FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Synchronizační middleware mezi projektovým systémem a aplikací na sledování času stráveného na projektech
Student:	Bc. Vít Štefan
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Vytvořte webovou aplikaci, která bude sloužit jako automatický synchronizační nástroj mezi projektovým systémem a aplikací monitoringu stráveného času - např. Toggl. Uživatel v jednom systému vykáže časový údaj k požadavku a výsledná aplikace ho sama doplní do systému druhého. Data budou synchronizována obousměrně.

Postupujte v těchto krocích:
Proveďte analýzu konkurence a srovnání s existujícími nástroji, které řeší podobný problém.
Udělejte sběr a analýzu požadavků budoucích uživatelů.
Analyzujte možné projektové systémy a aplikace na sledování času, pro které bude výsledná aplikace navržena.
S ohlédnutím k požadavkům uživatelů navrhněte vhodný middleware, který bude snadno rozšiřitelný.
Navrhněte webovou aplikaci, kde bude uživatel provádět konfiguraci middleware.
Pomocí vhodných webových technologií implementujte middleware a konfiguračního webového klienta.
Dbejte na univerzálnost, ale také na jednoduchost pro uživatele.
Zhodnoťte výsledky a uveďte další možná rozšíření.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 7. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Synchronizační middleware mezi projektovým systémem a aplikací na sledování času stráveného na projektech

Bc. Vít Štefan

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Hunka

5. května 2021

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Jiřímu Hunkovi za jeho rady během tvorby diplomové práce a při konzultacích. Dále těm, kteří mi pomohli při sběru požadavků a následném testování, zejména Ing. Oldřichu Malcovi. Děkuji také Ing. Pavlu Kovářovi, který mi pomohl s nasazením systému. Samozřejmě děkuji všem, kteří mne při studiu neustále motivovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 5. května 2021 ……

Odkaz na tuto práci

Štefan, Vít. *Synchronizační middleware mezi projektovým systémem a aplikací na sledování času stráveného na projektech.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá návrhem a implementací systému, který má za úkol synchronizovat data mezi projektovými systémy a aplikacemi na sledování času stráveného na projektech. Rešeršní část se věnuje návrhu takového řešení, které je dostatečně obecné, ale přesto jednoduché pro uživatele. V části je provedeno nastínění situace, analýza existujících podobných řešení, ale také sběr požadavků od potenciálních uživatelů. Na základě toho je navržena architektura systému a databázové struktury. Praktická část pak navazuje popisem implementace serveru i frontendové webové aplikace včetně krátkého popisu nasazení systému. Dále je předložen průběh testování systému společně se zpětnou vazbou od uživatelů. Na závěr jsou uvedena možná rozšíření a vylepšení celé aplikace.

Klíčová slova Timer2Ticket, aplikace pro sledování času, projektový systém, synchronizační middleware, middleware, NodeJS, Angular, TypeScript, REST API, Redmine, Toggl Track

Abstract

This thesis is dedicated to design and implement a synchronization system for project management tools and time tracking applications. The research part designs solution that is sufficiently general, but still simple to use. It evaluates the current situation and also analyzes similar existing solutions. The requirements collection is made with potential users. Based on that, the system architecture and database structure are designed. The practical part continues by describing server and frontend web application implementation. Also, the deployment process is briefly mentioned. Furthermore, the testing of the system together with user feedback is introduced. Possible extensions and improvements are suggested at the end of the thesis.

Keywords Timer2Ticket, time tracking application, project management tool, synchronization middleware, middleware, NodeJS, Angular, TypeScript, REST API, Redmine, Toggl Track

Obsah

Seznam obrázků

Seznam výpisů kódů

Úvod

Neexistuje jednoduché a přímočaré řešení, které by zajišťovalo synchronizaci projektových systémů a aplikací na sledování času. Přitom to není zas tak neobvyklý případ, obzvláště v oblasti IT. Ve firmách je obvykle po zaměstnancích vyžadováno, aby strávený čas prací zaznamenávali do nějakého systému. Ty ale většinou nejsou vhodné pro časté přepínání kontextu a efektivní zaznamenávání času. Uživatel tak může využívat jinou aplikaci, která je k tomuto úkolu uzpůsobena. Nastává však problém, neboť časy, které zadal do této aplikace, poté musí pro zaměstnavatele ještě manuálně přepsat do projektového systému. Tento proces ale není pohodlný a může být zdrojem chyb. Proto by realizace systému, který umožní tyto služby synchronizovat na pozadí, dávala velký smysl.

Téma práce jsem si vybral převážně proto, že se mi jevilo smysluplné a dobrý výsledek by mohl znamenat ušetření času a usnadnění práce pro ostatní. Zároveň se jednalo o výzvu, protože vytvořit takový systém, který bude fungovat spolehlivě, není vždy snadné. Vzhledem k tomu, že bude pracovat se službami třetích stran, nebude jednoduché zařídit, aby vše běželo bez problému. Bude potřeba se vyrovnat s chybami a situacemi, které mohou nastat. Navíc jedním z požadavků na systém je jeho co největší obecnost, aby bylo umožněno jej rozšířit o další služby, které bude podporovat. Dalším důležitým prvkem je nutnost, aby byl uživatelsky jednoduchý a přívětivý a přinášel co nejmenší další zátěž. Pokud by se tak nestalo, systém by pro uživatele ztrácel význam.

Výsledkem práce je fungující systém, který propojuje různé služby a zařizuje synchronizaci jejich dat. Prozatím je připravený a otestovaný pro dvě vybrané služby, nicméně je možné a ne tak pracné jej rozšířit o další.

Systém je rozdělen na čtyři celky. Jedná se o databázi, Core (které zajišťuje samotnou synchronizaci), klientskou webovou aplikaci (ve které uživatel službu konfiguruje) a také API (to slouží jako most mezi Core a klientem). Všechny implementované části (Core, klient, API) komunikují přes rozhraní REST (Representational State Transfer). To, že jsou celky od sebe funkčně oddělené, usnadňuje jejich vývoj a testování.

V práci nejprve provedu návrh celé aplikace. Prozkoumám zástupce služeb, které by měly být integrovatelné. Systém a jeho architekturu navrhnu na základě požadavků, které sesbírám od potenciálních uživatelů. Shrnu i datovou strukturu systému, stručně okomentuji bezpečnost a připravím návrh

obrazovek pro webového klienta. V poslední sekci kapitoly o řešerši uvedu technologie použité při implementaci. Na všechny tyto části budu navazovat při realizaci.

Praktická část se zabývá jednotlivými částmi a případnými zajímavostmi, které jsem musel při realizaci řešit. Popíšu i samotné nasazení systému na server. Ke konci celé práce proberu testování nasazené aplikace a uvedu možná vylepšení a rozšíření, které by mohly být do implementace v budoucnu zakomponovány. Na závěr rozeberu také možnosti, jak by mohl být systém monetizovaný proto, aby se vyplatil jeho provoz.

1 Cíle práce

Cílem práce je návrh a implementace takového systému, který bude synchronizovat časové záznamy napříč uživatelem zvolenými službami třetích stran. Služby budou typu projektový systém nebo aplikace na sledování času. Navržený systém bude od uživatele vyžadovat co nejméně práce navíc a výrazně mu zjednoduší správu synchronizace jeho aplikací.

Samotným návrhem se zabývá rešeršní část, kde budou prozkoumány různé služby, které by měly být integrovatelné do systému. Kromě toho také zkoumá, zda již neexistuje řešení tohoto problému. Návrh vychází z kvantitativního výzkumu mezi uživateli, kteří by tuto službu využívali. Návrhem je potřeba docílit co nejobecnější architektury, která bude připravená na případná rozšíření. Důležité je probrat různé scénáře, které mohou při vykonávání synchronizace nastat. Vše je nutné posléze brát vpotaz a vytvořit kvalitní databázový návrh, na kterém systém bude stavět.

Cílem praktické části je navázání na část předchozí samotnou implementací celého systému. Dojde na poukázání na zajímavé části, které musely být při realizaci řešeny. K dispozici bude i krátký popis toho, jak je aplikace nasazená na server. Cílem je také probrání možných vylepšení a rozšíření včetně nastolení možného budoucího rozvoje opřeného o výsledky z testování systému a zpětné vazby uživatelů.

2 Analýza a návrh

V kapitole nejprve uvedu celý problém a nastíním současnou situaci. Dále analyzuji služby, které by mohly být tímto systémem integrovány. Prozkoumám již existující řešení a provedu výzkum mezi uživateli. Také popíšu architekturu celého systému a procesy, které budou probíhat uvnitř aplikace. Nakonec se provedu návrh datové struktury, navrhnu obrazovky pro webového klienta a shrnu použité technologie pro implementaci.

2.1 Nastínění současné situace

Nejdříve popíšu problém, který bude celý systém řešit. Celá řada společností, obzvláště pak v oblasti IT, využívá pro správu projektů a požadavků nějaký nástroj. Mezi takové zástupce se řadí např. Jira [1] či Redmine [2]. V těchto systémech uživatelé pracují nad přiřazenými projekty, vytvářejí požadavky, vyplňují počet hodin, které strávili nad danými úkoly.

Nicméně takové systémy se běžně nehodí na časté přepínání kontextů mezi více projekty. Rovněž neřeší problém efektivního zaznamenávání stráveného času. Pokud uživatel v průběhu dne pracuje na pěti projektech a deseti úkolech, na konci dne už neví, kdy a kolik času na dané aktivitě strávil.

Tento problém lze samozřejmě řešit po svém – psát si záznamy ručně, vyplňovat do tabulky apod. Existují však nástroje, které přesně toto řeší efektivněji a přehledněji za nás. Patří mezi ně např. Toggl Track [3] a Clockify [4]. Jednoduše zadáme kontext, nad kterým zrovna pracujeme, můžeme přidat projekt, tag a spustíme časomíru. Vrhneme-li se na něco jiného, předchozí časomíru zastavíme a spustíme novou. A tak můžeme postupovat napříč dnem. Výhodou je, že počítání času dělá aplikace za nás a my na konci dne vidíme přehled toho, jak jsme byli efektivní a kdy jsme zrovna na onom projektu pracovali.

Potíž nastává samozřejmě ve chvíli, kdy se po uživateli chce tyto údaje vykazovat také v projektovém nástroji zmíněném výše. Musíme pak otevřít aplikaci na sledování času a ručně přepsat všechy údaje do systému druhého. Údaje může uživatel chtít později upravit, smazat, změnit název projektu a dělat další úpravy. Vše musí manuálně převádět z jednoho systému do druhého. Nehledě na stav, kdy k tomu všemu přibude ještě třeba třetí systém v rámci jiného zaměstnání.

Řešením by měl být systém Timer2Ticket (T2T), který je praktickým výstupem této práce. Uživatel si založí účet u T2T, v několika jednoduchých krocích nastaví konfiguraci – zvolí systémy pro synchronizaci, nastaví plánování synchronizace ad. (podrobněji popíšu dále v sekci 2.7). Ve finále by měl většinu práce provádět ve své oblíbené aplikaci, a do dalších mu vše bude převáděno automaticky na pozadí. Eliminuje se tak spousta rutinní práce převádění, odstraní se chyby při manuálním přepisování a problém s nekonzistencí. Do dalších systémů se převedou objekty jako jsou projekty, požadavky ad. dle služby. Následná synchronizace časových záznamů bude probíhat obousměrně. Tedy v jedné službě uživatel vykáže čas a do dalších se mu převede. V jiné pak tento čas upraví a následně se změna propíše opět do všech a tak dále.

Na obrázku 2.1 jsem znázornil příklady úkonů, které by uživatel musel provádět ručně v případě, že by byl v situacích popsaných výše. Jedná se o přiřazení nového úkolu uživateli a sledování stráveného času nad ním. Také jsem uvedl kroky, které by uživatel podstoupil, kdyby se projektu změnil název. Vždy je uvedena posloupnost kroků v aktuálním stavu a pak ve stavu, kdyby uživatel měl účet u T2T. Pro názornost rovnou uvádím konkrétní projektový systém (Redmine) a aplikaci na sledování času (Toggl Track). Dále v práci budu používat tyto dva příklady pro demonstraci. Zejména proto, že systém bude navržený na základě výzkumu mezi pracovníky spolupracující se společností Jagu s.r.o., kteří využívají právě zmíněné aplikace.

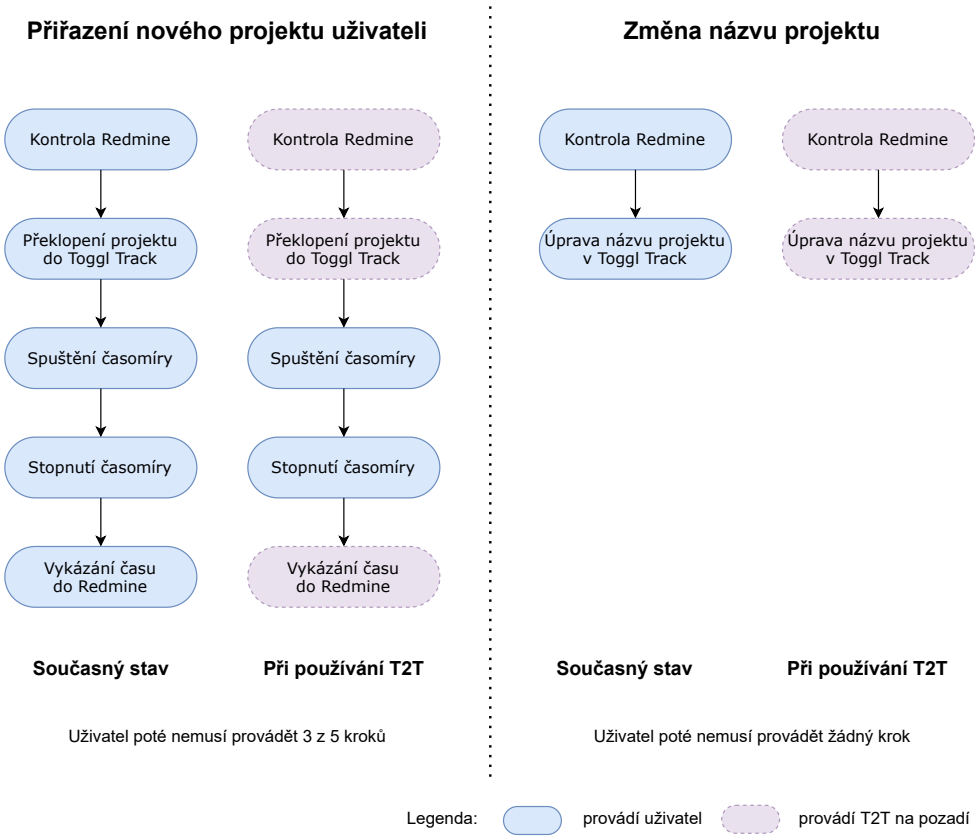
2.2 Analýza služeb pro synchronizaci

V celé sekci se věnuju službám, které by mohly být synchronizovány pomocí Timer2Ticket. Ekosystém bych měl koncipovat tak, aby byl co nejvíce obecný a aby mohl být snadno rozšířitelný o novou službu. Na základě analýzy pak budu implementovat integraci některých zde uvedených služeb. Ačkoliv by mělo být v principu jedno, zda se jedná o projektový systém, či o aplikaci na sledování času, rozhodl jsem se, že zde je popíšu zvlášť.

2.2.1 Projektové systémy

Projektový systém je zpravidla zavedený ve firmě a zaměstnanec do něj musí vykazovat svou pracovní činnost. Obvykle obsahuje projekty, kterými se firma zabývá. Projekty mohou mít požadavky (*issues*). Zaměstnanec vybírá projekt, nebo konkrétní požadavek. K němu vykáže strávený čas. Projekty a požadavky můžou „zestárnout“ – jsou vyřešené, zapomenuté. Zde se budu detailněji věnovat systémům Redmine, Jira a Nutcache [6]. Všechny uvedené jsou dostupné přes API klíč, ale ne všechny umožňují implementaci Web Hooks[1]. Samozřejmě existují další, ale ne každý projektový systém je založený na podobné bázi.

[1] Web Hooks („webové háčky“) dovolují vytvořit kód, který se spustí pouze při nějaké akci – např. při přidání nového časového záznamu. Znamená to, že pro tuto službu není nutné využívat polling – opakované dotazování se, zda se nezměnil stav.



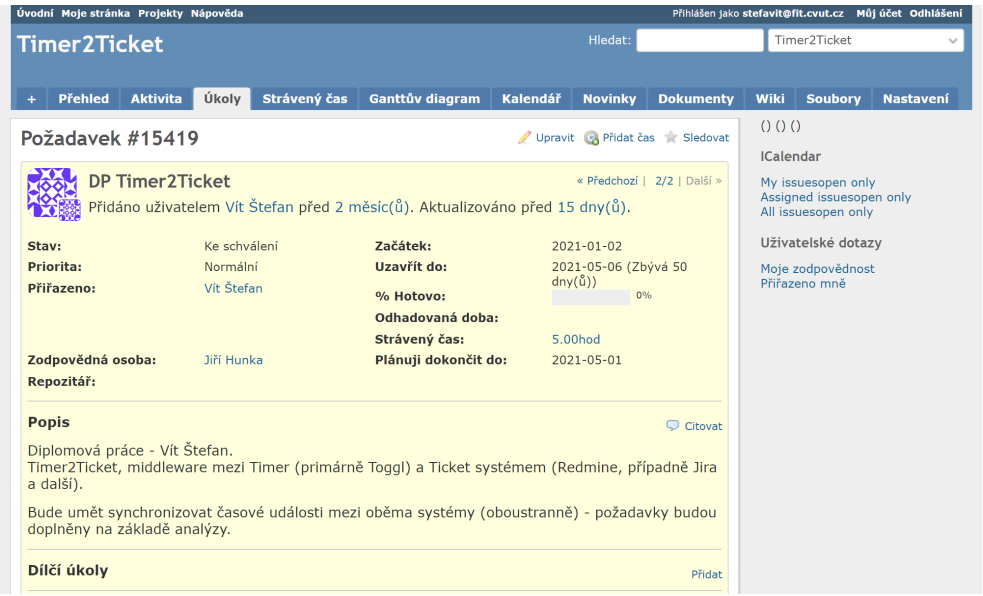
Obrázek 2.1: Aktuální stav vs. stav po konfiguraci T2T, základní příklady, naznačená posloupnost aktivit (vytvořeno pomocí [5]), bez notace

Redmine

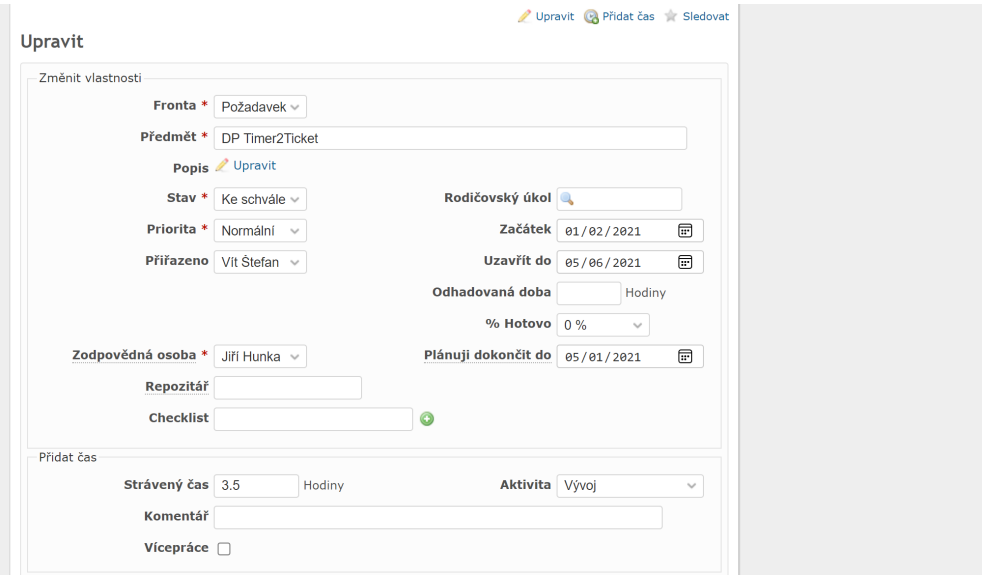
Redmine je open source systém, který si může kdokoliv nainstalovat na vlastní server a spravovat v něm projekty. Vyznačuje se svou jednoduchostí. Zde lze zakládat projekty, přiřazovat požadavky, vkládat časové záznamy (k projektům i požadavkům), přidávat k požadavkům pracovníky, psát komentáře ad. Na obrázku 2.2 (na další straně) uvádím snímek detailu požadavku přímo ze systému. Časový záznam k požadavku lze vkládat přes vlastní formulář po kliknutí na tlačítko „Přidat čas“. Také je možné jej vložit přímo při editaci požadavku (obrázek 2.3 rovněž na další straně; vlastní formulář umožňuje to samé a vypadá obdobně).

Speciální vlastností je zadávání aktivity k časovému záznamu (např. „návrh“, „vývoj“, „administrativa“). Aktivity jsou definované globálně pro organizaci a je nutné je k časovému záznamu přidat.

Protože je integrace Redmine do T2T jedním z požadavků, zaměřil jsem se na něj více. Pro přihlašování se za jiného uživatele přes REST, mi dovoluje využít API klíč. Ten uživatel získá v přehledu účtu. Znalost klíče dovolí T2T



Obrázek 2.2: Snímek ze systému Redmine, přehled požadavku



Obrázek 2.3: Snímek ze systému Redmine, vkládání časového záznamu při editaci požadavku (samotný časový záznam ve spodní části snímku)

vykonávat za uživatele nejdůležitější práci: načítat projekty a jejich časové záznamy, vkládat a upravovat časové záznamy.

Přístup k REST API[2] je poměrně přímočarý. Je nutné k dotazu přiložit API klíč a následně určit, na jakou adresu požadavek poputuje. Protože má každá organizace svůj vlastní přístupový bod (základ adresy), je nutné tuto informaci získat při konfiguraci od uživatele. Zmiňoval jsem aktivity časových záznamů – pro větší pohodlí uživatele navrhuji, aby si mohl zvolit výchozí, která se použije vždy, když uživatel nezvolí žádnou. Celkově tedy bude nutné při konfiguraci získat od uživatele API klíč, přístupový bod API a výchozí aktivitu. To je vše, co T2T potřebuje, aby dále vykonávalo synchronizaci samostatně. Nebudu zde uvádět přesné dotazy na API, lze je vyčíst v dokumentaci Redmine a ty důležité pro T2T jsou k dispozici také v příloze této práce (přímo ve zdrojovém kódu systému). Redmine umožňuje využívat technologii Web Hooks.

