FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Systém pro tvorbu a správu evolvabilních dokumentů
Student:	Tomáš Starý
Vedoucí:	Ing. Marek Suchánek
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Cílem práce je vytvořit snadno rozšiřitelný systém ve formě webové aplikace využívající existující značkovací jazyk pro snadnou týmovou tvorbu dokumentů s důrazem na jejich modularitu, provázanost, znovupoužitelnost a verzování.

- Analyzujte doménu tvorby dokumentů s využitím konceptuálního modelování a stanovte požadavky na systém.
- Seznamte se s reStructuredText (docutil v Python) a AsciiDoc (Asciidoctor v Ruby), porovnejte je a proveďte rešerši existujících řešení v těchto technologiích.
- Navrhněte vlastní řešení umožňující snadnou a efektivní tvorbu evolvabilních dokumentů pomocí vybraného značkovacího jazyka. Samotný systém musí být rovněž navržen s ohledem na budoucí rozvoj jak funkcí systémů, tak i rozšíření syntaxe dokumentů (například pomocí pluginů).
- Implementujte a otestujte řešení dle návrhu. Zdůvodněte výběr programovacího jazyka i dalších technologií a nástrojů.
- Zhodnoťte přínosy systému a porovnejte jej s řešeními analyzovanými v rámci rešerše.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 25. listopadu 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Systém pro tvorbu a správu evolvabilních dokumentů

Tomáš Starý

Katedra softwarového inženýrství
Vedoucí práce: Ing. Marek Suchánek

14. května 2019

Poděkování

Mé poděkování patří vedoucímu práce, Ing. Marku Suchánkovi za skvělou podporu při vytváření této práce. Dále bych chtěl poděkovat rodině a hlavně přítelkyni za jejich schovívavost a podporu nejen při psaní této práce, ale i po dobu celého studia.

Prohlášení

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

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

V Praze dne 14. května 2019 .....

Odkaz na tuto práci

Starý, Tomáš. *Systém pro tvorbu a správu evolvabilních dokumentů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce se zabývá vytvářením dokumentů. Cílem práce je analyzovat tvorbu dokumentů a na základě analýzy vytvořit návrh řešení pro tvorbu modulárních dokumentů. Na základě tohoto návrhu je následně vytvořena aplikace. Jedná se o webovou aplikaci, která má oddělené grafické a datové prostředí. V rámci aplikace je možné vytvářet moduly vně repozitářů, ze kterých je poté možné generovat celé dokumenty, dokumenty i jednotlivé moduly podporují verzování a je možné získat i dokument starší verze ve správném znění. Vše je poté otestováno, následuje zhodnocení přínosů systému a porovnání se závěry analýzy, také se podíváme na možnosti jak aplikaci dále rozšiřovat.

Klíčová slova dokumenty, modulární dokumenty, reStructuredText, AsciiDoc, python, ruby, webová aplikace

Abstract

This thesis is focusing on creating modular documents. One of our goals is to analyse document creation and base on that proppose application desing for our application. From this design we will then create web application with separate frontend and backend. Users will be able to create modules inside repositories, using these modules are users able to create and generate documents. Both documents and modules support versioning and thanks to that, we are able to generate even older versions of our documents. At the end we tests our solution and evaluate it. We also propose new features that could be implemented in the future.

Keywords documents, modular docments, reStructuredText, AsciiDoc, python, ruby

Obsah

Úvod **1**

1 Cíl práce **3**

2 Historie psaní dokumentů **5**
2.1 Od krunýřů ke knihtisku 5
2.2 Průmyslová revoluce 6
2.3 Moderní technologie 6
2.4 Shrnutí historie 7

3 Analýza tvorby dokumentů **9**
3.1 Monolitický přístup 9
3.2 Modulární přístup 10

4 Analýza nástrojů pro tvorbu dokumentů **13**
4.1 Markdown 14
4.2 reStructuredText 16
4.3 AsciiDoc 18
4.4 Porovnání 20
4.5 Požadavky na systém 21

5 Návrh **23**
5.1 Uživatelská sekce 23
5.2 Repozitáře 25
5.3 Dokumenty 27
5.4 Návrh architektury aplikace 28
5.5 Možnosti rozšíření 28

6 Realizace **31**
6.1 Backend 31

6.2 Frontend 38

7 Testování a nasazení **43**
7.1 Testování 43
7.2 Nasazení 44

8 Hodnocení a porovnání **45**
8.1 Zhodnocení přínosů systém 45
8.2 Porovnání s analýzou 45

9 Návrhy budoucích vylepšení **47**
9.1 Testování backend části 47
9.2 Integrace lepšího verzování 50
9.3 Možnost online náhledu 50

Závěr **51**

Bibliografie **53**

A Slovník **57**

B Obsah přiloženého CD **59**

Seznam obrázků

3.1 Průběh psaní dokumentu 11

4.1 Výstup Markdown 15
4.2 Výstup reStructuredText 17
4.3 Výstup AsciiDoc 19

5.1 Diagram přihlášení a registrace uživatele 24
5.2 Diagram UC pro uživatelskou sekci 25
5.3 Diagram UC pro repozitáře 26
5.4 Diagram UC pro dokumenty 27
5.5 Diagram komponent 28
5.6 Návrh rozložení modelů 29

Úvod

S dokumenty se každý z nás setkává každý den, dokonce i teď držíte jeden v ruce. S tím, jak se rozvíjela lidská civilizace, se také rozvíjelo psaní dokumentů, od rytí znaků do krunýřů až po moderní psaní na elektronických zařízení a možnost jejich šíření po internetu. Metoda psaní se ale tolik neměnila, dokument pro nás typicky představuje jednotný celek. Představte si ovšem následující případ, máme soubor návodů na náš top produkt, který má několik verzí, které se liší i jejich instrukčním manuálem. V našich manuálech se ale nachází odkaz na legislativu, která, jak už to tak bývá, se mění. Abychom aktualizovali naše návody, musíme změnit všechny soubory, které reprezentují jeden návod. To může být časově náročné a je zde větší riziko chyby. Tato práce má přispět ke změně pohledu na psaní dokumentů. Jednotlivé dokumenty se skládají z částí, které se vyskytují v různých dokumentech, ale jsou stejné, jako v případě našeho příkladu s návody. Jednotlivé části se pak dají aktualizovat a tyto změny se pak projeví ve všech dokumentech, změna tedy proběhne pouze na jednom místě, což sníží riziko chyby a zároveň se ušetří čas.

S rozdělováním na části se běžně setkáváme v softwarovém inženýrství. Naše aplikace dělíme do jednotlivých vrstev, komponent a ještě více specificky do tříd. S tímto rozdělováním se musely vypořádat dokumentační nástoroje, které se používají při vytváření dokumentace kódu. V této práci se na některé podíváme a poté navrhneme řešení, založené na těchto nástrojích, upravené tak, aby jej bylo možné použít k sestavování dokumentu.

Kapitola 1

Cíl práce

Cílem této práce bylo vytvořit aplikaci pro správu a úpravu modulárních dokumentů. Před vytvořením aplikace byla provedena analýza tvorby dokumentů, ve které jsme si probrali rozdíly mezi monolitickým a modulárním dokumentem. Na to navázala analýza nástrojů na vytváření dokumentů. Zaměřili jsme se na značkovací jazyky, které se dnes používájí při vytváření dokumentace jako například docutil pro jazyk Python či AsciiDoc pro jazyk Ruby. Po této analýze přijde na řadu návrh aplikace, který zde již lehce nastíníme.

Aplikace bude řešit oprávnění uživatelů, přístup do jednotlivých repozitářů, což jsou složky s jednotlivými moduly. U jednotlivých modulů musíme myslet na jejich verzování, tedy možnost vracet se zpět v jednotlivých úpravách. Díky tomuto verzování bude možné generování dokumentů i retrospektivně. Dokumenty, které jsou složeny z jednotlivých modulů, je možné také verzovat a to pomocí revizí, které si budou pamatovat jaké verze modulů byly použity.

Samotná implementace bude rozdělena na 2 části, backend a frontend. První částí je backend, který nám umožnuje přístup k datům pomocí webového aplikačního prostředí. Backend má také zajišťovat ověřování uživatelů a oprávnění a možnost jejich správy. Frontend nám bude zpřístupňovat backend ve webovém prohlížeči, jedná se ale o samostatnou aplikaci, která pouze komunikuje a získává data z backendu. Dokončenou implementaci podrobíme testování, zdali vše funguje v souladu s naším návrhem. Závěrem zhodnotíme výslednou aplikaci, jestli se nám podařilo naplnit všechny cíle, které jsme si vytyčili v návrhu.

Kapitola 2

Historie psaní dokumentů

Pokud se chceme zabývat psaním dokumentů, je vhodné se nejdříve podívat do historie, na to kdy a jak se začaly psát první dokumenty, jak se psaní dokumentů měnilo s tím jak se měnila vyspělost lidské civilizace.

V dnešní době má každý z nás možnost vytvořit dokument a sdílet jej s ostatními na celém světe a to všechno v rámci několika okamžiků. Toto ovšem nebylo vždy možné. Podívejme se, jak jsme se jako lidstvo dostalo od vyškrabávání znaků do krunýřů až po dnešní psaní a sdílení dokumentu online.

2.1 Od krunýřů ke knihtisku

Základem každého dokumentu je písmo, písmo jako takové se prvně objevuje už v 7. tisíciletí před naším letopočtem a to v Číně, kde se našly kosterní pozůstatky a blízko nich i krunýře želv, na kterých se našlo první písmo. [1] Toto je nejstarší doposud nalezený artefakt obsahující písmo.

Písmo se o něco později také objevuje v Mezopotámii. Zde se objevuje klínové písmo, které přišlo jako zjednodušení pro piktogramy, které se používaly pro kontrolu obilí a dobytka. Společně s klínovým písmem se nezávysle na sobě rozvíjí písmo i na území starého Egypta, zde v podobě hieroglyfů. Tyto dvě písma se pak rozšiřují po celém regionu. Díky tomu roste v celém regionu efektivita ekonomiky a objevují se první historické záznamy [2] a také první sepsané zákony.

V Mezopotámii se pro psaní klínového písma používaly hliněné destičky, kdežto ve starém Egyptě se používal papyrus. Papyrus se vyráběl ze stébel šáchoru papírodárného a byl rozšířen po celém středomoří. Jeho výroba byla zapomenuta okolo roku 1100.

