

Sem vložte zadání Vaší práce.



ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE  
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ  
KATEDRA SOFTWAREVÉHO INŽENÝRSTVÍ



Bakalářská práce

## **Systém správy úkolů pro jednotlivce a malé týmy**

***Martin Melka***

Vedoucí práce: Ing. Josef Pavlíček, Ph.D.

13. května 2016



---

## Poděkování

Mé poděkování patří vedoucímu práce, panu Ing. Josefu Pavlíčkovi, Ph.D., za jeho cenné rady a konzultace. Děkuji také své rodině a přítelkyni, kteří mi byli morální podporou během celého mého studia.



---

# Prohlášení

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

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

V Praze dne 13. května 2016

.....

České vysoké učení technické v Praze  
Fakulta informačních technologií

© 2016 Martin Melka. Všechna práva vyhrazena.

*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.*

### **Odkaz na tuto práci**

Melka, Martin. *Systém správy úkolů pro jednotlivce a malé týmy*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.



---

## Abstrakt

Tato bakalářská práce se zabývá srovnáním existujících aplikací a tvorbou serverové části nové aplikace pro správu úkolů. Uživatelé této aplikace budou jednotlivci a menší pracovní skupiny, které chtějí přidělovat zodpovědnosti a sledovat průběh práce na společných úkolech. Aplikace umožní lidem sdružovat se do skupin, spolupracovat na sdílených úkolech a zaznamenávat odvedenou práci. Součástí této práce je definice požadavků na aplikaci, srovnání navrhovaného řešení s existujícími aplikacemi, dále návrh, implementace, testování a nasazení aplikace. Výsledkem práce bude serverová část aplikace, vystavující funkcionalitu skrze REST rozhraní.

**Klíčová slova** Správa úkolů, produktivita, organizace týmů, backend, REST, Java Spring Framework

---

## Abstract

The aim of this thesis is to compare available software applications for task management and to subsequently create an original one. The users of this application will be individuals and small-scale workgroups, who need to assign responsibilities for and track the progress of shared tasks. The application will allow users to form groups, work together on shared tasks and report the work

done on them. This thesis consists of a definition of application requirements, comparison of current task management solutions and design, implementation, testing and deployment of the proposed application. The outcome of this thesis will be an application backend, which exposes its functionality through a REST interface.

**Keywords** Task management, productivity, team organization, backend, REST, Java Spring Framework

---

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Cíl práce a metodika</b>	<b>3</b>
1.1 Cíl práce . . . . .	3
1.2 Metodika . . . . .	3
<b>2 Současný stav</b>	<b>5</b>
2.1 Řešení pro firmy . . . . .	6
2.2 Řešení pro střední a menší týmy . . . . .	7
2.3 Řešení pro jednotlivce . . . . .	8
2.4 Shrnutí . . . . .	10
<b>3 Analýza</b>	<b>11</b>
3.1 Požadavky na aplikaci . . . . .	11
3.2 Model případů užití . . . . .	14
3.3 Doménový model . . . . .	25
<b>4 Návrh řešení</b>	<b>29</b>
4.1 Technologie . . . . .	29
4.2 REST API . . . . .	31
4.3 Urgentnost . . . . .	35
4.4 Autentifikace . . . . .	36
4.5 Architektura . . . . .	37
4.6 Návrhový model . . . . .	38
4.7 Databázový model . . . . .	41
4.8 Testování . . . . .	41
<b>5 Implementace</b>	<b>43</b>
5.1 Vývojové prostředí . . . . .	43
5.2 Služby . . . . .	43

5.3	Aktualizace urgentnosti . . . . .	44
5.4	Model . . . . .	44
5.5	Persistence . . . . .	45
5.6	Autentifikace . . . . .	46
5.7	Koncové body . . . . .	46
<b>6</b>	<b>Testování</b>	<b>49</b>
6.1	Mockito . . . . .	49
6.2	Jednotkové testy . . . . .	49
6.3	Integrační testy . . . . .	51
6.4	Výsledky testování . . . . .	53
<b>7</b>	<b>Nasazení</b>	<b>55</b>
7.1	Verze Java . . . . .	55
7.2	Uložiště dat . . . . .	55
7.3	Nasazení WAR archivu . . . . .	55
7.4	Nasazení JAR archivu . . . . .	56
7.5	Veřejně dostupná instance aplikace . . . . .	56
	<b>Závěr</b>	<b>57</b>
	<b>Literatura</b>	<b>59</b>
<b>A</b>	<b>Seznam použitých zkratk</b>	<b>63</b>
<b>B</b>	<b>Splnění kritérií řešerše</b>	<b>65</b>
<b>C</b>	<b>Doménový model</b>	<b>67</b>
<b>D</b>	<b>Detail REST API</b>	<b>69</b>
D.1	Chybové kódy . . . . .	69
D.2	Endpointy . . . . .	70
<b>E</b>	<b>Komponenty aplikace</b>	<b>83</b>
<b>F</b>	<b>Návrhový model</b>	<b>85</b>
<b>G</b>	<b>Databázový model</b>	<b>89</b>
<b>H</b>	<b>Obsah příloženého CD</b>	<b>93</b>

---

## Seznam obrázků

3.1	Diagram účastníků . . . . .	16
3.2	Případy užití nepřihlášených uživatelů . . . . .	17
3.3	Případy užití přihlášeného uživatele . . . . .	20
3.4	Případy užití účastníky úkolu . . . . .	21
3.5	Případy užití skupin . . . . .	22
3.6	Případy užití časových událostí . . . . .	25
4.1	Spring Boot . . . . .	30
4.2	Urgentnost úkolu s termínem . . . . .	37
4.3	Balíček dao . . . . .	39
4.4	Balíček services . . . . .	40
C.1	Doménový model . . . . .	68
E.1	Diagram komponent . . . . .	84
F.1	Balíčky tříd . . . . .	86
F.2	Model tříd . . . . .	87
G.1	Databázový model . . . . .	90
G.2	Databázový model nabídek . . . . .	91



---

# Úvod

Výpočetní technika umožnila rozvoj rychlejší a efektivnější komunikace, práce a vůbec způsobu života. Je běžné mít svůj kalendář on-line a sdílet ho s ostatními, případně používat některý nástroj s funkcí úkolníčku. Zejména ve velkých firmách, kde existuje silná potřeba koordinovat úsilí mnoha lidí, tak vznikla poptávka po nástrojích pro správu úkolů, které by jim umožnily efektivnější rozdělování zodpovědností a práce. Řešení, které na tento popud vznikly, slouží právě potřebám velkých firem. Potřeby menších skupin a jednotlivců jsou ale jiné. Pro ně jsou tyto nástroje příliš komplexní, těžkopádné a nedostatečně intuitivní.

Tato práce se zabývá přehledem existujících řešení pro správu úkolů malých i velkých skupin a tvorbou backendu nové aplikace. Aplikace bude zaměřena na potřeby menších týmů a jednotlivců a bude umožňovat uživatelům vytvářet úkoly, sdílet je s ostatními uživateli, sdružovat se do skupin a zaznamenávat průběh práce na úkolech.





# Cíl práce a metodika

V této kapitole představím cíle této bakalářské práce a metodiku použitou pro jejich dosažení.

## 1.1 Cíl práce

Cílem této práce je vytvořit přehled nejdůležitějších existujících aplikací pro správu úkolů, porovnat je a následně vytvořit prototyp serverové části nové aplikace, která bude přístupná přes rozhraní REST. Bude umožňovat vytvářet soukromé i sdílené úkoly, zakládat pracovní skupiny, přiřazovat zodpovědnosti a sledovat průběh práce na úkolech.

## 1.2 Metodika

Nejprve určím kritéria, na která se budu při tvorbě přehledu existujících aplikací zaměřovat. Poté na základě popularity a vlastní zkušenosti vyberu několik aplikací, které rozdělím do tří kategorií podle jejich zaměření. Ty zhodnotím s přihlédnutím k určeným kritériím a shrnu jejich přednosti a nedostatky.

V další části budu definovat funkční a nefunkční požadavky na aplikaci. Funkční požadavky budou ve formě příběhů (user stories). Na základě nich vytvořím případy užití (use cases) a následně doménový model, zachycující základní entity v aplikaci.

Na základě analytické části navrhnu řešení implementace. To bude zahrnovat výběr konkrétních technologií, návrh rozhraní aplikace, architekturu, model tříd a databáze. Také zde popíšu způsob uchovávání hesel a postup při testování aplikace.

Následně přistoupím k samotné implementaci aplikace za použití navržených technologií a postupů. Během implementace budu zároveň vytvářet automatizované testy. Výstupem této části bude nasaditelná aplikace a její zdrojový kód, dostupné na přiloženém médiu.

## 1. CÍL PRÁCE A METODIKA

---

Nakonec vytvořím instalační příručku, která popíše, jak vytvořenou aplikaci nasadit a jaké prostředí je pro to nutné. Také zde odkážu na běžící, veřejně dostupnou instanci aplikace.

## Současný stav

Způsobů, jak řešit správu úkolů, existuje spousta a liší se podle toho, kdo je má využívat. V této části práce představím několik zástupců pro každou ze tří kategorií. Těmi jsou:

1. řešení pro větší firmy s množstvím pracovníků;
2. řešení pro střední a menší týmy, jednotky až desítky pracovníků;
3. řešení pro jednotlivce.

Zástupce vybírám podle osobních zkušeností a podle výsledků získaných z vyhledávače Google, na základě jejich pořadí a popularity mezi uživateli. U uvedených zástupců uvedu krátký popis a jejich použitelnost cílovou skupinou této práce, tj. menšími týmy a jednotlivci.

Pracuji s tím, že pro cílovou skupinu této práce jsou důležitá následující kritéria:

1. **Nevyžaduje vlastní infrastrukturu, rychlé zprovoznění** – Uživatelé nechtějí spravovat vlastní hardware, kde by jim aplikace běžela. Začít používat aplikaci má být otázka maximálně několika málo minut.
2. **Použitelnost** – Kritérium jsem hodnotil subjektivně, podle mého názoru na pět kvalitativních složek použitelnosti [1]. Uživatel by se neměl ztratit ve funkcích aplikace a měl by být schopen rychle pochopit, jak s aplikací pracovat.
3. **Absence nepotřebné funkcionality** – Aplikace by měla obsahovat jen základní funkce, které uživatel využije. Větší množství funkcí, které uživatele nezajímají, ztěžují orientaci v aplikaci, což souvisí s předchozím bodem.
4. **Použití zdarma** – Aplikace by měla být použitelná zdarma. V případě, že se jedná o *freemium* model, měla by její neplacená část stačit k běžnému používání a neomezovat výrazně uživatele.

5. **REST API** – Aplikace by měla nabízet rozhraní REST API pro možnost vlastní integrace na její funkce.
6. **Otevřený zdrojový kód** – Zdrojový kód aplikace by měl být veřejně dostupný (*open source*).

Uvedený přehled není vyčerpávající, věnuji se jen některým z těch nejznámějších řešení.

### 2.1 Řešení pro firmy

Řešení této kategorie se zaměřují na větší počet uživatelů a mimo základní správu úkolů nabízí často další funkce pro řízení projektů a integraci s dalšími systémy. Používají se zejména v oblasti vývoje software, ale dají se využít i v jiných oblastech.

#### 2.1.1 JIRA

JIRA [2] je software, který nabízí systém pro řízení požadavků (angl. bug tracking či issue tracking) a funkce pro správu projektů. Je možné ho používat jak na vlastní infrastrukturu, tak on-line. V prvním případě je použití zdarma za určitých podmínek<sup>1</sup>, v druhém případě je použití placené.

Nabízí širokou funkcionalitu a např. možnost upravovat podle potřeb životní cyklus úkolů. To ho činí využitelným i mimo vývoj software. Množství nabízených funkcí jde ale nad potřeby cílové skupiny této práce a technicky méně zdatné uživatele může mást. Na úkolech lze pracovat ve více lidech, ale přiřazen může být v jednu chvíli jen jednomu uživateli (*assignee*). Úkolům lze také přiřadit termín splnění (*deadline*).

JIRA nabízí REST API a nemá otevřený zdrojový kód.

#### 2.1.2 Bugzilla

Bugzilla [5] je systém řízení požadavků, který je zaměřen hlavně na vývoj software. Je podobný nástroji JIRA, nicméně nenabízí takovou flexibilitu a i když by mohl být s dobře nastavenou politikou použitý pro správu úkolů u jiných než softwarových projektů, nebylo by použití intuitivní. Samotná správa a práce s úkoly funguje stejně jako u nástroje JIRA.

Bugzilla má otevřený zdrojový kód, licencovaný pod MPL, a lze ji využít zdarma i pro komerční účely. Je nutné ji provozovat na vlastním hardware. Existují hosting služby, které jsou ale neoficiální a placené. Bugzilla nabízí REST API.

---

<sup>1</sup>Zdarma pro veřejně dostupný open-source software projekt[3] a pro neziskové, nevládní, neakademické, nekomerční a sekulární instituce, které by si jinak nemohly software dovolit [4].

### 2.1.3 Redmine

Redmine [6] je systém řízení požadavků, který nabízí více flexibility než Bugzilla a obsahuje i některé nástroje pro řízení projektů. Tyto nástroje mohou být přínosné pro větší projekty, které mají danou strukturu, ale nepočítám s tím, že by cílovou skupinu této práce zajímaly. Funkcionalita, která se týká správy úkolů, je srovnatelná s předchozími dvěma nástroji.

Použití je zdarma, ale je nutné nainstalovat na vlastním hardware. Stejně jako v případě nástroje Bugzilla není Redmine oficiálně dostupný přes internet, soukromé služby, které jej nabízejí, jsou placené. Projekt má otevřený zdrojový kód a nabízí REST API.

## 2.2 Řešení pro střední a menší týmy

Nástroje v této kategorii se snaží cílit na týmy, spíše než celé firmy, a práce s nimi není tak formální. Používat je většinou lze přes internet, není nutná vlastní instalace. Oproti řešením v bodu 2.1 obsahují tyto nástroje zpravidla méně funkcí, chybí hlavně různé manažerské nástroje a integrace s dalšími systémy.

### 2.2.1 Trello

Trello [7] je aplikace, která vznikla v roce 2011. Lze ji používat přes internet. Způsob správy úkolů staví na konceptu *kanban* [8]. Umožňuje vytvářet nástěnky (*boards*), které reprezentují projekty. K nástěnkám lze přizvat další uživatele a pracovat na nich společně. Na nástěnkách se dají vytvářet seznamy (*lists*) a v nich karty (*cards*), které představují nejmenší jednotku práce – úkol. Ten má svou prioritu, termín splnění a zodpovědné uživatele. Těch může být libovolný počet.

Je to funkčně bohatý nástroj s jednoduchým ovládáním i pro netechnické uživatele. Na profilu uživatele lze zobrazit všechny přiřazené karty a ty seřadit podle nástěnky, kam patří, nebo podle jejich termínu.

Trello je možné používat zdarma s libovolným počtem spolupracovníků. Placené varianty přináší určité výhody [9], ale menší týmy se mohou obejít bez nich. Existuje i REST API [10], aplikace nemá otevřený zdrojový kód.

### 2.2.2 Trackie

Trackie [11] je aplikace, která je určena pro správu úkolů na společných projektech. Ty lze zakládat a zvát do nich uživatele, v rámci projektů pak tvořit úkoly. Úkol může být přiřazen vždy jen jednomu uživateli. Funkčností i vzhledem jednoduché na použití, ale některé funkce oproti předchozím nástrojům chybí. Úkolům například nelze nastavit termín splnění. Zobrazit je možné jen úkoly každého projektu zvlášť, nelze zobrazit přehled všech úkolů uživatele.

Aplikace nabízí 30denní zkušební dobu, po její uplynutí je placená. Nenabízí REST API a nemá otevřený zdrojový kód.

### 2.2.3 FogBugz

FogBugz [12] je nástroj řízení projektů, který kromě podpory řízení požadavků nabízí i možnost agilního plánování [13], správu podpory a zpětné vazby zákazníků, vytváření dokumentů ve stylu Wiki a další. Nabídkou funkcí je nejbohatší z trojice nástrojů v této kategorii, i přesto se v aplikaci uživatel neztratí.

Uživatelé spolu mohou pracovat na úkolech (*cases*), které se dělí do projektů (*projects*). Na úkolu lze evidovat informace potřebné pro využití agilní metodiky plánování, včetně *story points*. Úkolu je možné určit zodpovědného uživatele, odhad pracnosti a termín. Na úkolu je možné průběžně zaznamenávat odpracovanou práci a zobrazovat kolik času na něm zbývá.

FogBugz nabízí 7denní zkušební dobu zdarma, poté je nutné za používání platit. Je možné jej používat přes internet i na vlastním hardware. Nabízí REST API a zdrojový kód není otevřený.

## 2.3 Řešení pro jednotlivce

Poslední kategorií jsou nástroje pro správu úkolů jednotlivců. Některé z nich mohou umožňovat sdílení úkolů, takže by se daly zařadit i do kategorie 2.2, nicméně svým zaměřením cílí primárně na využití jako osobní úkolník, proto jsou zařazeny zde. Také jsem do této kategorie zařadil nástroje, které nejsou primárně určeny pro správu úkolů, ale někteří je k tomuto účelu využívají, například kalendář nebo poznámky.

### 2.3.1 Todoist

Todoist [14] je on-line aplikace pro organizaci úkolů, která má uživatele motivovat k lepší produktivitě. Za splněné úkoly jsou přidělovány body (karma), jejichž historický vývoj je možné sledovat v grafu, lze nastavit denní a týdenní cíl a získávat další „odměny“ za jeho splnění. Úkoly lze rozdělit do projektů, nastavit jim termín, prioritu a opakování. Projekty je možné sdílet s dalšími uživateli a dají se zobrazit buď po projektech, ke kterým patří, nebo všechny na jednom místě – ve schránce (*inbox*).

Vytvoření nového úkolu se může provést zadáním textu (v angličtině), aplikace sama rozpozná klíčová slova a není tak nutné nastavovat vlastnosti úkolů ručně. Například heslo *Go jogging at 1 PM every day* vytvoří úkol *Go jogging*, který má termín splnění ve 13 hodin a opakuje se každý den.

Placená verze nabízí větší množství otevřených projektů a úkolů, hledání v úkolech, notifikace a další [15]. Todoist nabízí REST API [16] a nemá otevřený zdrojový kód.

### 2.3.2 Toodledo

Toodledo [17] je on-line aplikace, která kromě úkolů umožňuje vytvářet si zvyky (*habits*). To jsou úkoly, které se opakují ve volitelné dny každý týden, jež mají uživatelům pomoci vypěstovat si a dodržovat dobré návyky. Po vykonání zvyku je možné označit ho za splněný a přidat k jeho splnění číslo nebo hodnocení. Mezi další možnosti patří poznámky (*notes*), seznamy (*lists*), nebo nástiný (*outlines*).

Úkoly lze třídit do složek, nastavit jim termín a prioritu. Spolupráce s dalšími uživateli vyžaduje placenou verzi aplikace a webové rozhraní aplikace je oproti nástroji Todoist poměrně nepřehledné. Toodledo nabízí REST API [18] a nemá otevřený zdrojový kód.

### 2.3.3 Google Inbox Reminders

Inbox [19] je e-mailový klient od společnosti Google, který kromě práce s e-maily umožňuje vytváření jednoduchých úkolů (*reminders*). Ty lze kromě aplikace Inbox také zobrazit v kalendáři Google Calendar [20]. Úkolům nelze nastavit termín splnění, ale je možné je odložit (*snooze*) tak, aby se zobrazily později ve stanovený čas. Aktivní úkoly se zobrazují mezi příchozími e-maily.

Úkoly nelze nijak třídit, ani u nich určovat prioritu. Jediná informace v úkolu je jeho popis. Pro základní potřeby dostačující, ale jinak funkčně chudý nástroj zatím nenabízí API [21] a nemá otevřené zdrojové kódy.

### 2.3.4 Poznámky, kalendář

Posledním zástupcem je kategorie jednoduchých nástrojů, které uvádím pro úplnost, jelikož je stále používá množství lidí. Patří sem všechny nástroje pro psaní poznámek a vytváření událostí v kalendáři, a to jak elektronické, tak papírové.

Některé elektronické nástroje umožňují sdílení poznámek či kalendářů, takže se dají při dobře definovaných pravidlech použít i pro týmovou práci. Jejich použití je ale složitější s rostoucím počtem úkolů a projektů, protože neumožňují žádné filtrování, řazení úkolů ani přiřazování zodpovědností. Pro nenáročného uživatele, který si chce zapisovat nejdůležitější úkoly a sám se postará o to, že na nich nezapomene včas začít, může být toto řešení dostačující.

Pro papírové „nástroje“ platí předchozí odstavec podobně, jen možnost spolupráce je ještě více omezena. A pokud celá skupina nemá společnou místnost pro práci, pak prakticky vyloučena. Navíc vzniká problém s archivací a uchováváním historie úkolů.

### 2.4 Shrnutí

Přehled kritérií stanovených na začátku kapitoly 2 a jejich splnění uvedenými nástroji prezentuji v příloze v tabulce B.1.

Všechny nástroje umožňují vytvářet úkoly, ale v možnostech se liší. Nyní zvážím, které funkce jsou užitečné pro cílovou skupinu této práce a mají být obsaženy ve výsledné aplikaci.

Většina nástrojů umožňuje úkoly sdílet s jinými uživateli a společně s nimi pracovat na projektu. Možnost seskupit se do jednoho celku se spolupracovníky je užitečná, proto budou v aplikaci existovat pro uživatele pracovní skupiny.

V aplikacích je u úkolu možné určit uživatele za něj odpovědné. Většinou lze přiřadit k úkolu v jednu chvíli jen jednoho uživatele, ale v nástroji Trello jich lze přiřadit libovolný počet. To může být pro některé typy úkolů užitečné.

Kromě termínu splnění a priority jako ve většině ostatních aplikací, lze v aplikaci FogBugz u úkolu nastavit i odhad pracnosti – dobu, jakou by měla práce na úkolu trvat. Uživatel pracující na úkolu pak může průběžně zaznamenávat odpracovanou dobu a celá skupina tak vidí, v jaké fázi se úkol nachází. Toto chci zahrnout do výsledné aplikace a rozšířit o další funkci. Na základě doby zbývající do termínu splnění a zbývající pracnosti na úkolu lze určit „urgentnost“, tedy hodnotu, určující „skutečnou“ prioritu úkolu v danou chvíli. Tato informace může uživateli pomoci s rozhodováním, kterému úkolu se věnovat, aby jej stihli dokončit před jeho termínem.

Některé nástroje, například Todoist, nabízí možnost vytvořený úkol opakovat v zadaném intervalu. Google Inbox pak kromě opakování umožňuje také úkol odložit na později, čímž se na stanovenou dobu přestane zobrazovat v přehledu úkolů. V aplikaci chci vyjít z těchto funkcí a vytvořit funkci podobnou. Opakující se úkoly nebudou mít pevně daný termín splnění. Uživatel se jim bude chtít věnovat zhruba po určitých časových intervalech, které nejsou nutně přesné. Při vytvoření úkol nebude důležitý, ale bude postupně nabývat na urgentnosti, až se důležitým stane. Poté uživatel úkol buď ukončí, nebo vynuluje a úkol znovu poroste. Jeden ze způsobů použití takového úkolu může být například občasné zavolání přátelům či rodině.

Serverová část aplikace, jež bude výstupem této práce, bude zaměřením spadat na pomezí sekcí 2.2 a 2.3. Bude obsahovat vybrané funkce pro správu práce týmů i jednotlivců, které analyzuji v kapitole 3.



## Analýza

V této kapitole se budu věnovat specifikování požadavků na aplikaci a jejich analýze. To mi poskytne konkrétní přehled o tom, co má výsledná aplikace umět a jaké entity v ní budou existovat.

### 3.1 Požadavky na aplikaci

Požadavky na aplikaci popíšu pomocí uživatelských příběhů (*user stories*) [22]. Jedná se o jednoduchý způsob, jakým vyjádřit požadavek na aplikaci z pohledu jejího uživatele. Měl by být krátký a napsán v terminologii uživatele budoucí aplikace.

Formální způsob zápisu uživatelského příběhu se dá popsat například takto: [23]

Jako  $\langle \text{role} \rangle$  chci  $\langle \text{něco} \rangle$ , abych dosáhl  $\langle \text{něčeho} \rangle$ .

Pomocí uživatelských příběhů lze také testovat, zda aplikace obsahuje všechny požadované funkce. Pokud jsou všechny příběhy uspokojeny, pak je aplikace z hlediska funkčnosti hotova.

#### 3.1.1 Uživatelské příběhy

##### 1. Uživatel

- 1.1. Jako uživatel chci mít svůj účet, abych mohl spravovat své úkoly a pracovní skupiny. Účet bude jednoznačně identifikován uživatelským jménem a pro přihlášení bude potřeba heslo. Dále bude účet obsahovat jméno a e-mail uživatele. Svůj účet budu moci zobrazit a upravit svoje jméno, heslo a e-mail.
- 1.2. Jako uživatel chci u úkolu mít vždy jednu ze dvou rolí – pozorovatel a pracovník – abych dal najevo, v jakém vztahu k úkolu jsem. Jako

### 3. ANALÝZA

---

pozorovatel mám jen pasivní roli, úkol vidím ve svém seznamu. Jako pracovník se na úkolu aktivně podílím.

- 1.3. Jako uživatel chci vytvářet úkoly soukromé nebo sdílené s ostatními uživateli a pracovními skupinami, abych nezapomněl na své nebo týmové úkoly, které je potřeba vypracovat. Úkol bude obsahovat následující informace:
  - název;
  - popis;
  - datum a čas začátku;
  - pracnost v člověkohodinách (*manhours*);
  - priorita;
  - typ:
    - s termínem – úkol má daný termín (datum a čas), kdy má být splněn;
    - rostoucí – úkol nemá daný termín, ale jeho urgentnost postupně roste.
  - termín splnění (jen pro úkoly s termínem);
  - rychlost růstu urgentnosti (jen pro rostoucí úkoly);
  - stav:
    - otevřený;
    - rozpracovaný;
    - splněný;
    - zrušený.
- 1.4. Jako pozorovatel nebo pracovník úkolu ho chci sdílet s dalšími uživateli.
- 1.5. Jako pracovník úkolu ho chci modifikovat, abych mohl reagovat na případné změny. Změnit budu moct všechny údaje, kromě jeho typu.
- 1.6. Jako uživatel chci znát *urgentnost* úkolů, abych věděl, kterým úkolem se mám věnovat tak, abych stihl jejich termín. Čím méně času zbývá na vypracování úkolu vzhledem k jeho pracnosti a termínu, tím více je urgentní.
- 1.7. Jako uživatel chci zobrazit všechny úkoly, kterých jsem součástí. Tedy moje osobní i sdílené se mnou nebo pracovní skupinou, již jsem členem, abych měl přehled o tom, co je potřeba udělat. Zobrazení by mělo jít filtrovat a řadit podle následujících kritérií:
  - filtr:
    - role (pozorovatel / pracovník);
    - typ;

- stav;
- prioritita.
- řazení:
  - název;
  - datum a čas vytvoření úkolu;
  - termín;
  - rozpracovanost (poměr odpracováno ku celkové náročnosti);
  - prioritita;
  - urgentnost.
- 1.8. Jako uživatel chci přijímat nebo zamítat příchozí návrhy na sdílení úkolů, abych se mohl rozhodnout, na kterých se chci podílet a na kterých ne.
- 1.9. Jako pracovník úkolu na něm chci zaznamenávat odvedenou práci, abych měl já i mí spolupracovníci lepší představu o tom, kolik práce na něm zbývá.
- 1.10. Jako pracovník úkolu chci mít možnost prohlašovat úkol za splněný, uzavřený, nebo znovuotevřený, abych dal najevo, v jakém stavu se úkol nachází.
- 1.11. Jako pracovník na rostoucím úkolu chci vynulovat jeho urgentnost, aby poté, co jsem ho splnil, dočasně přestal být důležitý. Tuto funkci využiji při opakovaných úkolech.
- 1.12. Jako uživatel chci zakládat pracovní skupiny a po založení se stát jejich administrátorem, abych se mohl sdružovat s dalšími uživateli a pracovat společně. Pracovní skupina obsahuje tyto informace:
  - název,
  - popis.
- 1.13. Jako uživatel chci vidět seznam skupin, jichž jsem členem, manažerem nebo administrátorem, a jejich detaily, abych rychle viděl, se kterými skupinami spolupracuji.

## 2. Manažer skupiny

- 2.1. Jako manažer skupiny do ní chci zvát a odebírat z ní uživatele. Členům chci přiřazovat role pozorovatel nebo pracovník na úkolech, které jsou se skupinou sdíleny, abych tím mohl organizovat skupinu a rozdělovat zodpovědnost za práci mezi její členy.
- 2.2. Jako manažer skupiny chci vytvářet úkoly pro skupinu a přijímat nebo zamítat návrhy na sdílení úkolů s mou skupinou, abych mohl určovat, na čem má skupina pracovat.

### 3. ANALÝZA

---

- 2.3. Jako manažer skupiny chci mít možnost odebrat ji z úkolu, který je s ní sdílen, aby úkoly, které již nejsou relevantní, nezůstávaly v seznamu úkolů skupiny.

#### 3. Administrátor skupiny

- 3.1. Jako administrátor skupiny chci mít možnost měnit její název a popis, abych je mohl udržovat aktuální.
- 3.2. Jako administrátor skupiny chci jmenovat manažery skupiny, abych mohl pověřit další uživatele správou skupiny.
- 3.3. Jako administrátor skupiny chci mít možnost přenechat svou roli jinému členovi skupiny, abych mohl skupinu opustit a ta pořád měla administrátora.
- 3.4. Jako administrátor skupiny chci mít možnost skupinu zrušit.

#### 3.1.2 Nefunkční požadavky

1. Aplikace bude dostupná přes REST API. Jedná se o pouze o serverovou část (*backend*), takže k dispozici nebude žádné grafické rozhraní.
2. Aplikace nebude z důvodu bezpečnosti uchovávat hesla jako prostý text. Ověření bude provedeno vhodným způsobem, který znemožní únik hesel.
3. Zdrojové kódy aplikace budou veřejně dostupné pod MIT licenci.

### 3.2 Model případů užití

Uživatelské příběhy ze sekce 3.1.1 nyní budu specifikovat detailněji pomocí modelu případů užití (*use case model*). Uživatelský příběh by měl obsahovat tolik informací, aby i technicky nezkušený uživatel dokázal pochopit, co má aplikace dělat. Oproti tomu případy užití popisují konkrétní interakce uživatele s aplikací, tedy chování, které má aplikace mít, aby splnila potřeby na ní kladené [24].

Případů užití aplikace je zpravidla více než uživatelských příběhů [25]. Příběhy jsou obecnější, a tak na pokrytí jednoho může být potřeba více interakcí s aplikací a tedy více případů užití.

#### 3.2.1 Účastníci

Aplikace bude rozlišovat účastníky na základě několika kritérií a těm bude dostupné různé množství funkcí. Účastníci jsou zobrazeni podle standardu UML na obrázku 3.1. Účastníky jsou v tomto případě uživatelé, kteří budou aplikaci používat. Každý uživatel může v různých případech užití vystupovat jako jiný účastník, v rámci jednotlivých případů užití ale musí být neměnný [26].

Účastníci od sebe mohou dědit, v tom případě potomek může kromě svých akcí vykonávat navíc všechny akce svého rodiče.

### **Nepřihlášený uživatel**

Nepřihlášený uživatel je účastník, který používá aplikaci poprvé, nebo se z ní po předchozím používání odhlásil.

### **Přihlášený uživatel**

Uživatel se stane přihlášeným po zadání svého jména a hesla. Jsou mu dostupné funkce aplikace a může ji používat.

### **Člen skupiny**

Uživatel je Člen skupiny, pokud k této skupině patří. Členové skupiny mohou mít také další role v rámci této skupiny.

### **Manažer skupiny**

Uživatel je Manažer skupiny, pokud mu Admin tuto roli přiřadil. Jako Manažer má více privilegií pro správu skupiny.

### **Administrátor skupiny**

Uživatel je Administrátor skupiny, pokud ji založil, nebo mu současný Administrátor tuto roli přenechal. Jako Administrátor má nejvyšší práva ve skupině a tato role v rámci skupiny patří vždy právě jednomu uživateli.

### **Účastník úkolu**

Účastník úkolu je abstraktní účastník. Reprezentuje uživatele, který je součástí úkolu v nějaké roli.

### **Pozorovatel úkolu**

Pozorovatel úkolu je součástí úkolu v pasivní roli. Na úkolu přímo nepracuje, ale má zájem sledovat jeho průběh.

### **Pracovník úkolu**

Pracovník úkolu je součástí úkolu v aktivní roli. Na úkolu pracuje a má tak zpřístupněny některé funkce navíc oproti Pozorovateli.

### **Čas**

Účastník Čas nereprezentuje žádného fyzického uživatele, ale spouští události, které mají nastat s uplynulým časem [26].

## 3.2.2 Případy užití

Případy užití získané analýzou *uživatelských příběhů* zobrazuji v diagramech případů užití. Rozdělil jsem je do čtyř balíků podle jejich účastníků pro lepší přehlednost.



Obrázek 3.1: Diagram účastníků aplikace

#### 3.2.2.1 Nepřihlášený uživatel

V diagramu 3.2 jsou zachyceny případy užití pro Nepřihlášeného uživatele.

##### UC 1.01 Registrovat se

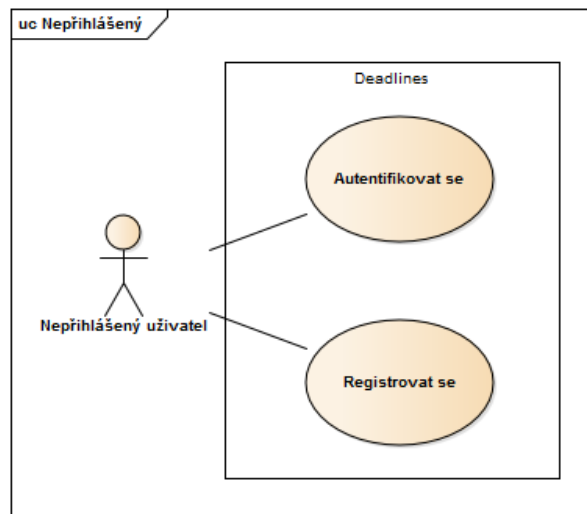
Umožní uživateli aplikace zaregistrovat se. Zadá svoje unikátní uživatelské jméno, e-mail, heslo a skutečné jméno (jméno a příjmení).

##### UC 1.02 Autentifikovat se

Umožní uživateli autentifikovat se pomocí svého uživatelského jména a hesla.

#### 3.2.2.2 Případy užití pro jednotlivce

V diagramu 3.3 a 3.4 jsou zachyceny případy užití aplikace jednotlivci. V prvním diagramu jsou případy užití z pohledu Přihlášeného uživatele v druhém pak z pohledu Pozorovatele a Pracovníka úkolu.



Obrázek 3.2: Diagram případů užití pro nepřihlášeného uživatele

**UC 2.01 Upravit svůj profil**

Umožní uživateli upravit si své jméno, heslo a e-mail.

**UC 2.02 Vytvořit úkol**

1. Scénář začíná, když se uživatel rozhodne vytvořit nový úkol.
2. Uživatel zašle požadavek s informacemi k vytvoření úkolu, povinně název a typ. Pokud je typ „s termínem“, pak je povinný i termín splnění. Pokud je typ „rostoucí“, pak je povinná rychlost růstu. Volitelně může zadat popis, pracnost, prioritu a seznam skupin.
3. Pokud některé povinné informace chybí, scénář končí.
4. Pokud uživatel není alespoň Manažerem všech skupin, které zadal, scénář končí.
5. Aplikace vytvoří úkol, uživatel se stane jeho Pozorovatelem a úkol bude nasdílen všem zadaným skupinám.

**UC 2.03 Sdílet úkol**

1. Scénář začíná, když chce Účastník sdílet úkol.
2. Účastník zašle požadavek obsahující úkol, uživatele a skupiny, se kterými chce úkol sdílet.
3. Aplikace pošle zadaným uživatelům a skupinám nabídku ke sdílení úkolu.

**UC 2.04 Zobrazit seznam mých úkolů**

1. Scénář začíná, když chce uživatel zobrazit své úkoly.
2. <Filtrovat úkoly>
3. <Řadit úkoly>

### 3. ANALÝZA

---

4. Uživatel zašle požadavek na zobrazení úkolů s vybranými filtry a řazením.
5. Aplikace vrátí seznam všech úkolů, kterých je uživatel součástí, se zvolenými parametry. Zobrazené informace budou identifikátor, jméno, priorita, status, urgentnost, termín či rychlost růstu, odpracovanost a typ.

#### **UC 2.05 Filtrovat úkoly**

Úkoly je možné filtrovat podle role uživatele v nich (pozorovatel nebo pracovník), podle jejich typu (s termínem nebo rostoucí), stavu a priority (pět úrovní).

#### **UC 2.06 Řadit úkoly**

Úkoly je možné řadit podle zvoleného kritéria. Tím může být název, datum a čas vytvoření úkolu, termín, rozpracovanost (v procentech odpracováno ku celkové pracnosti), priorita a urgentnost.

#### **UC 2.07 Rozhodnout o příchozí nabídce sdílení úkolu**

1. Scénář začíná, když chce uživatel přijmout nebo zamítnout nabídku ke sdílení úkolu.
2. Uživatel zašle požadavek s jeho rozhodnutím a nabídkou, které se týká.
3. Pokud je nabídka přijata, stává se uživatel pozorovatelem sdíleného úkolu. V opačném případě je nabídka smazána.

#### **UC 2.08 Zobrazit seznam nabídek na sdílení úkolu**

Zobrazí seznam nabídek pro uživatele k připojení se ke sdílenému úkolu, v němž bude u každé nabídky uvedeno uživatelské a skutečné jméno sdílejícího uživatele a název úkolu.

#### **UC 2.09 Založit pracovní skupinu**

1. Scénář začíná, když chce uživatel založit pracovní skupinu.
2. Uživatel zašle požadavek obsahující jméno skupiny a volitelně její popis.
3. Aplikace vytvoří novou skupinu s daným názvem a popisem. Zakládající uživatel se stane jejím Administrátorem.

#### **UC 2.10 Rozhodnout o příchozí nabídce ke vstupu do skupiny**

Uživatel rozhodne, zda chce přijmout nebo odmítnout nabídku ke vstupu do skupiny. Pokud ji přijme, stane se jejím Členem.

#### **UC 2.11 Zobrazit seznam nabídek ke vstupu do skupiny**

Zobrazí seznam nabídek ke vstupu do skupin. U každé nabídky bude uvedeno jméno skupiny, uživatelské a skutečné jméno uživatele, od něhož nabídka pochází.



**UC 2.12 Zobrazit seznam uživatelů**

Zobrazí seznam uživatelů aplikace. U uživatele bude uvedeno jeho identifikátor, e-mail, uživatelské a skutečné jméno.

**UC 2.13 Zobrazit seznam skupin**

Zobrazí seznam skupin v aplikaci. U skupiny bude uveden její identifikátor, jméno a informace o Administrátorovi – identifikátor, uživatelské i skutečné jméno a e-mail.

**UC 2.14 Filtrovat skupiny**

Umožňuje filtrovat skupiny podle toho, zda je uživatel jejich Členem a v jaké roli.

**UC 2.15 Zobrazit úkol**

1. Scénář začíná, když chce uživatel zobrazit detaily úkolu.
2. Uživatel zašle požadavek k zobrazení úkolu.
3. Pokud uživatel není účastníkem úkolu, scénář končí.
4. Aplikace zobrazí všechny informace o úkolu, tedy identifikátor, název, popis, datum a čas vytvoření úkolu, jeho termín, popřípadě rychlost růstu. Dále pracnost, prioritu, typ, stav, urgentnost, záznamy o práci a seznam jeho Pracovníků, Pozorovatelů a zúčastněných skupin.

**UC 2.16 Stát se pracovníkem úkolu**

Umožňuje Pozorovateli úkolu stát se jeho Pracovníkem.

**UC 2.17 Změnit stav úkolu**

Umožňuje Pracovníkovi změnit stav úkolu na jakýkoliv z jeho možných stavů.

**UC 2.18 Upravit úkol**

Umožňuje Pracovníkovi upravit u úkolu jeho název, popis, termín, pracnost a prioritu.

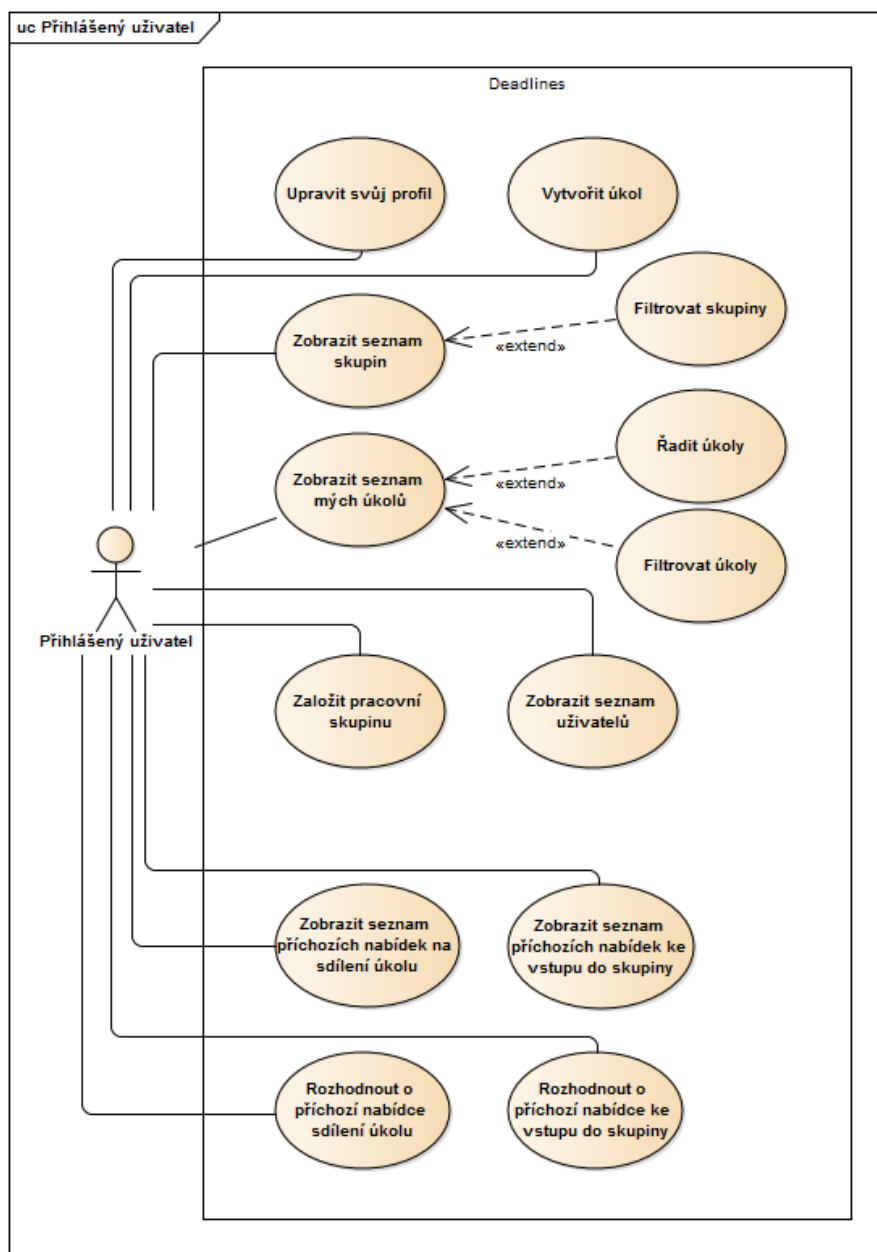
**UC 2.19 Stát se pozorovatelem úkolu**

Umožňuje Pracovníkovi úkolu stát se jeho Pozorovatelem.

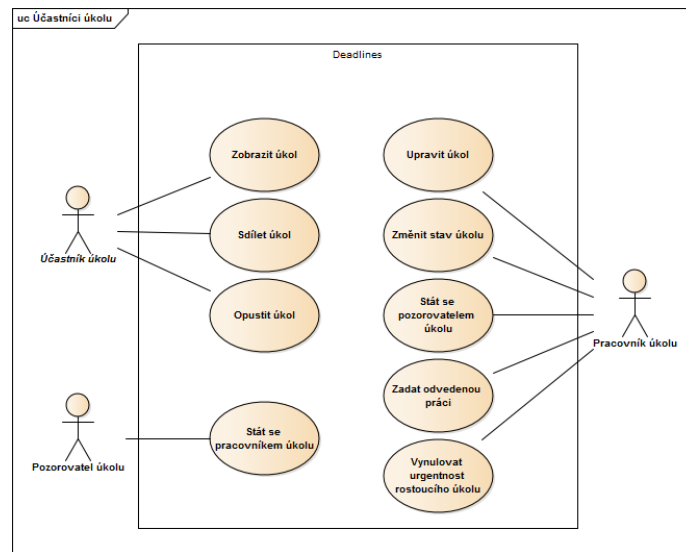
**UC 2.20 Zadat odvedenou práci**

1. Scénář začíná, když uživatel odvedl nějakou práci a chce ji na úkolu zaznamenat.
2. Uživatel zašle požadavek s identifikátorem úkolu a počtem odpracovaných hodin.
3. Pokud uživatel není Pracovníkem daného úkolu, scénář končí.
4. Aplikace k úkolu přidá zadaný počet odpracovaných hodin.

### 3. ANALÝZA



Obrázek 3.3: Diagram případů užití pro přihlášeného uživatele



Obrázek 3.4: Diagram případů užití pro účastníky úkolu

**UC 2.21 Vynulovat urgentnost rostoucího úkolu**

Umožňuje Pracovníkovi vynulovat urgentnost rostoucího úkolu.

**UC 2.22 Opustit úkol**

Umožňuje Účastníkovi úkol opustit ho. Účastníkem úkolu je možné být přímo nebo skrze skupinu. Opustit úkol lze pouze pokud je Účastník součástí úkolu přímo a ne jako člen skupiny.

1. Scénář začíná, když se Účastník úkolu rozhodne opustit úkol.
2. Účastník zašle požadavek na opuštění úkolu.
3. Pokud nebyl úkol sdílen přímo s Účastníkem (ale jen skupinou, ve které je Členem), scénář končí.
4. Aplikace odebere Účastníka z úkolu. Pokud byl ale Účastník navíc součástí úkolu skrze skupinu, jíž je členem, nadále zůstane součástí úkolu skrze tuto skupinu.

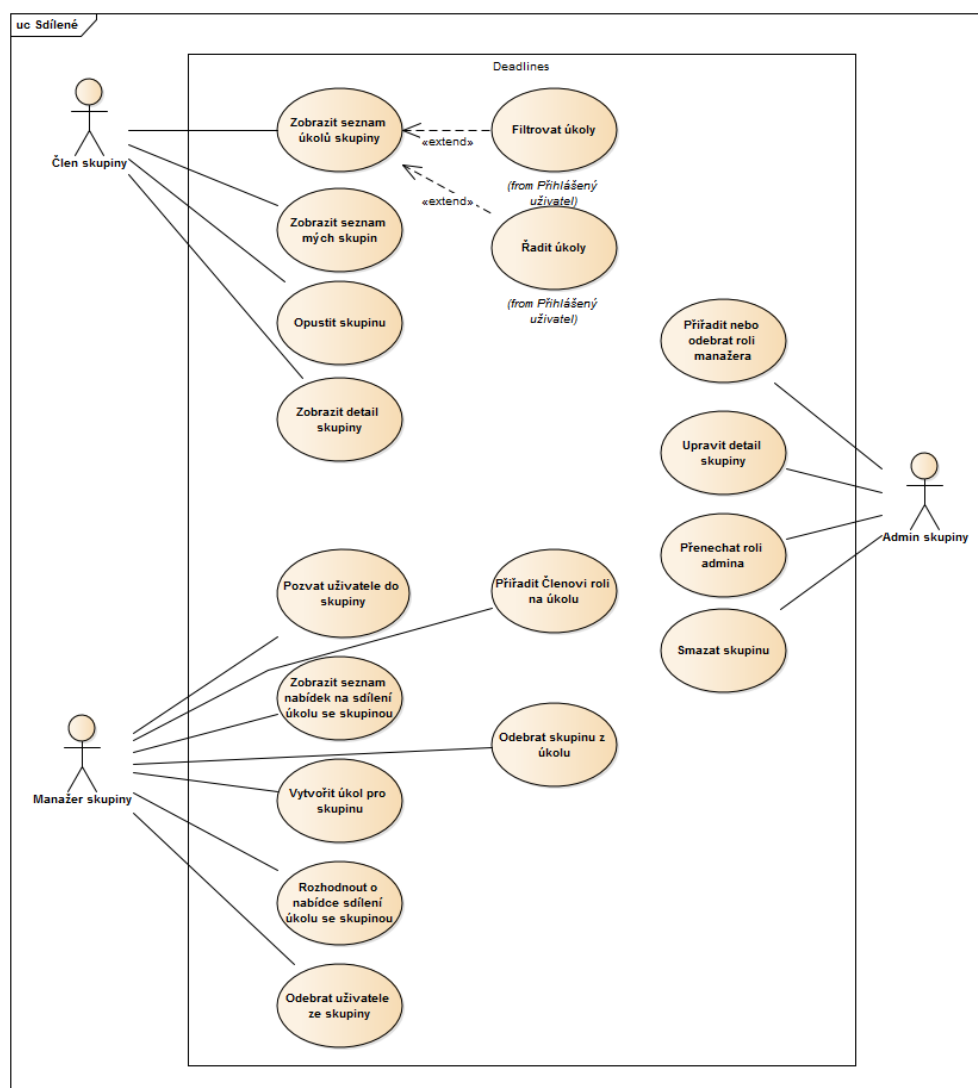
**3.2.2.3 Případy užití pro skupiny**

V diagramu 3.5 jsou zachyceny případy užití aplikace členy pracovních skupin. Týkají se správy skupinových úkolů a skupin samotných.

**UC 3.01 Zobrazit seznam úkolů skupiny**

1. Scénář začíná, když chce Člen skupiny zobrazit úkoly své skupiny.
2. <Filtrovat úkoly>
3. <Řadit úkoly>
4. Uživatel zašle požadavek na zobrazení úkolů dané skupiny se zvolenými filtry a řazením.

### 3. ANALÝZA



Obrázek 3.5: Diagram případů užití pro členy pracovní skupiny

5. Pokud Člen nepatří do zvolené skupiny, je zobrazení odepřeno a scénář končí.
6. Aplikace vráží seznam všech úkolů skupiny se zvolenými parametry.

### UC 3.02 Opustit skupinu

1. Scénář začíná, když chce Člen opustit skupinu.
2. Člen zašle požadavek na opuštění skupiny.
3. Pokud je Člen Administrátorem skupiny, opuštění je zamítnuto a scénář končí.
4. Aplikace odebere Člena ze skupiny a ze všech úkolů, jichž byl součástí skrze opouštěnou skupinu.

### UC 3.03 Zobrazit seznam mých skupin

Zobrazí Členovi seznam názvů všech skupin, ve kterých je Členem.

### UC 3.04 Zobrazit detail skupiny

1. Scénář začíná, když Člen chce zobrazit detaily skupiny.
2. Člen zašle požadavek na zobrazení detailů dané skupiny.
3. Pokud Člen nepatří do zvolené skupiny, je zobrazení odepřeno a scénář končí.
4. Aplikace zobrazí detaily skupiny, tedy její jméno, popis, seznam Členů, Manažerů a Administrátora.

### UC 3.05 Pozvat uživatele do skupiny

1. Scénář začíná, když Manažer chce pozvat uživatele do skupiny.
2. Manažer pošle požadavek obsahující uživatele, které chce pozvat do skupiny.
3. Aplikace vytvoří nabídku ke vstupu do skupiny zadaným uživatelům.

### UC 3.06 Přiřadit Členovi roli na úkolu

Umožňuje přiřadit Členům skupiny roli Pozorovatele nebo Pracovníka na úkolech skupiny.

### UC 3.07 Zobrazit seznam nabídek na sdílení úkolu se skupinou

Zobrazí seznam příchozích nabídek ke sdílení úkolů. U nabídky bude uživatelské a skutečné jméno uživatele, od kterého pochází, a název úkolu, kterého se týká.

### UC 3.08 Rozhodnout o nabídce sdílení úkolu se skupinou

1. Scénář začíná, když Manažer chce rozhodnout o nabídce sdílení úkolu se skupinou.
2. Manažer pošle požadavek na přijetí či odmítnutí nabídky.
3. Aplikace provede požadovanou akci. Pokud Manažer nabídku odmítá, je smazána. Pokud Manažer nabídku přijímá, Aplikace úkol zařadí mezi úkoly skupiny a všichni Členové skupiny se stanou jeho Pozorovateli.

### 3. ANALÝZA

---

#### UC 3.09 Vytvořit úkol pro skupinu

Umožňuje Manažerovi vytvořit nový úkol a ihned ho nasdílet se svými skupinami.

#### UC 3.10 Odebrat uživatele ze skupiny

Umožňuje Manažerovi odebrat zvoleného uživatele ze skupiny. Ten je odebrán ze všech úkolů, jichž byl součástí skrze skupinu. Pokud se jedná o Manažera nebo Admina, je odebrání zakázáno.

#### UC 3.11 Odebrat skupinu z úkolu

1. Scénář začíná, když chce Manažer odebrat skupinu z úkolu, aby už nebyla jeho součástí.
2. Manažer zašle požadavek na odebrání skupinu z vybraného úkolu.
3. Aplikace odebere úkol ze skupiny. U úkolu odebere skupinu a všechny Členy skupiny, pokud nejsou součástí úkolu i mimo skupinu.

#### UC 3.12 Upravit detail skupiny

Umožňuje Administrátorovi měnit popis skupiny.

#### UC 3.13 Přiřadit nebo odebrat roli manažera

Umožňuje Administrátorovi přiřazovat a odebírat roli Manažera u všech Členů skupiny.

#### UC 3.14 Přenechat roli admina

Umožňuje Administrátorovi přenechat svou roli jinému členovi skupiny.

#### UC 3.15 Smazat skupinu

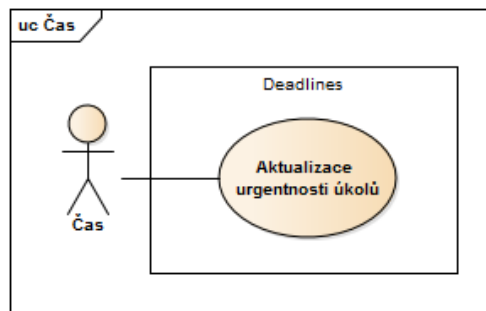
1. Scénář začíná, když se Administrátor rozhodne smazat skupinu.
2. Administrátor zašle požadavek na smazání skupiny.
3. Aplikace odebere všechny ostatní členy ze skupiny.
4. Aplikace odebere skupinu ze všech jejích úkolů.
5. Aplikace odebere Administrátora ze skupiny.
6. Aplikace smaže skupinu.

#### 3.2.2.4 Časově podmíněné případy užití

Diagram 3.6 zachycuje časem podmíněné případy užití, které se pravidelně vykonávají bez přičinění uživatelů aplikace.

#### UC 4.01 Zvýšení urgentnosti úkolů

1. Scénář začíná po uplynutí časového intervalu pro zvyšování urgentnosti úkolů.
2. Aplikace načte všechny úkoly, jejichž stav není Splněný ani Zrušený a které nebyly aktualizovány v nedávné době. Určení „nedávné doby“ je



Obrázek 3.6: Diagram případů užití pro časově podmíněné události

možné v nastavení aplikace.

3. Aplikace vypočte novou hodnotu urgentnosti pro každý z úkolů.

### 3.3 Doménový model

Doménový model slouží k popisu dat, zachycení hlavních entit, se kterými bude aplikace pracovat, a vztahů mezi nimi [27]. Jedná se o abstraktní pohled na problém, a tak diagram obsahuje jen entity a jejich vlastnosti dané předchozími kroky analýzy. Nejsou v něm implementační detaily, jako například konkrétní atributy a metody.

Diagram doménového modelu je zobrazen v příloze na obrázku C.1. Zná-zorňuje vztahy mezi entitami a některá omezení, jako například vzájemnou výlučnost [28]. V této části se dále budu věnovat popisu jednotlivých entit modelu.

#### 3.3.1 User

Entita User reprezentuje uživatele aplikace. Uchovává jeho uživatelské jméno, které je unikátní v rámci aplikace, e-mail, jméno a heslo k jeho přihlášení.

#### 3.3.2 Group

Entita Group reprezentuje pracovní skupinu. Má název a popis. Uživatelé s ní mohou být ve třech různých vztazích – jako Člen, Manažer, nebo Administrátor. S každou skupinou může být uživatel v jednu chvíli jen v jednom z těchto vztahů.

#### 3.3.3 Task

Entita Task reprezentuje úkol. Obsahuje informace stanovené v předchozích krocích analýzy. Uživatel může být součástí úkolu buď jako Pozorovatel, nebo

jako Pracovník.

#### 3.3.4 DeadlineTask

Entita DeadlineTask je typ entity Task, reprezentující úkol s termínem. Obsahuje datum a čas termínu splnění úkolu.

#### 3.3.5 GrowingTask

Entita GrowingTask je typ entity Task, reprezentující rostoucí úkol. Obsahuje informaci o rychlosti růstu.

#### 3.3.6 TaskParticipant

TaskParticipant je entita reprezentující dvojici entit User–Task. Tato dvojice je v aplikaci unikátní, takže nemohou existovat dvě stejné zároveň.

Tato entita popisuje vztah mezi uživatelem a úkolem. Součástí úkolu může být uživatel sám, pokud je nastaven příznak `solo`, nebo jako člen skupiny či skupin, pokud jsou k entitě TaskParticipant přiřazeny.

#### 3.3.7 TaskWork

Entita TaskWork reprezentuje práci vykonanou na úkolu uživatelem. Je spjata s entitami User a Task, takže i když uživatel úkol opustí a odpovídající TaskParticipant zanikne, záznam o jeho práci na úkolu bude stále existovat.

#### 3.3.8 Offer

Entita Offer reprezentuje nabídku, kterou uživatel někomu zasílá.

#### 3.3.9 TaskSharingOffer

Entita TaskSharingOffer je typ nabídky, která reprezentuje nabídku ke sdílení úkolu.

#### 3.3.10 UserTaskSharingOffer

Entita UserTaskSharingOffer je typ nabídky ke sdílení úkolu, která je určena pro uživatele. Vznikne, pokud jeden uživatel k úkolu, na kterém pracuje, pozve jiného uživatele.

#### 3.3.11 GroupTaskSharingOffer

Entita UserTaskSharingOffer je typ nabídky ke sdílení úkolu, která je určena pro skupinu. Vznikne, pokud jeden uživatel k úkolu, na kterém pracuje, pozve skupinu.



#### **3.3.12 MembershipOffer**

Entita MembershipOffer je typ nabídky, která reprezentuje nabídku ke vstupu do skupiny. Vznikne, pokud Manažer skupiny pozve nového uživatele.



## Návrh řešení

V této kapitole představím technologie zvolené k vypracování aplikace a navrhnu REST API, které bude aplikace nabízet. Dále se budu věnovat architektuře aplikace, návrhovému modelu tříd a databázovému modelu. Tím vytvořím podklad pro následující implementaci prototypu aplikace pomocí zvolených technologií.

Název aplikace vytvořené v rámci této práce bude *Deadlines*.

### 4.1 Technologie

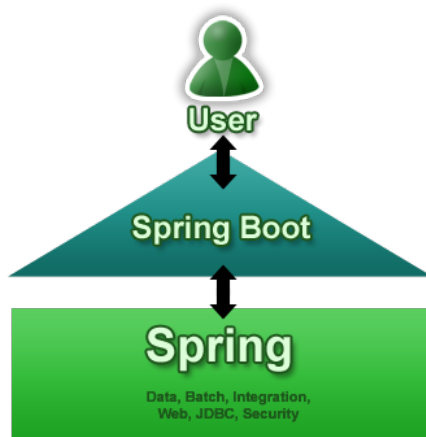
#### 4.1.1 Java

Již v zadání práce je avizováno, že aplikace bude postavena na platformě Java [29]. Jazyk Java byl vyvinutý společností Sun Microsystems a veřejnosti představen v roce 1996. Syntaxí je podobný C/C++ a je silně objektově orientovaný. Program napsaný v Javě je kompilován do podoby *bytecode*, který je následně spouštěn na *Java Virtual Machine* (JVM), který bytecode interpretuje. To znamená, že jazyk je platformně nezávislý – *bytecode* vzniklý kompilací zdrojového Java kódu je možné spustit na jakékoliv platformě, pokud na ní je přítomný odpovídající *JVM*.

Oproti jazykům C/C++ navíc Java disponuje automatickou správou paměti, nazvanou *garbage collection*, takže vývojáři se nemusí starat o životní cyklus objektů a případné úniky paměti (*memory leaks*) [30, 31].

#### 4.1.2 Spring

Spring je framework, který zjednodušuje tvorbu enterprise aplikací v Javě [32]. Jednou z výhod tohoto nástroje je jeho podpora návrhového vzoru *dependency injection*, díky kterému je program lépe modularizovatelný, testovatelný a znovupoužitelný [33].



Obrázek 4.1: Vztah Spring a Spring Boot [36]

Další součástí frameworku Spring je webový modul *Web MVC framework*. Ten umožňuje tvorbu webových aplikací podle architektonického vzoru MVC. Je také možné jej použít pro tvorbu *RESTful* aplikace, což je cílem implementační části této práce.

### 4.1.2.1 Spring Boot

Nadstavbou k frameworku Spring je projekt Spring Boot [34]. Vztah mezi nástroji Spring a Spring Boot je zobrazen na obrázku 4.1. Byl vyvinut k usnadnění tvorby Spring aplikací, hlavně v jejich počátečních fázích. Představuje více „dogmatický“ (*opinionated*) pohled na Spring, což znamená, že od vývojáře očekává „standardní“ chování podle zaběhlých postupů [35]. Vývojáře odštiňuje od nutnosti (nejen) počáteční konfigurace a nabízí navíc některé funkce, jako například vestavěný server *embedded server*, distribuovaný ve výsledném balíčku aplikace. V případě, že konfigurace, kterou Spring Boot poskytuje nevyhovuje, je možné si ji ručně přizpůsobit [36].

Spring Boot také umožňuje jednoduše sestavit aplikaci do podoby jednoho JAR souboru („*fat jar*“), který obsahuje vše, co je nutné k jejímu spuštění. Díky vestavěnému serveru není nutné mít vlastní aplikační server, jako například Tomcat či GlassFish. Lze zvolit způsob distribuce aplikace, takže je možné ji zabalit i do WAR souboru a nasadit na existujícím serveru.

### 4.1.3 Hibernate

Hibernate [37] je framework pro platformu Java, usnadňující řešení persistence (uchovávání) dat. Jedná se o nástroj pro ORM (objektově-relační mapování), který se stará o transformaci objektů aplikace do podoby vhodné pro uložení do

relační databáze, jejich uložení, a následné načtení a transformaci z databáze zpět na objekty. Způsob mapování objektů do databáze lze nastavit buď v konfiguračním XML souboru, nebo přímo ve zdrojovém kódu anotacemi.

Kromě usnadnění práce s databází je další výhodou Hibernate možnost jednoduché změny úložiště. Kód aplikace komunikuje s API Hibernate stejným způsobem, nehlédě na typ databáze, ke které je framework připojen.

#### 4.1.4 MySQL

Uložení dat aplikace bude zprostředkováno relační databází. Pro výběr databáze jsem se řídil jejich popularitou a podmínkou použití zdarma. Nejpopulárnější takovou databází je MySQL [38]. Aplikace nebude mít takové požadavky na persistentní vrstvu, aby nějak ovlivnily výběr databáze. Funkce, které budou potřeba, splňují všechny uvažované možnosti. Větší popularita MySQL bude znamenat lepší podporu komunity a snazší řešení problémů během implementace a nasazení.

#### 4.1.5 Apache Tomcat

Pro nasazení aplikace bude použit server Apache Tomcat [39]. Je nenáročný na hardware a jednoduchý na zprovoznění.

## 4.2 REST API

V této sekci nejprve krátce vysvětlím pojmy API a REST API. Poté přistoupím k návrhu konkrétního REST API pro aplikaci Deadlines.

### 4.2.1 API

API (*Application Programming Interface*) slouží k jednoduché a spolehlivé komunikaci mezi aplikacemi nebo moduly aplikace. API představuje kontrakt, kterým se *producent* zavazuje k nabízení svých služeb v dané zdokumentované podobě, jež mohou být využity *konzumenty*.

Autoři knihy *APIs: A Strategy Guide* [40] definují API takto (přeloženo mnou z anglického originálu):

*API je v podstatě kontrakt. Ve chvíli, kdy tento kontrakt existuje, vývojáři tíhnou k využívání API, protože ví, že se na něj mohou spolehnout. Kontrakt zvyšuje důvěru, což zlepšuje použitelnost. Tento kontrakt také zefektivňuje spojení mezi producentem a konzumentem, protože rozhraní jsou dokumentována, jsou konzistentní a předvídatelná.*

### 4.2.2 REST

Termín REST (REpresentational State Transfer) byl poprvé zaveden v roce 2000 Royem Fieldingem v jeho dizertační práci [41]. Ve zkratce se jedná o

koncept designu distribuované architektury. Definuje rozhraní, které je užitečné pro jednotný přístup ke zdrojům (*resources*). Těmi mohou být data nebo stavy aplikace, popsatelné konkrétními daty. Zdroje v kontextu REST jsou identifikované pomocí URI a komunikovány mezi klientem a serverem v podobě reprezentací v libovolné textové, strojově zpracovatelné podobě (například JSON nebo XML) [42].

Popis architektury REST není předmětem této práce, takže se mu dále nebudu věnovat.

### 4.2.3 REST API aplikace Deadlines

Kompletní popis API, včetně požadovaného obsahu požadavků a možných odpovědí, je vypsán v příloze v sekci D. Zde uvedu možné odpovědi aplikace na požadavky, dostupné koncové body (*endpoints*) a popíšu jejich funkci.

Pokud není uvedeno jinak, všechna volání API musí být autentifikovaná použitím *HTTP Basic Access Authentication* [43]. Jméno a heslo pro *Basic Authentication* jsou uživatelské jméno a heslo uživatele.

Těla požadavků i odpovědí jsou ve formátu JSON, obsah hlavičky *Content-Type* je *application/json*.

#### 4.2.3.1 Odpovědi

Každý koncový bod odpovídá na požadavek odpovědí, která obsahuje standardní HTTP kód popisující, jak volání dopadlo, a případné tělo obsahující data odpovědi. Zde krátce popíšu některé z HTTP kódů:

**200 OK** Požadavek proběhl v pořádku.

**201 CREATED** Požadovaný objekt či záznam byl vytvořen. V hlavičce odpovědi s názvem *Location* je uvedena relativní cesta k nově vytvořenému zdroji.

**400 BAD REQUEST** V požadavku je chyba. Detaily o konkrétní chybě jsou obsaženy v těle zprávy.

**401 UNAUTHORIZED** Požadavek neobsahuje autentifikační údaje, nebo jsou údaje chybné.

**403 FORBIDDEN** Volající uživatel nemá dostatečná práva k provedení požadavku.

**404 NOT FOUND** Požadovaný zdroj nebyl nalezen.

**409 CONFLICT** Objekt či záznam nemohl být vytvořen, protože již existuje.

Přehled všech HTTP kódů je k dispozici na stránkách W3 konsorcia [44].

Chybové odpovědi mohou obsahovat v těle odpovědi další informace o chybě. Tělo má následující formát:

```
{
  "errorCode": 0,
  "errorMessage": "Sample error message."
}
```

Výčet možných chybových kódů aplikace se nachází v příloze v sekci D.1.

#### 4.2.3.2 Koncové body

V této sekci popisují všechny koncové body (*endpoints*) nabízené aplikací. U každého koncového bodu uvádím jeho adresu, povolené HTTP metody a jeho funkci. Detailní popis všech koncových bodů, jejich parametrů a formátu požadavků a odpovědí je k dispozici v příloze v sekci D.2.

##### **/user [GET, POST]**

GET vrátí seznam všech uživatelů v systému.

POST vytvoří nového uživatele z poskytnutých dat.

Tento *endpoint* je přístupný i neautentifikovaným voláním.

##### **/user/{user\_id} [GET, PUT]**

GET vrátí informace uživatele s ID `user_id`.

PUT upraví profil uživatele s ID `user_id` předanými daty. Povoleno je měnit pouze profil volajícího uživatele.

##### **/task [GET, POST]**

GET vrátí seznam všech úkolů volajícího uživatele, které odpovídají zadaným parametrům.

POST vytvoří nový úkol z poskytnutých dat. Volající uživatel se stane jen Pozorovatelem.

##### **/task/{task\_id} [GET, PUT]**

GET vrátí detailní informace o úkolu se zadaným ID. Povoleno jen pro Pracovníky úkolu.

PUT upraví úkol se zadaným ID podle informací v těle požadavku. Povoleno jen pro Pracovníky úkolu.

##### **/task/{task\_id}/reseturgency [POST]**

Vynuluje urgentnost zadaného úkolu. Použitelné jen pro rostoucí úkoly. Povoleno jen pro Pracovníky úkolu.

##### **/task/share/{task\_id} [POST]**

Zašle nabídku na sdílení úkolu uživatelům a skupinám předaným v těle požadavku. Povoleno jen pro Pracovníky úkolu.

#### 4. NÁVRH ŘEŠENÍ

---

##### **/task/leave/{task\_id} [POST]**

Odebere volajícího uživatele z úkolu se zadaným ID.

##### **/task/role/{task\_id} [POST]**

Nastaví roli volajícího uživatele v úkolu se zadaným ID na roli předanou v parametru. Pozorovatel se tak může stát Pracovníkem a naopak.

Volitelně lze také zadat ID skupiny a uživatele, jehož se má volání týkat. Pokud je volající uživatel Manažerem zadané skupiny a cílový uživatel jejím členem, pak je Manažerovi umožněno změnit roli uživatele.

##### **/task/report/{task\_id} [POST]**

Vytvoří záznam o odvedené práci volajícím uživatelem na úkolu s předaným ID. Detaily záznamu jsou obsaženy v těle požadavku. Povoleno jen pro Pracovníky úkolu.

##### **/offer/task/user [GET]**

Vrátí seznam nabídek ke sdílení úkolu pro volajícího uživatele.

##### **/offer/task/user/resolve/{task\_offer\_id} [POST]**

Přijme nebo zamítne nabídku ke sdílení úkolu se zadaným ID. Volající uživatel musí být příjemcem nabídky.

##### **/offer/task/group/{group\_id} [GET]**

Vrátí seznam nabídek ke sdílení úkolu pro skupinu se zadaným ID. Povoleno jen pro Manažery skupiny.

##### **/offer/task/group/{group\_id}/resolve/{task\_offer\_id} [POST]**

Přijme nebo zamítne nabídku ke sdílení úkolu se skupinou se zadaným ID. Povoleno jen pro Manažery skupiny.

##### **/offer/membership [GET]**

Vrátí seznam nabídek ke vstupu do skupiny pro volajícího uživatele.

##### **/offer/membership/resolve/{membership\_offer\_id} [POST]**

Přijme nebo zamítne nabídku ke vstupu do skupiny s předaným ID. Volající uživatel musí být příjemcem nabídky.

##### **/group [GET, POST]**

GET vrátí seznam skupin v aplikaci. Parametrem lze určit, zda vracet všechny skupiny, nebo jen ty, jichž je volající členem s danou rolí.

POST vytvoří novou skupinu z informací předaných v těle požadavku. Volající uživatel se stane jejím Administrátorem.

##### **/group/{group\_id} [GET, PUT, DELETE]**

GET vrátí detaily skupiny se zadaným ID. Povoleno jen pro Členy skupiny.

PUT upraví detaily skupiny se zadaným ID daty předanými v těle



požadavku. Povoleno jen pro Administrátora skupiny.

DELETE smaže skupinu se zadaným ID. Povoleno jen pro Administrátora skupiny.

**/group/{group\_id}/member/offer [POST]**

Zašle nabídku ke vstupu do skupiny se zadaným ID uživateli předaným v těle požadavku. Povoleno jen pro Manažery skupiny.

**/group/{group\_id}/member/{user\_id} [PUT, DELETE]**

PUT upraví záznam o členovi skupiny, identifikovaného jeho ID. Změní jeho roli ve skupině na tu předanou v požadavku. Povoleno jen pro Manažery skupiny.

DELETE Odebere uživatele s ID `user_id` ze skupiny s ID `group_id`. Povoleno jen pro Manažery skupiny.

**/group/{group\_id}/task/{task\_id} [DELETE]**

Odebere skupinu s ID `group_id` z úkolu s ID `task_id`. Povoleno jen pro Manažery skupiny.

## 4.3 Urgentnost

Jedním z funkčních požadavků je výpočet *urgentnosti* úkolů. Urgentnost má vyjadřovat důležitost úkolu v daném momentě a má pomoci uživateli rozhodnout se, na kterém úkolu bude pracovat. Čím vyšší hodnota, tím je nutnější se danému úkolu věnovat. Výpočet hodnoty urgentnosti bude záviset na typu úkolu. Způsob výpočtu pro oba typy úkolů popíšu ve zbytku této sekce.

Hodnota urgentnosti bude reálné číslo v intervalu  $\langle 0; 200 \rangle$ . Hodnota 100 značí, že by měl uživatel na úkolu začít pracovat v tuto chvíli. Např. pro úkol s termínem urgentnost 100 znamená, že do termínu v tuto chvíli zbývá stejně času, jako je odhad zbývající pracovní doby úkolu.

### 4.3.1 Rostoucí úkol

Rostoucí úkol obsahuje informaci o rychlosti růstu urgentnosti. Tato hodnota představuje, za kolik hodin má být urgentnost rovna 100. Růst urgentnosti je lineární a řídí se vztahem

$$u = \frac{1}{hrs} \cdot z \cdot 100$$

kde  $u$  je hodnota urgentnosti,  $h$  je rychlost růstu a  $z$  je počet hodin, které uběhly od vytvoření či vynulování úkolu.

### 4.3.2 Úkol s termínem

Úkol s termínem dosáhne urgentnosti 100, pokud do termínu splnění zbývá přesně tolik času, jako je odhadovaná pracovní doba úkolu.

Pro úkol s termínem definuji rezervu  $r$  následujícím způsobem:

$$r = \frac{t}{w} + \frac{t}{d} \quad (4.1)$$

kde  $t$  je čas zbývajících do termínu splnění v hodinách,  $w$  je zbývajících pracnost v hodinách a  $d$  je konstanta. Experimentálně jsem stanovil, že  $d = 5$ .

První zlomek udává, v jakém poměru je zbývajících čas ku pracnosti. Například pokud do termínu zbývá 8 hodin a zbývajících pracnost jsou 2 hodiny, výsledkem bude 4. Druhý zlomek slouží k úpravě rezervy v závislosti na zbývajícím čase. Čím méně času zbývá, tím menší bude tento člen. To z důvodu, aby rezerva u úkolu, kterému zbývá více času, byla větší, než u úkolu, který má času méně, i když jsou poměry času ku práci stejné. K této úpravě jsem přistoupil, protože chci, aby úkoly blíže termínu měly větší urgentnost než stejně rozpracované úkoly dále od termínu.

Chci, aby hodnota urgentnosti rostla nejrychleji s rezervou pohybující se kolem čísla 1; urgentnost úkolů s velkou kladnou nebo zápornou rezervou není tak důležitá, jako urgentnost těch úkolů, na kterých by uživatel měl začít pracovat v tuto chvíli – ty mají rezervu blízko číslu 1.

Urgentnost úkolu s termínem se tak řídí následující funkcí:

$$f(r) = \begin{cases} |(a^{-r+1} - 1) \cdot b| + b + c & \text{pro } r \leq 0 \\ a^{r-1} \cdot b + c & \text{pro } r \geq 0 \end{cases} \quad (4.2)$$

kde  $a$ ,  $b$  a  $c$  jsou konstanty, které jsem empiricky zvolil jako  $a = 1.1$ ,  $b = 100$ ,  $c = 0$ .

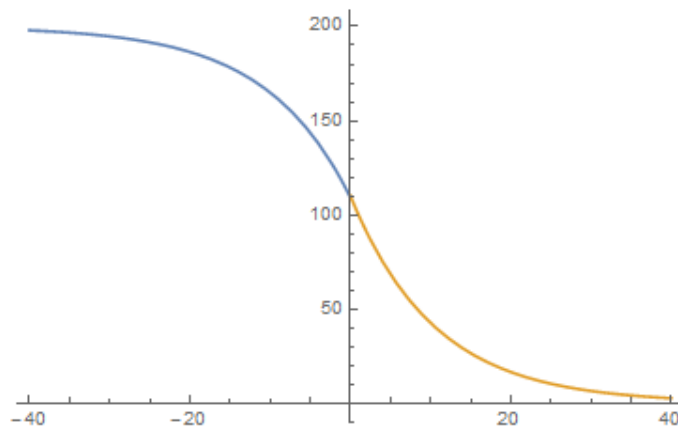
Urgentnost úkolu nezávisí na jeho prioritě. Vzorec pro urgentnost lze ale jednoduše upravit tak, aby se priorita zohlednila, a to dosazením různých hodnot v závislosti na prioritě do proměnné  $c$ .

Graf hodnoty urgentnosti vzhledem k rezervě je zobrazen na obrázku 4.2.

## 4.4 Autentifikace

Ukládání hesel uživatelů v databázi v čistém textu představuje bezpečnostní riziko. V případě, že dojde k úniku dat databáze, může útočník získat přihlašovací jména i hesla a tím i možnost přihlásit se do aplikace jako kterýkoliv uživatel. Z tohoto důvodu je vhodné neukládat samotná hesla, ale jen jejich „otisk“. Ten se z hesla vytvoří pomocí *hash funkce* [45]. Při použití vhodné funkce se jedná o relativně bezpečný způsob, kterým lze ověřovat uživatele, aniž by mohlo dojít k úniku jejich hesel.

Nevýhodou této metody autentifikace je fakt, že pro uživatele se stejným heslem funkce vygeneruje vždy stejný otisk. To znamená, že při prolomení hesla jednoho uživatele mohou být ohroženi i další uživatelé, kteří mají stejné heslo. Existují také tabulky, které obsahují předpočítané výstupy hash funkcí



Obrázek 4.2: Funkce urgentnosti pro úkol s termínem

a vstupní řetězce, které těmto výstupům odpovídají, tzv. *rainbow tables* [46]. Ty mohou být použity pro zrychlení prolamování slabších hesel.

Z tohoto důvodu v aplikaci kromě hashování používám metodu *solení*. Ta spočívá v tom, že se otisk negeneruje jen z hesla, ale navíc se k němu přidá náhodně vygenerovaný řetězec, tzv. *sůl*. Stejná hesla tak budou mít rozdílné otisky a znemožní se využití *rainbow tables* a slovníkových útoků. V databázi bude pro autentifikaci uložena dvojice  $(\text{hash}(\text{heslo}, \text{sůl}), \text{sůl})$  [45].

## 4.5 Architektura

Architektura aplikace popisuje způsob, jakým je členěna na logické celky a jak ty mezi sebou komunikují. Úroveň složitosti architektury souvisí s nefunkčními požadavky na aplikaci. Větší nároky, například na škálovatelnost, výkon, bezpečnost a další aspekty, znamenají nutnost navrhnout dostatečně robustní architekturu, která jim bude vyhovovat.

V této sekci popíšu, jak bude vypadat architektura aplikace Deadlines.

### 4.5.1 Architektura Deadlines

Aplikaci navrhují podle vzoru vícevrstvé architektury. Tvoří ji několik vrstev, z nichž každá je zodpovědná za jinou funkcionalitu. Způsob komunikace je definován rozhraním, které vrstvy nabízí. Je tak snazší oddělit konkrétní implementaci vrstev od jejich okolí. Jednou z výhod této architektury je možnost znovupoužitelnosti vrstev. To je užitečné například v případě, kde existuje několik různých rozhraní pro interakci s uživatelem. Všechny mohou komunikovat s odpovídající vrstvou, pro kterou nehraje roli jejich typy a počet.

Další výhodou je možnost poměrně pohodlně měnit konkrétní implementaci dané vrstvy (zejména v kombinaci s použitím *IoC*) a to bez zásahů do vrstev, které s ní komunikují.

Diagram architektury aplikace je zobrazen v příloze, na obrázku E.1.

### 4.6 Návrhový model

V této sekci se budu věnovat návrhovému modelu tříd a jeho popisu. Ten vychází z analytického modelu, rozšiřuje množství tříd a doplňuje implementační detaily. Mezi ně patří atributy a jejich typy, metody, viditelnost. Definují se zde také rozhraní, typy tříd a aplikují návrhové vzory [47].

Ve zbytku této sekce budu popisovat jednotlivé části návrhového modelu. Obrázek zobrazující všechny třídy, rozdělené do balíčků (*packages*), je k dispozici v příloze, diagram F.1.

#### 4.6.1 Balíček `config`

Tento balíček obsahuje konfigurační třídy využívané napříč celou aplikací. Jsou zde definovány hodnoty chybových kódů a zpráv, *Beans* pro využití frameworkem Spring a nastavení pravidel pro autentifikaci na *endpoints* aplikace.

#### 4.6.2 Balíček `controllers`

Tento balíček reprezentuje vrstvu *Controllers* v architektuře aplikace. Třídy *Controller* jsou zodpovědné za zpracování příchozích požadavků a zasílání odpovědí. Ověří autentifikaci požadavku a poté ho v odpovídajícím formátu delegují na vrstvu *Business Service*. Odpověď této vrstvy v odpovídajícím tvaru pošlou zpět.

Třída *JsonViews* definuje hierarchii rozhraní, kterými lze filtrovat množství informací, které mají být zasílány v odpovědi.

Balíček `httpbodies` obsahuje definice *POJO* objektů, které reprezentují strukturu těl požadavků a odpovědí.

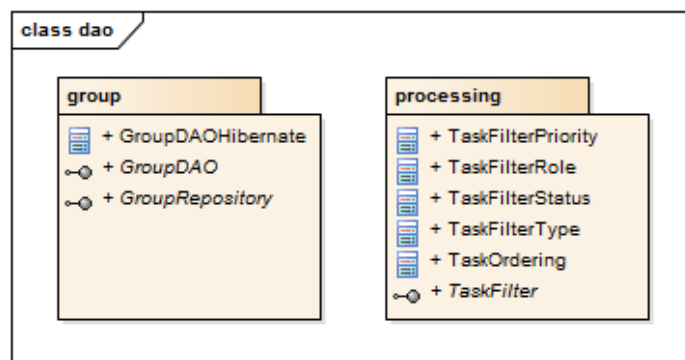
#### 4.6.3 Balíček `exceptions`

V tomto balíčku se nacházejí třídy výjimek aplikace.

#### 4.6.4 Balíček `dao`

Balíček reprezentuje vrstvu *Data Access*, obsahuje třídy zprostředkovávající persistenci dat.

Každá entita modelu zde má odpovídající podbalíček, který obsahuje rozhraní a třídy využité při jejím ukládání a načítání. Jeden z těchto podbalíčků,



Obrázek 4.3: Detail tříd balíčku dao

**group**, je zobrazen na obrázku 4.3. Rozhraní **GroupDAO** definuje metody, které jsou nabízeny ostatním komponentám aplikace pro ukládání a načítání objektů **Group** z databáze. Třída **GroupDAOHibernate** je konkrétní implementací tohoto rozhraní, které využívá *Hibernate*. Rozhraní **GroupRepository** realizuje komunikaci s frameworkem *Hibernate*.

Balíček **processing** obsahuje třídy a rozhraní, které umožňují filtrování a řazení načítaných dat.

#### 4.6.5 Balíček **services**

Detailní pohled na balíček je na obrázku 4.4.

Služby v balíčku reprezentují vrstvu *Business Services*. Balíček obsahuje rozhraní definující operace, které je možné v aplikaci spouštět, a v podbalíčku *implementation* třídy, které je implementují.

Podbalíček **security** obsahuje interní služby pro autentifikování přichozích požadavků a pro kontrolu, zda mají volající uživatelé dostatečná práva k provádění požadovaných akcí.

Podbalíček **helpers** obsahuje interní služby pro práci s entitami, které jsou využívány službami, ale nejsou součástí API této vrstvy.

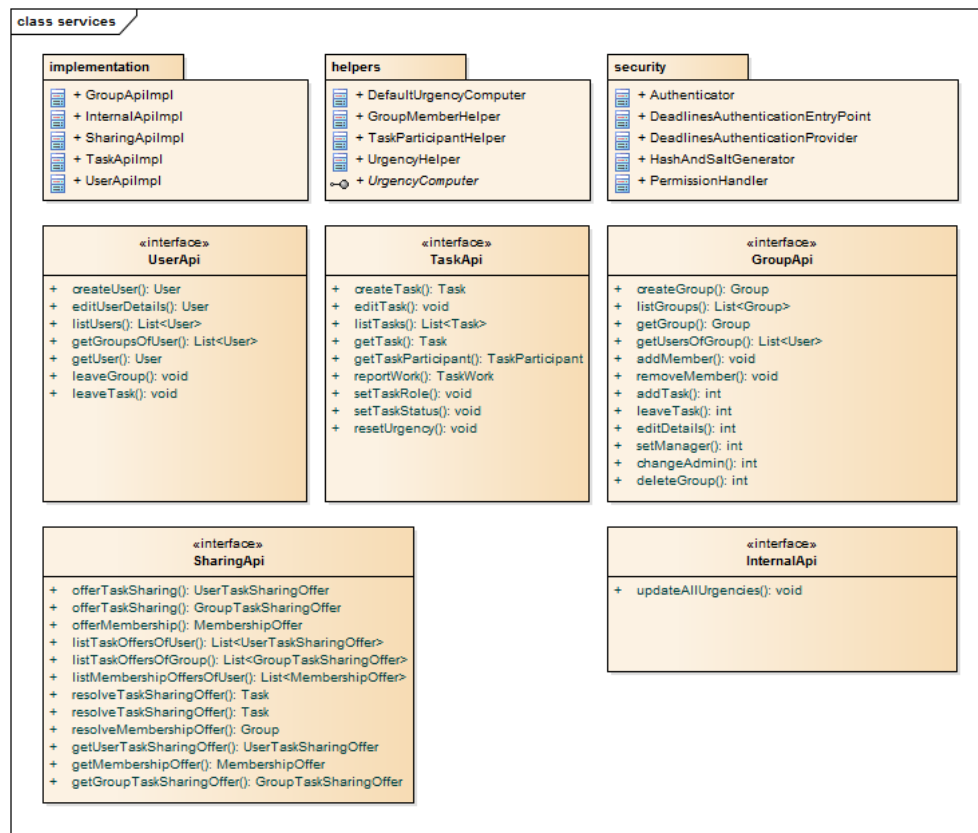
#### 4.6.6 Balíček **jobs**

Balíček reprezentuje v architektuře vrstvu *Background Jobs*. Třídy v něm spouští periodické akce, které se mají v systému vykonávat. Příkladem takové akce je pravidelné přepočítání urgentnosti všech aktivních úkolů.

#### 4.6.7 Balíček **utils**

Zde jsou zařazeny pomocné třídy, které nabízí různé statické metody, využívané zejména při testování.

## 4. NÁVRH ŘEŠENÍ



Obrázek 4.4: Detail tříd balíčku services

### 4.6.8 Balíček model

Tento balíček odpovídá stejnojmenné komponentě z diagramu architektury. Obsahuje třídy, které reprezentují business entity aplikace, analyzované v sekci 3.3. Třídy v sobě uchovávají informace prostřednictvím atributů a relací k dalším třídám tohoto balíčku. Mají jen základní metody pro čtení a ukládání atributů. Veškerá manipulace se vztahy mezi třídami a udržování konzistentních referencí v případě obousměrných vztahů je zodpovědností *Business Services* vrstvy.

Diagram popisující detaily tříd a vztahy mezi nimi je zobrazen v příloze, na obrázku F.2.

### 4.6.9 Třída DeadlinesApplication

Metoda `main()` této třídy je vstupním bodem celé aplikace.

## 4.7 Databázový model

Databázový model popisuje způsob uložení dat do databáze. Udává, jaké tabulky a sloupce v databázi existují, jaké mají klíče a integritní omezení. Dekomponují se v něm vztahy kardinality  $m:n$  vytvořením pomocné tabulky.

Databázový model vznikl na základě doménového a návrhového modelu a je zobrazen v příloze na obrázcích G.1 a G.2. Diagram byl z důvodu přehlednosti rozdělen na dvě části. První diagram zobrazuje tabulky databáze kromě nabídek, druhý zobrazuje tabulky nabídek a jejich vztah k tabulkám v prvním diagramu.

V návrhovém modelu je několik tříd, které tvoří dědičnou hierarchii a tu je potřeba převést do reprezentace tabulkami. Na to existuje několik postupů:

- Jedna tabulka pro rodiče, která obsahuje sjednocení atributů všech jejích potomků a sloupec identifikující typ záznamu.
- Tabulka pro každou konkrétní třídu.
- Tabulka pro rodiče a s ní asociovaná tabulka pro každého z potomků, obsahující atributy v něm definované.

Pro uložení dědičné hierarchie třídy `Task` jsem zvolil první způsob. Rozdíl mezi podtypy této třídy je jen v jediném atributu, takže při ukládání záznamů bude množství nevyužitých sloupců minimální.

Pro uložení třídy `Offer` jsem zvolil druhý způsob. Všechny podtypy mají společný atribut identifikující uživatele, který jej vytvořil, ale v ostatních attributech se liší. Proto má každý podtyp svou vlastní tabulku.

Inicializační SQL skript pro vytvoření databáze je přiložen na elektronickém médiu.

## 4.8 Testování

Testy aplikace jsou nezbytnou součástí jejího vývoje a dlouhodobé udržitelnosti. Určují, jaké chování se od testované komponenty očekává a v případě chybného zásahu do její implementace programátora na problém upozorní.

### 4.8.1 Nástroje

#### 4.8.1.1 JUnit

Framework Spring nabízí integraci s testovacím nástrojem JUnit [48]. Ten umožňuje psát automatizované jednotkové i integrační testy, v nichž programátor může kontrolovat a porovnávat obsah proměnných, vyhazované výjimky a další.

JUnit lze použít samostatně bez spuštění Spring kontejneru, kdy se programátor musí postarat o inicializaci všech testovaných komponent, nebo v

kombinaci s ním. V tom případě je dostupná veškerá funkcionální jako za běhu aplikace, zejména *dependency injection*.

### 4.8.1.2 Mockito

Mockito [49] je nástroj pro vytváření *mock* objektů. To jsou jednoduché „kopie“ objektů s pevně daným chováním. Jejich účelem je izolování testované komponenty tak, aby veškeré testované chování a případné chyby pocházely od ní a ne jejich závislostí.

### 4.8.2 Jednotkové testy

Jednotkové testy jsou určeny k testování nejmenších možných částí aplikace. Těmi jsou zpravidla jednotlivé třídy a jejich metody. Testy pak kontrolují, jestli se třída chová tak, jak by měla. Třídy, na nichž je testovaná část závislá, se zpravidla nahrazují *mock* objekty.

Jednotkové testy nevyžadují spuštění celého prostředí aplikace, jako například databáze a dalších komponent a jejich provedení je relativně rychlé.

### 4.8.3 Integrační testy

Integrační testy se zabývají testováním větších částí aplikace. Předpokládá se, že jednotlivé komponenty jsou otestované, fungují správně a ověřuje se správnost jejich spolupráce.

Tímto způsobem budu v aplikaci testovat zejména funkčnost vrstev, které jsou zodpovědné za hlavní logiku aplikace, tedy *Business Services* a *Controllers*. Testy budou odpovídat jednotlivým scénářům užití z kapitoly 3.2.2 a nabízených *endpoints* z kapitoly 4.2.3.2.

Při každém scénáři testování nejprve vytvořím počáteční, známý stav entit v aplikaci. Poté simuluji scénáře užití částí aplikace uživatelem, například zakládání skupin a úkolů, zasílání nabídek jiným uživatelům, přijímání nabídek a podobně. V průběhu scénáře ověřuji, že stav entit v aplikaci je takový, jaký očekávám.

Při integračním testování vrstvy *Controllers* u každého koncového bodu simuluji zasílání validních i nevalidních požadavků a kromě kontroly změn v entitách testuji, zda návratový kód a tělo odpovědi odpovídá tomu, co očekávám.



## Implementace

V této kapitole se věnuji bližšímu popisu průběhu implementace aplikace. V jednotlivých sekcích kapitoly komentuji některé aspekty implementace a uvádím příklady kódu. Kompletní zdrojový kód je přiložen na elektronickém médiu.

### 5.1 Vývojové prostředí

Aplikaci jsem vyvíjel v prostředí IntelliJ IDEA od firmy JetBrains. Poskytuje dobrou podporu práce s frameworkem Spring, například nabízí funkce pro vkládání, zobrazování a navigování definovanými Spring Beans. Součástí je také nástroj pro ruční vytváření a posílání HTTP požadavků, který je užitečný pro základní testování REST rozhraní aplikace. Studentům ČVUT FIT je poskytována licence pro edici Ultimate po dobu studia zdarma.

V projektu využívám nástroj Maven pro správu závislostí a verzí. Soubor `pom.xml` aplikace je potomkem `spring-boot-starter-parent`, který Spring Boot označuje jako *starter POM*. Ten obsahuje vybrané komponenty typicky využívané u Spring projektů. Vývojář podle svých potřeb může dodefinovat další závislosti na požadovaných modulech.

Aplikace je postavena na Spring Boot verze 1.3.3.

### 5.2 Služby

Služby (*services*) v aplikaci jsou implementovány jako stejnojmenné komponenty frameworku Spring. Každou třídu anotovanou pomocí `@Service` Spring automaticky při startu aplikace detekuje a zaregistruje jako *Bean*. V ostatních komponentách je pak tuto službu možné využívat následující deklarací:

```
@Autowired
private ServiceClass service;
```

Framework se postará o vložení instance odpovídající třídy, pokud je tato třída zaregistrována jako *Bean*. Deklarovaný typ může být i rozhraní, Spring pak poskytne třídu, která toto rozhraní implementuje a je zaregistrována s odpovídajícím jménem.

Třídy z vrstvy služeb jsou anotovány anotací `@Transactional`. Tím je zaručeno, že každá jejich metoda bude součástí jediné databázové transakce. Pokud metoda neskončí úspěšně a dojde v ní k výjimce, všechny dosud provedené změny budou automaticky vráceny a databáze zůstane nezměněna.

### 5.3 Aktualizace urgentnosti

Aktualizace urgentnosti úkolů se provádí v případě vytvoření nebo úpravy úkolu a také periodicky. Využívá se k tomu třídy `UrgencyHelper`, která deleguje výpočet hodnoty na třídu implementující rozhraní `UrgencyComputer`, poté uloží získanou urgentnost a upraví čas její poslední aktualizace.

Pro periodickou aktualizaci urgentnosti využívám *task scheduling* frameworku Spring. Tuto funkcionalitu lze aktivovat anotací `@EnableScheduling` na kterékoliv konfigurační třídě. Spring poté automaticky vykonává požadované metody v zadaném intervalu:

```
@Scheduled(fixedDelayString = "${update.urgency.interval}")
public void updateUrgencies() {
    logger.info("Updating urgencies of all tasks.");
    internalApi.updateAllUrgencies();
}
```

Interval aktualizace je načten z externího konfiguračního souboru z klíče `update.urgency.interval`.

### 5.4 Model

Třídy reprezentující model aplikace jsou anotovány pomocí `@Entity`. Hibernate takto anotované třídy zpracuje a bude je možné serializovat a deserializovat jako objekty do a z databáze. Anotace na attributech entitních tříd určují název odpovídajícího sloupce v tabulce, integritní omezení, způsob mapování asociací, generování hodnot primárních klíčů *ID* a další možnosti.

Hibernate umožňuje mapování *m:n* asociací pomocí anotace `@JoinTable`, která definuje pomocnou tabulku, ve které se bude uchovávat informace o asociaci. Není tak nutné definovat pomocnou entitu, která by vztah dekomponovala.

Pomocí anotací na attributech také definuji, jestli mají být tyto atributy zahrnuty při serializaci třídy do JSON. Spring pro manipulaci s formátem JSON využívá knihovnu Jackson. Ta umožňuje následující:

```
@Column(name = COL_TASK_DESCRIPTION)
@JsonView(JsonViews.Task.Detail.class)
protected String description;
```

Atribut *description* tak bude v JSON odpovědi zahrnutý, pouze pokud je metoda Spring Controlleru, zpracovávajícího požadavek, anotována rozhraním, které implementuje `JsonViews.Task.Detail.class`.

## 5.5 Persistence

Pro každou třídu modelu je definováno rozhraní DAO s metodami, které musí poskytovatel persistence implementovat. V aplikaci používám framework Hibernate, takže implementací těchto rozhraní jsou třídy `DAOHibernate`. Ty delegují volání metod na rozhraní implementující `CrudRepository<T, ID>`, kde `T` je typ třídy, kterou chci serializovat, a `ID` je datový typ jejího primárního klíče.

V rozhraní, které implementuje `CrudRepository`, lze definovat metody podle dané jmenné konvence. Podle nich se při spuštění aplikace automaticky vygenerují odpovídající SQL dotazy. Toto je část definice `GroupRepository`, využívané třídou `GroupDAOHibernate`:

```
@Repository
public interface GroupRepository
    extends CrudRepository<Group, Long> {
    Group findByName(String name);
    List<Group> findByMembers_User(User user);
    Group saveAndFlush(Group group);
}
```

Metody definované v rozhraní není nutné vůbec implementovat, jsou automaticky generovány frameworkem. Názvy metod tvoří jednoduché i složené databázové dotazy, informace o syntaxi a klíčových slovech je k dispozici v dokumentaci modulu *Spring Data* [50].

Například volání metody `findByName("jmeno")` se přeloží na následující databázový dotaz, jehož odpověď je deserializována do instance třídy `Group`:

```
select group0_.group_id as group_id1_1_,
       group0_.description as descript2_1_,
       group0_.name as name3_1_
from   group_table group0_
where  group0_.name="jmeno"
```

Dotazy mohou být i složitější, jako je tomu u metody `findByMembers_User`. Ta vrátí seznam všech skupin, jichž je uživatel členem. V jejím „přeloženém“ SQL tvaru jsou automaticky vygenerována potřebná spojení tabulek (*join*).

## 5.6 Autentifikace

Zabezpečení a autentifikaci uživatelů řeší v aplikaci modul *Spring Security* [51]. V konfigurační třídě `config.Security`, která je potomkem třídy `WebSecurityConfigurerAdapter`, provádím nastavení zabezpečení koncových bodů:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth.authenticationProvider(authenticationProvider);
}

@Override
public void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers(HttpMethod.GET, "/user").permitAll()
        .antMatchers(HttpMethod.POST, "/user").permitAll()
        .anyRequest().authenticated()
        .and().httpBasic()
        .authenticationEntryPoint(
            deadlinesAuthenticationEntryPoint);
}
```

S touto konfigurací budou povoleny anonymní dotazy na adresu `/user` s metodami GET a POST, u všech ostatních jsou požadovány přihlašovací údaje. Logika autentifikace uživatele ze zadaných údajů probíhá v mnou definovaném `authenticationProvider`. Pokud autentifikace uspěje, pak je ID volajícího uživatele předáno odpovídající třídě `Controller`.

Pokud nejsou poskytnuty údaje nebo autentifikace selže, volá se metoda v objektu `deadlinesAuthenticationEntryPoint`, která uživateli odpoví chybovým HTTP kódem.

## 5.7 Koncové body

Komunikaci REST API zajišťují třídy anotované pomocí `@RestController`. Ty obsahují metody obsluhující jednotlivé koncové body, které aplikace nabízí. Deklarace takové metody vypadá například takto:

```
@RequestMapping(value = "/user/{id}",
    method = RequestMethod.PUT,
    produces = MediaType.APPLICATION_JSON_VALUE,
    consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity editUser(
```

```
@AuthenticationPrincipal Long userId,  
@PathVariable("id") Long id,  
@RequestBody UserCreateRequestBody request) {}
```

Anotace `@RequestMapping` určuje, jaká adresa a HTTP metoda bude touto metodou zpracována a jaký MIME typ [52] je akceptován a produkován. V parametrech je metodě předáno ID autentifikovaného uživatele, parametry z URL adresy a tělo požadavku deserializované do *POJO* objektu.

Metoda objektu `Controller` volá odpovídající služby v závislosti na předaných parametrech a jejich výsledek transformuje do odpovědi, kterou zasílá zpět.



# Testování

V této kapitole popíšu, jakým způsobem jsem řešil testování aplikace při jejím vývoji tak, jak bylo popsáno v sekci 4.8.

Pro manuální spuštění testů na příkazové řádce je potřeba mít nainstalovaný nástroj Maven a JDK verze alespoň 1.8. Spuštění testů se provede příkazem `mvn clean test` v adresáři `project`.

## 6.1 Mockito

Pro jednotkové testy jsem využil nástroj *Mockito*, který umožňuje jednoduše vytvořit „napodobeninu“ objektu, definovat u ní chování a poté kontrolovat, jaké její metody s jakými parametry byly zavolány.

Příkladem „mockování“ v mnou implementovaných testech je například mock rozhraní `UserAPI`:

```
@Mock
UserApi userApi;

Mockito.when(
    userApi.getUser(any(Long.class))
).thenReturn(user);
```

Zde definuji chování objektu `userApi` takto: pokud je na objektu zavolána metoda `getUser` s jakýmkoliv parametrem typu `Long`, jejím výsledkem bude objekt `user`.

## 6.2 Jednotkové testy

Jednotkovými testy jsem testoval třídy se žádnými, nebo několika málo závislostmi na ostatních třídách. Testovat by se tímto způsobem daly i komplexnější

## 6. TESTOVÁNÍ

---

třídy, ale bylo by nutné místo jejich závislostí vytvářet mock objekty. Z důvodu upřednostnění úsilí na jiných částech práce jsem od tohoto upustil a u složitějších tříd využíval jen integrační testy, ve kterých měly třídy přístup ke všem svým závislostem skrze framework Spring.

Příkladem jednoho z jednotkových testů je následující kód, testující správnost funkce třídy, která autentifikuje uživatele pomocí objektu `User` a hesla:

```
@RunWith(MockitoJUnitRunner.class)
public class AuthenticatorTest {
    @Mock
    User mockedUser;

    @Spy
    private ShaPasswordEncoder passwordEncoder =
        new ShaPasswordEncoder();

    @InjectMocks
    private Authenticator authenticator = new Authenticator();

    @Before
    public void setUp() throws Exception {
        Mockito.when(mockedUser.getUsername()).thenReturn("User2");

        Mockito.when(mockedUser.getPasswordHash())
            .thenReturn("375c3858a8d203ac6393d948098f797d7d871e32");

        Mockito.when(mockedUser.getPasswordSalt())
            .thenReturn("c7badfd9725a9f2e");
    }

    @Test
    public void authenticate() throws Exception {
        String correctPassword = "password";
        String wrongPassword = "abcd";

        assertNull(authenticator.authenticate(mockedUser,
                                              wrongPassword));

        assertEquals(authenticator.authenticate(mockedUser,
                                              correctPassword),
                      mockedUser);
    }
}
```



V tomto testu třídy `Authenticator` nejprve za pomoci anotace `@Mock` vytvořím mock instanci třídy `User`. `@Spy` označuje „plnohodnotný“ Java objekt, který má být vložen do testované třídy jako závislost. Anotace `@InjectMocks` poté informuje framework Mockito o tom, že má do takto anotované třídy vložit její závislosti, v tomto případě objekty `passwordEncoder` a `authenticator`.

V metodě `setUp` definuji chování *mock* objektu `mockedUser`, kterého využívám v následující metodě. V ní kontroluji, že v případě chybné kombinace uživatele a hesla vrací třída `null`, v případě správné kombinace pak instanci třídy `User`.

## 6.3 Integrační testy

Integrační testy jsem během vývoje využil pro testování správné funkčnosti tříd s větším počtem závislostí a tříd, které ze své podstaty vyžadují komunikaci s vnějšími systémy. To jsou například třídy *Data Access Object*, které jsou zodpovědné za ukládání a načítání dat z databáze.

### 6.3.1 Vrstva Business Services

Důkladně jsem integračními testy otestoval vrstvu *Business Services*, která obsahuje hlavní logiku aplikace. Všechny případy užití z kapitoly 3.2.2 jsou v těchto testech obsaženy samostatně, i jako součást komplexnějších scénářů, kde jsou testovány v kombinaci s ostatními případy užití. Příkladem takového scénáře jsou následující kroky:

1. Vytvoření tří uživatelů a skupiny, v níž je první z nich Admin.
2. Vytvoření několika úkolů pro každého z uživatelů.
3. Sdílení úkolů Admina s jeho skupinou.
4. Kontrola, zda jsou uživatelé účastníky očekávaných úkolů.
5. Nabídka členství ve skupině od Admina jednomu z uživatelů a její přijetí uživatelem.
6. Kontrola, zda má skupina očekávané členy a zda je nový člen skupiny účastníkem svých soukromých i skupinových úkolů. Dále kontroluji, zda ostatním uživatelům nepříbyly ani neubyly úkoly.
7. Odchod člena ze skupiny.
8. Opět kontrola členů skupiny a účastníků úkolů.

### 6.3.2 Vrstva Controllers

Vrstva *Controllers* je zodpovědná za zpracování příchozích požadavků, autentifikaci uživatele a volání odpovídajících metod vrstvy *Business Services*. Ta je již otestovaná způsobem popsaným v sekci 6.3.1. Testy vrstvy *Controllers* jsou zaměřené na kontrolu zpracování příchozích požadavků a správných údajů v HTTP odpovědích aplikace, tedy jejich stavového kódu, hlavičky a těla. Jeden z testů má například následující strukturu:

1. Zaslání autentifikovaného požadavku na změnu údajů a hesla uživatele metodou PUT na adresu `/user/id`, kde `id` je ID uživatele.
2. Kontrola, zda je vrácen kód 200 OK a hlavička *Content-Type* je očekávaného typu.
3. Kontrola, zda je tělo odpovědi v očekávaném formátu a obsahuje upravená data.
4. Zaslání požadavku s původními autentifikačními údaji a kontrola, že byl vrácen kód 401 Unauthorized.
5. Zaslání požadavku s novými autentifikačními údaji a kontrola, že byl vrácen kód 200 OK.

K simulování zasílání požadavků na REST *endpointy* a kontrolu odpovědi využívám třídu *MockMvcBuilders*, která je součástí testovacího modulu frameworku Spring. Použití její instance *mvc* je znázorněno v kódu pod tímto odstavcem. Uvedená metoda představuje test, který se skládá z výše uvedených bodů.

```
@Transactional
@Test
public void userEditTest() throws Exception {

    String newPwd = "new-password";
    String newEmail = "another-email@hello.com";
    String newName = "Brand new name";

    MvcResult result = mvc.perform(put("/user/" + user1.getId())
        .header("Authorization", BasicAuthHeaderBuilder
            .buildAuthHeader(user1.getUsername(), "pwd"))
        .content("{\"password\":\"" + newPwd
            + "\",\"email\":\"" + newEmail
            + "\",\"name\":\"" + newName + "\"}")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
```

```
.andExpect(content().contentType(MediaType.APPLICATION_JSON))
.andReturn();

String response = result.getResponse().getContentAsString();

JsonParser parser = new JsonParser();
JsonObject object = parser.parse(response).getAsJsonObject();

Assert.assertEquals(newEmail,
    object.get("email").getString());
Assert.assertEquals(newName,
    object.get("name").getString());

mvc.perform(get("/user/" + user1.getId())
    .header("Authorization", BasicAuthHeaderBuilder
        .buildAuthHeader(user1.getUsername(), "pwd"))
)
.andExpect(status().isUnauthorized());

mvc.perform(get("/user/" + user1.getId())
    .header("Authorization", BasicAuthHeaderBuilder
        .buildAuthHeader(user1.getUsername(), newPwd))
)
.andExpect(status().isOk());
}
```

## 6.4 Výsledky testování

Celkem jsem při implementaci aplikace napsal více než 150 testovacích případů. Testy jsem nepsal pro všechny třídy a všechny jejich metody, ale jen pro ty, které byly svou funkcí netriviální. Testy pokrývají všechny vytyčené funkce, které má aplikace nabízet, jak je popisují případy užití v sekci 3.2.2 a všechny koncové body ze sekce 4.2.3.2.

Případné chyby, které nebyly zachyceny automatizovanými testy, by mohly být odhaleny testováním aplikace samotnými uživateli. Takové testování jsem neprovedl, protože aplikace disponuje pouze rozhraním REST. Nenabízí žádné grafické rozhraní, které by mohlo být použité k manuálnímu testování i technicky nevzdělanými uživateli.

Výsledná aplikace, která je přiložena k této práci, uspěla ve všech automatizovaných testech a neobsahuje tak žádné známé chyby.



## Nasazení

V této kapitole se budu věnovat popisu postupu při nasazování aplikace Deadlines do provozu.

Aplikace je distribuována a může být nasazena dvěma způsoby:

- WAR archiv pro nasazení do servlet kontejneru (například Apache Tomcat).
- Spustitelný JAR, obsahující vlastní webový server.

Distribuované soubory aplikace mají název `deadlines-x.y.z`, kde `x.y.z` je číslo verze. Na tyto soubory se budu níže odkazovat bez uvádění jejich verze, tedy jen jako `deadlines`.

### 7.1 Verze Java

K běhu aplikace je vyžadováno JRE alespoň verze 1.8.

### 7.2 Uložiště dat

Deadlines může pro persistenci dat využít dva typy uložistě. Prvním, doporučeným uložistěm je databáze MySQL. Návod na její instalaci je ale mimo rozsah této práce, proto jej zde neuvádím.

Druhým typem uložistě je databáze H2. Tuto databázi není nutné nijak instalovat, aplikace ji automaticky vytvoří a celá je obsažena v jednom souboru na disku. Vhodná je pro testování a zkušební provoz aplikace. Typ a konkrétní umístění datového uložistě lze nastavit v konfiguračním souboru.

### 7.3 Nasazení WAR archivu

K nasazení WAR archivu je nutné mít již nainstalovaný některý ze servlet kontejnerů. Nasazení aplikace bylo testováno na serveru Apache Tomcat verze

## 7. NAsazení

---

9.0.0.0 a návod se tak bude vztahovat právě k tomuto serveru. Předpokládané umístění instalace serveru je ve složce `$CATALINA_HOME`. Postup je následující:

1. Zkopírujte soubor `deadlines.war` do složky `$CATALINA_HOME/webapps/`.
2. Zkopírujte soubor `application.properties` do složky `$CATALINA_HOME/lib/deadlines/` a upravte potřebné informace.
3. Pokud jste v konfiguračním souboru nezvolili možnost automatického vytvoření schématu, spusťte v databázi inicializační skript `database_init_mysql.sql` pro vytvoření potřebných tabulek.
4. Spusťte server. Aplikace bude dostupná na adrese `<server>/deadlines` na portu nastaveném v konfiguraci serveru.

### 7.4 Nasazení JAR archivu

K nasazení JAR archivu je nutná jen přítomnost JRE odpovídající verze a případně MySQL databáze.

1. Zkopírujte soubory `deadlines.jar` a `application.properties` do složky, odkud chcete aplikaci spouštět.
2. Upravte konfigurační soubor.
3. Pokud jste v konfiguračním souboru nezvolili možnost automatického vytvoření schématu, spusťte v databázi inicializační skript `database_init_mysql.sql` pro vytvoření potřebných tabulek.
4. Spusťte aplikaci příkazem `java -jar deadlines.jar`. Dostupná bude na adrese `http://localhost/` na portu nastaveném v konfiguračním souboru.

### 7.5 Veřejně dostupná instance aplikace

Pro účely testování jsem aplikaci nasadil na vlastní server. Dostupná je na adrese `http://melkamar.cz/deadlines`.

---

## Závěr

Cílem této práce byla tvorba serverové části aplikace, komunikující přes rozhraní REST, která má usnadňovat správu soukromých i sdílených úkolů jednotlivcům a malým týmům. Dílčími úkoly práce bylo provedení rešerše existujících řešení a formulování požadavků na aplikaci. Následovala detailnější analýza požadavků a problémové domény, návrh řešení včetně nabízených rozhraní, jeho implementace a otestování správné funkčnosti.

V rámci rešeršní části jsem představil několik existujících aplikací, rozdělených do kategorií podle jejich cílové skupiny. Tyto aplikace jsem popsal a porovnal z hlediska funkčnosti, nabízených rozhraní a dalších kritérií. Na závěr jsem zvážil, jaké funkce těchto aplikací jsou pro mou cílovou skupinu důležité a mají být společně s koncepty urgentnosti a rostoucích úkolů zahrnuty i v mé aplikaci.

V analytické části jsem na základě shrnutí rešerše vytvořil uživatelské příběhy, které definují funkční požadavky kladené na navrhovanou aplikaci. Pomocí nich jsem následně vytvořil model případů užití. Tento model detailněji popsal očekávané interakce mezi aplikací a uživateli. Dále jsem vypracoval doménový model, zachycující entity aplikace a vztahy mezi nimi.

Na začátku návrhové části práce jsem se věnoval výběru technologií pro implementaci a specifikaci REST rozhraní nabízeného aplikací. Dále jsem popsal význam urgentnosti úkolu a postup při jejím výpočtu pro oba typy úkolů. Určil jsem způsob ukládání hesel, autentifikaci uživatelů a architekturu aplikace. Z architektury jsem vyšel při tvorbě návrhového modelu, ve kterém jsem popsal strukturu aplikace na úrovni balíčků a tříd, vysvětlil jsem jejich zodpovědnosti a vztah k architektuře aplikace. V závěru jsem nastínil, jakým způsobem budu aplikaci při vývoji testovat.

Vývoji aplikace jsem se věnoval v kapitole implementace. Zde jsem uvedl informace o použitém vývojovém prostředí a nástroji pro automatizované sestavování (build) aplikace. Dále jsem v této části popsal způsob řešení některých implementačních detailů spolu s úryvky zdrojových kódů a jejich vysvětlením.

Během implementace jsem paralelně psal testy, kterým jsem se věnoval v kapitole testování. Zde jsem popsal, jakým způsobem jsem testoval jednotlivé třídy i funkčnost aplikace jako celku. Uvedl jsem a vysvětlil příklady kódu testovacích metod spolu se scénáři, na nichž jsou testy založeny. Na závěr jsem zhodnotil výsledky testů a poukázal na jejich možné nedostatky.

V poslední části práce jsem uvedl, jaký software je nutný k používání aplikace a vysvětlil, jakým způsobem ji lze nasadit do provozu.

Cíl práce jsem ve stanoveném rozsahu splnil, aplikaci lze využívat podle případů užití definovaných během analýzy. Součástí práce je i kompletní dokumentace způsobu komunikace s aplikací. Vytyčená funkcionalita byla implementována, otestována a aplikaci je možné nasadit a provozovat.

Nedostatkem vytvořeného řešení je absence rolí uživatelů v rámci aplikace. Nelze tak určit administrátory s vyššími právy, kteří by se měli o její chod starat. To je, spolu s klientskými aplikacemi, nabízejícími grafické uživatelské rozhraní, možným námětem pro budoucí vývoj.

Jelikož se jedná jen o serverovou část aplikace bez grafického rozhraní, je možnost jejího využití uživateli v současné podobě omezena. Představuje ale základní stavební prvek s dobře definovaným rozhraním komunikace, který lze rozšiřovat a integrovat s klientskými aplikacemi.



---

## Literatura

- [1] Nielsen, J.: Usability 101: Introduction to Usability [online]. 2012-01-04, [cit. 2016-03-16]. Dostupné z: <https://www.nngroup.com/articles/usability-101-introduction-to-usability/>
- [2] Atlassian: JIRA Software [online]. © 2016, [cit. 2016-03-04]. Dostupné z: <https://www.atlassian.com/software/jira/>
- [3] Atlassian: Open Source Project License Request [online]. © 2016, [cit. 2016-03-04]. Dostupné z: <https://www.atlassian.com/software/views/open-source-license-request>
- [4] Atlassian: Community License Request [online]. © 2016, [cit. 2016-03-04]. Dostupné z: <https://www.atlassian.com/software/views/community-license-request>
- [5] Mozilla Foundation: Bugzilla [online]. 2015-12-22, [cit. 2016-03-04]. Dostupné z: <https://www.bugzilla.org/>
- [6] Lang, J.-P.: Redmine [online]. 2015-11-15, [cit. 2016-03-04]. Dostupné z: <http://www.redmine.org/>
- [7] Trello Inc.: Trello [online]. © 2016, [cit. 2016-03-06]. Dostupné z: <https://trello.com/>
- [8] What is Kanban? [online]. © 2009–2015, [cit. 2016-03-06]. Dostupné z: <http://kanbanblog.com/explained/>
- [9] Trello Pricing [online]. © 2016, [cit. 2016-03-06]. Dostupné z: <https://trello.com/pricing>
- [10] API Reference | Trello Developers [online]. © 2016, [cit. 2016-03-06]. Dostupné z: <https://developers.trello.com/advanced-reference>

- [11] Barzooka: Trackie [online]. [cit. 2015-11-21]. Dostupné z: <https://trackieapp.com/>
- [12] Fog Creek: FogBugz [online]. [cit. 2016-03-06]. Dostupné z: [www.fogcreek.com/fogbugz/](http://www.fogcreek.com/fogbugz/)
- [13] Agile Planning and Project Management [online]. ©1998–2016, [cit. 2016-03-06]. Dostupné z: <https://www.mountaingoatsoftware.com/presentations/agile-planning-and-project-management>
- [14] Doist: todoist [online]. Version 723, [cit. 2016-03-07]. Dostupné z: <https://en.todoist.com/>
- [15] Compare Todoist Free and Todoist Premium [online]. [cit. 2016-03-07]. Dostupné z: <https://todoist.com/compareVersions>
- [16] API Documentation | Todoist Developer [online]. [cit. 2016-03-07]. Dostupné z: <https://developer.todoist.com/>
- [17] Toodledo: Toodledo [online]. © 2004–2016, [cit. 2016-03-07]. Dostupné z: <https://www.toodledo.com/>
- [18] Toodledo: Toodledo [online]. © 2004–2016, [cit. 2016-03-07]. Dostupné z: <http://api.toodledo.com/3/index.php>
- [19] Google Inc.: Google Inbox [online]. [cit. 2016-03-07]. Dostupné z: <https://www.google.com/inbox/>
- [20] Google Inc.: Google Calendar [online]. [cit. 2016-03-07]. Dostupné z: <https://www.google.com/calendar/>
- [21] How to read reminders in google calendars - Stack Overflow [online]. 2016-02-21. Dostupné z: <http://stackoverflow.com/questions/35534774/how-to-read-reminders-in-google-calendars>
- [22] User Stories: An Agile Introduction [online]. © 2003–2014. Dostupné z: <http://www.agilemodeling.com/artifacts/userStory.htm>
- [23] Cohn, M.: *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, první vydání, ISBN 978-0321205681.
- [24] Stellman, A.: Requirements 101: User Stories vs Use Cases [online]. 2009-05-03. Dostupné z: <http://www.stellman-greene.com/2009/05/03/requirements-101-user-stories-vs-use-cases/>
- [25] Mlejnek, J.: Analýza a sběr požadavků [online]. © 2011. Dostupné z: <https://edux.fit.cvut.cz/oppa/BI-SI1/prednasky/BI-SI1-P03m.pdf>

- 
- [26] Jim Arlow, I. N.: *UML 2 a unifikovaný proces vývoje aplikací*. Computer Press (CPress), první vydání, ISBN 978-80-251-1503-9.
- [27] Čápka, D.: 4. díl - UML - Doménový model [online]. © 2011. Dostupné z: <http://www.itnetwork.cz/navrhove-vzory/uml/uml-domenovy-model-diagram/>
- [28] Constraints - UML Domain model - how to model multiple roles of association between two entities? [online]. © 2016. Dostupné z: <http://stackoverflow.com/questions/36100826/uml-domain-model-how-to-model-multiple-roles-of-association-between-two-entiti>
- [29] java.com: Java + You [online]. 2016, [cit. 2016-03-20]. Dostupné z: <https://www.java.com/en/>
- [30] What Is Java? [online]. 2007, [cit. 2016-03-20]. Dostupné z: <http://searchsoa.techtarget.com/definition/Java>
- [31] Balík, M.: Programování v jazyku Java [online]. 2011, [cit. 2016-03-20]. Dostupné z: <https://edux.fit.cvut.cz/oppa/BI-PJV/prednasky/pjv01uvod.pdf>
- [32] Spring Framework [online]. © 2016, [cit. 2016-03-20]. Dostupné z: <https://projects.spring.io/spring-framework/>
- [33] Hanák, D.: Dependency injection (předávání závislostí) [online]. 2016, [cit. 2016-03-20]. Dostupné z: <http://www.itnetwork.cz/navrhove-vzory/dependency-injection-navrhovy-vzor/>
- [34] Spring Boot [online]. © 2016, [cit. 2016-03-20]. Dostupné z: <http://projects.spring.io/spring-boot/>
- [35] design - What is opinionated software? - Stack Overflow [online]. 2009-04-29, [cit. 2016-03-20]. Dostupné z: <http://stackoverflow.com/questions/802050/what-is-opinionated-software>
- [36] Spring Boot – Simplifying Spring for Everyone [online]. 2013-08-06, [cit. 2016-03-20]. Dostupné z: <https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone>
- [37] Hibernate ORM - Hibernate ORM [online]. [cit. 2016-03-20]. Dostupné z: <http://hibernate.org/orm/>
- [38] DB-Engines Ranking - die Rangliste der populärsten Datenbankmanagementsysteme [online]. 2016-04, [cit. 2016-03-20]. Dostupné z: <http://db-engines.com/de/ranking>
- [39] Apache Tomcat - Welcome! [online]. © 1999–2016, [cit. 2016-03-20]. Dostupné z: <http://tomcat.apache.org/>

- [40] Daniel Jacobson, D. W., Greg Brail: *APIs - A Strategy Guide*. O'Reilly Media, Inc., první vydání, ISBN 978-1-449-30892-6.
- [41] Architectural Styles and the Design of Networbased Software Architectures [online]. 2000, [cit. 2016-03-21]. Dostupné z: [http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf)
- [42] Intro to REST (aka. What Is REST Anyway?) [online]. 2012-05-30, [cit. 2016-03-21]. Dostupné z: <https://www.youtube.com/watch?v=1lpr5924N7E>
- [43] RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication [online]. © 1999, [cit. 2016-03-21]. Dostupné z: <https://tools.ietf.org/html/rfc2617#section-2>
- [44] HTTP/1.1: Status Code Definitions [online]. [cit. 2016-05-07]. Dostupné z: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
- [45] Bezpečnost 5. Hašovací funkce, MD5, SHA-x, HMAC [online]. 2011, [cit. 2016-04-16]. Dostupné z: <https://edux.fit.cvut.cz/oppa/BI-BEZ/prednasky/bez5.pdf>
- [46] What is rainbow table? - Definition from WhatIs.com [online]. 2015-01, [cit. 2016-04-16]. Dostupné z: <http://whatis.techtarget.com/definition/rainbow-table>
- [47] Návrhový model tříd I. [online]. 2008-04-23, [cit. 2016-04-15]. Dostupné z: [http://www.fi.muni.cz/~buhnova/PV167/10\\_NavrhovyModelTrid\\_I.pdf](http://www.fi.muni.cz/~buhnova/PV167/10_NavrhovyModelTrid_I.pdf)
- [48] JUnit [online]. © 2002–2016, [cit. 2016-04-17]. Dostupné z: <http://junit.org/junit4/>
- [49] Mockito [online]. [cit. 2016-04-17]. Dostupné z: <http://mockito.org/>
- [50] Spring Data JPA - Reference Documentation [online]. © 2008–2012, [cit. 2016-04-17]. Dostupné z: <http://docs.spring.io/spring-data/jpa/docs/1.2.0.RELEASE/reference/html/#repository-query-keywords>
- [51] Spring Security [online]. © 2016, [cit. 2016-04-17]. Dostupné z: <http://projects.spring.io/spring-security/>
- [52] Media Types [online]. 2016-04-15, [cit. 2016-04-17]. Dostupné z: <http://www.iana.org/assignments/media-types/media-types.xhtml>

## Seznam použitých zkratk

**HW** Hardware

**SW** Software

**MVC** Model-View-Controller

**IOC** Inversion of Control

**POJO** Plain Old Java Object

**JRE** Java Runtime Environment

**JDK** Java Development Kit

**URL** Uniform Resource Locator



## Splnění kritérií řešerše

Tabulka B.1: Tabulka splnění kritérií řešerše

	Kr. 1	Kr. 2	Kr. 3	Kr. 4	Kr. 5	Kr. 6
JIRA	+				+	
Bugzilla	+			*	+	+
Redmine	+			*	+	+
Trello	+	+	+	+	+	
Trackie	+	+	+			
FogBugz	+	+			+	
Todoist	+	+	+	+	+	
Toodledo	+		+	**	+	
Inbox	+	+		+		
Poznámky	+	+	+	+	N/A	N/A

<sup>+</sup> Splněno

<sup>\*</sup> Zdarma na vlastním HW.

<sup>\*\*</sup> Zdarma jen bez spolupráce na úkolech

N/A Pro daný nástroj nemá kritérium význam.

Kr. 1 – Nevyžaduje vlastní infrastrukturu, rychlé zprovoznění

Kr. 2 – Použitelnost

Kr. 3 – Absence nepotřebné funkcionality

Kr. 4 – Použití zdarma

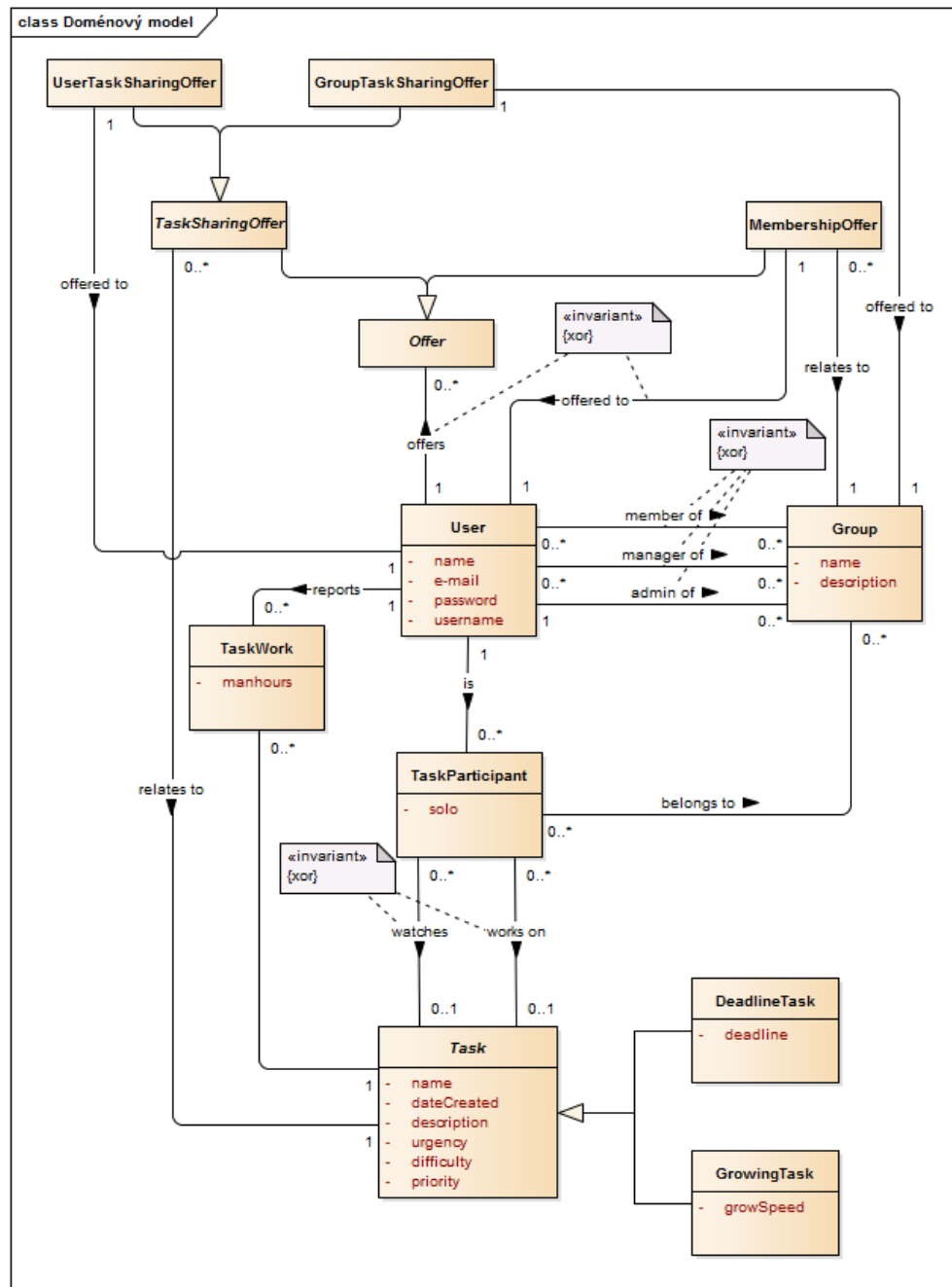
Kr. 5 – REST API

Kr. 6 – Otevřený zdrojový kód





## **Doménový model**



Obrázek C.1: Doménový model aplikace Deadlines

---

## Detail REST API

### D.1 Chybové kódy

V této sekci uvedu chybové kódy, které může aplikace vrátit v těle odpovědi. Její formát s lze nalézt v sekci 4.2.3.1.

- 1** Uživatelské jméno již existuje.
- 2** Parametry v požadavku jsou chybné. Chybí, nebo obsahují nepovolené hodnoty.
- 3** Chybný parametr pro filtr v požadavku.
- 4** Uživatel nemá požadovanou roli v úkolu – není pracovníkem.
- 5** Uživatel není účastníkem úkolu.
- 6** Chybné přihlašovací údaje.
- 7** Nelze nastavit deadline na rostoucím úkolu.
- 8** Nelze upravit rychlost růstu rostoucího úkolu.
- 9** Uživatel je již účastníkem úkolu.
- 10** Uživatel není členem skupiny.
- 11** Uživatel nemá dostatečná práva ve skupině.
- 12** Uživatel není vlastníkem zadané nabídky.
- 13** Uživatel, který vytvořil nabídku, již nemá dostatečná práva pro její uskutečnění. Pokud například Manažer skupiny pozve uživatele do skupiny a přestane být Manažerem dříve, než je nabídka přijata, pak nebude přijetí uskutečněno.

## D. DETAIL REST API

---

- 14 Uživatel je již členem skupiny.
- 15 Úkol je již sdílen se skupinou.
- 16 Parametr `role` má chybnou hodnotu.
- 17 Jméno skupiny již existuje.
- 18 Úkol není sdílen se skupinou.
- 19 Role Admina nemůže být změněna tímto způsobem.
- 20 Admin nemůže být odebrán ze skupiny.
- 21 Úkolu s deadline nemůže být vynulována urgentnost.

### D.2 Endpointy

V této části uvedu pro každý API endpoint detaily o formátu požadavku a odpovědi. Uvedené odpovědi předpokládají, že požadavek proběhl bez chyby. Pokud není uvedeno jinak, HTTP kód odpovědi je 200 OK. V případě chyby při zpracování požadavku aplikace vrací odpovídající chybový HTTP kód a v těle případně detaily chyby, jak jsem popsal v sekci 4.2.3.1.

Nepovinná pole v požadavku jsou označena symbolem `-`.

#### D.2.1 GET /user

Formát odpovědi:

