což je sice nereálné, ale např. v rámci testů možné), v diáři se ukázala v předchozím dni jako poslední, dělo se to z důvodu porovnávání data s časovou zónou s datem bez časové zóny (vyřešeno použitím `__date` v querysetu na straně API). Dále, při aktualizaci lekce

```
const Loading: React.FC<Props> = ({ text = "Načítání" }) => {
  const [loadingState, setLoadingState] = React.useState(
    LOADING_STATE.NORMAL_LOADING
  )
  ...
  React.useEffect(() => {
    const timeoutId = setTimeout(
      LOADING_STATE.LONG_LOADING
    )
    return (): void => window.clearTimeout(timeoutId)
  }, [])
  ...
}
```

Ukázka kódu 18: Ukázka použití hooků ze souboru Loading.tsx

v diáři v rámci nějakého dne došlo pouze k aktualizaci lekcí v rámci daného dnu, neprojevil se změny do jiných dnů (např. informace o „příště platit“). V souvislosti s výpočtem platby příště docházelo k nekorektním upozorněním, protože logika výpočtu nebrala v úvahu předplacené lekce.

V rámci iteračního vývoje byla vylepšována také stávající implementace nových požadavků, např. bylo potřeba dodat do zájemců o kurzy telefonní číslo klienta, protože bylo zjištěno, že lektorka často tento údaj při práci se zájemcem potřebuje a musí kvůli tomu do karty klienta. Dále bylo napříč aplikací potřeba zvýraznit neaktivitu skupin a klientů – tedy např. v kartě klienta/skupiny zobrazit upozornění na neaktivitu klientů/skupin/členů skupiny, v seznamu skupin zobrazit upozornění na neaktivního člena apod. – aby lektorka ihned věděla, že ji vzhledem k uvedenému mohou čekat omezení z hlediska funkcí (na základě omezení a validací v sekci 7.1).

8.3.3 Nástroj pro vývoj klientské části

Jak je popsáno v podsekcí 2.2.2, původní verze aplikace z bakalářské práce je založena na nástroji create-react-app 1, vzhledem k tomu, že bylo třeba propojit pro vývoj klientskou část s Djangoem, bylo třeba provést „eject“, tedy prakticky všechny konfigurační soubory ad. budou k dispozici, není ale cesty zpět. Pro rozjezd projektu to bylo důležité, protože pak stačilo konfiguraci upravit ku obrazu svému. Problém ale nastává z hlediska dlouhodobé údržby klientské části, protože při každé aktualizaci Reactu (nebo nástroje create-react-app) je potřeba vše znovu přegenerovat a ručně porovnat konfigurace a na příslušné řádky opět dodat potřebné skripty. To je samozřejmě neudržitelné. Z toho důvodu byly hledány alternativy, byly 2 možnosti. Použít nástroj pro klientskou část, který umožní konfiguraci příslušných částí a zároveň jednoduchou aktualizaci (tedy prakticky nástroj, který zaobalí všechny

potřebné nástroje pro vývoj a jednoduše umožní bez hlubokých znalostí vše potřebné nakonfigurovat). Nebo všechny nástroje a závislosti nakonfigurovat a spravovat po svém. V květnu 2019 byla tedy zvolena první varianta jiného nástroje a k tomu byl nalezen nástroj nwb – který byl do té doby dobře spravovaný, aktualizovaný a populární (poslední aktualizace proběhla v březnu 2019) a nabízel vše potřebné.

Zmíněná poslední aktualizace byla ale poslední doslova, postupem času se začaly kupit problémy týkající se optimalizace klientské části a bezpečnosti (včetně aktuálních požadavků v rámci této práce, např. „lazy“ načítání, CSP), které všechny čekaly na vyřešení různých problémů v tomto nástroji. K řešení ale nedošlo a autor přestal reagovat. V únoru 2020 tak bylo třeba přistoupit na změnu, opět byly 2 možné cesty – zvolit nový nástroj a nebo vše nakonfigurovat ručně.

Jako možný nový nástroj se ukázal nástroj Neutrino. Bylo rozhodnuto, že se provede testovací migrace jak na Neutrino, tak na vlastní konfiguraci všech nástrojů bez jakékoliv nadstavby v podobě abstrakce nad všemi nástroji (jako třeba Neutrino či nwb). Výsledkem byly nakonec tři finální verze ve třech větvích repozitáře – vlastní konfigurace, Neutrino a nejnovější verze nástroje nwb (tento nástroj jen pro referenci a srovnávání). První dva nástroje byly porovnávány jak z hlediska fungování, výstupního kódu klientské části po sestavení (především jeho velikosti) a z hlediska jednoduchosti rozšiřování konfigurace. Postupnými změnami konfigurace (nástrojů Webpack, Babel a mnoha dalších) bylo dosaženo stavu, kdy výstupy sestavené klientské části byly prakticky totožné (i velikostí), tedy nikde nebylo ve vlastní konfiguraci opomenuto nic podstatného oproti nástrojům jako nwb či Neutrino. Abstraktní vrstva nad nástroji, kterou nabízí Neutrino, nabízí výhodu v podobě zdánlivě jednoduché konfigurovatelnosti, to je ale za cenu toho, že je třeba podrobně studovat dokumentaci a složitě nalézat, jakou jednoduchou cestou provést nějakou konfiguraci. Naproti tomu totéž v případě vlastní konfigurace je naprosto jednoduché, protože z dokumentace příslušného nástroje (např. Webpack) přesně víme potřebné kroky pro zprovoznění (pro Neutrino samozřejmě tyto kroky neplatí).

Tedy, pokud vše shrnu, Neutrino nabízí jednoduchou konfiguraci, ale na úkor toho, že je třeba složitě hledat, kde se dá příslušný nástroj konfigurovat. Dále by zde opět existovala závislost na nástroji, který může být pomalu vyvíjený. Rozhodl jsem se tedy, že Neutrino a ani jemu podobné nástroje již z uvedených důvodů zavádět nebudu a provedu migraci na kompletní vlastní řešení v podobě vlastní konfigurace všech nástrojů a správy závislostí. K tomu bylo třeba podrobně nastudovat příslušné dokumentace, aby byla konfigurace v co nejlepším stavu, výhodou je ale naprostá nezávislost na nadstavbových nástrojích a v případě, že dorazí nějaká novinka, takový nástroj nebude tuto novinku blokovat v její implementaci a začlenění do projektu. V březnu 2020 tak mohlo být toto řešení (dokonce s rovnou začleněným TS) vydáno. Díky tomuto řešení pak bylo velmi snadné konfigurovat množství dalších nástrojů

8. IMPLEMENTACE

pro vývoj a údržbu, jako např. ESLint, stylelint, Prettier či Babel doplňky. Taktéž je velmi snadné nakonfigurovat jakýkoliv pokročilý aspekt nástroje Webpack.

Testování

V souladu se zvoleným přístupem k testování v sekci 5.1, zadáním práce a požadavkem **N2** na testování byly implementovány v Pythonu API a UI (E2E) testy. Jak zde bylo též uvedeno, bude k tomu využít nástroj behave založený na BDD – vzhledem k tomu, že v době vytváření testů byla již část aplikace hotová (mírně rozšířená verze oproti bakalářské práci), doslovně zde nelze aplikovat BDD přístup popsáný v podsekci 3.3.2 (protože aplikace je již napsaná), scénáře a testy tedy byly dopsány zpětně, při dalším rozšiřování již mohly být scénáře psány předem.

9.1 Scénáře

Scénáře v prakticky přirozeném jazyce (Gherkin) popisují požadované chování aplikace – v jednotlivých krocích („steps“). Pro tyto kroky jsou pak implementovány samotné testy, tomu se budu věnovat v následující sekci 9.2. Scénáře mohou být psány v mnoha jazycích, pro jednoduchost a srozumitelnost byla zvolena výchozí angličtina. Funkcionality („features“) se pak sestávají z více různých scénářů chování aplikace pro danou funkcionalitu, jeden ze scénářů pro funkcionalitu klientů (ze souboru `clients.feature`) je vidět v ukázce kódu 19. Kromě samotného scénáře je zde vidět použití tagů (pomocí „@“), díky tomu lze pak jednoduše selektivně spouštět např. pouze testy pro mazání či pro klienty. Mnoho ze scénářů je ale mnohem složitějších a používají klíčová slova **Scenario Outline**, díky kterým lze spouštět daný scénář s různými kombinacemi hodnot (v reálu např. různé údaje klientů). Tyto soubory jsou ve složce `tests/features`.