V Evropě se poté rozšiřuje používání papíru, který se k nám dostal z Číny díky Arabům. Papír byl oproti pergamenům, které jsou vyráběny z kůže, levnější na výrobu, ale byl horší kvality, tudíž pro důležité dokumenty se používal pergamen. Do poloviny 15. století se ovšem stále většina dokumentů

psala bez použití sofistikované techniky. Vše bylo stále opisováno ručně. To ovšem změnil Johan Gutenberg, který významně zdokonalil knihtisk a který se pak do konce století rozšířil po celé Evropě. Knihtisk zde byl již dříve, ale Johan přišel s nápadem odlití jednotlivých písmen z kovu, tím se zvýšila jejich životnost a díky tomu se snížila cena celého tisku. Snížení ceny mělo za následek rozšíření knih mezí více lidí a tím i zvýšení gramotnosti obyvatelstva. [3]

2.2 Průmyslová revoluce

Ještě před příchodem průmyslové revoluce se objevují první pokusy o psací stroj a to již v roce 1714, kdy byl v Anglii patentován stroj „vyrážející písmenka na papír“ [4], ovšem s psacími stroji měl pramálo společného. O více než 100 let později se objevuje první psací stroj, který přinesl rozložení kláves, které známe i z dnešních moderních počítačů. Toto rozložení bylo zavedeno kvůli problémům se zasekáváním znaků při stisku více znaků po sobě, které se nacházely blízko sebe. Psací stroje s námi byly po většinu 20. století a byly nahrazeny až počítači, ale o těch až později. Krom psacích strojů se s průmyslovou revolucí také rozšiřují možnosti tisku. Například roku 1799 si nechal N. L. Robert patentovat vynález stoje na výrobu papíru v tzv. „nekonečném“ pásu. [5]

2.3 Moderní technologie

První počítače byly masivních rozměrů a zaměřeny pouze na určité početní operace, pro které byly sestrojené. Například ENIAC, který byl za 2. světové války na univerzitě v Pensylvánii a zde jej používala americká armáda pro balistické výpočty. Tento počítač zabíral 185.81 m^2 plochy. [6] V roce 1947 přichází William Shockley, John Bardeen a Walter Brattain z Bell Laboratiories s vynálezem tranzistoru, pevného elektrického přepínače, který nepotřeboval vakuum [7] (elektronky, které se používali do této doby, jej ke svému provozu potřebují). Jak je ale patrné z rozměrů takového stroje, je jasné, že toto zařízení se masově nerozšířilo. V 70. letech 20. století přicházejí první osobní počítače. Mezi lety 1974-1977 se na trhu objevují první opravdové domácí počítače jako například Altair od Micro Instrumentation and Telemetry Systems (MITS). [7]

V roce 1976 Steve Jobs a Stephen Wozniak zakládají společnost Apple. Jejich prvním produktem je Apple I. Narozdíl od Altair měl tento počítač více paměti a levnější procesor a monitor. O rok později začinají s produkcí Apple II, který nabízel barevnou obrazovku a možnost záznamu dat na přenosné médium, kazetu, která poté byla nahrazena disketou. Apple zároveň s vydáním Apple II vyzýval programátory, aby pro své počítače vytvářeli aplikace, díky tomu například vznikl první tabulkový program nazývaný VisiCalc. Apple

tak našel cestu nejen do domácností, ale i do firemního sektoru. [7] „První zpracovaní textu na počítači se stáva skutečností, když MicroPro International vydává WordStar.“ [7]

2.3.1 Software

V dnešní době existuje nepřeberné množství sofwaru na vytvoření a zpracování textu. Uveďme si tedy ty nejznámnější i s jejich krátkou historií. Začneme tím nejznámnějším, nejpoužívanějším a v dnešním době již braným jako standard. Word přišel na svět v roce 1983 [8]. Tato verze byla určena pro systém MS-DOS. MS Office a s ním spojený Word jak jej známe dnes se ovšem objevil až v roce 1990, konkrétně 19. listopadu toho roku. S jednotlivými verzemi se zlepšovalo grafické rozhraní pro uživatele, s verzí 2016 přichází také optimalizace na mobilní zařízení a dotykové obrazovky. V roce 2011 je představen Office 365, online služba, ve které je možné vytvářet a sdílet dokumenty odkudkoliv. [9]

Neboť balíček Office je licencovaný firmou Microsoft a není možné jej používat bez licence, objevují se i open source řešení, mezi nejznámnější patří OpenOffice/LibreOffice, jejich příběh začíná v roce 1999, kdy Sun Microsystems kupuje Star Division a mění název jejich produktu na OpenOffice.org. Současně s tím vydávají kód k veřejnému užití, open source a každý si tedy může stáhnout kopii tohoto softwaru a užívat jej volně. V roce 2009 kupuje Sun Microsystems firma Oracle, neboť měli někteří vývojáři z The Document Foundatiuon obavy, co se stane s OpenOffice.org. Vytvořili si vlastní odnož (fork) OpenOffice.org a začali ji dále rozvíjet pod názvem LibreOffice. Později díky tomuto vzniká jednotný open source formát pro sdílení dokumentů, Open Document Format (ODF). Díky vývojářům z Document Liberation Project je možné otevírat i ostatní formáty jako je .doc, .docx nebo importovat soubory z Pages, což je Word v podání společnosti Apple. [10]

2.4 Shrnutí historie

Jak je vidět, s postupujícím se technologickým rozvojem lidstva přicházelo také spoustu změn týkajících se tvoření dokumentů. Od dřívější víceméně ruční práce, se lidstvo dopracovalo do moderního pojetí, kdy není potřeba šířit dokumenty jejich opisováním a je umožněno čím dál větší části lidstva vytvářet dokumenty, které mohou být sdíleny po celém světe v rámci okamžiků pomocí internetu. Co se ovšem nezměnilo, je přístup tvorby dokumentů. Stále tvoříme dokument jako jeden celek a o tom více v další kapitole.

Kapitola 3

Analýza tvorby dokumentů

Historii psaní dokumentů jsme si popsali v minulé kapitole, nyní je čas popsat princip, kterým dokumenty vznikají. Vznik dokumentů také prochází vývojem, ale tento vývoj přichází až v poslední době s rozvojem informačních technologí.

Tvorba dokumentů by se dala rozdělit na dva různé přístupy. První má za výsledek jeden soubor, který tvoří daný dokument a druhý pouze složí dokument z určitých částí.

Než si ovšem popíšeme tyto dva přístupy, je nutné se podívat, jak vlastně dokument vzniká. Nejdříve je nutné vytvořit návrh, pro který se také používá označení draft. Tento draft je pouze základní obrys toho, co by měl výsledný dokument obsahovat. Pokud se na dokumentu podílí více autorů, je tento draft kontrolován každým z nich. Z draftu se potom začne rozvíjet výsledný dokument, který se po dopsání předává ke kontrole korektorům, kde se kontroluje pravopis a stylistika. Po konečné kontrole díla autorem či autory je dílo předáno distributorovi. Celý tento postup je znázorněn na grafu. 3.1

3.1 Monolitický přístup

Monolitickým přístupem je myšleno to, že jednotlivé dokumenty jsou monolitické celky, jeden dokument je jeden celek. Pro představu, pokud si vezmeme ku příkladu Word, o kterém už zaznělo něco v kapitole o historii psaní dokumentů, vytvořením jednoho souboru s příponou .docx, jsme vytvořili jeden monolitický dokument a pokud jej budeme chtít použít v jiném dokumentu, budeme muset obsah tohoto souboru zkopírovat do nového souboru. Tento nový soubor bude obsahovat i náš původní dokument, pokud se ovšem něco změní v původním dokumentu, druhý dokument bude mít stále původní verzi. Toto poté přináší problémy, které byly již nastíněny v úvodu této práce.

Výhodami monolitického přístupu jsou jednoduché zpracování jednoho dokumentu. Není třeba jej dělit a je jednoduché jej sdílet i v původním, nezpracovaném tvaru, kdy je možné odeslat pouze jeden soubor. Nevýhody se hlavně projevují u větších textů, kdy může být nepřehledné orientování se v jednom

souboru a toto se ještě více prohlubuje u formátů, které nejsou jednoduché na čtení, pokud nejsou zpracovány například v HTML nebo XML.

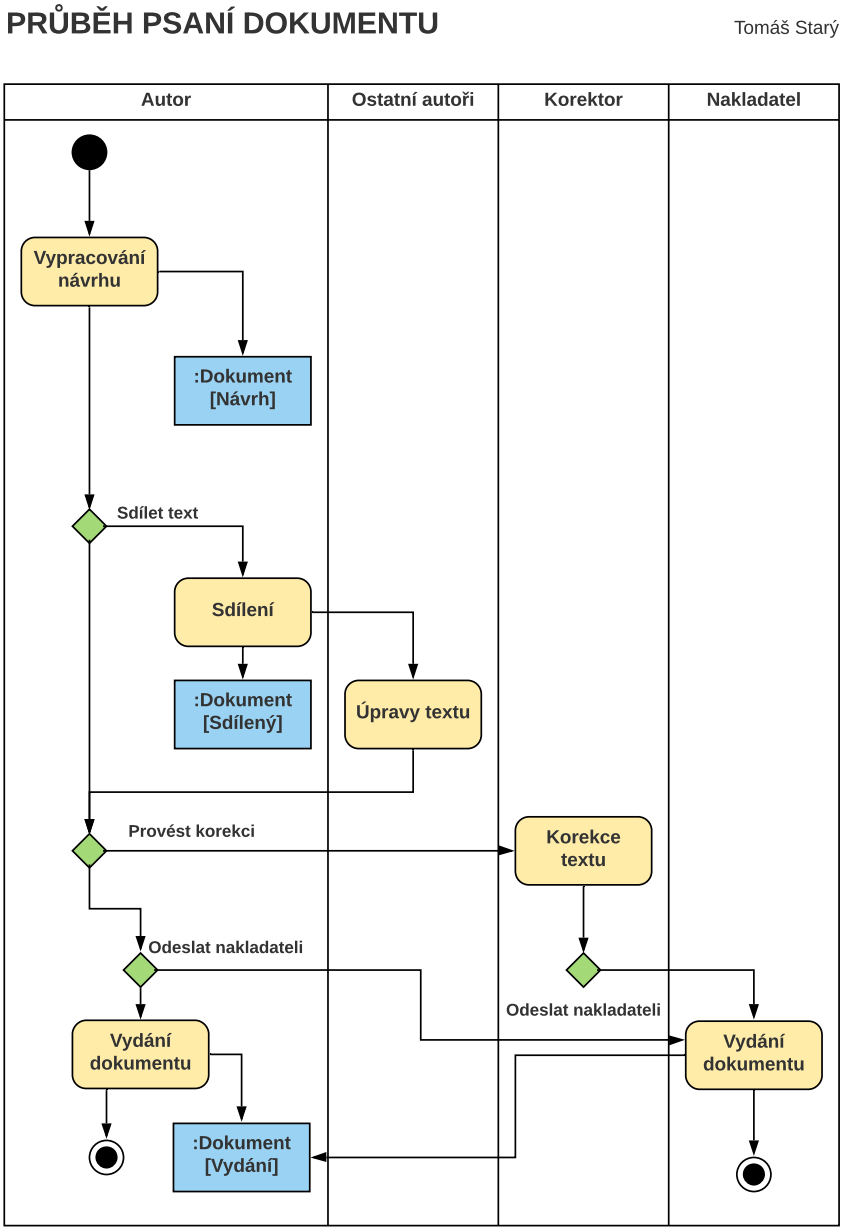
V souvislosti se zmíněným formátem .docx, který je používaný hlavně programem Word z balíčku Office od firmy Microsoft, je třeba zmínit, že minimálně od verze 2016, je možné rozdělit dokument na hlavní dokument a jeho části, které mohou být uchovávány odděleně. [11] Toto řešení je ovšem citlivé na přesuny souborů a není optimální.

3.2 Modulární přístup

Hlavní myšlenkou modulárních dokumentů je rozdělit dokument na jednotlivé moduly, tedy části textu, které je možné poté využít i v jiných dokumentech. Jako příklad uvedu tuto práci. Skladá se z jednotlivých kapitol. Tyto kapitoly jsou více či méně na sobě nezávislé a tudíž se dají rozdělit na jednotlivé moduly. Ovšem nemusíme tento dokument dělit pouze podle kapitol, je možné jej rozdělit i na měnší části. S tímto dělením si ovšem musíme dát pozor, abychom dokument nerozdělili na moc malé části, které potom samy o sobě ztratí smysl.

Jak nám toto pomůže zlepšit efektivitu psaní a rozšiřování dokumentů? Je potřeba si uvědomit, že pokud se k nám přídá nový člen týmu, který má spravovat naší dokumentaci, je pro něj složité se orientovat v jednom velkém monolitickém souboru. Pokud ovšem je možnost dokumentaci rozdělit na části, je i pro nově příchozího člena týmu jednodušší zorientovat se v upravované části a případně ji rozšířit. Pro lidi, kteří dokumentaci píšou již delší dobu, bude mimo jiné přínosné nutnost držet se jednoho tématu, který může definovat daný modul a tím pádem držet konzistentnost obsahu textu. [12]

Výhodou modulárního přístupu mimo těch nastíněných v přechozím odstavci, je možnost jednoduchého aktualizování textu, mezi vícero dokumenty, které sdílejí stejný modul. Také tento způsob přináší možnost přehlednějšího verzování, kdy je možné procházet pouze změny v jednotlivých modulech. Nevýhody jsou hlavně v krátkých a unikátních dokumentech, kdy není ani žádoucí je rozdělovat do vícero souborů. Další nevýhodou je, že ne všechny formáty, které slouží k psaní textu, tuto metodu podporují (Markdown).