Jira

Jira je stálicí v oblasti projektového systému. Jedná se o komerční řešení, které ale nabízí plány zdarma. Chová se podobně jako Redmine, místy nabízí pokročilejší funkce. Šikovnou věcí, kterou Jira nabízí (a Redmine v základu ne, resp. jsou k dispozici pouze jako rozšíření, které ne každá společnost může mít nainstalované), jsou štítky (*labels*). Uživatel si tak může jednotlivé úkoly označit svými vlastními značkami. To by se dalo využít pro integraci tohoto systému do T2T.

Jak jsem již psal, základní funkcionalitou se neliší od Redmine, také umožňuje zadávat k jednotlivým požadavkům časové záznamy.

Mají pěkně zpracovanou API dokumentaci, dovolují i vytvářet Web Hooks. Opět je možné zastupovat uživatele pomocí API klíčů. Samozřejmě je umožněno posílat dotazy na projekty a požadavky uživatele, rovněž tak provádět editaci časových záznamů.

Nutcache

Nutcache je další z řady komerčních produktů, které umožňují správu projektů a týmu. Stejně jako Jira ale nabízí plán zdarma. Chová se velmi podobně, umožňuje spravovat projekty, vytvářet požadavky (úkoly), nabízí podporu štítků. Má dostupné API, opět přes přístupový klíč. Technologii Web Hooks ale nepodporuje.

Přece jen se ale trochu liší v oblasti logování času. Čas se zadává přímo k úkolům (tedy ne ve formě časových záznamů, ale každý úkol má svůj jeden logovaný čas). Navíc přímo v aplikaci umožňuje sledovat čas – uživatel může spustit časomíru a vypnout, když na úkolu přestane pracovat. Kromě toho je

[2]Representational State Transfer Application Programming Interface – zjednodušeně rozhraní pro komunikaci mezi různými službami, aplikacemi…

zde k dispozici vlastní extra řešení sledování času. To je ale dostupné v rámci placeného účtu.

2.2.2 Aplikace na sledování času

Aplikace na sledování času může být zavedená ve firmě, může ji ale uživatel začít používat sám na firmě nezávisle. Často totiž nabízí snadné přepínání kontextu a hlavně sledování času, které pak uživatel využije při výkazu hodin do projektového systému. Zde se budu detailněji věnovat aplikacím Toggl Track, Clockify a Harvest [7]. Společnou vlastností je přehledový plán, kde jsou vidět jednotlivé časové úseky, na kterých uživatel pracoval. Obsahují jednoduchý ovladač pro zapnutí a vypnutí časomíry. K časovému záznamu se může zvolit nejčastěji projekt, případně úkol a štítek.

Toggl Track

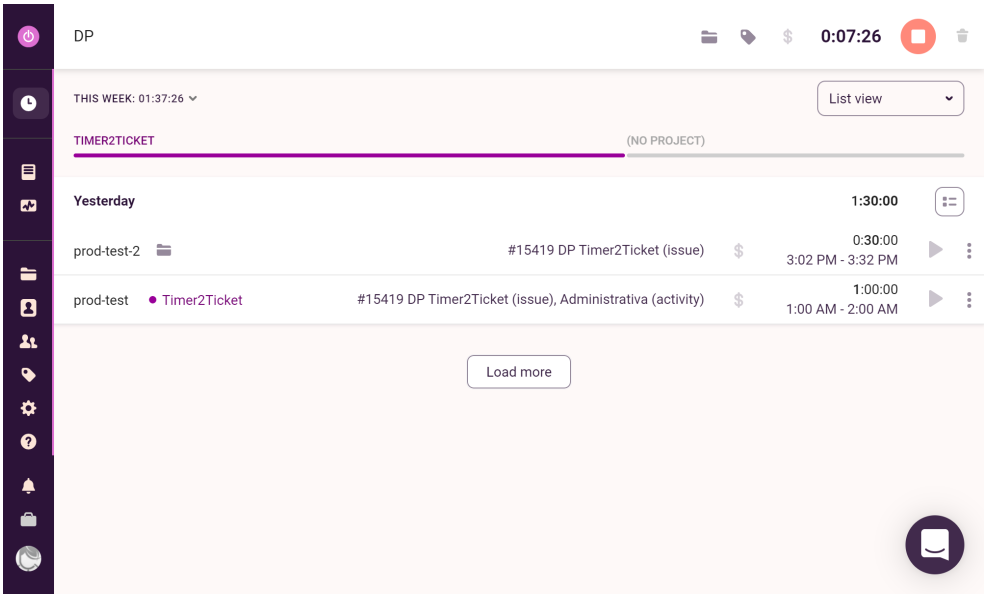
Toggl Track nabízí plán zdarma, je však ochuzený o nějaké prémiové funkce. Ty však pro mě a T2T nejsou důležité. Zásadní je práce s projekty a tagy (štítky), na které se dají mapovat jakékoli objekty. Pokud služba štítky disponuje, dá se říci, že aplikace může být integrovaná do T2T. Lehce tak mohu vzít aktivity z Redmine, štítky z Jira nebo úkoly z Nutcache a namapovat je na tagy v Toggl Track.

Jinak je práce v Toggl Track přímočará: zapnu časomíru, vyberu projekt, tagy a mohu pracovat. Tlačítkem stop pak ukončím časomíru. Jak jsem již naznačil, v aplikaci velmi snadno přepínám kontexty, jednoduše zapnu novou časomíru a přidělím jiný projekt. Vše mám pak dostupné v přehledných reportech. Na obrázku 2.4 uvádím snímek z aplikace. Jedná se o základní obrazovku, kde uživatel vidí zaznamenané časy a může zde také spouštět časomíru (viditelné v horní části snímku).

Při práci v aplikaci musím mít aktivní workspace (pracovní místo/plochu). Do ní pak umísťuji nové projekty a tagy. Toggl Track je dobře dostupný i přes rozhraní REST. Timer2Ticket opět bude za uživatele provádět příkazy pomocí svěřeného API klíče. Toggl Track nepodporuje Web Hooks.

Clockify

Tato aplikace je opravdu velmi podobná Toggl Track. Také umožňuje stopovat čas, volit projekt, přidávat tagy, měnit workspace. API je opět dostupné pomocí klíče. Clockify umožňuje implementovat Web Hooks, ale pouze částečně (pro mé využití): umožňuje poslouchat na vytvoření a úpravě/smazání časového záznamu, stejně tak na vytvoření projektu. Neumožňuje však reagovat na úpravu projektu. Proto, kdyby se o integraci pomocí T2T uvažovalo, stejně by se musel nasadit polling na editaci a odstranění projektů.



Obrázek 2.4: Snímek z aplikace Toggl Track, přehled včetně časomíry

Harvest

Harvest také nabízí mj. řešení sledování stráveného času ať pro jednotlivce, nebo pro týmy. Největším úskalím pro integraci s T2T je absence jakékoli podoby štítků. Znamená to, že by se jiné objekty než projekty musely vpisovat do poznámek a následně nějakým způsobem parsovat pro synchronizaci s jinou službou. To by byla velká překážka hlavně pro uživatele, pravděpodobně by to přestávalo být intuitivní. API je přístupné opět přes uživatelský klíč, technologie Web Hooks není podporována.

2.3 Již existující řešení

Problém synchronizace projektového systému a aplikace na sledování času samozřejmě řešila spousta uživatelů. Při manuálním převádění vzniká spousta chyb a spousta zbytečné práce, kterou uživatel nechce dělat.

Pokud používáme Toggl Track, jednou z možností je instalace oficiálního doplňku [8] do prohlížeče od společnosti stojící za touto aplikací. Je pro nás připravena celá škála doplňků pro různé projektové systémy, namátkou Jira, Redmine. Nicméně samotný doplněk pouze přidává tlačítko do vybrané služby k jednotlivým úkolům. Po stisknutí tlačítka vybereme popisek, tagy a toggl pouze spustí časomíru, kterou následně můžeme vypnout. Je to kostrbaté řešení, které nezajistí synchronizaci v pravém slova smyslu. Navíc jsme stále závislí na dané službě (Redmine, Jira) a musíme v ní pracovat. Toto řešení není ani obecné, je na míru dělané Toggl Track.

Toggl Track nově (od března 2021) nabízí lepší řešení [9] v podobě opravdové synchronizace. Jedná se však pouze o synchronizaci mezi Toggl Track a Jira. Uživatelé ostatních systémů tedy nemohou toto řešení využít. Jak jsem již zmínil, jedná se o skutečnou synchronizaci. Na pozadí se nejspíš vytvoří podobná mapování, na která cílíme v T2T. Nejedná se o obecné řešení, které by řešilo libovolnou službu. Na závěr dodám, že se jedná o placenou službu (pro prémiové účty).

Podívame-li se na stranu Redmine, jsou k dispozici Redmine doplňky vytvořené komunitou, které nějakou základní synchronizaci do různých služeb obsahují. Jedním takovým příkladem je doplněk (plugin) Toggl 2 Redmine [10]. Uživatel ho musí přímo vpravit do instalace Redmine. Po instalaci může uživatel v Redmine navštívit záložku Toggl, kde se nabídnou časové záznamy pro import. Uživatel zvolí aktivitu, případný popisek ad. Data se načtou přímo z Togglu, ale je nutné mít v popisku časového záznamu vyplněné číslo požadavku Redmine. Bez toho se záznamy nenabídnou k importu. Nakonec uživatel stiskne tlačítko pro import. Z popisu je vidno, že záznamy vlastně uživatel importuje sám – jen se mu předpřipraví. Myslím si, že tento doplněk není moc vhodný z důvodu manuálního převodu, nutnosti vlastní instalace do Redmine a také z toho důvodu, že funguje jen mezi Redmine a Toggl Track. Navíc „synchronizace“ probíhá pouze jedním směrem. Výhodou je samozřejmě to, že uživatel nepotřebuje speciální aplikaci či službu, vše běží jako doplněk v Redmine.

Robustním a obecným řešením by na první pohled mohla být automatizační služba Zapier [11]. V ní mohu vytvářet události, které se spustí na základě události v synchronizované službě. V principu bych mohl vytvořit např. událost při vložení Redmine časového záznamu, která mi vytvoří časový záznam v Togglu. Bohužel ale nejde nový časový záznam vytvořit dynamicky na základě hodnot v Redmine. Tudíž si pouze mohu vytvořit „šablonu“, kterou stejně posléze budu muset manuálně upravit. V průběhu procesu vkládám API klíč, nastavuji události na základě se vytváří jiné události. Musel bych vytvořit události v jedné službě na nový projekt, nový časový záznam… Musel bych také vytvořit zrcadlové události v dalších synchronizovaných službách. Tento postup je velice kostrbatý a matoucí. Navíc, celá konfigurace systému se mi nezdá uživatelsky přívětivá. Výhodou a zároveň nevýhodou je velká obecnost Zapier. Sice umožňuje automatizovat stovky služeb, není ale zaměřený na služby jako jsou Redmine a Toggl Track. Znamená to tedy, že námi chtěnou automatickou synchronizaci dělá polovičatě.

Celkově mi přijde, že jediné schůdné řešení nabízí nově Toggl Track a tím je jeho Jira sync. Jak jsem již ale psal, jedná se pouze o synchronizaci mezi dvěma danými službami. Navíc je vyžadován premium účet. Ostatní možnosti řeší synchronizaci pouze částečně či kostrbatě. Robustní a přesto jednoduché řešení dle mé rešerše neexistuje.

2.4 Výzkum mezi uživateli, sběr a analýza požadavků

Výzkumem jsem si chtěl ověřit, zda směr, kterým se vydávám, je správný a zda bude výsledek uživatelům vyhovovat. Na testování se podílelo pět uživatelů, kteří spolupracují se společností Jagu s.r.o. Všichni povinně využívají systém Redmine pro vykazování práce. Někteří z nich používají také Toggl Track z osobní iniciativy pro trasování svých aktivit během dne a stráveného času na nich. U těch, kteří ho nepoužívají jsem chtěl zjistit, proč tak nečiní, a zda by ho využívali, kdyby T2T zprostředkovávala synchronizaci. Veskrze jsem chtěl načerpat co nejvíce požadavků, které bych pak vtělil do samotné aplikace.

2.4.1 Nástin kvalitativního výzkumu

Protože uživatelů bylo relativně málo, mohl jsem si dovolit udělat průzkum do hloubky a každému se věnovat osobně. Výzkum probíhal vždy formou videohovoru. Hovor jsem nahrával a v průběhu si psal poznámky. S účastníky jsme se dohodli, že nahrávky nebudou zveřejněny. Nicméně je mám lokálně dostupné a s dodatečným souhlasem by se do nich nahlédnout mohlo. Poznámky, které jsem si v průběhu dělal, jsou k dispozici (anonymizované) na přiloženém médiu – viz příloha C s obsahem média.

Inspirací pro interview mi byla bakalářská práce studenta FIT ČVUT Denise Talára [12]. Rovněž tak jsem pro přípravu využil video [13], které nabízí tipy pro semi-strukturovaný rozhovor. Dalším zdrojem pro mě byl článek, ve kterém kupříkladu zdůrazňují, jak je důležité na začátku „prolomit ledy“. Navodí se tím lepší atmosféra a testovaní se budou cítit více uvolnění. [14]

Připravil jsem si jednoduchou strukturu, podle které jsem rozhovor vedl. Scénář je dostupný v příloze B.1 na konci textu. Snažil jsem se rozhovor vést ve formě diskuze, ale zároveň nechat dotazované mluvit co nejvíce, případně je jen trochu směřovat. Po úvodu, kde jsem nastínil funkci systému jsme se pobavili o tom, jak momentálně pracují s danými službami, a jak by si asi synchronizaci představovali. Na závěr jsme celý rozhovor shrnuli a ujistili jsme se, že zůstaneme v kontaktu i nadále. Mám možnost je kontaktovat přes firemní komunikační nástroj, kde máme i založený kanál, ve kterém jsme v průběhu návrhu a implementace diskutovali.

2.4.2 Výsledky

Rozhovory trvaly mezi třiceti a šedesáti minutami. Všichni se víceméně shodli na tom, že je pro ně nejdůležitější forma skutečně oboustranné synchronizace. Dovedou si totiž představit, že na běžné měření času využívají Toggl Track, ale někdy zase využijí Redmine. Každá změna v jednom systému se tedy musí převést i do dalších. V Timer2Ticket je potřeba důkladně vyřešit mapování, aby i změna názvu projektu byla skutečně správně převedena do další aplikace.

Každý by také uvítal nějakou možnost kustomizace, např. zvolení výchozí aktivity pro Redmine. Z uživatelského hlediska také vychází, že systém musí být co nejjednodušší. V ideálním případě by vše mělo běžet automaticky na pozadí. I za cenu toho, že neobvyklé funkcionality musí uživatel řešit manuálně (např. vykazování za jiného uživatele). Pokud by tam ta možnost byla, bylo by to nejlepší, ale pouze jako doplňková a rozšiřující. Systém by měl fungovat i v nějakém základu, bez nadstaveb.

Ohledně mapování objektů, nejvíce rozporů panovalo v tom, jak přenášet informaci o projektech, požadavcích a aktivit z a do Redmine. Možnosti jsou v zásadě dvě: buď takovéto věci namapovat do tagů v Toggl Track, nebo je psát ručně do komentáře a T2T by měla za úkol informace z textu extrahovat. Výhoda tagů je, že jsou připravené pro uživatele v případě, že se přiřadí nový projekt. Zároveň se dá vyřešit úprava názvu apod. Jedná se totiž o reálný objekt, který se dá editovat. Ruční psaní do komentáře by vyžadovalo mít znalost, jaký identifikátor požadavek má a bylo by nutné ho vždy kopírovat do aplikace. Dle mého názoru je to méně komfortní a intuitivní. Znamenalo by to ale, že se aplikace nezatěžují tolika objekty (požadavků mohou být stovky či tisíce na uživatele) a nemuselo by se ani řešit promazávání starých nepotřebných projektů ad.

Systém by také mohl nějakým způsobem notifikovat uživatele o aktuálním stavu, případně o chybách při synchronizaci. Minimálně tedy by na nějakém místě měla být dostupná informace, zda je vše v pořádku, případně datum a čas poslední úspěšné synchronizace. Ještě lépe by systém mohl notifikovat přes nějaký komunikační prostředek – např. e-mail. Vhodné by ale bylo, aby si uživatel mohl nastavit míru notifikace (a třeba ji i úplně vypnout).

2.4.3 Funkční a nefunkční požadavky

Jedním z cílů je navhrnout skutečně robustní řešení, které ale bude dostatečně jednoduché a pro uživatele bude znamenat pouze pár kroků při konfiguraci aplikace. Vše ideálně bude probíhat na pozadí a aplikace nebude vyžadovat uživatelský zásah.

Funkční a nefunkční požadavky vycházejí hlavně ze zadání práce, ale také z výzkumu mezi reálnými uživateli služeb, které by T2T měla integrovat. Celkově také vymezují rámec celého systému. Začnu funkčními požadavky (přímo popisují, co systém bude dělat):

F1 – Rozšířitelnost navržený middleware bude snadné rozšířit o další systémy pro synchronizaci.

F2 – Univerzálnost systém bude co nejvíce univerzální, ale tak, aby pokryl drtivou většinu základních požadavků uživatele.

F3 – Jednoduchost konfigurační proces a výsledná práce se synchronizací bude pro uživatele co nejjednodušší.

F4 – Flexibilita konfigurace konfiguraci půjde změnit i zpětně, tedy např. nebude problém přidat novou službu pro synchronizaci k existujícím.

F5 – Obousměrnost synchronizace synchronizace mezi systémy bude probíhat obousměrně, tedy jakákoli změna v jednom systému bude propagována do ostatních.

Z funkčních požadavků vyplývá, že pro uživatele musí být nastavení a konfigurace přímočará a mělo by toho být po něm vyžadováno co možná nejméně. To samé platí o samotných úkonech nutných ke správné synchronizaci časových záznamů. O to více si musím při návrhu systému dát pozor na jednoduchost a robustnost. Nefunkční požadavky (volí technologie):

N1 – Konfigurace ve webové aplikaci Ve webové aplikaci se uživatel registruje a nakonfiguruje systém. Vždy bude vyžadovat připojení k internetu. Nebude příliš dbáno na starší verze prohlížečů. Ovládání bude standardní jako u běžných webových aplikací.

N2 – REST API komunikace mezi klientem a serverem bude probíhat přes REST API. Pomocí téhož rozhraní bude také probíhat komunikace mezi dílčími službami uvntř systému T2T.

2.5 Architektura

Po důkladném rozboru všech podnětů jsem navrhoval architekturu celého systému. Z pohledu uživatele musí existovat webová aplikace, kde si vytvoří účet a kde bude nastavovat konfiguraci systému. Dále bylo jasné, že někde bude probíhat samotná synchronizace konfigurace a časových záznamů. Ideální by navíc bylo, kdyby tato služba byla izolovaná od klientské části. Protože bude ve webové aplikace prováděna také registrace, autentizace a nastavování, které přímo nesouvisí se synchronizací, bylo by vhodné, kdyby o tomto synchronizační služba nevěděla.

Rozhodl jsem se tedy pro ještě jednu službu, která bude sloužit jako jakýsi most mezi klientem a synchronizací. Ta by se kromě úkonů spojených s uživatelem ještě starala o manuální zapínání a vypínání synchronizace na příkaz uživatele. Bude tedy tyto požadavky přeposílat na synchronizační službu.

Celkově se tedy systém bude skládat ze čtyř částí:

T2T Core (jádro) stará se o synchronizaci všech služeb uživatele. Bude dostupné ve formě služby.

T2T API zvládá jednoduché úkony, které požaduje klient – registraci, autentizaci, úpravu konfigurace, zjištění stavu synchronizace, manuální zapnutí/vypnutí synchronizace. Vystaveno pomocí rozhraní REST ve formě služby.

T2T Klient slouží jako komunikační prostředek s uživatelem. Provádí ho procesem, který vede k úspěšné synchronizaci. Umožňuje zkontrolovat stav, ale také pozastavit a spustit synchronizaci. Webová aplikace.

Databáze ukládá stav systému.

Klient komunikuje pouze s API. To komunikuje s jádrem, ale také se službami na synchronizaci pro jednoduché dotazy při konfiguraci uživatele – detailněji sekce 2.7 dále. Jádro plánuje a spouští synchronizační joby[3] – k tomu potřebuje také komunikovat se službami třetích stran. Schématicky znázorněná architektura systému je k vidění na obrázku 2.5. U každé komunikační šipky jsem naznačil, při jaké příležitosti daná komunikace probíhá (výčet není nutně kompletní). Veškerá komunikace mezi jednotlivými částmi probíhá přes rozhraní REST.

2.6 Jak bude probíhat synchronizace

T2T Core bude obsahovat frontu jobů, která se bude periodicky kontrolovat. Bude-li obsahovat nějaké joby na splnění, vyjme první a vyřídí ho. Job by měl ohlásit, zda proběhl korektně, či ne. Pokud ne, bude vložen znovu na konec fronty. Z mého pohledu by však neměl být takto vkládán neustále, měl by být hlídán a po nějaké rozumné mezi opakování by již znovu neměl být vpuštěn do fronty. Další jeho spuštění bude pak opakováno dle prováděcího plánu.

Pomocí webového klientu si každý uživatel nastaví, jak často chce synchronizovat svá data. Dle toho tedy budou požadavky vkládány do fronty. Kromě toho bude uživatel mít možnost joby spouštět manuálně přes webového klienta. Pro rozumnější orchestraci jsem navrhl, že se požadavky budou dělit do tří kategorií:

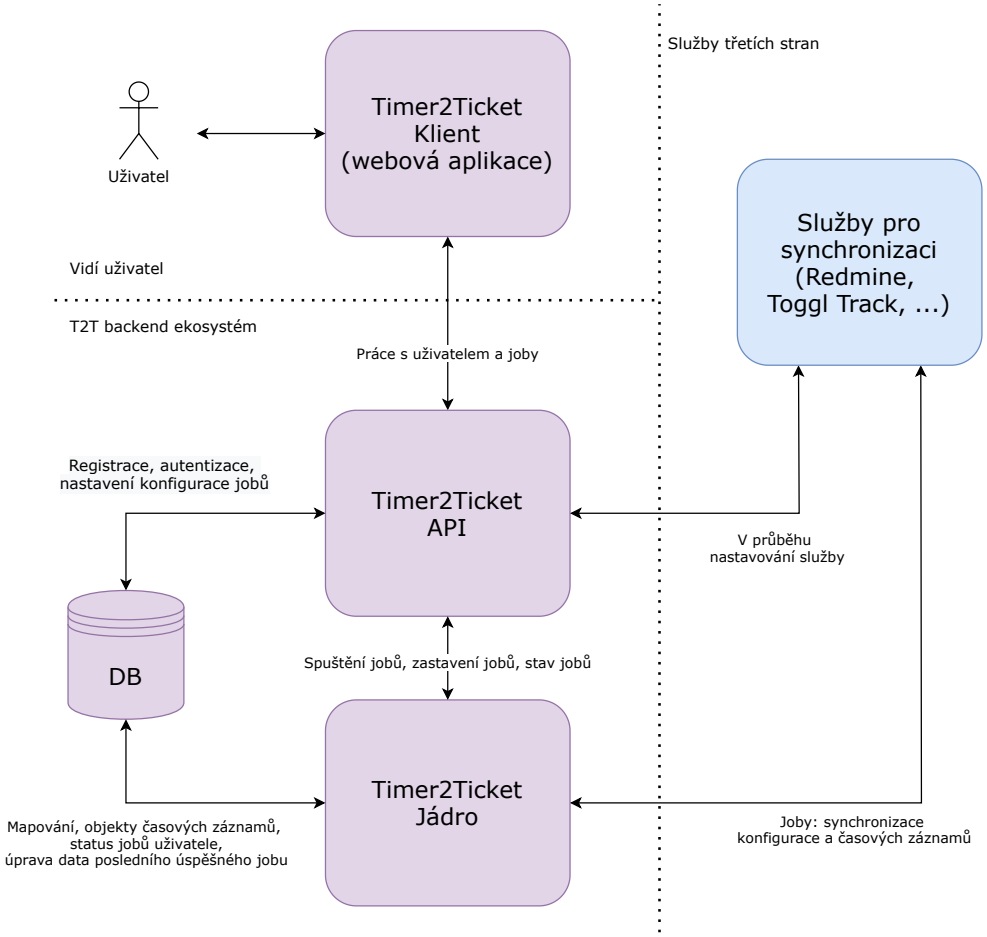
Job synchronizace konfigurace bude spravovat objekty služeb: projekty, požadavky, apod. Jeho úkolem je, aby všechny tyto objekty byly aktuální. Stane-li se nějaká změna, tento job zařídí, že bude propagována do ostatních služeb. Také bude rozlišovat primární a neprimární služby – více viz podsekce 2.6 dále.