```
[
  {
    "id": 1,
    "username": "jdoe",
    -"email": "sample@email.com",
    -"name": "John Doe"
  },
  {
    ...
  }
]
```

#### D.2.2 POST /user

Formát požadavku:

```
{
  "username": "jdoe",
  "password": "sample-password",
  "-email": "sample@email.com",
  "-name": "John Doe"
}
```

Formát odpovědi je stejný jako formát požadavku. Při úspěšném vytvoření uživatele je vrácen kód 201 CREATED.

### D.2.3 GET /user/{user\_id}

Formát odpovědi:

```
{
  "id": 1,
  "username": "jdoe",
  "-email": "sample@email.com",
  "-name": "John Doe"
}
```

### D.2.4 PUT /user/{user\_id}

Formát požadavku:

```
{
  "-password": "new-password",
  "-email": "new@email.com",
  "-name": "New Doe"
}
```

Formát odpovědi:

```
{
  "id": 1,
  "username": "jdoe",
  "-email": "new@email.com",
  "-name": "New Doe"
}
```

### D.2.5 GET /task

Možné parametry:

**order** určuje způsob řazení úkolů. Výchozí způsob je podle urgentnosti. Povolené hodnoty jsou:

## D. DETAIL REST API

---

**name** řazení podle jména;

**date** řazení podle data vytvoření úkolu;

**deadline** řazení podle deadlinu, úkoly bez deadlinu jsou umístěny na konec;

**worked** řazení podle procentuální rozpracovanosti;

**priority** řazení podle priority;

**urgency** řazení podle urgentnosti.

**orderdirection** určuje směr řazení. Výchozí směr je sestupně. Povolené hodnoty jsou:

**asc** řazení vzestupně;

**desc** řazení sestupně.

**rolefilter** určuje filtr rolí – budou zobrazeny pouze úkoly, v nichž má uživatel danou roli. Povolené hodnoty jsou:

**watcher** zobrazí jen úkoly, v nichž je uživatel Pozorovatelem;

**worker** zobrazí jen úkoly, v nichž je uživatel Pracovníkem.

**typefilter** určuje filtr podle typu úkolu. Povolené hodnoty jsou:

**deadline** zobrazí jen úkoly s deadlinem;

**growing** zobrazí jen rostoucí úkoly.

**statusfilter** určuje filtr podle stavu úkolu. Povolené hodnoty jsou:

**open** zobrazí jen úkoly ve stavu *otevřený*;

**inprogress** zobrazí jen úkoly ve stavu *rozpracovaný*;

**completed** zobrazí jen úkoly ve stavu *splněný*;

**cancelled** zobrazí jen úkoly ve stavu *zrušený*.

**priorityfilter** určuje filtr podle priority. Jednotlivé hodnoty lze kombinovat pro zobrazení různých priorit. Povolené hodnoty jsou *lowest*, *low*, *normal*, *high* a *highest*.

Příklad URL požadavku na tento endpoint, který zobrazí všechny moje úkoly s prioritou 2 nebo 3, seřazené podle rozpracovanosti vzestupně:

```
GET /task?order=worked
    &orderdirection=asc
    &priorityfilter=high
    &priorityfilter=highest
```

Formát odpovědi:

```
[
  {
    "id": 4,
    "name": "Task4",
    "priority": "LOWEST",
    "status": "OPEN",
    "urgency": {
      "lastUpdate": "2016-05-01 16:24",
      "value": 85.31240623606338
    },
    "deadline": "2016-05-02 02:24",
    "workedPercentage": 0.0,
    "type": "DEADLINE"
  },
  {
    ...
  }
]
```

#### D.2.6 POST /task

Formát požadavku:

```
{
  "name": "task name",
  "description": "task description",
  "priority": "LOW",
  "workEstimate": "13",
  "deadline": "2016-05-17 13:15",
  "hoursToPeak": "3",
  "groupIds": [1, 3, 8]
}
```

Požadavek musí mít vyplněno právě jedno z polí `deadline` a `hoursToPeak`. Podle nich se určí typ vytvořeného úkolu.

Pole `groupIds` může obsahovat ID skupin, v nichž je volající uživatel manažer. Úkol bude po vytvoření automaticky sdílen s těmito skupinami.

Formát odpovědi:

```
{
  "id": 2,
  "dateCreated": "2016-05-01 16:11",
```

## D. DETAIL REST API

---

```
"name": "task name",
"description": "task description",
"workEstimate": 13.0,
"priority": "LOW",
"status": "OPEN",
"urgency": {
  "lastUpdate": "2016-05-01 16:11",
  "value": 0.0
},
"workReports": [],
"participants": [
  {
    "solo": true,
    "role": "WATCHER",
    "user": {
      "id": 1,
      "username": "User1"
    }
  }
],
"hoursToPeak": 12.5 |or| "deadline": "2016-05-17 12:00",
"workedPercentage": 0.0,
"manhoursWorked": 0.0,
"groups": [
  {
    "id": 1,
    "name": "Group1"
  },
  {
    "id": 2,
    "name": "Group4"
  }
],
"type": "GROWING"
}
```

V tomto příkladu je úkol s ID 2 sdílen s jedním uživatelem (*User1*), který je v roli pozorovatele, a se dvěma skupinami (*Group1*, *Group4*).

Při úspěšném vytvoření úkolu je vrácen kód 201 CREATED.

### D.2.7 GET /task/{task\_id}

Formát odpovědi:



```
{
  "id": 1,
  "dateCreated": "2016-05-06 16:17",
  "name": "Task1",
  "description": "Descr tekynvoc",
  "workEstimate": 15.0,
  "priority": "NORMAL",
  "status": "OPEN",
  "urgency": {
    "lastUpdate": "2016-05-06 16:17",
    "value": 87.50850024146979
  },
  "workReports": [
    {
      "manhours": 10.2,
      "userId": 1
    }
  ],
  "participants": [
    {
      "solo": true,
      "role": "WATCHER",
      "user": {
        "id": 1,
        "username": "User1"
      }
    }
  ],
  "deadline": "2016-05-07 02:17",
  "workedPercentage": 0.0,
  "manhoursWorked": 0.0,
  "groups": [],
  "type": "DEADLINE"
}
```

#### D.2.8 PUT /task/{task\_id}

Formát požadavku:

```
{
  "description": "new task description",
  "priority": "NORMAL",
  "workEstimate": "42",
```

## D. DETAIL REST API

---

```
"deadline": "2016-05-17 13:15"
}
```

Formát odpovědi:

```
{
  "id": 1,
  "dateCreated": "2016-05-06 16:31",
  "name": "Task1",
  "description": "new task description",
  "workEstimate": 42.0,
  "priority": "NORMAL",
  "status": "OPEN",
  "urgency": {
    "lastUpdate": "2016-05-06 16:31",
    "value": 87.50850024146979
  },
  "workReports": [],
  "participants": [
    {
      "solo": true,
      "role": "WORKER",
      "user": {
        "id": 1,
        "username": "User1"
      }
    }
  ],
  "deadline": "2016-05-17 13:15",
  "workedPercentage": -1.0,
  "manhoursWorked": 0.0,
  "groups": [],
  "type": "DEADLINE"
}
```

### D.2.9 POST /task/{task\_id}/reseturgency

Tělo požadavku i tělo úspěšné odpovědi jsou prázdné.

### D.2.10 POST /task/share/{task\_id}

Formát požadavku:

```
{
  "userIds": [
```

```
    2, 3, 5
  ],
  "groupIds": [
    1, 3
  ]
}
```

Tělo odpovědi je prázdné.

#### D.2.11 POST /task/leave/{task\_id}

Tělo požadavku i tělo úspěšné odpovědi jsou prázdné.

#### D.2.12 POST /task/role/{task\_id}

URL parametry požadavku:

**newRole** Nová role, která má být uživateli přiřazena.

**targetUser** Nepovinný parametr. ID uživatele, u kterého má být změna provedena.

**targetGroup** Nepovinný parametr. ID skupiny, v rámci které má být provedena změna role uživatele identifikovaného parametrem **targetUser**.

Tělo požadavku i tělo úspěšné odpovědi jsou prázdné.

#### D.2.13 POST /task/report/{task\_id}

Obsahem URL parametru **worked** je desetinné číslo, určující počet odpracovaných hodin, které mají být zaznamenány.

Například POST /task/report/1?worked=5.3 zaznamená odpracování 5.3 hodin na úkolu s ID 1 volajícím uživatelem.

#### D.2.14 GET /offer/task/user

Formát odpovědi:

```
[
  {
    "id": 2,
    "offerer": {
      "id": 1,
      "username": "User1",
      "email": "email@address.com",
      "name": "User 1 name"
    }
  },

```

## D. DETAIL REST API

---

```
    "taskOffered": {
      "id": 5,
      "name": "Task5",
      "status": "OPEN",
      "type": "DEADLINE"
    }
  },
  {
    ...
  }
]
```

### D.2.15 POST /offer/task/user/resolve/{task\_offer\_id}

URL parametr `accept` určuje, zda bude nabídka přijata či odmítnuta. Možné hodnoty jsou `true` pro přijetí nabídky, či `false` pro odmítnutí nabídky.

Například `POST /offer/task/user/resolve/3?accept=true` přijme nabídku na sdílení úkolu s uživatelem s ID 3.

### D.2.16 GET /offer/task/group/{group\_id}

Formát odpovědi je stejný jako v sekci D.2.14.

### D.2.17 POST

`/offer/task/group/{group_id}/resolve/{task_offer_id}`

Parametry jsou stejné jako v sekci D.2.15.

### D.2.18 GET /offer/membership

Formát odpovědi:

```
[
  {
    "id": 6,
    "offerer": {
      "id": 1,
      "username": "User1",
      "email": "Email@address.com",
      "name": "User 1 name"
    },
    "group": {
      "id": 1,
      "name": "Group1"
    }
  }
]
```

```
    },  
    {  
      ...  
    }  
  ]
```

#### D.2.19 POST

`/offer/membership/resolve/{membership_offer_id}`

Parametry jsou stejné jako v sekci D.2.15.

#### D.2.20 GET /group

Nepovinný URL parametr **role** určuje filtr skupin, které se mají zobrazit. Jeho možné hodnoty jsou:

**any** Zobrazí všechny skupiny, kterých je volající uživatel členem v jakékoliv roli.

**admin** Zobrazí všechny skupiny, kterých je volající uživatel Admin.

**manager** Zobrazí všechny skupiny, kterých je volající uživatel Manažer.

**member** Zobrazí všechny skupiny, kterých je volající uživatel Člen.

Pokud není parametr zadán, zobrazí se všechny skupiny v aplikaci.

Formát odpovědi:

```
[  
  {  
    "id": 1,  
    "name": "Group1",  
    "adminInfo": {  
      "id": 1,  
      "username": "User1",  
      "name": "User 1 name",  
      "email": "Email@address.com"  
    }  
  },  
]
```

#### D.2.21 POST /group

Formát požadavku:

## D. DETAIL REST API

---

```
{
  "name": "NewGroup",
  "description": "Description of the new group."
}
```

Formát odpovědi:

```
{
  "id": 5,
  "description": "Description of the new group.",
  "name": "NewGroup",
  "participants": [],
  "sharedTasks": [],
  "members": [
    {
      "user": {
        "id": 1,
        "username": "User1",
        "email": "Email@address.com",
        "name": "User 1 name"
      },
      "role": "ADMIN"
    }
  ]
}
```

Při úspěšném vytvoření skupiny je vrácen kód 201 CREATED.

### D.2.22 GET /group/{group\_id}

Formát odpovědi je stejný jako odpověď v sekci D.2.21.

### D.2.23 PUT /group/{group\_id}

Formát požadavku:

```
{
  "description": "New description for the group"
}
```

Formát odpovědi je stejný jako odpověď v sekci D.2.21.

### D.2.24 DELETE /group/{group\_id}

Tělo požadavku i tělo úspěšné odpovědi jsou prázdné.

**D.2.25 POST /group/{group\_id}/member/offer**

Formát požadavku:

```
{
  "userIds": [1, 3, 8]
}
```

**D.2.26 PUT /group/{group\_id}/member/{user\_id}**

Formát požadavku:

```
{
  "role": "MANAGER"
}
```

Možné hodnoty pro pole `role` jsou *MEMBER* nebo *MANAGER*.

**D.2.27 DELETE /group/{group\_id}/member/{user\_id}**

Tělo požadavku i tělo úspěšné odpovědi jsou prázdné.

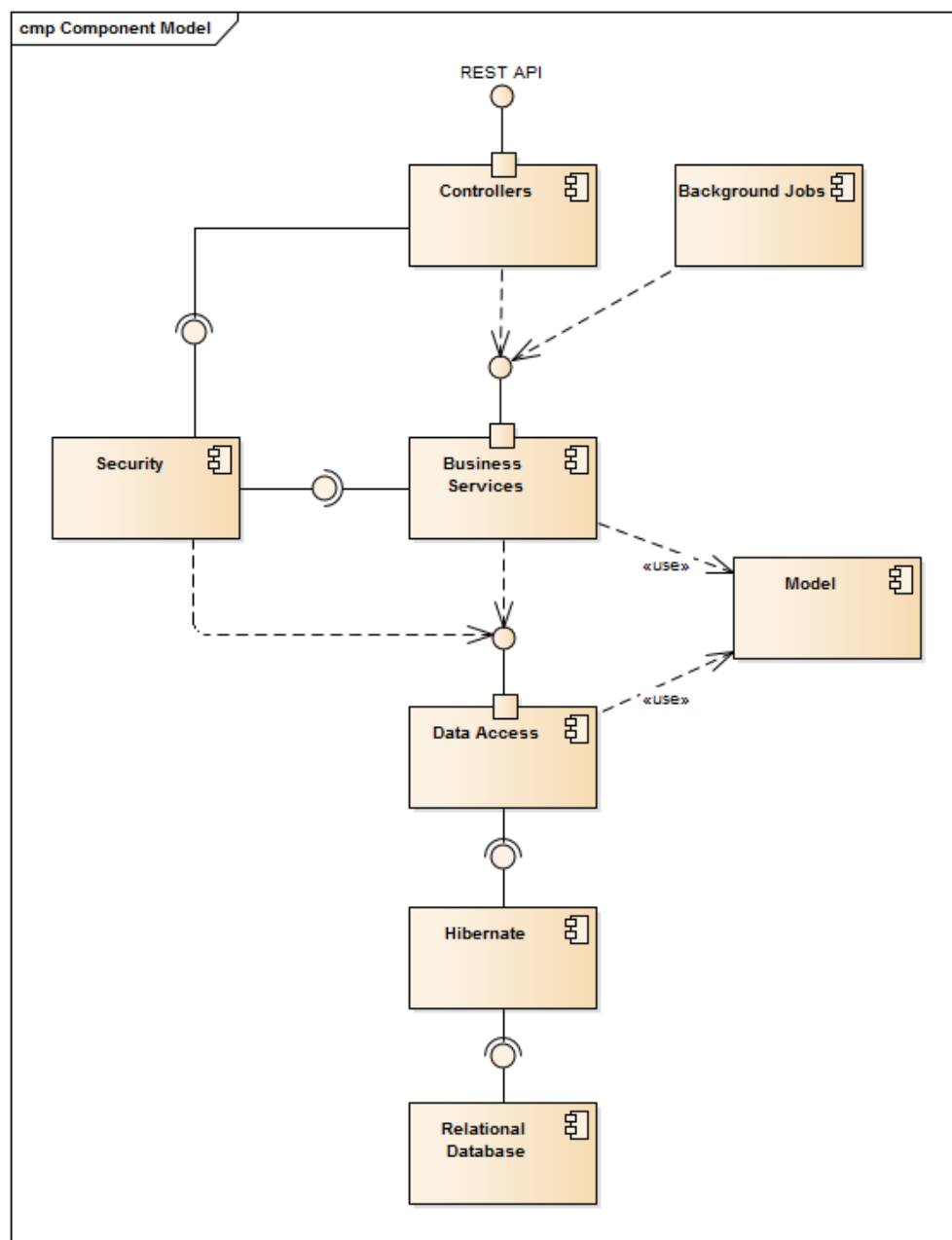
**D.2.28 DELETE /group/{group\_id}/task/{task\_id}**

Tělo požadavku i tělo úspěšné odpovědi jsou prázdné.



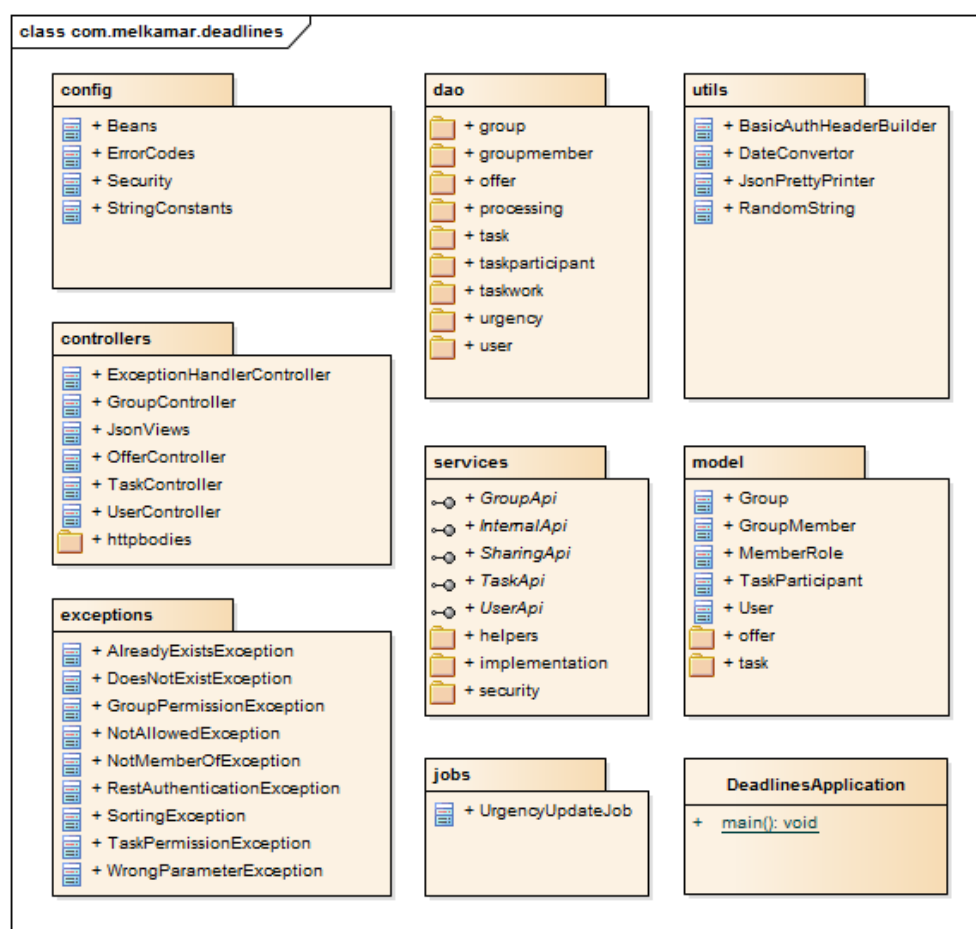


## **Komponenty aplikace**

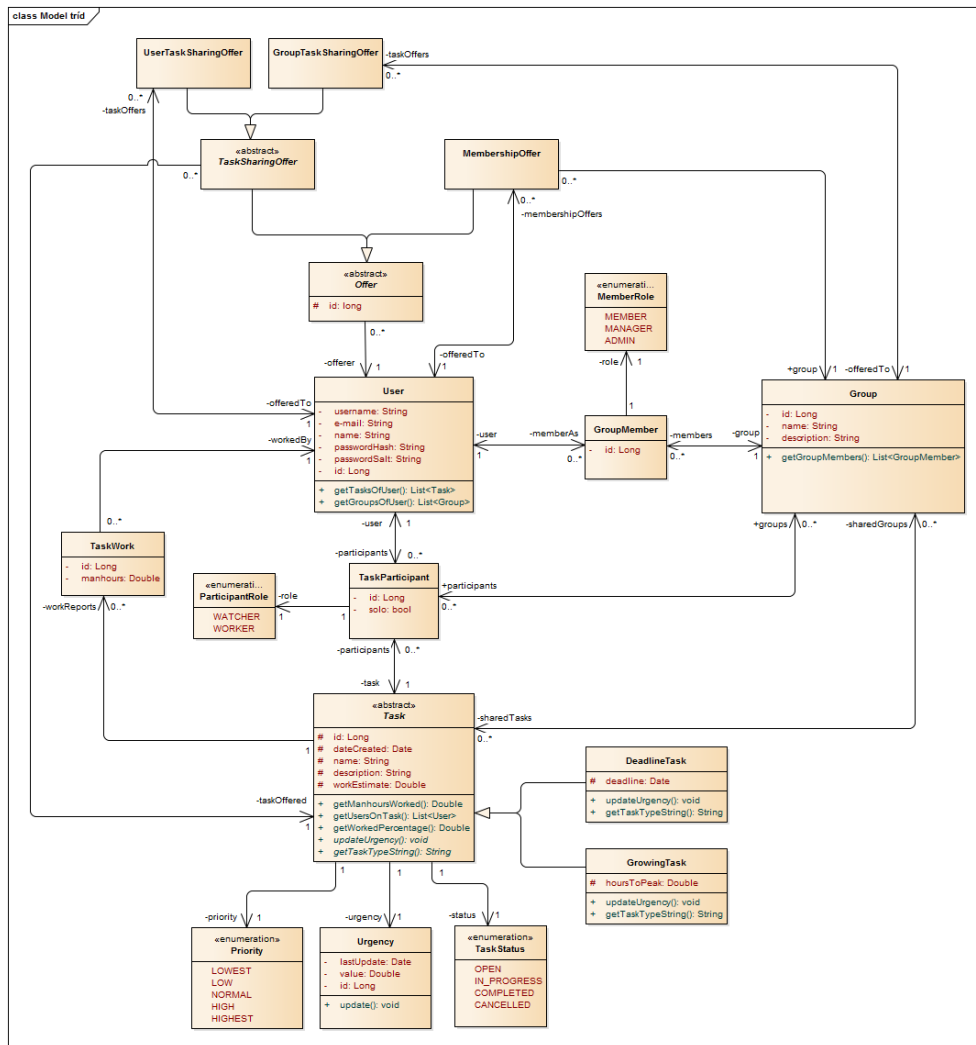


Obrázek E.1: Diagram architektury komponent aplikace

## Návrhový model



Obrázek F.1: Balíčky a třídy aplikace Deadlines

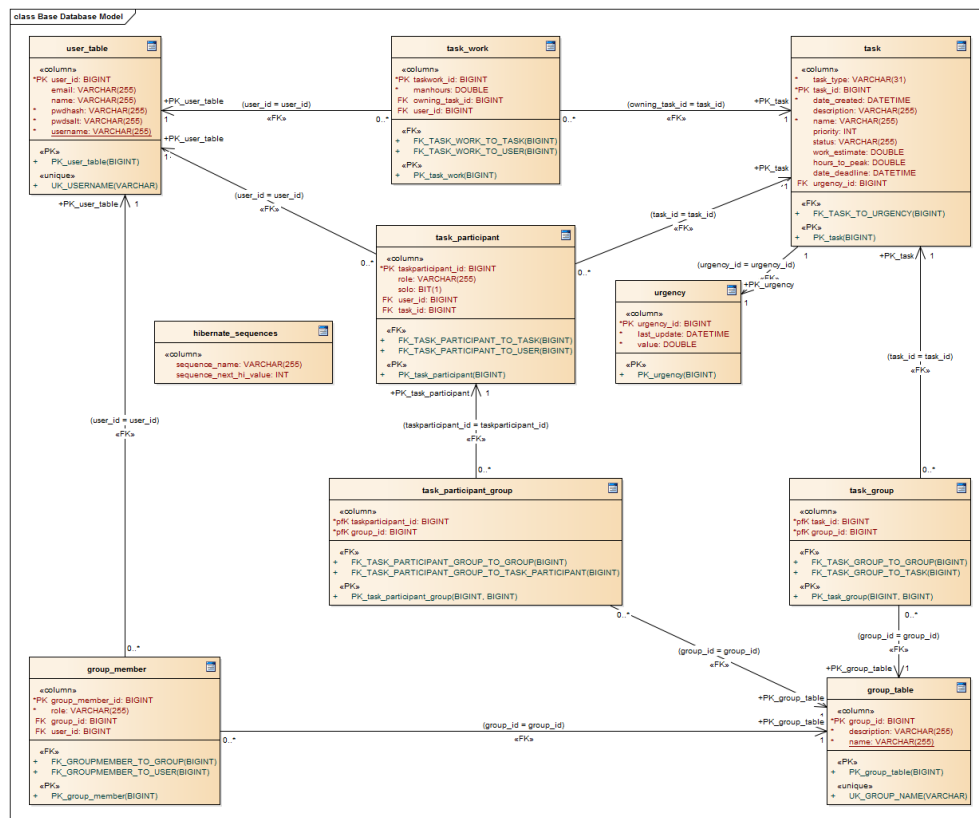


Obrázek F.2: Návrhový model tříd aplikace Deadlines



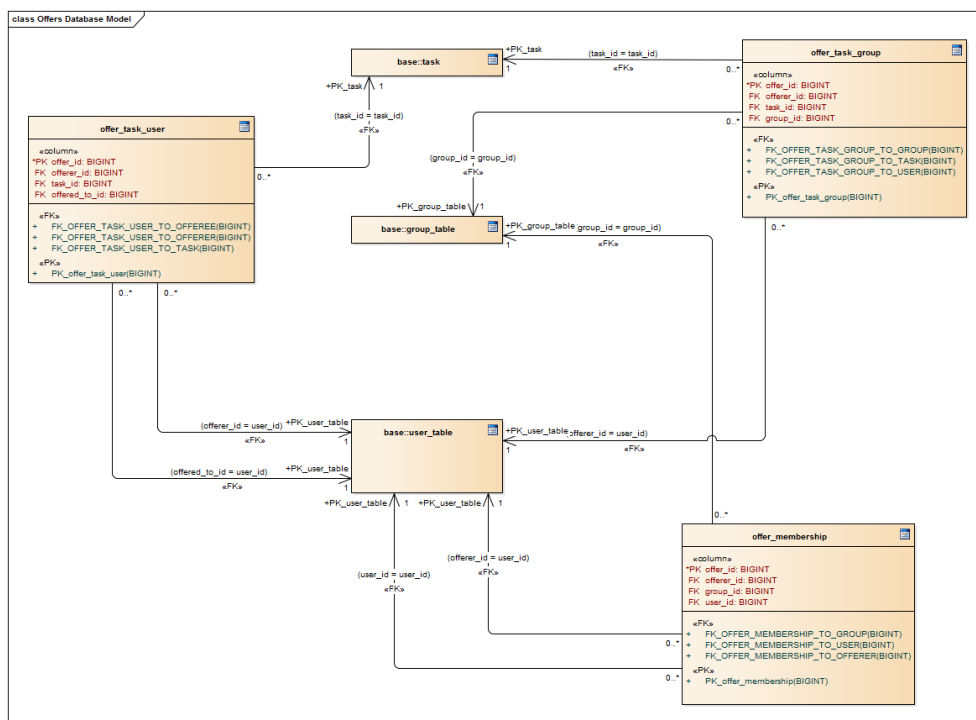
## **Databázový model**

## G. DATABÁZOVÝ MODEL



Obrázek G.1: Relační databázový model entit aplikace, kromě nabídek





Obrázek G.2: Relační databázový model nabídek



## Obsah přiloženého CD

readme.txt.....	stručný popis obsahu CD
deployment.....	soubory pro nasazení aplikace
├─ application.properties .....	konfigurační soubor aplikace
├─ database_init_mysql.sql .....	inicializační skript databáze
├─ deadlines-1.1.1.jar .....	spustitelný soubor aplikace
├─ deadlines-1.1.1.war .....	soubor aplikace pro nasazení na serveru
doc	
├─ ea.....	návrhová dokumentace v nástroji Enterprise Architect
├─ project .....	JavaDoc dokumentace zdrojových kódů
src	
├─ project.....	zdrojové kódy implementace
├─ thesis .....	zdrojová forma práce ve formátu L <sup>A</sup> T <sub>E</sub> X
text .....	text práce
├─ thesis.pdf .....	text práce ve formátu PDF
├─ thesis.ps .....	text práce ve formátu PS