Obrázek 3.1: Průběh psaní dokumentu

Kapitola 4

Analýza nástrojů pro tvorbu dokumentů

Nástrojů jak dnes psát dokumenty máme celou řadu, některé jsou volně dostupné a jiné jsou zpoplatněny. Některé nástroje jsme si již představili (Word, LibreOffice), ale nyní se zaměříme na takzvané značkovací jazyky. Značkovací jazyky nám umožnují psát text, který je poté možné zpracovat počítačem, který změní jeho formátování. Většina značkovacích jazyků má jasně rozlišitelné značky, nebo jinak také tagy, které upravují formátování při jejich strojovém překladu a díky tomu je původní text stále dobře čitelný. [13] Výhodou je u nich, kromě jednoduché čitelnosti původního textu, volnost užití. Není potřeba pro jejich úpravu žádný speciální software. Pokud je chceme převést do konečné podoby, stačí nám k tomu volně dostupné nástroje, které jsou ve většině případů dostupné i online a není třeba je tedy instalovat na naše zařízení.

Mezi značkovací jazyky například patří i jazyky používané při psaní webových stránek, jako je HTML či XML, ovšem tyto jazyky budeme spíše využívat pro zobrazování výstupu jednodušších značkovacích jazyků. My se budeme zabývat hlavně těmito 3 jazyky: Markdown, reStructuredText a AsciiDoc. Tyto jazyky se používají pro psaní dokumentace v jednotlivých programech (docutil, AsciiDoc), či slouží k vytváření uživatelských návodů (Markdown).

4.1 Markdown

Markdown je značkovací jazyk, který se typicky převádí do HTML. Jedná se o jednoduše čitelný a zároveň jednoduchý jazyk na psaní strukturovaného textu. Hlavní myšlenkou Markdown je, že text v něm psaný by měl být publikovatelný i bez jeho zpracování. Inspirací tomuto přístupu jsou čistě textové emaily (emaily se dnes ve většině případů píšou v HTML z důvodu grafického obsahu). [14]

Markdown se používá na některých verzovacích službách jako je například GitHub, ovšem v tomto případě se jedná o upravenou implementaci Markdown, která je rozšířena o další tagy/značky. Podobnou úpravu Markdown má i konkurenční služba GitLab.

Takto vypadá menší ukázka Markdown syntaxe 4.1 a také výsledek, který je poté generován 4.1.

```
1  # Ukázkový příklad Markdown syntaxe
2
3  Takto vypadá dokument psaný v Markdown,
4  jak je vidět, od čistého textu se moc neliší.
5
6  ## Víceúrovňové nadpisy
7
8  Zde je ukázka možnosti formátování textu *kurzíva*, **tučné**.
9  Více příkladů formátování například [zde](https://github.com
10 /adam-p/markdown-here/wiki/Markdown-Cheatsheet).
11
12 - příklad
13 - jednoduchého
14 - seznamu
15
16 - [ ] checkbox
```

Zdrojový kód 4.1: Příklad Markdown syntaxe

Ukázkový příklad Markdown syntaxe

Takto vypadá dokument psaný v Markdown, jak je vidět, od čistého textu se moc neliší.

Víceúrovňové nadpisy

Zde je ukázka možnosti formátování textu *kurzíva*, **tučné**. Více příkladů formátování například zde.

- příklad
- jednoduchého
- seznamu

☐ checkbox

Obrázek 4.1: Výstup Markdown

4.2 reStructuredText

Formát reStructuredText pro psaní dokumentů v prostém textu, který je snadný na čtení, kde je hned zřejmé, jak bude vygenerovaný text vypadat. Tento formát je jednoduchý na použití hlavně pro psaní programátorské dokumentace, menších webů a samostatných dokumentů. Hlavním cílem reStructuredText je definovat a uplatnit jednoduchý značkovací jazyk pro použití v Python, kde se používá k dokumentaci jednotlivých částí programu, a dalších dokumentačních nástrojích, který je jednoduše čitelný a jednoduše použitelný. [15]

„Docutil je open-source program pro zpracovaní dokumentace v textové podobě do, pro uživatele, přívětivého formátu, jako je například HTML, LaTeX či XML.“ [16] Docutil používá jako vstupní formát již zmíněný reStructuredText. Pro převod do formátu PDF lze poté použít utilitu Pandoc [17], která umí převěst různé značkovací jazyky na ostatní formáty jako například náš rst (reStructuredText formát) a to nejenom do PDF, ale také do formátů jako je Markdown, HTML a dalších. Pandoc lze také přímo využít v Pythonu, díky modulu pypandoc [18].

Stejně jako u Markdown, zde máme ukázku syntaxe 4.2 i výsledného dokumentu 4.2.

```
..File: example-rst.rst

Ukázkový text ve formátu reStructuredText
=========================================
Nižší úroveň nadpisu
----------------------

Podobně jako Markdown, nabízí reStructuredText také možnost
formátování textu *kurzíva*, **tučné**. Oproti Markdown má
reStructuredText možnost vkládat další soubory a tudíž, dokument
může být rozdělen na jednotlivé části, nebo lépe moduly.

.. include:: module.rst
```

```
..File: module.rst

Ukázkový reStructuredText modul
===============================

Toto je příklad takového modulu.
```

Zdrojový kód 4.2: Příklad reStructuredText syntaxe

Ukázkový text ve formátu reStructuredText

Nižší úroveň nadpisu

Podobně jako Markdown, nabízí reStructuredText také možnost formátování textu *kurzíva*, **tučné**. Oproti Markdown má reStructuredText možnost vkládat další soubory a tudíž, dokument může být rozdělen na jednotlivé části, nebo lépe moduly.

Ukázkový reStructuredText modul

Toto je příklad takového modulu.

Obrázek 4.2: Výstup reStructuredText

4.3 AsciiDoc

AsciiDoc je další formát pro psaní dokumentu, jedná se stejně jako reStructuredText, o modul pro jazyk Python. Tento modul je možné použít pro psaní nejenom poznámek, ale je možné jej exportovat i do formátů jako jsou .epub (formát pro elektronické čtečky knih), či PDF. [19] Syntaxe jazyku je podobná reStructuredText.

Kompilátor pro AsciiDoc byl původně psán pro jazyk Python, ovšem posléze byla syntaxe adoptována jako balíček pro jazyk Ruby a také přejmenován na AsciiDoctor [20]. AsciiDoctor lze posléze použít i přímo v kódu, a to konkrétně v jazyce Ruby. Zde je menší ukázka nejenom formátu AsciiDoc, ale i jeho použití v kódu.

Ukázka syntaxe 4.3 a výsledného dokumentu i pro AsciiDoc je vidět na obrázku 4.3.

```
1  //File: example-ascii.adoc
2
3  == Ukázkový text ve formátu AsciiDoc
4
5  === Nižší úroveň nadpisu
6
7  Podobně jako Markdown a reStructuredText, nabízí AsciiDoc také
   ↪  možnost formátování textu *kurzíva*, **tučné**. Stejně jako
8  reStructuredText nabízí i AsciiDoc možnost vkládání dalších
   ↪  modulů/částí.
9
10 include::module.adoc[]
```

```
1  //File: module.rst
2
3  === Ukázkový AsciiDoc modul
4
5  Toto je příklad takového modulu.
```

```
1  # example.rb
2  require 'asciidoctor'
3
4  Asciidoctor.convert_file 'example.adoc', to_file: true, safe:
   ↪  :safe
```

Zdrojový kód 4.3: Příklad AsciiDoc syntaxe a ukázka použití AsciiDoctor

Ukázkový text ve formátu AsciiDoc

Nižší úroveň nadpisu

Podobně jako Markdown a reStructuredText, nabízí AsciiDoc také možnost formátování textu **kurzíva**, **tučné**. Stejně jako reStructuredText nabízí i AsciiDoc možnost vkládání dalších modulů/částí.

Ukázkový AsciiDoc modul

Toto je příklad takového modulu.

1

Obrázek 4.3: Výstup AsciiDoc

4.4 Porovnání

Po představení těchto 3 jazyků je porovnáme a zhodnotíme jejich použitelnost pro modulární dokumenty a vybereme, který použijeme v rámci naší aplikace. V rámci porovnání musíme dát hlavně důraz na podporu modularity v rámci jazyka, tedy jestli je už v samostatném značkovacím jazyku podpořeno vkládání dalších souborů s textem, tedy modulů. Dále se také zaměříme na jednoduchost generování různých formátů, některé jazyky jsou podpořeny v Pandocu a pokud jsou, do jakých formátů je lze převést.

Prvně začněme s jazykem Markdown, jeho představení už máme za sebou a nyní se podívejme na možnost jeho využití při vytváření modulárních dokumentů. Bohužel Markdown nepodporuje přímé vkládání dalších souborů, tento nedostatek jej činí, oproti ostatním 2 jazykům, nepoužitelným pro modulární dokumenty. Je zde sice možná úprava kódu, který by text obohatil o moduly před výsledním zpracováním, ale tato úprava by byla náročná na provedení v takovém rozsahu, abychom mohli zaručit její bezchybnost.

Dalším na řadě je reStructuredText. Tento jazyk má možnost vkládat v rámci jednoho souboru i další soubory, které obashují další kód. Díky této vlastnosti lze u něho použít modulární přístup k vytváření dokumentů bez žádného předzpracování textu. Tento jazyk je podporován v rámci Pandocu [17] a je možné jej převest nejenom do formátu PDF, ale i spousty další, jmenovitě například Markdown, EPUB (formát pro čtečky knih) či již zmíněné PDF.

AsciiDoc je poslední jazyk na porovnání a stejně jako reStructuredText, nám i AsciiDoc nabízí možnost připojení dalších souborů, které budou převedeny na výsledný dokument. Protože je ovšem již podporován pouze v jazyce Ruby, budeme potřebovat na jeho převod do výsledných formátů instalovat jednotlivé moduly pro tento jazyk. Bohužel nelze vyzužít Pandoc, neboť tuto verzi AsciiDocu již nepodporuje. Použijeme tedy nástroj AsciiDoctor [20], tento nástroj nám výsledný dokument umí převést do formátu HTML, pro převod do jiných formátů jako například PDF je nutné mít kromě AsciiDoctor nainstalováno i jeho rozšíření asciidoctor-pdf [21]. Toto trochu znevýhodňuje AsciiDoc oproti reStructuredText, který má výhodu v možnosti použití jednoho nástroje na převod do vícero formátů.

Na závěr tohoto porovnání je třeba uvést jaký jazyk budeme v naší aplikaci používat. Tímto jazykem bude reStructuredText a to z několika důvodů:

1. Prvním důvodem je jeho snadný převod do ostatních formátů díky nástroji Pandoc [17] a jeho modulu pypandoc [18], který nám umožní používat Pandoc v přímo v aplikaci.

2. Druhým důvodem je autorova předešlá zkušenost v jazyce Python a také zkušenosti s nástojem Pandoc a modulem pypandoc.

4.5 Požadavky na systém

Na závěr celé analýzy a na základě požadavků na práci stanovme tyto funkční:

- správa modulů, možnost jejich úpravy či smazání
- generování dokumentů
- verzování dokumentů i modulů
- modularita aplikace
- rozšiřitelnost aplikace

a také tyto nefunkční požadavky:

- správa uživatelů, možnost jejich registrování a přihlášení
- oprávnění pro dokumenty i moduly
- rozdělení aplikace na backend a frontend.