Job synchronizace časových záznamů řídí napříč všemi službami synchronizaci časových záznamů.

Úklidový job se stará o promazávání starých, nepoužívaných, či nepotřebných objektů vzniklé konfiguračním jobem.

Aby T2T systém věděl, které reálné objekty spolu souvisí, musí si někde tuto informaci udržovat. V databázi tedy bude muset existovat ke každému uživateli přehled mapování. Tento postup je nutný jak u objektů (projekt,

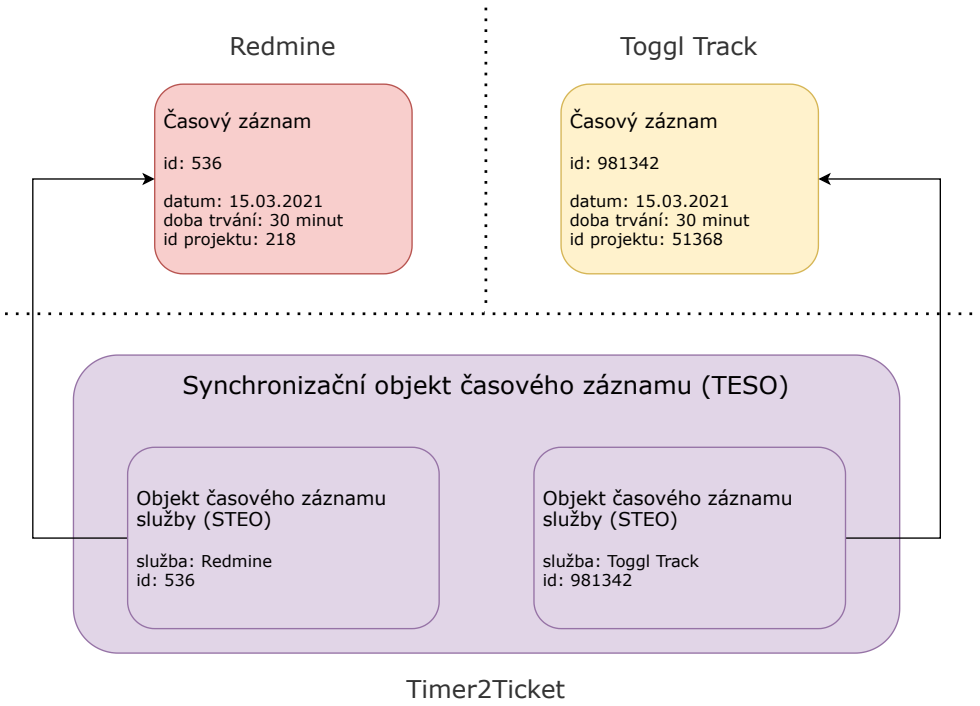
[3] Jobs – práce, také úkoly… Budu používat počeštěný výraz „joby“, vždy tak bude jasné, co mám na mysli.



Obrázek 2.5: Přehled architektury ekosystému T2T (vytvořeno pomocí [5]), schematicky zaneseno, bez notace

požadavek), tak i u časových záznamů. Dojde-li totiž ke změně v jednom systému, musí se dle mapování vyhledat příslušné objekty v jiných systémech a změna propagovat. U všech časových záznamů jsem tento mapovací objekt nazval „synchronizační objekt časového záznamu“ (Time Entry Synced Object, zkráceně TESO). Tento objekt pak obsahuje kolekci „objektů časového záznamu služby“ (Service Time Entry Object, zkráceně STEO), kde každý z nich představuje danou službu a identifikátor v ní. Pro ostatní objekty (projekty, požadavky…) je to jednoduše „mapování“ (Mapping) a kolekce „mapovacích objektů“ (MappingsObject).

Mapování časového záznamu ilustruje schéma na obrázku 2.6 (na další straně; jedná se o zjednodušenou podobu). Na něm jsou zobrazeny reálné časové záznamy, které existují v systémech Redmine a Toggl Track. Zároveň pak existuje mapovací objekt TESO v T2T databázi, který obsahuje dva STEO,



Obrázek 2.6: Zjednodušené schéma mapování časových záznamů (vytvořeno pomocí [5]), bez notace

jeden si uchovává odkaz na objekt v Redmine a druhý na objekt v Toggl Track pomocí jejich identifikátorů. Toto nastane tehdy, kdy uživatel v jedné ze služeb vytvoří časový záznam a ten pak T2T synchronizuje se službou další. Více ještě dále v jednotlivých podsekcích a také v sekci 2.8 věnované databázi. Následuje detailnější popis jobů.

Job synchronizace konfigurace

Zde je důležité rozdělení primárních a neprimárních služeb (volí uživatel v nastavení služeb). Z průzkumu požadavků totiž vyplývá, že vždy existuje jeden referenčí systém, od kterého se vyvíjí ty ostatní. V našem konkrétním případě se jedná o Redmine, jehož projekty, požadavky a aktivity se mají převádět do Toggl Track (a ostatních služeb).

Při prvním spuštění jobu to funguje takto: T2T Core si vyžádá z primární služby objekty vhodné pro synchronizaci. V případě Redmine se jedná o projekty a požadavky přiřazené uživateli, dále také všechny aktivity (více viz podsekce 2.2.1 o Redmine). Tyto by následně měl předat ostatním službám. To, jak je převedou, si definují samostatně. Např. Toggl Track převádí Redmine projekt na svůj projekt, vše ostatní do tagů. Poté se musí uživateli přiřadit interní T2T objekt, který bude informaci o mapování uchovávat.

Při každém dalším spuštění se zkontroluje seznam mapování[4] uživatele. Následně se ze všech služeb extrahují reálné objekty (např. reálný projekt v Redmine a Toggl Track) a zkontroluje se, zda u nich neproběhla změna. Zde se vždy pohlíží na hodnoty v primární službě. Pokud ano, aktualizují se dle primárního objektu. Pakliže objekt v primární službě chybí, je odstraněn z ostatních služeb. Celkově by se dal postup jobu shrnout následujícími scénáři (odvíjí se vždy od primárního reálného objektu)[5]:

a) **Mapování chybí:**
   1) propaguj objekt do ostatních služeb,
   2) vytvoř mapování s mapovacími objekty.

b) **Mapování existuje, ale není aktuální:**
   1) aktualizuj mapování,
   2) propaguj změny do ostatních služeb.

c) **Mapování existuje, ale primární reálný objekt neexistuje:**
   1) odstraň příslušné objekty z ostatních služeb,
   2) odstraň mapování.

d) **Mapování existuje, je shodné jako primární objekt:**
   1) nedělej nic, vše je aktuální.

e) **Mapování existuje, ale mapovací objekt pro nějakou službu neexistuje:**
   1) vytvoř objekt ve službě,
   2) přidej mapovací objekt k mapování.

f) **Mapování existuje, mapovací objekt pro danou službu také, ale reálný objekt neexistuje:**
   1) vytvoř objekt v dané službě,
   2) aktualizuj mapovací objekt.

Job synchronizace časových záznamů

Tento job se neopírá o primární službu, vše probíhá oboustranně. Jedině při mazání časového záznamu se dbá na jeho původ. Synchronizace probíhá následovně: ze všech služeb se načtou časové záznamy, ty se pak T2T pokusí spojit se „synchronizačním objektem časového záznamu“ (TESO – viz výše 2.6)

[4]Mapování bude muset být uloženo v databázi jako objekt, který bude uchovávat odkazy na reálné objekty ve službách, více také viz sekce 2.8 o struktuře dat dále.

[5]Reálně se vlastně nemusí rozlišovat, zda se jedná o první, či další běh jobu, vždy je postup shodný. V případě prvního běhu se vybere vždy možnost a).

a pro každou službu s jejím „objektem časového záznamu služby“ (STEO – opět viz 2.6). Postup se dá shrnout následovně (na základě reálného časového záznamu z jakékoli služby):

a) **TESO pro daný časový záznam neexistuje:**
   1) pokud je důvod[6], propaguj do ostatních služeb,
   2) vytvoř novou TESO s kolekcí STEO pro každou službu.

b) **TESO existuje a pro každou službu je zde i STEO:**
   1) zkontroluj, zda nějaká ze služeb neobsahuje novější verzi:
      - pokud ano, uprav záznamy v ostatních službách (včetně časové značky TESO),
      - pokud ne, vše je v pořádku, nedělej nic.

c) **TESO existuje, ale výčet STEO je nekompletní:**
   1) nejdříve zkontroluj, zda ve zbývajících službách není novější verze – jako b) výše,
   2) synchronizuj s chybějícími službami,
   3) vytvoř nové STEO a přiřaď do TESO,
   4) uprav časovou značku TESO.

d) **TESO existuje, ale časový záznam chybí v původní službě:**
   1) smaž přidružený časový záznam ze všech služeb,
   2) odstraň TESO.

e) **TESO existuje, STEO pro nějakou nepůvodní službu také, ale přidružený časový záznam v dané službě chybí:**
   1) vytvoř nový časový záznam,
   2) aktualizuj STEO,
   3) aktualizuj časovou značku TESO.

2.7 Proces konfigurace synchronizace a vložení nového časového záznamu z pohledu uživatele

V této sekci budu ilustrovat, jak budou uživatelé postupovat v případě, že mají založené služby Redmine a Toggl Track[7] a chtějí mezi nimi synchronizovat

[6] Časový záznam nutně nemusí být určen k synchronizaci – chybí přiřazený projekt, požadavek apod.

[7] Opět se spíše zaměřím na tyto dvě konkrétní služby. Slouží to pro lepší ilustraci. Pro jiné služby by uživatel postupoval obdobně.

data pomocí Timer2Ticket. V této situaci se nacházejí i osoby spolupracující se společností Jagu s.r.o.

2.7.1 Konfigurace Timer2Ticket z pohledu uživatele

Jak jsem již zmiňoval výše, konfigurace systému musí být pro uživatele co nejjednodušší. V ideálním případě by uživatel vše nastavil jednou, a posléze by se do aplikace nemusel vracet. Bylo by ale vhodné, kdyby si mohl zkontrolovat, zda je vše v pořádku, případně i vypnul/zapnul synchronizaci.

Konfigurace bude probíhat ve webové aplikaci. Bude tedy dostupná odkudkoliv ze zařízení, které disponuje přístupem k internetu a webovým prohlížečem. Uživatel se bude do aplikace přihlašovat pomocí uživatelského jména a hesla. Jméno by mělo být ve formě e-mailu. Je to dnes již rozšířená praxe a uživatele se může tímto e-mailem v budoucnu kontaktovat.

Po registraci a prvním přihlášení by si měl zvolit služby, které chce synchronizovat. Po výběru budou následovat konfigurace jednotlivých služeb. Nutné bude vyplnit API klíč, pomocí něhož T2T bude komunikovat s danou službou za přihlášeného uživatele. Tento klíč je běžně dostupný v nastavení takovýchto služeb a má formu delšího textového řetězce.

Také bude nutné vyplnit údaje specifické pro danou službu. V případě Redmine (její popis je k nalezení výše v podsekci 2.2.1) to je ještě přístupový bod (protože každá organizace má svůj). Po vyplnění uživatel potvrdí a T2T provede dotaz na danou službu. Načte tím další potřebná data pro zvolení. U Redmine stáhne možné aktivity pro výběr výchozí. V případě Toggl Track (popis v podsekci 2.2.2) pak všechny workspace opět pro volbu vychozího. Kromě načtení těchto dat také T2T zkontroluje, zda se lze pomocí vložených API klíčů se službou korektně spojit.

Po definování nutných hodnot pro každou zvolenou službu by si měl uživatel zvolit plánování synchronizačních jobů. Volí plánování pro konfigurační job a job pro synchronizaci časových záznamů. Měl by mít možnost dostatečného výběru: např. jednou denně, jednou za 2 hodiny, každé pondělí a středu… Nakonec vše zvolené potvrdí. Tím se postup uloží do databáze a proběhne naplánování jobů dle zvolených hodnot. Mimo jiné se uskuteční ještě první konfigurační job (je nutnou prerekvizitou pro druhý job).

Snažil jsem se, aby proces byl co nejjednodušší. Zároveň je ale jistě výhodné (vyplývá i z průzkumu mezi cílovými uživateli, viz 2.4), že uživatel má možnost pokročilého nastavení (např. defaultní aktivity). Celý postup jsem shrnul ve schématu v notaci UML[8] na obrázku 2.7 na další straně. Vše uvedené předpokládá hladký běh napříč konfigurační cestou. Tento proces se dá krásně rozvhrnout do dílčích kroků, které budou uživateli servírovány postupně.

[8]Unified Modeling Language – unifikovaný jazyk pro modelování procesů a dalších struktur využívaný zejména v oblasti IT.

2.7.2 Vložení nového časového záznamu tak, aby se korektně synchronizoval

Uvažujme následující modelovou situaci: uživatel již má nakonfigurovaný systém T2T, kde synchronizované služby jsou Redmine (primární) a Toggl Track. Zároveň korektně proběhl první konfigurační job. To odpovídá situaci, kterou jsem popsal v podsekci výše. Nyní ukážu, co budou muset uživatelé udělat pro to, aby se jejich nově založené časové záznamy správně synchronizovaly.

Chtějí-li uživatelé založit nový časový záznam v aplikaci Toggl Track, nejprve[9] spustí časomíru. Dále zvolí buď rovnou požadavek pomocí tagu (T2T pak nepotřebuje mít zvolený projekt), a nebo projekt (a nepovinně požadavek – pokud požadavek nezvolí, časový záznam se přiřadí k projektu). Poté nepovinně vyberou k záznamu tag s aktivitou (jinak se vyplní výchozí) a vepíšou nepovinný komentář. Po stopnutí časomíry je záznam připraven k synchronizaci (nevypnutá – běžící – časomíra se synchronizuje s dobou trvání nastavenou na minimální možnou hodnotu). Výhodou Toggl Track je snadné hledání požadavků a aktivit mezi jinými tagy pomocí zabudovaného samodoplňovacího textového políčka.

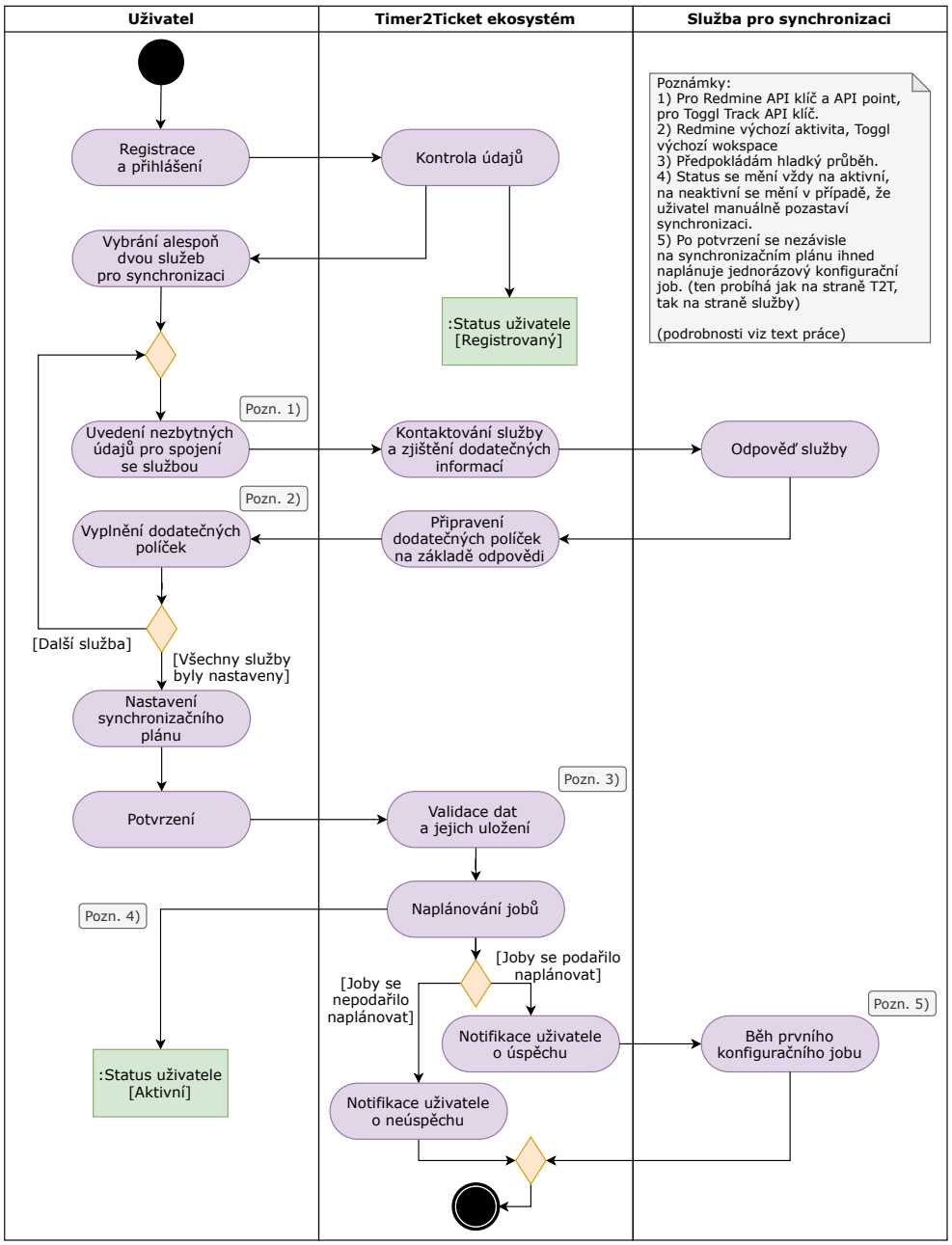
V rámci Redmine je postup shodný, jako doposud. Uživatelé vyberou projekt či rovnou požadavek, stisknou tlačítko „Přidat čas“ a vyplní povinná pole: hodiny a aktivitu (případně ještě změní datum z aktuálního). Můžou ještě doplnit nepovinný komentář. Posléze již jen stisknou tlačítko „Vytvořit“ a časový záznam je připravený k synchronizaci do Toggl Track.

Celý proces je naznačen v diagramu aktivit na obrázku 2.8 na další straně. V levé části je zobrazen proces u Toggl Track, v pravé pak Redmine. Modrou barvou jsou naznačeny nepovinné aktivity (slouží pro lepší orientaci a vyzdvihnutí, že stačí udělat pouze několik aktivit pro správnou synchronizaci).

Následně, až proběhne nový job synchronizace časových záznamů, se takto připravené záznamy synchronizují s druhou službou (či s dalšími). Chce-li uživatel záznam upravit, může ho editovat v libovolné službě. T2T zajistí propagaci změn napříč všemi službami (v případě více změn vyhrává ta poslední provedená). Pro smazání záznamu ze všech služeb je nutné ho odstranit z původní služby – byl-li vytvořen v Redmine, musí se akce provést tam. Tento mechanizmus je navržen proto, aby nedošlo k nechtěnému smazání z – pro uživatele – klíčové služby. Zmíněné chování se samozřejmě může v budoucnu upravit (např. na základě zpětné vazby) tak, aby se záznam odstranil kvůli podnětu z jakékoli služby.

Celkově tedy uživatel nemusí provádět nad rámec základních aktivit o moc více oproti běžnému stavu, kdyby využíval třeba jen jednu službu. Pokud by však měl používat dvě a více služeb bez T2T, počet nutných kroků by rázem stoupl. Nehledě na to, že většinou se jedná o rutinní přepisování. Přenechal-li

[9] Pro jednoduchost. Ve skutečnosti časomíru lze spustit kdykoliv v průběhu procesu, případně zadat datum a čas manuálně. Pořadí uvedených kroků nemusí být nutně dané, takto jsou voleny pro ilustraci.



Obrázek 2.7: Diagram aktivit: proces nastavení konfigurace Timer2Ticket (vytvořeno pomocí [5]), notace UML

by synchronizaci na Timer2Ticket, ušetřil by si spoustu práce a nepřineslo by mu to ani výrazný počet kroků navíc (jedná se o skutečně nutné kroky, také jsou často nepovinné). Ilustruje to také obrázek 2.1 v sekci 2.1 výše.

2.8 Návrh datové struktury

Pro uchovávání stavu a dat jsem se rozhodl využít nerelační dokumentovou databázi MongoDB [15]. Hlavními důvody byla jednoduchost při návrhu, flexibilita samotné databáze, která umožňuje velice dynamicky vyvíjet bez nutnosti předělávat schéma ad. Další výhodou je to, že databáze nativně pracuje s dokumenty (přímo JSON, resp. BSON[10]), které se převádějí do JavaScript objektů instantně bez nutnosti ORM[11] vrstvy. Navíc je MongoDB již poměrně zažitá a oblíbená mezi uživateli. Podle oficiálních stránek byla databázová platforma stažena více než 155 milionkrát s přes jedním milionem registrací přes univerzitu [16].

Z návrhu celého systému se jeví jako nejlepší mít tři kolekce dokumentů. Jednou bude kolekce uživatelů, kde bude dostupná i preference nastavení, ale i mapování jednotlivých objektů služeb. Protože při práci s mapováním je nutné mít k dispozici i údaje v konfiguraci uživatele, nedává příliš velký smysl tyto kolekce oddělit. Dále, množství mapovacích objektů nebude nikdy tak veliké (stovky, nižší tisíce), tudíž i otázku efektivity zde není třeba řešit.

Čeho by ale teoreticky mohlo být více je objektů pro mapování časových záznamů. Jeví se tedy rozumné je od uživatele oddělit. Nicméně každý takový objekt musí odkazovat na svého uživatele pomocí jeho identifikátoru.

Později v průběhu testování po zpětné vazbě uživatelů jsem navrhl ještě třetí dokument, a to objekt pro logování jobů. Každý job o sobě otiskne informace do databáze, včetně časových značek. Využiji ho při zobrazení historie jobů uživateli.

Celkově tedy budou existovat tři druhy databázových dokumentů:

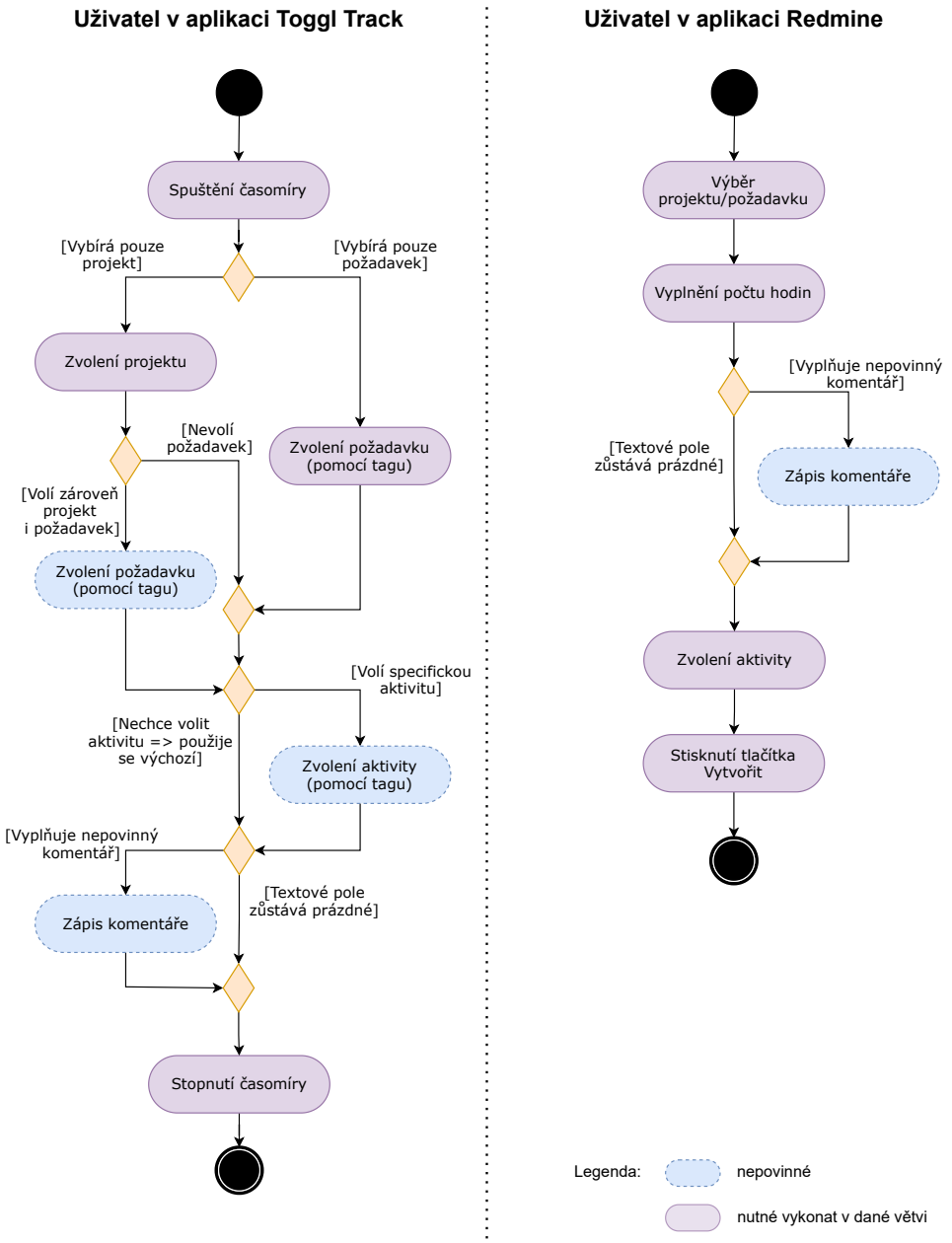
- uživatel (*User*),
- synchronizační objekt časového záznamu (*TimeEntrySyncedObject*),
- logovací záznam o jobu (*JobLog*).

Uživatel bude mít pod sebou nejen uživatelské jméno (*username*), heš hesla (*passwordHash*), datum a čas registrace (*registrated*), jeho stav (*status*)[12], ale i definici synchronizačních jobů (*SyncJobDefinition*), definici služeb (*serviceDefinitions*) a pole mapování (*Mapping*).

[10] JavaScript Object Notation a Binary JSON – formát textových dokumentů vhodný jak pro lidské čtení, tak pro strojové zpracování.

[11] Object-Relational Mapping – vrstva kódu, která převádí relační data na objekty v daném programovacím prostředí. Často zajišťuje externí knihovna.

[12] Rozlišuji tři základní stavy: registrovaný, aktivní, neaktivní – více viz API 3.1.1 dále.



Obrázek 2.8: Diagramy aktivit: procesy vytvoření časového záznamu tak, aby se korektně synchronizoval pomocí T2T v aplikacích Toggl Track a Redmine (vytvořeno pomocí [5]), notace UML

Definice synchronizačního jobu by měla obsahovat synchronizační plán (*schedule*) a pak časovou značku o posledním úšpěšně provedeném jobu (*lastSuccessfullyDone*). Uživatel může nastavit dvě definice jobů: konfiguračního (*config*) a jobu pro synchronizaci časových záznamů (*timeEntry*) – více viz 2.5. Uklízecí job je shodný pro všechny uživatele, tudíž není potřeba ho zavádět do databáze.

Definice služby obsahuje nutné objekty pro spojení se s danou službou: API klíč (*apiKey*), ukazatel, zda se jedná o službu primární (*isPrimary*). V konfiguraci služby (*serviceConfig*) je uložen identifikátor daného uživatele služby (*userId*), ale také informace nutné pro správné spojení s danou službou: pro Redmine je to adresa ke službě uživatele (*apiPoint*) a výchozí aktivita časového záznamu (*defaultTimeEntryActivity*; její identifikátor a název). Pro Toggl Track je nutné znát pouze výchozí pracovní místo (*workspace*)[13]. Tyto dva objekty uživatel nastavuje v průběhu konfigurace T2T po registraci (sekce 2.7).

Kromě již zmíněných obsahuje uživatel ještě pole mapování (*mappings*). Tento objekt slouží k identifikaci reálného objektu a jeho spojení s definovanou službou (pole *MappingsObject*). Podrobněji viz sekce 2.6.

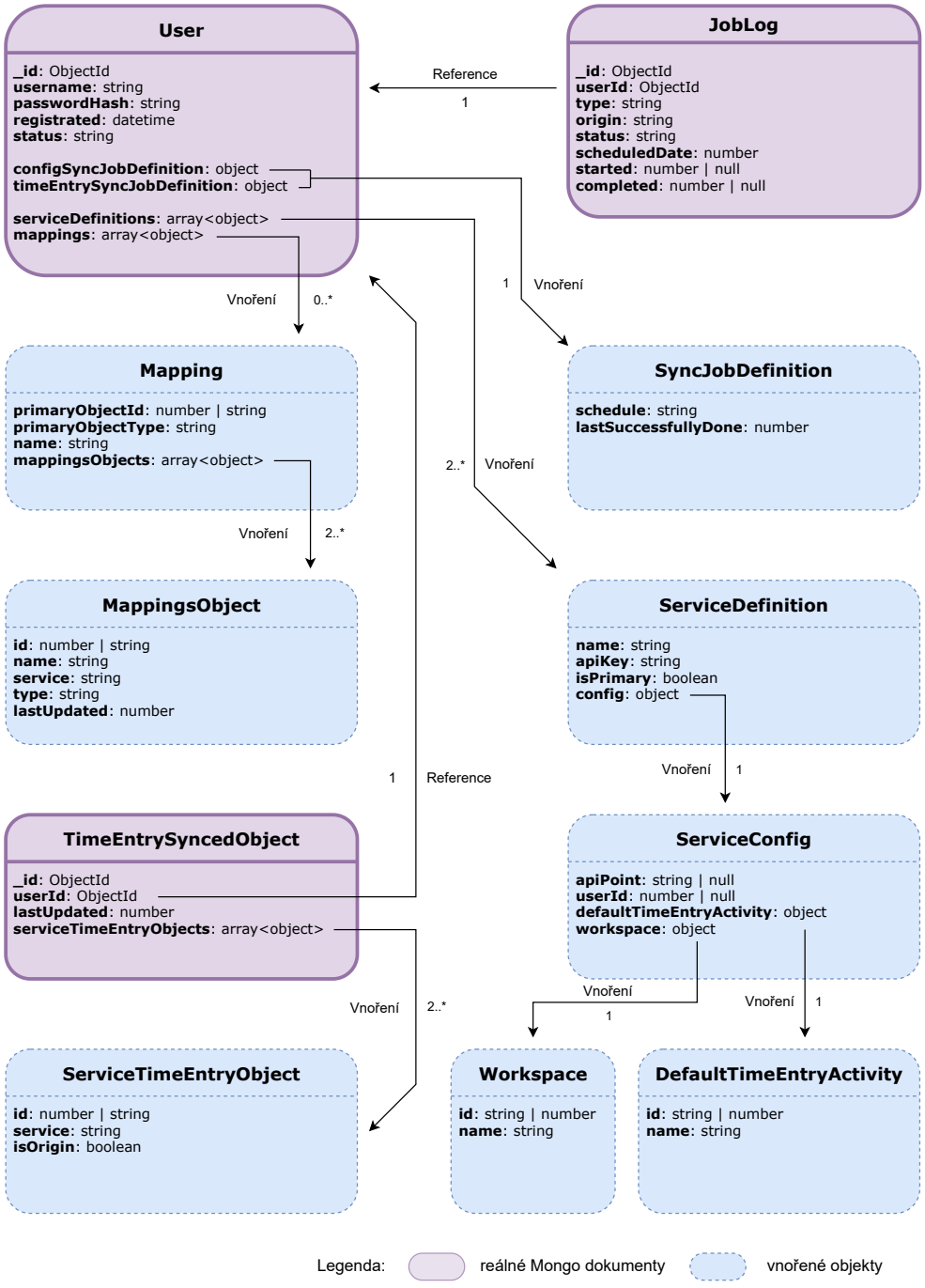
Druhým dostupným dokumentem je synchronizační objekt časového záznamu (TESO, více také viz sekce o synchronizaci 2.6). Ten slouží k obdobnému účelu jako mapování, jen s tím rozdílem, že identifikuje časové záznamy. Musí také obsahovat časovou značku poslední úpravy pro řešení aktualizovaných záznamů a případných konfliktů mezi službami. Každé TESO obsahuje pole objektů časového záznamu služby (STEO). To obsahuje identifikátor záznamu v dané službě a zda se jedná o původní záznam (tedy ten, na jehož základě se synchronizoval do ostatních služeb).

Posledním dokumentem je *JobLog*. Každý si uchovává identifikátor uživatele, aby se mohl k němu přiřadit. Dostupný bude jeho typ (*type*) – jedná se o znak konfiguračního jobu či jobu na synchronizaci časových záznamů. Dále pak původ (*origin*; automaticky podle plánu, případně manuálně spuštěný uživatelem), status (naplánovaný, běžící, úspěšný, neúspěšný), ale také časová značka naplánování (*scheduledDate*), začátek plnění jobu (*started*) i značku dokončení (*completed*). Status a časové značky se budou měnit a plnit na základě průběhu jobu.

Pro ilustraci jsem vytvořil obrázek 2.9, na kterém je znázorněno schéma databáze (ačkoliv je MongoDB schema-free[14], lze aktuální stav takto vyznačit – nedržel jsem se u něj nějaké notace, MongoDB ani nemá speciální notaci pro zanesení schématu). Fialovou barvou jsou vyznačeny reálné dokumenty v DB, modrou jsou odlišeny vnořené objekty, které jsem do schématu zakreslil odděleně pro lepší orientaci. Šipkami jsou naznačeny vztahy mezi objekty. Vícenásobný vztah značí to, že objekt se vyskytuje v poli. Identifikátor zna-

[13] Podrobněji o objektech jednotlivých služeb viz sekce Analýza služeb 2.2

[14] Nezná definici schématu. Pokud to tedy dovolíme, můžeme ukládat rozdílné objekty do stejné kolekce. V realitě je ale vhodné nějaké alespoň základní schéma mít definované.



Obrázek 2.9: Znázornění datového schématu pro nerelační databázi Mongo (vytvořeno pomocí [5]), bez notace

čený *_id* je vlastní MongoDB. Ostatní identifikátory bez podtržítka značí odkazy reálných objektů v dané službě. Takže třeba *id* v *MappingsObject* je vlastně identifikátor reálného objektu v reálné službě (např. id projektu v Redmine). Dost často takový identifikátor může být typu *číslo* nebo *řetězec*. Tuto neurčitost jsem zachoval z důvodu obecnosti systému. Může se stát, že jedna synchronizační služba bude pracovat s řetězci, druhá s čísly. Pro mou aplikaci tato neurčitost nezavádí žádné problémy. Kde je potřeba, jednoduše se atribut převede na řetězec (pokud je atribut ve formě řetězce, nestane se nic).

2.9 Bezpečnost

Krátce se zmíním o bezpečnosti aplikace. Protože bude aplikace vystavená veřejně, je nutné zajistit bezpečí uživatelů. Při registraci je nutné heslo zahešovat a do databáze uložit v této formě (nejlépe [17] pomocí algoritmu BCrypt [18]). Jinak hrozí, že kdokoliv s přístupem do databáze zjistí hesla uživatelů.

Po korektním přihlášení se uživateli vygeneruje JWT[15], kterým se následně prokazuje do API (resp. to za něj dělá klientská webová aplikace). Tím zajistím, že uživatel má právo přistupovat do nepřihlášeným uživatelům nepřístupným částím. V JWT je také uložen jeho identifikátor, tudíž je zkontrolováno, zda manipuluje pouze se svými prostředky, probíhá tedy autorizace. Pro správný běh aplikace je také nutné validovat data příchozí z klienta. Vše se děje proto, že klientská aplikace se dá jakkoliv upravit a obejít tak její validace. Je žádoucí tedy všechny validační úkony provést ještě jednou v serverové části.

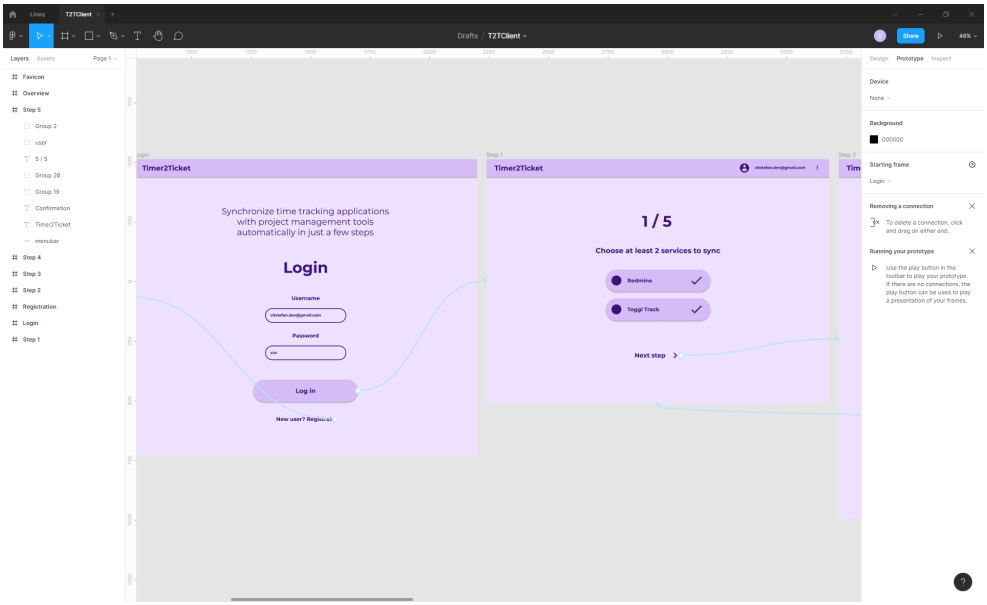
T2T Core komunikuje pouze s API, tudíž zde žádná autentizace a autorizace potřeba dělat není. Jiné služby se na ni připojit také nebudou moci. Veškerá komunikace z vnějšku do systému (kromě odpovědí na dotazy směrem k synchronizačním službám) bude probíhat přes klienta na API (nebo pouze REST dotazy na API).

2.10 Návrh obrazovek webové aplikace

Ještě před implementací klientské části jsem navrhl obrazovky ve specializované aplikaci Figma [19]. Tento nástroj nejen pro designery je vynikajícím pomocníkem při tvorbě uživatelského rozhraní. Dovoluje velmi rychle připravit očekávaný vzhled aplikace, aniž by programátor musel řešit HTML, data apod. Stačí jednoduše připravit hlavní obrazovky, dle nichž pak v průběhu implementace bude postupovat. Tento přístup se mi v minulosti již mnohokrát osvědčil.

V nástroji lze také prototypovat, tedy přidávat přechody mezi jednotlivými obrazovkami. Posléze lze celý průchod spustit přímo v dané aplikaci

[15] JSON Web Token, textový řetězec, ve kterém je mj. skryta informace o uživateli.



Obrázek 2.10: Snímek z nástroje Figma, ukázka několika obrazovek včetně šipek značících existující přechod v rámci prototypování

a pomocí standardních nástrojů (myš, dotyky na mobilním zařízení) se mezi pohledy pohybovat. Navíc umožňuje designerům přidávat komentáře, které se uplatní zejména tehdy, kdy spolupracujeme s více lidmi. Pro ilustraci přikládám snímek 2.10 z nástroje Figma, kde je naznačeno i prototypování (průchod se spustí pomocí tlačítka symbolizovaného šipkou v pravém horním rohu). Všechny exportované obrazovky ve formátu *pdf* a *png* lze nalézt v přiloženém paměťovém médiu. Rovněž tak je tam přístupný export kopie Figma projektu.

Cílil jsem na minimalistický vzhled a prvky, které by uživatele neměly nijak rušit. Jako primární barvy jsem zvolil pastelově fialovou a její odstíny, neboť mi přišlo, že se ve finále hodila k projektu nejvíce. Pracoval jsem se sloupcovitým rozložením, aby byla stránka nativně vhodná jak pro širší obrazovky (laptop, monitor), tak i pro mobilní zařízení. Nejprve uživatel uvidí tradiční přihlašovací obrazovku, odkud se dostane i na registraci pro nové uživatele. Dále ho aplikace provede konfiguračními kroky a nakonec přesměruje na přehled. Na přehled budou směřováni uživatelé jako první v případě, že konfiguraci mají již úspěšně provedenou. Posloupnost kroků a co se děje v jednotlivých částech aplikace detailněji popisuje sekce 2.7.1 výše.

Uživatel by měl vědět, v jakém stavu se nachází, bude vhodné tedy mu ukázat počet kroků, které absolvoval a které ještě musí navštívit pro správnou konfiguraci. Dále se musí posouvat vpřed a případně vzad, k tomu budou sloužit tlačítka pro posun v dolní části obrazovky. Umístění je záměrné, uživatel přirozeně postupuje zeshora dolů. Nejprve tedy zvládne povinné aktivity v daném kroku a následně přejde na další krok. V průběhu budou muset být

vložená data také kontrolována. Některá bude moci uživatel vyplnit až na základě získaných dat ze synchronizační služby (tedy po vyplnění a validaci některých polí). Tento a další problémy spojené s klientem popíšu ale až dále v kapitole 3 Realizace a sekci 3.2 Frontend.

2.11 Použité technologie

Na obou stranách technologií jsem využil služeb JavaScriptu, respektive TypeScriptu [20]. Psát vše ve stejném jazyce je velice pohodlné. Obojí si také nativně rozumí s JSON dokumenty, které konzumuji z Mongo databáze a služeb třetích stran.

Kromě toho mi též umožní sdílet kusy kódu – pomocí snadné kopie bez nutnosti převést kód. TypeScript dovoluje i přímo sdílet kusy kódu jako knihovny, ale to mi koncepčně nesedí do principu oddělenosti všech tří implementačních části (Core, API a klient). Kromě toho pouze modely jsou si podobné – ale ne stejné – pro všechny části. Tuto možnost jsem tudíž nevyužil. Mohu tak kdykoliv jejich kód oddělit a pracovat nad nimi kompletně nezávisle.

Asynchronní kód většinou zapisuji pomocí techniky *async – await*. Ve své podstatě se jedná o emulaci synchronního kódu tam, kde potřebuji čekat na výsledek [21, s. 171]. Vyhnu se tím tak často nepřehledému volání vnořených *callback* funkcí. Všechny mé metody pracující asynchronně tak vrací hodnotu obalenou *Promise* objektem.

2.11.1 Backend

Jako hlavní vývojové prostředí pro backend jsem využil NodeJS [22] a framework Express [23], tedy psaní serverového kódu v JavaScriptu. Vzhledem k tomu, že se v mém případě jedná o webovou službu, je NodeJS vhodné použít. Ostatně na to bylo původně vyvinuto [21, s. 147]. Vybudovat REST API bez frameworku samozřejmě v NodeJS lze, přišel bych ale o některé výhody, které Express nabízí (resp. pokud bych je potřeboval, musel bych je implementovat sám). Snadno tak mohu definovat např. routování [21, s. 148].

Kód jsem však nepsal v čistém JavaScriptu, ale v TypeScriptu, což je ve zkratce typovaný JavaScript. Jisté vodítko, že jeho vývoj postupuje správnou cestou může být to, že novější specifikace JavaScriptu přebírají spoustu výhod TypeScriptu. Hlavní výhodou je typované prostředí, které je ve finále rychlejší na vývoj, ale také bezpečnější. Ostatně [24] zjišťuje, že rozdíl mezi psaním čistým JS a TS lze poznat. Tento článek navíc bral v úvahu hlavně efektivitu psaní díky našeptávání ze strany editoru kódu. V rámci psaní se také odhalí spousta chyb, které by se jinak mohly projevit až po nasazení (to odhalí hlavně tzv. „lintery“[16]).

[16] Lintery jsou programy pomáhající při vývoji kódu. Existuje jich spousta a jsou specifické pro daný programovací jazyk. Mohou odhalit např. to, že objekt, na který se dotazuju nemusí být definovaný a je třeba si to ohlídat.

Na zasílání REST požadavků jsem použil knihovnu SuperAgent [25]. Pro validaci objektů, které přicházejí od klienta, se mi hodí knihovna ClassValidation [26]. Pro plánování jobů využívám služby knihovny Node Cron [27], která v podstatě imituje Cron plánovač na Unix systémech. Více pak v kapitole 3 o realizaci.

2.11.2 Frontend

Zde jsem se rozhodl pro JavaScriptový framework Angular [28], který je rovněž nativně v TypeScriptu. Použití frameworku usnadní implementaci tím, že spoustu běžných problémů má vyřešených – obsahuje např. nativní router.

Pro sdílení dat uvnitř klientské aplikace jsem využil RxJS Observables [29]. Jejich zabudovaný systém subskripce mi umožní měnit data a pohledy, vždy na základě změny dat automaticky. Protože Angular využívá obousměrně aktualizované vhledy na data, jsou pro něj Observables jako dělané. Pro některé prvky jsem využil modul Material, který je dostupný jako knihovna přímo pro Angular [30]. Podrobněji v sekci 3.2 věnované realizaci frontendu dále.

3 Realizace

V této kapitole popíšu všechny tři implementované části Timer2Ticket (databázová část je přiblížena v sekci 2.8 výše). Na závěr ještě shrnu samotné nasazení a navrhnu, jak do systému integrovat novou synchronizační službu.

T2T systém je rozdělený na služby tak, jak jsem zmiňoval v sekci 2.5 Architektura (rovněž tam je také k dispozici zjednodušené schéma na obrázku 2.5). Skládá se z API, Core (obě patří pod backend) a klienta (frontend).

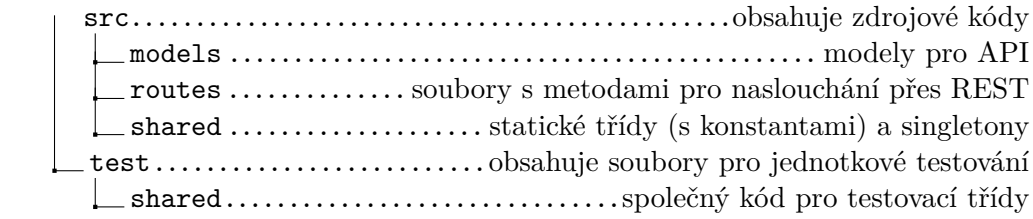
Každá ze tří částí obsahuje modely, které interpretují objekty načtené z databáze. Jsou si často dost podobné. Jednotlivé části ale mohou definovat ještě další, které u jiné nemusí nutně být potřeba (např. pouze Core pracuje se synchronizačními objekty časového záznamu – TESO, nebo naopak nepotřebuje objekt, který si posílají API a klient při registraci uživatele). Na straně API jsou pak některé modely anotovány validačními značkami. Více dále v jednotlivých sekcích.

3.1 Backend

Backend jsem rozdělil na dvě samostatné služby: API a Core. S API komunikuje uživatel prostřednictvím webového klienta (resp. s ním může komunikovat kdokoliv přes REST rozhraní). Umožňuje mu se registrovat, přihlásit, ale také pracovat s joby. Core zajišťuje samotnou synchronizaci mezi službami zvolenými uživatelem.

Společným prvkem backendu je databázová služba ve formě třídy *DatabaseService*. Pomocí ní probíhá komunikace s databází. Vždy při startu aplikace je nutné tuto službu inicializovat. Služba je navržená jako singleton[17], inicializuje se voláním metody *init()*, která je asynchronní (proto její tělo nemůže být v konstruktoru, který asynchronní být nesmí). V této metodě probíhá navázání spojení se samotnou databází. Kromě toho si instance uloží do privátních proměnných kolekce, se kterými bude pracovat (kolekce uživatelů, logů jobů, v případě Core ještě kolekce synchronizačních objektů časového záznamu – TESO). Posléze umožňuje s danými entitami pracovat. Pro práci např. s uživatelem poskytuje následující: výběr uživatele dle identifikátoru/jména, vytvoření uživatele (registrace) a jeho editace.

[17]Singleton je návrhový vzor, který zajišťuje, že existuje pouze jedna instance třídy, která je sdílená.



Obrázek 3.1: Adresářová struktura API části T2T

Databázová služba neprovádí žádnou validaci ani autorizaci. Objekty vstupující do třídy musí být ve validním stavu (API objekty přijaté od uživatele validuje, Core objekty od uživatele nepřebírá, jedná se tedy o interní validaci v rámci kódu). Toto je jediné[18] místo, které je závislé na databázovém stroji (v mém případě MongoDB). V případě, že by se v budoucnu uvažovalo o změně databáze, musela by se přizpůsobit právě tato třída.

3.1.1 API

API je část T2T, která zajišťuje komunikaci mezi uživatelem (klientem) a databází, uživatelem a Core, ale také uživatelem a službami pro synchronizaci. Jejím jádrem je REST API, na kterém přijímá požadavky. Aplikace je založená na NodeJS a napsaná v TypeScriptu. Rozdělení zdrojových souborů je naznačeno na obrázku 3.1. Nejdůležitějším podadresářem je *routes* obsahující definici REST metod.

Mnou navržené rozhraní je založeno na frameworku Express. Aby nemusely být všechny metody čekající na REST požadavek v jedné metodě, zařadil jsem je do logických celků a ty pak umístil do samostatných souborů. V řídícím souboru *app.ts* je pak registruji. Inspirací pro tento postup mi byl oficiální tutoriál dostupný v dokumentaci Expressu [31]. Do logických celků patří:

Autentizace autentizuje uživatele.

Registrace registruje uživatele.

Uživatelé pracuje s objektem uživatel.

Joby slouží jako most mezi Core a klientem pro příkazy ohledně jobů.

Konfigurace synchronizovaných služeb místo klienta posílá požadavky na synchronizační služby.

Logy jobů umožňuje získat logy jobů daného uživatele.

[18] V modelech existují ještě objekty – např. uživatel – jejichž možný typ identifikátoru je *ObjectId*, což je datový typ specifický pro Mongo. Jedná se však o pouhý řetězec, který je obohacen určitými metodami. Bez této úpravy bych se připravil o některé výhody při práci s danými objekty právě v databázové službě. Při změně databáze by zde nemuselo dojít k žádné změně, identifikátor by se choval jako klasický řetězec.

Na začátku každého celku může být připraven společný kód pro všechny definované metody. Kromě případů Autentizace a Registrace se takto může vložit validace JWT, který je zaslán od klienta. Samotná autorizace uživatele pak už probíhá uvnitř jednotlivých metod. Části Autentizace a Registrace disponují pouze jednou naslouchávající metodou pro POST požadavky. V první se vygeneruje heš hesla a vytvoří se základní kostra pro uživatele, který se následně vloží do databáze. Ve druhé je uživatel ověřen na základě dodaného hesla, také mu je vytvořen a poslán vygenerovaný JWT.

S uživatelem pracuje část Uživatelé. Hlavní metodou je PUT, která provádí modifikaci této entity. Kromě klasické autorizace je zde také důležitá validace. Ta je provedena za pomocí knihovny ClassValidation [26]. Realizuje se pomocí anotací v definici třídy. U každého atributu lze uvést sadu validátorů, které musí být pro úspěšnou validaci splněny. Takto mohu snadno zkontrolovat, zda je nějaká hodnota skutečně datum, zda má textový řetězec požadovanou délku, nebo třeba zda nabývá hodnoty z mnou určeného výčtu. Tam, kde potřebuji objekt zkontrolovat, už jen zavolám metodu *validate*, které předám daný objekt jako parametr. Metoda vrací výsledky validace, resp. pole chybových objektů. Je-li toto pole prázdné, validace proběhla v pořádku. Tímto způsobem je nutné zvalidovat zejména ty atributy, které se touto cestou editují (přes PUT je možné změnit jen menší část objektu, kterou uživatel nastavuje v průběhu konfigurace na klientu).

Na tomto místě je vhodné zmínit také možné stavy uživatele. Abych odlišil, v jakém stavu se aktuálně nachází, definoval jsem tři následující stavy (v databázi uložené pod atributem *status*):

- registrovaný (*registrated*),
- neaktivní (*inactive*),
- aktivní (*active*).

Registrovaným se uživatel stává nepřekvapivě ve chvíli, dojde-li k jeho registraci. Tento stav se změní v případě, že projde celým procesem konfigurace T2T a odešle jeho potvrzení (v posledním kroku při konfiguraci na klientu). Zde je mimochodem objekt uživatele poprvé editován přes výše zmíněnou PUT metodu a zde je také zajištěno, že stav se mění z registrovaného na neaktivního. Neaktivní uživatel je registrovaný, ale nemá zapnutou synchronizaci. Aktivním uživatelem se stává v případě, že:

a) si sám zapne synchronizaci (poté co ji manuálně vypnul),

b) či mu je automaticky zapnuta po procesu registrace.

Z posloupnosti událostí je pochopitelné, proč mu hned při prvním PUT není nastaven režim aktivní: na klientu je totiž zaslán požadavek na editaci a až po správném uložení upraveného uživatele do databáze je odeslán další požadavek na zapnutí synchronizace. Mohlo by se totiž stát, že se uživatel uloží, ale jeho

joby nebudou naplánovány třeba z důvodu nereagujícího Core. Projeví se to tak, že je uživatel přesměrován na obrazovku přehledu s notifikací, která mu říká, že jeho joby se nepodařilo naplánovat a je mu doporučeno je manuálně spustit později (více dále v sekci Frontend 3.2). Přechod mezi stavy aktivní a neaktivní řeší API na základě požadavku uživatele a odpovědi od Core při pokusu o zapnutí/vypnutí synchronizace

Celek pro joby v podstatě jen přeposílá požadavky na Core. Zajišťuje ale důležitou autorizaci, o kterou se následně Core tak nemusí starat. Touto cestou se tedy vyřizuje pozastavení/spuštění synchronizace, zjištění stavu synchronizace, ale také manuální naplánování jobu.

Dalším celkem je Konfigurace synchronizovaných služeb. Ten je využíván v průběhu konfigurace uživatele na klientu. Řeší autorizaci, validaci zaslaných údajů a přeposlání požadavků na služby pro synchronizaci. Např. u Redmine potřebuji načíst a poslat na klienta pole aktivit časových záznamů, aby si mohl vybrat výchozí. Výhodou je, že tím i ověřím validitu předaného API klíče a API přístupového bodu. Navíc mohu takto bez vědomí uživatele (aniž by musel tuto hodnotu někde vyplňovat) a klienta zaslat požadavek pro uživatelův identifikátor v Redmine, který je potřeba při pozdějším zasílání požadavků v rámci jobů. Protože komunikace se službou probíhá ze serveru a ne z klienta, nemusím tak ani řešit CORS[19].

Poslední celek umožňuje uživateli zkontrolovat logy jobů. Ty vybírá z databáze, kam je průběžně vkládá Core. Posílá vždy ale maximálně 100 nejnovějších (a také maximálně 90 dnů staré, ostatní totiž Core maže, více další podsekce 3.1.2).

3.1.2 Core

Core je opět postaveno na NodeJS a TypeScriptu. Jeho hlavní rolí je obstarávání fronty jobů, do kterých jsou podle plánování vkládány požadavky na synchronizaci konfiguračních objektů (projekty, požadavky…) a časových záznamů. Navíc poslouchá na svém REST rozhraní a přijímá přes něj požadavky od webového klienta (přes API). Mezi požadavky patří manuální vytvoření jobů, spuštění opětovné synchronizace, pozastavení synchronizace daného uživatele, ale také dotaz na stav synchronizace. Protože je Core ukryté před vnějším světem a jediná možná komunikace probíhá přes veřejně dostupné API, nemusí zde být prováděná autorizace. Uživatel a jeho práva jsou již ověřena právě na straně API.

Zdrojové kódy jsou opět dostupné ve složce *src*, na obrázku 3.2 je naznačena adresářová struktura. V adresáři *jobs* se nachází kód pro samotné joby, v *synced_services* pak kód pracující se službami pro synchronizaci. Kromě sdílených modelů Core obsahuje ještě model pro synchronizační objekt časového

[19]CORS – Cross-Origin Resource Sharing. Jedná se o bezpečnostní mechanizmus implementovaný v prohlížečích, který omezuje posílání požadavků na servery jiného původu, pokud to server explicitně nepovolí. Více např. zde [32].

```
src ..... obsahuje zdrojové kódy
├── jobs ..... obsahuje definici jobů vč. abstraktní třídy
├── models ..... modely pro Core
├── shared ..... statické třídy (s konstantami) a singletony
└── synced_services ..... specifický kód pro synchronizovanou službu
    ├── redmine ..... specifický kód pro komunikaci se systémem Redmine
    └── toggl ..... specifický kód pro komunikaci s aplikací Toggl Track
```

Obrázek 3.2: Adresářová struktura Core části T2T

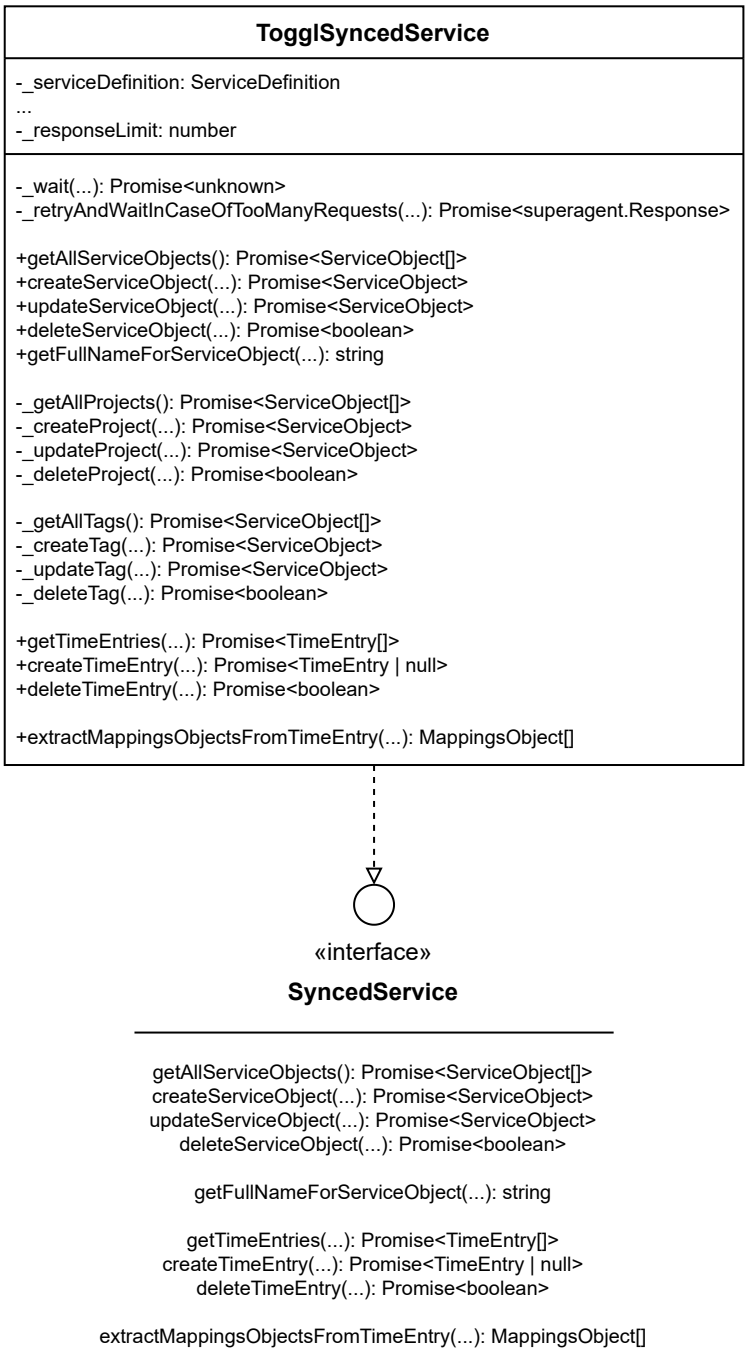
záznamu (TESO) vyskytující se v databázi. S ním jako jediná část T2T pracuje právě Core. Dále také obsahuje speciální pomocné entity, které nemají svůj původ v databázi. Jedná se o objekty služby (*ServiceObject*) a o pomocné objekty reprezentující časové záznamy jednotlivých služeb pro synchronizaci. Tyto objekty jsou extrahovány z REST odpovědi dané služby. Více v následující podsekci.

Služby pro synchronizaci

Joby prováděné Core jsou založené na obecném principu. Samotné joby se musí chovat nezávisle na službě (Redmine, Toggl Track…), u které se job zrovna provádí. Samozřejmě je ale každá služba pro synchronizaci jiná a vyžaduje specifický přístup. Proto jsem navrhl rozhraní (*interface*), které definuje chování a metody každé jednotlivé synchronizační služby. Metody tohoto rozhraní budou volány při provádění jobů a na základě aktivní služby bude volán specifický kód. Co musí každá služba splňovat je k vidění v diagramu tříd na obrázku 3.3 na další straně. Musí umět pracovat se všemi konfiguračními objekty (projekty, požadavky, tagy…) na základě uvedeného typu – načíst všechny takové objekty, vytvořit nový objekt, upravit ho a smazat. Samozřejmě také umožňuje pracovat s časovými záznamy – všechny načíst, vytvořit nový, smazat existující (úprava je prováděna z praktických důvodů smazáním a následným vytvořením, vychází to z posloupnosti scénářů 2.6 a spojení části kroků; detailněji přímo v kódu).

Správné přiřazení specifické třídy zajišťuje pomocná třída *SyncedServiceCreator*, která na základě dodané definice služby (vnořený objekt uživatele) vybere konkrétní implementaci rozhraní.

Konkrétní metody volají koncové body dané služby a vyžadují data, případně chtějí data měnit. Musel jsem řešit stránkování – v obou službách jsou data dávkovaná, tedy na jeden dotaz jich přijde omezené množství. U Redmine limit nastavuje uživatel v parametrech požadavku (maximálně 100). V případě Toggl Track je tato hodnota fixně nastavená na 50. Pokud vyžaduji např. všechny časové záznamy uživatele, musím s tím počítat. Proto odesílám opakované dotazy s posunutým indexem (stránky). To jsem vyřešil pomocí jednoduché úpravy, kterou demonstruji na ukázce výňatku kódu 3.1 dále. Ukázka je



Obrázek 3.3: Diagram tříd: rozhraní pro synchronizační službu (*SyncedService*) a příklad implementace třídou *TogglSyncedService* (vytvořeno pomocí [5]), notace UML

```
// ...

let totalCount = 0;

const queryParams = {
  // ...
  page: 0,
};

// paginate
do {
  // next page (first page is 1)
  queryParams.page++;

  // ...
  // send HTTP GET request
  // load data from the response
  // ...

  // extract total data count from the response
  totalCount = response.body?.total_count;
  // (this._responseLimit = 50 for Toggl Track)
} while (queryParams.page * this._responseLimit < totalCount);

// all data loaded

// ...
```

Výpis kódu 3.1: Výňatek z metody pro načtení všech časových záznamů třídy *TogglTrackSyncedService* sloužící jako ukázka řešení stránkování

vybrána z kódu pro Toggl Track, s mírnými obměnami je mechanizmus použit i u Redmine a dal by se použít u jakékoli jiné služby.

Dále, služby mohou z důvodu zamezení přetížení svých zdrojů povolovat jen určitý počet požadavků z jedné adresy. Po překročení tohoto limitu pak posílá odpověď např. s chybovým návratovým kódem 429. Vždy je třeba si ověřit u dané služby, jak na přetížení reaguje a jakým mechanizmem tento problém vyřešit. Jednou z možností je vždy po každém požadavku na službu čekat fixní dobu. Je to přímočaré a jednoduché řešení, nicméně má za následek to, že se provedení protáhne i v případě, že se volá jen pár požadavků.

Já tak zvolil (u Toggl Track) jinou strategii: každý požadavek se může opakovat. Nový pokus proběhne tehdy, vrátí-li služba specifický chybový kód

```
// (note: code example is simplified)
// every 10 seconds call given function
cron.schedule('*/10 * * * * *', () => {
  // do all the jobs in the queue
  while (!jobQueue.isEmpty()) {
    const job = jobQueue.dequeue();
    job.start();
  }
});
```

Výpis kódu 3.2: Plánování jobů, zjednodušené

(může to být výhodné i třeba v případě nestabilního internetového spojení). Pokud je ale tento kód rovný 429, proces se uspí (na 1500 milisekund). V dokumentaci API Toggl Track [33] je napsané, že vhodná doba čekání je 1 sekunda. Raději jsem zvolil delší interval mezi dalšími požadavky.

Abych nemusel toto volání řešit u každého požadavku zvlášť, implementoval jsem obalovou metodu, která ve svém parametru přebírá volaný požadavek. Tento požadavek pak volá a případně uspí a opakuje. Stejnou taktiku jsem zvolil i u Redmine, ale vzhledem k tomu, že tuto službu každý hostuje sám, její chování v případě zatížení není obecně nijak definované.

Plánování a fronta jobů

Uživatel volí, jak často se budou provádět synchronizační joby. Na pozadí ve skutečnosti vybírá Cron vzor. To je textový řetězec, který následně vhodná knihovna správně interpretuje a vytvoří dle toho intervalové volání procedur. Pro NodeJS existuje knihovna Node Cron [27] vytvářející vhodné *setTimeout* a *setInterval* JS funkce (umožňuje mj. i pracovat se sekundami). Ve výpisu kódu 3.2 jsem naznačil, jak plánování funguje. Plán **/10 * * * * ** značí: každých deset sekund, volej předanou funkci. Aplikace se podívá na stav fronty jobů a vyprázdní ji. Reálně je potřeba odchytat chyby a logovat je, to jsem v ukázce pro zjednodušení vypustil.

Joby se do fronty mohou dostat dvěma způsoby. Jsou tam buď přidány uživatelem přes API – při dokončení konfigurace systému, při zapnutí synchronizace, a nebo když jej uživatel naplánuje jednorázově. Druhou možností je, že jsou naplánovány joby automaticky všech aktivních uživatelů při spuštění T2T. Odebrány mohou být pouze na požadavek uživatele a to v případě, že chce synchronizaci vypnout (všechny příkazy se vykonávají přes API a přes klienta).

Ve finále se tedy vyvolávají metody plánované knihovnou Cron. Tyto metody přidají joby do fronty, která je periodicky kontrolována. Zároveň je ale

nutné udržovat informaci o uživatelových plánovaných úkolech. Vede k tomu prostý důvod, plánování jobů je nutné ukončit, když o to uživatel požádá. Tyto údaje pak pro snadné hledání uchovávají mapy *uživatel : plánovaný úkol* (úkolem se rozumí metoda pro vložení jobů do fronty).

Jednotlivé joby

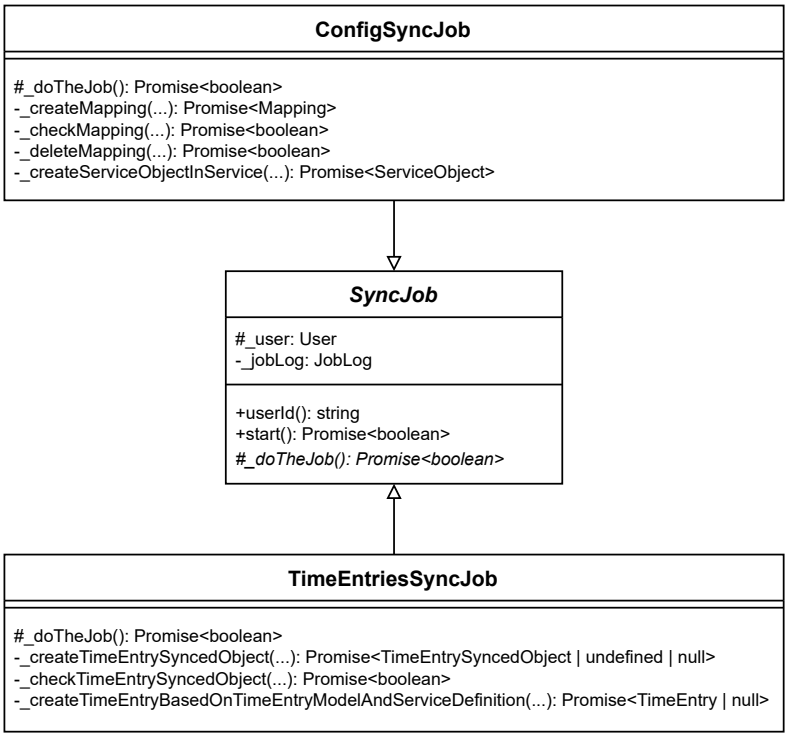
Každý job musí vycházet z abstraktní třídy *SyncJob*. Při návrhu popsaném v sekci 2.6 výše jsem počítal se třemi typy jobů: job synchronizace konfigurace, job synchronizace časových záznamů a úklidový job. Při realizaci jsem si ale prozatím vystačil pouze s prvními dvěma. Úklidový job měl sloužit k odstraňování starých (nepoužívaných…) objektů vzniklé konfiguračním jobem, ale ukázalo se, že třeba Redmine zavřené joby defaultně přestává posílat přes API (pokud o to není přímo požádáno).

Znamená to, že konfigurační job bude staré objekty mazat sám (jelikož nepřijdou od Redmine, bude je požadovat za „smazané“ a odstraní je i z Toggl Track i z mapování). Toto chování bude potřeba dále sledovat a případně job implementovat (stačí jen použít existující abstraktní třídu a plánovat ho společně s ostatními joby). Stejně tak může být T2T rozšířeno o jakýkoliv jiný nový typ jobu. V aktuální verzi systému tedy existují dva typy jobů a oba vychází ze zmíněné abstraktní třídy (schématicky naznačeno na obrázku 3.4 na další straně diagramem tříd).

Abstraktní třída *SyncJob* musí znát uživatele, za kterého job vykonává. Tento parametr je nastavován v konstruktoru. Zároveň obsahuje veřejnou metodu pro získání identifikátoru uživatele. Ten je použit logovací službou. Konkrétní třídy pak musí implementovat abstraktní metodu *_doTheJob()*, která fakticky vykoná potřebnou práci. K tomu si každá z nich zavádí pomocné privátní metody. Řídí se scénáři, které jsou popsány v podsekcích návrhu 2.6 výše. Při provádění joby vytvářejí požadavky na služby pro synchronizaci – např. vytváření časového záznamu ad. Třídy si také definují vlastní pomocné neveřejné třídy. Většinou se jedná o spojení několika objektů dohromady pro snadnější a rychlejší práci.

V průběhu navíc probíhá kontrola, zda nedošlo k nějaké chybě. Na konci se vyhodnotí, které změny se zapíší do databáze, a pokud vše proběhlo v pořádku, nastaví se i časová značka posledního úspěšného provedení jobu.

Uživateli je dostupná nedávná historie logů. Proto je metoda *_doTheJob()* v abstraktní třídě obalená metodou *start()*, která kromě volání již zmíněné také zajišťuje logování jobů. Kód dostupný v přiloženém médiu je bohatě okomentován (vč. vylíčení scénářů zmíněných výše) pro usnadnění pochopení celé logiky.



Obrázek 3.4: Diagram tříd: abstraktní třída (*SyncJob*) a třídy z ní vycházející (vytvořeno pomocí [5]), notace UML

Logování jobů

Každý job, který je vložen do fronty o sobě vytvoří záznam do databáze (objekt *JobLog*). Objekt a jeho jednotlivé atributy jsou popsány v sekci 2.8 o databázové struktuře výše. Na začátku je objekt ve stavu *naplánovaný*. Je-li vybrán z fronty a spuštěn, přepne se do stavu *běžící*. Po skončení se změní na *úspěšný* v případě, že skončil bez chyby, jinak bude ve stavu *neúspěšný*. Při každé změně stavu se do něj vtiskne i aktuální čas.

Tyto objekty jsou dostupné pro přehled uživatele. Vzhledem k tomu, že jejich počet se může i po krátkém časovém úseku dost zvýšit, je potřeba je čas od času pročistit. Jedním řešením by bylo přesouvat je např. do jiné databáze, pro tuto chvíli se ale promazávají. Core jednou měsíčně spouští proces, který maže všechny záznamy starší než 90 dnů.

3.2 Frontend

Frontend část je v rámci systému T2T jediná. Jedná se o webovou aplikaci ve formě SPA[20]. Skládá se tedy z jednoho HTML souboru, který obsluhuje a mění JS kód na základě chování a navigace uživatele – tento způsob je nativní frameworku Angular, který jsem pro implementaci využil. Výhodou tohoto způsobu je fakt, že při přechodu na jinou stránku[21] se nemusí načítat kód HTML a další podpůrné soubory znovu. Vše je načteno na začátku při příchodu do aplikace.

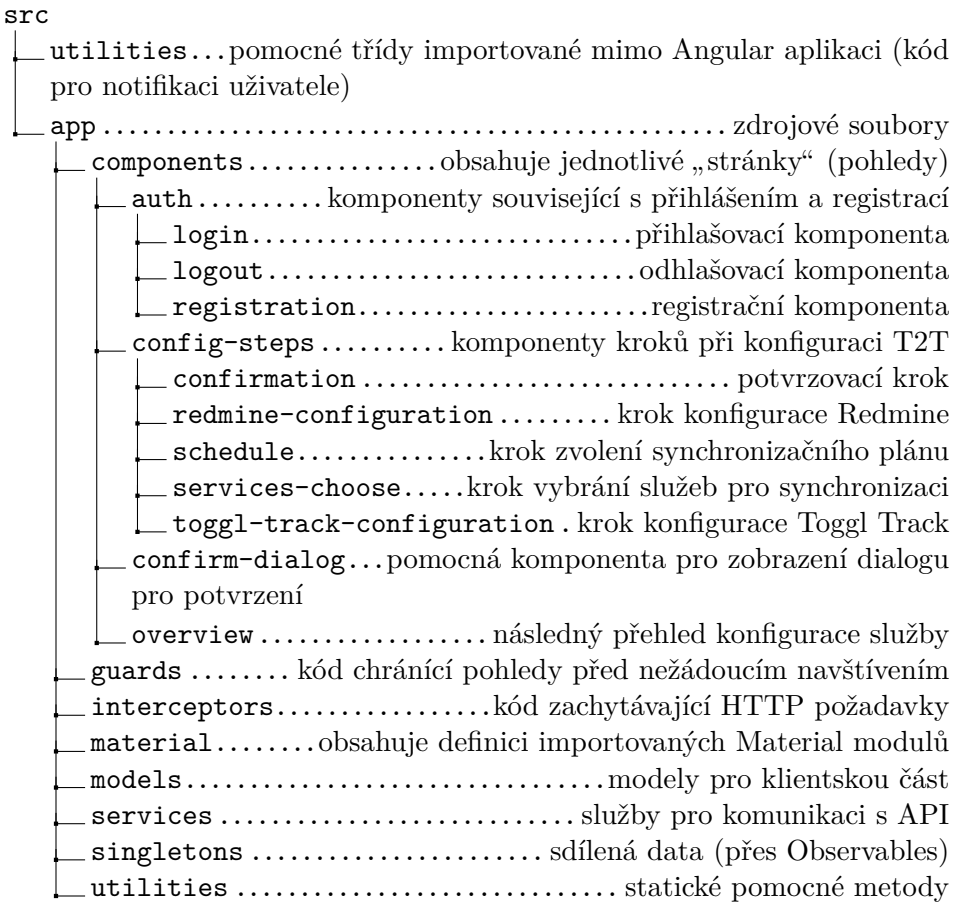
To je zároveň ale i teoreticky nevýhodou SPA, neboť se načtou data, která uživatel vlastně ani nemusí potřebovat. Nicméně klient T2T je poměrně minimalistický, tudíž zde to není takový problém. Některá data závislá na jednotlivých pohledech a na chování uživatele jsou načítána dynamicky za běhu pomocí AJAX[22]. Při implementaci jsem se řídil návrhem celého systému, ale také návrhem obrazovek, který jsem popsal v sekci 2.10 výše.

Klient T2T slouží k tomu, aby se uživatel registroval, nastavil službu a posléze mohl kontrolovat, zda synchronizace probíhá v pořádku. Jednotlivé kroky konfigurace bude možné navštívit i zpětně. Klient také bude umožňovat uživateli pozastavovat a spouštět jeho synchronizační joby. Bude umožněno i plánovat joby manuálně.

Angular aplikace stojí na komponentách. Každá komponenta definuje pohled aplikace. Skládá se obvykle z jednoho HTML, jednoho TS a jednoho CSS souboru. HTML slouží jako šablona pro umístění kódu stránky, TS pak definuje obslužný kód. Styly dostupné jen pro danou stránku se nachází v CSS souboru. Globální styly sdílené napříč aplikací definuji v globálním souboru *styles.css*. Pro komunikaci s API a pro sdílení dat využívám služeb. Všechny jednotlivé části se registrují v souboru *app.module.ts*, pomocí něhož pak Angular aplikaci složí dohromady. Jak bude probíhat přesměrování a pohyb uživatele mám definované v souboru *app-routing.module.ts*. Pomocné třídy a objekty jsou pak v adresářích *guards*, *interceptors*, *material*, *utilities*, *singletons*.

V adresáři *models* se nachází modely, které definují objekty, s kterými v aplikaci pracuji. Ty jsou velmi podobné napříč celým systémem T2T a opět reflektují databázové schéma. Rozdíly se objevují např. v objektu uživatel, který neobsahuje heš hesla, ale navíc má k dispozici token (JWT, viz sekce Bezpečnost 2.9). Na klientu se také vůbec nevyskytuje databázový objekt synchronizační objekt časového záznamu (TESO), ten je důležitý pouze pro jádro T2T. Na obrázku 3.5 na další straně je zachycena adresářová struktura klientské aplikace. Nyní detailněji prozkoumám některé její části.

[20] Single-Page Application – jednostránková aplikace.
[21] Zde se formálně nejedná o stránku, nýbrž o pohled.
[22] Asynchronous JavaScript and XML, v mém případě ale posílám a přijímám JSON.



Obrázek 3.5: Adresářová struktura klientské části T2T

3.2.1 Komponenty

V podstatě platí, že každý pohled (stránka) má svou komponentu. Kromě toho ještě existuje aplikační komponenta, která je nadřazená ostatním vnořeným. Ta mj. definuje hlavičku aplikace, ale také obsahuje načítací symbol (detailněji podsekce 3.2.4). Ještě jsem také oddělil komponentu pro potvrzovací dialog – zobrazí se pokaždé, chci-li po uživateli potvrdit nějakou akci se zásadním dopadem, aby měl případně šanci ji odvolat. To se využije např. při snaze vypnout celou synchronizaci. Navíc je komponenta díky oddělení použitelná na více místech bez zbytečné duplikace kódu.

Komponenty pohledů jsem pak rozdělil do tří kategorií, které obstarávají:

Nepřihlášeného uživatele (celek *auth*): sem patří obrazovka registrace, přihlášení a odhlášení.

Konfigurační kroky (*config-steps*): dělí se dále na volbu služeb pro synchronizaci, konfiguraci jednotlivých služeb, plánování a potvrzení konfigurace.

Přehled služby Timer2Ticket (*overview*).

Komponenty obstarávající nepřihlášeného uživatele

Každá komponenta z první kategorie nevyžaduje již přihlášeného uživatele. Na obrazovce registrace se uživatel může registrovat. Je po něm vyžadováno uživatelské jméno ve formě e-mailu, dále heslo (alespoň osm znaků dlouhé, musí obsahovat čísla) a potvrzení hesla. Po registraci se uživatel může přihlásit do Timer2Ticket, k tomu slouží komponenta přihlášení.

Ta po vyplnění uživatelského jména a hesla pošle požadavek na API, aby zkontrolovalo, zda uživatel existuje a vyplnil korektní údaje. Pokud je vše v pořádku, přesměruje uživatele na další obrazovku podle jeho stavu. Nebo pokud je nově registrovaný, aktivuje se konfigurační komponenta (první krok). Pokud již má konfiguraci provedenou, bude přesměrován na přehled služby. Odhlašovací obrazovka jen promaže uložená data a umožňuje se uživateli vrátit na přihlašovací obrazovku.

Komponenty konfiguračních kroků

Těchto komponent je obecně $k + 3$, kde k značí počet podporovaných služeb pro synchronizaci. V tuto chvíli je komponent dohromady pět (tři obecné a pak komponenty pro konfiguraci Redmine a Toggl Track). Počet kroků, které uživatel musí při konfiguraci navštívit je $1 + l + 2$, kde $l, 2 \leq l \leq k$ značí počet služeb, které si uživatel vybral v prním kroku. Zápis $1 + l + 2$ zase symbolizuje posloupnost kroků. Nejprve volí služby pro synchronizaci, následně konfiguruje zvolené služby, nakonec rozhoduje o plánování T2T a potvrzuje své volby při konfiguraci.

Každá komponenta obsahuje společné prvky: aktuální číslo kroku, počet celkových kroků, název kroku, ale také možnosti pohybu na následující a předchozí krok. Aby bylo uživateli umožněno pokračovat na další krok, většinou je po něm vyžadována nějaká akce. První komponentou je tedy volba služeb. Vždy musí zvolit alespoň dvě, jinak není vpuštěn dále. Jak jsem již zmiňoval, v tuto chvíli T2T podporuje služby Redmine a Toggl Track. Klient je nicméně připraven pro případné rozšíření o další služby.

V pořadí další komponenty umožňují základní nastavení synchronizačních služeb. V případě Redmine je po uživateli vyžadován API klíč, API přístupový bod, pak také výchozí aktivita časového záznamu a volba, zda se bude

jednat o primární službu (viz 2.6; v tuto chvíli musí být Redmine nastaven jako primární služba). Po vyplnění klíče a přístupového bodu jsou data zkontrolovaná pomocí T2T API, až poté uživatel může pokračovat. Při konfiguraci Toggl Track uživatel zadává pouze API klíč a po jeho potvrzení volí workspace, kam se mu data budou synchronizovat (v tuto chvíli není možné jej nastavit jako primární službu).

Poté, co uživatel projde nastavením služeb, si může zvolit intervaly provádění jobů. Nejprve volí plánování konfiguračního jobu, poté jobu pro synchronizaci časových záznamů (o jobech detailněji výše 2.6). Formulář pro to je shodný: uživatel volí buď synchronizaci denní – pak má na výběr interval v rámci dne (např. každou hodinu, každých šest hodin), nebo zvolí jeden či více dnů v týdnu a k němu pak i čas (tedy např. každé pondělí, středu a pátek o půlnoci).

Posledním krokem je potvrzení celého procesu. Po stisknutí tlačítka se všechna konfigurační data uloží do databáze a kromě naplánování jobů dle zvolených intervalů je rovnou naplánovaný konfigurační job. Pokud v průběhu celé konfigurace nastane chyba (Core, API či synchronizační služba nereaguje, vložená data nejsou správná…) je uživatel informován.

Komponenta přehledu služby Timer2Ticket

Zobrazuje uživateli stav celého systému. První informaci, kterou uživatel vidí při přesměrování na přehled je, zda je služba aktivní, není aktivní (odpovídá stavům uživatele *active* a *inactive*), nebo je v neznámém stavu. Umožňuje také manuálně spustit oba synchronizační joby[23].

Kromě toho pohled poskytuje uživateli informaci o již proběhlých jobech. Zde si tedy uživatel může ověřit, že vše je v pořádku a jeho data jsou synchronizovaná. Protože ke všem jobům dochází na pozadí a na provedení závisí různé faktory (pořadí ve frontě, dostupnost synchronizačních služeb…), je nutné aktivně komunikovat s API o aktuálním stavu. Data jsou brána z logů (více viz také Core 3.1.2). V přehledu jsou ihned vidět vždy poslední z obou typů jobů (konfigurační a job synchronizace časových záznamů). Po kliknutí na tlačítko „Další logy“ se zobrazí tabulka s podrobnostmi o 100 posledních jobech. Uživatel u každého vidí časové značky a také zda dopadl úspěšně, či ne. Aktuálně naplánované (nebo běžící) jsou označeny speciálním symbolem. K získání aktuálních dat je využita technika *polling*, kdy je každé tři sekundy posílán dotaz na API pro aktuální data.

Dále je z této komponenty možné celou službu vypnout a posléze také znovu spustit. Po stisknutí tlačítka se uživateli zobrazí dialog, kde ještě volbu musí potvrdit pomocí tlačítka „Ano“. Komponenta logicky musí odchytávat chyby přicházející z API: mimo jiné musí samozřejmě registrovat, že Core

[23] Přesněji: jsou naplánovány a provedou se následně podle zátěže serveru – tedy jsou vloženy na konec fronty.

neodpovídá a oznámit to uživateli (s oznámením, aby své požadavky zkusil poslat později).

Ze spodní části obrazovky lze taktéž navštívit jednotlivé kroky konfigurace. V případě, že chce např. uživatel Redmine v budoucnu změnit výchozí aktivitu časového záznamu, může to odsud provést.

3.2.2 Služby

Služby používám pro komunikaci s T2T API přes rozhraní REST. Jejich úkolem je zaslat specifický požadavek a čekat na odpověď. Až odpověď dorazí, předá zprávu volající komponentě. Všechny služby vracejí typ *Observables*, který umožní komponentě se registrovat a být notifikované v případě, že dorazila odpověď. Metody rovněž odchytávají chybu (odpovědi se stavovým kódem vyšší než či rovno 300) a předají jej komponentě, aby mohla uživatele v takovém případě upozornit.

Služeb mám k dispozici pět (fakticky reflektují API): autentizační, registrační a služby pro práci s joby, synchronizačními službami a samotným objektem uživatele. Služba pro práci s joby zajišťuje plánování jobů, zapnutí a vypnutí synchronizace a dotaz na stav synchronizace. Služby pracující se synchronizačními službami třetí strany se umožňují ptát na specifické objekty, např. u Toggl Track potřebuji znát pole *workspaces*, ze kterých si uživatel vybírá.

Do služeb by se dala také zařadit třída *AppData*. Ta je opět vkládána v aplikačním modulu, jedná se tak o singleton[24]. Obsahuje objekty, které se sdílí napříč aplikací – jedná se např. o objekt přihlášeného uživatele. S objekty pracuji ve formě *Observables*, každá změna se tedy propaguje do ostatních částí. Kromě uživatele také služba obsluhuje proces konfigurace a krokování v něm. Kroky se totiž mohou měnit dynamicky na základě zvolených synchronizačních služeb. Podle toho se rozhoduji, který další krok bude následovat. Pokud tedy uživatel zvolí v prvním kroku Redmine a Toggl Track, další kroky budou konfigurace těchto služeb. Obecně může ale zvolit jinou, případně nějakou další. Při navigaci napříč kroky se pak správně musí uživatel přesměrovávat. První krok je vždy výběr služeb, další pak jsou dynamicky volené synchronizační služby. Předposledním krokem je zvolení plánování a posledním pak akceptace konfigurace.

[24]Klasicky se toho docílí privátním konstruktorem a vystavením dané instance, Angular to však zajišťuje právě pomocí vkládání závislosti (*dependency injection*). Více viz oficiální stránky Angular [34].

3.2.3 Guards a Interceptors

Inspirací pro tyto pomocné třídy mi byl návod od autora Jason Watmore [35], kde se přesně o *guards* a *interceptors*[25] zmiňuje. První jmenovaní slouží k tomu, aby ochránily nějakou komponentu před nechtěným navštívením od uživatele. V praxi to funguje tak, že se v *app-routing.module.ts* nastaví k dané cestě i její ochránce. Když je uživatel na cestu přesměrován, nejprve se vykoná metoda *canActive* definovaná v příslušném (či příslušných, může jich být více) *guard*. Já použil služeb dvou *guards*. První zamezuje nepřihlášenému uživateli ve vniknutí na jiné stránky než na přihlašovací, registrační a odhlašovací obrazovku. Druhý pak kontroluje, zda při příchodu na přehled služby má uživatel nastavenou konfiguraci (tedy korektně prošel všemi kroky). Ochrana dobře poslouží např. při manuálním zadání URL adresy (může se stát i z důvodu kliknutí na sdílený link od jiného uživatele, nebo v rámci našeptávání od prohlížeče dle historie návštěv).

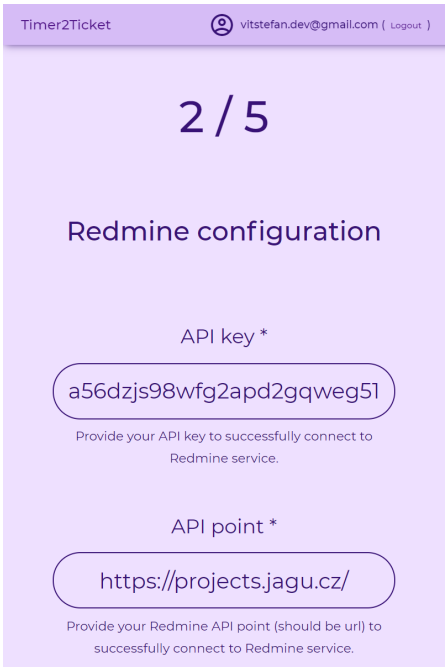
Interceptors se chovají obdobně, jen s tím rozdílem, že nechrání komponentu, ale nějakou akci. Já je využívám pro odchytávání HTTP požadavků a odpovědí. Vyšle-li služba nějaký požadavek na API, je ještě předtím navštíven *interceptorem*, resp. jeho metodou *intercept*. Ta mu vloží do hlavičky token pro autorizaci uživatele. Takto je obsloužen každý požadavek. Kdybych této možnosti nevyužil, musel bych hlavičku modifikovat u každé jednotlivé metody služby odesílající požadavek. Druhý *interceptor* naopak odchytává odpovědi od API. V případě, že přišla odpověď se stavovým kódem 401, automaticky uživatele odhlásím. API totiž vyslalo signál, že daný uživatel nemá práva pro nakládání s danými prostředky (cíleno na dotazy po vypršení platnosti tokenu). Při odhlášení dojde také k smazání uživatelských dat na klientu a uživatel je vyzván k opětovnému přihlášení.

3.2.4 Vzhled a notifikace uživatele

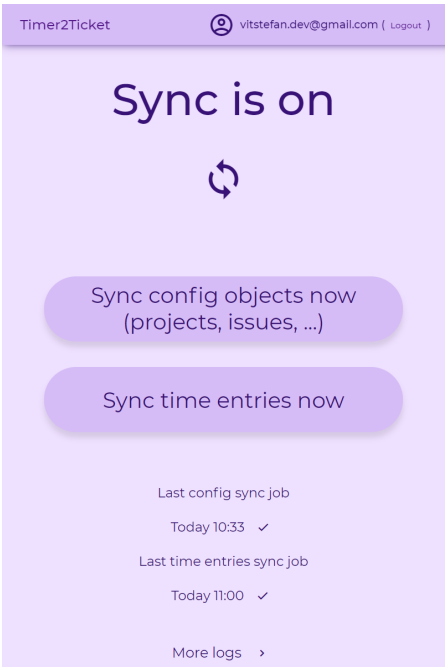
Jak jsem již psal v sekci 2.10 o návrhu obrazovek, chtěl jsem, aby se webový klient uživateli používal co nejlépe. Zvolil jsem tedy jednoduché sloupcové rozložení. Vzhledem k tomu, že při konfiguraci uživatel provádí kroky postupně, je přirozené, že postupuje zeshora dolů. Navíc je webový klient nativně připraven pro mobilní zařízení.

Řídil jsem se návrhem, který jsem si předpřipravil (sekce 2.10). Výsledná aplikace pak až na drobnosti odpovídá své předloze (na obrázcích 3.6a, 3.6b a 3.6c jsou k vidění snímky obrazovek). Rozdíl může být v některých částech, které se ukázaly nutné až při implementaci. Např. přidání tlačítek na potvrzení zadaných údajů ad. Rovněž tak přibyla tlačítka na manuální synchronizaci a některá zjednodušení pro uživatele – přidání ikonek, zjednodušení plánování. Snímky z celé webové aplikace jsou pak k dispozici v přiloženém médiu.

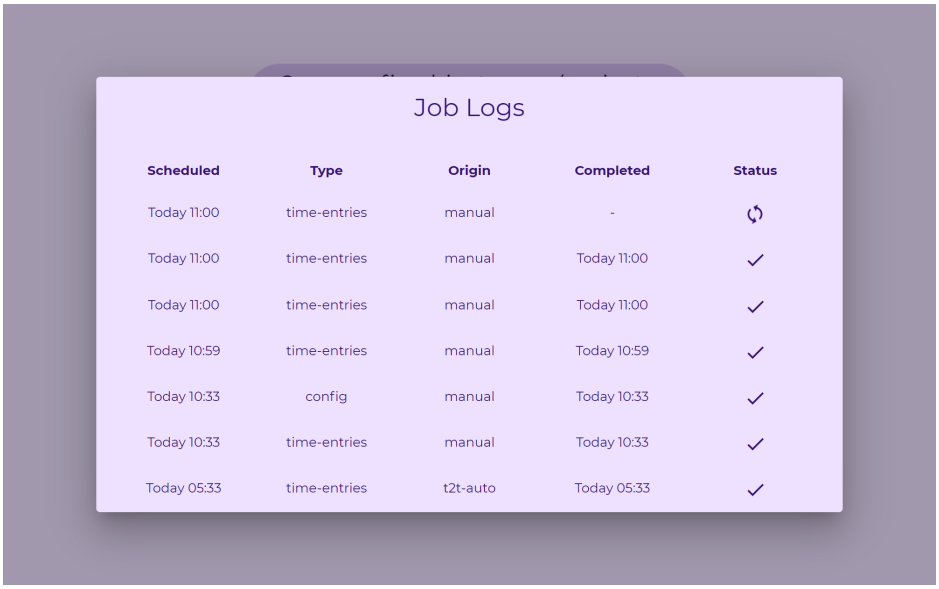
[25] Přeložitelné jako „strážci“ a „zachytávači“, ponechávám v originále



(a) Výsek z konfiguračního kroku



(b) Výsek z přehledu, hlavní část



(c) Zobrazení logů z přehledu

Obrázek 3.6: Snímky z webového klienta, mobilní a desktopové zobrazení

Samotný vzhled jsem v drtivé většině realizoval pomocí vlastního CSS. Na některé prvky jsem si ale vypůjčil již zmíněný Material modul. Zjednodušil mi tak stylizaci zaškrtávacích políček, vytvoření dialogového okna, vkládání ikon a popisků (*tooltips*). Výhodou tohoto modulu je, že jej vytvořil a používá Google, tudíž je v praxi již ověřený.

Výše (3.2.1) jsem zmiňoval aplikační komponentu, která je nadřazena všem ostatním a lze se na ni globálně odkazovat. Vzhledem k tomu, že hlavička aplikace je dostupná ze všech pohledů, umístil jsem ji právě tam. Přihlášenému uživateli i umožní se přes tlačítko odsud odhlásit. Zároveň přes komponentu zobrazuji načítací symbol. I přes to, že všechny HTTP požadavky na API zasílám asynchronně, v některých případech je vhodné uživateli naznačit, aby počkal než přijde odpověď. Například je nutné počkat na validaci vložených API klíčů. Až se klíče validují, může uživatel pokračovat na další krok. V mém případě zamezím akci ztmavením obrazovky a zobrazením rotujícího načítacího symbolu. Po příchodu dat jej opět schovám.

Často také upozorňuji uživatele na výsledky jednotlivých akcí. Provádím to pomocí notifikační lišty, která se objeví v dolní části obrazovky. Poskytuje uživateli přehled o tom, jestli je vše v pořádku. Lišta se dá kliknutím schovat, případně pak po nějaké době sama zmizí (důležitá oznámení – např. když došlo k nějaké chybě na straně Core – musí uživatel manuálně zavřít). Mechanizmus zobrazování lišty jsem implementoval již dříve sám jako samostatný modul[26], který jsem si upravil k obrazu Timer2Ticket (upravené barvy aj.). Do Angular aplikace jej stačí přidat jako externí knihovnu a odkázat se na ni v aplikační komponentě.

[26]Dostupný zde: `https://github.com/vitstefan/notifs`.

3.3 Nasazení systému

Samotné nasazení probíhalo ve spolupráci s Ing. Pavlem Kovářem, který spolupracuje se společností Jagu s.r.o. Po jeho doporučení jsme každou ze čtyř částí nasadili jako Docker kontejner. Navíc, vytvoření Docker obrazů (*images*) z NodeJS aplikací probíhá automaticky pomocí GitHub Actions[27].

Stručně se dá proces nasazení popsat takto: k potřebnému commitu v GitHub repozitáři[28] vytvořím vydání (*release*) ve formě git tagu. V tu chvíli se spustí GitHub akce, která aplikaci sestaví, vytvoří Docker image a nahraje ho do GitHub Package Registry. Ten je pak odkazovaný na serveru. Pomocí příkazu `docker-compose up -d` ze serveru se na základě souboru *docker-compose.yml* spustí nasazení všech čtyř částí (viz Architektura 2.5). V případě, že chci nasadit nové změny, postupuji shodně. Spouští se pouze ty části, u kterých proběhla změna.

Přikládám zjednodušený seznam kroků, které bylo potřeba provést pro přípravu nasazení. První tři kroky jsem provedl já (kolega posléze zkontroloval a drobně opravil), poslední bod celý navrhl a udělal Ing. Pavel Kovář. Bylo tedy nutné:

1. Pro části Core, API a klienta připravit soubor *publish-image.yml*, na jehož základě se provádí GitHub akce (čerpal jsem z oficiální dokumentace [37]).
2. Definovat HTTP server NGINX [38], na kterém poběží klientská Angular aplikace (důležitým zdrojem pro mě byl článek [39] od autora Bhargav Bachina).
3. Rovněž pro tyto části vytvořit soubor *Dockerfile*, který zařídil kompilaci kódu v kontejneru (dokumentace [40]).
4. Připravit *docker-compose.yml*, který vše včetně databáze spustí.

Systém je dostupný na adrese `https://timer2ticket.jagu.cz`. Zdrojové soubory všech částí, včetně konfiguračních souborů jako je např. *Dockerfile* jsou k dispozici v přiloženém médiu. Repozitáře jsou umístěné na GitHub. Odkaz `https://github.com/vitstefan/timer2ticket` slouží jako rozcestník na jednotlivé části.

[27] Automatizované akce, které běží přímo nad GitHub repozitářem. Vývojář definuje kdy a jaké akce se provádí. Více [36].

[28] Každá ze tří impementačních částí má svůj vlastní.

3.4 Rozšíření o další synchronizační službu

Na závěr kapitoly o realizaci ještě předkládám kroky, které je potřeba provést v případě integrace nové synchronizační služby. Při návrhu systému bylo dbáno na jeho obecnost, většina kroků tak vychází z již integrovaných služeb. Vše je samozřejmě závislé na dané službě a může se objevit odlišnost od současných tak, že bude potřeba nějakou část upravit více. Naopak se ale může stát, že integrace bude přímočará a víceméně bude vycházet ze služeb již podporovaných.

API

- Doplnit identifikátor služby do výčtu názvů v modelu *ServiceDefinition*. Tento výčet je zde definován kvůli validaci objektu.
- Připravit HTTP GET metodu v souboru *synced_service_config.ts* pro získání všech potřebných údajů nové služby.

Core

- Vytvořit třídu implementující rozhraní *SyncedService*. Odkázat se na ni ve třídě *SyncedServiceCreator*.
- Navrhnout třídu pro model časového záznamu, který přijde z dané služby. Stačí implementovat připravené rozhraní *TimeEntry*.

Klient

- Implementovat další komponentu, která bude hrát roli pohledu při konfiguraci nové služby. Je důležité na ni vložit referenci do pole deklarací v *AppModule*.
- Definovat novou metodu ve třídě *SyncedServicesConfigService*. Ta bude vytvářet požadavek na vytvořenou metodu API zmíněnou výše.
- Přidat nový objekt do pole *_possibleServicesAndRoutes* v singletonu *AppData*. V objektu se nachází mj. identifikátor a název služby, ale také správná cesta ke komponentě.

Společné

- Případně doplnit objekt *user.serviceDefinition.config* o další atribut, který bude nová služba potřebovat (Redmine např. vyžaduje identifikátor uživatele a výchozí aktivitu časových záznamů).

4 Testování

Kapitola se zabývá testováním aplikace. Stručně popíšu testování kódu, ale také jak jsem postupoval v průběhu implementace. Dojde i na popis testování po nasazení, kdy systém již využívali uživatelé s reálnými daty. Na závěr prozkoumám i zpětnou vazbu od uživatelů.

4.1 Testování kódu aplikace

První a základní variantou testování jsou jednotkové testy. Ty by měly testovat vždy izolované jednotky kódu. Pro NodeJS existuje celá řada jednotkových frameworků, já jsem zvolil Mocha [41]. Svým způsobem zápisu mi vyhovoval ze všech nejvíce.

Takto jsem otestoval API část. A to celky jobs, autentizaci a registraci. Vytvořil jsem prostředí, které pracuje se samostatnou databází určenou pouze pro jednotkové testy. Na začátku se do ní nahrají potřebná data a na konci testovacích celků se všechna data smažou. Mocha umožňuje definici speciálních funkcí, které jsou volány před/po každém testu/celku. Právě ty jsem využil na počáteční inicializaci databáze a následné promazání.

Pro jednotkové testování Core části by bylo potřeba simulovat chování služeb třetích stran. Další možností je založení testovacích účtů u daných služeb, na kterých by se funkcionality strojově testovaly.

4.2 Testování na reálných datech

V průběhu vývoje jsem měl založených několik účtů ve službě Toggl Track s několika *workspaces*. Měl jsem rovněž přístup do Redmine společnosti Jagu jako uživatel. Tam jsme založili testovací projekt a požadavky, na kterých jsem mohl zkoušet funkčnost systému.

Velice šikovným API klientem se ukázalo být řešení od Insomnia [42]. Pomocí této aplikace lze velmi snadno testovat jak lokální REST rozhraní, tak i rozhraní služeb třetích stran (Redmine, Toggl Track). Umožňuje velmi snadno zadávat potřebné parametry dotazu, včetně např. autorizace pomocí API klíče. Jednotlivé dotazy lze ukládat a volně mezi nimi přepínat. Také si pamatuje odpovědi i při příštím spuštění, není tedy nutné vše posílat znovu.

Základní funkcionalitu jsem vyzkoušel na svých účtech a lokálním nasazení. Hlavním cílem však bylo rychlé nasazení na server, aby systém mohli vyzkoušet i další uživatelé na svých datech ve větším měřítku.

4.3 Testování po nasazení

Vzhledem k tomu, že systém propojuje několik dalších nezávislých služeb (Redmine, Toggl Track), je nejvhodnější jej testovat na reálných datech. 25. března 2021 byl systém nasazen včetně odladěných prvotních kritických problémů. Prvotním testovacím uživatelem byl Ing. Oldřich Malec. Na základě jeho pozorování a připomínek jsem systém dále dolaďoval.

Za běhu bylo přidáno i logování jobů (doposud byla dostupná pouze informace o posledním úspěšném jobu). Každý job si uchovává informaci o běhu, včetně časových značek. Uživatel si tedy nyní na webovém klientu může zobrazit logy v přehledné tabulce. Také jsem v rámci refaktoringu omylem zanesl chybu, která způsobovala duplikaci nejstarších záznamů (způsobeno špatným stránkováním časových záznamů u Toggl Track).

Další výzvou bylo sdílené pracovní místo (*workspace*) v aplikaci Toggl Track. S tím, že by mohlo být sdílené pracovníky se nejdříve při návrhu nepočítalo. Po vznesení požadavku byl systém mírně poupraven tak, aby i toto umožňoval. Znamenalo to zejména to, že v rámci sdíleného *workspace* mají uživatelé sdílené projekty a tagy. Zde jsem upravil systém tak, aby při vytvoření objektu na základě primární služby nejprve vyzkoušel, zda již neexistuje[29]. Pokud ano, T2T využilo jej (předtím tento stav signalizoval chybu na straně třetí služby). Časové záznamy si uživatelé vytváří sami za sebe. Vše se zdálo být v pořádku, ale po zavedení nového uživatele T2T, který pracovní místo sdílel, se časové záznamy chybně duplikovaly. To jsem vzápětí opravil a sdílení nyní již funguje bez výhrad.

Kromě toho se v průběhu objevily také další menší chyby či problémy, o kterých zde již nepíšu. Sice neohrozily hlavní funkcionalitu, ale bylo samozřejmě potřeba je odchytit a opravit.

Stojí za to ale zmínit, že v produkční verzi systému je implementované logování chyb pomocí služby Sentry [43]. Tu je možné včlenit přímo do kódu tak, že posílá mnou definované zprávy do speciální webové aplikace Sentry, kde je mohu dále zkoumat. Nastavil jsem ji tak, že vytvoří novou zprávu pokaždé, skončí-li job nějakou chybou. Zpráva následně obsahuje i výpis z konzole, ale také např. proběhlou HTTP komunikaci. Takto snadno mohu kontrolovat, zda nedochází k nežádoucím chybám. Navíc rovnou vidím, co proběhlo špatně a mohu podle toho chybu dohledat v kódu a opravit ji.

[29]Toggl Track totiž neumožňuje vytvářet více např. tagů se stejným jménem.

4.4 Zpětná vazba od uživatelů

Poté, co byl systém nasazený, se mohlo začít testovat nad reálnými daty. Protože jsem byl po celou dobu v kontaktu s lidmi, s kterými jsem probíral požadavky ještě před samotným návrhem, bylo logické, že budou systém i zkoušet. Ještě než systém začali používat, poprosil jsem je, aby zhlédli mnou nahrané průvodní video, které osvětluje základní koncepty systému (cíleno na uživatele, se kterými jsem prováděl úvodní dotazník). To je dostupné na YouTube zde: `https://youtu.be/y8OLSy3vXRE`.

Se všemi uživateli jsem po nějaké době, kdy Timer2Ticket používali, provedl kratší interview ohledně funkčnosti systému. To jsem vedl podobně jako interview úvodní (viz sekce 2.4). Opět jsem provedl záznam všech rozhovorů. Po dodatečném odsouhlasení od příslušné osoby je možné do něj nahlédnout. Rozhovor jsem dělal se čtyřmi osobami (tři osoby byly shodné jako u prvního dotazníku) a zde přináším jejich shrnutí. Kompletní (anonymizovaná) verze poznámek, které jsem si dělal v průběhu, je k nalezení uvnitř přiloženého paměťového média.

Pro rozhovor jsem si připravil scénář (k prozkoumání v příloze B.2 na konci práce). Chtěl jsem mít jistotu, že probereme vše podstatné.

Všem dotazovaným se systém velice líbil a synchronizace podle nich probíhá velmi dobře. Každý z nich nejprve pár dní vždy večer systém kontroloval, zda záznamy převádí správně. Posléze získali v systém důvěru, takže již kontrola nebyla třeba. Všichni oceňují funkčnost a spolehlivost. Většinou systém používají (vztaženo k datu rozhovoru) zhruba týden, jeden uživatel téměř měsíc.

Ptal jsem se jich na proces registrace a konfigurace systému. Ten všichni hodnotí jako přímočarý a jasný. Ocenili by, kdyby byl trochu více uživatelsky přívětivý: popis (či odkaz), kde naleznou API klíče od daných služeb, nebo třeba lepší zpětnou vazbu při zadávání hesla (např. není dostatečně dlouhé ad.). Kritické výhrady se nevyskytly.

S potěšením jsem zjistil, že kvůli přechodu na synchronizaci pomocí T2T nemuseli dělat takřka nic v již používaných službách kromě vytvoření nového *workspace* v Toggl Track. Samotná příprava na přechod tedy náročná nebyla.

Uživatelé zmiňovali nutnost adaptace u nového stylu vykazování v Toggl Track: nyní bylo navíc povinné pro správnou synchronizaci vybrat tag s požadavkem (nebo projektem). To však proběhlo hladce. Každý z nich si přechod na T2T velice pochvaluje, hlavně z toho důvodu, že nemusí řešit každodenní ruční převádění výkazů z jednoho systému do druhého. Vše je nyní převáděno automaticky na pozadí.

Jeden uživatel do doby před přechodem na T2T využíval pouze Redmine. S přechodem na T2T začal čas nově vykazovat čas v Toggl Track. Uživatel je se změnou spokojen a velmi by si rozmýšlel, zda by bez T2T přešel na nový nástroj pro efektivnější zaznamenávání času napříč dnem. Musel by totiž zá-

znamy stejně manuálně převádět do Redmine. Takto je naprosto spokojený jak s Toggl Track, tak i s Timer2Ticket.

Když jsem se uživatelů ptal, čím by systém vylepšili, zmiňovali převážně drobné změny ve webovém klientu. Hlavně proto, aby byl pro uživatele průchod konfigurací ještě více intuitivní. Např. by jeden dotazovaný uvítal možnost vidět časy synchronizace v přehledu hned po přihlášení do webové aplikace (nyní jsou dostupné až po kliknutí na daný konfigurační krok). K tomu se váže i požadavek mít možnost vidět čas, kdy proběhne další automatický job dle jeho nastavení.

Jeden uživatel zmínil, že by bylo vhodné Toggl Track záznamy ze stejného dne a stejnými parametry seskupovat. Rovněž tak by bylo užitečné, kdyby systém automaticky zpetně doplňoval do Togglu Track k záznamům i projekt. Ten lze vybrat ručně, ale není to pro korektní synchronizaci třeba (stačí zvolit požadavek). Mohl by to tak systém dělat automaticky na základě přiřazeného požadavku (tím je Redmine projekt jednoznačně určen).

Celkově se dá říci, že jsou uživatelé se systémem velmi spokojení. Většina nápadů na změny se týkala samotného webového klienta a konfigurace. Co se týče samotné synchronizace a nového způsobu vykazování času, téměř nemají výhrad. Všichni vyzdvihují, že mnou navržený systém jím každý den ušetří dost času – dokonce i až 15-20 minut denně. Dle jejich slov by se nyní neradi vraceli zpět do doby, kdy Timer2Ticket nepoužívali.

5 Možná vylepšení a rozšíření aplikace, budoucí rozvoj

V kapitole prozkoumám možnosti, jak systém vylepšit a rozšířit. Zhodnotím i jiné možné přístupy při synchronizaci. Nakonec se zaměřím na budoucí rozvoj a zpřístupnění systému dalším uživatelům.

5.1 Synchronizace časových záznamů

Nyní se synchronizují všechny časové záznamy, které připadají na datum starší než den registrace uživatele do Timer2Ticket. To je pro uživatele samozřejmě nejvíce přínosné, nicméně časem se nahromadí kvanta záznamů. Znamená to, že služba musí nejprve všechny tyto záznamy načíst z ostatních služeb, ale také se musí uchovávat informace o mapování v databázi. Celkově se tedy jedná o narůstající počet dat, která budou muset být obhospodařována. Navíc se uživatel jen málokdy dívá do historie (obzvlášť do té vzdálené) a tyto záznamy skutečně mění.

Služba Timer2Ticket by tedy mohla brát vpotaz pouze výsek ze všech záznamů. Ideálně ty, jejichž data se měnila nedávno – každý časový záznam obsahuje svou časovou značku, kdy se měnil. Bohužel však API Toggl Track ani Redmine neumožňují podle tohoto údaje filtrovat a data načíst.

Jediným rozumným atributem, podle kterého by se záznamy daly filtrovat, zůstává datum záznamu. Znamenalo by to, že záznamů, které by musel každý job synchronizace projít, by bylo výrazně méně (a hlavně jich prakticky je vždy omezené menší množství). Na druhou stranu by případy, kdy uživatel změní časový záznam staršího data, T2T již nezohlednil. Tudíž by si případnou synchronizaci mezi všechny služby musel provést manuálně sám.