Při psaní scénářů, jak bylo uvedeno v sekci 5.1, byly scénáře vytvářeny na základě reálných požadavků a nejdůležitějších funkcionalit a vlastností aplikace. Výsledkem je 7 funkcionalit (stavy účasti, zájemci o kurzy, klienti, kurzy, skupiny, lekce, přihlášení/odhlášení), sestávajících se z celkem 94 scénářů, scénáře se celkem skládají ze 374 kroků (to ale neznamená, že bude naimplemen-

9. TESTOVÁNÍ

toováno 374 metod pro testování, bude jich méně, protože mnoho z nich využívá opakovaně též kroky např. díky již zmíněnému Scenario Outline).

```
Feature: Operations with clients
  Background: Prepared database and logged user
    Given the database with some clients
    And the logged user

    @delete @clients
  Scenario: Delete client
    When user deletes the client "Rod Lukáš"
    Then the client is deleted
```

Ukázka kódu 19: Ukázka scénáře pro smazání klienta v souboru clients.feature

9.2 Implementace testů

Samotné implementace testů spouští behave na základě scénářů z předchozí sekce, každý krok scénáře přísluší dané implementaci v kódu. Vzhledem k tomu, že se testuje API a UI, tedy dvě vrstvy, bylo třeba tyto testy držet od sebe oddělené – aby jedinou společnou věcí byl fakt, že implementují kroky ze scénářů (souborů *.feature). Bylo zde zavedeno dělení, které behave umožňuje – kroky UI a API jsou v separátních složkách `api_steps` a `ui_steps`. Dále je zavedena složka `common_steps` obsahující společné testovací kroky pro obě části, které zařídí připravenou a naplněnou databázi připravenými daty ze souboru `fixtures.py`.

V ukázce kódu 20 je implementovaný krok ověřující úspěšné smazání klienta, konkrétně pro část API. Testy API, jak bylo uvedeno v sekci 5.1 jsou méně křehké a více stabilní oproti UI, jsou také mnohem jednodušší, protože jsou pouze založené na modelu požadavek a odpověď, kdežto testování UI je mnohem komplexnější.

Jedním z cílů bylo, aby i přes větší křehkost UI testů byl kladen důraz na její co možná největší snížení. Pro představu, pokud bychom ve stávající klient-ské části přistupovali přesně ke každému elementu a jeho obsahu bez jakéhokoliv přemýšlení, můžeme díky zápisu XPath (který Selenium používá) získat kód velmi závislý na konkrétním UI kvůli explicitně uvedené cestě k elementu. Pokud místo toho budeme v rámci aplikace při testování přistupovat k obsahu elementu pomocí jiného mechanismu nezávislého na konkrétní implementaci, stabilita testů by byla mnohem vyšší. Tento mechanismus je známý a používá se v rámci Selenium komunity [191], využívá vlastního HTML atributu s názvem obvykle `data-qa`, který obsahuje příslušný název obsahu, implementovaný test pak nepřistupuje k elementu v závislosti na aktuální podobě


```
@then("the client is deleted")
def step_impl(context):
    assert context.resp.status_code ==
        status.HTTP_204_NO_CONTENT
    assert not helpers.find_client_with_full_name(
        context.api_client, context.full_name)
    assert clients_cnt(context.api_client) <
        context.old_clients_cnt
```

Ukázka kódu 20: Implementace kroku ověřujícího, že je klient smazaný – ze souboru `api/clients.py`

UI, ale pouze pomocí selektoru najde ve stránce příslušný element s názvem v tomto atributu, viz ukázka kódu 21. Pokud tedy nedojde ke změnám v oblasti logiky aplikace, ale pouze se určitý element přesune ve struktuře HTML do jiného uzlu, pokud mu ponecháme tento atribut, testy nadále fungují a není třeba je nijak upravovat a jsou tak mnohem stabilnější. Pro některé knihovny UI (např. `react-select`) dodání vlastního atributu není možné, proto používají atributy `id` nebo `class`.

```
surname_field = context.browser.find_element_by_css_selector(
    "[data-qa=client_field_surname]")
```

Ukázka kódu 21: Přístup k elementům v UI nezávislý na konkrétní implementaci UI

Pro integraci `behave` s Django se používá knihovna `behave-django`. Pro Selenium se používá pro integraci s prohlížečem Firefox webdriver `geckodriver`, toto je zvoleno právě na základě primárního prohlížeče používaného lektorkou. Firefox je pak možné používat jak v rychlejší „headless“ módu bez GUI, tak v pomalejším běžném režimu, kdy je možné sledovat průchody UI aplikace při testech.

Při implementaci UI testů se objevilo několik problémů, v první řadě muselo být kompletně upraveno napříč aplikací načítání (součástí problému s použitelností **P11**), protože skripty potřebují vědět, kdy se mohou začít rozhlížet po stránce a v ní obsažených datech. Toto bylo napříč aplikací opraveno a díky testům tak víme, že je lektorka informována o načítání skutečně vždy (v testovaných případech) korektně. Dále během vývoje v iteracích nastalo několik situací, kdy se testy ukázaly slabé. Například při pokusu o úpravu stávajících dat nekontrolovaly, zda jsou stávající data ve formuláři zobrazena korektně (což umožnilo vznik chyby v souvislosti se špatným zobrazením doby trvání

kurzu), tyto kontroly byly přidány. Dále v rámci úprav formuláře pro lekce byl hlášen problém ohledně tlačítka pro přidání lekce, které nebylo aktivní a lekci tak nešlo uložit, navzdory tomu, že testy procházely (protože používaly pro odeslání formuláře stisk ENTER, reálné chování uživatele je ale obvykle stisk tlačítka pro odeslání), opět došlo jak k opravě aplikace, tak testů, aby simulovaly skutečné reálné chování uživatele. Během testování se náhodně objevovaly nefunkční testy, zde se jednalo o falešně pozitivní hlášení, protože aplikace fungovala korektně – nastal zde ale problém s localStorage, kdy se uložené údaje (JWT token) nestihly pro další scénář smazat, po každém scénáři tak nyní behave automaticky localStorage smaže. UI testy byly také zpočátku poměrně pomalé, podrobnější inspekci bylo zjištěno, že se v určitých krocích zbytečně dlouho čeká, toto bylo opraveno.

Dalším problémem při práci se Seleniem byla dokumentace jeho API pro konkrétní jazyk, která byla v případě Pythonu poměrně nepřehledná, navíc někdy odlišná od API pro Selenium v Javě, kde se vyskytovalo více možností oproti Pythonu.

Pro všechny důležité části aplikace jsou testy naimplementovány, napsané scénáře také slouží jako dokumentace, čehož bylo s úspěchem využito při práci na tvorbě testů v rámci předmětu *MI-PYT*, kde vyučující mohl jednoduše vidět, co je testováno za funkce a jaké je pokrytí. Co se týče procentuálního pokrytí kódu testy, je cca 84 %, toto číslo ale nemá příliš vypovídající hodnotu a důležitější je tak fakt, že jsou pokryty všechny důležité průchody v aplikaci. Paralelně k tomu je zároveň veden seznam zbývajících průchodů (soubor `tests/MANUAL_TESTS.md`), které nejsou pokryty automatizovanými testy.

9.2.1 Další testování

Automatizované testování je možné spouštět jak na lokálním stroji, tak i na CI – tomu se budu věnovat v následující kapitole 10.

Automatizované testy nejsou ale jedinou formou testování v projektu. Samozřejmě je také akceptační testování v rámci iterací při zavádění různých požadavků. Dále je též při větších změnách prováděno manuální testování zbylých průchodů v aplikaci, které nejsou automatizovány, toto se vyplatí, protože se nejedná o průchody nejdůležitější a velké změny se nedějí často (naopak by se nevyplatilo automatizovat úplně vše, viz podsekce 3.2.1).

Nasazení

V této kapitole popíši nový způsob řešení nasazování aplikace do více prostředí (viz požadavek **N7**). Také se zaměřím na další související oblasti, jako například zavedení pravidelných záloh databáze z produkce (viz požadavek **N8**). Konfigurace Travisu je v kořenové složce v souboru `.travis.yml` a kvůli rozsáhlosti ji zde neuvádím.

10.1 Aktualizace prostředí a vylepšení

Před provedením samotných změn týkajících se nasazení do více prostředí bylo třeba provést aktualizace prostředí na integračním serveru a další změny. V původní verzi aplikace byl Travis nakonfigurován pro původní verze Node.js, PostgreSQL a Python – zde bylo potřeba vše zmigrovat na verze nejnovější v souladu s aplikací, pro PostgreSQL bylo třeba učinit pokročilejší konfiguraci, protože Travis zatím nepodporuje jednoduchý výběr nejnovějších verzí PostgreSQL. Aby bylo možné používat novější Python, bylo třeba přejít na nový linuxový obraz systému verze „bionic“.

Pro Pipenv (na který bylo na Travisu také třeba zmigrovat z pip) a Yarn bylo třeba zavést cache, protože jinak trvalo sestavení aplikace na CI velmi dlouho. V rámci zavedení proměnných prostředí konfigurovaných z Django (v rámci požadavku **N4**, konkrétně problému **B5**, k implementaci došlo v podsekcí 8.2.3) bylo toto třeba taktéž přizpůsobit a nakonfigurovat v Travisu. Pro správné fungování českého jazyka při řazení v databázi bylo třeba stáhnout na Travisu příslušné balíčky. Toto i další operace jsou nově ve zvláštních skriptech ve složce `scripts`, rozdělené dle toho, zda se používají jen na Travisu, na Heroku, nebo na obou platformách.