Kapitola 5

Návrh

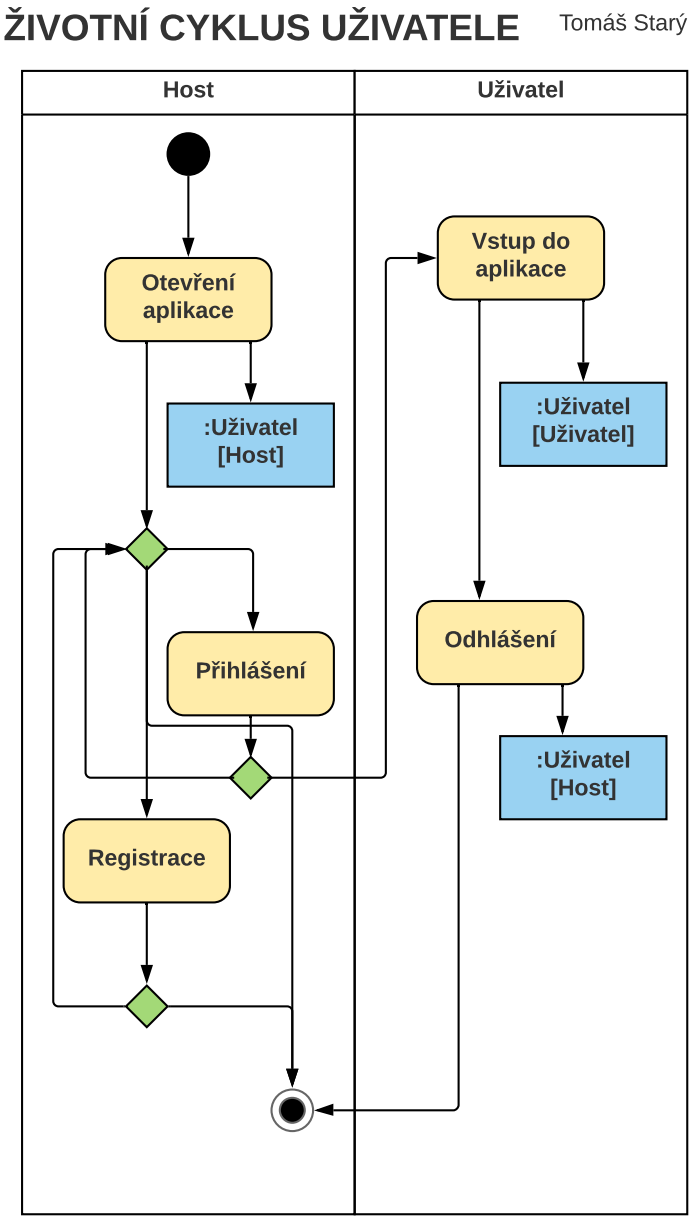
Po analýze tvorby dokumentů a nástrojů na vytváření dokumentů připravíme návrh aplikace, která bude řešit správu, vytváření a verzování modulárních dokumentů. Tato kapitola je rozdělena do jednotlivých sekcích, ve kterých budeme podrobně popisovat návrh dané sekce v naší aplikaci. V rámci návrhu nesmíme opomenout možnosti budoucího rozšíření.

5.1 Uživatelská sekce

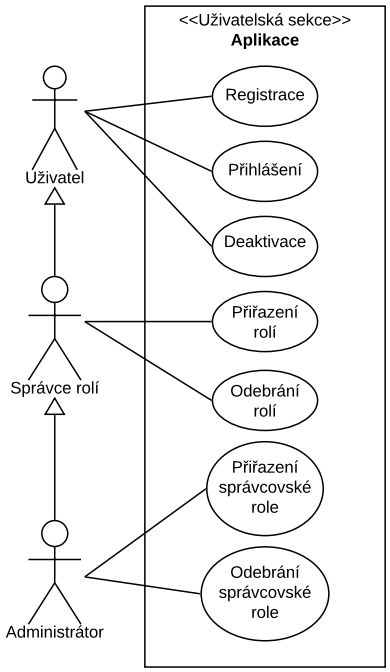
Tato sekce je zde hlavně pro potřeby vytváření neveřejných dokumentů, které nesmí být přístupné veřejnosti. Dále je také potřeba vytvořit možnost správy systému, která bude jednoduchá na použití a nebude nutné například zasahovat přímo do databáze naší aplikace. Pro tuto funkcionalitu bude ovšem nutné vytvořit speciální roli, která bude tohoto administrátora jasně identifikovat. Role ale nebude pouze administrátorská, všichni uživatelé budou mít možnost být nositeli až několika rolí zároveň. Tato vlastnost je zde kvůli jednodušší obsluze přiřazování oprávnění k dokumentům a repozitářům, které si rozebereme dále v textu. Aby hlavní administrátor aplikace nemusel tyto požadavky na přidělení rolí řešit sám, bude zde ještě další pevně definovaná role a to role správce přístupu, který bude mít možnost, stejně jako administrátor, hromadně přidělovat či odebírat role. Ovšem pouze administrátor může přiřadit roli správce přístupu.

Nyní si probereme vlastnosti uživatele, budeme potřebovat uživatele jednoznačně identifikovat, je tedy potřeba od uživatele získat nějaký údaj, který bude unikátní jako například uživatelské jméno. Dále si od uživatele vyžádáme email, který může být posléze použit k doručení zpráv a upozornění z aplikace. Tímto se ovšem dostáváme k nutnosti myslet na ochranu osobních údajů, která prošla v posledních letech změnou v důsledky nabytí účinosti evropského nařízení GDPR.

V tomto UC diagramu 5.2 naleznete rozpis všech UC, které se týkají uživatelské sekce. Zároveň k tomu je zde také stavový diagram 5.1, který nám ukazuje jednotlivé stavy uživatele, vzhledem k systému a které akce mají vliv na jeho stav.



Obrázek 5.1: Diagram přihlášení a registrace uživatele



Obrázek 5.2: Diagram UC pro uživatelskou sekci

5.1.1 GDPR

Naše aplikace bude pracovat s osobními údaji, je nutné toto brát na vědomí. Proto se nyní podívejme na to, co musí naše aplikace splňovat, aby dodržela všechny zákony o ochraně osobních údají a směrnice Evropské unie, GDPR. První co je nutné brát na vědomí je účel, za kterým údaje chceme údaje sbírat a uchovávat a jestli je náš účel oprávněný. V tomto případě o uživateli získáme osobní údaj v podobě emailu, který stačí k identifikaci určité osoby. V našem případě je email identifikátor uživatele a budeme jej při registraci žádat o souhlas se zpracováním osobních údajů. Dále musíme myslet na možnost smazání či anonymizace údajů uživatele. Toho docílíme ručními zásahy do databáze, kde využijeme možnosti změnit email na jeho hash. [22]

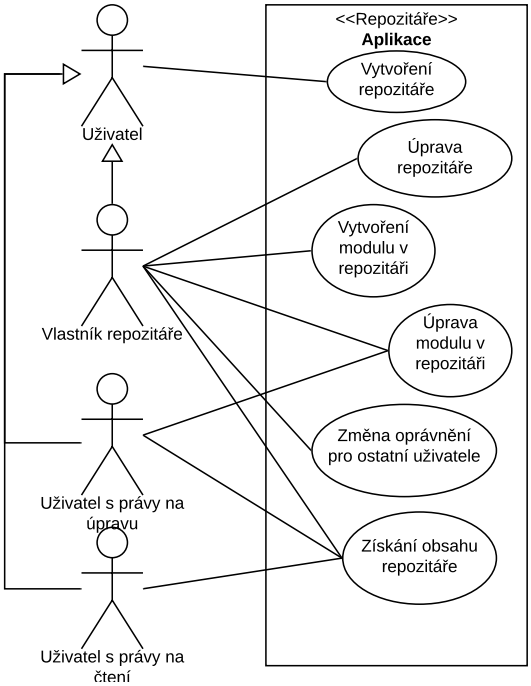
5.2 Repozitáře

Než začneme popisovat samotné generování dokumentů, je potřeba si definovat repozitáře našich modulů. Jednotlivé repozitáře nám budou sloužit jako složky, které budou obsahovat moduly, ze kterých se pak budou skládat výsledné dokumenty. V rámci repozitářů je potřeba hlavně zajistit správně fungování oprávnění. Pokud si uživatel založí nový repozitář, má vůči němu všechny

práva, ostatní uživatelé se namejí o tomto repozitáři v tomto stavu nemají jak dozvědět, je pro ně skrytý. Pokud se ovšem zakladatel rozhodne svůj obsah repozitáře sdílet, má možnost sdílet repozitář s ostatními uživateli. Dále má také možnost definovat, jestli je repozitář pro jednotlivé uživatele pouze pro čtení, či použití modulu v dokumentu, nebo jestli má možnost moduly i upravovat.

V neposlední řadě musíme myslet na verzování našich jednotlivých modulů, zde se nabízejí 2 možnosti jak tomuto přistoupit. Buď by bylo možné integrovat řešení založené na nějaké VCS, tj. systém, který zařizuje verzování souborů, nebo je možné implementovat verzování pomocí databáze. V této práci budeme volit verzování pomocí databáze a to hlavně z časových důvodů, neboť pro použití VCS by bylo nutné udělat kompletní obal na nějaký již existující verzovací systém. Na digramu 5.6 je vidět, jak by měly vypadat vztahy mezi jednotlivými moduly.

Pro jednodušší představu toho jaké nám z návrhu vyplynuly příklady užití je vytvořen diagram UC pro repozitáře 5.3. V tomto diagramu je dbáno i na rozdělení uživatelů podle úrovně jejich oprávnění ve vztahu k repozitáři.



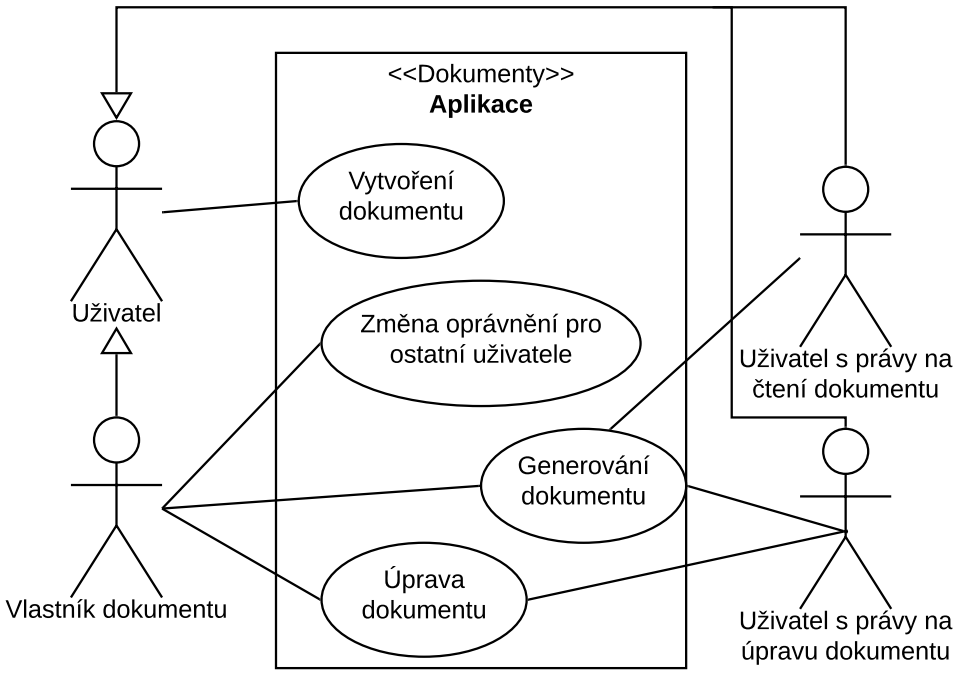
Obrázek 5.3: Diagram UC pro repozitáře

5.3 Dokumenty

Jelikož už máme definované chování pro jednotlivé repozitáře a jejich moduly, můžeme se pustit do návrhu dokumentů, které se budou skládat z modulů. Každý dokument je tedy soubor modulů, který je možné převést do tisknutelné podoby. Co ale nesmíme opomenout je, že stejně jako moduly bude možné dokumenty také verzovat, zde to bude provedeno pomocí revizí. Každá revize bude nést informace o tom, kdo danou revizi vytvořil a také ponese všechny informace o verzích modulu, které jsou na danou revizi použity.