Nicméně by to stále nevyřešilo problém s množstvím dat v databázi. Mapování by stále musela být uchovávána i pro staré záznamy. Mohla by totiž nastat situace, že uživatel náhle změní datum ze starého záznamu na aktuální. Musí se nejprve ověřit v databázi, zda tento záznam již nebyl mapován. Pokud by se tak nestalo, systém by vytvořil mapování nové a založil záznamy do ostatních služeb. Jeden záznam by tedy vytvářel duplikované záznamy v ostatních službách. Navíc, musí být v databázi i uchovávaná informace o datu, ke kterému záznam patří. Mohlo by se zase stát, že z aktuálního ho uživatel převede do minulosti, kdy se bude vyskytovat už mimo T2T filtr. Musí být načteny všechny

záznamy i dle databázového nového políčka, aby se tento případ identifikoval a korektně spároval s daným záznamem, a záznamy v ostatních synchronizačních službách byly správně převedeny na nové datum v minulosti. Tyto zmíněné situace nejsou moc pravděpodobné, ale nastat teoreticky mohou.

Ve zpětné vazbě od uživatelů jsem uváděl, že by stálo za to se zamyslet nad seskupováním shodných (stejné tagy a stejný komentář) časových záznamů z jednoho dne. Toggl Track totiž toto dělá automaticky. Nicméně reálně jsou to různé objekty, proto se do Redmine zapíší individuálně. Abych tohoto efektu docílil, a přesto ponechal informaci o tom, že se jedná o různé objekty, musel bych mapovacím objektům (TESO) přiřadit identifikátor skupiny. Vždy, pokud by systém narazil na nový záznam, by musel zkontrolovat, zda neexistuje z toho dne ještě další, který má stejné parametry. Pokud ano, vytvořil bych novou skupinu a sečtené časy by byly vloženy do Redmine pod jedním objektem. Pokud by se náhodou některý z nich změnil tak, že by již nevyhovoval podmínce seskupení (např. by mu byl změněn projekt), musela by se skupina rozdělit a stejně tak příslušné záznamy v Redmine.

5.2 Synchronizace konfiguračních objektů

Podnětem k zamyšlení je také objem dat, které zpracovává konfigurační job. Uživateli zprostředkovává všechny důležité objekty (projekty, požadavky…) z primární služby do všech ostatních. Do Toggl Track se vpisují ve formě projektů a tagů. Těchto objektů však může být hodně. Práce konfiguračního jobu je zkontrolovat tyto objekty, zda u nich nenastala nějaká změna. Pokud ano, budou změny propsány i do ostatních služeb. Tato funkcionalita je pro uživatele velice pohodlná, všechny objekty včetně změn má dostupné rovnou v každé službě.

Každý uživatel ale pracuje s odlišnými objekty. Někdo často přepíná kontext a pracuje nad mnoha projekty a požadavky. Naopak ale někdo může dlouhodobě pracovat pouze s pár požadavky na jednom projektu. Toto nelze obecně posoudit. Filtraci nad těmito objekty také nelze přímo provádět – tedy kromě toho, že uzavřené požadavky již nemusí být synchronizovány.

Jedním možným řešením by bylo kompletně konfigurační job vypustit a nechat uživatele vkládat potřebné informace manuálně. Tedy např. by musel do komentáře (nebo do tagu) v Toggl Track vyplnit číslo požadavku a identifikátor aktivity. Tyto údaje by pak musel T2T z textu extrahovat a do Redmine vyplnit. To by ale znamenalo velký krok směrem k menšímu pohodlí uživatele. Takto by musel nejprve otevřít Redmine, kde by našel číslo požadavku, na kterém aktuálně pracuje a zkopírovat ho do Toggl Track. V aktuální verzi T2T jen vyhledá mezi již předpřipravenými tagy ten, který odpovídá požadavku (buď podle čísla, nebo názvu). Také má přehled i třeba o tom, změní-li se název projektu/požadavku.

Další variantou je nechat uživatele zvolit si na webu Timer2Ticket ty objekty, které by chtěl synchronizovat. Musely by se do prohlížeče vždy načíst všechny ty, které jsou k dispozici a informace o nich, zda jsou již synchronizovány. Uživatel by pak vybral ručně ty, které chce synchronizovat. Problém tohoto řešení je to, že pokaždé, chce-li uživatel pracovat na jiném požadavku (na kterém dosud nepracoval), musí jej nejprve v T2T označit. Znamená to další starost, kterou uživatel musí řešit a systém by se tak stal náročnějším na používání. Mohlo by ho to i odradit v začátcích, kdy se službou teprve začíná.

5.3 Budoucí rozvoj

Pro rozšíření mezi nové uživatele by bylo potřeba vylepšit převážně webového klienta. To vychází i z průzkumu mezi uživateli v průběhu testování (viz kapitola o testování, sekce 4.4 Zpětná vazba od uživatelů). Implementace úvodní stránky, která přehledně ukáže zájemcům co systém dělá, bude nutná. V ideálním případě by mohlo být vytvořeno i instruktážní video, které uživatelům vysvětlí základní koncepty systému (navázat se může na video, které jsem vytvořil pro testovací uživatele – zmínil jsem se o něm v kapitole 4 Testování výše). Do klienta by bylo dobré zabudovat větší zpětnou vazbu pro uživatele: více návodných informací, popisků a vysvětlivek.

Kromě toho by bylo vhodné rozšířit současný systém logů. Uživatel by jistě ocenil, pokud by měl u jednotlivých jobů přehled o tom, co udělal. Pokud by nastala nějaká chyba, se kterou si T2T neporadí, je nutné uživatele informovat, aby mohl zajistit nápravu. Přidanou hodnotou bude, pakliže mu přijde tato informace i do e-mailové schránky.

Systém samozřejmě získá další potenciální klienty také tím, že rozšíří počet synchronizačních služeb, které podporuje. Úpravy, které budou muset být provedeny, může udělat v podstatě kdokoliv: vzhledem k tomu, že kód je dostupný jako open source. Kroky, které je potřeba pro integraci nové služby jsem popsal v sekci 3.4.

Případný model využívání pro budoucí uživatele

Běh a správa aplikace něco samozřejmě stojí. Zde navíc platí, že pro každého nového uživatele poběží automatické joby, které budou spouštěny nezávisle na aktivitě daného uživatele. Je tedy logické, že by uživatel za využívání služby měl platit (pokud by měl být systém v budoucnu výdělečný). Zobrazování reklam zde nepřipadá v úvahu, neboť je systém založený na běhu na pozadí.

Dle mého názoru je vždy přínosné, pokud systém nabízí nějakou formu účtu zdarma. Zde se přímo nabízí omezit automatické joby (třeba maximálně jednou týdně, jednou denně…), případně je kompletně vyřadit: pokud by uživatel chtěl službu využívat bezplatně, musel by být závislý na manuálním spouštění jobů. Placené účty by restriktivní nebyly. Nicméně i ty mohou být odstupňovány: pro základní potřeby by stačil omezený běh na dvakrát denně,

existovala by i prémiovější varianta. U ní by bylo omezení plánování ještě menší (např. jedenkrát za hodinu, případně ještě častěji). Kromě toho by prémiovým uživatelům mohla být poskytnuta podpora – aktivní řešení problémů, nebo přidání funkcionality do systému (např. integrace zbrusu nové služby).

Funkcionalita by se dala také omezit tím, že by automatické běhy byly povoleny jen pro jeden druh jobu (např. pro synchronizaci časových záznamů).

K tomu všemu by každý uživatel při registraci mohl mít nárok na bezplatné demo, které by bylo omezené časově. Např. by po dobu 30 dnů mohl službu využívat naplno bez jakýchkoli omezení. Poté by musel přejít na bezplatné (omezené) používání, nebo postoupit na některý ze zpoplatněných účtů.

Závěr

Cílem práce bylo vytvořit systém synchronizující data z projektových systémů a aplikací na sledování času tak, aby je uživatelé nemuseli převádět ručně. Takový systém jsem navrhl, implementoval a nasadil do produkce. Vycházel jsem přitom ze sběru požadavků od uživatelů právě takovýchto služeb. Zanalyzoval jsem aplikace, jejichž data budou takto synchronizována, a provedl rešerši existujících řešení. Neopomněl jsem ani na databázovou strukturu. Vše jsem pak skloubil dohromady při realizaci aplikace. Systém je nasazený a pro synchronizaci dat mezi aplikacemi Redmine a Toggl Track ho používají reální uživatelé. Od nich jsem získal cennou zpětnou vazbu. Tu jsem využil i při navržení možných vylepšení a rozšíření.

Při analýze jsem se dozvěděl, že takový obecný systém vlastně neexistuje. Uživatelé tak musí data převádět mezi službami ručně. Takový způsob je velmi neefektivní a navíc náchylný k chybám při přepisu. Výsledkem této práce je tedy systém, který toto provádí za uživatele. Po jednoduché registraci a konfiguraci systému přes webového klienta se na pozadí automaticky spouští joby, které data synchronizují. Uživatelé se tak mohou více soustředit na práci a nemusí na konci každého dne ještě data převádět.

Systém je navíc rozdělený do tří na sobě nezávislých služeb (plus databáze), které spolu komunikují přes rozhraní REST. Je tak snáze testovatelný a úprava jedné se vůbec nemusí dotknout dalších. Systém je dostatečně obecný a připravený na rozšíření o novou synchronizační službu.

Navrhl a realizoval jsem spolehlivý systém, který uživatelům ulehčuje práci a šetří čas. Aplikace je nyní dostupná na `https://timer2ticket.jagu.cz` pro všechny, kteří řeší tento problém. Do doby, než se přejde na nějakou formu monetizace z důvodu pokrytí nákladů, je používání zdarma a bez omezení.

Použitelnost v praxi mi byla potvrzena i v průběhu testování a zpětnou vazbou od uživatelů. Dává smysl tedy práci dále rozšiřovat. Kromě zlepšení webového klienta je do budoucna plánovaná i integrace nových synchronizačních služeb do systému. Systém je již nyní plně využíván v praxi prvními uživateli a počítá se i s nabízením této služby uživatelům dalším.

Seznam použité literatury

1. ATLASSIAN. *Jira Software* [aplikace]. 2021 [cit. 2021-03-15]. Dostupné z: https://www.atlassian.com/software/jira/.
2. LANG, Jean-Philippe et. al. *Redmine* [aplikace]. 2021 [cit. 2021-03-15]. Dostupné z: https://www.redmine.org/.
3. TOGGL. *Toggl Track* [aplikace]. 2021 [cit. 2021-03-13]. Dostupné z: https://toggl.com/track.
4. CLOCKIFY. *Clockify* [aplikace]. 2021 [cit. 2021-03-15]. Dostupné z: https://clockify.me/.
5. JGRAPH LTD. *diagrams.net* [aplikace]. 2021 [cit. 2021-03-10]. Dostupné z: https://app.diagrams.net.
6. NUTCACHE. *Nutcache* [aplikace]. 2021 [cit. 2021-03-15]. Dostupné z: https://www.nutcache.com/.
7. HARVEST. *Harvest* [aplikace]. 2021 [cit. 2021-03-15]. Dostupné z: https://www.getharvest.com/.
8. TOGGL. *Toggl Track Integrations* [online]. 2021 [cit. 2021-03-13]. Dostupné z: https://toggl.com/track/integrations/.
9. TOGGL. *Jira Sync [Beta feature]* [online]. 2021 [cit. 2021-03-13]. Dostupné z: https://support.toggl.com/en/articles/4794538-jira-sync-beta-feature.
10. MEHTA, Jigar. *Toggl 2 Redmine* [software knihovna]. 2021 [cit. 2021-03-13]. Dostupné z: https://github.com/jigarius/toggl2redmine.
11. INC., Zapier. *Zapier* [aplikace]. 2021 [cit. 2021-03-13]. Dostupné z: https://zapier.com.
12. TALÁR, Denis. *Optimalizace nejčastějších procesů ve skladovém systému* [online]. Praha, 2020 [cit. 2021-03-16]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/88290/F8-BP-2020-Talar-Denis-thesis.pdf. Bakalářská práce. České vysoké učení technické v Praze.
13. SPECIALISTS, IfD_Qualitative Research. *Semi-structured interviews guide* [online]. 2018 [cit. 2021-03-16]. Dostupné z: https://www.youtube.com/watch?v=8z8XV1S7548.

14. TALEBOOK. *Secrets of perfect user interview by Talebook* [online]. 2018 [cit. 2021-03-16]. Dostupné z: https://uxplanet.org/secrets-of-perfect-user-interview-by-talebook-ee61a7e155c7.

15. MONGODB, Inc. *The database for modern applications* [online]. 2021 [cit. 2021-03-13]. Dostupné z: https://www.mongodb.com.

16. MONGODB, Inc. *About Us* [online]. 2021 [cit. 2021-03-13]. Dostupné z: https://www.mongodb.com/company.

17. FENTON, Jim. *Informatický večer – Ověřování uživatelů – když (jen) hesla nestačí* [online]. 2019 [cit. 2021-03-13]. Dostupné z: https://www.youtube.com/watch?v=80gxq3xWLk4%5C&feature=youtu.be%5C&fbclid=IwAR2_XQYTxkB5Cv8emhioeysf_29COWR_5sCRObJOMTp6E6ZYDVG-sgCHmhQ. [Kanál uživatele AVC ČVUT].

18. CAMPBELL, Nick. *node.bcrypt.js* [software knihovna]. 2021 [cit. 2021-03-13]. Dostupné z: https://github.com/kelektiv/node.bcrypt.js.

19. FIGMA. *Figma* [aplikace]. 2021 [cit. 2021-04-02]. Dostupné z: https://www.figma.com/.

20. MICROSOFT. *TypeScript* [online]. 2021 [cit. 2021-03-13]. Dostupné z: https://www.typescriptlang.org/.

21. WILSON, Jim R. *Node.js 8 the right way: practical, server-side JavaScript that scales*. P1.0. Raleigh, North Carolina: Pragmatic Bookshelf, [2018]. ISBN 978-1-68050-195-7.

22. FOUNDATION, OpenJS. *NodeJS* [software framework]. 2021 [cit. 2021-03-13]. Dostupné z: https://nodejs.org/.

23. FOUNDATION, OpenJS. *Express* [software framework]. 2017 [cit. 2021-03-15]. Dostupné z: https://expressjs.com/.

24. FISCHER, Lars; HANENBERG, Stefan. An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability Using TypeScript and JavaScript in MS Visual Studio. *SIGPLAN Not.* 2015, roč. 51, č. 2, s. 154–167. ISSN 0362-1340. Dostupné z DOI: 10.1145/2936313.2816720.

25. VISIONMEDIA. *SuperAgent* [software knihovna]. 2021 [cit. 2021-03-15]. Dostupné z: https://visionmedia.github.io/superagent/.

26. TYPESTACK. *ClassValidation* [software knihovna]. 2021 [cit. 2021-03-15]. Dostupné z: https://github.com/typestack/class-validator/.

27. CAMPBELL, Nick. *node-cron* [software knihovna]. 2020 [cit. 2021-03-13]. Dostupné z: https://github.com/kelektiv/node-cron.

28. GOOGLE. *Angular* [online]. 2021 [cit. 2021-03-13]. Dostupné z: https://angular.io.

29. LESH, Ben et. al. *RxJS: Reactive Extensions Library for JavaScript* [software knihovna]. 2021 [cit. 2021-03-13]. Dostupné z: `https://rxjs-dev.firebaseapp.com/guide/overview`.

30. GOOGLE. *Angular Material* [software knihovna]. 2021 [cit. 2021-04-02]. Dostupné z: `https://material.angular.io/`.

31. FOUNDATION, OpenJS. *Express routing* [online]. 2021 [cit. 2021-04-06]. Dostupné z: `https://expressjs.com/en/guide/routing.html`.

32. CONTRIBUTORS, Mozilla. *Cross-Origin Resource Sharing (CORS)* [online]. 2021 [cit. 2021-04-06]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

33. DOCKER. *Toggl API Documentation* [online]. 2021 [cit. 2021-04-10]. Dostupné z: `https://github.com/toggl/toggl_api_docs`.

34. ANGULAR. *Singleton services* [online]. 2021 [cit. 2021-04-02]. Dostupné z: `https://angular.io/guide/singleton-services`.

35. WATMORE, Jason. *Angular 10 – JWT Authentication Example & Tutorial* [online]. 2020-07-09 [cit. 2021-04-02]. Dostupné z: `https://jasonwatmore.com/post/2020/07/09/angular-10-jwt-authentication-example-tutorial`.

36. GITHUB, Inc. *Github Actions: Automate your workflow from idea to production* [online]. 2021 [cit. 2021-04-09]. Dostupné z: `https://github.com/features/actions`.

37. GITHUB, Inc. *GitHub Docs: GitHub Actions* [online]. 2021 [cit. 2021-04-09]. Dostupné z: `https://docs.github.com/en/actions`.

38. SYSOEV, Igor. *NGINX* [HTTP server]. 2021 [cit. 2021-03-13]. Dostupné z: `https://nginx.org/en/`.

39. BACHINA, Bhargav. *How To Serve Angular Application With NGINX and Docker* [online]. 2021 [cit. 2021-04-09]. Dostupné z: `https://medium.com/bb-tutorials-and-thoughts/how-to-serve-angular-application-with-nginx-and-docker-3af45be5b854`.

40. DOCKER. *Docker Docs: Use containers for development* [online]. 2021 [cit. 2021-04-09]. Dostupné z: `https://docs.docker.com/language/nodejs/develop/`.

41. FOUNDATION, OpenJS. *Mocha* [software framework]. 2021 [cit. 2021-04-24]. Dostupné z: `https://mochajs.org/`.

42. KONG, Inc. *Insomnia: Build APIs that work.* [aplikace]. 2021 [cit. 2021-04-28]. Dostupné z: `https://insomnia.rest/`.

43. FUNCTIONAL SOFTWARE, Inc. *Sentry* [aplikace]. 2021 [cit. 2021-04-24]. Dostupné z: `https://sentry.io/`.

A Seznam použitých zkratek

AJAX Asynchronous JavaScript And XML

BSON Binary JSON

CORS Cross-Origin Resource Sharing

ČVUT FIT České vysoké učení technické v Praze, Fakulta informačních technologií

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IT Information Technology

JS JavaScript

JSON JavaScript Object Notation

JWT JSON Web Token

ORM Object-Relational Mapping

REST API Representational State Transfer Application Programming Interface

SPA Single-Page Application

STEO Service Time Entry Object

T2T Timer2Ticket

TESO Time Entry Synced Object

TS TypeScript

UML Unified Modeling Language

XML eXtensible Markup Language

B Scénáře pro rozhovor s uživateli

V kapitole jsou uvedeny dva scénáře, které jsem si připravil jako podklad pro rozhovory s uživateli. První jsem prováděl úplně na začátku ještě před samotným návrhem. Druhý naopak na konci v průběhu testování, poté co byla služba nasazena na server.

B.1 Scénář pro prvotní rozhovor s uživateli

Výzkum provedený úplně na začátku ještě před samotným návrhem. Jeho cílem bylo zjištění, jak uživatelé jednotlivé služby používají a sběr požadavků na nový systém. Postup:

1. Představení mě, aplikace, proč ji dělám.
2. Informování o aplikaci Timer2Ticket (T2T). Co by měla dělat, proč ji chceme, jak ji využijeme.
3. Krátké info o dotazovaném: kdo, věková skupina, zda je technicky znalý, jak by využil aplikaci.
4. Jak uživatel aplikace používá (Redmine a hlavně Toggl Track)?
5. Jak by si uživatel představoval synchronizační aplikaci, jak by mohla fungovat. Používal by ji? Čím by zjednodušila život a práci.
6. Příklad prvního nastavení a konfigurace T2T (registrace systémů Toggl Track a Redmine):
   a) vytvoření účtu u T2T,
   b) zadání API klíčů služeb.
7. Příklad konkrétního sestavení konfigurace pro mapování časových záznamů. Jednoduché sestavení – projekt, na něm záznam.
8. Mapování objektů:
   a) projekty,
   b) požadavky,
   c) komentář.
9. Co s Redmine aktivitou? –> přes tagy?
10. Pokročilé funkce, atributy, tagy, atp.
11. Možnosti synchronizace – „real-time“, denně, týdně…

12. Synchronizace jednotlivých částí, manuální synchronizace (T2T – tlačítko synchronizovat vše nyní).
13. Notifikace uživatelů – např. API requesty na jednu stranu nefungují, expirovaný API klíč, atd. – log chyb v aplikaci a asi i e-mailem?
14. Shrnutí procesů a aplikace.
15. Další připomínky, poznámky. Pokud se v budoucnu objeví nějaká připomínka, neváhat se ozvat – není problém udělat dodatkovou diskuzi.
16. Poděkování, zdůraznění, že čas strávený rozhovorem bude zužitkován při návrhu a implementaci aplikace.

B.2 Scénář pro rozhovor s uživateli systému v průběhu testování

Průzkum proveden poté, co byl systém nasazený. Jeho cílem je sběr zpětné vazby od uživatelů a zjištění, zda je systém využitelný. Od průzkumu se bude vyvíjet další vývoj a bude provedena analýza možných rozšíření a vylepšení systému. Postup:

1. Úvod.
2. Jak dlouho dotazovaný Timer2Ticket (T2T) používá pro synchronizaci.
3. Jak hodnotí jednoduchost registrace a průchodu konfigurace.
4. Náročnost a počet úkonů, které musel provést před přechodem na T2T synchronizaci (či těsně po) ve službách Toggl Track a Redmine. (tedy byl přechod hladký?)
5. Který systém uživatel primárně využívá? (Toggl Track / Redmine)
6. Jak hodnotí samotný přechod, stav před a po:
   - Musel výrazně změnit styl vykazování (v návaznosti na bod výše v jeho preferované službě)?
   - Zjednodušilo to práci a stálo to za přechod?
7. Podařilo se eliminovat nudnou nezáživnou činnost na minimum? Nebo přibyla nová, která je také (dle dotazovaného) neužitečná. Případné příklady.
8. Využívá uživatel manuální sync, nebo vše přenechává na automatice? (pokud manuální sync využívá, tak v jakém případě)
9. Objevily se nějaké dodatky, které by velmi rád viděl ve službě?
10. Nějaké další připomínky a náměty?
11. Poděkování za čas dotazovaného a zakončení.

C Obsah přiloženého média

- `readme.txt` ... stručný popis obsahu média
- `src`
  - `impl` ... zdrojové kódy implementace
    - `timer2ticket-core` ... zdrojové kódy implementace části Core
    - `timer2ticket-api` ... zdrojové kódy implementace části API
    - `timer2ticket-client` ... zdrojové kódy implementace klientské části
  - `attachments` ... doplňující části, na které je odkazováno v textu
    - `client-design` ... exportované obrazovky z návrhu klientské části
    - `client-final` ... snímky obrazovky z nasazené webové aplikace
  - `thesis` ... zdrojová forma práce ve formátu LaTeX
- `text` ... text práce
  - `dp_stefan_vit_2021.pdf` ... text práce ve formátu PDF
  - `survey-notes` ... obsahuje mnou psané poznámky z dotazníků (anonymizované)