VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO PODPORU SPOLUPRÁCE PŘI AGILNÍM MODELOVÁNÍ A VÝVOJI SOFTWARE

A COLLABORATION TOOL FOR AGILE MODELLING AND SOFTWARE DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. JIŘÍ SEMMLER**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2017

Abstrakt

Tato práce se zabývá definicí a popsáním problému, který je na pomezí projektového a znalostního managementu s důrazem na agilní vývoj a agilní modelování. Na základě nalezeného a ověřeného problému se v práci hledá stávající řešení a následně analyzuje, specifikuje a navrhuje řešení vlastní. Právě zaměření na pokrytí potřeb ze třech různých pohledů činí návrh výsledné aplikace jedinečným. Pro navržené řešení jsou dále definovány technologie a je popsána detailní implementace výsledné aplikace. Při implementaci je použita sada externích technologií, které jsou skrze tuto aplikaci propojeny. Právě propojení služeb třetích stran je přínosem pro výslednou aplikaci a uživatele při procesu agilního vývoje softwaru a agilním modelování.

Abstract

The ain of this thesis is to define and describe specific challenges occuring on the crossroard between project management and knowledge management with the focus on agile software development and agile modeling. Based on the found and verified problem it tries to find a existing solution. After that, it analyses, specifies and designs an own solution. Focusing on covering of three different perspectives makes this thesis unique. After process of design, there are technologies defined. For all used technologies there is described detailed implementation of the application. The third-party technologies are connected in this application. This connection creates the extra added value for the application and user in processes of agile software development and agile modeling.

Klíčová slova

agilní vývoj, agilní modelování, projektový management, data, cloud, online aplikace, sdílení, kolaborace, Google Drive, Google Realtime API

Keywords

agile development, agile modeling, project management, data, cloud, online applications, sharing, collaboration, Google Drive, Google Realtime API

Citace

SEMMLER, Jiří. *Nástroj pro podporu spolupráce při agilním modelování a vývoji software.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rychlý Marek.

Nástroj pro podporu spolupráce při agilním modelování a vývoji software

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Semmler
23. května 2017

Poděkování

Děkuji svému vedoucímu práce RNDr. Markovi Rychlému, Ph.D. za odborné vedení, cenné rady a správné nasměrování při práci. Dále děkuji všem kolegům a odborníkům z praxe, kteří byli nápomocni při zpětné vazbě a návrhu projektu.

Obsah

Kapitola 1

Úvod

Na základě studia na Fakultě informačních technologií Vysokého učení technického v Brně jsem došel k oblastem IT činnosti, jako je projektové řízení, agilní metodika a modelování. Tyto tři oblasti jsem si osobně vyzkoušel v praxi při působení v několika českých i zahraničních firmách. Došel jsem k názoru, že postrádám vhodný nástroj pro správu znalostí, dat, zdrojů a modelů při řešení komplexního projektu. Realizace takového nástroje v podobě webové aplikace je tématem této práce.

V kapitole 2 nejdříve přiblížím motivaci a zaměření práce formou případové studie z praxe, kdy demonstruji problematiku na reálném příkladě. Cílem kapitoly je přiblížit situaci čtenáři a z pohledu potecionálního uživatele vysvětlit, kde je problém, jaké jsou možnosti řešení a jak bude situaci řešit výsledná aplikace.

Dále se v kapitole 3 budu zabývat přístupem agilního vývoje a agilního modelování k problematice. Cílem je zjistit, jak tato odvětví přispívají nebo by mohla přispět k řešení problému a jak je zakomponovat do výsledné aplikace. Zaměřím se na odhalení principů a praktik, které jsou již (často nevědomě) aplikovány v praxi a budu hledat nové principy, které by z teoretické nebo praktické roviny měly přispět do návrhu aplikace a výsledného řešení. Výstupem třetí kapitoly bude sada principů a praktik, které budu dále integrovat do aplikace.

V následující kapitole 4 se budu zabývat specifikací a návrhem výsledné aplikace. Při návrhu se zaměřím na zpětnou vazbu od potencionálních uživatelů, kteří budou validovat první návrh, tak jak doporučuje metodika Lean Software Development. Před samotným návrhem provedu důkladnou analýzu stávajících řešení a najdu pozitiva a negativa zkoumaných existujících aplikací. Na základě zpětné vazby následně porovnám jednotlivé myšlenky a zaintegruji je do výsledného návrhu.

Dále pro tento konceptuální návrh najdu technologie, které budou základem pro vývoj výsledné aplikace. Tyto technologie nejdříve teoreticky popíšu, poukážu na jejich přínosy a rizika, která do vývoje přinesou a definuji kompletní architekturu aplikace. V předposlední kapitole 5 se budu zabývat detailními prvky implementace aplikace. Cílem této kapitoly bude poukázat na specifické implementační detaily, které nejsou typické pro běžné webové aplikace nebo informační systémy.

V poslední kapitole 6 zhodnotím výsledek a zanalyzuji výstup a přínos aplikace pro agilní modelování a vývoj softwaru. Přínos aplikace budu demonstrovat formou případové studie nad reálným projektem. Vzhledem k předpokládanému komerčnímu přesahu navrhnu další rozšíření a potenciál rozvoje.

Kapitola 2

Motivace a cíl práce

Motivací pro vznik této práce je osobní zkušenost z pozice projektového manažera malé softwarové firmy zabývající se vývojem webových stránek a internetových obchodů. Zde došlo k reflexi následujících dvou faktů a vlastností zejména malých a středních projektů (do několika stovek pracovních hodin):

- Řešený projekt má většinou jednu osobu (z pravidla právě projektového manažera), který má projekt tzv. v hlavě (zdroje, analýzy, události v projektu, business cíle apod.), a to zejména pro malé a střední projekty. Tento fakt vede k nenahraditelnosti a závislosti na jednom člověku, což z hlediska udržitelnosti a bezpečného doručení projektu či služby je krajně rizikové. Je vhodné tuto znalost dokumentovat a sdílet, vzhledem k velikosti projektů s co nejmenší námahou i cenou.

- Do projektů vstupuje řada lidí z různých oblastí, kteří generují množství dokumentů, diagramů, návrhů a dalších zdrojů a to ideálně v online podobě (online služba poskytující svůj nástroj v podobě online aplikace dostupné skrze webový prohlížeč s možností sdílet – typicky Google Docs[1]). Tyto zdroje nazývejme artefakty. Tyto artefakty jsou různě roztroušeny po internetu (různé využité služby, úložiště, informace v komunikačních kanálech apod.) a vyžaduje značné úsilí všechny tyto zdroje dostatečně spravovat (kategorizace, distribuce, uchovávání...).

Je možné namítat, že při správném nastavení procesů a dostatečném úsilí na poli projektového a znalostního managementu je možné těmto negativním vlivům předcházet, ale bavíme-li se o malých a středních firmách, pak zpravidla není prostor pro důkladný znalostní management.

2.1 Případová studie

Demonstrujme si motivaci a potřeby pro realizaci této práce na následující případové studii. Fakta a informace zmíněné v této práci jsou založeny na skutečné realizaci projektu AngerRoom.cz[2] ve firmě DactylGoup s.r.o[3], kde v době řešení práce vystupuje autor na pozici projektového manažera.

[1] http://docs.google.com/
[2] http://www.angerroom.cz/
[3] http://dactylgroup.cz/

2.1.1 Zasazení do kontextu

Realizujeme webové stránky pro firmu poskytující zábavní služby (např. Laser game). Tento web bude mít malý modul pro přijímání objednávek a rezervaci termínů. Realizace projektu má za cíl pokrýt všechny aspekty vývoje od prvního kontaktu s klientem a projektem až po provoz služby. Obsahuje tedy následující činnosti:

- **Analýza** – Analýza businessových potřeb, Specifikaci a hrubý návrh, Validace návrhu, Porovnání s konkurenty
- **SEO** – Analýzu klíčových slov, Copywriting, Překlady, Podklady pro PPC kampaně
- **Grafický návrh** – Grafická identita, Drátěný model, Grafický návrh webu, Responzivní verze návrhu
- **Implementaci** – Testovací prostředí, Testovací scénáře, Plán spuštění
- **Provoz** – Zajištění produkčního prostředí, Správa incidentů a problémů
- ...

2.1.2 Definice problému

Všechny tyto činnosti znamenají velké množství materiálů (dokumenty, soubory), zdrojů (výstupy z online nástrojů – drátěný model, myšlenková mapa, emaily), úkolů a lidí, které musí projektový manažer znát a udržovat jejich znalost, a to ideálně ve strukturované podobě. Projekt má rozpočet v řádu několika stovek hodin a je realizovaný malou firmou v několika lidech za účasti několika třetích stran (outsourcing), tedy není prostor pro tvorbu důmyslných procesů. Pravděpodobně bude také prodlužován a komplikován různými vlivy (měnící se potřeby zákazníka, prioritizace jiných prací, externí zdržení), a tedy bude potřeba paralelně pracovat na jiných projektech. Zároveň se potřeby zákazníka v čase mění, tedy je více než vhodný agilní model vývoje.

V praxi tedy dochází k jevům, které byly zmíněny výše – definuje se jedna role a jedna osoba v podobě projektového manažera, který drží přehled o všech zúčastněných procesech, operacích, zdrojích, vstupech a výstupech apod.

Z praxe projektového manažera a dle rozhovorů s odborníky, které jsou zmíněny v dalších kapitolách práce, můžeme usuzovat, že pro organizaci těchto elementů se v praxi používají tři úrovně organizace těchto zdrojů:

- **Žádné** – Lokace všech zdrojů je čistě náhodná, ale projektový manažer má o nich znalost a přehled.
- **Offline přehled** – Použití nějakého znázornění na tabuli, papíře apod.
- **Online přehled** – Využití nějakého nástroje pro soupis těchto zdrojů a ideálně pro další sdílení a znovupoužití. Tímto bodem se budeme zabývat nejvíce.

Problémem online přehledu nad projektem je absence nástroje (dle následující analýzy), který by tento přehled podporoval jako klíčový bod své funkcionality (nástroj by byl pro to přímo navržen). Obecně lze říci, že stávající řešení se zaměřuje vždy pouze na jednu detailní oblast řízení a přehledu projektu, a to v následujících pohledech:

- **Správa úkolů a práce** – Jira, Redmine, EasyProject, Asana, Trello...
- **Správa souborů a dokumentů** – zameření na online sdílení souborů – Google Drive[4], DropBox[5], Mega.nz[6]...
- **Management znalostí** – dokumentování know-how, FAQ, wiki

V praxi pak dochází ke snaze o integraci těchto tří pohledů (úkol, zdroj, know-how) na jedno místo, ale výsledek bývá většinou pouze uzpůsobení jiného nástroje, který na to není navržen (například použití myšlenkové mapy pro tento přehled).

Přehled podobných nebo používaných řešení a příslušná analýza je řešena v kapitole 4.2.2.

2.2 Návrh řešení

Cílem této práce je navrhnout a následně vytvořit nástroj v podobě webové aplikace, který bude sledovat dva cíle. Primární cíl bude seskupovat všechny vstupy a výstupy projektu na jednom přehledném místě a sekundární cíl bude správa těchto zdrojů souladu s agilními myšlenkami. Tedy to nebude úkolovací nástroj pro správu práce, ani sdílené úložiště pouze pro správu dokumentů a souborů, ani nástroj pro správu znalostí, neboť všechny tyto přístupy řeší pouze úzkou problematiku. Bude se jednat o konptuálně odlišné řešení, které budeme definovat a popisovat v dalších kapitolách. Výslednou aplikaci pojmenujme pracovním názvem *Sherito*.

Vzhledem k častým změnám v zadání a v business potřebách je vhodné se zamyslet nad zapracováním agilních myšlenek do řízení projektu a iterativním vývojem. Důvody a principy zapracování agilní metodiky jsou popsány dále. Zmíněný nástroj, který má tato práce za cíl, by měl tedy sledovat následující cíle:

- Správa artefaktů na jednom místě.
- Dokumentace a správa dat (informace o zdroji, datum a autor vytvoření, poznámky, vazby...).
- Usnadnění s agilním řízením projektu.

[4] úložiště https://drive.google.com/
[5] úložiště https://www.dropbox.com/
[6] úložiště https://mega.nz/

Kapitola 3

Agilní přístup k vývoji softwaru

Agilní vývoj softwaru není těžká, detailně specifikovaná a deterministická metodika jako například RUP pro vývoj softwaru, nebo jako ITIL [7] pro poskytování služeb, ale je to pouze myšlenkový základ; ten je následně aplikován v nějaké individuální metodice, jako je například SCRUM, Lean SD, XP apod. [18] Tento myšlenkový základ je shrnut v agilním manifestu[1]:

- Jednotlivci a interakce před procesy a nástroji.
- Fungující software před vyčerpávající dokumentací.
- Spolupráce se zákazníkem před vyjednáváním o smlouvě.
- Reagování na změny před dodržováním plánu.

Na základě těchto myšlenek zadefinujme agilní prvky a principy, které budou potřebné a užitečné pro práci a řešený problém nastíněný v případové studii. Tyto agilní prvky, které postupně zasahují do různých procesů a činností (vývoj, modelování, testování apod.), kterými se budeme v této práci zabývat, nazývejme pojmem *Agile*. Tento pojem se běžně využívá v literatuře, protože se s ním dobře pracuje v textu a dobře zastřešuje hodnoty a principy agilního vývoje.

O Agile samotném a základních principech již bylo řečeno a napsáno mnoho, předpokládejme znalost základních pojmů, jako je Agile manifesto, SCRUM apod. Případně odkažme na příslušnou literaturu [9][25]. Zadefinujme ale pojmy potřebné pro tuto práci – *agilní modelování*, *agilní model a zdroj*, *agilní dokumentování* a *agilní vývoj softwaru* a podívejme se na ně z hlediska sdílení a dostupnosti, které jsou také silnými pilíři agilního myšlení.

Tato práce nemá za cíl pracovat s Agile čistě jako s vývojovou metodikou a vytvořit tak další nástroj pro správu úkolů nebo pro vizualizaci kanbanu[2]. Cílem práce je vytvořit integrované prostředí pro sdílení znalostí, modelů a dalších artefaktů agilního modelování potřebných k vývoji softwaru, a to v souladu s agilními principy. Přímé chápání pojmů agilního modelování a modelu vzhledem k této práci je popsáno v dalších částech této kapitoly.

Cílem kapitoly je ujasnit si pojmy, agilní priority a myšlenky, které následně budeme aplikovat při návrhu aplikace.

[1] http://agilemanifesto.org

[2] vizualizace procesu ve sloupcích, http://www.svetprodukti vity.cz/slovnik/Kanban.htm

Tyto myšlenky chceme aplikovat do výsledné aplikace. Vycházejme z předpokladu, že Agile jako samotný je souhrn tzv. best practises (nejlepších praktik). Zapracování těchto hodnot můžeme tedy považovat za potencionální přínos do výsledné aplikace. Důležitost a ověření těchto hodnot následně budeme validovat při demonstracích a rozhovorech s odborníky a s lidmi z praxe.

3.1 Modelování

Následující kapitola je založena na pracech Scotta W. Amblera v publikacích Agile modeling a Disciplined Agile Delivery [8] dostupné na agilemodeling.com.

3.1.1 Tradiční modelování

Modelování softwaru (například pomocí UML[3] nebo sysML[4]) je jednou z praktik softwarového inženýrství, které částečně řeší softwarovou krizi [15]. Hlavní cíle modelování pokrývají hlavní problémy vývoje softwaru, a to je doručení softwaru ve správných parametrech [6]:

- **Správnosti** – Software odpovídá business potřebám klienta.
- **Spolehlivosti** – Aplikace je stabilní a bezchybová.
- **V termínu.**
- **Za předpokládanou cenu.**
- **S možností dalšího rozšíření** – Existuje možnost (dokumentace, připravený návrh, znalosti...) rozšíření další funkcionality nebo duplikace.

Výše zmíněné body vidí modelování jako způsob snížení rizika, ale prakticky můžeme vidět v modelování a například v UML následující funkce [24]:

- **Lepší komunikace** – Model definuje společný jazyk a chápání pojmů mezi jednotlivými rolemi vývoje; tedy podporuje komunikaci mezi vývojáři, analytiky, architekty, uživateli a dalšími zúčastněnými stranami.
- **Vizualizace** – Modely jsou zejména vizuální, tedy dávají lepší přehled nad rozsahem architektury.
- **Hlubší porozumění systému a problematiky** – Díky vizualiazaci a analýze je možné hlouběji pochopit problém, snadněji jej přenést na další roli a najít možnosti pro zjednodušení a znovupoužití.

Modelování je nejlépe demonstrovatelné na vodopádovém modelu, kde hned ve druhém kroku se má podílet na návrhu výsledného softwaru. Na tomto modelu je ale zásadní problém v předpokladu, že se specifikace a business potřeby cílového vlastníka softwaru nemění během celého cyklu. Právě na tomto předpokladu a na neochotě trávit čas nad příliš detailním modelem často kolabuje princip modelování se snahou co nejrychleji dojít k funkčnímu prototypu a následné validaci [24].

[3] http://www.uml.org
[4] http://www.omgsysml.org

3.1.2 Agilní modelování

Rychle se měnící potřeby vedly k návrhu agilního vývoje softwaru. Agile má dopad na všechny aspekty vývoje softwaru, samozřejmě i na modelování, kde definuje pojem agilní modelování a agilní model.

Prvním krokem je aplikace výše uvedených myšlenek agilního manifesta v podobě následujících hodnot, praktik a principů, které definoval Scott W. Ambler [8], na klasické modelování s cílem o spojení agilního vývoje a modelování[5].

- **Komunikace** – Je důležité zajistit kvalitní a přínosnou komunikaci mezi členy týmu, zúčastněnými subjekty a mezi týmy. Vytvořené modely musí tuto komunikaci podporovat a být snadno srozumitelné.
- **Jednoduchost** – Jednoduchost a porozumění modelu je kritické pro jeho další využití a přínos při vývoji. Není-li element modelu jednoduchý a srozumitelný, je zbytečný.
- **Zpětná vazba** – Každý impulz zpětné vazby je nutné nechat rezonovat. Ať už přichází od kohokoliv (vývojář, klient, uživatel...) v jakémkoliv formátu (obrázek, text, diagram).
- **Odvaha** – Při agilním vývoji, které je uzpůsobeno na reakce na nové a nečekané podněty, je nutné mít odvahu pro razantní rozhodnutí. Někdy je zapotřebí prosadit a razantně změnit směr vývoje, který se vydal nesprávnou cestou (např. refaktoring[6]).
- **Pokora** – Pokora na poli softwarového vývoje je schopnost připustit chybu a neznalost a akceptovat individuální expertní schopnost někoho jiného. Nemá-li vývojář znalost pro řešený problém, měl by si říct o pomoc beze strachu. Zároveň je nutné respektovat fakt, že každý zúčastněný subjekt má své pole expertízy.

Přínosy aplikace agilního modelování jsou dle S.W. Amblera zejména **komunikace**, **efektivita** (pramenící z výše uvedených argumentů), **flexibilita** na použité metodice a **zlepšení procesů** modelování a dokumentace.

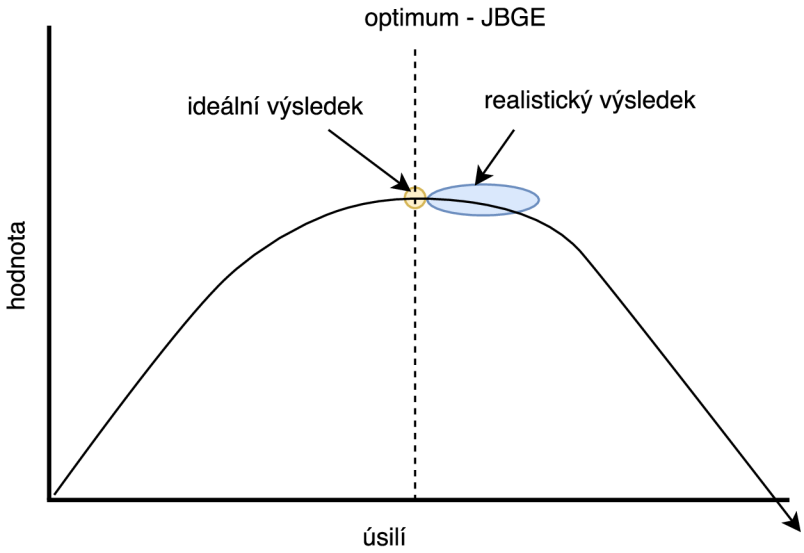
První rozdíl mezi odlišnými přístupy k modelování je tedy cíl. Zatím co klasické modelování má za cíl zejména ušetření nákladů, tak pro agilní modelování je důležitý spíše proces vedoucí k přidané hodnotě (komunikace, zpětná vazba a efektivita).

Dále definujme pojmy *Just Barrely Good Enough*[7] a *modelstorming*[8], které následně využijeme pro definici agilního modelu a modelování.

[5] http://www.agilemodeling.com/values.htm
[6] metoda vylepšování a čištění zdrojového kódu programu
[7] http://agilemodeling.com/essays/barelyGoodEnough.html
[8] http://agilemodeling.com/essays/modelStorming.htm

Just Barely Good Enough

Důležitými pojmy agilního modelování, kterým se nejvíce odlišuje od klasického přístup je Just Barely Good Enough (JBGE, volně přeloženo jako *akorát dostačující*). Jedná se o princip podobný principu KISS (Keep it simple, stupid), který říká, že dokument nebo model musí být tak úplný nebo detailní pouze pro potřeby řešeného úkolu nebo řešené potřeby. Tento princip je aplikován na agilní modely i agilní dokumenty. Cílem tedy není dělat modely nebo dokumentaci v maximálním rozsahu nebo detailu, ale zajistit úplnost pro stávající potřeby a hlavně ji co nejefektivněji doručit všem zúčastněným stranám, které ji potřebují v co nejjednoduší podobě. Jednoduchost je důležitá, neboť je-li model příliš složitý nad rámec potřeb, pak z komplikuje situaci, místo, aby ji ulehčil.



Obrázek 3.1: Just Barely Good Enough

Scott W. Ambler tímto obrázkem [9] demonstruje ideální situaci, kdy snaha a investovaná energie do tvorby (modelu, dokumentu...) je zastavena ve chvíli, kdy má model nejvyšší hodnotu. Další snaha o zlepšení modelu by již nevedla k navýšení jeho hodnoty, naopak by jeho hodnota mohla klesat (zejména z důvodu složitosti a komplexnosti). Zároveň Ambler přiznává, že nalezení tohoto bodu (demonstrovaného přerušovanou čarou) je utopické, nedeterministické a bylo by naivní se domnívat, že je možné jej plně navrhnout a splnit. Což ale není důvodem, abychom se k němu nesnažili alespoň přiblížit. Například u tohoto obrázku Ambler uvádí, že si není jist, zda jsou šipky potřebné či nikoliv.

Nyní nacházíme **druhý rozdíl** mezi klasickým přístupem k modelování a agilním modelováním, a to v rozdílu na detailní přístup k modelování. Zatímco klasické modelování říká – modeluj plně do detailu, agilní modelování se drží principu – modeluj jen tak, jak je potřeba.

[9] zdroj http://agilemodeling.com/essays/barelyGoodEnough.html

Modelstorming

Scott W. Ambler dále v souvislosti s agilním modelováním uvádí jako další praktiku tzv. modelstorming. Popisuje ji jako sadu několika rychlých schůzek a porad s kolegy pro návrh a model nové funkcionality. Využívá se princip brainstormingu[10], ale se zaměřením na modelování a návrh. Cílem je analyzovat a rychle modelovat problematiku. V běžném přístupu by pro novou funkcionalitu mohla být (konkrétní realizace se může lišit) sepsána formální specifikace, následně by byla specifikace předána analytikovi, který by vytvořil potřebné detailní modely (např. v UML). Modelstorming naopak říká, že stačí několik krátkých porad (v řádu několika minut) s vhodnými účastníky (výběr na základě přínosu a zainteresovanosti návrhu, nikoliv na základě formálních důvodů) s jakýmkoliv vhodným nástrojem (tabule, papír, CRC štítky) a za použití běžných diagramů nebo nákresů je možné vytvořit rychlý, dostatečně objemný a zároveň lehký model pro další vývoj.

Kromě míry detailu je také potřeba uvažovat vhodné typy modelů s vhodnými nástroji. Není nutné využívat hned sofistikované modelovací aplikace (Visual paradigm[11] nebo MS Visio[12]), ale stačí kreslit na tabuli nebo použít rychlé online aplikace jako např. draw.io[13], Moqups[14] nebo i Google Docs[15]. Zároveň je vhodné využít různých diagramů a návrhů pro různé pohledy na implementaci. Ve svém příkladu používá Ambler drátěný model, vývojový diagram, sekvenční diagram a CRC štítky.

Všimněme si, že mluvíme o modelování, a přitom jsme z klasického UML použili pouze jeden diagram ze čtyř vytvořených.

Tedy vidíme již **třetí rozdíl** proti klasickému modelování, a to z hlediska volby nástrojů a interesovaných stran. Zatímco v tradičním modelování by modeloval analytik za použití sofistikovaného nástroje (software i jazyk), tak v agilním přístupu může mít stejnou váhu i náčrt na tabuli od lidí různých rolí.