Podobně jako u repozitářů i u dokumentů budeme řešit oprávnění a to stejným způsobem, tudíž každý dokument má svého zakladatele nebo také vlastníka, vlastník má možnost rozhodovat o tom, kdo bude moci dokument upravovat (vytvářet nové revize, měnit obsah dokumentu), a kdo bude mít možnost si pouze dokument přečíst.

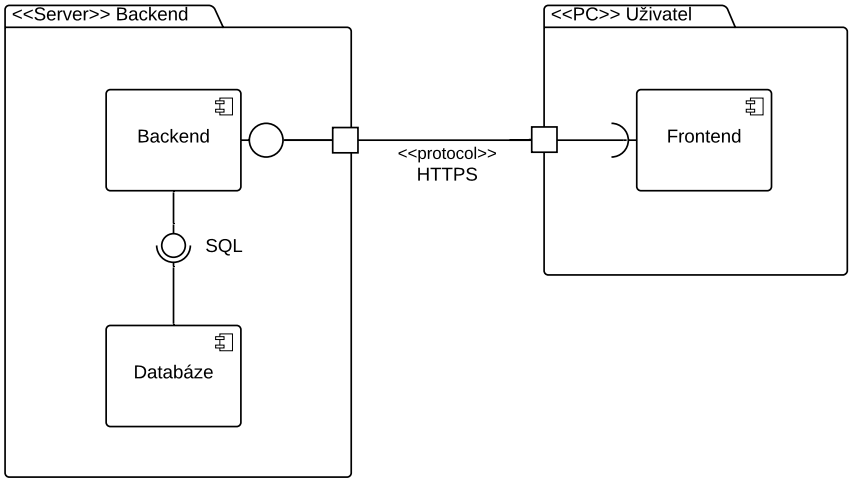
Pro lepší představu o jednotlivých případech užití je zde přiložen UC diagram pro dokumenty 5.4.



Obrázek 5.4: Diagram UC pro dokumenty

5.4 Návrh architektury aplikace

Jak bylo již řečeno v kapitole o cílech práce, aplikace by měla být rozdělena na 2 části backend a frontend. Nyní si pojdmě navrhnout architekturu, která bude vyhovovat tomuto cíly. Pro backend je důležité, aby měl přístup k databázi, ve které budou uložena všechna data, jediná možnost jak komunikovat s backendem bude skrze webové rozhraní. Propojeni s backendem nám bude zajišťovat frontendová část, ta bude umět komunikovat a zobrazovat data v grafické podobě, nadále nám také poslouží k udržení identity po přihlášení. Na tomoto diagramu 5.5 je znázorněno rozložení jednotlivých komponent dle návrhu.

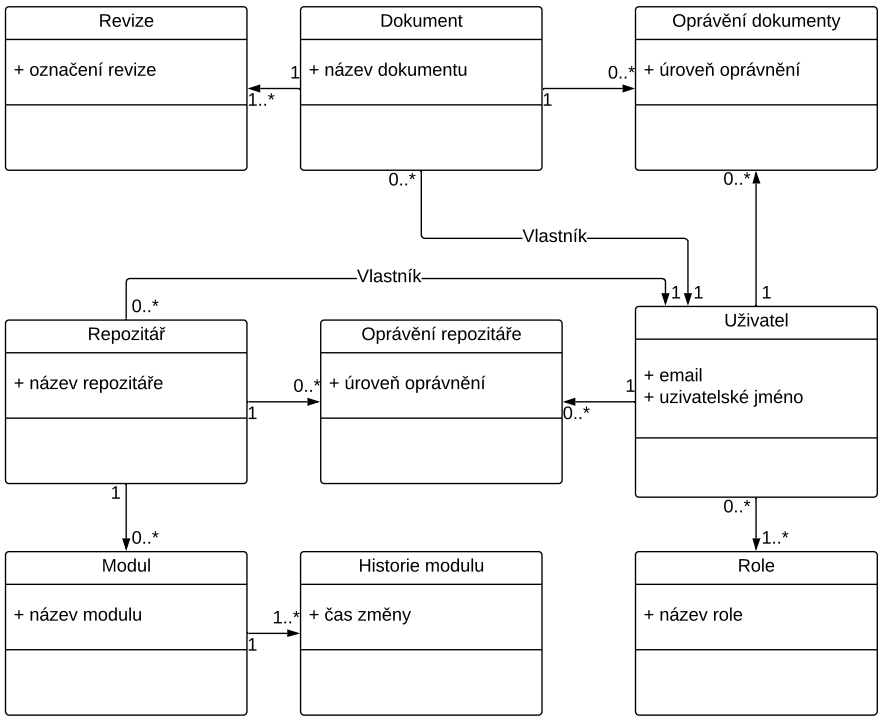


Obrázek 5.5: Diagram komponent

5.5 Možnosti rozšíření

Protože každý systém se postupně rozvíjí a rozšiřuje, musíme na tuto skutečnost myslet již při návrhu, snažíme se tedy, aby jednotlivé části systému byly od sebe oddělené a mohly fungovat i bez ostatních částí. Například propojení mezi moduly a uživateli je pouze na základě jedné vazební tabulky, která slouží k dekompozici vztahu M:N. U modulů je také podstatné, aby bylo možné jim v dalším vývoji přidat další atributy jako například typ modulu. Toho docílíme tím, že moduly budou samostatnou tabulkou v databázi a bude ji tedy možné rozšířit o další atributy, při této úpravě ovšem neohrozíme již původní data

a nebude pro nás problém takovou změnu implementovat. Podobně je tomu i u dokumentů a uživatelů. Modely se snažíme dělat co nejmenší s jednoduchou možností rozšíření viz 5.6.



Obrázek 5.6: Návrh rozložení modelů

Kapitola 6

Realizace

Implementaci rozdělíme na 2 části a to na část zabívající se backendem, tedy mozkem naší aplikace, která bude zodpovídat za ukládání a zpracování dat a bude nám také sloužit jako autentifikační server. A poté na frontend, tedy část, která je odpovědná za komunikaci s backendem a také bude zpostředkovávat zobrazení dat z backendu uživateli.

6.1 Backend

V analýze nástrojů, jsme v samotném závěru porovnávali a také vybírali, jaký značkovací jazyk použijeme jako formát pro naše dokumenty. Při výčtu výhod pro jazyk reStructuredText, je uvedena autorova předešlá zkušenost s jazykem Python, tento jazyk se tedy stane jazykem, který budeme používat při vývoji backendové části naší aplikace. Dalším důvodem pro výběr tohoto jazyka je modul *pypandoc*, který nám umožňuje převádění reStructuredText na jiné formáty a díky tomuto modulu budeme moci jednoduše generovat naše dokumenty. Mezi jazyky Python a Ruby nejsou velké rozdíly, oba jsou skriptovací jazyky. Ruby je většinou vnímán pouze jako jazyk na vytváření webových stránek, na druhé straně Python je zaměřen na univerzálnost jeho použití. [23]

Jako základ backendu využijeme *Flask*, „což je mikroframework pro Python založený na Werkzeug, Jinja 2 a dobrých záměrů.“ [24] Tento mikroframework nabízí vytváření stránek ve formátu HTML, toto ovšem nebude třeba, neboť celý backend je složen pouze z REST metod, které slouží k získání či úpravě dat. Konkrétně použijeme HTTP metody GET, POST a DELETE, které provedou určitou akci na základě volané adresy. Pro další funkcnionality využíváme v mnoha případech moduly, které rozšižují základní *Flask*. Důvodem pro výběr tohoto frameworku je jeho snadná rozšiřitelnost pomocí modulů a také nám umožňuje vytvořit jednoduchou webovou aplikaci bez zbytečných částí, které by nebyly použity.

6.1.1 Ověřování uživatele

V rámci ověřovaní uživatelů musíme zajistit zabezpečenou komunikaci mezi backendem a frontendem. Toho docílíme sdílením tokenu, který bude ověřovat každý dotaz do té části aplikace, kam mají přístup pouze přihlášení uživatelé. Toho docílíme pomocí modulu, který rožšiřujeme *Flask*, *Flask-jwt-extended*. Pro správnou funkci tohoto modulu je třeba mít nastavený tajný klíč pro aplikaci, který slouží k šifrování JWT, ve kterém se nachází identifikátor uživatele. Dále je potřeba definovat funkce pro načítání uživatele a jeho rolí, toto poté použijeme pro kontrolu oprávnění uživatele a získávání dat, která jsou vázána na uživatele viz ukázka 6.4. Kromě možnosti sdílení tokenu musíme také v naší aplikaci vyřešit možnost bezpečného ukládání hesla v naší databázi. Z bezpečnostního hlediska je nutné zajistit, nemožnost zpětného zjištění hesla zadaného uživatelem. Toto není úplně možné díky *brute force* útokům, neboli útokům hrubou silou, při které jsou zkoušeny všechny možné kombinace pro dané heslo. Tomuto se dá lépe předejít silným heslem. Pro šifrování hesla na databázi použijeme modulu *flask-bcrypt*, který implementuje užití hashovací metody bcrypt ve *Flasku*.

```
1  # File: moddoc/service/auth_service.py
2
3  @app.jwt.user_claims_loader
4  def add_claims(user):
5      """
6      Load claims into JWT
7      """
8      return {'roles': user['roles']}
9
10 @app.jwt.user_identity_loader
11 def load_user(user):
12     """
13     Function for loading user
14     """
15     return user
```

Zdrojový kód 6.4: Ukázka kódu pro *Flask-jwt-extended*

6.1.2 Propojení s databází

Na toto použijeme další rozšíření pro *Flask* a to konkrétně *Flask-SQLAlchemy*. Jedná se interpretaci *SQLAlchemy*, což je ORM nástroj pro Python. Podporuje většinu SQL implementací a serverů. V našem případě budeme brát *postgresql* jako nejvhodnější databázový server se zaručenou podporou pro *sqlite*. Protože v aplikaci používáme jako identifikátor všech modelů, používáme

GUID. Základní implementace *SQLAlchemy* nám tento datový typ nativně nepodporuje, stejně jako některé SQL servery. Proto si vytvoříme vlastní datový typ GUID. Ukázku jak na to, nalezeme i v oficiální dokumentaci *SQLAlchemy* [25]. Jak lze vidět v ukázce kódu 6.5, při vytváření tohoto typu je provedena kontrola, s jakou databází bude aplikace pracovat a podle toho se rozhodne, jaký datový typ bude v rámci databáze použit. Toto se provádí proto, že ne všechny databáze si umí poradit s GUID datovým formátem.

```python
#File: moddoc/utils.py
from sqlalchemy.types import TypeDecorator, CHAR
from sqlalchemy.dialects.postgresql import UUID
import uuid

class GUID(TypeDecorator):
"""Platform-independent GUID type.

Uses PostgreSQL's UUID type, otherwise uses
CHAR(32), storing as stringified hex values.

"""
impl = CHAR

def load_dialect_impl(self, dialect):
    if dialect.name == 'postgresql':
        return dialect.type_descriptor(UUID())
    else:
        return dialect.type_descriptor(CHAR(32))

def process_bind_param(self, value, dialect):
    if value is None:
        return value
    elif dialect.name == 'postgresql':
        return str(value)
    else:
        if not isinstance(value, uuid.UUID):
            return "%.32x" % uuid.UUID(value).int
        else:
            # hexstring
            return "%.32x" % value.int

def process_result_value(self, value, dialect):
    if value is None:
        return value
    else:
        if not isinstance(value, uuid.UUID):
            value = uuid.UUID(value)
        return value
```