Součástí zmíněných spouštěných skriptů je nově například také skript pro substituci řetězců napříč celou aplikací (tedy klientskou i serverovou částí). Ukázalo se, že je totiž potřeba v aplikaci spravovat několik řetězců na obou těchto částech a mít jednotnou možnost jejich přizpůsobení – bylo by totiž

samozřejmě možné např. dodat token pro Sentry do Pythonu z prostředí, ale jelikož klientská část je sestavená ze statických souborů, které nemají přístup k proměnným prostředí, nemohla by tímto způsobem klientská část k danému tokenu přistoupit. Dalším příkladem jsou informace o verzi aplikace, větvi, commitu ad., které jsou k vidění v nastavení aplikace a používají se také pro další nástroje, aby tyto měly k dispozici informaci o aplikaci (např. Sentry). V ukázce kódu 22 je vidět způsob řešení dvou řetězců, Travis tyto řetězce pak nahradí příslušnými hodnotami z proměnných prostředí (pro soubory klientské i serverové části) – pro zápis těchto proměnných se používá vlastní syntaxe ve formátu `%NAZEV_PROMENNE`.

```
Sentry.init({
    dsn: "%SENTRY_DSN",
    environment: getEnvName(),
    release: "%GIT_COMMIT",
})
```

Ukázka kódu 22: Substituce řetězců na Travisu

10.2 Testování

Na Travisu se spouští mnoho testů. Nejprve je při samotném sestavení klientské části spouštěn např. ESLint, stylelint. Poté se spustí testy pro typovou kontrolu klientské části (TS) a poté serverové části (mypy). Následně se automaticky nejdříve spustí několik základních smoke testů (složka `admin/tests`) a pakliže vše projde, spustí se nejprve automatizované testy API (které jsou rychlejší a tím pádem rychleji dojde k odhalení případné chyby, která by pravděpodobně ovlivnila i následující UI testy). Poté se spustí i UI testy, které trvají déle, ale výsledkem je zaručení, že důležité části aplikace jsou s nejvyšším pravděpodobností funkční. Výsledné pokrytí se reportuje do služby codecov.

10.3 Nasazování na více prostředí

Původní způsob nasazování každé revize na produkční prostředí byl dle návrhu v sekci 7.4 nahrazen nasazením na prostředí „testing“ pro každý commit z jakékoliv větve a na prostředí „staging“ a produkci při vytvoření release na GitHubu.

V sekci 2.3 bylo zmíněno, že ačkoliv se všechny kroky sestavení aplikace odehrají na Travisu, totéž se znovu děje na Heroku, ačkoliv to nemá význam, zpomaluje to nasazování a může vyústit v nefunkční sestavení kvůli mnoha různým důvodům. Z toho důvodu bylo Heroku nakonfigurováno tak, aby pouze

provedlo migrace databáze a následně aplikaci spustilo díky všem dodaným závislostem nahraným z Travisu. Na Travisu bylo nově zařízeno, aby se nenahrávaly na Heroku nepotřebné soubory (např. testy, dokumentace).

Mimo návrh bylo kvůli požadavku na zveřejnění aplikace jako open-source v rámci zadání práce rozhodnuto, že dojde k zavedení ještě jednoho prostředí. Toto prostředí s názvem „demo“ obsahuje demoverzi aplikace, kterou si může kdokoli vyzkoušet – přihlašovací údaje budou veřejně dostupné. Aby se zde nehromadila nesmyslná data v databázi, díky doplňku Heroku Scheduler dojde každý den v noci k automatickému smazání všech data a nahrání vzorových dat, která ukazují možnosti aplikace. Oproti ostatním prostředím zde nejsou povoleny výpisy z banky. Nasazení nové verze aplikace na prostředí oproti ostatním prostředím nesouvisí s přidáním revize či releasu, je založeno na přístupu zmíněném v sekci 2.3, kde je vytvořena příslušná větev demo, do které se vždy pomocí sloučení změn z vývojové větve `master` přidají nové změny. Tento jiný způsob je zvolen kvůli tomu, aby bylo jasné pod kontrolou, která revize je nahraná na tomto prostředí a dostupná pro veřejnost. Tato demoverze totiž slouží například i jako ukázka výsledků této práce pro oponenta, není tedy možné, aby zde kvůli vydání nové verze proběhly nějaké změny, verze musí zůstat zachovaná, dokud nebude ručně nasazena jiná.

Kromě zmíněných změn bylo na Heroku samozřejmě potřeba zavést korektně všechny proměnné prostředí. Dále pro prostředí „testing“ bylo umožněno spouštění nástroje Django Debug Toolbar (ve výchozím stavu toto není možné, ale prostřednictvím proměnných prostředí lze nástroj aktivovat a umožnit tak profilování aplikace a databáze i na vzdáleném prostředí, nikoliv pouze na prostředí lokálním). Toho bylo při optimalizaci práce s databází v rámci požadavku **N6** taktéž využito.

Na produkci bylo zavedeno automatické zálohování databáze, Heroku toto umožňuje jednoduše nastavit [192] – k automatické záloze dojde každý den v nočních hodinách, kdy není aplikace používána a zálohování tedy nezpomalí odezvu. Zálohy jsou ukládány každý den a zpětně je dostupných vždy 7 záloh pro předchozích 7 dnů a 1 týdenní záloha, ty je možné obnovit.

10.4 Ukázka aplikace

V rámci této sekce pro ilustraci uvedu na obrázku 10.1 jednu ukázkou z reálně nasazené aplikace (data jsou fiktivní), konkrétně z diáře – zde je velmi dobře vidět, jak jsou například nově začleněny barvy kurzů do UI aplikace (viz požadavek **F12**), je zde vidět mnoho možností zefektivňujících práci v rámci aplikace (např. rychlé přidání lekce pro daný den, pro jiný den, rychlá úprava lekce, rychlý přechod do karty klienta/skupiny přes jméno, viz požadavek **F11**), vyhledávání v aplikaci (v horní části, viz požadavek **F4**). Další snímky z obrazovky jsou k dispozici ve veřejně dostupném repozitáři s aplikací zveřejněném v rámci následující kapitoly 11.

ÚPadmin Přehled **Diář** Klienti Skupiny Zájemci Nastavení Odhlásit

← Týden 22. 4. – 26. 4. → Dnes +

pondělí 22. 4.	úterý 23. 4.	středa 24. 4.	čtvrtek 25. 4.	pátek 26. 4.
Volno	15:00 Předškolák s ADHD 2 Skupina Program Hypo Kubišová Karolína pro 2. lekci si přijde ve čtvrtek omluven Kašná Helena OK	Volno	15:30 Kurz Slabika 19 Mládek Fraňo omluven 16:30 Rozvoj grafomotoriky 5 Křivánek Václav OK 17:00 Rozvoj grafomotoriky 4 Mana Josef OK 17:30 Rozvoj grafomotoriky 9 Chráková Jana OK 18:00 Rozvoj grafomotoriky 5 Havlíčková Jaroslava Náhrada lekce (12. 4. 2019) OK	13:00 Feuersteinova metoda 18 Skupina Doučování Boček Zdeněk zapláceno do konce kurzu OK Valentová Zuzana zapláceno do konce kurzu OK 14:00 Kurz Slabika 23 Skupina Kurz Slabika 5 Roušar Igor OK Lexa Lukáš OK Novák Petr omluven Chrástová Petra zapláceny 4 lekce OK 15:00 Ropratem 4 Škava Petr OK 16:00 Feuersteinova metoda 27 Sůvová Renata OK 17:00 úvodní konzultace 1 Nosková Ema OK

Obrázek 10.1: Snímek obrazovky s týdenním přehledem

Zveřejnění jako open-source

Jedním z úkolů této práce je její zveřejnění jako open-source. Vzhledem k tomu, že je aplikace vytvořena na míru projektu ÚP, nepočítá se zde, že by snad byla využívána ostatními i v jiných projektech (ačkoliv je toto samozřejmě teoreticky možné). Hlavní pointou zveřejnění je možnost vývojářů nahlédnout na reálnou aplikaci vytvořenou pomocí nejnovější technologií, projít si její konfiguraci, inspirovat se. Taktéž se jedná o mou vlastní referenci.

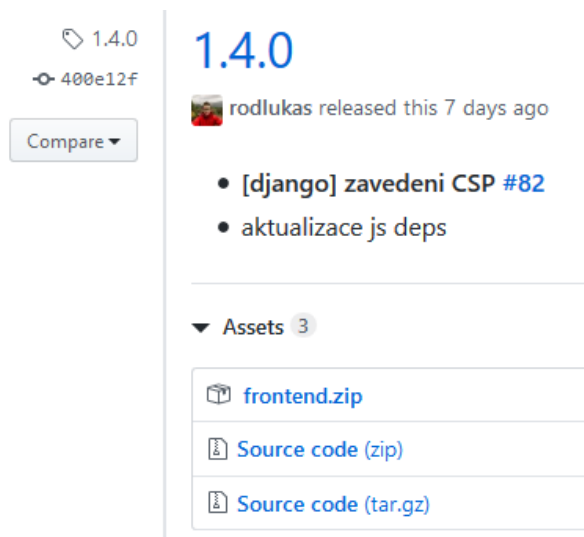
V rámci příprav na zveřejnění bylo třeba zpřehlednit strukturu repozitáře, zkontrolovat, že neuniknou žádné soukromé přístupové údaje (např. v případě již zmíněného problému s tokenem FontAwesome v problému s bezpečností **B5** jej bylo třeba přegenerovat, jinak by byl dostupný v historii repozitáře). Dále bylo třeba zvolit vhodnou licenci, zde byla pro svou jednoduchost zvolena licence MIT. Dále bylo třeba doplnit vzorové hodnoty proměnných prostředí do souboru `.env.template` (zde využívám toho, že je již z podsektce 8.2.3 připravena právě práce s proměnnými prostředí pro konfiguraci celé aplikace).

Hlavním problémem, který souvisí se zveřejněním, je využívání privátních npm registrů pro FontAwesome PRO – tedy běžný uživatel nemá k dispozici příslušný token a nemůže si sestavit klientskou část. Toto bylo vyřešeno propojením Travis CI a GitHub tak, že sestavená klientská část z Travisu (který z proměnných prostředí má k dispozici tokeny) se nahraje při releasu k tzv. „assets“ u daného release, viz obrázek 11.1. Tento zip soubor obsahuje statické soubory klientské části připravené k servírování jak na produkci, tak při lokálním spuštění dle návodu v repozitáři – díky tomu, že je již klientská část takto sestavená, není třeba žádných tokenů.

V repozitáři je **README**, které je ve dvou jazycích – anglickém a českém. Obsahuje informace pro otevření již nasazené demo verze aplikace (viz sekce 10.3) – zde tedy doporučuji čtenáři tuto stránku⁶ navštívit a demo verzi vyzkoušet. Dále obsahuje základní informace o aplikaci, klíčové funkce, použité technologie a nástroje pro serverovou i klientskou část, informace o nasazených apli-

⁶<https://github.com/rodlukas/UP-admin#demo>

kacích a nástrojích a struktuře repozitáře. V neposlední řadě obsahuje také samozřejmě podrobný srozumitelný návod pro spuštění aplikace v lokálním prostředí (včetně minimálních požadavků a vzorových dat pro naplnění databáze), popis práce s testy, licenci a screenshoty z aplikace. V dalších souborech jsou pak dostupné pokročilejší informace o testech či diagramy.



Obrázek 11.1: Sestavená klientská část ke stažení na GitHubu

Repozitář byl úspěšně zveřejněn⁷ a obsahuje všechny zmíněné informace a návody dostupné komukoliv. Spuštění aplikace je možné buď na lokálním stroji, nebo stačí otevřít již zmíněnou demo verzi.

⁷<https://github.com/rod Lukas/UP-admin>

Možná rozšíření

V rámci této práce byly pokryty implementací funkčních požadavků všechny oblasti aplikace a zatím nejsou známy žádné funkční požadavky do budoucna. Z mého pohledu je možné, že se bude zasahovat do oblasti klientů, kde v aplikaci bude v budoucnu přibývat více a více klientů a lektorka možná bude požadovat jednoduchou možnost mazání velmi starých klientů – v aktuální podobě lze díky omezením ze sekce 7.1 klienta smazat pouze pokud nemá žádné lekce, toto omezení by bylo třeba v této souvislosti upravit, zmírnit či obcházet.

Z bakalářské práce zůstaly dvě možné oblasti rozšíření vyřazeny (viz podsekce 6.1.3) – evidence pomůcek a učebnic pravděpodobně nebude s aplikací potřeba slučovat, protože postačuje její současné řešení. Offline přístup a SSR je ale oblast, o kterou lze na základě rešerše a prozkoumání možností aplikaci rozšířit.

Co se týče dalších vylepšení klientské části, je zde plánovaná migrace na `PureComponent` a `React.memo` – to umožní dále optimalizovat výkon klientské části a odstranit další zbytečná překreslování [193] (další optimalizace byly již provedeny v rámci implementace požadavku **N6**, viz podsekce 8.2.5). Dále je plánováno zjednodušení některých rozsáhlých komponent, zejména pomocí „hooků“, byly zmíněny v sekci 8.3, či rozdělení na více komponent. Výhledově bude také možné v aplikaci použít komponentu `Suspense` (která je již v aplikaci použita pro „lazy“ načítání stránek aplikace, viz podsekce 8.2.5) pro práci s API – umožnilo by to výrazné zjednodušení a vylepšení práce s API v rámci klientské části, zatím ale tuto komponentu pro toto načítání použít nelze, tato funkcionality bude až v některé z dalších verzí Reactu [194]. S úpravou načítání také souvisí možnost přidání tlačítka pro obnovení dat v aplikaci bez překreslení celé aplikace (to by totiž udělalo nativní tlačítko v prohlížeči). Dalším možným rozšířením Reactu, které prozatím nebylo možné začlenit kvůli knihovnám, které používají zastaralé API Reactu, je použití komponenty `StrictMode` – ta umožní ve vývojovém režimu upozornit na používání zastaralého API Reactu a také může pomoci při zachycení některých chyb

v kódu související s vedlejšími efekty při překreslování [195].

V plánu je také doplnění dokumentace do kódu pro metody v komponentách klientské části a dále upřesnění typových anotací parametrů na serverové části, kde je např. očekáván obecný slovník, ale je vhodnější zavést striktnější kontrolu obsahu slovníku. Další možnou oblastí je zavedení kontejnerizace (např. Docker), to by mohlo usnadnit např. práci s databází.

Všechna zmíněná možná vylepšení jsou evidována jako „issues“ v GitHub repozitáři⁸ (který byl zveřejněn v rámci kapitoly 11).

⁸<https://github.com/rodlukas/UP-admin/issues>

Závěr

Úspěšně jsem rozšířil webovou aplikaci o všechny požadavky lektorky. Umožňuje tak nově např. evidovat zájemce o kurz či pracovat mnohem jednodušeji s předplacenými lekci. Aplikace je lépe použitelná, díky mnohým vylepšením se dá mnohem efektivněji používat, díky zavedeným kontrolám omezení dbá na korektní a konzistentní data (např. časové konflikty lekcí). Zavedl jsem také další zvolené nástroje pro usnadnění vývoje a údržby, dokumentaci v kódu, automatizované testy API a UI (E2E) a nakonfiguroval jsem nasazování do více prostředí. Díky tomu je možné spolehlivější a rychlejší dodávání nových verzí a na fungování aplikace se dá více spolehnout, mnoho chyb bylo odstraněno a mnohým je díky testům předcházeno. Aplikace je také díky pokročilé optimalizaci mnohem rychlejší a splňuje více bezpečnostních standardů.

Vývoj probíhal v rámci iterací po celé dva roky a tato práce je jakýmsi završením a shrnutím mnoha učiněných kroků (nikoliv samozřejmě všech). Aplikace je v projektu úspěšně každodenně používána od května 2018 a už od svého počátku lektorce výrazně usnadnila práci a ušetřila mnoho času. Díky všem zavedeným změnám aplikace pokrývá více oblastí projektu a lépe řeší mnoho již pokrytých oblastí a umožňuje tak práci zrychlit a usnadnit ještě více.

Taktéž se lze zpětně ohlédnout za samotnou podobou návrhu a implementace původní verze z bakalářské práce. Vzhledem k tomu, jak jednoduché bylo aplikaci rozšířit, lze říci, že použité technologie a návrh byl proveden dobře, původní implementace vyžadovala zásahy např. v podobě refaktoringu, ale z obecného pohledu vzhledem ke snadnému rozšiřování v rámci této práce zde taktéž nebyly větší problémy. Aplikace díky všem provedeným krokům v rámci této práce dospěla a je dobře připravena pro další údržbu a případné rozšiřování.

Aplikace je nyní nasazená na všechny prostředí včetně produkce a lektorka ji spokojeně každý den používá.

Bibliografie

1. ROD, Lukáš. *Webová aplikace pro evidenci klientů projektu „Úspěšný prvňáček“* [online]. 2018 [cit. 2020-03-09]. Dostupné z: <https://dspace.cvut.cz/handle/10467/76850>.
2. EVANS, Dave. *WhiteNoise 5.0.1 documentation* [online]. © 2013–2019 [cit. 2020-03-10]. Dostupné z: <http://whitenoise.evans.io/en/stable/>.
3. NEOTERIC. *Single-page application vs. multiple-page application* [online]. 2016 [cit. 2020-03-10]. Dostupné z: <https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58>.
4. GRIGORYAN, Alex. *The Benefits of Server Side Rendering Over Client Side Rendering* [online]. 2017 [cit. 2020-03-10]. Dostupné z: <https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8>.
5. REACTSTRAP. *reactstrap* [online]. 2020 [cit. 2020-03-10]. Dostupné z: <https://github.com/reactstrap/reactstrap>.
6. FACEBOOK. *Create React App* [online]. 2020 [cit. 2020-03-09]. Dostupné z: <https://github.com/facebook/create-react-app>.
7. JANČA, Marek. *Webpack – moderní Web Development* [online]. 2017 [cit. 2020-03-10]. Dostupné z: <https://www.ackee.cz/blog/moderni-web-development-webpack/>.
8. WEBPACK. *Hot Module Replacement* [online]. 2020 [cit. 2020-03-10]. Dostupné z: <https://webpack.js.org/concepts/hot-module-replacement/>.
9. CODECOV. *Code Coverage Done Right* [online]. 2020 [cit. 2020-03-10]. Dostupné z: <https://codecov.io/>.

10. VAUGHN, Brian. *React v16.3.0: New lifecycles and context API – React Blog* [online]. 2018 [cit. 2020-03-10]. Dostupné z: <https://reactjs.org/blog/2018/03/29/react-v-16-3.html>.
11. ANWERY, Syed. *Manual Testing vs Automation Testing: Which Is Right for My Software Project?* [online]. 2018 [cit. 2020-03-12]. Dostupné z: <https://www.bdo.com/digital/insights/application-development/manual-testing-vs-automation-testing>.
12. KITNER, Radek. *Tápe váš tým jak začít s automatizací testování?* [online]. © 2015 [cit. 2020-03-12]. Dostupné z: https://kitner.cz/testovani_softwaru/tape-vas-tym-jak-zacit-s-automatizaci-testovani/.
13. VOCKE, Ham. *The Practical Test Pyramid* [online]. 2018 [cit. 2020-03-16]. Dostupné z: <https://martinfowler.com/articles/practical-test-pyramid.html>.
14. TESTING GENEZ. *Automated Testing vs Test automation / Benefits, Comparisons & Features* [online]. 2019 [cit. 2020-03-12]. Dostupné z: <https://testinggenez.com/automated-testing-vs-test-automation/>.
15. JEZ HUMBLE, Stewart Hardy. *Continuous Testing* [online]. © 2008-2017 [cit. 2020-03-12]. Dostupné z: <https://continuousdelivery.com/foundations/test-automation/>.
16. KINSBRUNER, Eran. *Manual Testing vs. Automated Testing vs. Continuous Testing* [online]. 2019 [cit. 2020-03-12]. Dostupné z: <https://www.perfecto.io/blog/automated-testing-vs-manual-testing-vs-continuous-testing>.
17. KITNER, Radek. *Benefits of Introducing Automated Testing in Software Development* [online]. © 2015 [cit. 2020-03-12]. Dostupné z: <https://kitner.cz/pokrocily/benefits-of-introducing-automated-testing-in-software-development/>.
18. AMIR. *Common Test Automation Misconceptions* [online]. 2019 [cit. 2020-03-12]. Dostupné z: <https://devqa.io/common-myths-test-automation/>.
19. HLAVA, Tomáš. *Automatizované testování* [online] [cit. 2020-03-12]. Dostupné z: <http://testovanisoftwaru.cz/automatizovane-testovani/>.
20. HLAVA, Tomáš. *Progresní a regresní testy* [online]. 2011 [cit. 2020-03-12]. Dostupné z: <http://testovanisoftwaru.cz/tag/regresni-testy/>.
21. REHKOPF, Max. *What is automated testing?* [online]. © 2020 [cit. 2020-03-12]. Dostupné z: <https://www.atlassian.com/continuous-delivery/software-testing/automated-testing>.

-
22. SMARTBEAR SOFTWARE. *What is Automated Testing?* [online]. © 2020 [cit. 2020-03-12]. Dostupné z: <https://smartbear.com/learn/automated-testing/what-is-automated-testing/>.
 23. FISHMAN, Stephen H. *Testing is Good. Pyramids are Bad. Ice Cream Cones are the Worst* [online]. 2016 [cit. 2020-03-17]. Dostupné z: <https://medium.com/@fistsOfReason/testing-is-good-pyramids-are-bad-ice-cream-cones-are-the-worst-ad94b9b2f05f>.
 24. PITTET, Sten. *The different types of software testing* [online]. © 2020 [cit. 2020-03-12]. Dostupné z: <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>.
 25. COCHRAN, Tim. *Test Pyramid: the key to good automated test strategy* [online]. 2017 [cit. 2020-03-15]. Dostupné z: <https://medium.com/@timothy.cochran/test-pyramid-the-key-to-good-automated-test-strategy-9f3d7e3c02d5>.
 26. FOWLER, Martin. *TestPyramid* [online]. 2012 [cit. 2020-03-15]. Dostupné z: <https://martinfowler.com/bliki/TestPyramid.html>.
 27. BRØTAN, Hallstein. *End-to-end with Selenium: Why you should UI-test* [online]. 2017 [cit. 2020-03-13]. Dostupné z: <https://blog.novanet.no/why-you-should-ui-test/amp/>.
 28. JUNG, June. *How to Test Software, Part II: TDD and BDD* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://dzone.com/articles/how-to-test-software-part-ii-tdd-and-bdd>.
 29. STEVENS, P.J. *Enhance Your Testing Pyramids with BDD* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://cucumber.io/blog/bdd/enhance-your-testing-pyramids-with-bdd/>.
 30. DODDS, Kent C. *Testing JavaScript with Kent C. Dodds* [online] [cit. 2020-03-15]. Dostupné z: <https://testingjavascript.com/>.
 31. ROTH, Mateusz. *Why the test pyramid is a bullsh*t — guide to testing towards modern frontend and backend apps* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://medium.com/@mateuszroth/why-the-test-pyramid-is-a-bullshit-guide-to-testing-towards-modern-frontend-and-backend-apps-4246e89b87bd>.
 32. SOFTWARE TESTING HELP. *How the Testers are Involved in TDD, BDD & ATDD Techniques* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://www.softwaretestinghelp.com/testers-in-tdd-bdd-atdd-techniques/>.
 33. SOFTWARE TESTING HELP. *TDD Vs BDD – Analyze The Differences With Examples* [online]. 2020 [cit. 2020-03-15]. Dostupné z: <https://www.softwaretestinghelp.com/tdd-vs-bdd/>.

34. STATE OF JAVASCRIPT. *The State of JavaScript 2019: Testing* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://2019.stateofjs.com/testing/>.
35. JETBRAINS. *JavaScript 2019 - The state of Developer Ecosystem in 2019 Infographic* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://www.jetbrains.com/lp/devecosystem-2019/javascript/>.
36. SOFTWARE TESTING HELP. *Top 6 BEST Python Testing Frameworks [Updated 2020 List]* [online]. 2020 [cit. 2020-03-15]. Dostupné z: <https://www.softwaretestinghelp.com/python-testing-frameworks/>.
37. STEVENS, P.J. *Understanding the Differences Between BDD & TDD* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://cucumber.io/blog/bdd/bdd-vs-tdd/>.
38. MOHAN, Prashant. *Test Automation & BDD: How They Fit Together* [online]. 2019 [cit. 2020-03-15]. Dostupné z: <https://smartbear.com/blog/test-and-monitor/test-automation-bdd-how-they-fit-together/>.
39. SMARTBEAR SOFTWARE. *Installation - Cucumber Documentation* [online]. © 2019 [cit. 2020-03-15]. Dostupné z: <https://cucumber.io/docs/installation/>.
40. SELENIUMHQ. *The Selenium Browser Automation Project :: Documentation for Selenium* [online]. 2020 [cit. 2020-03-17]. Dostupné z: <https://www.selenium.dev/documentation/en/>.
41. MACDONALD, Andy. *BDD: Writing an Automated Test Suite isn't Rocket Science* [online]. 2018 [cit. 2020-03-15]. Dostupné z: <https://hackernoon.com/bdd-writing-a-test-suite-before-writing-code-6279e4cf4be6>.
42. HIDAYAT, Ariya. *Archiving the project: suspending the development · Issue #15344 · ariya/phantomjs* [online]. 2018 [cit. 2020-03-16]. Dostupné z: <https://github.com/ariya/phantomjs/issues/15344>.
43. RAJKUMAR. *Best Selenium Alternatives in 2020* [online]. 2019 [cit. 2020-03-16]. Dostupné z: <https://www.softwaretestingmaterial.com/best-selenium-alternatives/>.
44. KATALON LLC. *Katalon Studio versus Selenium-based open source frameworks* [online]. © 2019 [cit. 2020-03-16]. Dostupné z: <https://www.katalon.com/resources-center/blog/katalon-studio-vs-selenium-based-open-source-frameworks/>.
45. INDEED. *27 Selenium Interview Questions You Might Encounter During Interviews* [online]. © 2020 [cit. 2020-03-16]. Dostupné z: <https://www.indeed.com/career-advice/interviewing/selenium-interview-questions>.

-
46. COWAN, Paul. *Cypress.io: The Selenium killer* [online]. 2019 [cit. 2020-03-16]. Dostupné z: <https://blog.logrocket.com/cypress-io-the-selenium-killer/>.
 47. RUSTAMZADEH, Amir. *Proposal: Support for Cross Browser Testing* [online]. 2020 [cit. 2020-03-16]. Dostupné z: <https://github.com/cypress-io/cypress/issues/310>.
 48. TAYAR, Gil. *Cypress vs. Selenium WebDriver: Which Is Better for You?* [online]. 2018 [cit. 2020-03-16]. Dostupné z: <https://dzone.com/articles/cypress-vs-selenium-webdriver-which-is-better-for>.
 49. SELENIUMHQ. *SeleniumHQ/docs: Project to rewrite the Selenium documentation.* [online]. 2019 [cit. 2020-03-16]. Dostupné z: <https://github.com/SeleniumHQ/docs>.
 50. STACKSHARE. *StackShare - Software and technology stacks used by top companies* [online]. © 2020 [cit. 2020-03-20]. Dostupné z: <https://stackshare.io/>.
 51. WIGGINS, Adam. *The Twelve-Factor App* [online]. 2017 [cit. 2020-04-29]. Dostupné z: <https://12factor.net/>.
 52. ARSENAULT, Cody. *Error Tracking - Top Suggestions and Tools* [online]. 2018 [cit. 2020-03-19]. Dostupné z: <https://www.keycdn.com/blog/error-tracking>.
 53. HEROKU. *Airbrake Error Monitoring - Add-ons - Heroku Elements* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://elements.heroku.com/addons/airbrake>.
 54. AIRBRAKE. *Plans and Pricing* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://airbrake.io/pricing>.
 55. ORACLE. *Co je software jako služba (SaaS)?* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://www.oracle.com/cz/applications/what-is-saas/>.
 56. STACKSHARE. *What are the best Exception Monitoring Tools?* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://stackshare.io/exception-monitoring>.
 57. SENTRY. *Sentry Pricing* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://sentry.io/pricing/>.
 58. SENTRY. *Platforms* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://sentry.io/platforms/>.
 59. ROLLBAR. *Flexible plans for teams of every stage and size* [online]. © 2012-20 [cit. 2020-03-19]. Dostupné z: <https://rollbar.com/pricing/>.

60. ROLLBAR. *Instrument any application* [online]. © 2012-20 [cit. 2020-03-19]. Dostupné z: <https://rollbar.com/platforms/>.
61. BUGSNAG. *Pricing Plans* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://www.bugsnag.com/pricing>.
62. BUGSNAG. *Platforms* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://www.bugsnag.com/platforms>.
63. HONEYBADGER. *Put Honeybadger On-Call* [online] [cit. 2020-03-19]. Dostupné z: <https://www.honeybadger.io/plans/>.
64. HONEYBADGER. *React Integration Guide* [online]. © 2011-2020 [cit. 2020-03-19]. Dostupné z: <https://docs.honeybadger.io/lib/javascript/integration/react.html>.
65. HONEYBADGER. *Honeybadger for Python* [online]. © 2011-2020 [cit. 2020-03-19]. Dostupné z: <https://docs.honeybadger.io/lib/python.html>.
66. LOGROCKET. *LogRocket / Logging and Session Replay for JavaScript Apps* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://logrocket.com/>.
67. LOGROCKET. *Pricing* [online]. © 2020 [cit. 2020-03-19]. Dostupné z: <https://logrocket.com/pricing/>.
68. LOGROCKET. *React plugin* [online]. 2019 [cit. 2020-03-19]. Dostupné z: <https://docs.logrocket.com/docs/react-plugin>.
69. HEROKU. *Logging* [online]. 2020 [cit. 2020-03-20]. Dostupné z: <https://devcenter.heroku.com/articles/logging>.
70. TRUSTRADIUS. *Log Management Tools* [online]. © 2013-2020 [cit. 2020-03-20]. Dostupné z: <https://www.trustradius.com/log-management>.
71. STACKSHARE. *What are the best Log Management Tools?* [online]. © 2020 [cit. 2020-03-20]. Dostupné z: <https://stackshare.io/log-management>.
72. PAPERTRAIL. *Plans - paid and free log management* [online]. © 2020 [cit. 2020-03-20]. Dostupné z: <https://www.papertrail.com/plans/>.
73. HEROKU. *Papertrail - Add-ons - Heroku Elements* [online]. © 2020 [cit. 2020-03-20]. Dostupné z: <https://elements.heroku.com/addons/papertrail>.
74. LOGENTRIES. *Billing & Pricing* [online]. 2019 [cit. 2020-03-20]. Dostupné z: <https://docs.logentries.com/docs/pricing>.
75. LOGENTRIES. *Pricing Plans* [online]. © 2019 [cit. 2020-03-20]. Dostupné z: <https://logentries.com/pricing/>.

-
76. HEROKU. *Logentries - Add-ons - Heroku Elements* [online]. © 2020 [cit. 2020-03-20]. Dostupné z: <https://elements.heroku.com/addons/logentries>.
 77. GAO, Zheng; BIRD, Christian; BARR, Earl T. [online]. 2017 [cit. 2020-03-25]. Dostupné z: http://ttendency.cs.ucl.ac.uk/projects/type_study/documents/type_study.pdf.
 78. DEVELOPER HOW-TO. *Statically typed Javascript : Why and How* [online]. 2019 [cit. 2020-03-25]. Dostupné z: <https://developerhowto.com/2019/01/05/statically-typed-javascript-why-and-how/>.
 79. ECMAScript DAILY. *Status of Static Typing in ECMAScript* [online]. 2017 [cit. 2020-03-25]. Dostupné z: <https://ecmascript-daily.github.io/pages/status-of-static-typing-in-ecmascript/>.
 80. VOLKMANN, Mark. *Flow – JavaScript Type Checker* [online]. 2017 [cit. 2020-03-25]. Dostupné z: <https://objectcomputing.com/resources/publications/sett/may-2017-flow-javascript-type-checker>.
 81. SCHULZ, Marius. *TypeScript vs. Flow* [online]. 2017 [cit. 2020-03-25]. Dostupné z: <https://mariusschulz.com/blog/typescript-vs-flow>.
 82. WAN, Benny. *Comparing Flow with TypeScript* [online]. 2018 [cit. 2020-03-25]. Dostupné z: <https://medium.com/the-web-tub/comparing-flow-with-typescript-6a8ff7fd4cbb>.
 83. KYLE, Jamie. *Adopting Flow & TypeScript* [online] [cit. 2020-03-25]. Dostupné z: <https://jamie.build/adopting-flow-and-typescript.html>.
 84. TYPESCRIPT. *Angular* [online]. © 2012-2020 [cit. 2020-03-25]. Dostupné z: <https://www.typescriptlang.org/docs/handbook/angular.html>.
 85. TURNBULL, Matt. *TypeScript With Babel: A Beautiful Marriage* [online]. 2019 [cit. 2020-03-25]. Dostupné z: <https://iamturns.com/typescript-babel/>.
 86. NÆSS, Børge. *Migrating the Sanity.io codebase from Flow to TypeScript* [online]. 2019 [cit. 2020-03-25]. Dostupné z: <https://www.sanity.io/blog/from-flow-to-typescript>.
 87. GOLDFINGER, Roger. *Now or Never: Migrating 300k LOC from Flow to TypeScript at Quizlet* [online]. 2019 [cit. 2020-03-25]. Dostupné z: <https://medium.com/tech-quizlet/now-or-never-migrating-300k-loc-from-flow-to-typescript-at-quizlet-d3bae5830a1>.
 88. SKOVHUS, Kenneth. *Migrating from Flow to TypeScript using flow-to-ts* [online]. 2020 [cit. 2020-03-25]. Dostupné z: <https://skovhus.github.io/flow-to-typescript-migration/>.

89. PARDOE, Andrew. *What we're building in 2020* [online]. 2020 [cit. 2020-03-25]. Dostupné z: <https://medium.com/flow-type/what-were-building-in-2020-bcb92f620c75>.
90. GABOR, Bernat. *the state of type hints in Python* [online]. 2018 [cit. 2020-03-27]. Dostupné z: <https://www.bernat.tech/the-state-of-type-hints-in-python/>.
91. MYPY. *Mypy: Optional Static Typing for Python* [online]. 2020 [cit. 2020-03-27]. Dostupné z: <https://github.com/python/mypy>.
92. LEHTOSALO, Jukka. *Our journey to type checking 4 million lines of Python* [online]. 2019 [cit. 2020-03-27]. Dostupné z: <https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python>.
93. HJELLE, Geir Arne. *Python Type Checking (Guide)* [online]. © 2012–2020 [cit. 2020-03-27]. Dostupné z: <https://realpython.com/python-type-checking/#static-type-checking>.
94. GEITGEY, Adam. *How to Use Static Type Checking in Python 3.6* [online]. 2017 [cit. 2020-03-27]. Dostupné z: <https://medium.com/@ageitgey/learn-how-to-use-static-type-checking-in-python-3-6-in-10-minutes-12c86d72677b>.
95. PYTHON. *What's New In Python 3.7* [online]. 2018 [cit. 2020-03-27]. Dostupné z: <https://docs.python.org/3/whatsnew/3.7.html>.
96. MYPY. *Getting started* [online]. 2019 [cit. 2020-03-27]. Dostupné z: https://mypy.readthedocs.io/en/stable/getting_started.html.
97. TYPESHED. *python/typeshed: Collection of library stubs for Python, with static types* [online]. 2020 [cit. 2020-03-27]. Dostupné z: <https://github.com/python/typeshed>.
98. PYDANTIC. *Overview* [online]. 2020 [cit. 2020-03-27]. Dostupné z: <https://pydantic-docs.helpmanual.io/>.
99. SAORIN, Javier Ortiz. *Continuous code quality and automated code review tools* [online]. 2017 [cit. 2020-03-22]. Dostupné z: <https://medium.com/devgurus/continuous-code-quality-and-automated-code-review-tools-aa911dd1b263>.
100. LALITHRAJ, Karthik. *Static vs Dynamic Code Analysis: How to Choose Between Them* [online]. 2019 [cit. 2020-03-22]. Dostupné z: <https://blog.overops.com/static-vs-dynamic-code-analysis-how-to-choose-between-them/>.
101. GURU99. *14 BEST Code Review Tools in 2020 [Static Code Analysis]* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.guru99.com/code-review-tools.html>.

-
102. BARTON, John. *Awesome Code Review* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://github.com/joho/awesome-code-review>.
 103. STACKSHARE. *What are the best Code Review Tools?* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://stackshare.io/code-review>.
 104. SLIKTS. *Automated code review tools missing* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://github.com/joho/awesome-code-review/issues/23>.
 105. GLOBEMA. *Cloud vs On-Premise: Stručná příručka* [online]. 2017 [cit. 2020-03-22]. Dostupné z: <https://www.globema.cz/cloud-vs-premise-strucna-prirucka/>.
 106. CODEBEAT. *How is codebeat different?* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://hub.codebeat.co/docs/how-is-codebeat-different>.
 107. DEEPCAN. *DeepScan vs ESLint* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://deepscan.io/docs/guides/why-deepscan>.
 108. DEEPSOURCE. *Features* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://deepsources.io/features/>.
 109. CODACY. *Pricing* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.codacy.com/pricing>.
 110. CODACY TEAM. *Supported Languages* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://support.codacy.com/hc/en-us/articles/207994735>.
 111. CODACY TEAM. *Post-Commit Hooks* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://support.codacy.com/hc/en-us/articles/214085885-Post-Commit-Hooks>.
 112. CODACY TEAM. *Engines* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://support.codacy.com/hc/en-us/articles/213632009-Engines>.
 113. CODE CLIMATE. *Quality Pricing* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://codeclimate.com/quality/pricing/>.
 114. CODE CLIMATE. *Everything You Need to Ship Better Code, Faster* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://codeclimate.com/quality/>.
 115. CODE CLIMATE. *Adding Code Climate to Your Workflow* [online]. 2019 [cit. 2020-03-22]. Dostupné z: <https://docs.codeclimate.com/docs/workflow>.
 116. CODE CLIMATE. *Configuring Your Analysis* [online]. 2019 [cit. 2020-03-22]. Dostupné z: <https://docs.codeclimate.com/docs/configuring-your-analysis>.

117. CODEFACTOR. *Plans & Pricing* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.codefactor.io/pricing>.
118. CODEFACTOR. *Automated Code Review* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.codefactor.io/>.
119. JONES, Sandra. *Analysis Tools (Open Source)* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://support.codefactor.io/i24-analysis-tools-open-source>.
120. CODE INSPECTOR. *Our Plans* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.code-inspector.com/pricing>.
121. CODE INSPECTOR. *All-in-One Static Code Analysis Tool* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.code-inspector.com/features>.
122. CODE INSPECTOR. *List of Metrics* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://doc.code-inspector.com/docs/metrics/>.
123. CODEBEAT. *Pricing* [online] [cit. 2020-03-22]. Dostupné z: <https://codebeat.co/pricing>.
124. CODEBEAT. *CODEBEAT - Automated code review for mobile and web* [online] [cit. 2020-03-22]. Dostupné z: <https://codebeat.co/>.
125. HOUNDCI. *Automated code review for GitHub pull requests* [online] [cit. 2020-03-22]. Dostupné z: <https://houndci.com>.
126. SCOTT. *General Hound configuration* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <http://help.houndci.com/en/articles/2138473-hound>.
127. SIDER. *Pricing* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://sider.review/pricing>.
128. SIDER. *Supported Analysis Tools* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://sider.review/features/tools>.
129. SIDER. *Product* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://sider.review/features>.
130. DEEPSOURCE. *DeepSource: Find and fix issues during code reviews* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://deepsource.io>.
131. DEEPSOURCE. *DeepSource Documentation* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://deepsource.io/docs/>.
132. DEEPCAN. *Pricing* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://deepscan.io/pricing/>.
133. DEEPCAN. *How to ensure JavaScript code quality* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://deepscan.io>.
134. DEEPCAN. *Using ESLint* [online]. 2020 [cit. 2020-03-23]. Dostupné z: <https://deepscan.io/docs/guides/get-started/using-eslint/>.

-
135. LGTM. *LGTM - Continuous security analysis* [online] [cit. 2020-03-22]. Dostupné z: <https://lgtm.com>.
 136. LGTM. *Analysis FAQs* [online] [cit. 2020-03-22]. Dostupné z: <https://lgtm.com/help/lgtm/analysis-faqs>.
 137. SONARCLOUD. *Clean Code, Rockstar Status* [online]. © 2008-2020 [cit. 2020-03-22]. Dostupné z: <https://sonarcloud.io/>.
 138. SONARCLOUD. *SonarJS* [online]. © 2008-2020 [cit. 2020-03-22]. Dostupné z: <https://www.sonarsource.com/products/codeanalyzers/sonarjs.html>.
 139. SONARCLOUD. *Automatic Analysis Feature* [online]. © 2008-2020 [cit. 2020-03-22]. Dostupné z: <https://sonarcloud.io/documentation/analysis/automatic-analysis/>.
 140. SONARCLOUD. *Code Analyzers* [online]. © 2008-2020 [cit. 2020-03-22]. Dostupné z: <https://www.sonarsource.com/products/codeanalyzers/>.
 141. SONARCLOUD. *Importing External Issues* [online]. © 2008-2020 [cit. 2020-03-22]. Dostupné z: <https://sonarcloud.io/documentation/analysis/external-issues/>.
 142. DEEPCODE. *From Code to Predictions* [online]. © 2020 [cit. 2020-03-22]. Dostupné z: <https://www.deepcode.ai/>.
 143. DEEPCODE. *What is DeepCode?* [online]. 2020 [cit. 2020-03-22]. Dostupné z: <https://deepcode.freshdesk.com/support/solutions/articles/60000346607-what-is-deepcode->.
 144. GOLDBERG, Josh. *Static Analysis in JavaScript: A Technical Introduction* [online]. 2019 [cit. 2020-03-24]. Dostupné z: <https://medium.com/codecademy-engineering/static-analysis-in-javascript-a-technical-introduction-859de5d444a6>.
 145. VANTOL, Alexander. *Python Code Quality: Tools & Best Practices* [online]. © 2012-2020 [cit. 2020-03-24]. Dostupné z: <https://realpython.com/python-code-quality/>.
 146. AWESOME CODE. *format code vs and lint code* [online]. 2019 [cit. 2020-03-24]. Dostupné z: <https://medium.com/@awesomecode/format-code-vs-and-lint-code-95613798dcb3>.
 147. LUOTO, Jesse. *What's the difference between ESLint and Prettier?* [online] [cit. 2020-03-24]. Dostupné z: <https://restishistory.net/blog/whats-the-difference-between-eslint-and-prettier.html>.
 148. REACT. *Rules of Hooks* [online]. © 2020 [cit. 2020-03-24]. Dostupné z: <https://reactjs.org/docs/hooks-rules.html>.

149. PALANTIR. *TSLint in 2019* [online] [cit. 2020-03-24]. Dostupné z: <https://medium.com/palantir/tslint-in-2019-1a144c2317a9>.
150. STYLELINT. *stylelint* [online] [cit. 2020-03-24]. Dostupné z: <https://stylelint.io/>.
151. POTTER, John. *csslint vs stylelint* [online]. 2020 [cit. 2020-03-24]. Dostupné z: <https://www.npmtrends.com/csslint-vs-stylelint>.
152. AGILIQ. *Linters and formatters* [online]. 2018 [cit. 2020-03-24]. Dostupné z: <http://books.agiliq.com/projects/essential-python-tools/en/latest/linters.html>.
153. PETERS, Kevin. *Auto formatters for Python* [online]. 2018 [cit. 2020-03-24]. Dostupné z: <https://www.kevinpeters.net/auto-formatters-for-python>.
154. MOZILLA. *Mozilla Observatory* [online]. 2020 [cit. 2020-03-31]. Dostupné z: <https://observatory.mozilla.org/>.
155. DJANGO. *Settings* [online]. © 2005-2020 [cit. 2020-03-31]. Dostupné z: <https://docs.djangoproject.com/en/3.0/ref/settings/>.
156. WILKERSON, Mike. *Locking the Vault on Font Awesome NPM Tokens* [online]. 2018 [cit. 2020-03-31]. Dostupné z: <https://blog.fontawesome.com/locking-the-vault-on-font-awesome-npm-tokens-2/>.
157. WIGGINS, Adam. *III. Config* [online]. 2017 [cit. 2020-04-29]. Dostupné z: <https://12factor.net/config>.
158. NIELSEN, Jakob. *10 Usability Heuristics for User Interface Design* [online]. 1994 [cit. 2020-03-30]. ISSN 1548-5552. Dostupné z: <https://www.nngroup.com/articles/ten-usability-heuristics/>.
159. PAVLÍČEK, Radek. *Web Content Accessibility Guidelines (WCAG): seznamte se, prosím* [online]. 2019 [cit. 2020-03-30]. Dostupné z: <https://www.zdrojak.cz/clanky/web-content-accessibility-guidelines-wcag-seznamte-se-prosim/>.
160. W3C. *Web Content Accessibility Guidelines (WCAG) 2.1* [online]. 2018 [cit. 2020-03-30]. Dostupné z: <https://www.w3.org/TR/WCAG21/>.
161. DJANGO. *Model field reference* [online]. © 2005-2020 [cit. 2020-04-01]. Dostupné z: <https://docs.djangoproject.com/en/3.0/ref/models/fields/>.
162. FIO BANKA. *FIO API BANKOVNICTVÍ* [online]. 2020 [cit. 2020-04-04]. Dostupné z: https://www.fio.cz/docs/cz/API_Bankovnictvi.pdf.
163. SABANIN, Ilya. *Deployments Best Practices* [online]. © 2007-2019 [cit. 2020-04-04]. Dostupné z: <http://guides.beanstalkapp.com/deployments/best-practices.html>.

-
164. ORO. *Testing and Staging Environments in eCommerce Implementation* [online]. © 2020 [cit. 2020-04-04]. Dostupné z: <https://oroinc.com/b2b-ecommerce/blog/testing-and-staging-environments-in-ecommerce-implementation/>.
 165. PITNET, Sten. *Continuous integration vs. continuous delivery vs. continuous deployment* [online]. © 2020 [cit. 2020-04-04]. Dostupné z: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>.
 166. WIGGINS, Adam. *X. Dev/prod parity* [online]. 2017 [cit. 2020-04-29]. Dostupné z: <https://12factor.net/dev-prod-parity>.
 167. KOCH, Henning. *Test if two date ranges overlap in Ruby or Rails* [online]. 2011 [cit. 2020-04-06]. Dostupné z: <https://makandrads.com/makandra/984-test-if-two-date-ranges-overlap-in-ruby-or-rails>.
 168. DJANGO. *Django's cache framework* [online]. © 2005-2020 [cit. 2020-04-07]. Dostupné z: <https://docs.djangoproject.com/en/3.0/topics/cache/>.
 169. MDN. *Window: beforeunload event - Web APIs* [online]. 2019 [cit. 2020-04-08]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Window/beforeunload_event.
 170. REACT. *Error Boundaries* [online]. © 2020 [cit. 2020-04-10]. Dostupné z: <https://reactjs.org/docs/error-boundaries.html>.
 171. OPENAPI INITIATIVE. *The OpenAPI Specification* [online]. 2020 [cit. 2020-04-16]. Dostupné z: <https://github.com/OAI/OpenAPI-Specification>.
 172. DJANGO REST FRAMEWORK. *Documenting your API* [online]. 2020 [cit. 2020-04-16]. Dostupné z: <https://www.django-rest-framework.org/topics/documenting-your-api/>.
 173. VANTOL, Alexander. *Pipenv: A Guide to the New Python Packaging Tool* [online]. © 2012-2020 [cit. 2020-04-11]. Dostupné z: <https://realpython.com/pipenv-guide/>.
 174. SECURITYHEADERS. *Referrer-Policy* [online]. © 2018 [cit. 2020-04-11]. Dostupné z: <https://securityheaders.cz/referrer-policy>.
 175. CIPAN, Vibor. *Explained: noopenner, norereferrer, and nofollow Values* [online]. 2019 [cit. 2020-04-11]. Dostupné z: <https://pointjupiter.com/what-noopenner-norereferrer-nofollow-explained/>.
 176. SECURITYHEADERS. *HTTP Strict Transport Security* [online]. © 2018 [cit. 2020-04-11]. Dostupné z: <https://securityheaders.cz/hsts>.

177. JANOVSKEÝ, Dušan. *HSTS = zákaz přístupu přes http* [online]. 2017 [cit. 2020-04-11]. Dostupné z: <https://www.jakpsatweb.cz/server/hsts.html>.
178. HOFFMANN, Nicolas. *Deploying CSP: a 5-step approach* [online]. 2018 [cit. 2020-04-15]. Dostupné z: <https://adamj.eu/tech/2019/04/10/how-to-score-a-plus-for-security-headers-on-your-django-website/>.
179. JOHNSON, Adam. *How to Score A+ for Security Headers on Your Django Website* [online]. 2019 [cit. 2020-04-15]. Dostupné z: <https://blog.dareboost.com/en/2018/03/deploying-csp-a-5-step-approach/>.
180. FARAGLIA, Daniele. *Django-environ* [online]. 2015 [cit. 2020-04-11]. Dostupné z: <https://django-environ.readthedocs.io/en/latest/>.
181. TICHÝ, Jan. *Robots.txt neslouží k zákazu indexace stránek* [online]. 2015 [cit. 2020-04-11]. Dostupné z: <https://digichef.cz/robots-txt-neslouzi-k-zakazu-indexace-stranek>.
182. HEROKU. *How to change an order result by locale on Heroku Postgres?* [online]. © 2020 [cit. 2020-04-12]. Dostupné z: <https://help.heroku.com/JSPK1LZU/how-to-change-an-order-result-by-locale-on-heroku-postgres>.
183. DJANGO. *QuerySet API reference* [online]. © 2005-2020 [cit. 2020-04-12]. Dostupné z: <https://docs.djangoproject.com/en/3.0/ref/models/queries/>.
184. DJANGO. *Databases* [online]. © 2005-2020 [cit. 2020-04-12]. Dostupné z: <https://docs.djangoproject.com/en/3.0/ref/databases/>.
185. AGGARWAL, Anshuman. *database query to user table on each request* [online]. 2017 [cit. 2020-04-12]. Dostupné z: <https://github.com/jpadilla/django-rest-framework-jwt/issues/350>.
186. PADILLA, José. *REST framework JWT Auth* [online]. 2019 [cit. 2020-04-12]. Dostupné z: <https://github.com/jpadilla/django-rest-framework-jwt>.
187. MARKBÄGE, Sebastian. *React v16.6.0: lazy, memo and useContext* [online]. 2018 [cit. 2020-04-12]. Dostupné z: <https://reactjs.org/blog/2018/10/23/react-v-16-6.html>.
188. REACT. *Context* [online]. © 2020 [cit. 2020-04-12]. Dostupné z: <https://reactjs.org/docs/context.html>.
189. VAUGHN, Brian. *You Probably Don't Need Derived State* [online]. 2018 [cit. 2020-04-17]. Dostupné z: <https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html>.

- 190. REACT. *Introducing Hooks* [online]. © 2020 [cit. 2020-04-17]. Dostupné z: <https://reactjs.org/docs/hooks-intro.html>.
- 191. ZIEROLD, Christine. *Data-QA Attribute! A better way to select elements for UI test automation* [online]. 2019 [cit. 2020-04-18]. Dostupné z: <https://dev.to/chriszie/data-qa-attribute-a-better-way-to-select-elements-for-ui-test-automation-48lm>.
- 192. HEROKU. *Heroku PGBackups* [online]. 2020 [cit. 2020-04-19]. Dostupné z: <https://devcenter.heroku.com/articles/heroku-postgres-backups>.
- 193. REACT. *React Top-Level API* [online]. © 2020 [cit. 2020-04-22]. Dostupné z: <https://reactjs.org/docs/react-api.html>.
- 194. ABRAMOV, Dan. *React 16.x Roadmap* [online]. 2018 [cit. 2020-04-22]. Dostupné z: <https://reactjs.org/blog/2018/11/27/react-16-roadmap.html>.
- 195. REACT. *Strict Mode* [online]. © 2020 [cit. 2020-04-22]. Dostupné z: <https://reactjs.org/docs/strict-mode.html>.

Seznam použitých zkratek

API Application Programming Interface

AST Abstract Syntax Tree

AJAX Asynchronous JavaScript and XML

BDD Behaviour-Driven development

CD Continuous Delivery

CI Continuous Integration

CLI Command Line Interface

CRUD create-read-update-delete

CSP Content Security Policy

CSS Cascading Style Sheets

DRF Django REST Framework

DRY Don't repeat yourself

E2E End-to-End

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

HSTS HTTP Strict Transport Security

IDE Integrated Development Environment

A. SEZNAM POUŽITÝCH ZKRATEK

JS JavaScript

JSON JavaScript Object Notation

JWT JSON Web Token

MPA Multi-Page Application

PaaS Platform as a Service

PR Pull Request

SaaS Software as a Service

SPA Single-Page Application

SSR Server-Side Rendering

SQL Structured Query Language

TDD Test-Driven development

TS TypeScript

UI User Interface

URL Uniform Resource Locator

ÚP Úspěšný prvňáček

UX User Experience

WCAG Web Content Accessibility Guidelines

W3C World Wide Web Consortium

XSS Cross-Site Scripting

Obsah přiloženého CD

	readme.txt	stručný popis obsahu CD
	src	
	impl	zdrojové kódy implementace
	thesis	zdrojová forma práce ve formátu \LaTeX
	text	text práce
	rod Lukas_dp.pdf	text práce ve formátu PDF