Agilní modelování tedy definujme jako proces tvorby agilních modelů (bude popsáno v další kapitole), kde hlavní roli hrají výše uvedené hodnoty (komunikace, jednoduchost, zpětná vazba, odvaha a pokora) a principy (JBGE, modelstorming). V agilním modelování vidíme tři základní rozdíly od klasického modelování, a to v cíli (snížení nákladů proti procesu a přidané hodnotě), v komplexnosti (aplikace principu JBGE) a v procesu (modelstorming).

3.1.3 Agilní model

Jak již bylo zmíněno v ukázkovém případě, agilní modelování (ale i jiné přístupy) pracuje s myšlenkou různých pohledů na model a použití různých artefaktů jako modelů, jelikož každý model nabízí jiný pohled na věc a řeší jiný aspekt vývoje softwaru, zvláště v dnešní vysoké komplexitě softwaru. Ambler nabízí základní přehled asi čtyřiceti[16] artefaktů, které se považují za model. Je mezi nimi UML, sysML, myšlenkové mapy, User story[17], drátěný model... Popisuje modelování a výběr vhodného modelu k výběru vhodného nástroje při

[10] skupinová kreativní technika pro generování nápadů a hledání řešení

[11] modelovací nástroj, http://www.visual-paradigm.com/

[12] modelovací nástroj firmy Microsoft, http://products.office.com/cs-cz/visio/flowchart-software

[13] online nástroj pro tvorbu diagramů a nákresů, draw.io

[14] online nástroj pro tvorbu drátěných modelů a prototypů, http://moqups.com

[15] balíček kacenlářských nástrojů od firmy Google, http://docs.google.com

[16] http://agilemodeling.com/artifacts/

[17] Pojem z metodiky SCRUM popisující uživatelskou operaci se softwarem. Jedná se o vysokoúrovňovou definici požadavku, typicky ve tvaru „Já jako uživatel A chci provést operaci B, tedy se stane C“.

opravě domu. Každý problém potřebuje jiný nástroj (nebo sadu) a proto je vhodné mít v zásobě sadu různých nástrojů.

Právě roztříštěnost a potřeba více druhů modelů pro efektivní modelování je jedním z prvků motivace této práce.

Ukažme si zmíněnou teorii víceaspektového modelu na následujícím příkladě [11], kde ukážeme vhodnost použití dvou modelů – User story (pojem metodiky SCRUM) a Use case (diagram UML).

Mějme internetový obchod, který podporuje platby převodem na účet. Z hlediska uživatele nebo specifikace pro zákazníka je ideální použít tzv. User story, které známe například z metodiky SCRUM. User story je uživatelsky velice přívětivý způsob specifikace, protože se dá velice jednoduše vysvětlit, použít při analýze s klientem a hlavně je v souladu s použitím v metodice SCRUM. Nazvěme si tedy User story *Platba na účet* v podobě „Já jako *registrovaný zákazník internetového obchodu* chci *zaplatit za nákup převodem na účet*, tedy *mě systém přesměruje na platební bránu mého internetového bankovnictví*“. User story je jednoduchá, pochopená klientem. Jsme na úrovni specifikace, tedy je dostačující použít tento model.

Platba na účet ovšem vyžaduje spoluúčast systému s pravidelným kontrolováním plateb. Jak modelujeme tento případ? Není to User story (nevstupuje uživatel, jsou nutné vstupní a výstupní podmínky apod.), je to výrazně technický problém, tedy se hodí použít například Use case diagram z UML – například případem užití *UC 01 Pravidelné kontrolování stavů plateb*. Jednoduše se mu dají specifikovat prvky vstupních a výstupních podmínek, opakovatelnost, systém jako aktér atd. Následně například pro model komunikace s API[18] banky by bylo vhodné použít sekvenční diagram.

Na uvedeném příkladě vidíme, že jedna operace má více různých pohledů a úrovní a je vhodné použít více různých přístupů.

Můžeme tedy agilní model (nebo chápání modelu z pohledu Agile) zadefinovat jako jakýkoliv exaktní artefakt (bez ohledu na formu) použitý pro modelování (či jinou metodu jako specifikování nebo dokumentování), který definuje vlastnosti nebo chování řešené problematiky a přináší přidanou hodnotu do procesu vývoje.

Tedy dle Scotta W. Amblera jsou agilním modelem například testy, myšlenkové mapy, marketingové persony, technické požadavky, drátěné modely, fotografie tabulí, příklady existujících řešení apod. Samozřejmě agilním modelem jsou také diagramy jazyků UML nebo sysML, ale agilní modelování nespoléhá pouze na tyto jazyky[19].

3.2 Agilní dokumentování

Jak už bylo řečeno v motivaci vzniku práce, cílem je zaměřit se na různé zdroje, které vstupují do vývojového procesu softwaru; což jsou modely, ale také dokumenty.

Už v Agile manifestu se uvádí, že fungující software má přednost před vyčerpávající dokumentací. Z tohoto konstatování ale nesmíme vyvodit, že by Agile plně vynechával dokumentaci. Nikoliv, má k ní jen odlišný přístup. Zatímco tradiční dokumentace se zaměřuje pouze na snižování rizik, Agile vidí dokumentaci jako strategii, která naopak může zvýšit rizika projektu (při nutnosti obnovovat či přepisovat dokumentaci nebo při chybách způsobených zastaralou dokumentací), proto se snaží být co nejefektivnější při její tvorbě [10]. Proto se k ní definují odlišné přístupy, než jen psaní statických dokumentů.

[18] Application Programming Interface, rozrahraní pro komunikaci aplikací

[19] http://agilemodeling.com/artifacts/

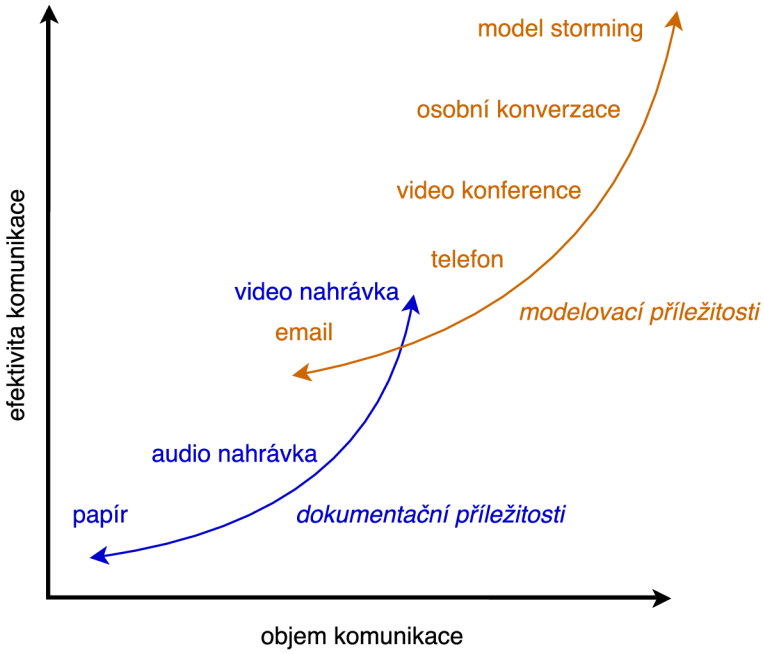
Scott W. Ambler definuje pro agilní dokumentování celkem osmnáct principů[20], které se zaměřují na jednoduchost, účelnost a efektivnost dokumentace, aby sloužila požadovanému cíli, který jsme definovali výše. Pro tuto práci jsou důležité zejména následující principy.

Uchovávat informace a dokumentaci na co nejvhodnějším místě

Není-li dokumentace dostupná, pak se její užitečnost rapidně snižuje. Je tedy nutné ji uchovávat co nejblíže k použití. Jako ideální způsob dokumentace je komentování zdrojového kódu a případné generování výsledné dokumentace. Jsou ale zdroje, které není možné do kódu vložit (například audiovizuální materiály), proto je vhodné dokumentaci uchovávat alespoň na jednom dobře přístupném místě. A právě toto místo má být cílem řešené aplikace.

Najít nejlepší způsob, jak dokumentaci komunikovat

Pokud jsme ukázali, že model nemusí být pouze UML, pak dokumentace nemusí být pouze dokument. Cíl dokumentace je dobře uchovávat a komunikovat nějakou informaci k následnému použití zúčastněnými stranami, nikoliv preferovat kvantitu. Proto se volí různé způsoby dokumentace dle jejich síly komunikační efektivity, jak demonstruje následující obrázek[21] vycházející opět z chápání agilní dokumentace Scotta W. Amblera.



Obrázek 3.2: Znázornění komunikačních forem ve vztah efektivity a objemu komunikace

Cílem tedy je použít vždy správný kanál pro dokumentaci s co nejmenší námahou na tvorbu, poněvadž interpretace zvyšuje riziko zastarání. Tedy například dodá-li klient poznámky k softwaru, je vhodné je uchovávat v původní podobě i s historií.

[20] http://www.agilemodeling.com/essays/agileDocumentationBestPractices.htm
[21] zdroj, http://agilemodeling.com/essays/communication.htm

Začít s modely, které zůstaly aktuální

Modely, které zůstaly aktuální přes celý implementační proces (ať už z důvodu nezměněných potřeb nebo po aktualizaci), mají svou dokumentační úlohu, proto je vhodné je začlenit do dokumentace. Doporučuje se ale s nimi pracovat (nebo alespoň ponechat dostupné) v surové podobě (aplikační soubory modelovacích nástrojů, odkazy do online služeb, fotografie tabulí...).

Výstupem z agilního dokumentování tedy pro práci vychází fakt, že i dokumentace je zdroj, který je potřebné držet co nejblíže vývoji (ideálně v kódu či jiné formě). Tyto zdroje pak držet v použitelné podobě, tedy nenechat je zastarat a podřídit je agilním principům.

Agilní modelování tedy dle výše uvedených principů a definic můžeme popsat jako metodiku pro efektivní modelování (proces tvorby modelů) a dokumentování softwarových produktů. Tato metodika vychází z myšlenek úzké spolupráce, důrazu na efektivitu a drží základní agilní princip dodávání přidané hodnoty v co nejkratším čase.

3.3 Agilní vývoj

Agilní model jsme v sekci 3.1.3 stanovili jako jakýkoliv exaktní artefakt, který má přidanou hodnotu pro vývoj softwaru a agilní modelování jsme definovali jako podpůrý proces vývoje. Tedy definujme jak agilní vývoj zpětně reaguje na agilní modelování a jaké nové principy do něj vnáší.

Připomeňme si některé z vlastností agilního vývoje tak, abychom je pak mohli integrovat do řešené aplikace [23].

- Iterativní řížení – Vývoj je organizován do iterací.
- Dedikovaný model – Finanční model pro klienta.
- Stále funkční verze
- Úzká vazba s klientem
- Reakce na změny – Neuvažuje se „zmražená specifikace“, ale se změnami je počítáno.
- Omezení rigidních a neefektivních procesů na základě zpětné vazby – Může se jednat i o agilní prvky, které jednoduše vývojovému týmu nevyhovují.
- Maximální dostupnost artefaktů – eliminace neefektivních omezení v přístupu k potřebným artefaktům.

Z výše uvedených vlastností vývoje můžeme vidět průnik s již definovanými principy, ale i nové, které jsme ještě dostatečně nezdůraznili. Z již uvedených principů akcentujme **Maximální dostupnost artefaktů** a z nových jsme ještě neuvedli stěžejní agilní praktiku a to je **Iterativní řízení**. Z hlediska Maximální dostupnosti artefaktů můžeme při popsaném distribuovaném prostředí narazit na problémy s přístupovými právy. Artefakty mohou být součástí prostředí, které principiálně není veřejné. Konkrétně si můžeme situaci představit nad emailovou schránkou. Některé emaily se dají vnímat jako artefakty potřebné k vývoji (například surová specifikace od klienta), ale určitě není vhodné, aby projektový manažer sdílel svoji schránku veřejně. Toto je opět příležitost pro realizovanou aplikaci.

Dále nový, ještě nediskutovaný, princip je **Iterativní řízení**. Agilní vývoj obecně probíhá v iteracích, kdy v každé iteraci je produkt obohacen o novou přidanou hodnotu (například v podobě funkcionality). Tento princip, který je známý například z metodiky SCRUM, kde je vývoj organizován do tzv. sprintů, by měl být respektován i z pohledu modelů, dokumentů a dalších artefaktů. V každé iteraci mohou být důležité jiné artefakty, nebo mohou mít jinou roli. Zde tedy definujme průnik mezi agilním vývojem a agilním modelováním, jako iterativní organizaci artefaktů, řízenou dle iterací vývoje.

3.4 Přínos Agile pro řešenou aplikaci

V této kapitole jsme se snažili ukázat, že do vývojového procesu vstupuje řada zdrojů (modelů, dokumentů, poznámek, komunikačních kanálů...), které existují s různými záměry, reflektují odlišné aspekty vývoje a jsou také v odlišné podobě a formě. Správa těchto artefaktů – emailů, PDF dokumentů, výstupů různých online služeb a jiných forem je jedním z problémů, které musí řešit zúčastněné strany (a zejména osoby řídící projekt). A právě tuto správu má řešená aplikace řešit a poskytovat ji v souladu s diskutovanými principy.

Zopakujme tedy jednotlivé body plynoucí z výše uvedené kapitoly, které budou z hlediska Agile klíčové pro návrh aplikace. Proti těmto bodům budeme následně validovat návrh.

- Maximální dostupnost jednotlivých artefaktů
- Iterativní přístup, možnost definování iterací a jejich návrat či porovnání
- Podpora komunikace a spolupráce mezi zůčastněnými stranami
- Maximální snaha o vizualizaci
- Univerzální podpora odlišných forem modelů
- Důraz na jednoduchost, dostupnost a přehled, eliminace procesních překážek

Kapitola 4

Návrh

4.1 Vstupní specifikace

V předchozí kapitole 3 jsme definovali agilní přístupy, které chceme pro výslednou aplikaci dodržet. Můžeme namítat, že existují aleternativy, jako je využití těžkých metodik (ITIL nebo E/RUP), případně nově přicházející novinka DevOps, ale současný stav vývojářské a projektové komunity ukazuje snahu o maximální přibližování k Agile. Zároveň jsme v první kapitole nastínili motivaci a cíl výsledné aplikace. Specifikujme tedy podrobněji záměry a funkčnosti výsledné aplikace dle výše uvedených cílů. Výstupem kapitoly bude návrh výsledné služby v podobě interaktivního prototypu, který bude specifikovat funkcioanlity a možnosti výsledné služby.

4.1.1 Správa entit, jejich agregace a vizualizace

Entity

Připomeňme si základní terminologii a základní cíle definované v kapitole 3. Do vývoje softwaru a řízení projektu obecně vstupuje mnoho artefaktů v různých formách a z různých zdrojů. Tyto artefakty nadále označujme *entity*. Může se jednat o cokoliv, co má digitální podobu. Ať už je to online či offline – soubor z offline softwaru (kancelářský dokument od klienta, PDF, video, PSD návrh...), výstup z online aplikace – myšlenková mapa z Coggle[1], grafický návrh v Zeplin[2], drátěný model v Moqups, dokument na Google Drive, sdílená složka na Dropboxu[3], email, úryvek z komunikace na Slacku[4], úkol z Asana[5], změna v repozitáři, odkaz na webovou stránku, UML model z Creately[6] nebo export UML diagramu z VisualParadigm. Všechny tyto příklady jsou entity, které vstupují do projektu a mají v něm nějakou úlohu. Mají svoji roli z hlediska agilního modelování (jsou modelem) nebo dokumentování (jsou formou dokumentace). Cílem aplikace je tyto entity spravovat a vizualizovat v jedné aplikaci. Entity graficky vizualizujme kruhem s popisem.

[1] nástroj pro tvorbu myšlenkových map, https://coggle.it/
[2] exportování a sdílení grafických návrhů, https://zeplin.io/
[3] sdílené úložiště, https://www.dropbox.com/
[4] komunikační platforma, https://slack.com/
[5] úkolovací nástroj, https://app.asana.com/
[6] modelovací platforma, https://creately.com/

Agregace

Máme definovaný pojem entita jako samostatnou jednotku, a tedy se zcela nabízí začít vytvářet mezi entitami vazby nebo je seskupovat; definujme tedy *agregaci*. Jak jsme již nastínili v případové studii, všechny tyto entity jsou různě rozmístěné po internetu a souhrnný přehled se udržuje jen stěží. V úvodní kapitole 2 jsme již zmínili, že řešení, jak udržet přehled je dle úrovní následující:

- **Žádné** – Lokace všech zdrojů je čistě náhodná, ale projektový manažer má o nich znalost a přehled.
- **Offline přehled** – Použití nějakého znázornění na tabuli, papíře apod. Tento přístup je velice oblíbený, ale chybí mu sdílení a práce online.
- **Online přehled** – Využití nějakého nástroje pro soupis těchto zdrojů a ideálně pro další sdílení a znovupoužití. Tímto bodem se budeme zabývat nejvíce.

Právě zmíněný online přístup k správě těchto entit nazvěme *agregací entit*, kdy je aplikace bude shromažďovat na jednom místě. Toto místo – grafický prostor v online aplikaci, kde se budou entity vizualizovat a kde budeme s nimi pracovat, označme jako *dashboard*. Při jeho grafické realizaci vycházejme z oblíbeného principu tabule (whiteboardu), tedy se na dashboard můžeme dívat jako na jednu virtuální tabuli, kde jsou vizualizované agregované entity.

Nahrazení stávajících nástrojů

Jak si ukážeme dále v analýze stávajících řešení, v žádném případě se nemá jednat o náhradu jakéhokoliv z uvedených nástrojů. Princip, který chceme sledovat, je neměnit zavedené způsoby práce jednotlivých zúčastněných stran, pouze nabídnout možnost agregace výstupů (tedy entit) na jednom místě. Tedy na příkladu – chce-li analytik pracovat v nástroji draw.io a využívat jako úložiště Google Drive, pak nechť tak činí i nadále. Aplikace má za cíl pouze registrovat tuto informaci o umístění entity na online úložišti, náležitě ji vizualizovat a nabídnout k další správě, ale v žádném případě nechce simulovat ani nahrazovat zavedené nástroje. Snaha o nahrazení těchto nástrojů by vedla pouze k duplikaci již existujících sofistikovaných řešení a nebyla by ani v souladu s agilním myšlením, protože by se takto definoval nástroj k použití.

4.1.2 Dokumentace a správa dat

Budeme-li mít agregované entity, je vhodné nad nimi definovat nějaké dodatečné funkcionality. Zde buďme ale výrazně obezřetní, abychom nenahrazovali jiný nástroj a ani nekomplikovali situaci. Praxe ukazuje, že stávajícím trendem je zejména jednoduché a promyšlené uživatelské rozhraní (jak ukazuje popularita aplikací Trello, Asana nebo Slack), které neobtěžuje zbytečnou funkcionalitou, která není potřeba a nesleduje primární cíle. Proto nebudeme prozatím definovat ani atributy entity ani jednotlivé operace, neboť by se jednalo pouze o spekulaci.

4.2 Konceptuální návrh aplikace

V předchozích kapitolách jsme definovali principy, které chceme následovat a určili základní specifikaci. Cílem této sekce je udělat konkrétní návrh aplikace pokrývající všechny uživatelské aspekty.

4.2.1 Proces návrhu

Při procesu návrhu budeme postupovat dle principů Lean Software Development (dále jen LeanSD) [20][22], který je známý zejména z prací Erica Riese a Ashe Maurye. Pro návrh aplikace se tedy chceme vyvarovat tradičnímu přístupu a pouze na základě našich předpokladů navrhnout celou službu. Jak ukazuje Maurye v knize Running Lean, tento model se na řadě projektů ukazuje jako chybný, nákladný a vedoucí k neúspěchu projektu. Naopak LeanSD se ukazuje jako silně perspektivní. Tento princip, podobně jako Agile, říká, že bychom měli iterovat, začít s jednoduchým prototypem a postupně zdokonalovat návrh na základě zpětné vazby potencionálních uživatelů. Proces návrhu bude tedy následující:

1. Na základě stanovených principů a případové studie se pokusíme najít podobné služby, které se snaží řešit stejný nebo podobný problém. Tyto služby vyzkoušíme a zhodnotíme klady a zápory, které budeme chtít napodobit nebo se jim vyvarovat.

2. Dále na základě definovaných hodnot, případové studie, výsledků analýzy potencionální konkurence a autorova pohledu na věc vytvoříme první návrh v podobě interaktivního prototypu – drátěného modelu.

3. Nad prototypem provedeme několik hloubkových rozhovorů [20] s potencionálními uživateli – odborníky z praxe, kteří se potkávají se stejným problémem a budeme sledovat jejich reakce a zpětnou vazbu. Výsledkem bude sada návrhů na změnu přístupu k aplikaci a jednotlivým funkcionalitám a principům.

4. Na základě zjištěné zpětné vazby provedeme revizi návrhu.

5. Druhý návrh zdokumentujeme a navrhneme technické podklady pro implementaci.

Budeme-li dodržovat tento postup, pak bychom měli dojít k výrazně kvalitnějšímu produktu, který bude lépe plnit potřeby uživatelů.

4.2.2 Analýza stávajících podobných řešení

Z analýzy nástrojů a produktů, které jsou na trhu, vyplývá, že výše uvedený problém se snaží řešit nástroje, které můžeme rozřadit do následujících kategorií:

- Správu úkolů
- Správa znalostí
- Myšlenkové mapy
- Komplexní řešení
- Modelovací nástroje
- Kombinace

Uvedené kategorie nejsou překvapivé, neboť problém zasahuje do projektového managementu, managementu znalostí a organizace zdrojů. Na základě těchto určených kategorií definujme pro každou kategorii několik zástupců (vedoucí hráče odvětví) a pokusme se najít objektivní zhodnocení situace na základě podkladových dat. Celkově docházíme k analýze třiceti odlišných nástrojů. Definujme pro nástroje a služby tato kritéria:

- Být online a dostupný z webového prohlížeče. Offline aplikace pro povahu řešeného problému nepovažujeme za vhodné.
- Mít dostupné demo aplikace nebo mít možnost se dostat k reálné aplikaci po předchozí registraci.
- Mít živý vývoj. Historické zastaralé aplikace není vhodné brát v úvahu.
- Být lokalizovaný alespoň do angličtiny.

Pro jednotlivé kategorie řešení uveďme několik zástupců a obecná pozitiva a negativa. Kompletní detail analýzy najdeme v příloze.

Nástroje pro správu úkolů

– Jira[7], Asana, Redmine[8], Trello[9], VersionOne[10].

- Pozitiva
  - Jednoduchý přístup a dobrá orientace.
  - Viditelný přínos pro projektové řízení.
  - Častá podpora hlubších prvků projektového řízení – Ganttův diagram, SCRUM, Oceňování...
- Negativa
  - Neřeší další aspekty – správa znalostí, komunikace...
  - Často komplikované uživatelské rozhraní a zbytečná komplexita.

Obecný výstup

Tyto nástroje v obecném pohledu jsou všechny stejné. Snaží se o maximální důraz na projektový management, integraci některých z metodik (většinou SCRUM) a případně o propojení s externími službami, ale pouze ve velmi omezené míře. Rozdíly mezi nimi jsou v zaměření na cílový trh. Některé jsou obecné (jako Asana), některé výrazně zaměřené například na vývoj softwaru (Jira), některé přizpůsobitelné (Redmine).

[7] úkolovací nástroj, https://www.atlassian.com/software/jira
[8] úkolovací nástroj, http://www.redmine.org/
[9] nástroj pro organizaci a správu dat, https://trello.com/
[10] úkolovací nástroj pro Agile a DevOps, https://www.versionone.com/

Nástroje pro správu znalostí

– DokuWiki[11], Screenstep[12], Keeeb[13], Knowledge plaza[14]

- Pozitiva
  - Možnost uchovávat téměř všechny znalosti.
  - Možné filtrovací možnosti typu FAQ.
  - Sdílení s ostatními.
  - Historie a iterace.
- Negativa
  - Ruční a nepohodlný zápis a správa – vše je manuálně.
  - Statický obsah.
  - Slabá agregace.

Obecný výstup

Základ myšlenky těchto řešení je v běžně známém přístupu wiki stránek – tedy jednoduše ale manuálně upravitelné stránky s persistencí v aplikaci. Nad touto myšlenkou byl dále vystavěn agregační princip ústící například v FAQ a rozhodovací stromy pro uživatelskou podporu. Obecný princip je ale málo flexibilní a vyžaduje značné úsilí od správce pro vložení a organizaci znalostí. Je také silně náchylný k chybám a nemá sebevalidační složku jako je notifikace o zastarávání apod.

Myšlenkové mapy

– Mindmup[15], MindMeister[16], Coggle[17]

- Pozitiva
  - Moderní přístup k problematice.
  - Silný vývoj.
  - Jednoduché a účelné uživatelské rozhraní.
  - Velký důraz na vizualizaci.
  - Snaha o integraci s dalšími službami.
- Negativa
  - Původní účel v opoře učení a kreativity, nikoliv v projektovém řízení.
  - Málo prostoru pro poznámky nebo komunikaci.

[11] platforma pro tvorbu wiki stránek, https://www.dokuwiki.org/
[12] nástroj pro sdílení znalostí, http://www.screensteps.com/
[13] sdílení informací a poznámek, https://www.keeeb.com/
[14] platforma pro sdílení znalostí, https://www.elium.com/
[15] nástroj pro správu myšlenkových map, https://www.mindmup.com/
[16] nástroj pro správu myšlenkových map, https://www.mindmeister.com/
[17] nástroj pro správu myšlenkových map, https://www.coggle.it/

Obecný výstup

Myšlenkové mapy jsou opravdu populární nástroj a jsou používány k mnohem více různým činnostem, než byly původně navrženy. Což znamená snahu o integraci nových prvků (funkcionality, třetích stran, dat...), ale také to znamená omezení vzhledem k původní myšlence. Můžeme ale říct, že z hlediska vizualizace, znázornění vazeb, jednoduchého uživatelského rozhraní a s snahou o integraci s novými službami (populární je integrace Google Drive) jsou myšlenkové mapy velkou inspirací a také konkurencí pro výslednou aplikaci.

Podnikové řešení

– eXo Platform[18], EasyRedmine[19], xWwik[20], Confluence[21]

- Pozitiva
  - Pokrývá téměř všechny oblasti.
  - Integrace a pokrytí téměř všech elementů při vývoji softwaru a řízení firmy.
- Negativa
  - Příliš komplexní a složité.
  - Cena.
  - Složitý přístup k jednotlivým prvkům.
  - Snaha o vnucení implementovaných principů, malá míra svobody.

Obecný výstup

Analýza takto velkých podnikových řešení vyžaduje velký objem času, protože tyto nástroje obsahují velké portofolio funkcionalit a možností. Analýza takového objemného nástroje do detailu, jako je třeba Confluence od Atlassian, by zabrala zbytečně mnoho času a výsledek by nebyl přínosný. Vhodné je dodat, že když se při vývoji použije kompletní sada nástrojů, kterou tato řešení mají, pak je pravděpodobné, že nebude potřeba další nástroj, ale také dojde k jevu známému jako uzamčení [16] či závislost na jednom dodavateli.

Modelovací nástroje

– Draw.io[22], Gliffy[23], LucidChart[24], Creately[25]

- Pozitiva
  - Kolaborativní přístup.
  - Velká svoboda.

[18] otevřená kolaborativní platforma, https://www.exoplatform.com/
[19] podnikové řešení založené na produktu Redmine, https://www.easyredmine.com/
[20] podnikové řešení, http://www.xwiki.org/
[21] komplextní platforma pokrývající celý proces vývoje softwaru, https://confluence.atlassian.com/
[22] online nástroj pro tvorbu diagramů a nákresů, https://draw.io
[23] online nástroj pro tvorbu diagramů a nákresů, https://www.gliffy.com/
[24] online nástroj pro tvorbu diagramů a nákresů, https://www.lucidchart.com/
[25] modelovací platforma, https://creately.com/

  - Snaha vizualizovat téměř vše.
  - Integrace třetích stran (ukládání na Google Drive).
  - Rychlý start – přihlášení přes OAuth[26].

- Negativa
  - Nulová sémantika elementů.
  - Zbytečná snaha o všemodelující přístup.
  - Nulová validace.

Obecný výstup

Analýza takto velkých podnikových řešení vyžaduje velký objem času, protože tyto nástroje obsahují velké portofolio funkcionalit a možností. Analýza takového objemného nástroje do detailu, jako je třeba Confluence od Atlassian, musela být tedy komplexní, aby byl výstup přínosný. Vhodné je dodat, že když se při vývoji použije kompletní sada nástrojů, kterou tato řešení mají, pak je pravděpodobné, že nebude potřeba další nástroj, ale také dojde k jevu známému jako uzamčení [16] či závislost na jednom dodavateli.

Zvláštní aplikace – Zapier

Zapier[27] je aplikace patřící rodiny aplikací tzv. „If then else“. Jsou aplikace, které reagují na různé podněty a vykonávají další operace. Například „pokud dojde email s přílohou, archivuj přílohu na Google Drive do složky Práce/Klient“. Zajímavostí této aplikace je „čtení“ různých zdrojů a následné spuštění procesu, což, jak si dále ukážeme, je jeden z principů, které budeme chtít integrovat. Důležitá poznámka této služby je, že integruje až 750 různých aplikací dohromady, což je její síla. Na základě této služby je možné modelovat firemní procesy, automatizovat běžné úkony apod.

Zvláštní aplikace – Mural.ly

Služba Mural.ly[28] mi byla prvně představena ve firmě IBM při konferenci IBM Agile 2k16 Retrospective, kde IBM používá tuto aplikaci jako nástroj pro komunikaci mezi disalokovanými členy týmu. Vlastností této aplikace je práce s virtuální tabulí, na které se řeší projekt a jsou na ní umístěny jednotlivé elementy vstupující do řešení projektu. Tento princip úzce připomíná výše uvedené řešení naší aplikace. V této aplikaci se určitě budeme chtít inspirovat v principu práce s tabulí a v kolaborativním přístupu.

4.2.3 Výsledek analýzy

Celkovou analýzu popisující všech třicet testovaných aplikaci s detailnějším popisem najdeme v příloze D. Pro další návrh si stanovme sadu vlastností, které jsme nalezli u analyzovaných služeb. Tyto vlastnosti budeme buď chtít zapracovat do aplikace, nebo se jim naopak vyhnout.

[26] technologie pro autorizaci a autentizaci skrze účet třetí strany, například skrze Google účet https://developers.google.com/identity/protocols/OAuth2WebServer

[27] služba pro automatizaci procesů, https://zapier.com/

[28] služba pro komunikaci dislokovaných týmů, https://mural.ly/

- **Pozitivní vlastnosti – prvky k integraci**
  - Kolaborativní přístup – Uživatelé by měli vidět, kdo jiný pracuje s aplikací a jak.
  - Maximální svoboda při práci – Nijak nenutit uživatelům náš pohled na věc.
  - Integrace maximálního počtu třetích stran.
  - Práce s virtuální tabulí.
  - Jednoduchý start v použití aplikace – Eliminovat kroky, kterými musí uživatel projít, než aplikaci začne používat.
  - Maximální důraz na vizualizaci a jednoduché uživatelské rozhraní.
  - Historie a iterace.
- **Negativní vlastnosti – k eliminaci a návrh řešení**
  - Manuální správa – Všechna data je nutné do aplikace vložit ručně a to ještě s velkým úsilím.

    Maximální automatizace.
  - Vynechání sémantiky.

    Vložená data si musí alespoň v nějaké míře držet původní sémantiku.
  - „Modelování vesmíru“ – snaha o pokrytí úplně všech případů užití.

    Zaměřit se na jeden klíčový řešený problém.
  - Vnucování principů a vzorců chování.

    Dát uživatelům volnost.
  - Komplikované ovládání a komplexní přístup.

    Elimimovat počet prvků a entit a zpřístupnit operace ve kvalitním uživatelském protředí.

4.3 První návrh

V této kapitole si popíšeme postup prvního návrhu, jeho vstup, proces a výstup. **Vstupem** je veškeré předešlé úsilí, které zadefinujeme následovně. Chceme navrhnout a následně vytvořit webovou aplikaci, která bude podporovat agregaci různých artefaktů, které označujeme za entity, vyskytujících se na internetu v různých podobách, jak jsme je již definovali v předchozích kapitolách. Tyto entity bude aplikace agregovat, vizualizovat jejich vazby, definovat jejich atributy a zadávat jim sémantiku. Aplikace bude kolaborativní. Prostředí aplikace nazýváme dashboard, bude připomínat tabuli, neboť se jedná o prostředí, které uživatelé již dobře znají a jsou něj zvyklí, případně může pracovat s vhodnými podklady jako je Kanban[29] nebo SWOT[30]. Tyto podklady mohou podporovat agilní vývoj nebo jiný proces. Aplikace se nebude snažit o nahrazení funkcionality jiných aplikací, bude mít jednoduché uživatelské rozhraní, které bude založené na Material Design[31].

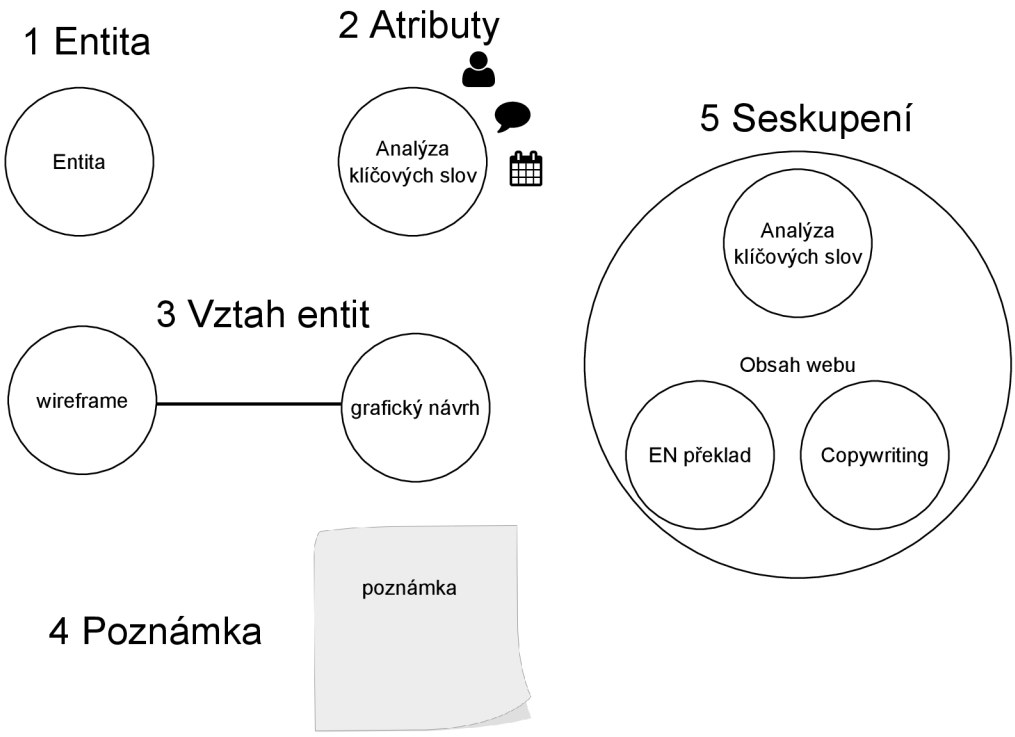
Výstupem návrhu bude interaktivní prototyp (drátěný model) aplikace znázorňující jednotlivé elementy a práci s nimi. Pro účely práce je přiložen ve formátu PDF, který je

[29] vizualizace procesu ve sloupcích, http://www.svetproduktivity.cz/slovnik/Kanban.htm

[30] metoda statické analýzy, vizualizace vstupů ve kvadrantech výhod, slabin, hrozeb a příležitosti, http://www.businessvize.cz/planovani/kde-se-vzala-a-k-cemu-vsemu-je-vlastne-swot-analyza

[31] grafický manuál od firmy Google, https://material.io/guidelines/

v elektronické podobě také interaktivní. Základní snímky z prvního prototypu nalezneme v příloze A. Základem aplikace je vizualizace entity a jejich vazeb, seskupení a poznámky, jak můžeme vidět na následujícím obrázku.

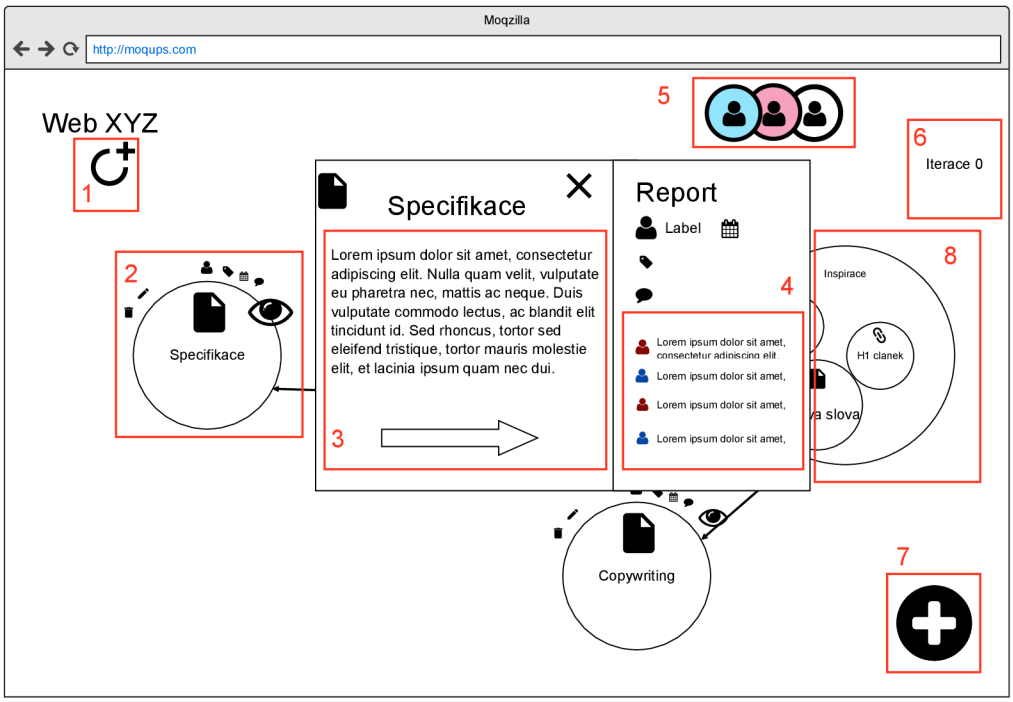


Obrázek 4.1: Prvky vstupující do aplikace

Vztahem entit je myšleno jejich propojení či definovaná závislost nebo následnost. Dále definujeme sadu atributů entity, ale zatím v otevřené podobě, protože tyto atributy ještě nejsou klíčové a budou upraveny po zpětné vazbě. Zatím uvažujeme atributy – uživatel, datum tvorby či editace a popis. Poznámku, jako poslední z elementů, držíme pouze jako zděděný přežitek z principu tabule, který jsme našli ve službě Mural.ly. Zmíněnou velice omezenou sadou elementů dodržujeme přístup jednoduchosti pro první použití, které jsme si definovali výše.

Ukažme si další snímek z návrhu, který ukazuje další prvky. Všimněme si iterace v pravo nahoře, které chceme dodržet z důvodu agilního přístupu. Iterace mají fungovat pro definování milníků při realizaci. Zároveň mezi iteracemi bude stále k dispozici historie, která bude udržovat jednotlivé stavy projektu v různých časových okamžicích v závislosti na změnách, tak jak tento princip známe například z Google Drive.

V detailu entity, který zde vidíme, je znázorněn základní přehled, který následně odkazuje do aplikace dané entity (v tomto případě Google Doc), tedy ukazuje jasně dané rozhraní a konečné body mezi aplikací a externí službou.



Obrázek 4.2: Návrh práce s elementy v aplikaci

Ve výše uvedeném návrhu vidíme:

1. Vytvoření nové iterace.
2. Entitu.
3. Náhled entity.
4. Detail a diskuzi nad entitou.
5. Kolaboraci.
6. Iterace a historii.
7. Přidání nové entity.
8. Seskupení entit.

4.3.1 Zpětná vazba

Jak již bylo uvedeno v plánu návrhu, cílem je výše uvedený první návrh následně diskutovat s odborníky z praxe, tedy potencionálními uživateli. Při rozhovorech byla nejdřív vysvětlena motivace práce. Jako ukázka byla použita zmíněná případová studiie. Po ujištění pochopení problému bylo přistoupeno k získávání zpětné vazby. Ve většině případů byla zaznamenána pozitivní reakce, kdy stejný problém také trápí i dotazované odborníky. Odborníci z praxe, kteří byli osloveni, jsou lidé pracující zejména na vedoucích pozicích v IT, ale zkoumán byl i názor vývojářů nebo marketérů. Kompletní výstup zpětné vazby najdeme v příloze C.

Ukažme si ale myšlenky, které obohacují aplikaci a přidávají nové zajímavé funkcionality nebo pohledy:

- Obarvovat entity.
- Jsou-li entity seskupené, pak podíl barev zobrazit v koláčovém grafu.
- Přiblížení a oddálení tabule (pro přehled projektu).
- Rozdělení tabule do sekcí.
- Automatizace – čtení externích zdrojů jako je Google Drive, email apod.
- Zanořitelnost seskupení.
- Integrace komunikačních kanálů jako abstrakce do jednoho kanálu.
- Záznamy z externích zdrojů zpracovávat ve frontě, tedy postupně přidávat nové záznamy, ty pak odebírat operací „Přidat jako entitu“ nebo „Odstranit ze fronty“.
- K seskupení se chovat jako další k entitě, tedy seskupení je pouze entita obsahující entity.
- Iterace.
- Přejmenovat tabuli na dashboard.
- Notifikace o změnách entit.
- Komentáře entit, není potřeba diskuze.
- Kreslení – barvy a zvýranění.

Tyto myšlenky a potřeby zapracujeme do dalšího návrhu.

4.4 Druhý návrh

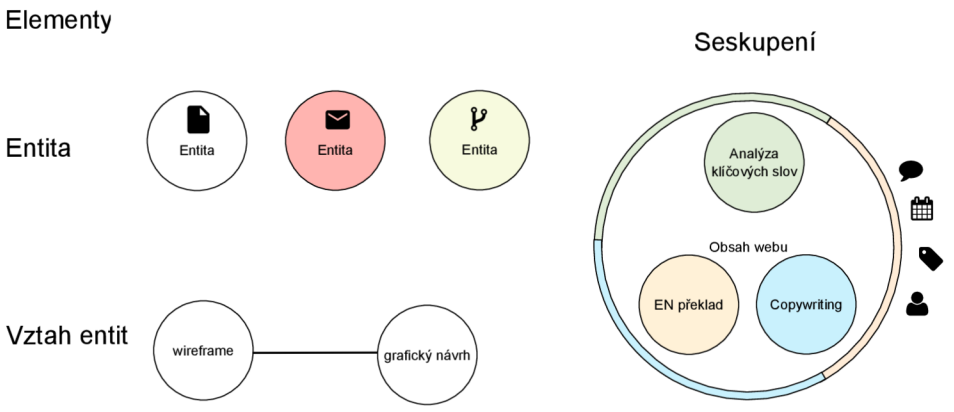
Ve druhém návrhu jsme zapracovali zmíněné myšlenky pramenící z diskuze s odborníky. Základní rozdíly můžeme vidět v grafickém znázornění entit s barevným zvýrazněním, kde seskupení mohou dědit barevné znázornění z objemu svých entit a zejména možnost vnořit entity do jiných entit. V první verzi se žádná grafická reprezentace vůbec neuvažovala. Další silný funkční prvek můžeme vidět v oblasti integrace externích služeb, ne pouze z hlediska externích adres, ale v podobě přímé integrace. Tedy uvažujme integraci služeb jako jsou Google Drive, emailové schránky (skrze IMAP) nebo Git repozitáře. Základní změny vidíme na následujících snímcích.

Druhý návrh, který reprezentuje dílčí výstup této diplomové práce, je přiložen ve formátu PDF[32]. Základní snímky z druhého prototypu nalezneme v příloze B.

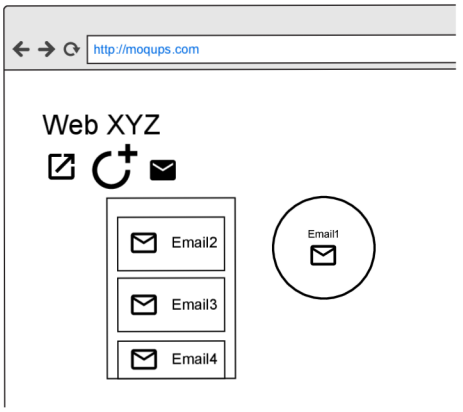
Jelikož jsme stanovili vývoj a dodání výsledné služby dle metodiky LeanSD, pak stanovme klíčové funkcionality v diagramu případů užití B.4. Tyto vlastnosti má aplikace poskytovat uživateli po první implementační verzi odpovídající MVP[33], která je součástí této diplomové práce. Tyto funkcionality, jsou klíčovou oporou implementace a musí být splněny. Další funkcionality jsou vítány, ale nejsou nutnou podmínkou pro MVP.

[32] v elektronické podobě interaktivní, https://goo.gl/VBmX6v

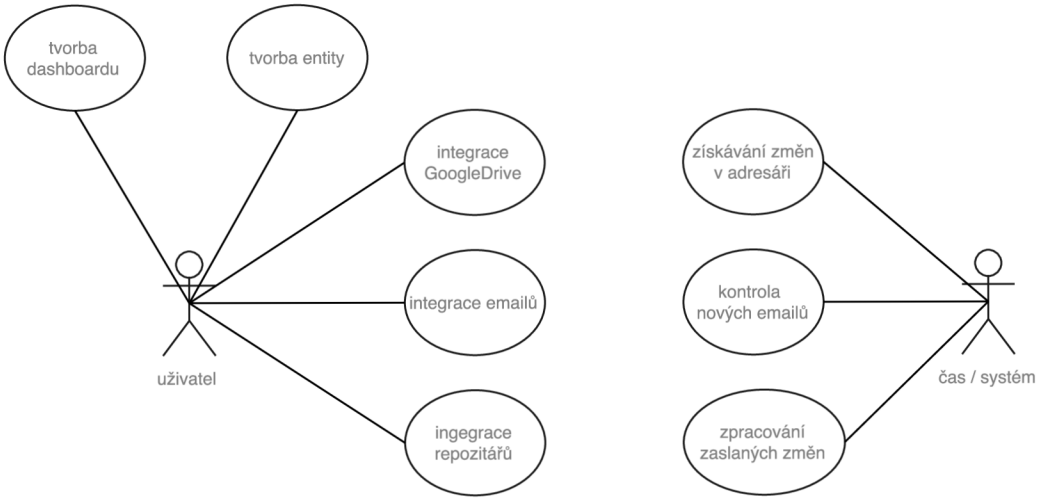
[33] Minimum Viable Product, produkt s nejmenší možnou funkcionalitou pro validaci konceptu [20]



Obrázek 4.3: Druhý návrh elementů aplikace



Obrázek 4.4: Integrace emailů do aplikace



Obrázek 4.5: Případy užití první verze aplikace

4.5 Technologie

Jelikož se aplikace jeví jako velice heterogenní z hlediska plánovaných technologií, je vhodné věnovat jisté úsilí pro popis technického návrhu. V následující kapitole budou tedy popsány technické aspekty tvořené aplikace. Budou popsány jednotlivé technologie s vysvětlením důvodu, způsobu využití a přidané hodnoty. Dále budou technologie znázorněny v architektuře systému, tedy jak jednotlivé komponenty spolu interagují. Tyto komponenty budou pracovat se sadou dat, jejichž struktura bude popsána v podsekci datového návrhu.

Definujme základní technologické otázky, pro které hledáme řešení v této kapitole:

- Webový server
- Serverová aplikace
- Databáze
- Jazyk na straně klienta
- Práce s grafickými elementy
- Technologie kolaborace
- Balíčkovací nástroje

Ve výsledku se má jednat o webovou aplikaci, která bude dostupná skrze webový prohlížeč. Jelikož se má jednat o pracovní aplikaci, uvažujme pouze použití na klasických počítačích a ignorujme mobilní platformy a responzivní řešení. Webové aplikace jsou aplikace typu klient-server, kde musíme uvažovat technologie na obou stranách. Pro jednoduchost popisu předpokládejme základní znalost webových technologií, objektového návrhu, jazyka Python a relačních databází.

4.5.1 Technologie serveru

Na straně serveru potřebujeme pokrýt zejména technologie webového serveru, serverové aplikace a databáze.

Serverová aplikace

Pro vývoj webových projektů je velice populární jazyk PHP nebo JavaEE, ovšem v tomto případě byl ale vybrán jazyk Python s webovým frameworkem (aplikačním rámcem) *Django*[34]. Při popisu práce s tímto frameworkem budeme vycházet zejména z oficiální dokumentace projektu [3]. Výběr frameworku vychází z implementované výborné podpory techniky *ORM* – Objektového relačního mapování [1] a silné podpory nejrůznějších specifických technologií a řešených problémů, ať už přímo ve frameworku Django, tak v jazyce Python. Framework Django podporuje obě verze jazyka Python a to jak 2.7 tak i ve verzi 3. Ale z důvodu vyšší rozšířenosti jazyka Python2.7, zejména v hotových modulech, jsem se rozhodl pro použití verze 2.7. Framework Django používá architekturu MVC[35] s šablonovým systémem Jinja2. Jako velice silné pozitivum frameworku Django se ukazuje právě přesunutí vývojáře

[34] https://www.djangoproject.com/
[35] http://djangobook.com/model-view-controller-design-pattern/

od databáze až na úroveň objektového mapování a správa aplikace skrze sadu nástrojů pro správu. Naopak se ukázalo, že slabinou je technologie Ajax. Detailní struktura frameworku a individuální implementace bude popsána v daších kapitolách. Pro výběr jazyka Python přispěly dva argumenty. Prvním je silný tržní podíl, a tedy řada existujících modulů pro tento jazyk, které řeší mnoho konkrétních situací. Například modul pro práci s IMAP, modul pro šiftování apod. Druhým argumentem byl populární principy DRY (Don't repeat yourself), který výrazně zjednodušuje a urychluje vývoj. Jedná se o princip psaní kódu pro jeho maximální znovupoužitelnost a rozšiřitelnost, což ústí právě v množství existujících modulů.

Webový server

Použijeme webový server Apache2[36]. Jedná se o otevřená a svobodný webový server, který podporuje protokol HTTP ve verzi 1.0, 1.1 a v omezené míře i HTTP2. Ač HTTP2 má přinášet množství zejména výkonnostních výhod, tak z důvodu relativně nové technologie (specifikace v RFC byla publikována v květnu 2015) a omezené podpoře využijeme HTTP ve verzi 1.1. S výběrem webového serveru souvisí také výběr operačního systému, který uvažuji v této práci a zejména v manuálech na spuštění a rozšíření aplikace. Aplikace byla testována na linuxových distribucích z rodiny RHEL, tedy na distribucích Fedora 24 a výše nebo CentOS 7.2 a výše. Na těchto distribucích nenajdeme server Apache pod tímto názvem, ale pod pojmem (a balíčkem) httpd.

Výběr vychází z předchozí zkušenosti s tímto serverem, neboť je propojitelný s mnoha serverovými aplikacemi. Ovšem zde narážíme na jednu zajímavou odlišnost propojení jazyka Python a serveru Apache, a to v použití technologie WSGI (Web Server Gateway Interface[37]), kdy aplikace je propojena se serverem skrze rozhraní. Jednoduše řečeno, musí být definován právě jeden skript, který je použit pro zpracování příchozích požadavků na server.

Databáze PostgreSQL

Jak již bylo řečeno, framework Django nabízí vlastní ORM řešení[38], objektové relační mapování, kde aplikace je zcela odstíněna od relační databáze a k datům se přistupuje jako k objektům, definice databázových tabulek pak odpovídá jednotlivým třídám. To vše se zachováním vlastností objektového návrhu, jako je například dědičnost. Zajímavý je také fakt, že aplikační vrstva přebírá řadu databázové logiky. To znamená, že například požadavek na unikátní data v sloupci neřeší unikátním klíčem na úrovni databáze, ale kontroluje tuto vlastnost ještě před persistencí dat na úrovni aplikace.

Tyto vlastnosti dávají velkou svobodu při výběru a implementaci databáze. Musíme se jen omezit na skupinu relačních databází, ale následná implementace spočívá pouze ve výběru ovladače v aplikaci a o zbytek se postará framework Django.

Z tohoto důvodu je pro vývoj aplikací ve frameworku Django typické, že se začíná s vývojem nad databází SQLite[39] a v jisté pokročilejší fázi se přechází na jiné databáze vyšší úrovně. V tomto případě volíme databázi PosgreSQL[40] a to ze dvou důvodů. Očekáváme potřebnou práci s objekty JSON na úrovni datatabáze, což PostgreSQL ovládá velice dobře a očekáváme přínos v implementaci klauzule DISTINCT ON.

[36] https://httpd.apache.org/
[37] http://wsgi.readthedocs.io/
[38] https://docs.djangoproject.com/en/1.11/topics/db/models/
[39] https://www.sqlite.org/
[40] https://www.postgresql.org/

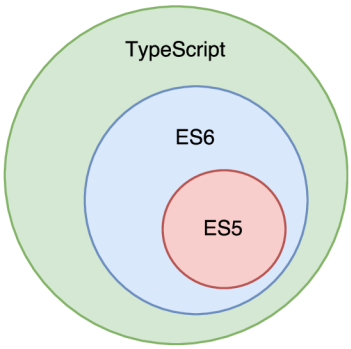
Dle měření jsou rozdíly mezi rychlostmi databází relativní marginální [19], tedy rychlost se neukazuje jako hlavní argument pro výběr. Výběr databáze byl v tomto případě veden zejména předchozí zkušeností a širší nabídkou funkcionalit a možností ze strany PostgreSQL.

4.5.2 Technologie klienta

Pokud se vyhneme již téměř zapomenutým technologiím jako Adobe Flash nebo Java Applety [13], pak nám samozřejmě zbývají na straně klienta typické technologie HTML, CSS a JavaScript s rozšířením v podobě Twitter Bootstrap[41], Less[42] nebo jQuery[43]. Tyto základní technologie rozšiřujme následujícími nástroji a technologiemi.

TypeScript

Při výběru technologií je zřejmé, že se bude jednat o hodně práce na straně klienta a že bude potřeba dobře pracovat s technologií JavaScript. JavaScript je prototypově orientovaný jazyk, nad kterým firma Microsoft v roce 2012 vydala nadstavbu TypeScript [14]. TypeScript přidává do JavaScriptu statické typování a prvky objetově orientovaného jazyka. Konkrétní rozdíly a jejich implementaci mezi TypeScriptem a JavaScriptem budeme rozebírat v kapitole zabývající se implementací, ale je potřeba zavést existenci TypeScriptu do kontextu standardů jazyka JavaScript. Pro standardizaci JavaScriptu se využívá standard ECMAScript, který spravuje organizace ECMA International[44]. Ten přišel v roce 2011 s verzí 5 a v roce 2015 s verzí 6. Obě verze přidávají sadu nových možností a konstrukcí, ale co je důležité, všechny tyto konstrukce zajišťuje právě TypScript, jak je vidět z následujícícho obrázku.



Obrázek 4.6: Znázornění pozice jazyka TypeScript

[41] HTML, CSS a JS aplikační rámec pro tvorbu webových stránek, http://getbootstrap.com/
[42] preprocesor pro CSS, http://lesscss.org/
[43] JavaSscriptový aplikační rámec, https://jquery.com/
[44] https://www.ecma-international.org/

SVG

Vzhledem k faktu, že se má jednat o aplikaci, která má manipulovat s grafickým objekty, skrze které bude vizualizovat různá data, je nutné se zamyslet, jakou formou takové grafické objekty budeme reprezentovat. Jako zcela ideální řešení se ukazuje vektorový grafický formát SVG a to z několika následujících důvodů:

- SVG je definován pomocí XML.
- Plně pokrývá potřeby aplikace.
- Je snadno napojitelný na další technologie jako je JavaScript a CSS.
- Má silnou podporu v prohlížečích.
- Jedná se o moderní rozvíjející se technologii zejména v souvislosti s webovým vývojem.
- Má silnou podporu ze strany vývojářů, neboť existuje mnoho předpřipravených řešení, jak pracovat s SVG v prohlížeči.
- Jedná se o XML data, která se dají snadno transformovat do JSON struktury nebo ukládat v databázi.

Jak bylo zmíněno, pro SVG existuje řada existujících knihovních řešení, které stačí pouze použít. Pro práci s SVG se nabízely následující alternativy (kromě vlastního řešení):

- SVG.js[45]
- Snap.svg[46]
- Raphael.js[47]

Všechny tři zmíněné možnosti jsou v aktivním vývoji, což je pro výběr technologie výrazně důležité, všechny nabízí velice podrobnou a aktuální dokumentaci. Na základě rychlostních testů[48] a vyzkoušené demo aplikace jsem ve výsledku vybral variantu knihovny *SVG.js*. Silný argument pro tuto variantu je fakt, že má rozhraní popisující chování aplikace v jazyku TypeScript, kterou plánuji použít pro imlementaci a modulární přístup. Aplikace v základu poskytuje pouze základní práci s SVG a pokročilé operace se volí formou modulů, tedy výsledná aplikace je menší a rychlejší.

[45] http://svgjs.com/
[46] http://snapsvg.io/
[47] http://dmitrybaranovskiy.github.io/raphael/
[48] testy dostupné na http://svgjs.com/

Kolaborace

Při návrhu jsme aplikaci definovali jako kolaborativní. Tento přístup je uživateli velice dobře znám například z aplikací Google Docs, kde spolupráce nad jedním dokumentem je dobře vizualizovaná a uživatel vidí operace ostatních účastníků v reálném čase. Stejného efektu chceme dosáhnout i v tomto případě.

Opět se snažíme najít již existující podpůrná řešení a momentálně se nabízí následující živé projekty zabývající se tímto tématem. Nacházíme:

- DerbyJS[49]
- SwellRT[50]
- TOGETHERJS[51]
- Google Realtime API[52]

Všechny zmíněné projekty splňují požadavky živého vývoje, licenční dostupnosti i dokumentace, ale zde přichází zásadní důvod výběru. Zcela určitě budeme chtít integrovat v dalším vývoji dokumenty z Google Drive, a tedy volíme v tomto případě technologii od Google, neboť nechceme tříštit pozornost mezi více rozdílných technologií. Google Realtime API poskytuje jak kolaborativní řešení, tak i principy uchovávání a sdílení dokumentů skrze Google Drive, které přináší velkou přidanou hodnotu.

Důležitým faktorem pro výběr Google Realtime API byl také fakt, že tuto technologii využívá řada zavedených aplikací, jako je *draw.io* nebo *Gantter* [17].

Balíčkovací systém

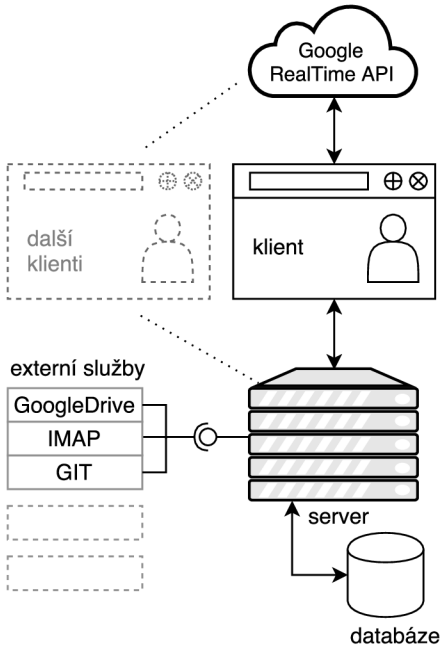
Vzhledem k výběru řady externích technologií, které integrujeme do výsledné aplikace je vhodné zavést nějakou technologii pro správu balíčků. Pro Python se používá populární balíčkovací systém *pip*[53]. Tento balíčkovací systém je zaveden a je téměř nepsaným standardem jazyka Python.

V případě světa JavaScriptu není situace ještě zcela vyřešená, ale pro použití této aplikace instalujeme všechny externí balíčky skrze nástroj *Bower*[54]. Ten funguje jako autoritativní seznam repozitářů jednotlivých balíčků. Každý modul, který chceme skrze Bower instalovat, musí být v aplikaci zaregistrován se svými údaji a s odkazem na repozitář (typicky Git), který při instalaci stahuje a instaluje jeho zdrojové kódy. Velkou výhodou tohoto přístupu je, že můžeme držet zdrojové kódy externích modulů mimo aplikaci (i z hlediska vývojového repozitáře) a ve výsledných instancích aplikace (produkční či testovací prostředí) pouze pak použijeme tyto balíčkovací nástroje pro instalaci externích knihoven.

[49] http://derbyjs.com/
[50] http://swellrt.org/
[51] https://togetherjs.com/
[52] https://developers.google.com/google-apps/realtime/overview
[53] https://pypi.python.org/pypi/pip
[54] https://bower.io/

4.6 Architektura

Jak je patrné z předešlé kapitoly, při vývoji aplikace budeme pracovat s mnoha komponentami, které popíšeme v této části. Obecně můžeme aplikaci rozdělit na základní čtyři části.



Obrázek 4.7: Znázornění architektury aplikace

V první části se jedná o klasickou serverovou aplikaci, v našem případě realizovanou v jazyce Python s frameworkem Django. Tato aplikace má běžné rysy MVC architektury[55], která rozděluje aplikaci na datovou (Model), prezentační (View) a řídicí (Contoler) část. Tuto architekturu si rozebereme později. Tato serverová aplikace zpracovává HTTP požadavky ze strany klienta. Pro persistenci dat používá relační databázi. Tuto vrstvu nazývejme anglickým pojmem *backend*.

Druhým blokem aplikace je strana webového prohlížeče, kde je skrze SVG a JavaScript realizované celé uživatelské rozhraní aplikace ve formě webové stránky. Tato část se skrze HTTP požadavky kooperuje s backend vrstvou systému, kde se dotazuje na data nebo je zasílá k persistenci. Tuto vrstvu nazývejme *frontend*.

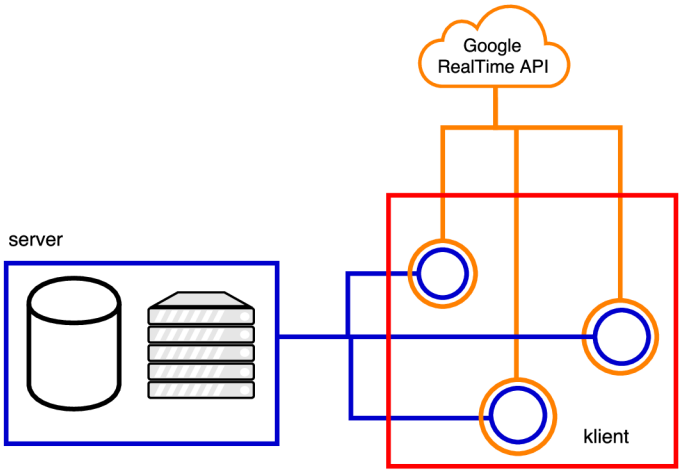
Třetím důležitým prvkem struktury jsou služby třetích stran, které komunikují s backend částí skrze API[56]. Každá aplikace vyžaduje individuální přístup a vlastní implementanci, tedy pro každou z integrovaných služeb třetí strany je v backend vrstvě vytvořené samostatné API. Jak uvidíme v datovém modelu, je silná snaha o unifikaci tohoto přístupu, aby integrace dalších služeb byla co nejjednodušší a nerozbíjela strukturu aplikace.

Čtvrtým a nejvýraznějším prvkem aplikace, který odlišuje aplikaci od běžných webových systémů, je část zajišťující kolaboraci mezi uživateli v reálném čase. Frontend vrstva komunikuje se službou Google Realtime API, která zajišťuje distribuci dat nad stejným modelem v reálném čase dalším uživatelům. Tato komponenta pracuje s podmnožinou

[55] http://djangobook.com/model-view-controller-design-pattern/
[56] Application Programming Interface – rozhraní pro napojení externích aplikací

dat celé aplikace a to právě takovou, která je nutná k dosažení datové a vizuální integrity mezi jednotlivými klienty, kteří pracují se stejným modelem. Modelem v tomto případě rozumějme jednu virtuální tabuli (dashboard), kterou jsme si zadefinovali v předchozích kapitolách zabývajících se návrhem. Z hlediska Google Realtime API je model jeden strukturovaný dokument (v podobě JSON), který se ukládá do úložiště Google Drive pod účtem zakládajícího uživatele, což nám dále umožní sdílení dokumentu. Zároveň to ale předpokládá zaregistrování aplikace do platformy Google Drive jako aplikaci třetí strany.

Zmíněná podmnožina dat, která se sdílí jak mezi backend částí aplikace tak i do Google Realtime API, je vysvětlena v následujícím obrázku.



Obrázek 4.8: Znázornění datových množin v aplikaci

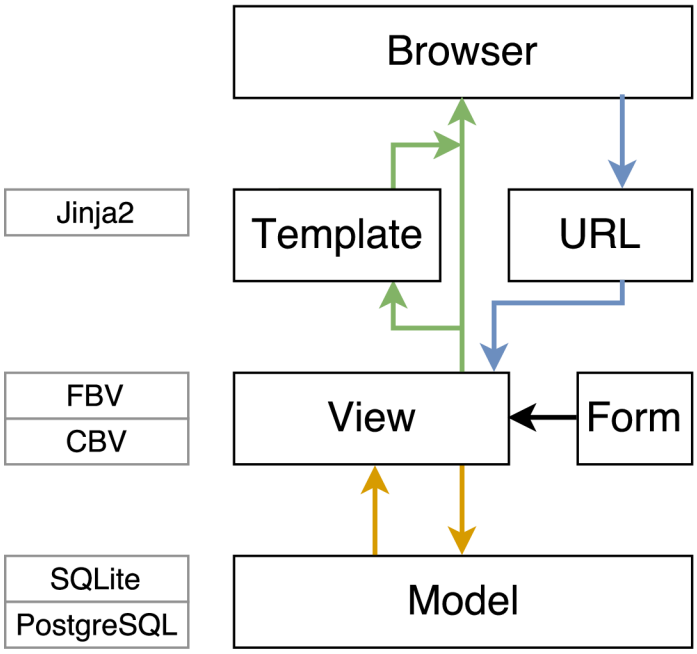
Společným jmenovatelem obou úložišť je frontend vrstva (červeně), která data (v našem případě zejména entity) vizualizuje a komunikuje jak s backend vrstvou (modře), tak s Google Realtime API (oranžově). Jak bylo řečeno, Google Realtime API dostává pouze podmnožinu dat, kterou potřebuje distribuovat mezi další klienty. Podrobnější data o jednotlivých entitách má pak backend vrstva uložené v databázi. Obě úložiště samozřejmě identifikují entitu stejným identifikátorem. Konkrétně si můžeme situaci představit následovně.

Nechť existuje entita, která má unikátní číselný identifikátor. Tato entita má atributy *názvu*, *barvy* a *popisu*. Pro distribuci mezi klienty je potřebná pouze podmnožina, a to *název*, *identifikátor* a *barva*. Tato podmožina se tedy posílá do Google Realtime API. Popis entity, který uživatel dostává až na požádání, se uchovává pouze v databázi backend vrstvy.

4.6.1 Architektura backend vrstvy

Na straně backend vrstvy vychází architektura z frameworku Django. Ten drží architekturu MVC, tedy rozdělení aplikace mezi následující části. Dále uváděné informace čerpáme z oficiální dokumentace projektu Django [3].

- Model (datová část)
- Controller (řídicí část zpracovávající požadavky)
- View (prezentační vrstva)



Obrázek 4.9: Znázornění MVC architektury backend vrstvy

Tuto architekturu demonstrujme následujícím obrázkem.

Z výše uvedeného obrázku vidíme[57], že struktura je mírně odlišná. A to ze dvou důvodů. První je rozdílná terminologie, kdy MVC pouze definuje strukturu, ale konkrétní termonologie pak závisí na jednotlivých implementacích frameworku. V tomto případě můžeme mluvit o MTV, tedy *Model-Template-View*[58]. Podobné odskočení od terminologie můžeme vidět i v dalších frameworcích. Například framework Nette také drží stejnou strukturu, ale označuje ji jako MVP – *Model-View-Presenter*[59].

Druhým rozdílem je rozšíření základní struktury o část *URL* a *Form*. Část *URL* se stará o správné směrování požadavků na základě jejich URL adresy. Modul *Form* se stará o definici, validaci a případně i persistenci dat z formulářů. Jedná se o typický dopad přístupu DRY, který jsme popisovali v předešlých kapitolách.

V obrázku jsou barevně rozlišené toky dat vzhledem ke zpracování HTTP požadavků. Modrá barva reprezentuje příjem požadavku, zelená pak generování odpovědi. Všimněme si, že při odpovědi se nemusí pracovat s modelem, tedy s databází. Zároveň se také může obejít šablonovací systém a přímo odpovídat z *View* vrstvy. Toho využíváme například při Ajaxové komunikaci, kdy se mohou zasílat data ve formátu JSON.

V levé části obrázku vidíme použité technologie každé vrstvy. Jak už bylo zmíněno v popisu frameworku, Django díky objektovému mapování dat může bez problému pracovat s různými databázemi. Při vývoji je typické použití SQLite pro počátky vývoje a následný přechod na „pokročilejší“ databáze, v tomto případě PostgreSQL.

Na úrovni *View* rozlišuje Django dva základní přístupy. *View* může být implementováno jako sada funkcí, tedy se pak bavíme o View založeném na funkcích, tedy FBV – *Function*

[57] čerpáno z https://www.slideshare.net/ssuserbf9115/django-introduction-amp-tutorial-pdf

[58] View v případě frameworku Django znamená řídící vrstvu, ne prezentační, tak jako u jiných frameworků. Prezentační vrstva je zde nazývána Template

[59] https://doc.nette.org/cs/0.9/model-view-presenter

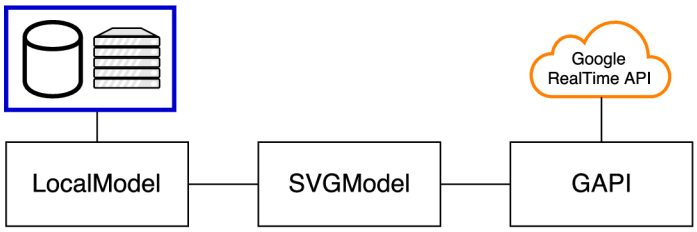
Based View. Tento přístup má výhody v jednoduchosti implementace, čitelnosti a možnosti rozšíření skrze dekorátory.

Proti tomu existuje tzv. CBV — *Class Based View*, kde každé *View* je třída a můžeme tak využít všech vlastností plynoucí z objektové orientace. Silným argumetem je také předdefinované rozdělení metod pro jednotlivé typy HTTP požadavků (pro POST požadavek máme metodu post() apod.). Komunita kolem vývoje frameworku nedochází k explicitním závěrům a využití těchto přístupů pak záleží na vývojáři. Konkrétní příklady a rozdíly si ukážeme v kapitole 5, která se zabývá implementací.

Poslední zmíněnou technologií je šablonovací systém *Jinja2*. Tento systém je populární i mezi dalšími webovými frameworky založenými na jazyce Python (například Flask). Cílem šablonovacích systémů je rozdělit logiku aplikace a uživatelské rozhraní. Tedy v šablonách definujeme skrze jazyky HTML a CSS grafickou podobu aplikace.

4.6.2 Architektura frontend vrstvy

Jak už bylo zmíněno, frontend vrstva má tři základní úlohy. Vizualizovat data zejména skrze SVG formát, komunikovat s backend vrstvou a kooperovat s Google Realtime API. Zároveň jsme vybrali k použití rozšíření TypeScript, které lépe podporuje objektový přístup. Tyto úlohy demonstrujme následujícím obrázkem.



Obrázek 4.10: Znázornění MVC architektury backend vrstvy

Z výše uvedeného obrázku vidíme, že na straně modelu definujeme pro každou z rolí samostatnou třídu. Každá ze tříd má pevně definovanou zodpovědnost. Třída LocalModel se stará o interakci s backed vrstvou a využívá zejména technologii Ajax. SVGModel využívá knihovnu SVG.js pro práci s SVG grafickými prvky. A třída GAPI pracuje s Google Realtime API. Tato třída má na starost nejen kolaboraci, ale také autentizaci uživatelů a tvorbu souborů na Google Drive. Konkrétní implementaci těchto tříd a další detaily budeme diskutovat v kapitole 5, která se zabývá implementací.

Architekturu posledních dvou částí, tedy API a Google Realtime API, budeme popisovat současně s implementací, neboť je s ní úzce spojena.

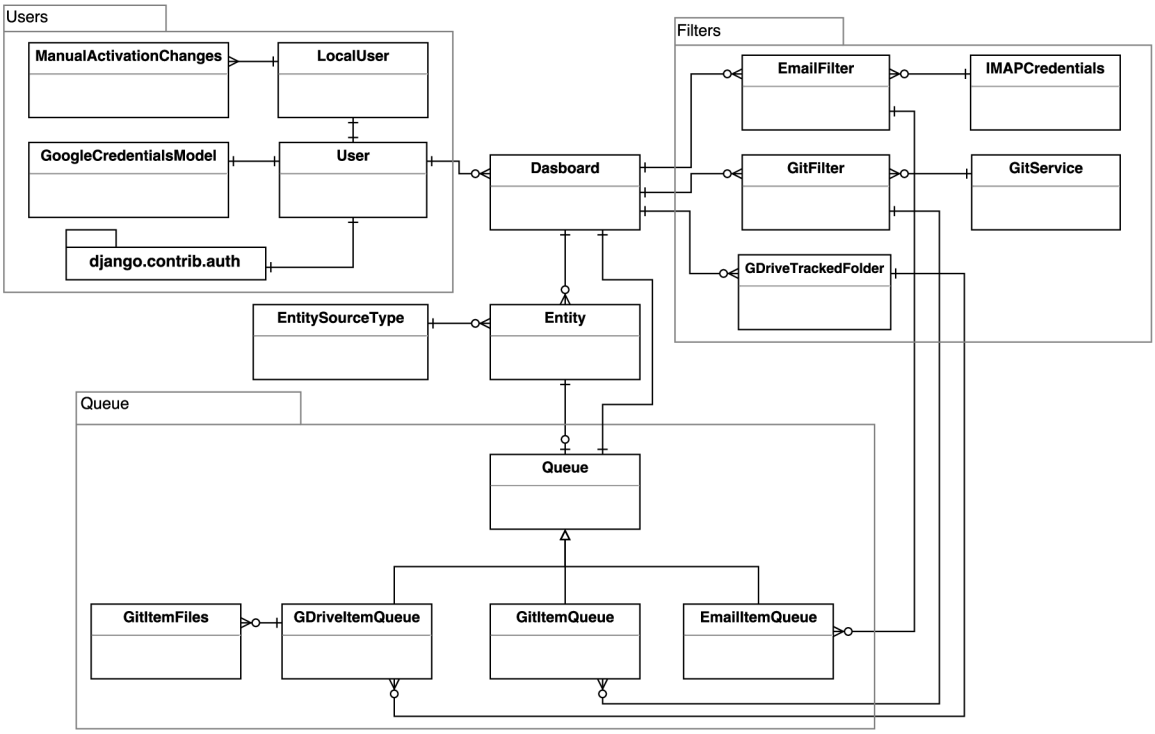
4.6.3 Datový model

Z hlediska datového modelu se budeme bavit o dvou oblastech, které uchovávají data aplikace. Jedná se o databázi backend vrstvy a strukturu dat v Google Realtime API.

Datový model databáze

Datový model databáze demonstrujme následujícím diagramem tříd. Vzhledem k technice ORM, která již byla zmíněna v předchozích kapitolách, by bylo irelevantní pracovat s koncepčním modelem jako například ERD, protože samotná cílová realizace v dané databázi

záleží na frameworku. Zároveň uvažujme vytvořené instance tříd za persistentní (na úrovni databáze mluvíme o řádcích v tabulce), neboť se skrze ORM přímo ukládají do databáze.



Obrázek 4.11: Diagram tříd reprezentující datový model

Ve výše uvedeném diagramu můžeme vidět čtyři základní bloky. První blok *Users* se stará o práci s uživateli. Zde pracujeme s již vestavěným balíčkem frameworku Django, který plně pokrývá práci s uživateli systému, tedy například obnovu hesla, přístupové údaje apod.

Jak již bylo popsáno v předchozích kapitolách, hlavní třídou, na kterou budou navázané další části aplikace, je třída *Dasboard*. Tato třída obsahuje entity. Entity mohou být různých typů, a to právě definuje třída *EntitySourceType*. Pokud budeme uvažovat nový zdroj entit (tedy pravděpodobně integraci další služby), pak musíme vytvořit novou instanci třídy *EntitySourceType*.

Další dva bloky *Filters* a *Queue* pracují s externími službami a jejich integrací. Pro každou službu existuje samostatná třída, která určuje filtr, dle kterého se mají získávaná data zařazovat do fronty. Pro lepší představu si problematiku ukažme na příkladě.

Chceme do aplikace integrovat emaily. Vytvoříme tedy třídu *EmailFilter*, v jejíž instancích budeme definovat kritéria, podle kterých chceme vybírat emaily k integraci. Tedy například chceme do aplikace integrovat pouze emaily obsahující vybrané klíčové slovo a přicházející z určené emailové adresy. Všechny tyto informace drží instance třídy *EmailFilter* s třídou *IMAPCredentials*, která udržuje šifrované přihlašovací údaje ke schránce. Pokud při čtení emailové schránky narazíme na email, který splňuje kritéria, pak vytváříme objekt ze třídy *EmailItemQueue*, která dědí z třídy *Queue*. Tento objekt plní roli položky zařazené do fronty a čeká na operaci uživatele, který ji může buď zavést jako entitu nebo odstranit. Pokud se uživatel rozhodne vytvořit z položky entitu, pak se vytváří objekt třídy *Entity*. Ta ale stále drží odkaz na původní položku ve frontě, neboť právě tato položka drží

ve svých podtřídách všechny potřebné dodatečné informace o vzniklé entitě. Například pro zmíněný email by držela jeho kopii.

Kolaborativní datový model

V další podkapitole si ukažme datovou strukturu kolaborativního modelu. Jak již bylo ukázáno v části zabývající se architekturou, rozdělujeme data mezi lokální serverovou databázi a kolaborativní datový model realizovaný skrze Google Realtime API [5]. Tento model má k dispozici pouze podmnožinu dat, neboť je až sekundárním zdrojem. V případě odstavení služby Google Realtime API musí aplikace fungovat dále, a to i bez kolaborativního modelu. Tato podmnožina obsahuje tedy jen data nutná k vykreslení entit v reálném čase mezi klienty. Další podrobnosti o entitě nebo dodatečné informace se získávají z hlavní databáze skrze Ajax požadavky. Google Realtime API nabízí tři vestavěné kolaborativní datové struktury:

- Kolaborativní textový řetězec
- Kolaborativní seznam
- Kolaborativní mapu[60]

Z výše uvedených kolaborativních datových struktur se pro naši aplikaci hodí nejlépe kolaborativní mapa. Prakticky se jedná o asociativní pole, kde každý prvek je uvozen unikátním klíčem, který může být řetězec, nebo číslo. Jedná se tedy o ideální strukturu, neboť uchovávané objekty budou odpovídat jednotlivým SVG elementům, nad kterými se provádí operace. Celkem jsou v aplikaci použity dvě takové kolaborativní mapy. Jedna mapa pro globální data dasboardu (nastavení pozadí, velikosti apod.) a druhá pro jednotlivé prvky.

[60] abstraktní datový typ známý jako asociativní pole nebo vyhledávací tabulka

Struktura jednotlivých prvků je nejlépe demonstrovatelná konkrétním příkladem. Tyto dva vnořené JSON objekty udávají grafickou podobu entity – kolečko a její obsah – textový název.

```
{
    "SvgjsG1021": - identifikátor v~rámci SVG modelu
    {
        data : Object - interní data
        groupID : "SvgjsG1021" - skupina do které tento element patří
        id : "SvgjsText1019" - identifikátor v~rámci SVG modelu
        msg : "moved" - poslední operace
        pos : Object  - obsahuje údaje o~pozici SVG elementu
        type : "text" - typ elementu
    }
    "SvgjsCirc1001":
    {
        data : {
            entity_id: 1 - databázový identifikátor
            fill: "#fafafb" - barva výplně
            radius: 100 - velikost
            stroke: "#21ba45" - barva ohraničení
            type: "email" - typ enetity
        }
        groupID : "SvgjsG1021"
        id : "SvgjsCirc1001"
        msg : "moved"
        pos : Object
        type : "circ"
    }
}
```

Kapitola 5

Implementace

V následující kapitole budeme popisovat implementaci jednotlivých částí a technologií. Budeme postupovat dle jednotlivých logických celků aplikace. Cílem je identifikovat klíčové body implementace, nikoliv triviální konstrukce jako například HTML, CSS nebo Twitter Bootstrap. Při popisu budeme předpokládat základní znalost HTML, CSS, XML, JavaScriptu, objektového návrhu, jazyka Python a relačních databází. Při každé části implementace budeme zdůrazňovat jakou přidanou hodnotu dodává implementace vybrané části do aplikace.

5.1 Implementace frontend vrstvy

Na straně frontend vrstvy budeme popisovat implementaci TypeScriptu, SVG.js, Google Realtime API a jejich integraci do celé aplikace dle výše uvedeného návrhu.

5.1.1 TypeScript

Jak již bylo řečeno v předešlých kapitolách, TypeScript (dále jen TS) je rozšíření JavaScriptu (dále jen JS), které v roce 2012 uvolnila firma Microsoft pod licencí Apache 2.0, což umožnilo volné šíření a tak velkou oblibu mezi vývojáři. Samotný TS kód je nutné přeložit do JS skrze balíček *typescript*. Platí tedy, že každý validní JS kód je zároveň validním TS kódem. Při implementaci tedy musíme vždy spouštět překlad, což většina vývojových prostředí realizuje automaticky při každé změně. Můžeme pak uchovávat jako zdrojový kód pouze TS a při každé instanci aplikace (nainstalování na server nebo jiné prostředí) TS překládat na JS (například skrze nástroj Gulp). Nebo můžeme udržovat součástí projektu (v repozitáři) oba soubory, které nám poskytuje vývojové prostředí. V tomto případě byla zvolena druhá varianta a to z důvodu jednoduchosti, neboť nemusíme vytvářet skripty pro překlad. Existuje řada rozdílů mezi TS a JS od primitivních až po složité konstrukce. Výčtem můžeme zmínit následující [2]:

- Nutnost překladu
- Anotace datových typů
- Rozhraní
- Třídy
- Moduly

Základním rozdílem mezi TS a JS je anotace datových typů a tedy možnost provádět typovou kontrolu při kompilaci. Samozřejmě musíme stále mít na paměti, že se jedná ve výsledku o JS kód, tedy tato anotace je nepovinná, ale zjednodušuje vývoj a eliminuje řadu chyb.

V následující ukázce kódu vidíme nejdříve čistý JavaScript a následně realizaci v TypeScriptu. Anotace datových typů jsou zřejmé a tedy i s nimi spojené vlastnosti.

```
function add(a,b) {
    return a + b;
}

function add(a: number, b: number): number {
    return a + b;
}
```

V aplikaci této vlastnosti využiváme například definováním typů jednotlivých prvků SVG.

```
type Rect = svgjs.Rect;
type Group = svgjs.G;
type Doc = svgjs.Doc;
```

Další důležitou vlastností TS je pohodlnější podpora nových verzí standardu ECMAScript, neboť TypeScript překládá svůj kód do vybrané cílové verze ECMAScriptu. Programátor tedy může využívat množství pokročilých konstrukcí jako například lambda funkce nebo rozhraní a stále přitom výsledná aplikace bude podporovat starší standardy.

S tím je spojena podpora mnoha typicky objektových přístupů. V aplikaci používáme běžné třídy TypeScriptu a skrze rozhraní importujeme chování externích modulů, v našem případě zejména SVG.js. Celkově tedy definujeme tři základní třídy aplikace:

- `LocalApp` – Třída komunikuje s lokální databází, většinou skrze požadavky Ajax.
- `AppGapi` – Třída pro komunikaci s Google Realtime API.
- `SvgModel` – Třída manipulující s SVG.

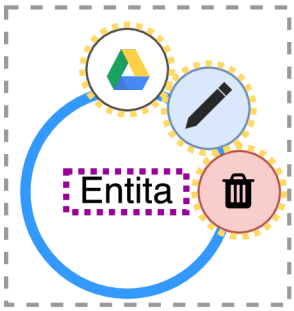
Jednotlivé výše uvedené třídy budou blíže popsány v dalších kapitolách.

Při vývoji aplikace v TypeScript je potřebné definovat konfigurační soubor tsconfig.json, který udává překladači cílovou verzi ECMAScript, zdroje externích knihoven a další nastavení. V tomto případě bylo potřeba definovat cestu k externím knihovnám, jejichž datové typy a třídy byly použity. Pro psaní knihoven je důležité definovat tzv. d.ts soubory, kde se definuje rozhraní tříd, pro další použití. Tento soubor nebylo potřeba definovat, neboť jsme netvořili knihovnu.

5.1.2 SVG a SVG.js

Cílem implementace SVG a připojené knihovny SVG.js je vizualizace a manipulace s grafickými prvky, které vizualizují entity. Použití grafického formátu SVG a knihovny SVG.js si nejlépe demonstrujme konkrétním použitím nad jednou entitou. Celkově v ní využíváme čtyři nativní SVG elementy:

- **Circle** – Vykresluje kruh. Definujeme parametry barvy, okraje, výplně a velikosti.
- **Text** – Obsahuje text.
- **ForgeinObject** – Cizí objekt, může obsahovat například HTML.
- **Group** – Abstraktní uskupení více objektů. Umožňuje jednodušší manipulaci se všemi vnořenými objekty v jedné operaci. Typicky využíváme pro operace posunu.



Obrázek 5.1: Nákres využití SVG elementů pro vykreslení entity

Ve výše uvedeném obrázku vidíme grafickou reprezentaci entity skrze SVG s vyznačenými částmi. Jako hlavní vidíme objekt Circle (modrý), který vizualizuje entitu, dále objekt typu Text (fialový), skupinu tří objektů typu ForgeinObject (žlutý) a celý útvar je vsazen do SVG skupiny Group (šedě). Spojení do skupiny je z důvodu jednodušších operací s objekty. Tento vizualizovaný objekt je v XML realizován následovně:

Kód 5.1: SVG realizace entity

```
<g id="SvgjsG1019" transform="matrix(1,0,0,1,39,␣3,147)">
    <circle id="SvgjsCircle1016" stroke="#db2828" stroke-width="5"></circle>
    <g id="SvgjsG1012" class="editElement">
        <foreignObject id="fobjSvgjsG1012">
            <button class="btn␣btn-info␣btn-edit"></button>
        </foreignObject>
        <foreignObject id="fobjDelSvgjsG1012">
            <button class="btn␣btn-danger␣btn-edit"></button>
        </foreignObject>
        <foreignObject id="fobjViewSvgjsG1012">
            <button class="btn␣btn-default␣btn-edit␣btn-entity_type" ...>
                <img src="/static/imgs/gdrive.png" class="icon_image" />
            </button>
        </foreignObject>
    </g>
    <text id="SvgjsText1017" y="33.703125" x="0.5">
        <tspan id="SvgjsTspan1009" dy="20.8" x="0.5">Entita</tspan>
    </text>
</g>
```

Vrátíme-li se v kontextu do datového návrhu, pak můžeme demonstrovat rozdělení dat mezi lokální a kolaborativní úložiště. Z výše uvedeného obrázku je do kolaborativní struktury zaveden pouze element **Circle** a **Text**. Šedě vyznačená Group je pak zmíněna pouze odkazem. Zbylé elementy jsou pouze lokální.

Z hlediska knihovny SVG.js je pro realizaci takového SVG elementu nutné vykonat následující kroky. Prvně musíme vytvořit „kreslicí prostor“, tedy nad vybraným HTML elementem incializovat objekt SVG.js. Tento objekt nemá bližší specifika, pouze je nutné jej idenitifikovat uníkátním ID, v našem případě se jedná o objekt s ID `drawing`. Následující ukázky zdrojového kódu najdeme v souboru *sherito/static/app/app.ts*.

```
<div id="drawing"></div> - HTML
this.draw = SVG('drawing').size(1200, 800); - inicializace SVG
```

Tedy v instanční proměnné `draw` máme k dispozici objekt, do kterého můžeme následně přidávat další SVG objekty. Přidávání objektů provádíme voláním jednotlivých metod, které poskytuje SVG.js knihovna, tedy například:

```
this.draw.circle(10); //  přidání kruhu nebo
this.draw.text('test'); // přidání textového elementu
```

Důležitým SVG prvkem, který výrazně zjednodušuje další vývoj jsou skupiny, tedy prvek typu **Group**. Ty uskupují přidané elementy, jak můžeme vidět v následující ukázce kódu.

```
let group = this.draw.group();  // vytvoření skupiny
let fobjDel = this.draw.foreignObject(50, 50);  // vytvoření cizího objektu
let fobjEdit = this.draw.foreignObject(50, 50);

group.add(fobjEdit);  // přidání objektu do skupiny
group.add(fobjDel);
```

Důležité je také myslet na pořadí přidávání elementů do SVG nebo do skupiny, protože pořadí má přímý vliv na překrývání objektů. Překryv jde ale změnit buď CSS vlastností `z-index`, nebo skrze metody `forward()`, `backward()`, `front()` a `back()`.

Jelikož SVG je stále XML jazyk, je možné jej stylovat pomocí CSS[1] nebo upravovat pomocí jQuery. Toho využíváme například při průhlednostech jednotlivých elementů, nebo při změně pozadí kreslicího plátna.

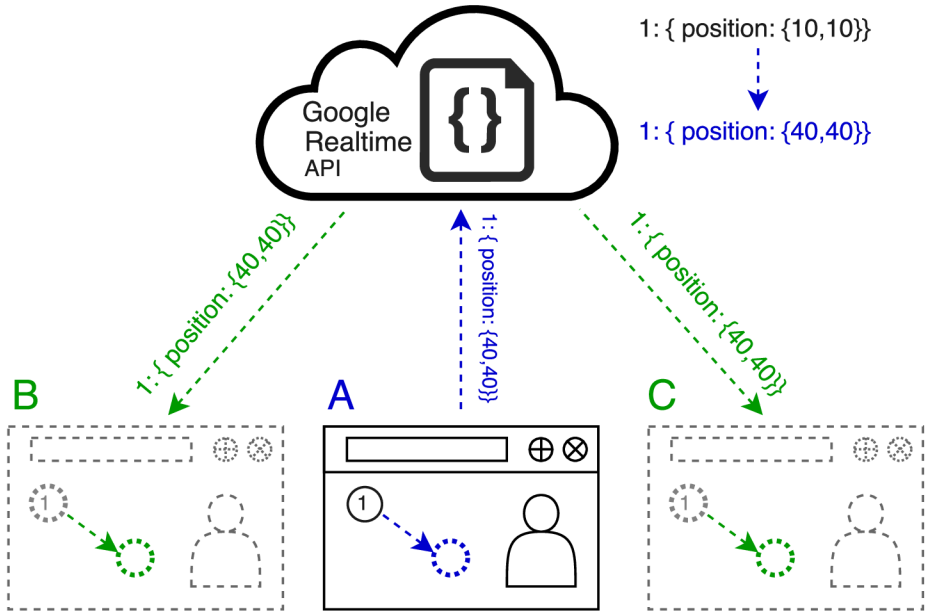
Velkou výhodou při výběru knihovny SVG.js byly také externí moduly, kterými je možné rozšířit chování knihovny. V aplikaci využíváme tedy moduly například pro operaci posunu, spojování elementů nebo změnu měřítka některých objektů. Například modul *svg.draggable.js*[2] umožňuje manipulovat s výše udevenými skupinami principem „táhni a pusť“, který je velice populární a známý z jiných aplikací. V tomto případě opět využíváme spojení elementů do skupiny, protože stačí následující volání metody `draggable()` nad skupinou a všechny vnořené elementy se budou posouvat současně.

[1] dle https://www.w3.org/TR/xml-stylesheet/
[2] http://svgjs.com/plugins/svg-draggable-js/

5.1.3 Google Realtime API

Pro kolaboraci mezi uživateli jsme v sekcích 4.5.2 a 4.6.3 vybrali technologii Google Realtime API. Pro pochopení níže uvedených principů je vhodná paralela s Google Docs. Tento kancelářský balík od Googlu využívá právě stejnou technologii, jakou se chystáme implementovat do aplikace.

Google Realtime API poskytuje technologické možnosti pro distribuci dat nad jedním datovým modelem. Princip operací si demonstrujme následujícím obrázkem. Pokud klient



Obrázek 5.2: Znázornění kolaborace a distribuce skrze Google Realtime API

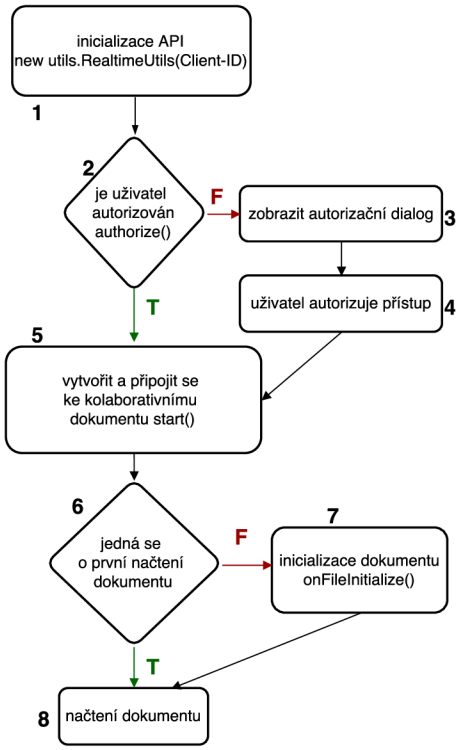
A má otevřený sdílený soubor a pokud je kolaborativní aplikace správně incializovaná, pak při změně prvku s identifikátorem 1 se zašle informace o změně na server a Google Realtime API následně distribuuje informaci o změně prvku mezi další klienty, kteří mohou mít soubor otevřený. V tomto případě se změnila pozice z `{10,10}` na `{40,40}`[3] a modrou barvou vidíme zaslání zprávy na server. Zelenou pak vidíme distribuci zprávy dalším klientům. Zároveň si server uchovává nejnovější informace pro případ, že by se připojil další klient.

Incializace aplikace kooperující skrze Google Realtime API

Pro použití API je nutné nejdříve zaregistrovat aplikaci v portálu developers.google.com, kde je následně aplikaci vygenerován klíč, kterým se identifikuje do prostředí Google Drive, které je úložištěm pro technologii Google Realtime API.

Následně je potřeba do aplikace přiložit zdrojové kódy Google Realtime API. Incializace kolaborativní aplikace pak probíhá následovně.

[3]Souřadnice X a Y kartézké soustavy souřadnic



Obrázek 5.3: Incializace kolaborativní aplikace do prostředí Google Realtime API

Cílem diagramu je vysvětlit, jak od prvního načtení inicializační funkce dochází ke sdílení dokumentu na Google Drive.

- (1) Vytváří se samotný objekt RealtimUtils (třída zabezpečující práci s Google Realtime API). Zde je nutný právě klíč poskytnutý z konzole GoogleDevelopers.
- (2) Dále se zkoumá, zda uživatel je přihlášen Google účtem a má přístup k dokumentu.
  - (3, 4) V negativním případě se nabízí operace pro přihlášení skrze modální okno a uživatel autorizuje přístup.
- (5) Po autentizaci se volá funkce `start()`, která vytváří dokument a přidává do stávající URL parametr s unikátním klíčem. Ve výsledku URL vypadá například následovně `http://<URL>?id=0B-yr9Qly_NWkQjcwTEpDLVVDNUk`. Tento klíč nazývá dokumentace jako *fileID*. Na tento pojem budeme dále odkazovat.
- (6) Dále se ověřuje stav dokumentu.
  - (7) Pokud se jedná o první načtení dokumentu, pak je potřeba inicializovat vstupní data, volá se tedy funkce `onfileInitialize()`.
- (8) Vstupní data z dokumentu se předávají funkci `onFileLoaded()`.

Všimněme si URL adresy v bodu 5 výše uvedeného diagamu. Do URL se přidává identifikátor *fileID*. Jedná se o unikátní identifikaci sdíleného dokumentu na Google Drive. Podobný identifikátor mají i běžné Google Docs dokumenty. Můžeme jej najít například v `https://docs.google.com/document/d/1QyRhsDulFzdf7JOqhtd9swXtcB8w/`. Tento klíč musíme uchovávat v lokální databázi, abychom věděli k jakému dokumentu se připojit, až uživatel bude chtít aplikaci znovu otevřít.

Komunikace v rámci Google Realtime API

Po inicializaci kolaborativní aplikace je potřeba zpracovávat události, které buď generujeme na straně klienta, nebo přicházejí skrze Google Realtime API od jiného uživatele. Skrze tyto události dokumenty komunikují. Proto ve funkci `onFileLoaded()` voláme funkci `initEvents()`, která k jednotlivým příchozím událostem přiřazuje funkce, které tyto události mají zpracovat.

Než ale budeme pokračovat zpracováním událostí, pojďme si o nich říct více informací. Události se dělí do několika typů a ke každému kolaborativnímu datovému modelu máme z principu jinou sadu událostí. Nejvíce jich má kolaborativní seznam, neboť se může například měnit pořadí. V případě kolaborativní mapy pracujeme nejčastěji s událostí typu `VALUE_CHANGED`, kterou bychom mohli definovat jako „něco se změnilo“. Do kolaborativní mapy zadáváme a měníme položky metodou `set(id, data)`, která právě vyvolává tyto události. Příchozí událost má charakter JSON zprávy v následujícím tvaru:

```
{
    ...
    isLocal : true - je změna lokální
    ...
    newValue : Object - nová hodnota
    oldValue : Object - stará hodnota
    ...
    type : "value_changed"
}
```

Ve výše uvedené ukázce kódu můžeme vidět několik nejdůležitějších informací. Nejdříve vidíme informaci `isLocal`, tedy údaj, zda se jedná o lokální změnu, nebo změnu vyvolanou jiným klientem. Tato informace nás zajímá, neboť aplikovaná událost se nedostává pouze dalším instancím dokumentu, ale také do lokálního prostředí. Nejdůležitější atributy jsou ale prvky `newValue` a `oldValue`. Ty obsahují staré a nové hodnoty zadané položky (ve výše uvedené struktuře), podle kterých můžeme operaci dále zpracovat. Jako poslední vidíme atribut `type`, který určuje typ operace, který jsme ale už diskutovali výše.

Zpracování operace pak probíhá formou informování SVG modelu o změně. Ten pak musí provést stejné operace s uživatelským rozhraním jako uživatel v jiné instanci aplikace. Tedy pokud například externí uživatel změnil barvu entity, pak stejnou operaci musíme provést „ručně“ i v lokální instanci. Pro hlubší definici operace máme ve struktuře položky (viz 4.6.3) definovanou zasílanou zprávu, například `rename`, `new` nebo `moved`, tedy víme, jakou operaci máme provést.

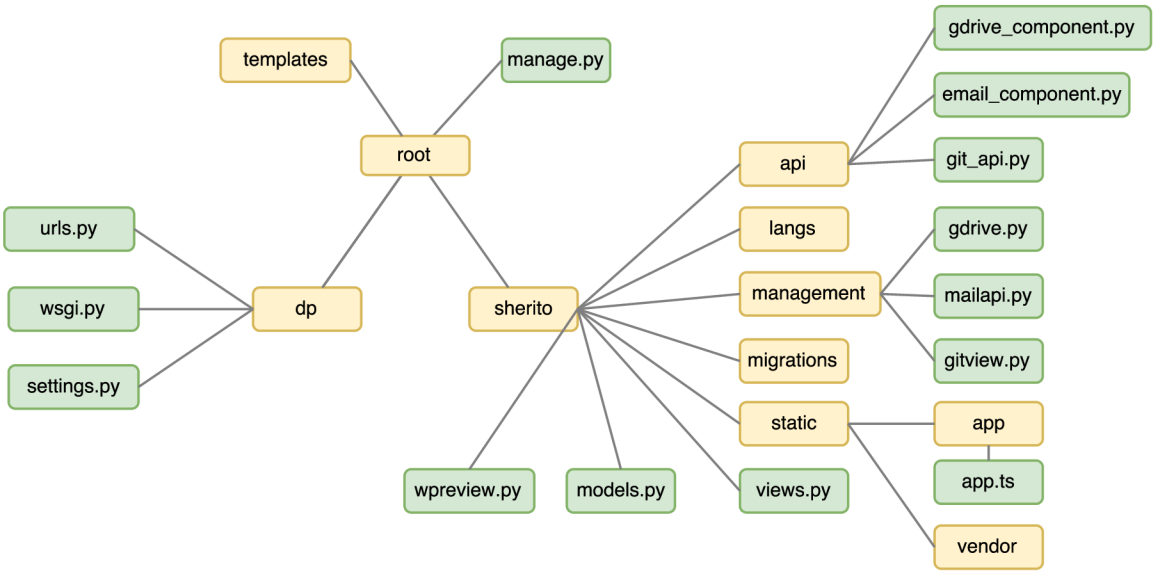
5.2 Implementace backend vrstvy

Backend vrstva poskytuje úložiště a aplikační logiku, ze které následně čerpá frontend vrstva ve vizualizaci. Na straně serveru se tedy budeme zabývat následujícími technologiemi:

- **Django** – Webový framework v jazyce Python poskytující aplikační logiku.
- **Google Drive REST API** – Serverové čtení z Google Drive pro integraci dat z Google Drive.
- **IMAP** – Práce s emailovými schránkami, pro integraci emailů.
- **GitWebhook** – Technologie pro napojení Git repozitářů na externí služby.

5.2.1 Webový framework Django

Webový framework Django je postaven na jazyce Python a to ve verzích 2.7 i 3. Využívá MVC strukturu, kterou jsme si popsali v sekci 4.6.1 zabývající se architekturou. V následující sekci se budeme zabývat jeho implementací. Nejdříve si ale ukažme adresářovou a souborovou strukturu aplikace[4].



Obrázek 5.4: Souborová struktura aplikačního rámce Django

Identifikujme a popišme jednotlivé adresáře (v jazyce Python se označují jako balíčky) a soubory.

- **dp** – Adresář projektu. Framework Django rozlišuje pojmy projekt a aplikace. Tento adresář obsahuje základní nastavení společná pro všechny aplikace.
  - **wsgi.py** – Skript, který se používá jako rozhraní pro napojení na webový server.
  - **urls.py** – Definování jednotlivých URL a směřování na konkrétní kontrolery.
  - **settings.py** – Konfigurační soubor projektu.
- **manage.py** – Nástroj pro práci s aplikací z příkazové řádky. Skrze `manage.py` je možné aplikaci instalovat, spravovat apod.
- **sherito** – Adresář aplikace. Django umožňuje mít v jednom projektu více aplikací. Název adresáře vychází z názvu aplikaci, který jsme definovali v sekci 2.2.
  - **api** – Adesář pro jednotlivé soubory realizující komunikaci s API nebo přímo nějaké API reprezentující.
    - **gdrive_component.py** – Skript komunikující s Google Drive REST API.
    - **email_component.py** – Komunikace s IMAP servery.
    - **git_api.py** – API pro zpracování požadavků typu webhook.
  - **langs** – Adresář obsahující překlady aplikace.

[4]Struktura je omezena pouze na klíčové prvky. Některé méně důležité soubory chybí a najdete je na přiloženém CD.

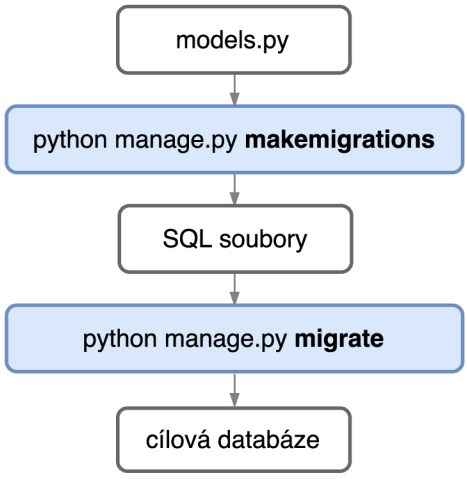
  - **management** – Adresář obsahující skripty pro zpracování příkazů z prostředí příkazové řádky. Využívají se zejména pro volání z operačního systému.
    * **gdrive.py** – Inicializuje komunikaci s Google Drive.
    * **email.py** – Inicializuje komunikaci s IMAP.
    * **init.py** – Tvorba inicializačních dat při zavedení instance aplikace.
  - **static** – Obsahuje statické soubory (z hlediska serveru). Jedná se tedy o obrázky, kaskádové styly a JavaScript soubory.
    * **app** – Soubor app.js, který je základem výsledné kolaborativní aplikace.
    * **vendor** – Adresář pro instalaci externích knihoven skrze nástroj Bower.
  - **views.py** – Vrstva Controlleru z MVC architektury. Obsahuje funkcionalitu zpracovávající HTTP požadavky.
  - **models.py** – Modelová vrstva MVC. Obsahuje třídy reprezentující datový model aplikace.
  - **wpreview.py** – Doplňkový nástroj pro získávání náhledů webových stránek.
- **templates** – Prezentační vrstva MVC. Obsahuje HTML šablony využívající Jinja2.

Správa aplikace

Ještě než se podíváme na základní funkcionality, je dobré se podívat, jaké jsou možnosti aplikace komunikovat mimo klasické HTTP požadavky. Pro ovládání ze strany prostředí příkazové řádky slouží skript `manage.py`, který umožňuje například instalaci aplikace, spuštění lokálního serveru, apod. Ukažme si dvě hlavní operace s aplikací, které zajišťuje `manage.py`.

Migrace

Jak už bylo několikrát zmíněno, Django pracuje s ORM principem. Ač pro programátora je databáze abstrahována na úroveň tříd, pak stejně je nutné cílové tabulky vytvořit v databázi. Django prochází dvěma kroky – nejdříve je nutné vytvořit SQL soubory (ty se ukládají do adresáře *migrations*), které odpovídají cílové databázi a následně se k databázi připojit a tyto soubory aplikovat, jak ukazuje následující diagram.



Obrázek 5.5: Znázornění procesu migrace

Spouštění aplikace z příkazové řádky

Zejména pro volání z tzv. CRONjobů[5], tedy pravidelně se opakujících operací, je potřeba zajistit možnost spouštění aplikace z příkazové řádky. Nemůžeme spustit přímo daný cílový Pyhton skript, nebot' je nutné spustit celou aplikaci, iniciazovat připojení k databázi apod. Proto vytváříme samostatné skripty, tzv. Commands (příkazy) uložené v adresáři *management*. Tyto jednotlivé skripty se následně spouštějí voláním skriptu `manage.py`, kde jako parametr uvedeme název tohoto příkazu (skriptu). Takto zajišťujeme pravidelné kontrolování IMAP schránek nebo instalaci inicializačních dat při tvorbě instance aplikace (například v novém prostředí).

5.2.2 Implementace modelu

O principech ORM jsme již v této práci řekli mnohé. Ukažme si ale, jak konkrétně vypadá práce s jednotlivými modely[6].

```
class Entity(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255)
```

Ve výše uvedené ukázce vidíme deklaraci modelu, který následně bude v databázi reprezentován tabulkou se sloupci `id` a `name`. Třída musí dědit z `models.Model`. Jednotlivé atributy odpovídají fomulářovým prvkům, neboť Django skrze princip DRY předpokládá persistenci formulářů přímo do modelů. Samozřejmě je možné zadefinovat vlastní. Takto definujeme celou datovou vrstvu, kterou jsme definovali v sekci 4.6.3.

K jednotlivým objektům (a tedy persistovaným datům) přistupujeme skrze atribut `objects`, který následně podle aplikovaného filtru vrací jeden nebo více objektů. Konkrétně tedy v následujících ukázkách vidíme invokace metod `get()` a `filter()`.

```
entity = Entity.objects.get(pk=entity_id)
accounts = IMAPCredentials.objects.filter(user=board.user)
```

Skrze tyto metody se pak modelová vrstva dotazuje databáze na data a vrací je v podobě objektů.

Jelikož Django je schopno pracovat s různými databázemi, pak se nemůže ve všech ohledech spoléhat na implementaci pokročilých vlastností na úrovni databáze, ale realizuje je na úrovni modelu. Proto například nenajdeme v databázi cizí klíče nebo unikátní indexy. Všechna tato omezení drží Django na aplikační úrovni.

5.2.3 Implementace řídící vrstvy

Řídící vrstvu Django implementuje v souboru `views.py`. V tomto souboru najdeme oba výše uvedené přístupy – *ClassBasedView* (CBV) a *FunctionBasedView* (FBV).

Při vývoji aplikace jsme použili oba přístupy. Pro globální práci s dashboardem jsme vytvořili třídu `BoardView`. Při použití přístupu CBV musí tyto třídy dědit od některé nadtřídy, která dědí ze třídy `View`. Těchto nadtříd může být celá řada, v našem případě dědíme z `TemplateView`. V kapitole 4.6 jsme jako výhodu CBV uváděli předdefinované metody pro zpracování odlišných HTTP požadavků. Přesně této vlastnosti zde využíváme.

[5] automatizované volání příkazů v pravidelných intervalech ze strany operačního systému

[6] v souboru models.py

Základní rozdělení podle typu požadavku řeší framework sám, bylo tedy jen potřeba implementovat metody `post()` a `get()`. V těle těchto metod si můžeme být jisti, že požadavky (reprezentované parametrem request) jsou příslušných typů. Identifikaci požadavku pak určujeme parametrem `action`, který definujeme dle vstupní URL v souboru `urls.py`. Konkrétní použití můžeme vidět na příkladu

```
url(r'^ajax/addFilter/', BoardView.as_view(action='addFilter')),

def post(self, request, *args, **kwargs):
if self.action == "addFilter":
   filter_id = None if args.__len__() == 0 else args[0]
   self.get_filter_form(request, filter_id)
```

Funkcionální přístup řídící vrstvy můžeme vidět ve funkci `track_directory()`, která zpracovává POST požadavky na zápis dat. Důvod pro použití FBV může být mnoho, v tomto případě volíme FBV, neboť není kam takový logický prvek zařadit a je zbytečné pro pouze jeden požadavek zřizovat celou třídu.

S kompletním zpracováním HTTP požadavku souvisí jak jeho příjem, tak také odpověď. Zde se kloubí dva přístupy. Buď musíme zvolit šablonu, kterou použijeme pro odpověď, nebo odpovídáme přímo a to většinou skrze formát JSON.

Pokud odpovídáme skrze šablonu, pak platí následující postup. Je nutné zvolit soubor, který šablonu obsahuje a následně vytvořit slovník[7] dat, který se do šablony zasílá. Tyto dva parametry následně předáváme k vykreslení odpovědi. Konkrétní použití vidíme v následující ukázce.

```
self.template_name = 'board/githook_form.html'

self.context = {
   'form': form,
   'edit_id': webhook_id,
   'board_id': request.session['board_id']
}
return render(request, self.template_name, self.context)
```

Kromě odpovědí s obsahem můžeme také využít zbylých možností HTTP protokolu, tedy například v odpovědi vrátit přesměrování na jinou adresu.

5.2.4 Napojení aplikace na webový server

Poslední částí, kterou je potřeba popsat v rámci implementace aplikace je napojení na webový server. Django je pouze webový framework a pro plnohodnotné využití aplikace je potřeba napojit aplikaci na webový server. V tomto případě byl použit webový server Apache.

Pro napojení na webový server se využívá rozhraní WSGI – Web Service Gateway Interface[8]. Prakticky se jedná o spouštění definovaného Python skriptu, který spustí celou aplikaci (v našem případě podporovanou frameworkem Django) a tedy reaguje na HTTP požadavky, které přicházejí skrze webový server. Použití WSGI předpokládá podporu ze strany webového serveru, což je v případě serveru Apache realizováno modulem *mod_wsgi*.

[7] asociativní datová struktura v jazyce Python

[8] http://wsgi.readthedocs.io/

Prakticky se jedná o nastavení Apache serveru skrze direktivu `VirtualHost`, kterou můžeme vidět v následujícím příkladu.

```
<VirtualHost *:80>
    ServerName  dp.jirisemmler.eu
    WSGIScriptAlias / /www/html/dp/dp/wsgi.py - cesta k~wsgi skriptu
<Directory /www/html/dp>
   <Files wsgi.py>
   Require all granted
   </Files>
</Directory>
</VirtualHost>
```

5.3 Služby třetích stran

5.3.1 Google Drive REST API

V předešlých kapitolách jsme mluvili o Google Realtime API, zde budeme diskutovat integraci Google Drive REST API[9]. Informace popsané v této kapitole čerpáme z oficiální dokumentace GoogleDevelopers [4]. Pro ujasnění rozdílu a přístupů mezi zmíněnými API vysvětleme situaci. Google Realtime API používáme z důvodu kolaborace nad daty v aplikaci v reálném čase. Google Realtime API vytváří dokument v úložišti Google Drive, ale jedná se pouze o jeden soubor, se kterým pracujeme. Cíl integrace Google Drive REST API je jiný. V úvodu jsme řekli, že chceme ze souborů uložených na Google Drive vytvářet entity do výsledné aplikace. Tedy skrze Google Drive REST API čteme Google Drive přihlášeného uživatele a nabízíme mu soubory a adresáře k vytváření entit nebo k označování za sledovaný adresář. Sledované adresáře pak následně v pravidelných intervalech kontrolujeme. Pokud se objeví nový soubor, zařazujeme jej do fronty pro možnost vytvoření entity. Tedy skrze diskutované Google Drive REST API pouze čteme Google Drive uživatele a stahujeme informace o souborech a adresářích. Neprovádíme žádné operace zápisu. Tato integrace je čistě prováděna ze strany serverové aplikace.

Přístupové údaje ke Google Drive

Pro integraci Google Drive můžeme využít stejný přístupový klíč jako pro práci s Google Realtime API. Ale ač se jedná o serverovou aplikaci, stále musí přihlášený uživatel dát svolení k přistupu ke svému Google Drive účtu. Při přidělení přístupu se pro kombinaci *Google Drive uživatel – přístupový klíč aplikace* vytváří přístupový token, který se později využívá pro další přístupy ze strany serveru ke Google Drive uživatele. Pro uložení přístupového tokenu používáme třídu Google `DriveCredentialsModel`, kterou známe z datového modelu. Zároveň pro přímé uložení tokenu se využívá speciální typ modelu `CredentialsField()`, který se pak interně v databázi realizuje textovým polem. Jedná se ale o specifický datový typ, který je možné použít na úrovni modelu aplikace, o kterém jsme se zmínili v předešlé kapitole.

[9] Diskutované postupy a funkcionality najdeme v zdrojovém kódu api/gdrive_component.py.

Přístupové údaje ke Google Drive

Google Drive REST API je typu REST[10], tedy využíváme pouze HTTP požadavky. Ale díky knihovně pro jazyk Python přímo od vývojářů Google nemusíme vytvářet samostatné REST požadavky, ale můžeme pracovat s knihovní podporou.

Nebudeme ale využívat plně možností Google Drive API, neboť našim cílem je pouze číst obsahy adresářů a získávat dodatečné informace o jednotlivých souborech. Pro tyto operace musíme provést následující kroky:

1. Ustanovení spojení – Autentizace aplikace a prokázání povolení přístupu ke Google Drive vybraného uživatele.
2. Sestavení vyhledávacího dotazu.
3. Získání dodatečných informací o souboru a lokální zpracování.

Pro ustanovení spojení se ve třídě `GoogleComponent` využívá metoda `set_service()`. Ta musí nejdříve získat přístupové údaje a na jejich základě ustanovit spojení. Spojení může existovat pro více druhů Google služeb (v tomto případě využíváme jen Google Drive, ale můžeme pracovat s daty Gmailu, Hangouts...) a také musí vybrat verzi API, na které budeme komunikovat, neboť jsou momentálně podporovány dvě verze Google Drive REST API. Ustanovení spojení vidíme v následující ukázce kódu.

```
credentials = self.get_credentials()
http = credentials.authorize(httplib2.Http())
self.service = discovery.build('drive', 'v3', http=http)
```

Dále musíme sestavit vyhledávací dotaz. Ten se skládá z mnoha povinných nebo volitelných atributů, podle kterých můžeme chtít hledat soubory. Jak klíčové je vhodné zmínit následující atributy:

- `corpora` – Data v Google Drive jsou uspořádány do mnoha různých kolekcí na základě vlastnického vztahu uživatele a souborů. Tedy nacházíme kategorie:
  - Moje soubory
  - Sdíleno se mnou
  - Sdíleno v týmu
- `parents` – Soubor může mít více rodičů (adresářů) a právě tyto adresáře ve vyhledávacím dotazu určujeme dle identifikátoru *fileID*, který jsme definovali v kapitole 5.1.3.
- `mimeType` – Vyhledávat můžeme dle datových typů. Zajímavý je přístup k různým datovým typům, neboť Google je schopen pokrýt názvem a deterministickým datovým typem pouze soubory vycházející z vlastních Google služeb (například Kresba). Také Google Drive obsahuje mnoho souborů od aplikací třetí strany, ty označuje jako „Aplikace třetí strany“.

[10] REST (Representational State Transfer) – architektura rozhraní využívající HTTP požadavky pro manipulaci dat v externí službě

Samozřejmě můžeme vyhledávat dle mnohem více atributů, ale validní pro naši aplikaci je jen několik z nich.

Po vyhledání potřebných souborů ukládáme identifikátor *fileID* do databáze společně s názvem souboru. Více informací není potřeba, neboť dodatečné informace můžeme získat na požádání z Google Drive REST API.

Získání dodatečných informací o souboru pak funguje dle stejného principu jako vyhledávání ve složce, ale dotazujeme se přímo na vybraný soubor (souboru určujeme uloženým identifikátorem *fileID*). O souboru můžeme zjistit opět sadu informací, pro účely aplikace se dotazujeme na název, typ souboru, popis, odkaz pro zobrazení, odkaz pro editaci, náhled a datum poslední změny. Tyto informace následně vizualizujeme uživateli.

5.3.2 IMAP

Emaily jsou další službou, kterou chceme integrovat do aplikace a vytvářet z nich entity. Pro příjem emailů máme k dispozici dva protokoly – IMAP a POP3. Vzhledem k faktu, že opět chceme provádět jen čtecí operace, pak je vhodnější protokol IMAP. Pro implementaci IMAP komunikace používáme knihovnu `imaplib`[11] a její rozšíření `imaplib_list_parse` z modulové sady PyMOTW[12].

Uložení údajů

Zásadní problém při práci s protokolem IMAP nastává už u způsobu uložení přihlašovacích údajů. Při každém vytvořeném spojení (přihlášení na IMAP server) se musíme autentizovat uživatelským jménem a heslem, a to v otevřené podobě, tedy musíme tyto údaje ukládat. Kardinální chybou by bylo ukládat tato data v otevřené podobě (takzvaný plaintext), tedy musíme zvolit princip šifrování, abychom data uchránili pro případ úniku databáze.

Pro šifrování musíme použít symetrický algoritmus, neboť potřebujeme zpětně získat původní podobu šifrovaného textu. Jako ideální volba se ukazuje šifra AES[13] implementovaná v knihovně Crypto v jazyce Python. Jedná se o blokovou šifru, která pro šifrování i dešifrování využívá klíč pevné délky, v našem případě 128 bitů. Tento klíč uchováváme v souboru `settings.py`.

Při práci s šifrovanými daty se opět ukazuje jako velice dobrý přístup ORM. Jak můžeme vidět v modelové třídě `IMAPCredentials`, šifrování a dešifrování můžeme delegovat na vrstvu modelu a při použití jsme od této práce navíc oproštěni. Zároveň předpokládáme podobné požadavky na šifrování i u dalších služeb, tedy vytváříme třídu `Credenials`, která implementuje šifrování i dešifrování, ze které třída `IMAPCredentials` pouze dědí.

Operace se schránkou

Celá práce s IMAP je realizována v jednom Python skriptu `email_component.py`, ve třídě `EmailComponent`. Třída prakticky dělá pouze dvě operace a to vytvoření spojení a následné čtení a filtrování emailů z dané schránky. Při kontrole emailové schránky je nutno nejdříve vytvořit spojení metodou `set_connection()`. Tato metoda vyžaduje jako parametr objekt třídy `IMAPCredentials`. Zde vidíme opět použití ORM. Není potřeba data nějak připravovat, dešifrovat hesla nebo provádět jinou režii. Data přichází v objektu dokonce zašifrovaná a následně jsou rozšifrována až při přistoupení k atributům objektu. Při ustanovení spojení

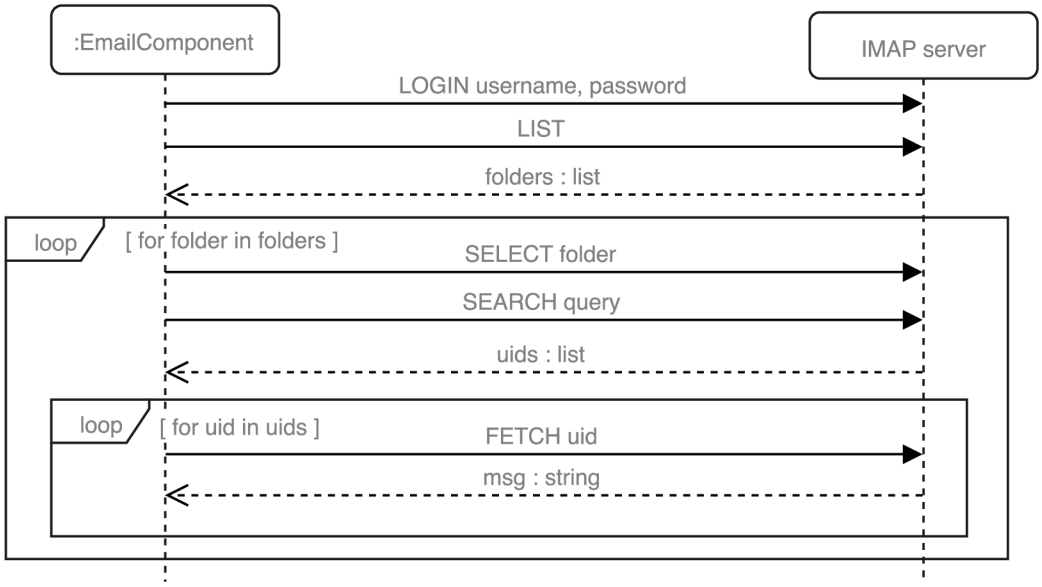
[11] https://docs.python.org/2/library/imaplib.html
[12] https://pymotw.com/3/
[13] https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.AES-module.html

předpokládáme základní nastavení IMAP serveru a použitého portu 993 pro komunikaci. Pokud uživatel používá a tedy vyplnil jiný port, pak toto nastavení přetížíme.

Při implementaci metody pro čtení a filtrování dat z emailové schránky vycházíme z RFC 3501 [12], které popisuje používané příkazy `LIST`, `SEARCH`, `SELECT` a `FETCH`. Při čtení emailové schránky sestavujeme vyhledávací dotaz, stejně jako v případě čtení Google Drive. Ten se může skládat z následujících klauzulí. Tyto klauzule při kombinaci se spojují logickou spojkou konjunkce.

- `UID` – Výběr dané zprávy ve složce dle unikátního číselného identifikátoru. Používáme v podobě zarážky pro čtení emailů. V podobě `UID 10:*` se vrací všechny emaily s `UID` větší než 10.
- `SUBJECT` – Používáme pro vyhledání klíčového slova v předmětu zprávy.
- `BODY` – Vyhledání klíčového slova v těle zprávy.
- `FROM` – Vyhledávání dle zdrojové emailové adresy.
- `TO` – Vyhledávání dle cílové emailové adresy.

Samotné čtení ze schránky postupuje dle následujícího sekvenčního diagramu.



Obrázek 5.6: Komunikace mezi lokální aplikací a IMAP.

Nejdříve musíme získat seznam adresářů (příkaz `LIST`), následně iterujeme (příkaz `SELECT`) nad adresáři schránky a nad každým adresářem aplikujeme (příkaz `SEARCH`) vyhledávací dotaz. Jako odpověď dostáváme `UID` jednotlivých zpráv, které vyhovují dotazu. Ty následně příkazem `FETCH` načítáme a zpracováváme. Při zpracování přijatého emailu se musíme vypořádat se dvěma problémy. První je rozdělení zprávy na více částí, tzv. multipart zprávy. Ty musíme konkatenovat. A druhým problémem je znaková sada zprávy. Pro dekódování zprávy využíváme třídu `Quopri`. Pro následnou práci s emailem v podobě entity ukládáme jednotlivé části zprávy, podle kterých můžeme následně filtrovat, ale také kompletní zprávu, kdybychom chtěli získat více informací. Takto získané emaily zařazujeme do fronty pro tvorbu entit (třída `EmailItemQueue`).

5.3.3 Metoda webhook

V následující kapitole se budeme zabývat integrací Git repozitářů do aplikace skrze metodu webhook. Cílem integrace je vytvářet entity z jednotlivých změn v repozitáři (tzv. commitů). Výše uvedené integrace externích služeb vychází z aktivního přístupu, kdy se aplikace dotazuje v pravidelných intervalech externí služby. Ovšem můžeme použít i pasivní přístup, kdy vystavíme API, na které se bude dotazovat externí služba. Přesně tento model splňuje metoda webhook. Webhook označuje metodu propojení dvou webových systémů, kdy strana příjemce (v tomto případě naše aplikace) definuje URL, na kterou je možné zasílat data skrze HTTP požadavky na základě interního spouštěče v externí službě [21]. Často se tato metoda využívá u repozitářů, které chtějí informovat další služby o provedené operaci, většinou při operaci „push“, kdy jsou na server nahrány nové změny. Právě tento případ nastává i zde. Uživatel na straně aplikace definuje poskytovatele repozitáře (momentálně je možné pracovat se službami BitBucket[14] a Github[15]) a sadu klíčových slov, podle kterých se pak filtrují jednotlivé zaslané změny. Po vytvoření tohoto filtru dostává uživatel URL s definovaným tokenem (z důvodu identifikace a omezení přístupu), na kterou může externí služba zasílat data. Prakticky tedy definujeme v `urls.py` URL adresu a následně přijímané požadavky zasíláme do vrstvy `View`, kde zaslaná data zpracujeme.

```
url(r'^webhook/git/(?P<token>[\w-]+)$', HookView.as_view()),
```

Ve třídě HookView[16] opět využíváme přístup CBV. Důležité je zmínit, že formát zasílaných dat může každá služba definovat ve vlastní režii, tedy pro podporu dalších poskytovatelů repozitářů je potřeba definovat metodu, která bude pracovat se specifickou formou zasílaných dat od poskytovatele. Po zpracování dat a filtraci dle klíčových slov, které zadal uživatel (typicky se může jednat o klíčová slova jako „release“), vytváříme objekt `GitItemQueue`, kterým zařazujeme data do fronty.

Tímto bodem uzavíráme kapitolu implementace, neboť byly diskutovány všechny klíčové technologie funkcionality, podílející se na funkčnosti aplikace.

[14] https://bitbucket.org/
[15] https://github.com/
[16] v souboru api/git_api.py

Kapitola 6

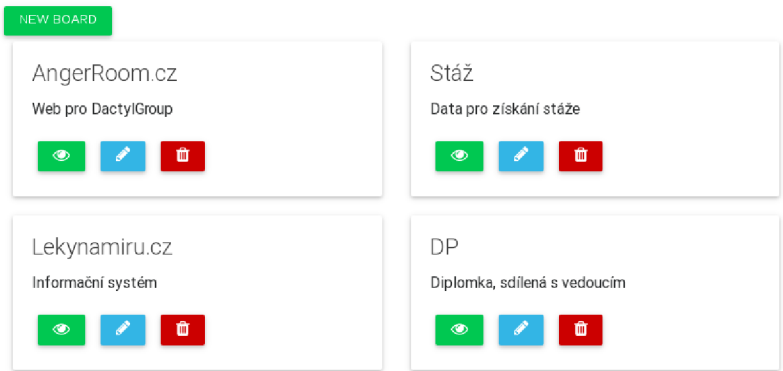
Vyhodnocení a přínos výsledné aplikace

Výsledná aplikace, která je realizována a popsána v této práci, splňuje základní požadavky, které jsme stanovili v návrhu pro první iteraci vývoje, napodobuje vhodné a obchází nevhodné praktiky stávajících konkurenčních služeb, které jsme definovali v analýze konkurence a podporuje agilní vývoj, tak jak jsme tento přínos definovali v úvodní analýze. Výše uvedené tvrzení budeme v této kapitole demonstrovat několika body. Nejdříve stanovíme sadu funkcionalit, které aplikace poskytuje a přiznáme i slabé stránky jako příležitosti dalšího rozvoje. Následně zhodnotíme proces vývoje a stanovíme další možná rozšíření.

6.1 Funkcionalita aplikace

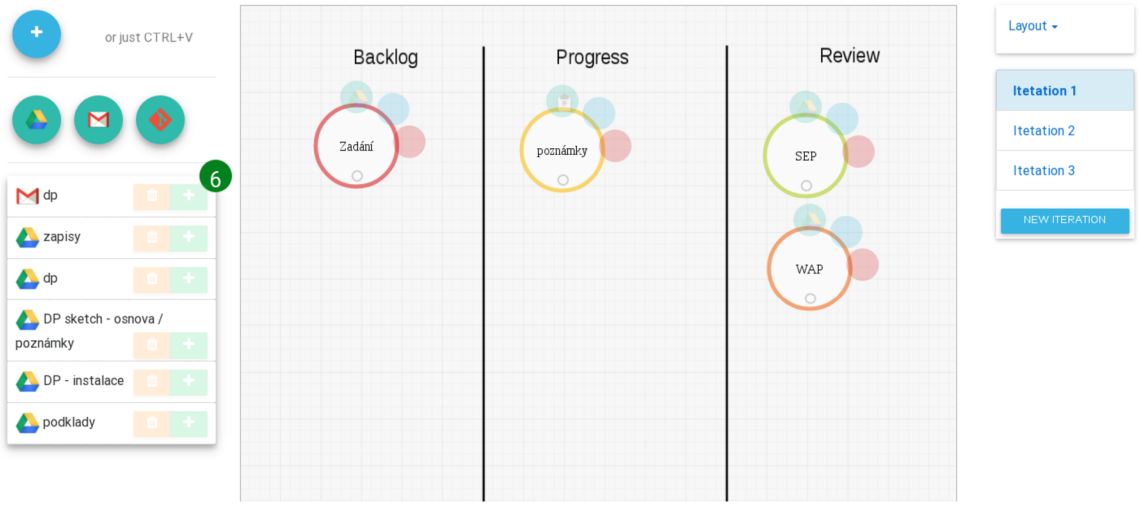
Aplikace pokrývá funkcionalitu definovanou pro první iteraci vývoje služby. Podařilo se implementovat klíčové funkcionality definované pro MVP[1] aplikace. Zbylé funkcionality mohou být implementovány při dalším vývoji. Tento stav můžeme považovat za příležitost validovat koncept, získat cennou zpětnou vazbu a dále pokračovat ve vývoji. Tato praktika je také, ač sekundárně, doporučena z metodiky LeanSD.

Kromě implicitní práce s uživateli, aplikace nabízí správu jednotlivých dashboardů, které agregují entity. Tyto dashboardy lze uspořádat, sdílet, editovat a archivovat.



Obrázek 6.1: Přehled s dashboardy

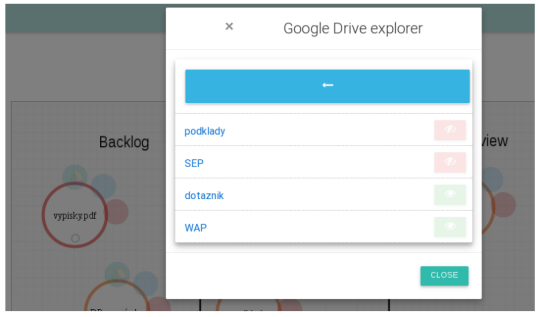
[1] Minimum Viable Product, produkt s nejmenší možnou funkcionalitou, tak aby mohl validovat koncept



Obrázek 6.2: Snímek z realizované aplikace

Dashboard pak poskytuje hlavní funkcionalitu aplikace. Lze v něm vytvářet entity, editovat je, uspořádat dle podkladové vrstvy (Kanban, Lean Canvas, SWOT apod.) a zobrazovat jejich detail. Klíčovým elementem je integrace externích služeb. Ve stávající verzi aplikace podporuje propojení s Google Drive, IMAP emailovým serverem a Git repozitáři. Pro tyto služby aplikace nabízí možnosti definovat filtry, podle kterých se pak nabízí položky těchto služeb (jednotlivé soubory z Google Drive nebo emaily) k zavedení jako entity. V následujících snímcích vidíme některé z výše uvedených funkcionalit.

Obrázek 6.3: Práce s iteracemi a podklady



Obrázek 6.4: Procházení uživatelského Google Drive.

Důležitým cílem aplikace je podpora agilního modelování a vývoje. To aplikace podporuje možností tvorby iterací a agregací jednotlivých entit, kde všechny se dle úvodní analýzy považují za agilní modely. Ve výše uvedeném náhledu vidíme práci s iteracemi.

6.2 Slabiny aplikace

Vytvořená aplikace má rysy MVP, kdy pokrývá základní funkcionalitu a ověřuje definovaný koncept. Pro komerční využití bude potřeba odstranit následující nedostatky, které zde přiznáváme.

- Málo propojených externích služeb. V úvodu jsme analyzovali aplikaci Zapier, která integruje kolem 750 externích služeb. V tomto případě integrujeme pouze několik jednotek. Pro businessové rozšíření aplikace je potřeba identifikovat základní sadu externích aplikací a implementovat jejich napojení
- Nedostatečná a neudržitelná technologie na straně klienta. Po několika iteracích vývoje, kdy přibyly postupně další a další funkcionality, se ukazuje, že nebyla vhodně zvolena technologie obsluhující uživatelské rozhraní. Z typické aplikace server-klient by bylo potřeba rozšířit aplikaci o JavaScriptovou technologii pracující kvalitněji a udržitelněji s uživatelským rozhraním – například ReactJS nebo Angular2.
- Úroveň grafické realizace. Aplikace byla postavena ze základních grafických elementů frameworku Bootstrap s prvky Material Design. Pro komerční použití by bylo vhodné zpracovat individuální grafický návrh.

6.3 Nové postřehy k zapracování

Během vývoje aplikace jsme narazili na řadu nových vlastností a možností, které nebyly viditelné v počátcích realizace. Tyto postřehy ovlivnily vývoj a budou podnětem pro další rozvoj aplikace.

V prvním bodě je nutné zmínit velice kvalitní a přínosné vývojové prostředí od firmy Google – Google API a Google SDK. Toto prostředí nabízí velice širokou technologickou základnu pro vývojáře a jejich projekty. V případě řešené aplikace využíváme pouze Google Realtime API a Google Drive REST API, a to pouze v omezené míře. Možnosti, které nabízí tato dvě API, jsou mnohem širší (editace dokumentů, notifikace...) a měly by být rozpracovány více.

Dále je překvapující výsledná datová struktura, zejména na úrovni databáze. Je to způsobené velkou datovou podporou ze strany technologií Google, snahou o maximálně univerzální řešení (zejména z hlediska entit) a DRY principem, který využíváme při práci s uživateli, a tedy je využita funkcionalita a struktura ze strany frameworku Django.

Naopak co se ukázalo jako překvapení v negativním významu, je roztříštěnost implementací některých externích služeb. Ta se ukázala jako silná u IMAP, neboť se ukázaly servery s nestandardně implementovanou podporou IMAP. Například Gmail má podporu IMAP tak omezenou, že je výrazně vhodnější pracovat s emaily Googlu skrze SDK. Překvapivé bylo také zjištění, že implementace jednotlivých webhook řešení mezi různými službami poskytující Git repozitáře je velice odlišná. Každý poskytovatel zasílá data v jiné struktuře, což vede k faktu, že pro každého poskytovatele musí být implementována individuální zpracující metoda nebo alespoň konverzní rozhraní.

6.4 Případová studie demonstrující přínos aplikace

Cílem aplikace je podpořit agilní modelování a agilní vývoj softwaru. V úvodní kapitole jsme definovali přínos aplikace pro agilní modelování v podobě agregace dat, které považujeme

za modely. Ukažme v následující kapitole výsledný přínos, který aplikace v praxi[2] může mít. Demonstraci provedeme formou případové studie na reálném procesu v již zmíněné firmě DactylGroup s.r.o, kde autor působí na pozici projektové manažera.

Je realizován softwarový projekt A, do kterého vstupuje celkově pět lidí – za dodavatele to je projektový manažer, hlavní vývojář a grafik. Za zadavatele se účastní vedoucí ICT odboru a manažerka pro vnější vztahy. Cílem projektu je dodat nové webové stránky spojené s informačním systémem.

V první fázi projektu vzniká následující sada artefaktů, které všechny se mohou považovat za model:

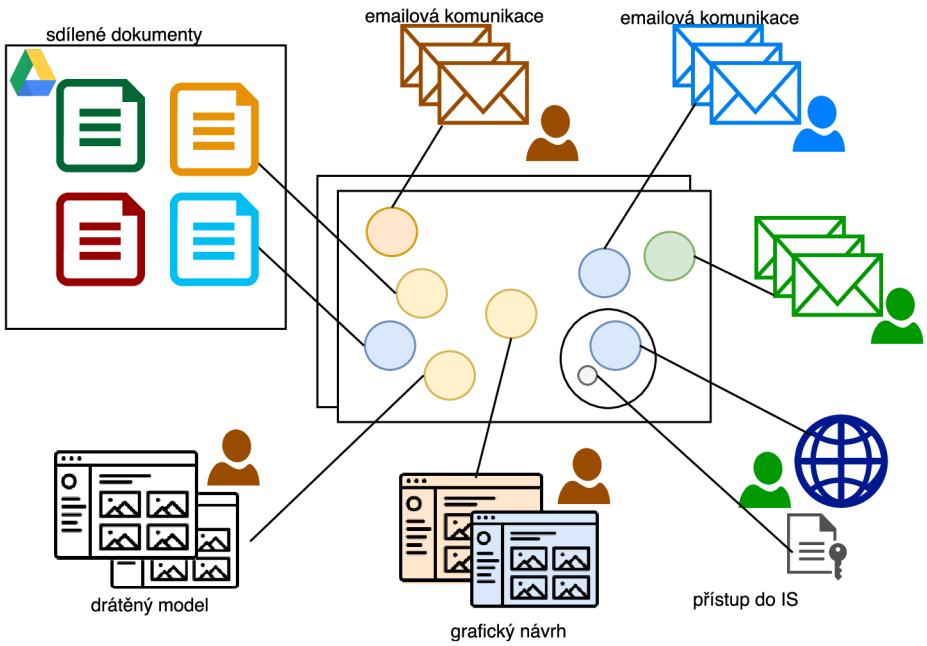
- Dokumenty
  - Zadávací dokumentace
  - Zápisy ze schůzek
  - Vstupní analýza
  - Projektová dokumentace
- Grafické artefakty
  - Drátěný model ve dvou verzích A a B
  - Grafický manuál
  - Grafický návrh ve variantě 1,2 a 3
- Komunikační artefakty
  - Sada emailů
  - Revize specifikace
  - Stávající webové stránky a přístup do stávajícího IS
- Interní artefakty
  - Repozitář
  - Sada úkolů v interním úkolovém systému

Ve výsledku tedy situace a projektové řízení vypadá následovně: Na sdíleném Google Drive úložišti je uložena sada dokumentů.

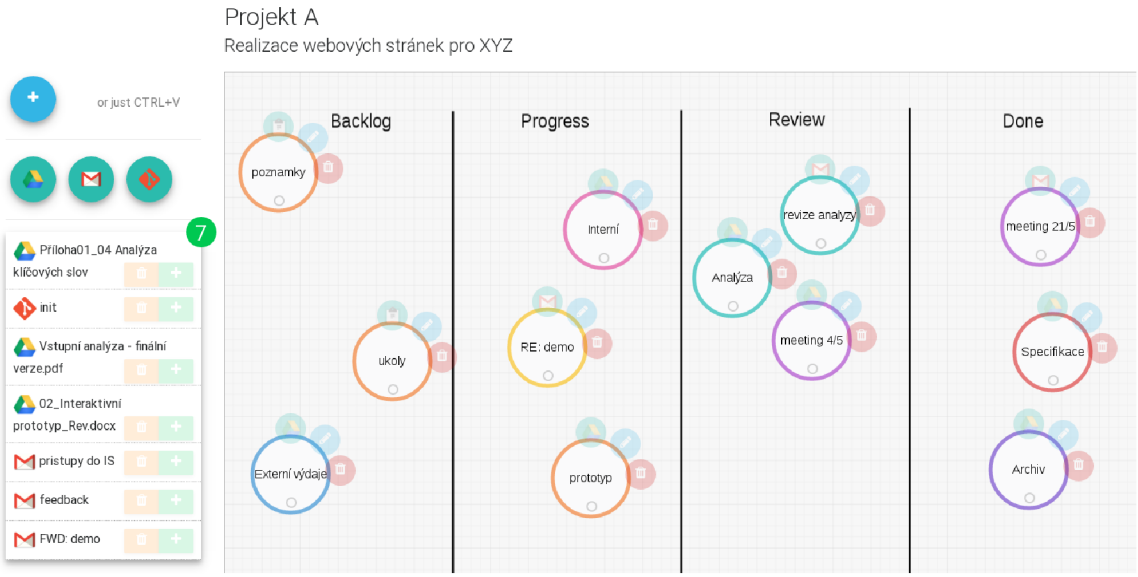
- Grafické artefakty sdílí grafik skrze úkoly formou odkazů do externích služeb a formou přílohy.
- Emaily jsou v emailových schránkách jednotlivých uživatelů bez možnosti sdílení.
- Pro přehlednost demonstrováno obrázkem

Z obrázku je tedy zjevná distribuce dat mezi různými zdroji a je zde vidět i myšlenka integrace do aplikace. Výsledná integrace do aplikace je vidět na následujícím snímku.

[2] při správném použití



Obrázek 6.5: Znázornění organizace zdrojů bez použití aplikace



Obrázek 6.6: Znázornění organizace s využitím aplikace

Kapitola 7

Závěr

Primárním účelem této práce bylo projít celým tvůrčím procesem při tvorbě aplikace, která má za cíl podporovat agilní vývoj a modelování. Na rozdíl od úzce specifikovaných prací s jasně daným zadáním a očekávaným výstupem se zde jednalo o kompletní proces realizace nové služby, a to od prvního nápadu až po výslednou realizaci. V první části jsme zadefinovali řešený problém jako absenci nástroje pro kolaborativní správu zdrojů vstupující do procesu modelování a vývoje softwaru za použití agilní metodiky. Bylo tedy potřeba se vnést do problematiky agilního vývoje a vyrovnat se s nejasným pojmem agilního modelování a jeho přínosem k agilnímu vývoji. Tyto pojmy jsou obecné a mohou se vykládat různě s odlišným přínosem pro projekt. V kapitole 3 jsme při využití teoretické opory Scotta W. Amblera tyto pojmy přesně definovali a ujasnili přínos pro tuto diplomovou práci. Vycházeli jsme ze základních prvků agilního myšlení a určili jsme sadu principů, které jsou přínosné pro výslednou aplikaci a její podporu agilního modelování a vývoje softwaru.

Tyto teoretické poznatky jsme dále v kapitole 4 zapracovali do konceptuálního návrhu řešené aplikace. Jako hlavní přínos aplikace jsme stanovili integraci zdrojů a služeb třetích stran s dodržením jejich sémantiky a jejich vizualizaci na jednom místě ve formě webové aplikace s podporou kolaborativního přístupu. Pro tento cíl a s teoretickou podporou výše uvedených agilních principů jsme vytvořili konceptuální návrh aplikace. Při návrhu jsme se snažili o dodržování doporučených principů Lean Software Development od Erica Riese a Ashe Maurya a výsledný návrh tedy nevycházel čistě z představy autora, ale byl postupně konfrontován s realitou. A to jak v podobě konkurenčního srovnání, tak i při diskuzích s odborníky z praxe, kteří koncept validovali jako potencionální uživatelé. Tento návrh jsme ve dvou iteracích realizovali formou interaktivního prototypu. Tato forma se ukázala jako velice přínosná, poněvadž přesně definovala práci s aplikací, výslednou funkcionalitu a dobře specifikovala výsledný přínos pro uživatele. To se ukázalo jako klíčové při hloubkových rozhovorech s odborníky z praxe.

Po takto definovaném návrhu jsme vybrali potřebné technologie pro celkovou realizaci. V návrhu se ukázalo, že výsledná aplikace bude obsahovat řadu odlišných a vzájemně spolupracujících technologií, zejména z důvodu integrace mnoha služeb třetích stran jako je například IMAP, Google Drive nebo služby jako BitBucket a Github. Pro všechny tyto technologické otázky jsme nalezli konkrétní řešení a zařadili je do architektury aplikace. Pro realizaci jsme zvolili formu webové aplikace za použití jazyka Python a aplikačního rámce Django, který skrze techniku ORM poskytuje kvalitní datovou podporu. Pro uživatelskou spolupráci a sdílení dat jsme čerpali z technologické podpory firmy Google, vybrali jsme tedy Google Realtime API pro datovou kolaboraci a Google Drive pro úložiště. Výsledné

technologie se ukázaly jako vhodná řešení, ale došli jsme k závěru, že pro další vývoj bude potřebné integrovat další vrstvu pro práci s klientskou aplikací.

V kapitole 5, která se zabývá implementací, jsme popsali klíčové technologické aspekty a zejména vzájemnou komunikaci jednotlivých komponent. Během implementace jsme vycházeli z definovaného návrhu, ale při samotné realizaci jsme došli k závěru, že některé prvky bude nutné přehodnotit. Jednalo se zejména o integraci externích služeb. I přes problémy v odlišnostech třetích stran jsme dbali na možnosti dalšího rozšíření pro další rozvoj aplikace. Při popisu implementace jsme vycházeli přímo z demonstrace na zdrojovém kódu, kde jsou nejlépe viditelné myšlenkové postupy a komunikační mechanismy, které jsme chtěli ukázat. V závěru kapitoly jsme se věnovali možnostem nasazení a požadavkům prostředí, které je potřeba splnit, zejména ze strany serveru, aby bylo možné aplikaci provozovat. V poslední kapitole jsme podrobili aplikaci vyhodnocení a porovnání s předpoklady. Došli jsme k závěru, že výsledek odpovídá produktu s nejmenší možnou funkcionalitou, která je potřebná pro potvrzení konceptu, tzv. MVP. Identifikovali jsme silné a slabé stránky výsledné aplikace a navrhli další rozšíření. Pozitiva i negativa jsme našli v technologické i konceptuální rovině, neboť cílem práce není pouze praktická část realizace, ale i část návrhu a tvorby výsledné služby. Silné stránky vidíme zejména v propojení různých služeb a v principech automatizace, naopak slabé stránky v počtu doposud pokrytých třetích stran. S výslednou aplikací se dále předpokládá další vývoj a nasazení do komerčního prostředí.

Literatura

[1] What is Object/Relational Mapping? [Online; navštíveno 15.02.2017].
URL http://hibernate.org/orm/what-is-an-orm/

[2] Migrating from JavaScript. 2012, [Online].
URL https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html

[3] Django project. 2016, [Online; navštíveno 18.11.2016].
URL https://docs.djangoproject.com/

[4] Google Drive REST API Overview. 2016, [Online].
URL https://developers.google.com/drive/v3/web/about-sdk

[5] Google Realtime API Overview. 2016, [Online].
URL https://developers.google.com/google-apps/realtime/overview

[6] *A guide to the project management body of knowledge (PMBOK guide)*. Newtown Square, Penn., USA: Project Management Institute, 2000 vydání, c2000, ISBN 1880410230.

[7] Agutter, C.: *ITIL foundation handbook*. London: TSO, třetí vydání, 2012, ISBN 9780113313495.

[8] Ambler, S. W.: *Agile Modeling*. Wiley, 2002, ISBN 978-0471202820.

[9] Cockburn, A.: *Agile software development*. Boston: Addison-Wesley, 2002, ISBN 0201699699.

[10] Cohn, M.: *Agile estimating and planning*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, c2006, ISBN 0131479415.

[11] Coughlan, G.: MUse cases vs user stories in Agile development. 2012, [Online; navštíveno 2.01.2016].
URL http://www.boost.co.nz/blog/2012/01/use-cases-or-user-stories

[12] Crispin, M.: INTERNET MESSAGE ACCESS PROTOCOL. 2003, [Online].
URL https://tools.ietf.org/html/rfc3501

[13] Elliot, I.: Death Of Flash And Java Applets. 2015, [Online].
URL http://www.i-programmer.info/news/86-browsers/8783-death-of-flash-and-java.html

[14] Foley, M. J.: Microsoft takes the wraps off TypeScript, a superset of JavaScript. 2012, [Online].
URL http://www.zdnet.com/article/microsoft-takes-the-wraps-off-typescript-a-superset-of-javascript/

[15] Kartik Rai, K. A., Lokesh Madan: SOFTWARE CRISIS. 2014, [Online; navštíveno 2.01.2016].
URL http://www.ijirt.org/vol1/paperpublished/IJIRT101671_PAPER.pdf/

[16] Śárka Květoňová: *Studijní opora k předmětu Ekonomie informačních produktů*. Brno, 2010.

[17] Lardinois, F.: Google Launches Drive Realtime API To Let Developers Build Apps With Real-Time Collaboration. 2013, [Online].
URL https://techcrunch.com/2013/03/19/google-launches-drive-realtime-api-to-let-developers-build-apps-with-real-time-collaboration/

[18] Larman, C.: *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003.

[19] Lim, N. H.: PostgreSQL [9.5.0] vs MariaDB [10.1.11] vs MySQL [5.7.0] year 2016. 2016, [Online; navštíveno 5.03.2017].
URL http://nghenglim.github.io/PostgreSQL-9.5.0-vs-MariaDB-10.1.11-vs-MySQL-5.7.0-year-2016/

[20] MAURYA, A.: *Running Lean: Iterate From Plan A to a Plan That Works*. O REILLY & ASSOCIATES, 2012, ISBN 1449305172.

[21] Quinlan, N.: What's a Webhook? 2014, [Online].
URL https://sendgrid.com/blog/whats-webhook/

[22] Ries, E.: *Lean startup*. Brno: BizBooks, první vydání, 2015, ISBN 9788026503897.

[23] Ruprecht, M.: *Agilní modelování při vývoji software*. Diplomová práce, FIT VUT, 2011.

[24] Thomas, D.: Why use UML? Why deploy the Unified Modelling Language. 2014, [Online; navštíveno 19.05.2017].
URL http://dthomas-software.co.uk/resources/frequently-asked-questions/why-use-uml-2/

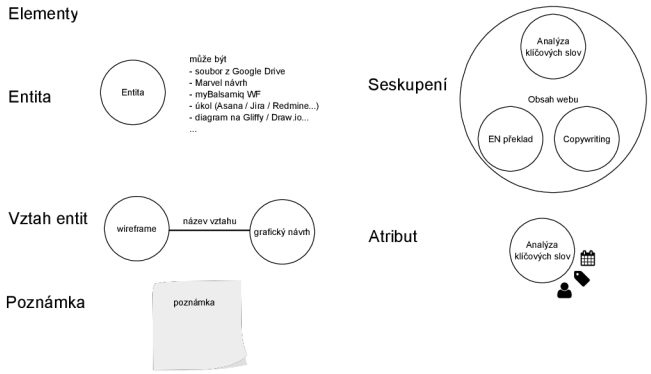
[25] Veverka, P.: *Nástroj pro podporu agilního vývoje softwaru*. Diplomová práce, FIT VUT, 2014.

Přílohy

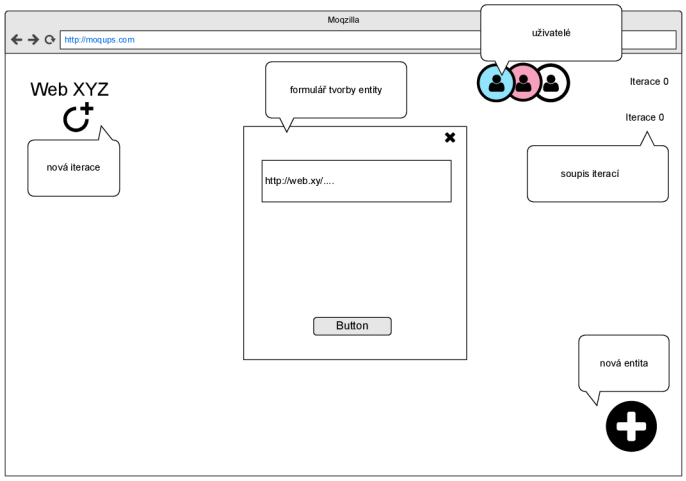
Příloha A

První verze návrhu

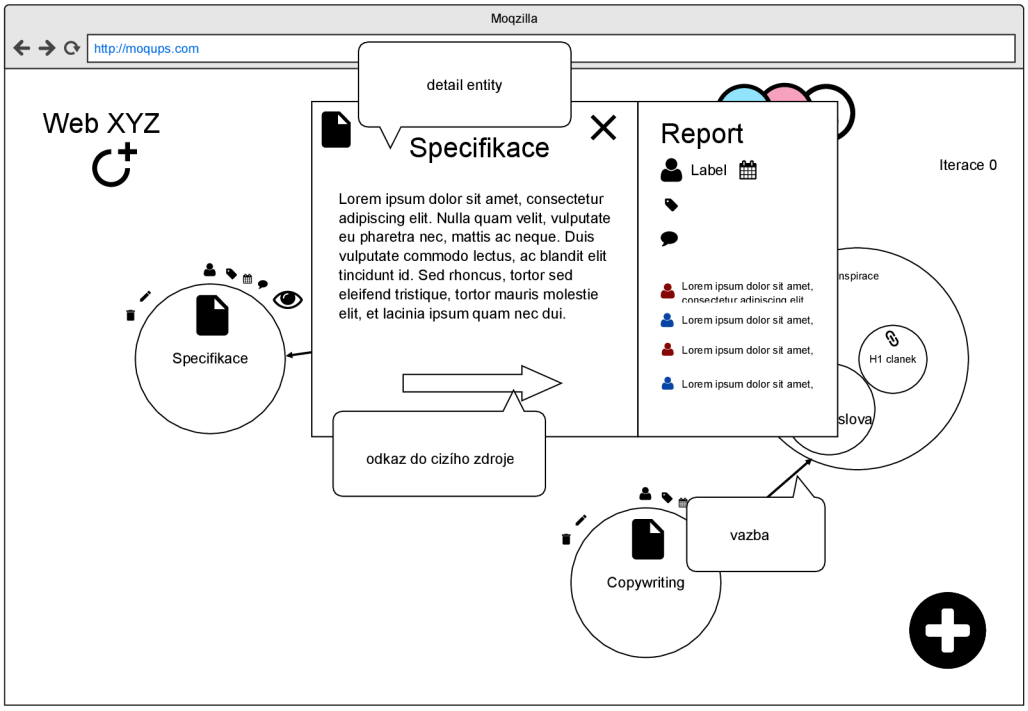
Prototyp je interaktivní na https://goo.gl/VBmX6v



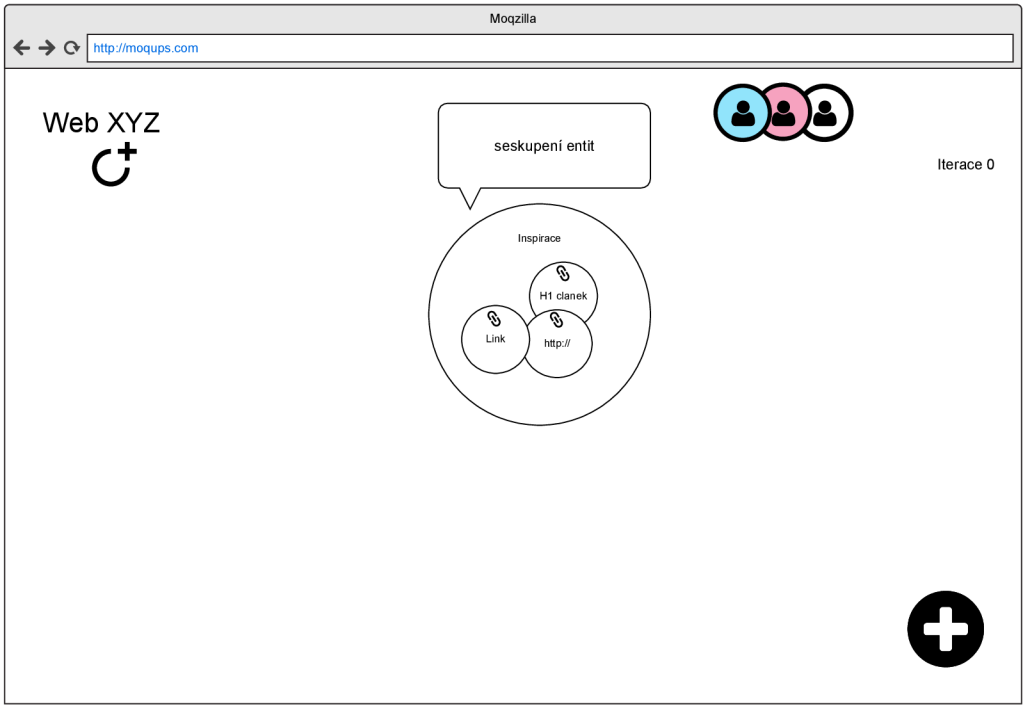
Obrázek A.1: Prvky aplikace



Obrázek A.2: Tvorba nové entity



Obrázek A.3: Detail entity s ovládacími prvky

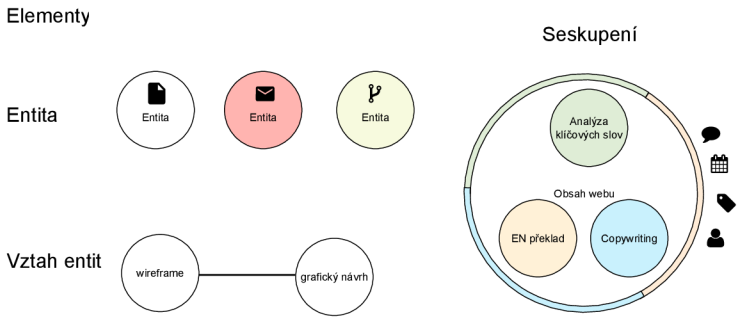


Obrázek A.4: Znázornění seskupení entit

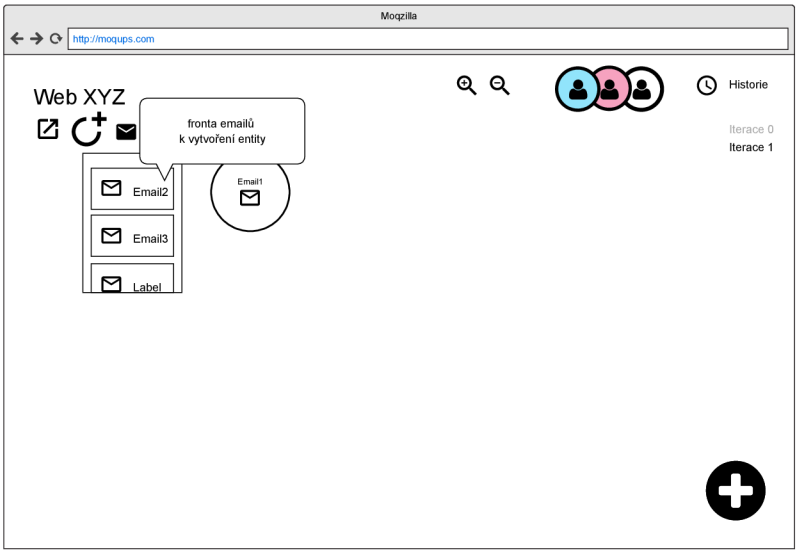
Příloha B

Druhá verze návrhu

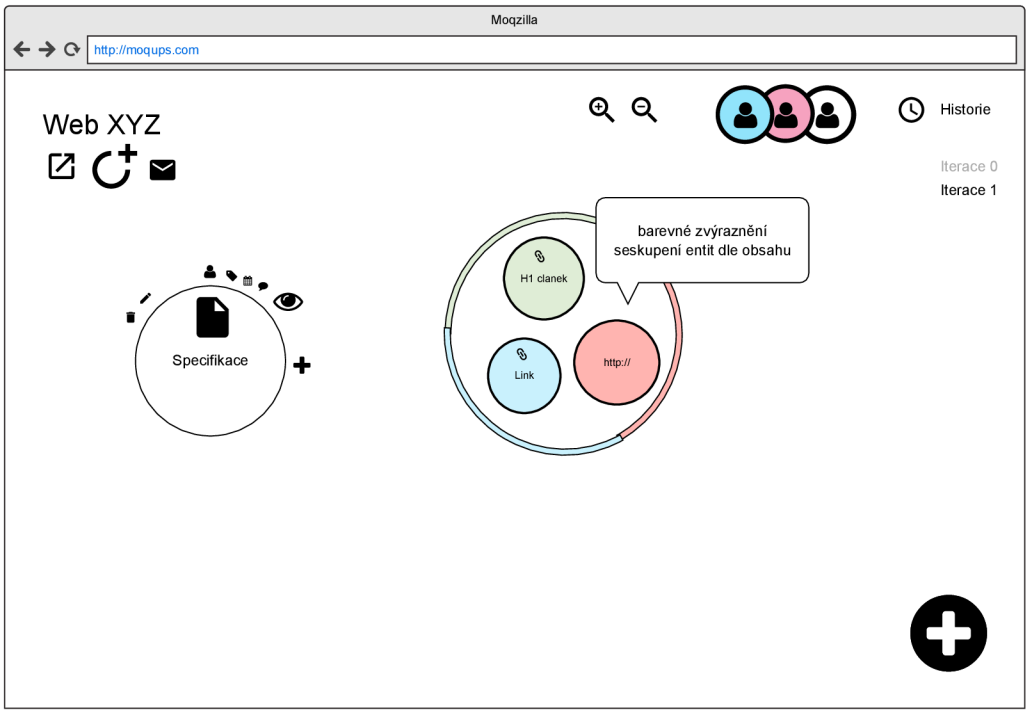
Prototyp je interaktivní na https://goo.gl/VBmX6v



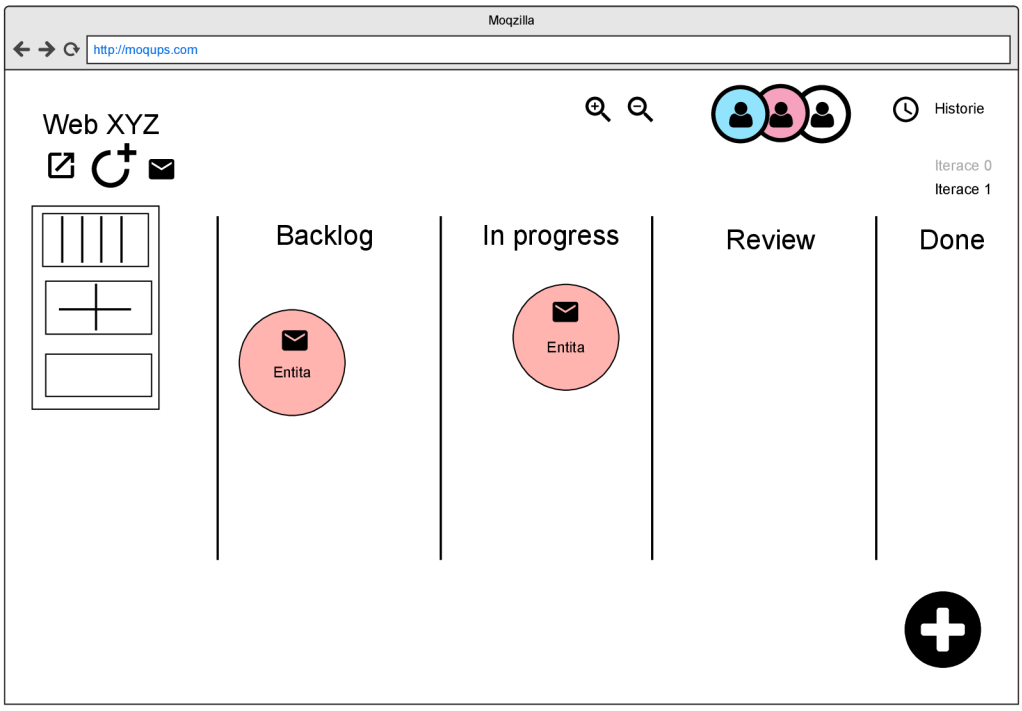
Obrázek B.1: Prvky aplikace



Obrázek B.2: Integrace emailu



Obrázek B.3: Znázornění seskupení entit



Obrázek B.4: Znázornění tvorby různých podkladů a organizace entit

Příloha C

Rozhovory

Jan Řehák

- Mindfulness
- 17.12.2016, restaurace Forkys
- Poznámky
  - Maximálně využít potencál barev
  - Nastavit semantiku barev
  - Seskupení může ukazovat barvy uvnitř (koláčový graf)
  - Velký přehled projektu
  - Rozdělení tabule do sekcí

Ing. Jakub Ludwig

- Mathesio, projektový manažer, Lead developer
- 20.12.2016, ImpactHUB, Brno
- Poznámky
  - Odlišit se od myšlenkových map (Coggle)
  - Odstranit UML – je to zcela nesourodý pojem. UML editor je také entita, která má být externě
  - Budu-li muset dělat věci ručně, pak to nikdo dělat nebude – bude to krok navíc; automatizovat tedy ve stylu Zapier či podobných if-then-else služeb
  - Automatické čtení zdrojů
  - Při čtení zdrojů nastavit zanořitelnost
  - Princip iterace z hlediska „smazání whiteboardu“

Luboš Zíma

- FlowMon networks, vývojář
- 21.12.2016, Kavárna Kafec, Brno
- Poznámky
  - Jednotný komunikační kanál
  - Netvořit entity z externího zdroje automaticky, ale vybírat si skrze filtr

Ing. Milan Doubek

- Dactyl Group, projektový manažer
- 30.12.2016, Dactyl Group, Brno
- Poznámky
  - Seskupení je také entita
  - Nekonečné zanoření entit jako v Asaně
  - Offline soubory jsou také entita
  - Dokumentační úloha – aby se všechno našlo

Bc. Kristýna Kotková

- AISEC, projektová koordinátorka
- 3.1.2017, Brno
- Poznámky
  - Potřeba přehledu více projektů – dashboard
  - Použití předdefinovaných vzorů
  - Snaha o integraci vyší úrovně projektového řízení – termíny, plány
  - Notifikace o změnách entit

Mgr. Michaela Šebestová

- House of Řezáč, projektová manažerka
- 4.1.2017, kavárna Šestá větev, Brno
- Poznámky
  - Komentáře – jako na GoogleDrive
  - Archivace projektů – vyšší úroveň nad entitami
  - Časová osa
  - Integraovat jednoduché kreslení a barvy

Příloha D

Analýza konkurence

	nástroj	klady	zápory
1	Asana asana.com	jednoduchost konverzace organizace do sekcí podrobná historie táhni a pusť časové notifikace a omezení propojení mezi elementy	nulové přílohy špatné odkazování nevýrané vazby mezi elementy
2	Trello trello.com	jasně daná sémantika a cíl import z více třetích stran přímé cílení na Kanban	příliš moc kliků štítky bez semantiky není cílený na vývoj softwaru pouze kanban, minimální svoboda
3	Freelo freelo.cz	česky není cílený na vývoj softwaru	nutí vlastní logiku uživateli
4	ScrumDo scrumdo.com	SCRUM a Kanban dobře vysvětlené táhni a pusť	složité formuláře pouze SCRUM
5	Version one versionone.com	komplexní podnikové řešení	příliš složité vůbec neřeší modelování moc grafů a vstupů
6	OnTimeScrum axosoft.com	stejné výhody jako VersionOne	stejné problémy jako VersionOne
7	Easy Project easyproject.com	komeční varianta Redmine kvalitní support modulární řešení	dosti komplikované prostředí snaha o řešení všeho nepřehledné hlavní body
8	Gliffy gliffy.com	kolaborativní přihlášení přes FB, Google mnoho tvarů (UML, UX...)	bez semantiky nucení vlastního přístupu
9	Draw.io draw.io	totožné s Gliffy integraované s Atlasianem táhni a pusť integrace s dalšími služba,o více stránek nad dokumentem	opět bez semantiky není jasný řešený problém

	nástroj	klady	zápory
10	Creately creately.com	mnoho diagramů a vizualizací Částečná sémantika Promyšlené uživatelské rozhraní	
11	MindMup mindmup.com	Dobré uživatelské rozhraní různý typ obsahu v uzlech Náhledy obsahů Integrace Google Drive	
12	MindMeister mindmeister.com	Výborná vizualizace myšlenkové mapy	
13	Coggle coggle.it	verzování vyborná vizualiazce jednoduše cílené komentáře vyborná uživatelské rozhraní	bez semantiky
14	DokuWiki dokuwiki.org	verzováni plná volnost	minimálni semantika náročné na úpravu
15	XWiki xwiki.org	jednoduchost použití	nutná znalost pro úpravu běžná wiki s dasboardem nulová inovace
16	Keeeb keeeb.com	aregace dat sdílení	pouze poznámky
17	eXo platform exoplatform.com	mířeno na schůzky komlexní	příliš složité
18	Knowledge Plaza knowledgeplaza.net	Obecně pro správu znalostí Podpora různých zařízení	Zaměřeno opět na Wiki Textové a statické
19	Confluence atlassian.com	komletní prostředí Atlassian tagování integrace vícenež 40 služeb notifikace přímá práce ve wiki	komplikované - velké nastavení
20	Redmine redmine.org	umí téměř vše skrze své moduly zaměřeno na organizací úkolů zaměřeno na vývoj softwaru	příliš komplikovaný přehled projektu je čistě textový
21	Mural.ly Mural.ly	whiteboard přiblížení a oddálení kolaborativní tagování lidí konference předdefinované vzory automatické vložení ze schránky vrstvy komentáře	artefakty nejsou aktivní prvky není zaměřený na vývoj softwaru neumí sdíůet artefakty vlastní logika nástrojů není pevná vazba mezi elementy volné a bez sémantiky
22	Zapier zapier.com	přes 700 externích služeb do každé služby se dá přihlásit navržené na procesy výborné GUI	není vývoj softwaru

Příloha E

Instalace a testování aplikace

Pro spuštění instance aplikace v novém prostředí je nutno zajistit následující prerekvizity a provést následující kroky.

E.1 Instalace

E.1.1 Potřebné vybavení prostředí

- GNU/Linux distribuce, testováno na Fedora 24 a CentOS 7.2
  - Lze pracovat i s jinými distribucemi, ale pak je nutné individuálně vyřešit závislosti ručně, bez použití instalačního skriptu
  - Potřebné nástroje a technologie
    * yum nebo dnf
    * Python 2.7+ (aplikace je technicky migrovatelná na python3)
    * aplikace Pip, Git, PostgreSQL, Bower
    * Sluzba crond

E.1.2 Proces instalace

- Naklonujte následující repozitář

  `https://jirkasemmler@bitbucket.org/jirkasemmler/dp-django.git`

- Instalace aplikace
  - Proveďte konfiguraci v `config.sh`
  - Spusťte `./run.sh install` – instalace závislostí, příprava prostředí
    * Případně doinstalujte chybějící závislosti dle Vaší distribuce
    * Dokončete přípravu databáze dle instrukcí
    * Dokončete napojení aplikace na WSGI dle instrukcí
  - Nakonfigurujte `dp/settings.py` – heslo k databázi a `ALLOWED_HOSTS`
  - Spusťte `./run.sh reset` – instalace závislostí, příprava prostředí
  - Spusťte `./run.sh setcron` – zavedení cronjobů
- Spusťte `bower install` pro instalaci externích knihoven

- Lokální aplikaci spustíte `python manage.py runserver`
- Pro spuštění skrze webový server a WSGI rozhraní nastavte server Apache2 (v RHEL distribucích HTTPD) server. Pro napojení například na Ngnix je vhodné postupovat dle manuálu Djanga https://docs.djangoproject.com/en/1.11/howto/deployment/wsgi/modwsgi/

E.1.3 Testování

Pro účely testování aplikace byly vytvořeny následující údaje a scénáře. Jedná se pouze o demonstrační data, data je možné změnit a testovat aplikaci vlastní sadou.

- Instance aplikace je v provozu na http://dp.jirisemmler.eu/
- Přihlašte se údaji
  - Účet: dp.test.semmler@gmail.com
  - Heslo: sherito
- Ve zmíněném účtu je již vytvořen Dashboard
- Pro vyzkoušení jednotlivých služeb postupujte následovně
  - Email – zašlete email na dp-test@seznam.cz s klíčovým slovem „dp“ v předmětu
  - Git – naklonujte si výše uvedený repozitář a proveďte operaci „push“ s klíčovým slovem „release“ ve zprávě změny
  - GoogleDrive – přihlaste se na zmíněný Gmail účet a do složky ProjektB vložte nějaký obsah (třeba vytvořit soubor)
- Nově přidané prvky se zobrazí ve frontě v aplikaci.

E.1.4 Testovací účty

- Seznam email
  - Účet: dp-test@seznam.cz
  - Heslo: sherito
- GoogleDrive úcet
  - Účet: dp.test.semmler@gmail.com
  - Heslo: sherito1993
- Git
  - Testovací Bitbucket účet
    - https://bitbucket.org/dp-sherito/dp-test1/overview
    - Účet: dp.test.semmler@gmail.com
    - Heslo: sherito1993
  - Testovací repozitář : https://bitbucket.org/dp-sherito/dp-test1
  - Heslo: sherito1993
  - Webhook nastavte v Settings/Webhooks dle http://prntscr.com/f4ryc7

Příloha F

Obsah CD

Přiložené CD obsahuje následující adresářovou strukturu

- src/ – zdrojové kódy aplikace
- doc/ – technická zpráva a její zdrojové kódy
- prototyp.pdf – interaktivní prototyp
- licence.txt – licence
- readme.pdf – úvodní dokument popisující strukturu CD a instalaci projektu