Zdrojový kód 6.5: Implementace GUID datového typu

Další úpravou, kterou provedeme na implementaci *Flask-SQLAlchemy* je zavedení vlastního základního modelu, ze kterého pak budeme vytvářet všechny ostatní modely. Tento základní model bude mít v sobě definovaný identifikátor v GUID formátu, který jsme si zavedli v naší aplikaci. Díky tomu bude mít každý objekt unikátní identifikátor, neboli také primární klíč. Díky tomu bude snadnější a rychlejší vyhledávání dat v databázi a to s využitím indexů, které si databáze automaticky vytváří. Také si definujeme rozšíření pro modely, které jim bude přidávat určité atributy. Tyto atributy jsou stejné v některých tabulkách a nechceme je při vytváření nových tabulek znovu vypisovat a proto vytvoříme *model mixin*. Tato třída v sobě bude obsahovat pouze definice sloupců, které jsou stejné v těchto určitých tabulkách. Pro upřesnění, bude se jednat o zavedení takzvaného „soft delete“ systému, který nám bude sloužit k virtuálnímu mazání objektů, objektu se pouze nastaví příznak, že byl smazán, ale nebude smazán z databáze, tudíž jej bude možné obnovit. Tato úprava si ale také žádá definovaní upravených databázových dotazů a s tím nám pomůže definice vlastní `QueryClass`, třídy zodpovědné za vytváření dotazů v *Flask-SQLAlchemy* na naší databázi. V této třídě a třídách, které dědí tuto třídu si poté vytvoříme nové metody, které budou připraveny na filtrování dat s příznakem `deleted`, jako by byly smazané. Ná závěr této části tu máme ukázku tříd 6.6, které jsme si nyní popisovali.

```
#Source:
↪ http://flask-sqlalchemy.pocoo.org/2.3/customizing/#model-class
↪ # noqa 501
class IdModel(Model):
    @declared_attr
    def id(cls):
        for base in cls.__mro__[1:-1]:
            if getattr(base, '__table__', None) is not None:
                type = sa.ForeignKey(base.id)
                break
        else:
            type = GUID()

        return sa.Column(type, primary_key=True)


class SoftDeleteModel(object):
    created = sa.Column(sa.DateTime, nullable=False,
    ↪ default=datetime.utcnow)
    updated = sa.Column(sa.DateTime, nullable=True,
    ↪ default=None)
    deleted = sa.Column(sa.DateTime, nullable=True,
    ↪ default=None)

    def delete(self):
        self.deleted = datetime.utcnow()


class SoftDeleteQuery(BaseQuery):
    def soft_get(self, id, default=None):
        object = self.get(id)
        if object and object.deleted is None:
            return object
        else:
            return default

    def soft_all(self):
        return self.filter_by(deleted=None).all()
```

Zdrojový kód 6.6: Ukázka rozšiřujících tříd pro *Flask-SQLAlchemy*

6.1.3 Kontrola příchozích dat

Jako v každém programu je potřeba kontrolovat vstupní data, je také v našem případě nutné kontrolovat data, která přijdou z našeho frontendu, nebo například z mobilní aplikace, která se bude připojovat na náš backend. Pro kontrolu dat použijeme implementaci schémat z dalšího modulu pro Python, *marshmallow*. Tento modul nám umožní definovat nejen formát dat, ale také definuje formát výstupních dat, která budou načtena přímo z výstupu dotazu nebo objektu z *SQLAlchemy*.

6.1.4 Generování dokumentů

Asi nejdůležitější částí je generování samotných dokumentů. Jak již bylo v textu této práce zmíněno, bude využit software Pandoc. [17], Pro komunikaci s tímto softwarem využijeme modul *pypandoc*, který nám poskytne API rozhraní pro komunikaci s Pandoc. Pro vytvoření našeho dokumentu již máme skoro vše potřebné. Nyní je potřeba pouze zajistit zdrojové soubory, ze kterých by se mohl výsledný dokument skládat. Protože si u každého dokumentu pamatujeme, jaké repozitáře jsou použity, můžeme tyto repozitáře projít a pro každý modul v nich obsažený vytvořit soubor, který bude obsahovat data z modulu. Název takového souboru se pak bude určovat podle identifikátoru z databáze pro tento modul. Pokud bude v dokumentu použit odkaz na tento identifikátor, bude zaručeno, že dokument bude generován vždy s nejnovější verzí modulu. Pro generování dokumentu se specifickou verzí modulu, je třeba uvést jako odkaz identifikátor této verze. Před každým generování dokumentu se nejdříve vytvoří všechny potřebné soubory z modulů, které mohou být použity v dokumentu a poté se už provede samotné generování dokumentu, který je poté vrácen skrze API.

6.1.5 Struktura backendu

Přehlednost kódu je jedna z důležitých vlastností aplikace, kterou je snadné udržovat a rozšiřovat. Tomu může i velice pomoct přehledná a intuitivní adresářová struktura 6.7. V rámci backendu máme ještě možnost rozdělit si tuto strukturu podle jednotlivých vrstev, které se v naší aplikaci objevují. Například databázové modely mohou být ve vlastní složce, nebo všechny API metody lze přesunout do jedné složky a rozdělit do souborů, podle toho, k jaké entitě se dané metody vážou.

```
instance ..... složka s konfigurací, jedná se pouze o lokální složku
migrations ..... složka s migracemi databáze
moddoc
    api ..... složka obahující všechny API metody
    dto ..... objekty pro přesun informací mezi databází a uživatelem
    model ..... entity
    seed ..... obsahuje základní set dat pro databázi
    service ..... pomocné služby
    __init__.py ..... základní soubor, obsahuje vytvoření instance Flasku
    utils.py ..... utility používáné v aplikaci
```

Zdrojový kód 6.7: Popis strutkury adresářů pro backend

6.2 Frontend

Jako jazyk, ve kterém budeme vyvíjet frontend, volíme JavaScript, který je podporován všemi webovými prohlížeči a umožňujeme nám vytvářet aplikace přímo v něm. Další jeho přednosti jsou jednoduchost implementace, efektivita a bezpečnost. S popularitou tohoto jazyka roste počet framewroků a knihoven, které lze použít pro vytváření webových aplikací. Dnes mezi nejznámější patří *VueJS*, *React* a *Angular*. Pouze *Angular* je ovšem plnohodnotný framework, zbylé 2 jsou pouze knihovnami. [26] Pro naší aplikaci zvolíme ovšem *React* a to z několika důvodů:

1. jednoduchý a lehce pochopitelný
2. snadná rozšiřitelnost, díky velkému množství balíčků
3. velká komunita a s tím spojená dobrá dokumentace
4. jednoduchá možnost dalšího rozšiřování již napsané aplikace, díky znovuvyužití již napsaných komponent.

React je, jak již bylo řečeno výše, knihovna psaná v JavaScriptu pro vytváření webových aplikací. Tato knihovna nám zajištuje základ pro vytváření webového grafického rozhraní. *React* rozděluje jednotlivé prvky z rozhraní do

komponent, které se poté dají kombinovat, znovu využívat a také se dá u nich definovat rozšiřující chování. Na psaní v *Reactu* není potřeba definování si vlastních šablon. Vše je psáno přímo v JavaScriptovém kódu. [27] Samotný *React* poté ještě rozšíříme o další knihovny, které nám usnadní práci s daty, přidají další komponenty, abychom je nemuseli znovu vytvářet.

6.2.1 Redux

Nejdůležitějším rozšířením *Reactu* v této práci je knihovna *Redux*. *Redux* je, stejně jako *React*, také knihovna pro JavaScript. Jeho hlavní funkcí je ukládání si stavů aplikace, tudíž je možné předem nadefinovat všechny stavy, ve kterých se naše aplikace může objevit a není problém se pohybovat na časové ose díky této vlastnosti. Je dobré podotknout, že tyto stavy jsou perzistentní. Pro propojení s *Reactem* existuje knihovna *React-Redux* přímo od vývojářů *Reduxu*. [28] Díky tomuto je v *Reactu* poté možné znovu načítat pouze jednotlivé komponenty a není potřeba načítat celou stránku znovu, což má za následek rychlejší odezvu stránek. Propojení *React* komponenty s *Redux* je jednoduché. Stačí definovat takzvaný *reducer*, neboli definici změny stavů v závislosti na akcích. Toto je následně jednoduše přemapováno na *props*, což jsou vstupní hodnoty pro komponety. Komponenty mají přehled o aktuálním stavu aplikace a při změně se znovu vykreslují.

6.2.2 Reactstrap a formuláře

Pro frontend je také důležité, aby všechny ovládácí prvky byly stejné a celá aplikace měla jednotný design. *Reactstrap* je knihovna rozšiřující *React* o komponenty, které jsou již nastylované a to za použití *Bootstrap* stylů. Tyto komponenty jsou následně použity v celé aplikaci a díky Bootstrapu nemusíme vytvářet vlastní styly, pokud si vystačíme s tím, co nám nabízí *Bootstrap*. *Reactstrap* nabízí sadu stylů pro formuláře. Jejich validace bez odeslání dat by byla náročná, proto využijeme balíčku *availity-reactstrap-validation*, který obaluje *Reactstrap* vlastními komponenty. Tyto komponenty nabízejí jednoduchou formou možnost definovat pravidla, která musí platit pro jednotlivá pole ve formulářích. Zde 6.8 je menší ukázka formuláře z komponenty pro přihlášení. Prvky `AvInput` jsou deklarovány s atributem `required`, který říká, že dané pole nesmí být prázdné. Pokud by bylo, objeví se zpráva, která je definována pro daný `AvInput` v `AvFeedback`. Definování `AvGroup` je zde pro to, abychom odlišili jednotlivé části formuláře a mohli pro ně definovat 2 různá chybová hlášení v závislosti na tom, kterého vstupu se chyba týká. Pokud by se přesto někdo pokusil odeslat formulář s neuplnými daty, je zde podmínka, která tomu braní a to v metodě `this.handleSubmit` na ukázce 6.9.

```
//File: src/Auth/LoginPage.jsx LoginPage:render
<AvForm name='login' onSubmit={this.handleSubmit}>
<AvGroup>
    <AvInput type="text" name="username" id="loginUsername"
    ↪  placeholder={t("Username or email")} required />
    <AvFeedback>{t("Username or email is
    ↪  required.")}</AvFeedback>
</AvGroup>
<AvGroup>
    <AvInput type="password" name="password" id="loginPassword"
    ↪  placeholder={t("Password")} required />
    <AvFeedback>{t("Password is required.")}</AvFeedback>
</AvGroup>
<Button>{t("Login")}</Button>
</AvForm>
```

Zdrojový kód 6.8: Přihlašovací formulář

```
//File: src/Auth/LoginPage.jsx LoginPage
handleSubmit(e, err, val) {
    if (err.length == 0) {
        this.props.login(val);
    }
}
```

Zdrojový kód 6.9: Podmínka odeslání formuláře

6.2.3 Další moduly

V rámci aplikace je dobré od jejího začátku myslet na možnou lokalizaci textů a k tomu nám pomůže knihovna *react-i18next*, díky které bude jednoduché překládat jednotlivé části aplikace. Pro snažší komunikaci s backendem využijeme balíčku *refresh-fetch*. Tento balíček nám poslouží pro jednoduchout implementaci obnovování přístupového JWT tokenu za pomoci obnovovacího tokenu.

6.2.4 Struktura aplikace

Pro snažší orientaci při vytváření aplikace, investujeme chvilku času do definování a vytvoření adresářové struktury 6.10. Ta nám pomůže jednoduše určit, kde se nacházejí jednotlivé části aplikace. Je dobré rozdělit části týkající se pouze *Reduxu* do vlastní složky, či roztřídit jednotlivé komponenty podle toho, které části aplikace se týkají.

```
dist ..... složka obsahující produkční verzi aplikace
public ..... složka obsahující HTML s výsledným JavaScriptem
src
    actions ..... zde se nacházejí všechny akce volané z komponent
    components ..... pomocné komponenty
    constants ..... konstanty využívané v Reduxu
    helpers ..... pomocné funkce a třídy
    reducers ..... Redux reducers
    store ..... Redux store
    App ..... základní komponenta
```

Zdrojový kód 6.10: Adresářová struktura frontendové aplikace

Kapitola 7

Testování a nasazení

7.1 Testování

V rámci vývoje je potřeba testovat a zároveň i dokumentovat jednotlivé části. V našem případě při vývoji použijeme k testování API nástroj Postman [29], který nám poslouží nejen k dokumentaci našeho backendového API, ale také bude fungovat k okamžitému testování našich metod. V Postman je možné definovat jednotlivé dotazy pro náš server s možností jejich parametrizace a automatického přidání JWT tokenu pro ověření uživatele. Mimo možnosti definovat jednotlivé dotazy, lze také vytvořit pro celé prostředí jednotnou sadu parametrů, které mohou být využity v dotazech, ať již ve formě parametrů v URL nebo k ukládání proměnných, které se mohou lišit mezi zařízeními (například adresa testovacího serveru). Na přiloženém médiu je kompletní výstup z tohoto programu. Program je volně ke stažení a je možné sdílet data z jednoho zařízení na další v rámci jednoho účtu, nebo sdílet data v týmu. Tato možnost je již ovšem za poplatek.

Výsledné testování proběhne pomocí akceptačních testů. Jedná se o testy, které jsou provedeny manuálně před samotným nasazením aplikace na produkční prostředí. Testy jsou prováděny podle příkladů užití z návrhu.

Testování probíhalo v konfiguraci co nejvíce podobné produkčnímu prostředí, aby nevznikly potíže v důsledku jiného prostředí, či rozdílných konfigurací. Testy se prováděly podle příkladů užití z návrhu. Na základě těchto testů ještě byly provedeny patřičné změny v aplikaci. Zde je ukázka takového testu pro vytvoření nového uživatelského účtu, příhlášení se a také možnosti tento účet upravit. Nejdříve jsme si vytvořili seznam částí programu, které chceme podrobit testování.

1. Uživatel musí mít možnost se zaregistrovat s platnými údaji, pokud údaje jsou neúplné, či neplatné je třeba o tom uživatele informovat.

2. Registrovaný uživatel je poté schopen se přihlásit do svého nově vytvořeného účtu.

3. Přihlášený uživatel má možnost měnit své údaje.

Na základě tohoto seznamu proběhly kontroly všech zmíněných částí. Problém nastal pouze u změny údajů, kdy se uživateli nezobrazovalo chybové hlášení při změně údajů na údaje, které ovšem v systému má již jiný uživatel. Podobně byl poté takto zkontrolován celý systém. A podle jednotlivých průchodů jsme posléze opravili chyby, které se v systému vyskytovaly. Po těchto testech je naše aplikace připravena k nasazení na produkční prostředí.

7.2 Nasazení

Nasazení aplikací se od sebe trochu liší. Proto tuto část rozdělíme na 2, ve kterých popíšeme nasazení jednotlivých částí na produkční prostředí.

7.2.1 Backend

Pro nasazení backendu použijeme knihovnu pro jazyk Python, *Setuptools* [30], tato knihovna nám umožní jednoduše nainstlovat všechny potřebné balíčky, které budeme potřebovat pro běh naší aplikace. Pro nasazení na server stačí na serveru spustit příkaz `python setup.py install`. Jedinou podmínkou je přitomnost *Setuptools* na serveru. Před instalací je dobré nastavit nové připojení na databázi a bezpečnostní klíč. Toto se provede vytvořením souboru `config.py` ve složce `instance`, kterou je potřeba vytvořit v hlavním adresáři aplikace. *Flask* sice nabízí možnost vlastního serveru, jedná se ovšem pouze o funkci, která má sloužit vývojářům při vývoji. Pro ostré nasazení je toto řešení nestabilní, proto je potřeba použití WSGI (Web Server Gateway Interface). Jedná se o interface mezi webovým serverem a webovou aplikací psanou v jazyce Python. Některé webové servery mají tuto vlastnost již přímo integrovanou (Apache), které WSGI bude zvoleno je čistě na správci serveru. Na stránkách dokumentace k *Flask* naleznete ukázky pro jednotlivá řešení [31].

7.2.2 Frontend

Pro frontend využijeme *webpack*. Jde se o modul, který slouží k vytváření balíčků z našeho zdrojového kódu. Tento modul využijeme již v průběhu vývoje naší aplikace, kdy nám bude kompilovat naše soubory pro lokální použití a také kontroluje importy a exporty souborů, aby nenastal problém s chybějící referencí. *Webpack* si umí poradit se soubory ve formátu SCSS, které přeloží do formátu CSS. *Webpack* neslouží pouze k obsluze vývojového serveru, ale dá se použít i jako kompilátor pro výsledné produkční prostředí. Pro tuto vlastnost stačí použít příkaz `webpack -p --config webpack.config.js`, *webpack* poté vytvoří ve složce `dist` výsledný `main.js` a další soubory. Tyto soubory lze následně nahrát na produkční server, kde aplikace bude umístěna.

Kapitola 8

Hodnocení a porovnání

8.1 Zhodnocení přínosů systém

Nyní je na čase zhodnotit přínosy našeho systému. Jednou z hlavních předností našeho systému je jeho forma šíření, jedná se o OSS (open source systém), neboli otevřený software. Všechny zdrojové kódy jsou veřejně dostupné a to na verzovacím portálu GitHub, kde si každý může stáhnout aktuální verzi a to jak frontendu, tak i backendu. Každý si může vytvořit fork našeho kódu. Fork je vlastně pouze kopírování verzovaného kódu do nového repozitáře. Díky tomu může kdokoliv vytvořit vlastní změny a případně požádat o jejich zapracování do původního repozitáře. Mimo jiné i zdrojový kód této práce se nachází na tomto portálu a je také veřejně přístupný.

Práce kromě svého primárního účelu, kterým je generování dokumentů, může také posloužit jako výukový materiál pro začínající vývojáře. V práci i v implementaci jsou popsané základní postupy pro další rozvoj a při programování této práce byl kladen důraz na co největší kvalitu kódu a také na jeho snadné pochopení a jednoduchou orientaci v něm.

8.2 Porovnání s analýzou

V analýze jsme sepsali seznam funkčních a nefunkčních požadavků pro náš systém. Podíváme se jak jsme je v naší implementaci naplnili a případně, co nového jsme oproti analýze přidali. Ke každému požadavku přiřadíme tu část aplikace, která naplňuje daný požadavek. U některých také dodáme, co jsme oproti analýze přidali či odebrali.

1. Správa modulů, možnost jejich úpravy či smazání.

   Tuto část jsme v rámci naší implementace splnili a dokonce jsme ji ještě doplnili o repozitáře, které nám zajistí jednodušší správu použitých modulů v dokumentech a také zjednodušší sdílení modulů.

2. Generování dokumentů.

   Generování dokumentů jsme splnili v plném rozsahu podle námi definovaných specifikací.

3. Verzování dokumentů i modulů.

   Verzování modulů i dokumentů jsme zajistili pomocí ukládání předchozích stavů do databáze. Toto řešení ovšem bude trpět na zatížení daty. Čím více dat budeme mít, tím horší můžeme očekávat výkon. Pro zlepšení této situace by bylo třeba implementovat VCS pro verzování dokumentů i modulů.

4. Modularita aplikace.

   Z pohledu frontendu nám s modularitou aplikace velmi pomůže samotný *React* a jeho komponenty, které lze znovu využít a jednoduše rozšířit. Při vytváření datového modelu pro backend jsme mysleli na modularitu a snažili se o co největší nezávislost mezi jednotlivými moduly.

5. Rozšiřitelnost aplikace.

   S modularitou se pojí i jednoduchá rozšiřitelnost aplikace. Díky modulárnímu návrhu i implementaci je jednoduché rozšířit naší aplikaci.

6. Správa uživatelů, možnost jejich registrování a přihlášení.

   Společně se správou uživatelů jsme navrhli a implementovali i role, které nám mají pomoci zjednodušit správu oprávnění pro dokumenty i repozitáře. Role nám pomohou v dalších rozvojích aplikace s rozdělením uživatelů do určitých skupin.

7. Oprávnění pro dokumenty i moduly.

   Oprávnění jsme již i v tomto závěrečném zhodnocení několikrát zmínili. V rámci implementace bylo potřeba dát si pozor na správné filtrování, tuto část aplikace jsme museli pečlivě otestovat.

8. Rozdělení aplikace na backend a frontend.

   Aplikace jsme již v návrhu rozdělili na frontend a backend a proto máme možnost vytvořit jednoduše například mobilní aplikaci, neboť hlavní komunikace probíhá s REST API na backendu a obě části jsou na sobě více či méně nezávislé.

Kapitola 9

Návrhy budoucích vylepšení

Každou aplikaci je možné zlepšovat. Naše není výjimkou. V této části konkrétněji popíšeme možná vylepšení aplikace, které mohou usnadnit testování a pomohou při vývoji nebo zlepší celkové chování aplikace. Za nejdůležitější si dovolíme označit automatické testování backendu pomocí integračních testů. Zde si ukážeme menší ukázku, jak by takové testování mělo vypadat.

9.1 Testování backend části

Na backendu je potřeba otestovat hlavně jeho REST API metody, které se používají ke komunikaci. Pro jednoduché testování v rámci vývoje aplikace bylo využito nástroje Postman [29], tento nástroj ale není použitelný pro rozsáhlé a repetitivní testování. Proto se podíváme na využití jiného programu. Tím bude *pytest*. Jde o rozšíření pro jazyk Python, který dovoluje jednoduché definování automatických testů.

Než napíšeme první test v *pytest*, je potřeba nastavit *test fixtures*. *Test fixture* je základní část testu, která je pro všechny testy stejná. *Test fixture* například představuje připojení k databázi, nebo může posloužit k načtení celé aplikace v rámci integračních testů. Zde je menší ukázka jak v naší aplikaci vypadá taková *test fixtures* 9.11.

Pro testování backendu by mělo být použito integrační testování, kdy budeme testovat správné odpovědi na jednotlivé HTTP dotazy z našeho testovacího prostředí. Před samotným dotazem provedeme patřičné nasazení dat, na kterých provedeme testování naší odpovědi. Při této metodě testování se provádí test celého funkčního celku. Existuje připojení na databázi a testovací klient, který se připojuje na testovací server. Zde je ukázka takového testu na příkladu s testem přihlášení 9.12. V této ukázce je vidět volání funkce `test_login`. Parametry této funkce jsou *test fixtures*. Prvním je náš testovací klient, který umí odeslat HTTP dotaz na backend, druhým je připojení k testovací databázi, můžeme tedy přímo vkládat data do databáze, díky tomu

také máme možnost kontrolovat správnost dat přímo v databázi a zajistit tak správnost výsledku.

```
1  # Ukázka z test/conftest.py
2  import pytest
3
4  from moddoc import create_app
5
6
7  @pytest.fixture(scope='session')
8  def app():
9      """
10     Creates instance of application with test configuration for
   ↪   each test session
11     """
12     app = create_app('test')
13
14     # register our blueprints
15     with app.app_context():
16         app.init_api()
17
18     # return generator for app
19     yield app
```

Zdrojový kód 9.11: Ukázka test fixture

Díky těmto testům můžeme snadno a rychle pokrýt celou funkcionalitu backendu, a můžeme zaručit správné zpracování dat ze strany backendu. Ovšem tato metoda testování nám v případě chyby v některé z části kódu neukáže hned přesné místo, kde problém vznikl. S tím by nám pomohly *unit* testy, neboli testy jednotlivých funkcí. Tyto testy se vyznačují tím, že testují pouze danou funkci či metodu. Pokud se v testované funkci vyskytuje volání další funkce, je třeba toto volání takzvaně *mockovat*. *Mockování* je nahrazování původního volání novým voláním a lze takto nahradit například komunikaci s databází, která by v rámci unit testů neměla být využita. *Unit testování* je dalším dobrým rozšířením pro naší aplikaci.

```python
# ukázka  test/test_auth.py
from moddoc.model import User


def test_login(client, db):
    # prepare data
    username = 'test'
    password = 'SuperSecret1'
    user = User(username=username, email='test@example.com',
    ↪  password=password)
    db.session.add(user)
    db.session.commit()

    # create data
    data = {'username': username, 'password': password}

    # get server response
    response = client.post('auth/login', json=data)

    assert response.status_code == 200
    assert response.get_json()['access_token'] is not None
```

Zdrojový kód 9.12: Integrační test příhlášení

9.2 Integrace lepšího verzování

V hodnocení aplikace jsme již na toto téma narazili. Nyní si řekněme, jak by takové řešení mělo vypadat. Možností jak toto udělat, je několik, ale asi nejjednodušší verzí se zdá být integrace příkazů z *gitu* (*git* je implementace VCS) přímo v našem kódu a možnost propojení buď s veřejným repozitářem anebo vlastním hostovaným VCS serverem. S tímto je také spojená nutnost změnit princip ukládání modulů i dokumentů a také vytvoření nové logiky pro vytváření předchozích verzí, kdy by se verze orientovaly podle příslušných verzí z verzovacího systému *git*.

9.3 Možnost online náhledu

Dalším možným rozšířením pro naší aplikaci by byla možnost náhledu dokumentu a modulů ve fázi úprav. Uživatel by tak měl možnost jednodušší kontroly formátování výsledného dokumentu nebo modulu. Pro vytváření těchto náhledů by bylo možné využít *docutils*, které jsme již zmínili v analýze nástrojů. Zároveň s náhledem by bylo možné také definovat vlastní úpravy pro styl, ve kterém bude výsledný dokument generován.

Závěr

Práci jsme uvedli historií vývoje písma a vytváření písmeností, prošli jsme dopad moderních technologií na psaní a šíření dokumentů a představili jsme některé programy, které se nyní používají. V následůjicí kapitole jsme popsali rozdíl mezi monolitickým a modulárním přístupem k psaní dokumentů a také načrtli, jak probíhá psaní nového dokumentu.

Popsali jsme značkovací jazyky, které se čím dál tím více používají i mimo programátorskou komunitu. Podívali jsme se na *reStructuredText*, který jsme posléze použili v naší implementaci. Dostali jsme se tak k realizaci naší aplikace, které byla rozdělena na 2 části pro lepší možnosti rozšíření, či propojení s případnou mobilní aplikací. V části o backendu jsme se zaměřili na jednotlivé propojení modulů a důvody jejich použití.

Celou práci jsme zakončili otestováním naší implementace. Zhodnotily přínosy naší aplikace a také jsme naše řešení podrobili porovnání se závěrem z naší analýzy. Jako poslední část této práce jsme vyjmenovali možná vylepšení, které by bylo dobré v dalším vývoji implementovat, ať již za účelem zjednodušení testování, či zvýšení výkonu aplikace.

Bibliografie

1. RINCON, Paul. *'Earliest writing' found in China* [online]. 2003 [cit. 2019-02-18]. Dostupné z: http://news.bbc.co.uk/2/hi/science/nature/2956925.stm.
2. WHIPPS, Heather. *How Writing Changed the World* [online]. 2008 [cit. 2019-03-15]. Dostupné z: https://www.livescience.com/2283-writing-changed-world.html.
3. ČORNEJ, Petr; ČORNEJOVÁ, Ivana; PARKAN, František. *Dějepis pro gymnázia a střední školy*. 2. vyd. Praha: SPN - pedagogické nakladatelství, 2009. ISBN 978-80-7235-430-6.
4. DRVOTA, Jiří. *Klepání z minulosti*. 2010. Dostupné také z: https://plus.rozhlas.cz/klepani-z-minulosti-6647958.
5. ZUMAN, František. *Papír: historie řemesla a výrobní techniky*. Praha: Svaz průmyslu papíru a celulózy, 1983, 1983.
6. HISTORY.COM EDITORS. *Invention of the PC*. 2011. Dostupné také z: https://www.history.com/topics/inventions/invention-of-the-pc.
7. ZIMMERMANN, Kim Ann. *History of Computers: A Brief Timeline*. 2010. Dostupné také z: https://www.livescience.com/20718-computer-history.html.
8. COMPUTER HISTORY MUSEUM. *Timeline of Computer History*. Dostupné také z: https://www.computerhistory.org/timeline/1983/.
9. KUMAR, Arun. *History & Evolution Of Microsoft Office Software*. 2014. Dostupné také z: https://www.thewindowsclub.com/history-evolution-microsoft-office-software.
10. WATKINS, Don. *LibreOffice: A history of document freedom*. 2018. Dostupné také z: https://opensource.com/article/18/9/libreoffice-history.

11. NICK. *MS Word 2016: Creating Master Document and Sub documents* [online]. 2016 [cit. 2019-04-28]. Dostupné z: https://mywindowshub.com/ms-word-2016-creating-master-document-sub-documents/.

12. KRATKY, Robert. *Modular documentation: How to make both writers and users happy* [online]. 2017 [cit. 2019-04-14]. Dostupné z: https://opensource.com/article/17/9/modular-documentation.

13. KYRNIN, Jennifer. *What are Markup Languages?* [online]. 2018 [cit. 2019-04-12]. Dostupné z: https://www.lifewire.com/what-are-markup-languages-3468655.

14. GRUBER, John. *Markdown* [online]. 2004 [cit. 2019-04-13]. Dostupné z: https://daringfireball.net/projects/markdown/.

15. GOODGER, David. *reStructuredText Markup Syntax and Parser Component of Docutils* [online]. 2018 [cit. 2019-03-28]. Dostupné z: http://docutils.sourceforge.net/rst.html.

16. GOODGER, David. *Docutils: Documentation Utilities: Written in Python, for General- and Special-Purpose Use* [online]. 2018 [cit. 2019-04-19]. Dostupné z: http://docutils.sourceforge.net/.

17. MACFARLANE, John. *Pandoc a universal document converter* [počítačový program]. 2019. Verze 2.7.2. Dostupné také z: https://pandoc.org/index.html.

18. CONTRIBUTORS. *pypandoc* [počítačový program]. 2019. Verze 1.4. Dostupné také z: https://pypi.org/project/pypandoc/.

19. RACKHAM, Stuart. *AsciiDoc* [online]. 2017 [cit. 2019-04-14]. Dostupné z: http://asciidoc.org/index.html.

20. ALLEN, Dan; WHITE, Sarah. *AsciiDoctor* [počítačový program]. 2019. Verze v2.0.9. Dostupné také z: https://asciidoctor.org.

21. ALLEN, Dan; WHITE, Sarah. *asciidoctor-pdf* [počítačový program]. 2019. 1.5.0.alpha.17. Dostupné také z: https://rubygems.org/gems/asciidoctor-pdf.

22. SEGEŤA, Luboš. *Připravte svůj web na GDPR* [online]. 2018 [cit. 2019-04-15]. Dostupné z: https://proficio.cz/pripravte-svuj-web-na-gdpr.

23. MALDONADO, Kami. *Comparison: Ruby vs. Python* [online]. 2019 [cit. 2019-05-12]. Dostupné z: https://stackify.com/ruby-vs-python/.

24. PALLETS TEAM. *Flask* [online]. 2010 [cit. 2019-04-20]. Dostupné z: http://flask.pocoo.org/.

25. BAYER, Michael. *Custom Types* [online]. 2019 [cit. 2019-04-21]. Dostupné z: https://docs.sqlalchemy.org/en/13/core/custom_types.html#backend-agnostic-guid-type.

26. CUELOGIC TECHNOLOGIES. *Top 3 Best JavaScript Frameworks for 2019* [online]. 2018 [cit. 2019-05-12]. Dostupné z: `https://medium.com/cuelogic-technologies/top-3-best-javascript-frameworks-for-2019-3e6d21eff3d0`.

27. FACEBOOK INC. *React - A JavaScript library for building user interfaces* [online] [cit. 2019-05-02]. Dostupné z: `https://reactjs.org/`.

28. ABRAMOV, Dan. *Redux A predictable state container for JavaScript apps.* [online] [cit. 2019-05-02]. Dostupné z: `https://redux.js.org/`.

29. POSTMAN, INC. *Postman Simplifies API Development* [počítačový program]. 2019. Verze 7.0.9. Dostupné také z: `https://www.getpostman.com`.

30. PYTHON PACKAGING AUTHORITY. *Welcome to Setuptools' documentation!* [počítačový program]. Verze 41.0.1. Dostupné také z: `https://setuptools.readthedocs.io/en/latest/`.

31. PALLETS TEAM. *Deployment Options* [online] [cit. 2019-05-08]. Dostupné z: `http://flask.pocoo.org/docs/1.0/deploying/`.

Příloha A

Slovník

API Application Programming Interface - rozhraní pro programování aplikací. Určuje volání, které bude použito pro komunikaci s aplikací nebo knihovnou.

CSS Cascading Style Sheets, jedná se kaskádové styly používané pro definici vzhledu prvků na webových stránkách.

GDPR General Data Protection Regulation v překladu obecné nařízení o ochraně osobních údajů. Jedná se legislativu EU, kterou přijala i Česká republika. GDPR částečně přepisuje původní znění zákona o ochraně osobních údajů z roku 2000.

GUID Globally Unique Identifier, jedná se o unikátní identifikátor, který by měl být naprosto unikátní, jinak se také označují jako UUID (Universaly Unique Identifiers).

HTML HyperText Markup Language je značkovací jazyk používaný pro většinu webových stránek.

HTTP HyperText Transfer Protocol je internetový protokol používající se pro komunikaci s webovými servery.

JWT JSON Web Token jedná se o přístupová data ve formátu JSON dle specifikace v RFC 7519.

ORM Object Relational Mapper, neboli propojení objektů z prostředí programu do tabulek v databázi.

PDF Portable Document Format, formát pro přenášení dokumentů mezi zařízeními.

REST Representational State Transfer, jedná se o architekturu, která slouží k vytvoření, čtení, uprávě nebo smazání dat ze serveru pomocí HTTP volání.

SCSS Jedná se o rozšíření CSS, které doplňuje tento jazyk o možnost definování funkcí, je nutné jej ovšem kompilovat.

SQL Structured Query Language, neboli strukturovaný dotazovací jazyk, využívá se v rámci relačních databází k získávání dat.

URL Uniform Resource Locator - jednotná forma odkazování se na umístění webových zdrojů.

VCS VCS je zkratkou pro version control systém, což v překladu znamená verzovací systém. Tyto systémy se používájí pro sledování změn mezi jednotlivými úpravami souborů.

XML Extensible Markup Language, jazyk používající se například pro serializaci dat, hojné zastoupení má v internetové komunikaci.

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF