VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SOFTWAROVÁ PODPORA PROJEKTOVÉHO ŘÍZENÍ

SOFTWARE SUPPORT FOR PROJECT MANAGEMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. LUKÁŠ HABARTA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ŠÁRKA KVĚTOŇOVÁ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Habarta Lukáš, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Softwarová podpora projektového řízení**
Software Support for Project Management
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s problematikou řízení projektů. Zaměřte se na procesy plánování a realizace týmových projektů.
2. Seznamte se s komerčními i volně dostupnými systémy pro podporu plánování a řízení projektů. Proveďte jejich analýzu a celkové zhodnocení.
3. Na základě získaných poznatků (výhody a nevýhody) navrhněte vlastní systém pro správu projektů.
4. Navržený systém prakticky realizujte a prověřte jeho funkčnost na vhodně zvoleném vzorku dat, který pokrývá celou šíři oblasti. Musí umožňovat správu projektů různého charakteru, rozsahu, včetně správy všech zainteresovaných stran projektu (zadávání členů týmu, přiřazování k jednotlivým úkolům atp.).
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího rozvoje.

Literatura:

- Rosenau, M.D.: Řízení projektů, Computer Press, 2003, 344 s. ISBN 80-7226-218-1
- Kliem, L. R.: Project Management Methodology. Marcel Dekker Inc., 1997.
- Schwalbe, K.: Řízení projektů v IT - Kompletní průvodce, Computer Press, 2007, ISBN: 978-80-251-1526-8
- Schulte, P.: Complex IT Project Management. AUERBACH PUBLICATION, 2004, ISBN 0849319323

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Květoňová Šárka, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Tématem této práce je řízení projektu a popis programového řešení pro podporu řízení projektu, které bylo v rámci práce implementováno. První část obsahuje popis základních pojmů z oblasti řízení projektu a popis samotného řízení projektu. V druhé části práce je popsána technická analýza řešení a samotná implementace webové aplikace pro podporu řízení projektu.

Abstract

The topic of this thesis is project management and description of software solution for project management support. This software solution was also implemented within the thesis. The first part contains a description of the terminology used in project management. There is described project management as well. The second part contains a description of technical analysis and implementation of the web application for project management.

Klíčová slova

Řízení projektu, projekt, scrum, software pro řízení projektu

Keywords

Project management, project, scrum, software for project management

Citace

HABARTA, Lukáš. *Softwarová podpora projektového řízení*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Šárka Květoňová, Ph.D.

Softwarová podpora projektového řízení

Prohlášení

Prohlašuji, že jsem tuto diplomou práci vypracoval samostatně pod vedením paní Šárky Květoňové. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Habarta
20. května 2019

Poděkování

Rád bych poděkoval především Ing. Šárce Květoňové, Ph.D. za odborné vedení mé diplomové práce a ochotu při výběru tématu.

Obsah

Kapitola 1

Úvod

Cílem této diplomové práce je implementace nástroje, který může sloužit jako software pro podporu řízení projektů v podniku. Zaměření výsledné aplikace je hlavně na oblast informačních technologií, konkrétně na vývoj softwaru, jelikož tato oblast je mi blízká a znám její požadavky. Ostatní oblasti mají diametrálně odlišné požadavky pro správu projektů, je téměř nemožné využít jednu a tu samou aplikaci při řízení projektu v oblasti vývoje softwaru a například v průmyslu při výrobě automobilu.

Řízení projektu je jednou z klíčových oblastí pro úspěšné dokončení vývoje softwaru, kde se na vývoji podílí alespoň jeden tým s 3 a více členy. Při nižším počtu členů je pokročilé řízení projektu kontraproduktivní.

Práce je rozdělena do tří samostatných tématických celků. První část práce se bude zabývat teoretickým popisem řízení projektů a vlivem specifického softwaru na toto řízení. V rámci teoretického popisu řízení projektu bude využita i osobní zkušenost z praxe. Konkrétně se jedná o náhled na řízení projektu a zkušenosti se softwarem pro řízení projektů na úrovni středně velkého podniku či nadnárodní korporace. Osobní zkušenost bude mít velký vliv na jednotlivé pasáže této práce. Dále si zde definujeme některé základní pojmy nutné pro pochopení problematiky. Mezi pojmy, které si v další kapitole vysvětlíme, patří **projekt**, **řízení projektu**, **tradiční přístup**, **agilní přístup**, **agilní metodiky v oblasti vývoje softwaru**.

Ve druhé části se podíváme na důvody použití softwaru, jaké jsou jeho klady a zápory. Dále se zde zaměříme na existující řešení, jejich rozbor a zdůraznění nedostatků těchto aplikací. Každá aplikace se totiž zaměřuje na jiné aspekty řízení projektů. Konkrétně se zde zaměříme na tři relativně rozšířené aplikace, které jsem zvolil jako vzorek pro porovnání.

Poslední část se bude zabývat samotným řešením aplikace. Je zde vysvětleno, na jakých základech je aplikace postavena ať už z pohledu technologií nebo z pohledu formálních požadavků, jak tyto požadavky vznikly, jak se je nakonec podařilo implementovat a jaký je jejich dopad na celkové fungování aplikace a její funkcionalitu. Dále si zde ještě ukážeme vývoj a testování této aplikace. Poslední část bude taky sebereflexe ohledně funkcionality a zhodnocení, co aplikace neobsahuje, co by bylo vhodné upravit a jak poskytnout kvalitnější řešení.

Kapitola 2

Řízení projektu

Na začátku této práce si musíme předat několik základních informací o pojmech využívaných v definici řízení projektu, jaký je dopad řízení projektu v běžném životě a kde se s řízením projektu můžeme setkat.

2.1 Základní pojmy

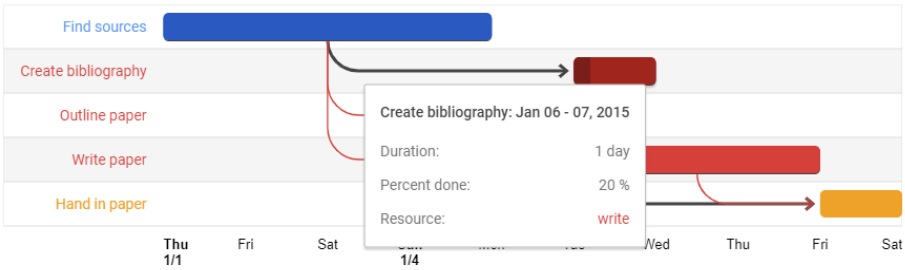
K porozumění tomuto textu je nutné si nejdříve definovat základní pojmy, které budou častou používány [4]. První pojem, nalezneme již v samotném názvu této práce. Co je to tedy **projekt**? *Projekt je chápán jako jedinečná, dočasná, multidisciplinární a organizovaná snaha o realizaci dohodnutých výstupů (dodávek) z předem definovaných požadavků a omezení.* [6] Dohodnutým výstupem je nějaký **produkt**. *Produkt je konkrétní, pojmenovaný výsledek činnosti, etapy, projektu. Produktem může být i provedení služby.* [6] Do činností při vytváření produktu nám mohou zasáhnout vnější vlivy, které můžeme označit jako rizika. *Riziko je nejistá událost nebo podmínka, která, pokud nastane, má negativní nebo pozitivní vliv na dosažení cíle projektu.* [6] Zbývá nám jeden poslední důležitý pojem, a tím je **proces**. *Proces můžeme definovat jako na sebe vzájemně navazující posloupnost činností, jež se opakují.* [7] Nyní máme definovány všechny potřebné pojmy, můžeme se tedy podívat na to, co rozumíme pod pojmem **projektové řízení** nebo-li **projektový management**.

Řízení projektů je možné chápat jako časově ohraničenou a ucelenou sadu činností a procesů, jejímž výsledkem je určitý produkt. Řízení projektů se snaží maximálně zefektivnit tyto činnosti tak, aby bylo dosaženo minimálních nákladů, maximální efektivity práce a zisku. Dále slouží k tomu, aby projekt byl naplánován tak, aby jej bylo možné stihnout v plánovaném termínu za splnění předem definovaných požadavků a specifikací. Zjednodušeně řečeno nám řízení projektu umožňuje vytvořit produkt podle specifikací ve stanoveném čase s minimálními náklady a maximálním ziskem.

2.2 Grafické znázornění

Samotné řízení projektu je ohraničeno určitým časovým rámcem, který je definován začátkem projektu, a termínem dodání. Každý proces či činnost je svázaná s časem, máme stanovený začátek a konec nebo délku trvání činnosti bez specifikování začátku a konce. Díky tomu se nám nabízí možnost grafického zobrazení posloupnosti činností. Toho si byl vědom i **Henry Laurence Gantt**, který dal vzniknout grafickému znázornění nazývaného **Ganttův diagram**. Ganttův diagram je dvoudimenzionální diagram, kde se na vertikální

ose zobrazuje seznam činností a na horizontální ose je zobrazen čas. Plochu diagramu vyplňuje zobrazení trvání jednotlivých činností včetně návazností.



Obrázek 2.1: Ukázka Ganttova diagramu. [3]

Použití Ganttova diagramu v procesu řízení projektu není nikde přesně definováno, ale nejčastěji se využívá pro plánování návaznosti v rámci několika projektů či k plánování aktivit v rámci konkrétního projektu. Ganttův diagram je častou součástí pokročilých softwarových řešení pro podporu řízení projektu. V jeho složitější variantě je možné jej využít třeba pro zobrazení *kritické cesty*.

2.2.1 Metoda kritické cesty

Jedná se o základní metodu síťové analýzy [2]. V této metodě se využívá síťového diagramu pro nalezení kritické cesty v rámci projektu. Každý projekt má alespoň jednu kritickou cestu. Kritickou cestu můžeme jednoduše definovat jako nejdelší možný průchod v grafu z počátečního stavu do koncového stavu. První úkol v kritické cestě značí začátek projektu a konec posledního úkolu značí ukončení projektu. Jakékoliv vybočení z očekávaného časového rámce úkolu na kritické cestě se tedy logicky promítne do plánovaného konce tohoto projektu. Pokud se jeden úkol zpozdí, bude zpožděn následně celý projekt. Toto platí ovšem i obráceně, pokud stihneme úkol dodat dříve, tak dokončíme i projekt dříve.

2.2.2 PERT

Metoda PERT (Program Evaluation and Review Technique) je další z metod síťové analýzy [2] [5]. PERT je zobecnění metody kritické cesty. V této metodě se využívá výpočtu pravděpodobnosti k zjištění délky trvání projektu. Doba trvání úkolu je v PERT brána jako náhodná proměnná s určitým pravděpodobnostním rozložením. Dobu trvání tedy vyjadřujeme jako pravděpodobnost dokončení daného úkolu v daný čas. Pro výpočet rozložení pravděpodobnosti se nejvíce osvědčilo rozdělení beta. Pro beta rozdělení se využívá odhadů expertů v daném oboru na danou činnost. Jednotlivé odhady se dělí na tři typy:

- Optimistický odhad - značí časový odhad při hladkém průběhu celého úkolu, bez komplikací
- Nejpravděpodobnější odhad - jedná se o střízlivý odhad se zahrnutím možných komplikací
- Pesimistický odhad - zde se do odhadu promítají všechny možné komplikace, které mohou nastat

Tyto odhady se následně využijí k výpočtu nejpravděpodobnějšího trvání projektu. Často se využívá vzorec, který klade největší důraz na nejpravděpodobnější odhad: $T = \frac{t_o+4t_n+t_p}{6}$. t_o značí optimistický odhad, t_n značí nejpravděpodobnější odhad a t_p značí pesimistický odhad.

2.3 Přístupy k řízení projektu

Řízení projektu je činnost, bez které se žádný větší projekt nemůže obejít, je tedy nedílnou součástí téměř všech odvětví, která vytvářejí nějaký produkt. Proto vznikla potřeba nadefinovat si základní pravidla a postupy. Tyto postupy a pravidla se pak aplikují na různé projekty s různými produkty, ať už se jedná o součástky, auta či software. Dále je možné tyto postupy aplikovat na odvětví, jež jsou založena na toku událostí - týmové vytváření dokumentací, technických zpráv, posudků, ...

Řízení projektu je součástí běžného života všech lidí. Lidé se mohou podílet buď aktivně (přímo), nebo pasivně (nepřímo). Všichni jsme totiž součástí na sebe navazujících událostí a komplexních projektů, aniž si to možná uvědomujeme. Kdo si to uvědomuje, může zaslepeně uvažovat jen o řízení projektů v rámci zaměstnání, kde je součástí jak management, tak i například dělník ve výrobě. V této rovině se bavíme o abstraktní hierarchii, kde každá nižší úroveň této hierarchie je více odstíněna od detailů a zaměřuje se jen na určitou část, respektive činnost nutnou k dosažení cíle, který stanovila vyšší úroveň v hierarchii. Zaměstnání není jediným místem, kde dochází k řízení projektů. Všichni se můžeme považovat za vstupy do globálnějšího řízení, kde probíhá plánování projektů na úrovni států, kontinentů či celé Země. Může se jednat například o plánování státního rozpočtu, kde finanční podíl je určen počtem lidí žijících v dané oblasti. Mezi dalšími známými kontinentálními či globálními projekty můžeme nalézt třeba **NATO**, **Evropskou unii**, **UNESCO**, ...

Záměrně jsem v posledním odstavci odbočil od tématu této sekce, abych ukázal jiný pohled na projekty. Projekty nemůžeme považovat za něco malého na úrovni podniku, škol či institucí. Existují totiž i projekty, které mají globální dopad. Jak jsem ukázal, není možné definovat něco jako typický či univerzální projekt, nemůžeme tedy mít ani unifikovaný přístup k řízení projektů. Každý projekt je jedinečný svým vlastním způsobem, ať už se jedná o projekty napříč různými odvětvími či projekty v rámci jednoho podniku.

Pokud se chceme rozhodnout, jak budeme přistupovat k řízení projektu, máme na výběr ze dvou možností, které jsou pevně definované.

- Tradiční přístup
- Agilní přístup

Jedná se ale o poměrně novou možnost volby. Za základ tradičního přístupu by se dal považovat projekt Manhattan vedený americkou armádou v období 1942 - 1946 za účelem vývoje nové generace zbraní (konkrétně se jednalo o atomovou bombu)[8]. Na tomto projektu bylo potřeba koordinovat a korigovat obrovské množství lidí, profesí a subjektů. Zároveň ale bylo potřeba dodat každému jen informace nezbytné pro jeho práci, aby nedošlo k vyzrazení záměru tohoto projektu. U agilního přístupu můžeme říct, že vznikl až v 21.století. Pojďme se tedy blíže podívat na zmíněné přístupy.

2.3.1 Tradiční přístup

Tradiční přístup k plánování projektů je vhodný převážně pro projekty, které vyžadují pečlivé naplánování před samotným začátkem realizace. Jedná se o projekty, kde jsou předem

známé jasně dané specifikace a požadavky, které se během realizace nemění. Jde například o oblast stavebnictví, kde projektová dokumentace stavby prochází schvalováním na několika úřadech a její vypracování je časově náročné a není tedy možné, aby zákazník (zadavatel projektu) měnil požadavky během samotné realizace. Nemůže například změnit rozlohu stavby či použitý materiál, všechny tyto změny musí být znovu přepracovány do projektové dokumentace. Pokud by k takové změně došlo, proběhlo by znovu plánování celého projektu. Součástí takového plánování je většinou i koordinace více subjektů jako jsou dodavatelé, externí firmy, úřady atd.

Tradiční přístup se skládá z pěti fází projektu, které můžeme nazvat ŽIVOTNÍM CYKLEM PROJEKTU:

- **Iniciace** - Zjistili jsme, že potřebujeme naplánovat projekt, ověříme nezbytné skutečnosti, které potřebujeme pro samotné plánování. Např. existuje poptávka po tomto produktu? Bude tento produkt generovat zisk? A další otázky.
- **Plánování a návrh** - Víme, že produkt splní naše očekávání, je potřeba naplánovat jeho výrobu. Zahrnujeme zde spoustu dalších subjektů. Kdo dodá materiál? Kdo zajistí připojení elektrické přípojky? Kdo vypracuje projektovou dokumentaci?
- **Realizace** - Jedná se o samotnou realizaci projektu, během níž vytváříme cílový produkt.
- **Kontrola** - Sledujeme, zda je produkt realizován podle stanovených plánů. Zda se plní všechny specifikace, normy, časové odhady a pracovní postupy.
- **Uzavření** - Samotné dokončení produktu, předání zákazníkovi, dodání potřebné dokumentace.

V případě *plánování a návrhu* můžeme vidět, že ne všichni mají všechny informace o projektu. Dodavatel materiálu nepotřebuje vědět, jak vypadají kompletní plány stavby. Pro něj se jedná pouze o objednávku. Dalším aspektem tradičního přístupu je tedy omezené sdílení informací. Množství informací klesá směrem od managementu k zaměstnancům.

Po uvedení jednotlivých fází můžeme vidět, že tradiční přístup není úplně ideálním řešením pro vývoj softwaru. Pouhý zlomek projektů ve vývoji softwaru je úplně zadefinován již při zadávání. Většinou potřebujeme pružně reagovat na změnu požadavků od zákazníka. Použití nových technologií, změna zadání z důvodu změny situace na trhu atd.

2.3.2 Agilní přístup

Agilní přístup je v jistém slova smyslu opakem tradičního přístupu. Agilní přístup je založen na postupném doplňování specifikace a požadavků. Zákazník (zadavatel projektu) je zde jeden z klíčových faktorů, který vstupuje do každé fáze realizace na rozdíl od tradičního přístupu, kde je zákazník součástí pouze **iniciace** a **uzavření**. Interakcí se zákazníkem se postupně upřesňuje produkt do finální podoby. Zákazník interaguje s výstupem aktuální iterace a přidává poznámky a poznatky k následující iteraci. Při využití analogie k diamantovému brusu můžeme říct, že výsledkem první iterace je surový diamant, který se s každou další iterací přibližuje svému požadovanému tvaru.

Agilní přístup našel největší uplatnění v oblasti vývoje softwaru. Zde je totiž potřeba pružně reagovat na požadavky zákazníka, respektive trhu. Při vývoji softwaru není téměř možné využít tradiční přístup. Od tohoto bodu se tedy budu zaměřovat čistě na oblast vývoje softwaru. Agilní přístup vznikl v reakci na chaotický vývoj projektů v 80. a 90.

letech 20. století. Velké množství projektů v tomto období končilo neúspěchem. Projekty se neúměrně prodlužovaly a prodražovaly, takže často docházelo k předčasnému ukončení projektu z důvodu nedostatku financí či zániku původní potřeby produktu.

Základem pro vznik a pojmenování agilního přístupu je *Manifest agilního programování*. Autory manifestu jsou Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Biran Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland a Dave Thomas. Tito autoři manifestu agilního programování měli velké zkušenosti z řízení projektů, firem či publikování článků v oblasti vývoje softwaru. Například **Mike Beedle** je zakladatel CEO společnosti zaměřující se na konzultační činnost v oblasti vývoje aplikací. **Ward Cunningham** je považován za jednoho z otců *extrémního programování*. **Martin Fowler** je autorem knih jako *Analysis patterns, UML Distilled, Planning Extreme Programming*. Jedná se tedy o odborníky na vývoj softwaru, řízení projektů a návrh softwaru.

Manifest agilního programování definoval 12 principů, kterými je potřeba se při agilním vývoji řídit: [1]

- Naší nejvyšší prioritou je vyhovět zákazníkovi časným i průběžným dodáváním softwaru.
- Vítáme změny v požadavcích, a to i v pozdějších fázích vývoje. Agilní procesy podporují změny vedoucí ke zvýšení konkurenceschopnosti zákazníka.
- Dodáváme fungující software v intervalech týdnů až měsíců s preferencí kratší periody.
- Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu.
- Budujeme projekty kolem motivovaných jednotlivců. Vytváříme jim prostředí, podporujeme jejich potřeby a důvěřujeme, že odvedou dobrou práci.
- Nejúčinnějším a nejefektivnějším způsobem sdělování informací vývojovému týmu z vnějšku i uvnitř něj je osobní konverzace.
- Hlavním měřítkem pokroku je fungující software.
- Agilní procesy podporují udržitelný rozvoj. Sponzoři, vývojáři i uživatelé by měli být schopni udržet stálé tempo trvale.
- Agilitu zvyšuje neustálá pozornost věnovaná technické výjimečnosti a dobrému designu.
- Jednoduchost - umění maximalizovat množství nevykonané práce - je klíčová.
- Nejlepší architektury, požadavky a návrhy vzejdou z týmů, které se samy organizují.
- Tým se pravidelně zamýšlí nad tím, jak se stát efektivnějším, a následně koriguje a přizpůsobuje své chování a zvyklosti.

Dále manifest definuje čtyři různé oblasti, kde uvádí protiklady. Cílem je zdůraznit potřebu hodnot na levé straně před hodnotami na straně pravé. Jedná se o následující oblasti:

- Jednotlivci a interakce vs procesy a nástroje

- Fungující software vs obsáhlá dokumentace
- Spolupráce se zákazníkem vs definice obsáhlého kontraktu
- Reagování na změny vs striktní dodržení původního plánu

Jednotlivci a interakce vs procesy a nástroje

Manifest zdůrazňuje, že jednotlivci dosahují menších výsledků než interagující jednotlivci v týmu. Tento tým ale nesmí mít striktně definované procesy a nástroje. Manifest nijak nepopírá potřebu procesů a nástrojů. Pouze zdůrazňuje, že je potřeba, aby si každý tým vybíral procesy a nástroje sám podle vlastní potřeby.

Fungující software vs obsáhlá dokumentace

Dokumentace je považována za důležitou součást dokončení projektu. Nemůžeme ale stavět obsáhlou dokumentaci nad funkční verzi softwaru. Funkční verze softwaru má nejvyšší prioritu. Dokumentace je v agilním přístupu chápána pouze jako nezbytná součást dokumentující části, jež nejsou intuitivní nebo dobře pochopitelné.

Spolupráce se zákazníkem vs definice obsáhlého kontraktu

V agilním přístupu je kontrakt (smlouva) brána pouze jako rámec, který vymezuje nezbytné detaily. Zbytek by měl být řešen operativně se zákazníkem mimo rozsah kontraktu. Díky tomu je možné dosáhnout výhodné spolupráce pro obě strany.

Reagování na změny vs striktní dodržení původního plánu

Plány jsou pro agilní přístup pouze hrubou osnovou, která slouží jako vodítko. Agilní přístup je totiž založen na tom, že zákazník své požadavky postupně mění a upřesňuje. Je tedy potřeba měnit i plány, proto nemohou být pevně dané.

2.4 Agilní metodiky v oblasti vývoje softwaru

Pro agilní vývoj softwaru existuje množství metodik, která specifikují vlastní pravidla, ale zároveň dodržují principy z manifestu agilního programování. Metodikou rozumíme soubor doporučených postupů a praktik, které využíváme během celého životního cyklu vývoje.

V této práci uvedu konkrétně pouze jednu metodiku, která je z mého pohledu a podle průzkumu trendů nejpoužívanější agilní metodika.

- **Scrum**
- Kanban
- Crystal
- Extrémní programování (XP)
- Vývoj řízený testy - známější pod zkratkou TDD (zkratka anglických slov Test Driven Development)
- Lean development

Metodika Scrum bude uvedena pouze zjednodušeně, umožní totiž dokreslit pozadí samotné implementace aplikace. Zjednodušený popis je nutný, jelikož metodika je skutečně obsáhlá a pro její kompletní popis by byla potřeba samostatná práce. Pro zvídavého čtenáře je dostupné velké množství dodatečného materiálu, a to ve formě webových stránek nebo knih. Dále bych doporučil podívat se na metodiku TDD, jedná se totiž o velmi zajímavou metodiku, která ale bohužel není kvůli svým nárokům příliš používána.

2.4.1 Scrum

Byť jsem Scrum zařadil mezi metodiky agilního přístupu, tak se ve skutečnosti nejedná o metodiku. Jedná se pouze o procesní nástroj nebo framework, který nám dává osnovu a postupy pro řízení jednotlivých procesů projektu.

Scrum je asi nejrozšířenější agilní přistup využívaný primárně při vývoji softwaru. Scrum preferuje ponechání zodpovědnosti za řešení problému na týmu místo předem kompletně definovaného popisu, jak bude co uděláno a kdo má jakou část projektu na starost. Daný tým totiž ví nejlépe, jak si rozdělit práci.

Pro práci ve Scrum se používá definovaný časový úsek označovaný jako **sprint**. Sprint je opakující se časový blok, který bývá několikadenní až několikatýdenní. Zřídka kdy se vyskytují jednodenní sprinty nebo sprinty delší jak měsíc. Nejčastější jsou čtrnáctidenní sprinty. Plánování sprintu vždy probíhá v předcházejícím sprintu. Během plánování sprintu se preferuje dodání seznamu požadavků funkcionalit místo úplného popisu úkolů, akceptačních kritérií atd.

Scrum tým je samostatná organizační jednotka. Pod pojmem samostatná organizační jednotka rozumíme to, že si daný tým vše plánuje, rozhoduje a rozděluje sám bez nutnosti nějakého týmového vedoucího. Tým si tedy sám kolektivně rozhodne, kdo bude kterou funkcionalitu dělat a jak bude daná funkcionalita vyřešena. Tým dostane pouze seznam požadavků, co má být uděláno, proto musí vytvořit vše od návrhu řešení až po samotnou implementaci.

Žádný tým nemůže fungovat správně bez hierarchie zodpovědností, proto je potřeba mít v týmu dvě podpůrné role. První je tzv. **Scrum Master**, druhá je **vlastník produktu - (anglicky Product owner)** někdy označovaný jako projektový manažer. Roli Scrum master může zastávat přímo někdo z vývojářů, ale u vlastníka produktu se to nedoporučuje. Scrum master se stará o správné fungování týmu, oddělení členů týmu od byrokracie a vede tým k dodržování zásad Scrum. Vlastník produktu představuje někoho z byznysu nebo přímo zákazníka a má za úkol vést tým k dodání správného produktu.

Plánování sprintu

Jak bylo již zmíněno, plánování sprintu probíhá vždy v předchozím sprintu. Plánování sprintu probíhá nadefinováním požadavků od vlastníka produktu na novou funkcionalitu. Požadavky je potřeba mít kvalitně sepsané, protože tým musí jasně vědět, co bude jeho cílem při vývoji. Následuje mítink, kterého se účastní celý tým včetně vlastníka produktu. Tento mítink se označuje jako GROOMING. Na tomto mítinku se prochází jednotlivé funkcionality rozdělené na samostatné úkoly. Úkoly by měly být přiměřeně náročné. Pokud máme 14denní sprint, nemůže nám úkol trvat měsíc. Optimální délka úkolu je závislá na délce sprintu, při 14denním sprintu je to od půl dne do tří dnů. Tým zde diskutuje o složitosti řešeného problému, technickém zázemí nutném k řešení a zároveň se zde určuje časová náročnost jednotlivých úkolů. Časová náročnost je nezbytná pro plánování sprintu. Seznam úkolů obsažených ve sprintu musí odpovídat časovým možnostem týmu. Pokud má tým 4 členy

a máme 14denní sprint, tak potřebujeme naplánovat práci na 40 pracovních dní. Reálně to bude méně, protože je potřeba počítat s ojedinělými událostmi, jako je třeba nemoc a dále je potřeba započítat čas, který tým musí věnovat dalším mítinkům a plánování. Při časové náročnosti je nejčastěji využíváno dvou jednotek. První jednotkou je člověkoden, druhá jednotka je abstraktní. Celý sprint si oceníme na určitý počet abstraktních bodů (anglicky story points), které jsme schopní za sprint splnit. Následně se jednotlivé úkoly ohodnotí počtem bodů, které odpovídají jejich složitosti. Nevýhodou tohoto abstraktního ohodnocení je to, že body by neměly být přepočítávány na člověkoden. Pro hodnocení v člověkodnech i bodech se používá upravená Fibonacciho posloupnost. Fibonacciho posloupnost je *1, 1, 2, 3, 5, ...* nebo-li další člen je součtem dvou předchozích členů. Ve Scrum se používá posloupnost 0.5, 1, 2, 3, 5, ..., nejmenší jednotkou je tedy 0.5 dne/bodu.

Práce ve sprintu

Výstupem grooming mítinku a plánování sprintu je seznam úkolů, které je potřeba v daném sprintu udělat, otestovat a integrovat s již existující funkcionalitou. Tomuto seznamu úkolů se říká **backlog**. Backlog je shromaždiště nepřidělených úkolů, které si jednotliví vývojáři berou a začínají na nich pracovat. Každý den ráno se celý tým včetně vlastníka produktu sejde na krátkém mítinku. Tento mítink se označuje jako denní Scrum mítink (anglicky Daily Scrum meeting). Na tomto mítinku každý z členů řekne, co dělal předchozí den, jestli narazil na nějaký problém, který ho blokuje, a případně jak daný problém řešit. Dále zde oznámí, co bude jeho náplní tohoto dne. Tímto způsobem probíhá každý den sprintu. Sprint je pak zakončen prezentací dokončených úkolů vlastníkovi produktu, který poté poskytne zpětnou vazbu na danou funkcionalitu. Zda odpovídá jeho požadavkům a případně, co je potřeba změnit. Z případných požadavků na změnu se vytvoří další úkoly, které se na dalším grooming mítinku opět proberou a zařadí do backlog.

Tímto můžeme považovat základní popis metodiky Scrum za dokončený. Provedeme si ještě shrnutí zmíněných informací. Scrum se tedy opírá o PROJEKT, který se skládá z jednotlivých funkcionalit. Každá funkcionalita je rozdělena do přiměřeně velkých ÚKOLŮ a následně ohodnocena určitým ČASOVÝM ODHADEM náročnosti. Projekt je realizován konkrétním TÝMEM. Každý úkol je samostatnou prací JEDNOHO Z ČLENŮ týmu. Úkoly jsou součástí ČASOVÉHO ÚSEKU zvaného sprint. O specifikaci požadavků se stará VLASTNÍK PRODUKTU (PROJEKTOVÝ MANAŽER). Povšimněte si zvýrazněných pojmů, tyto pojmy budou velmi důležité při popisu samotné implementace. Poprosím tedy o jejich zapamatování.

Kapitola 3

Analýza stávajících řešení

Pro řízení projektů existuje nespočet softwarových řešení. Každé řešení je svým způsobem unikátní. K řízení projektu je možno přistupovat i jiným způsobem, než který byl zatím zde popsán. Je možné se zaměřit například na analýzu ceny, nákladů či rizik. Při výběru softwarových řešení k analýze jsem volil podle několika kritérií. Prvním kritériem bylo rozšíření dané aplikace mezi podniky a její popularita. Dalším kritériem bylo, aby zvolené aplikace byly rozdílné a zaobíraly se jiným přístupem k řízení projektu. Posledním a řekněme, že asi i nejdůležitějším kritériem je její dostupnost, abych měl možnost jejího praktického vyzkoušení. Po aplikaci těchto kritérií zůstaly tři možnosti:

- Microsoft Team Foundation Server
- JIRA
- Easy Project

Z těchto možností jsem se rozhodl vyřadit Microsoft Team Foundation Server, jelikož jeho funkcionalitu pokrývá JIRA a navíc neposkytuje tak rozsáhlé možnosti řízení projektů. Řízení projektů je tady pouze jako podpůrný nástroj, a primárním účelem je verzování zdrojových kódů týmu, které je možné integrovat přímo do vývojového prostředí Visual Studio.

3.1 JIRA

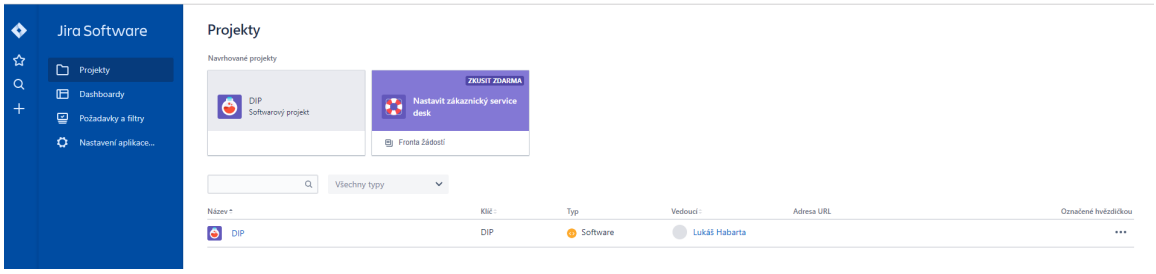
JIRA je nejrozšířenější softwarové řešení pro řízení projektu v agilním prostředí. Jelikož s JIRA pracuji každý den, tak bylo logické ji zařadit jako jeden z nástrojů, který bude analyzován a hodnocen. Jak bylo zmíněno, k řízení projektu se dá přistupovat více způsoby. JIRA je vhodný nástroj, pokud řízení projektu odpovídá tomu, co bylo napsáno v této práci. Tedy jde o řízení projektu, kde je primární složkou úkol a jeho vazba na projekt, tým a plánování sprintu.

JIRA obsahuje obrovské množství vestavěných funkcí, které z ní dělají flexibilní nástroj, jenž se přizpůsobí téměř všem požadavkům projektových manažerů. Pokud by přece jen nebyly požadavky splněny, je možné JIRA rozšířit pomocí externích aplikací nebo zásuvných modulů.

Pro porovnání jsem využil dočasného účtu na JIRA cloud, kde jsem vytvořil projekt s úkoly. Úkoly zařadil do backlog a postupně jsem provedl plánování sprintu a vytvoření tabule.

3.1.1 Funkcionalita

Výčet všech funkcionalit v JIRA je téměř nemožný, jelikož je tento systém ve vývoji několik let a má obrovskou základnu zákazníků. Mezi základní a nejpoužívanější funkcionalitu můžeme zařadit správu projektu a úkolů s ním souvisejících.



Obrázek 3.1: Přehled projektů.

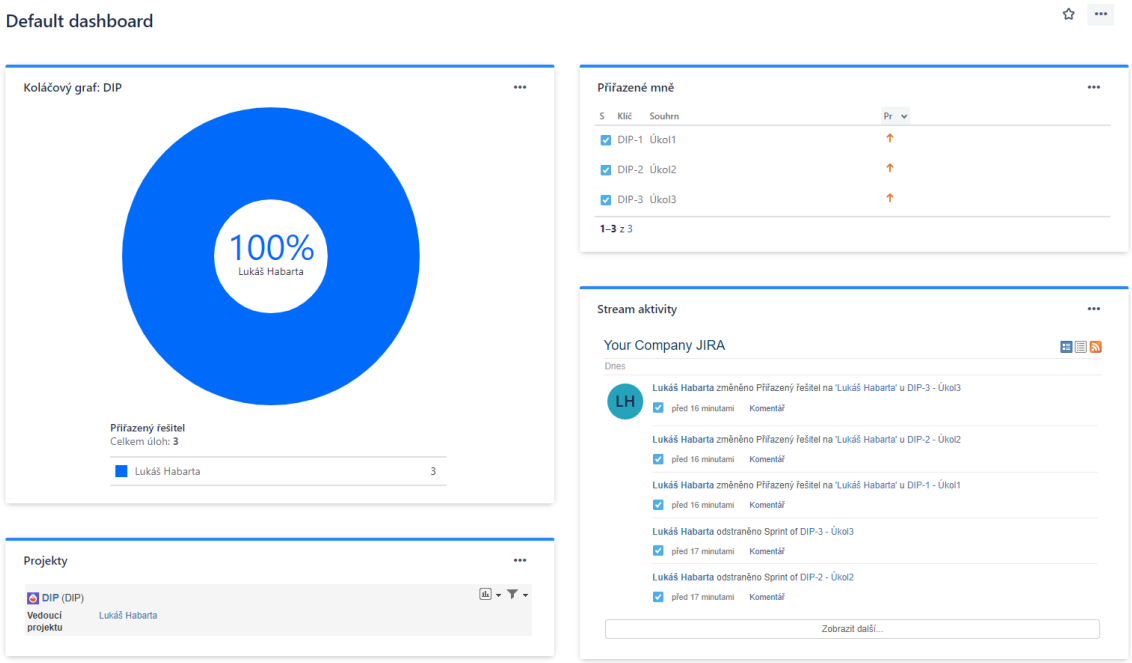
Na obrázku 3.1 můžeme vidět soupis projektů, které máme v aplikaci jako projektový manažer dostupné. Jedná se o jednoduchý a stručný přehled. Tyto projekty jsou přístupné i uživatelům, kteří mají přidělená dostatečná práva, případně se na projektu podílejí. Projekt *DIP* by tedy byl viditelný i pro uživatele, který má přiřazenou zodpovědnost za některý z úkolů v rámci toho projektu.

Další důležitou a často používanou položkou je *Dashboardy* (3.2). Tato stránka nám umožňuje vytváření vlastních nastavení pomocí speciálního dotazovacího jazyku *JQL - JIRA Query Language*. Je zde na výběr z třiceti předdefinovaných aplikací. Patří mezi ně například:

- grafy
  - čtyřrozměrný bublinkový graf pro zobrazení oblíbených požadavků
  - sloupcový graf s dobou řešení jednotlivých úkolů
  - koláčový graf, který umožňuje zobrazení například stavu úkolů
- statistiky
- podpůrná zobrazení pro řízení sprintu
- reportovací nástroje
  - zbývající práce ve sprintu
  - zobrazení předpokládaných datumů dokončení aktuálního sprintu v závislosti na stavu, počtu a předchozím trvání úkolů

Všechny zobrazené aplikace je možné upravit podle potřeby pomocí již zmíněného jazyka JQL. Dashboard je jednou ze součástí, která umožňuje rychlý náhled na aktuální stav projektu. Tento jazyk je bohužel pro lidi neznalé dotazovacích jazyků poměrně složitý. Pro skládání dotazů zde chybí grafické rozhraní, které by umožňovalo naklikat potřebné dotazy.

Pro detailnější náhled na projekt nám slouží tabule projektu (3.3). Tabule projektu obsahuje backlog se všemi úkoly, které jsou vytvořeny pro daný projekt. V backlog nenajdeme



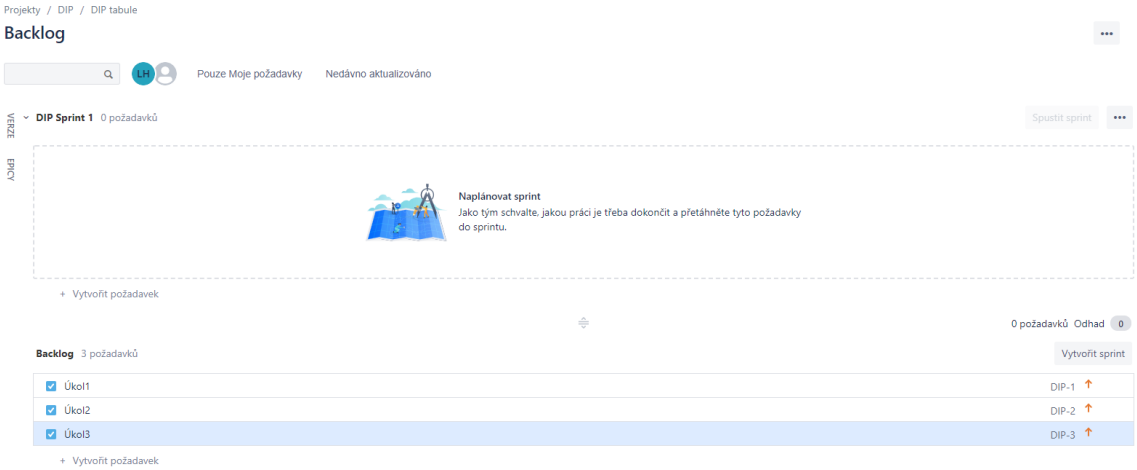
Obrázek 3.2: Dashboard

úkoly, které jsou součástí již naplánovaného sprintu. Jedná se tedy o úložiště úkolů, u kterých je ještě například potřeba doplnit požadavky, dodat odhady nebo prostě jen nebyly naplánovány.

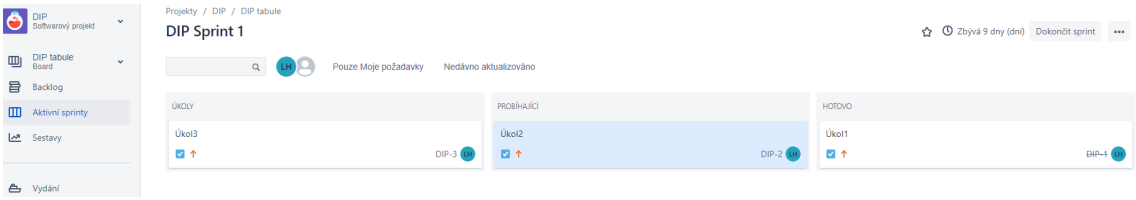
Ze zobrazeného backlog můžeme vytvořit nový sprint (3.4). Sprint se skládá z jednotlivých úkolů. Jediným nastavením sprintu je zde doba trvání (předdefinovaná doba je 2 týdny) a začátek sprintu. Po výběru úkolů a nastavení datumů se sprint s úkoly přesune do záložky *Aktivní sprinty*. Tato záložka je koncipována jako kanban. Tedy jednotlivé úkoly jsou zobrazeny ve sloupci a každý uživatel si může převzít zodpovědnost za některý z úkolů. Postupně u úkolů měníme jeho stav. Zde například je úkol označen jako probíhající a další jako hotový.

Pro zobrazení jednotlivých detailů úkolu je zde podrobný popis, který je dostupný přes odkaz na jméno úkolu (3.5). V základním nastavení vidíme, o jaký typ úkolu se jedná, kdo je za něj zodpovědný, v jakém stavu se nachází a několik dalších polí s upřesňujícími údaji. Ke každému úkolu je možné přidávat přílohy. Tyto přílohy mohou být například obrázky, textové dokumenty či kusy kódu. Pro přílohy zde není omezení. Přílohy je vhodné využívat třeba pro obrázky zobrazující chybu, která se má v daném úkolu opravit. Nebo mock-up náhled zobrazující očekávané grafické rozhraní. Poslední dolní část označená jako *Aktivita* obsahuje informace ohledně konkrétního úkolu. Jsou to komentáře, které uživatelé přidali k úkolu. Komentáře se využívají v JIRA k diskuzi nad úkoly. Komentáře umožňují označování dalších uživatelů, čímž dojde k zaslání notifikace na jejich registrační email, dále umožňují hypertextové odkazy napříč úkoly, přílohami i projekty. Další záložky této části obsahují informace o změně stavu, editaci popisu, zaznamenaném čase a další.

Seznam polí zobrazených na detailu úkolu se může zdát jako nedostatečný. JIRA proto umožňuje dodefinování vlastních polí pro jednotlivé typy úkolů. Tyto pole mohou být textová nebo předdefinovaný seznam povolených hodnot (výčet povolených hodnot). Je možné nastavit pole jako povinné, bez jehož vyplnění není možné úkol vytvořit. Při reálných pro-



Obrázek 3.3: Detail projektu



Obrázek 3.4: Sprint

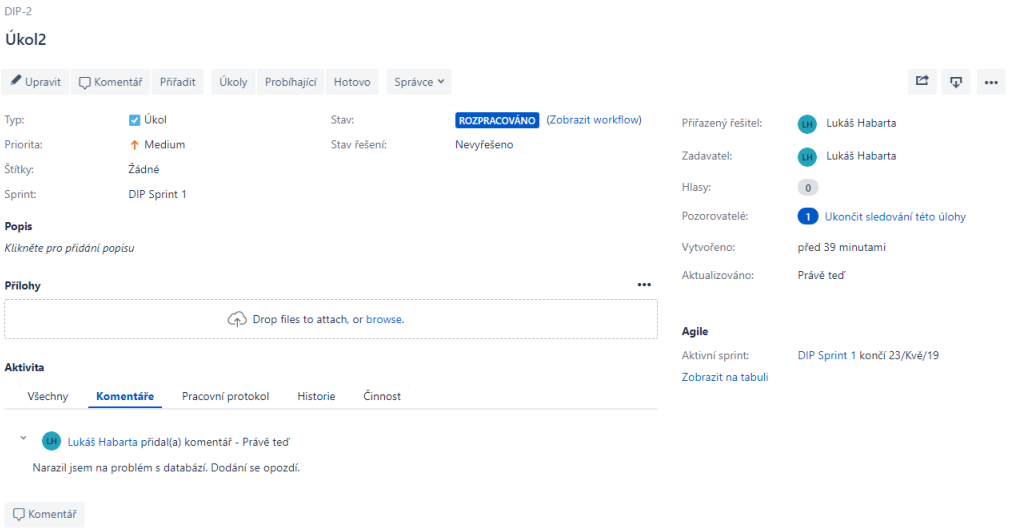
jektech je počet polí několikanásobný, mezi častá povinná pole patří vývojová verze softwaru nebo časový odhad úkolu.

Poslední zajímavou částí, kterou nalezneme jako uživatelský prvek jsou sestavy (3.6). Jedná se o podobné aplikace, jako jsou dostupné v dashboard. Jsou to tedy převážně grafy. Rozdíl je ten, že tyto grafy jsou vytvářeny v rámci konkrétního projektu a jsou vázané na jeho dashboard. Sestavy slouží uživatelům k rychlému náhledu na aktuální stav daného projektu.

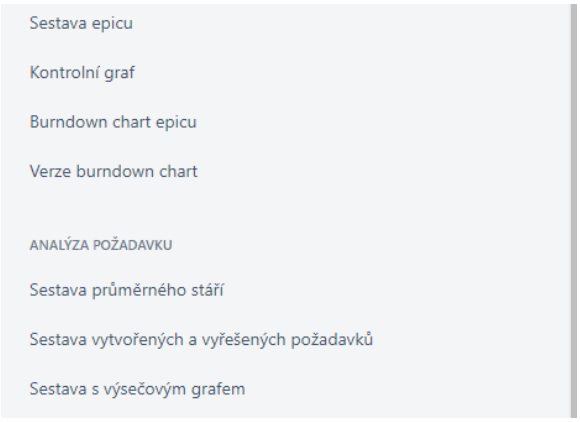
3.1.2 Nastavení

Všechny uvedené informace jsou prezentovány pro základní nastavení aplikace. JIRA ovšem pro každou z výše uvedených položek poskytuje nepřeberné množství individuálního nastavení. Je možné si nastavit téměř vše, na co si vzpomeneme. Ať už se jedná o požadované dodatečné informace na projektech či úkolech nebo o rozšíření typů úkolů. Díky nastavení aplikace je možné nastavit posílání notifikací emailem pro uživatele, kteří se účastní projektu nebo jsou nějak zainteresovaní do úkolů. Další obrovskou výhodou je poskytnutí propojení úkolů s verzovacím systémem.

JIRA poskytuje nástroje pro propojení s verzovacími systémy *git* a *SVN*. Pro použití s SVN stačí doinstalovat zásuvný modul a vytvořit spojení mezi SVN repozitářem a JIRA. Po vytvoření spojení je možné v SVN commit používat odkazy na JIRA úkoly. Díky tomu je pak u úkolu vidět nová záložka *Subversion*. Tato záložka obsahuje informace z commit zprávy a náhled na změněné soubory. Pokud JIRA propojíme s GitLab.com, tak nám GitLab umožní navázat konkrétní větev (anglicky branch) a merge request. Budeme tedy mít v JIRA odkaz vedoucí na merge request, který souvisí s daným úkolem.



Obrázek 3.5: Detail úkolu



Obrázek 3.6: Výřez sestav

3.1.3 Zhodnocení

JIRA je všestranný nástroj, který poskytuje prostředí pro řízení projektů. Poskytuje ale i dodatečné aplikace či zásuvné moduly, které umožňuji další rozšíření funkcionality. Z hlediska řízení projektu poskytuje vše, co pro správné řízení potřebujeme. JIRA se vyskytuje ve 2 verzích. Starší verze byla z uživatelského rozhraní méně přívětivá. Spousta nastavení byla skrytá, takže jejich aplikování bylo časově náročné. Navíc se zde vyskytovalo hodně nepříjemných chyb. Byl zde například problém s editací popisu úkolů. Pokud editace probíhala příliš dlouho, tak došlo k odpojení od serveru a text nebyl uložen. Mohla tedy být ztracena i několikahodinová práce. Protože nachystat úkol s dobrý popisem a akceptačními kritérii není otázkou 5 minut. V nové verzi, která je zde ukázána, se zdají tyto problémy již vyřešeny. Tato nová verze je již uživatelsky daleko přívětivější a přehlednější. JIRA na novou verzi nasadila kvalitní, ale zároveň jednoduchý design, který práci s ní hodně zpříjemňuje.

Pokud se někdo poohlíží po kvalitním softwaru pro řízení projektu, tak bych doporučil zvažovat JIRA jako jeden z prvních nástrojů. Má za sebou roky vývoje a nasazení v tisících podniků, spousta chyb je tedy již odhalena a opravena. JIRA navíc poskytuje přívětivé

ceny i pro malé a střední podniky. Do 10 uživatelů má paušální cenu $10. Pro středně velké týmy (do 100 členů) má cenu $7 za jednoho uživatele.

3.2 Easy project

Easy project jsem zvolil, protože jsme k němu měli přístup v rámci výuky a navíc se Easy project zaměřuje, na rozdíl od JIRA, více na stránku samotného řízení. Obsahuje propracované prostředí, které nám umožňuje se věnovat detailněji jiným částem řízení projektu, než je vytváření a správa úkolů. V rámci řízení projektů se specializuje více na analýzu rizik, nákladů a návaznost. Toto vyplývá z historie samotného podniku. Easy Project byl založen na otevřeném zásuvném modulu *Redmine* a jeho první funkcionalita byla zaměřena na řízení podnikových financí. Až s odstupem času (zhruba 4 roky) se sem dostaly funkcionality pro samotné řízení projektu.

Z mojí zkušenosti s tímto softwarem není vhodný pro použití v agilním prostředí. Více najde uplatnění ve stabilnějším prostředí, kde nedochází ke změně požadavků ze strany klienta příliš často. Případně v agilním prostředí pro vyšší management, který se zaměřuje na souběžné plánování chodu více projektů a je odstíněn od změn probíhajících na úrovni jednotlivých úkolů v projektu. Jedná se ale pouze o můj osobní pohled na tento software. Easy Project totiž poskytuje i sadu nastavení pro agilní projekty. Je zde například Scrum a Kanban.

Pro porovnání jsem stejně jako u JIRA zvolil dočasný účet a využil Easy Project, který běží v cloudu. Easy Project při prvním spuštění připravil přehledné demo se základními úkoly a jedním projektem. Jedná se o projekt, který má za úkol nás seznámit s funkcemi, které Easy Project poskytuje.

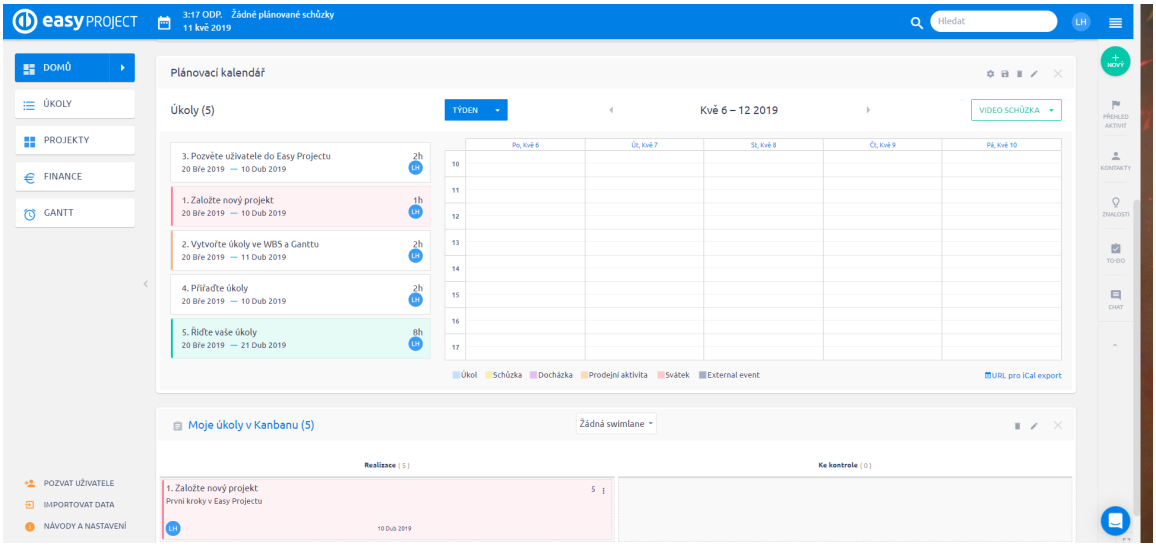
3.2.1 Funkcionalita

Po načtení aplikace se zobrazí úvodní stránka, která obsahuje plánovací kalendář s přehledem aktivních úkolů. Úkoly obsahují v přehledu pouze základní informace, je zde třeba informace o době trvání úkolu či odhadovaný čas na jeho dokončení. Na levé straně se nachází jednoduché menu s pěti záložkami.3.7 Nás bude zajímat pouze záložka projektů. V záložce financí je možné ale například nastavit informace o příjmech, ceně práce či dodatečných výdajích.

Položka *Projekty* zobrazuje pouze přehled všech našich projektů.3.8 Mnohem zajímavější je ale pohled na detail projektu, který má bohatou nabídku dalších záložek. Ze všech záložek se výběrem zaměříme na ty nejzajímavější:

- WBS
- Gantt
- Scrum
- Rozpočet

Záložka *WBS* nám umožňuje vytvářet hierarchickou dekompozici projektu, která je následně graficky zobrazená. Základní hierarchie je automaticky vygenerovaná ze zvoleného projektu a všech úkolů, které se na projektu nachází. Pro každý zobrazený element můžeme přidávat další a další elementy. Tyto elementy představují postupnou dekompozici elementu. Elementem v tomto případě je konkrétní úkol z projektu.



Obrázek 3.7: Úvodní stránka

Další záložkou je *Gantt*3.10. Tato záložka zobrazuje Ganttův diagram pro daný projekt. Každý úkol je možné tady přímo upravovat. Povoleno je upravovat trvání úkolu, úkol můžeme zkrátit, prodloužit nebo úplně posunout na časové ose. Další možností je zde vytvářet návaznost úkolu tím, že spojíme konec jednoho úkolu se začátkem dalšího úkolu. Jednotlivé úkoly v grafu zobrazují po najetí myší dodatečné informace. Bohužel tento graf postrádá informaci o odpracovaném čase. V celkovém popisu projektu je zde graficky oddělena část, která je již odpracovaná a která ještě zbývá (tmavě a světle modrá barva). Tato odpracovaná část je ale brána pouze z datumů, není zde nijak započítán časový odhad na projektu či jednotlivých úkolech. Pro generování Ganttova diagramu tu chybí možnost vytváření vazby přímo na jednotlivých úkolech a projektech. Pokud chceme přidat závislost úkolu na jiném úkolu, tak to lze až ve vygenerovaném diagramu.

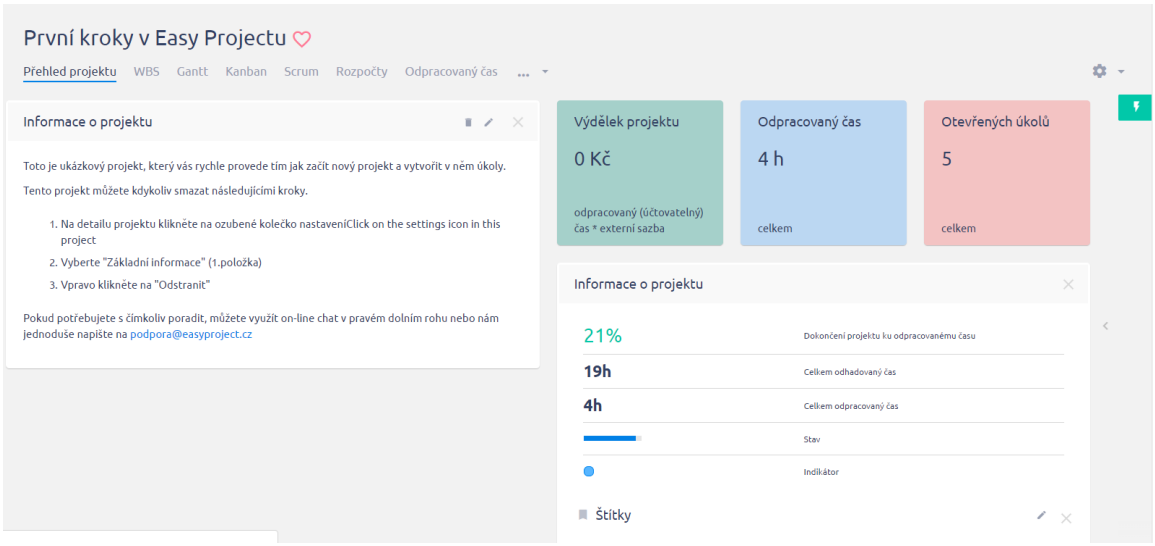
Scrum je pouze jednoduchá záložka, kde se definuje začátek a konec sprintu. Dále se zde nastaví jednotlivé úkoly pro daný sprint. Není to tedy rozsáhlé nastavení. To ale odpovídá primárnímu zaměření této aplikace.

Poslední z vybraných záložek je *rozpočet*3.11. Tady se přiznám, že postrádám význam takovéto volby na projektu. Jedná se o souhrnný přehled příjmů a nákladů. Absolutně nesouvisí s řízením samotného projektu. Tato informace by dle mého názoru měla být součástí nabídky *Finance* z menu. Navíc jsem zjistil, že výpočet na této záložce nepočítá správně.

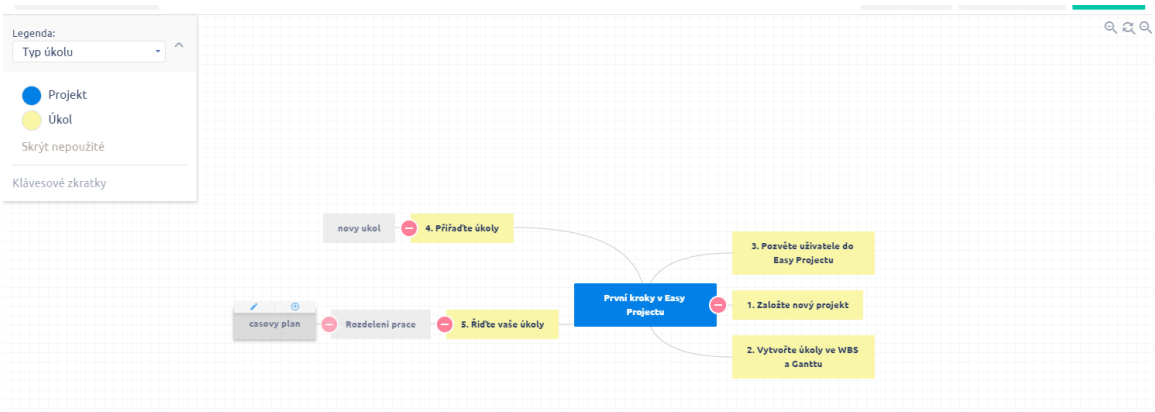
Pokud se na obrázku 3.11 podíváme do sloupce *skutečnost* a na řádek *Osobní náklady*, tak zde vidíme informaci o odpracovaném čase. Jsou zde uvedeny 4 hodiny práce. Při prozkoumání méně intuitivního nastavení zjistíme, že výchozí hodnota za hodinu práce je nastavena na 240 Kč. Hodnota práce by tedy měla být 960 Kč, v řádku je bohužel uvedeno 0 Kč. Neodpovídá to tedy skutečnosti.

3.2.2 Zhodnocení

Easy Project byl součástí výuky před 2 lety a musím uznat, že se aplikace hodně změnila. Mile mě překvapila dosažená změna za tento relativně krátký čas. Stal se mnohem více přehlednějším, než býval. Uživatelské rozhraní je nyní velmi přívětivé a moderní. V Easy

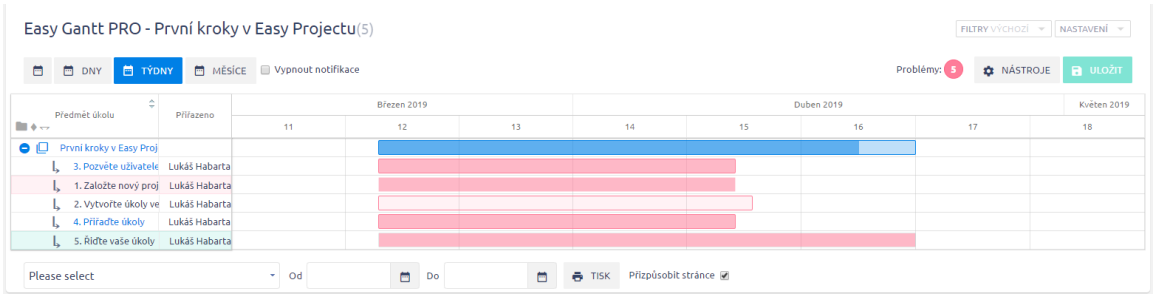


Obrázek 3.8: Detail projektu



Obrázek 3.9: Vytváření WBS

project je pořád cítit zaměření směrem k řízení financí a rizik, které se staly základem pro tento software. Na první pohled se může zdát nabídka strohá, jelikož součástí hlavního menu je pouze 5 položek. Spousta další funkcionality je totiž skryta uvnitř jednotlivých položek. Jedná se jistě o kvalitní nástroj pro řízení projektu, ale bohužel je zde spousta nedostatků, které kazí dojem. Ať už se jedná o malé nedostatky v podobě chybějícího nastavení pro Ganttův diagram či hrubé nedostatky, jako je špatně fungující *rozpočet*. Kladně hodnotím to, že Easy project navíc ještě poskytuje školení pro projektové manažery. Je tedy možné pracovat a vzdělávat se v rámci jedné platformy. Problémem ale je, že Easy project nastavil ceny vyšší než třeba JIRA, která je daleko vyspělejší. Cena za jednoho uživatele se zde pohybuje kolem 300 Kč za měsíc. Je to tedy zhruba dvojnásobek ceny, kterou si účtuje JIRA. Pro společnosti do 10 lidí je to patnáctinásobek ceny JIRA.



Obrázek 3.10: Ganttův diagram

Obrázek 3.11: Rozpočet

Kapitola 4

Návrh řešení

V této kapitole se zaměříme na analýzu a návrh konkrétního řešení. Dále se podíváme na výběr technologií zvolených pro implementaci a na odůvodnění tohoto výběru. V poslední části se podíváme na popis zvolené architektury.

Vytvoření použitelné a funkční aplikace pro řízení projektu není snadné. Jak jsme mohli vidět u dvou zvolených stávajících řešení, tak součástí daných programů je nepřeberné množství funkcionalit. Jednotlivé funkcionality jsou navíc provázány mezi sebou a vzájemně se prolínají. Pro vývoj takové aplikace je tedy nezbytné mít dobře zvládnutou analýzu konkrétního řešení. Vhodné je také udělat si průzkum dostupných technologií a zvolit technologie odpovídající náročnosti aplikace.

4.1 Analýza a návrh řešení

Jak jsme si již řekli, potřebujeme robustní systém, který bude schopen zvládnout velké množství vzájemně propojených funkcionalit. Jako vhodná varianta zde vychází webová aplikace. Jednou z výhod webových aplikací je to, že klient pouze zobrazuje výsledky. Většina zátěže je tedy na serveru. V dnešní době jsou navíc webové aplikace populární, jelikož je možné na ně přistupovat i z mobilních telefonů.

Při samotném návrhu řešení jsem operoval s rozdělením na základní a pomocné entity. Základní entita je něco, co představuje důležitou funkcionalitu. Pomocná entita pouze upřesňuje/rozšiřuje základní entitu. Pokud se podíváme na požadavky z funkcionality, tak základní součástí všech systémů je uživatel, projekt a úkoly. Budeme tedy potřebovat mít možnost přidávat a editovat uživatele. Dále vytvářet a editovat projekty a úkoly. Všechny funkcionality si můžeme označit jako základní entity. Úkol sice rozšiřuje projekt, ale je tak významný, že bude základní entitou.

Výběr implementovaných funkcionalit byl problematický. Za relativně krátký časový úsek nebylo možné, abych sám implementoval aplikaci na úrovni JIRA nebo Easy project. Bylo tedy potřeba nějak zúžit výběr. Pro tento náročný úkol jsem využil znalostí svých kolegů z praxe. Dotázal jsem se tří kolegů, kteří dennodenně pracují s JIRA nebo Microsoft Team Foundation Server (TFS). Jedná se o 2 projektové manažery a jednoho byznys analytika. Poskytli mi seznam věcí, které považují za základ systému tohoto typu, a také seznam toho, co jim na JIRA nevyhovuje nebo chybí.

Díky této zpětné vazbě bylo jasné, že se bude jednat o aplikaci ve stylu *proof-of-concept (PoC)*. PoC je styl psaní aplikací, kdy se zaměříme na možnost implementace požadovaných funkcionalit na úkor chybovosti a případně i uživatelského rozhraní. Je tedy tolerováno, po-

kud aplikace obsahuje větší množství chyb nebo není příliš uživatelsky přívětivá. Průnikem požadavků od kolegů byl tento seznam:

- rozdělené uživatelské role, ne jen pravomoci uživatelů
- sdružování uživatelů v týmu
- projekt je brán jen jako schránka pro jednotlivé úkoly
- úkoly jsou to nejdůležitější v systému - celá aplikace pracuje jen s úkoly
- žádaný byl real-time chat, JIRA i TFS poskytuje pouze možnost komentářů k úkolu
- grafický přehled úkolů na samostatné stránce

Pokud se tedy podíváme na seznam požadavků, tak obsahem aplikace bude projekt složený z úkolů. Pro každý úkol nebo projekt bude možnost komunikovat pomocí real-time chatu se všemi zúčastněnými uživateli. Připravená bude stránka s grafickým přehledem progresu jednotlivých projektů. Uživatelé zde budou vystupovat v různých rolích, kde každá role bude mít vlastní pravomoce. Bude možné vytvářet týmy a přiřazovat jednotlivé uživatele do těchto týmů. Projekt tedy bude jen název zastřešující seznam úkolů.

Pro vytváření úkolů jsem se rozhodl operovat se třemi typy úkolů

- Bug - chyba
- Task - běžný úkol, je možné pro něj vybrat některý z předdefinovaných rozšiřujících typů:
  - Project management - označuje úkol související se samotným řízením projektu
  - Business analysis - označení pro analýzu z byznys pohledu
  - Technical analysis - technická analýza řešení
  - Development - samotný vývoj funkcionality
  - Testing - testování dodané funkcionality
- Story - jedná se o spojení více úkolů do jednoho celku. Může se jednat třeba o podobné úkoly nebo o složitější funkcionalitu, která je kvůli Scrum přístupu rozdělena na menší části

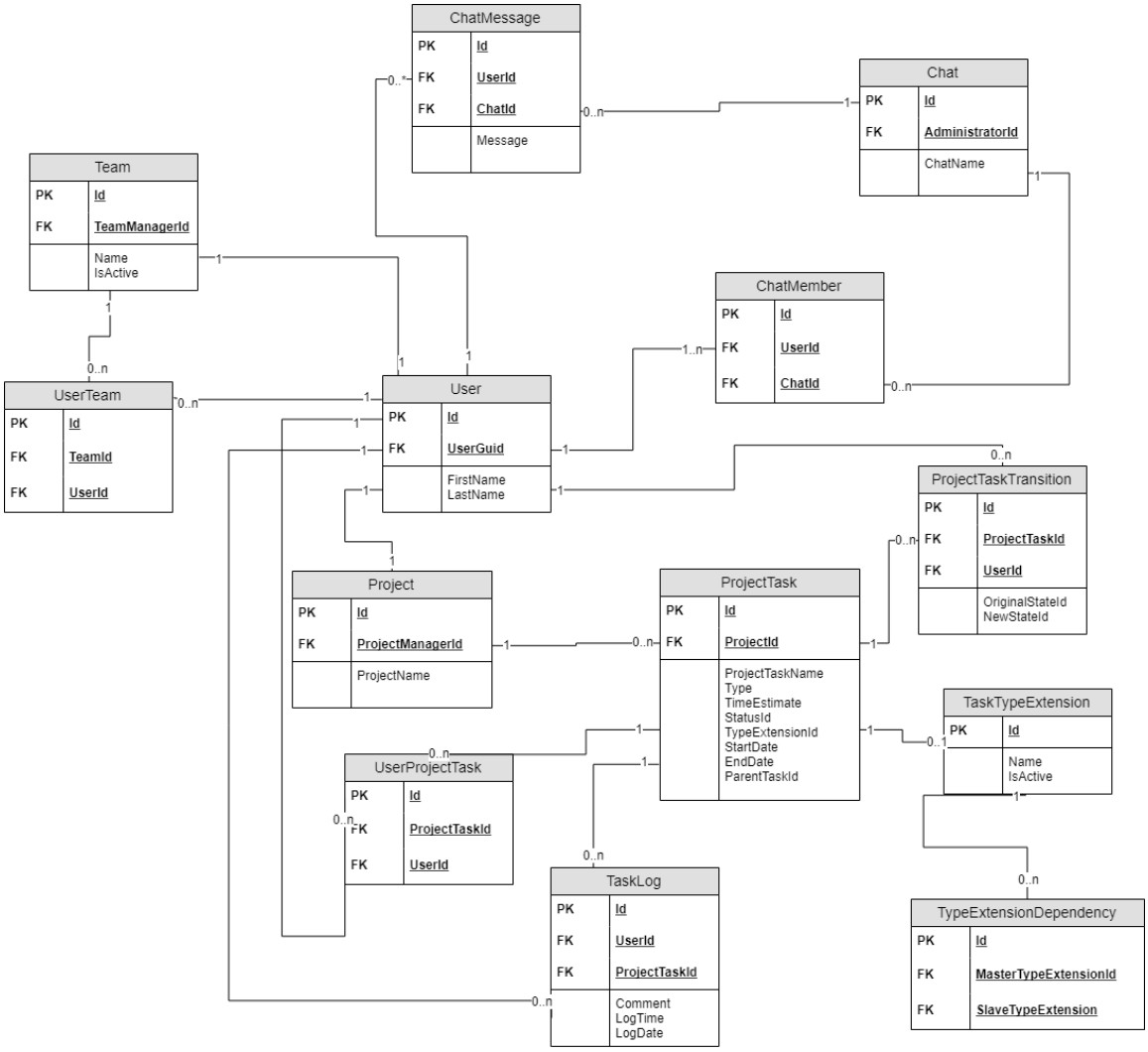
Rozšiřující typy pro úkoly bude možné upravovat přímo v aplikaci. Jelikož se jedná o rozšíření úkolu, bude to tedy pomocná entita.

Chat bude existovat pro každý nově vytvořený projekt. Každý z uživatelů, který bude přiřazen na některém z úkolů v rámci toho projektu, bude zařazen do skupiny, která se účastní chatu. Rozhodl jsem se ještě vytvořit jeden globální chat, jehož součástí budou všichni uživatelé v systému. Tento globální chat má využití v případě, že chceme něco sdělit všem lidem v systému. Nemusíme posílat email a můžeme o dané věci zrovna diskutovat. Je ale otázkou, jak by takový chat byl přehledný v případě, kdy v systému jsou stovky uživatelů.

Z vlastní iniciativy jsem se rozhodl ještě přidat Ganttův diagram. Tento diagram je generován na základě projektu. Operuje s úkoly na projektu a jejich rozšiřujícími typy. Je zde tedy vytvořena hierarchie posloupnosti přímo v grafu. Proto není nutné jako v Easy project vytvářet ji ručně. Pro tento účel je potřeba vytvořit seznam závislostí jednotlivých rozšíření úkolů. Ganttův diagram jsem zvolil, protože se podle mě jedná o pěkné grafické znázornění průběhu projektu.

4.1.1 Návrh databáze

Pro vyobrazení návrhu databáze přikládám ER diagram4.1. Tento diagram zobrazuje všechny tabulky, které se v databázi nacházejí. Tabulky obsahují informace o primárních i cizích klíčích a také dodatečné sloupce s hodnotami. ER diagram obsahuje 13 tabulek. V databázi se ale nachází 20 tabulek. 7 tabulek, které chybí v ER diagramu, jsou automaticky generované tabulky z *ASP.NET Identity*. Vazba mezi těmito automaticky generovanými tabulkami a mými je v entitě *User*. Konkrétně se jedná o sloupec UserGuid. UserGuid je cizí klíč na GUID v AspNetUsers tabulce. GUID je označení pro globálně unikátní identifikátor, jedná se tedy o primární klíč, jehož hodnota by měla být unikátní i napříč všemi systémy. Z důvodů, které následně uvedu níže, není pro moji aplikaci vhodné GUID využívat, proto je zde mapovací tabulka *User*.



Obrázek 4.1: ER diagram

4.2 Výběr technologií

Jelikož se jedná o diplomovou práci, která spadá pod akademické práce, rozhodl jsem se využít experimentálních technologií. Pro tento typ technologií jsem se rozhodl i z důvodu, že se jedná o PoC. Experimentální technologie jsou technologie, které jsou teprve v raném stádiu vývoje a nejsou vhodné pro zavedení do podnikových systémů. Tento typ technologií může obsahovat ještě velké množství nevyřešených chyb. Dalším nedostatkem je, že ještě nemusí být podporovány všechny potřebné funkce.

Při výběru jsem se zaměřil na technologie, které mají společný základ s tím, co už znám. Dalším kritériem bylo, aby dané technologie umožňovaly rychlý vývoj a bylo možné v nich implementovat větší množství funkcí, aniž by nedošlo k vytvoření chaotického kódu. Chtěl jsem také, aby technologie byly ideálně multiplatformní. To se mi z větší části podařilo. Rozhodně jsem se chtěl vyhnout jazyku Javascript. Tento jazyk sice umožňuje extrémně rychlý vývoj, ale kvůli absenci typové kontroly je lehké v něm vytvářet nežádoucí chyby. Navíc typová kontrola je jedna z věcí, kterou preferuji pro bezpečný systém.

4.2.1 Back-end

Jednou ze základních povinností pro každou rozsáhlejší aplikaci je databázový systém, který slouží jako úložiště perzistentních dat. Osobní zkušenosti mám s databázovým systémem od společnosti Microsoft a Oracle. Pro multiplatformní řešení by bylo vhodnější použít Oracle. Bohužel s ním nemám dobré zkušenosti a ve všech disciplínách nad ním vítězí MSSQL od Microsoftu. V případě použití na jiném systému než Windows je možné databázi registrovat do Docker, který již multiplatformní je.

Pokud šlo o výběr programovacího jazyku pro back-end, tak volba byla z mé strany jediná možná, a to C#. Tento jazyk je mi vlastní a množství funkcí, které podporuje, je větší než u JAVA. Navíc JAVA se mi jeví v některých věcech těžkopádná a pomalá, ať už se jedná o dodávání nových funkcí nebo samotnou funkcionalitu. Na vině je dost možná to, že JAVA běží v Java Virtual Machine a JVT je samo o sobě multiplatformní, takže je složitější jej udržovat aktuální a funkční na všech platformách. Jako framework k C# jsem zvolil .NET. Nejedná se ale o tradiční .NET, nýbrž o .NETCore. Framework .NETCore je nová technologie od Microsoftu, která umožňuje běh aplikací napsaných nad .NET framework i na Unixových systémech. Takže je možné využívat všechny dostupné funkce z tohoto rozsáhlého framework.

Pro přístup k databázi bylo možné využít Entity Framework (EF), který je dodáván spolu s .NETCore framework. EF patří do skupiny ORM framework. ORM je objektově relační mapování, které poskytuje sadu nástrojů pro mapování relačních dat z databáze do objektů v objektově orientovaném programování. EF je robustní nástroj pro práci s databází, který kromě základních funkcí podporuje i pokročilé funkce pro práci s databází. Umožňuje například i kompletní mapování strukturovaných objektů do databáze. Bohužel EF je díky rozsáhlé funkcionalitě relativně pomalý. Z tohoto důvodu jsem raději sáhl po framework ve stylu microORM. MicroORM poskytuje pouze základní sadu nástrojů, ostatní pokročilejší nástroje je potřeba si sám napsat. MicroORM framework existuje pro .NET nespočet. Vybral jsem základní Dapper framework od komunity vývojářů ze StackOverflow (největší internetové fórum pro získávání rad ohledně programování a řešení problémů). Tento framework je díky zkušené komunitě velmi rychlý a kvalitní. Základní verze Dapper je dostupná jako otevřený software. Poskytuje všechny základní funkce, ale s použitím rozšíření je možné využívat i LINQToSQL, který značně zrychluje vývoj.

Datový typ v C#	Datový typ v MSSQL
int	INT
long	BIGINT
decimal	DECIMAL(M,N)
DateTime	DATETIME
bool	BIT
Guid	UNIQUEIDENTIFIER
string	NVARCHAR(X)

Tabulka 4.1: Konverze datových typů

Při samotném vývoji se očekává neustálá změna databázového modelu. Je totiž běžné, že postupně přidáváme další a další sloupce do databáze. Můžeme požadovat změnu datového typu nebo můžeme chtít označit sloupec jako povinný. Všechny tyto operace požadují napsání SQL skriptu, který je potřeba následně na databázi spustit. Pokud se podíváme na skript, který vytvoří základní strukturu tabulek podle ER diagramu, tak zde se jedná o stovky řádků kódu v SQL. Pro psaní skriptů existují asi jen dvě možnosti.

- **jeden skript** - po celou dobu máme k dispozici pouze jeden skript, který podle potřeby upravujeme. Nevýhodou tohoto skriptu je, že pokud není součástí nějakého verzovacího nástroje (například git), tak nemáme přístup k jeho předchozí verzi. Další nevýhodou je, že není možné upravit stávající databázi, ale musí se vždy zahodit aktuální databáze a vytvořit ji znovu.
- **inkrementální spouštění skriptů** - původní skripty se nemění. Pro každou změnu se vytváří nový skript, který je poté spuštěn. Tento přístup nám umožňuje aplikovat skripty na stávající databázi, ale je problematický na údržbu. V ideálním případě je potřeba mít nástroj, který skripty spouští v požadovaném pořadí automaticky. Pokud totiž potřebujeme databázi migrovat na jiné zařízení nebo ji prostě z nějakého důvodu vytvořit znova, tak spouštění skriptů může být časově náročné a může se na nějaký skript zapomenout.

4.2.2 Vlastní rozšíření pro Dapper

Jelikož oba zmíněné přístupy nejsou pro můj styl psaní aplikací vhodné, rozhodl jsem se napsat si na začátku vlastní framework, který rozšiřuje Dapper. Tento framework je použitelný pouze s Dapper, ale není omezen na .NETCore. Je možné jej tedy použít i s klasickým .NET framework. Tento framework využívá reflexi v C# k prohledání konkrétního projektu. V rámci tohoto projektu hledá všechny třídy označené pomocí speciálního značkovacího rozhraní. Pokud třída obsahuje toto značkovací rozhraní, pak je považována za model, který je potřeba namapovat na databázi. Tento framework zároveň převádí datové typy ze C# na datové typy, které se vyskytují v MSSQL. Aktuálně je hotová konverze pro datové typy obsažené v tabulce 4.1.

U *DECIMAL* reprezentuje M počet čísel a N počet desetinných míst. Pokud není specifikováno jinak, tak $M = 20$ a $N = 10$. Při *NVARCHAR* je výchozí nastavení pro $X = MAX$. Tedy maximální velikost *NVARCHAR*. Hodnoty pro *DECIMAL* a *NVARCHAR* je možné definovat pomocí speciálních atributů k tomu určených. Součástí jsou ještě další atributy:

- **MapperIgnore** - umožňuje vynechat takto označený atribut třídy

- **NotPk** - tímto atributem označuje atribut třídy, pokud je to kandidát na primární klíč v databázi, ale nechceme jej
- **IsNullable** - v C# existují datové typy, které nemohou být Nullable. Tyto datové typy ale na straně MSSQL mohou být označeny jako volitelné, mohou tedy obsahovat nedefinovanou hodnotu NULL. Tento atribut nám tedy umožní tyto typy nechat volitelné
- **Fk** - atribut, který očekává dva parametry. První parametr specifikuje název tabulky a druhý název sloupce. Pomocí této kombinace se vytvoří cizí klíč

Součástí framework je i vytváření primárních a cizích klíčů. Pro primární klíč, bez použití atributu, jsou předdefinované dvě varianty. Pokud atribut třídy má jméno *Id* nebo *NázevTřídy + Id*, tak je označen jako primární klíč dané třídy. Cizí klíče se taky hledají podle shody na jméno třídy. Například máme třídy *Uživatel* a *Zaměstnanec*. Pokud ve třídě *Uživatel* je atribut s názvem *ZaměstnanecId*, pak je tento atribut namapován do databáze jako cizí klíč na tabulku *Zaměstnanec* a sloupec *Id*. Pro jinou variantu cizího klíče je potřeba použít již zmíněný atribut.

Framework při každém spuštění aplikace projde označený projekt a porovnává modely v kódu s existujícím databázovým modelem. Pokud nalezne rozdíl v těchto dvou modelech, tak spustí na databázi skript, který přizpůsobí databázi aktuálnímu modelu v kódu. Přidáme-li novou třídu, která dědí ze značkovacího rozhraní, tak je automaticky vytvořena v databázi se všemi sloupci. Pokud do existující namapované třídy přidáme nový atribut, do tabulky se přidá nový sloupec. Pokud atribut odebereme, odstraní se i sloupec.

Poslední částí framework je zabalení Dapper metod. Metody jsou zabaleny tak, aby bylo možné pomocí jednoduchých dotazů vkládat, aktualizovat či mazat data. Základní dotazy jsou tedy automatizované. Pokud ale požadujeme pokročilejší dotaz se spojením několika tabulek, nebo pouze s výběrem některých sloupců, tak musíme použít vlastní dotaz napsaný v SQL a tento dotaz pomocí Dapper spustit.

Z výše uvedeného popisu pro framework je zřejmé, že tento framework umožňuje extrémně rychlý vývoj. Díky automatické úpravě databázového schématu se totiž nemusíme vůbec starat o databázi a vytváření skriptů. Samozřejmě že toto prohledávání projektu pomocí reflexe je časově náročné, ale nikoliv tak jako použití EF. EF navíc požaduje, abychom udržovali jím vygenerované modely na straně kódu aktuální. EF je pomalý neustále, tento framework má časově náročnou pouze inicializační část. Pokud požadujeme nasazení aplikace do produkce, kde je taková úprava databáze nežádoucí, můžeme mapování zakázat jediným řádkem kódu.

Poslední technologií použitou na back-end aplikace je SignalR. Nejedná se čistě o použití na back-end. Již z definice této technologie je znát provázání s klientem. SignalR nám umožňuje propojit klienta se serverovou částí aplikace. Vytváří jakýsi pomyslný tunel mezi klientem a serverem, přes který je možné komunikovat v reálném čase. Důležité je, že tato komunikace je obousměrná. V případě webových aplikací komunikuje klient se serverem a server pouze klientovi odpovídá a posílá požadované výsledky. V případě SignalR je možné ze serveru komunikovat s klientem. Na jeden tunel může být připojeno i více klientů současně. Nejčastějším použitím SignalR je posílání notifikací a informací více klientům současně, případně mezi dvěma klienty. Typickou aplikací SignalR je chat. Bez použití SignalR se po odeslání zprávy prvním uživatelem musí čekat, až se druhý uživatel dotáže serveru na zprávy, teprve v tento moment vidí novou zprávu. SignalR nám umožní předat

zprávu druhému uživateli, aniž by posílal jakýkoliv dotaz na server. SignalR ve zkratce posílá notifikace přes oboustranný WebSocket definovaný v HTML5.

4.2.3 Front-end

Jak jsem již zmiňoval, chci se za každou cenu vyhnout jazyku Javascript. Z tohoto důvodu jsem využil experimentálního framework Blazor. Blazor je nová generace technologií pro psaní webových aplikací. Za framework Blazor stojí opět společnost Microsoft, která se v posledních letech začala orientovat i na otevřený software, který dlouhá léta zavrhovala. Díky tomu vznikl unikátní projekt ve spolupráci Microsoftu a komunity kolem otevřeného softwaru.

Blazor v době začátku psaní aplikace byl ještě označen jako experimentální technologie. Nebyl tedy určen pro použití v komerčním prostředí nebo v aplikacích potřebující stabilní prostředí. Vývoj totiž probíhal rychle a téměř každý měsíc dostával Blazor novou sadu funkcí. Tradiční webové aplikace staví na kombinaci HTML, CSS a Javascript. Blazor se pokouší využít technologii WebAssembly (Wasm). Jedná se o první takhle rozsáhlý pokus o využití Wasm. Díky Blazor je možné psát webové aplikace v jazyce C#, proto máme možnost využívat všech výhod a vlastností tohoto jazyku. Blazor se využívá v kombinaci se značkovacím jazykem Razor, který je založen na HTML. Pro stylování komponent je pořád nutné využívat CSS. Pro zajištění bezpečnosti, autorizace a autentizace je využit JSON Web Token (JWT).

WebAssembly

Technologie WebAssembly byla vydána v roce 2017 jako standard konsorciem World Wide Web, známějším pod zkratkou W3C. Pod záštitou W3C se na vývoji Wasm podílely společnosti jako Microsoft, Google či Apple.

WebAssembly definuje vlastní zásobníkový virtuální stroj, který zpracovává binární instrukce. Jelikož se jedná o technologii pro webové prohlížeče, tak je přenositelný. Je možné jej tedy využít jak na počítačích, tak na mobilních telefonech v prohlížečích, jež tuto technologii podporují. Hlavním cílem WebAssembly je umožnit vývoj aplikací za pomoci pouze jednoho jazyku. Umožňuje totiž běh aplikací napsaných například v C nebo C++ přímo v prohlížeči. Běh aplikací převedených do WebAssembly by neměl trpět penalizací za běh ve virtuálním prostředí. Rychlost běhu tedy odpovídá rychlosti běhu nativního kódu.

Aplikace spouštěné jako WebAssembly by měly být bezpečnější než aktuální aplikace běžící na Javascript. WebAssembly totiž využívá odděleného sandboxu, který zabraňuje spouštění cizího kódu v prohlížeči. Nemělo by tedy být možné napadnout cizí počítače pomocí webových stránek tím, že se na pozadí načte aplikace v Javascript, která se uloží na pevný disk a spustí přímo v počítači uživatele.

Blazor a WebAssembly

Jak bylo již naznačeno, tak Blazor využívá kompilaci do WebAssembly a následné spouštění kódu napsaného v C# a .NET v prohlížeči. Pro převod do WebAssembly je nutné mít kompilátor, který převede nativní kód do spustitelné verze v prohlížeči. Microsoft zde mohl využít své dceřinné společnosti Xamarin, která stojí za vývojem otevřeného kompilátoru Mono. Mono je standardizováno nezávislou společností ECMA jako prostředí, které umožňuje běh aplikací napsaných ve framework .NET. Mono implementuje .NET framework podle ECMA standardu pro jazyk C# o Common Language Runtime (CLR - prostředí

pro běh aplikaci napsaných v .NET). Díky tomu je možné spouštět tyto aplikace na libovolné platformě. CLR bylo dlouhou dobu omezeno pouze na systémy Windows, Mono jej ale implementovalo jako ECMA Common Language Infrastructure (CLI) a otevřelo dveře framework .NET na Unix systémy. Kód napsaný v Blazor je tedy pomocí Mono převeden do multiplatformního spustitelného kódu, který následně běží pomocí WebAssembly.

Jelikož je Blazor ještě nová technologie, kterou čekají dlouhé roky vývoje, než se dostane na úroveň kombinace HTML + Javascript, tak je potřeba zachovat možnost používat i kód napsaný v Javascript. K tomuto účelu obsahuje Blazor Javascript Interop službu. Tato služba umožňuje z .NET metody volat Javascript funkce, případně i z Javascript volat zpět statické .NET metody. Díky této možnosti volání máme k dispozici všechny funkce dostupné v Javascript. Další výhodou je možnost převedení aplikace ve WebAssembly do Javascript. Toto je využito v případě, že aplikace v Blazor detekuje zastaralý prohlížeč, který nepodporuje spouštění WebAssembly aplikací. V tento moment celá WebAssembly aplikace bude běžet pomocí speciální Javascript knihovně. Pro tento účel je tedy potřeba mít nachystané prostředí pro volání Javascript funkcí.

4.3 Architektura aplikace

Jelikož jsme si již udělali základní přehled technologií, jež budou použity, tak se můžeme zaměřit na samotnou architekturu aplikace. Architektura aplikace tvoří jeden ze základních kamenů úspěšného vývoje. Pokud nepoužijeme žádnou architekturu nebo zvolíme špatnou architekturu, tak se vývoj aplikace v určitý moment může stát velmi náročným. Výběr a návrh architektury tedy nesmí být podceňován.

Pojem architektura softwaru se dá definovat jako strukturování aplikace do samostatných částí. Každá z částí má svou vlastní zodpovědnost za určitou funkcionalitu. Hranice mezi jednotlivými částmi musí být dobře definována a zároveň jasně oddělena.

Existuje mnoho používaných architektonických vzorů pro vývoj aplikace. Mezi obecné vzory, které se asi používají nejčastěji, můžeme zařadit

- **vrstvená architektura** - aplikace je rozdělena do samostatných vrstev, kde každá vrstva se stará o jinou funkcionalitu. Nejčastějším řešením je třívrstvá architektura
- Klient-server - aplikace je rozdělena na 2 samostatné aplikace. První aplikací je klient, který se stará pouze o zobrazení výsledků. Druhou aplikací je server, který zpracovává požadavky od klienta a vrací požadovaný výsledek
- Model-view-controller - rozděluje aplikaci do tří logických celků:
  - Model - obsahuje byznys logiku aplikace a operace nad databází. Model nijak nekomunikuje s klientem, pouze přijme dotaz a vrátí požadovaná data. Model je součástí back-end aplikace
  - View - stará se o samotné zobrazení dat. Uživatel pracuje s jednotlivým view, které posílá dotazy konkrétnímu modelu. View se řadí mezi front-end aplikace
  - Controller - zajišťuje komunikaci mezi modelem a view. Klient pomocí view komunikuje s kontrolérem pomocí HTTP dotazů. Kontrolér předává tyto dotazy na model a následně vrací výsledek na view

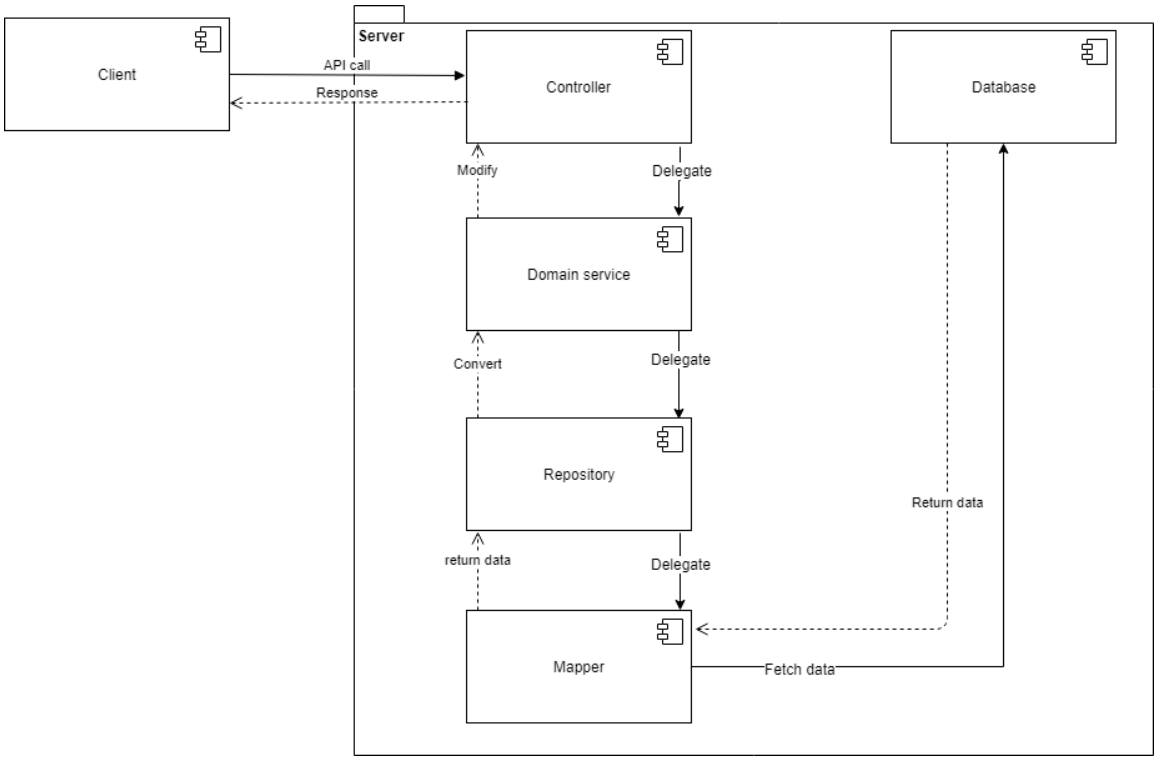
Všechny tři popsané přístupy se mohou na první pohled zdát podobné, ale je mezi nimi velký rozdíl v použití. Klient-server se dá použít hlavně pro menší aplikace. Klient

nemusí být nutně jen webová aplikace, může se jednat o počítačovou aplikaci. Rozdíl mezi vícevrstvou architekturou a MVC je hlavně v osobních preferencích. MVC poskytuje pouze 3 oddělené vrstvy, kde každá má předem daný obsah z hlediska funkcionality. U vícevrstvé architektury je rozdělení funkcionality mezi vrstvy naší volbou. MVC je tedy třívrstvá architektura s jasně definovaným rozdělením.

4.3.1 Hybridní vícevrstvá architektura

Z popisu uvedeného výše by se tato aplikace řadila někde mezi klient-server a vícevrstvou architekturu. Nejedná se totiž o úplně jednoduchou aplikaci, která by mohla fungovat čistě v klient-server režimu. Serverová část aplikace je zde díky použitým technologiím a přístupu náročnější a je potřeba ji rozdělit do více celků.

Zvolil jsem tedy kombinaci klient-server s upravenou třívrstvou architekturou. Na první pohled se jedná o klienta (webová stránka) a server. Server zde komunikuje s klientem přes API, jedná se konkrétně o RESTApi. Klient zde představuje prezentační vrstvu. Samotný server jsem se rozhodl rozdělit na 2 základní vrstvy - datová vrstva a aplikační vrstva. Datová vrstva je dále rozdělena na dva samostatné celky. Aplikační vrstva je rozdělena na samostatnou byznys logiku a kontrolér, který obsluhuje požadavky klienta. Nákres architektury a zpracovávání volání na obrázku 4.2.



Obrázek 4.2: Nákres architektury a zpracování volání

4.3.2 Datová vrstva

Datová vrstva je nejdůležitější vrstva z pohledu fungování serveru. Musí být dostatečně rychlá s operacemi nad databází i s výslednými daty. Pokud se stane, že datová vrstva je

pomalá a operace trvají příliš dlouho, tak se to projeví i na klientovi. Klient v tento moment dostává odpověď se zpožděním a vypadá, že došlo ke ztrátě spojení a neodpovídá.

Jak již bylo zmíněno, datová vrstva je rozdělena na 2 menší vrstvy. První částí je mapovací vrstva (mapper). Tato vrstva je nejblíže databázi a přímo s ní komunikuje pomocí předdefinovaných dotazů nebo uživatelských SQL skriptů. Mapovací vrstva se stará o volání operací nad databází. Výsledky z těchto operací předává nadřazené vrstvě. Tato vrstva se nazývá repozitář (repository). Na úrovni repozitářů je povoleno volat operace z libovolné mapovací vrstvy. Jelikož má každý model vlastní tabulku v databázi, tak se místo složitých dotazů, které obsahují spojení několika tabulek, používají dotazy přímo na jednu z tabulek. Dotazy se posílají na mapovací vrstvu a na úrovni repozitářů se tyto dotazy kombinují do složitějších objektů.

Tento přístup je zde potřeba, jelikož aplikace je postavená na microORM Dapper. Pokud se používá microORM framework pro práci s databází, doporučuje se využívat většího množství malých dotazů místo jednoho velkého. MicroORM je totiž extrémně rychlé, pokud výsledek databáze mapuje pouze na jednoduchý objekt. Pro usnadnění práce vývojářům umožňují některé microORM framework mapovat výsledky i na objekty složené z několika podobjektů. Umožňuje to například framework NPoco. Toto mapování je zde ale za cenu náročnější režie a pomalejšího přístupu k datům.

4.3.3 Aplikační vrstva

Datová vrstva by sama o sobě postrádala smysl, jelikož nepředstavuje žádnou byznys logiku. Tato logika se nachází v aplikační vrstvě. Aplikační vrstvu jsem se rozhodl taky rozdělit na dva samostatné celky. Konkrétně na kontrolér (controller) a doménovou službu (domain service). K tomuto rozdělení mě vedly dva hlavní důvody:

- **Bezpečnost** - v aplikaci využívám pro autorizaci a autentizaci JWT. Každý zabezpečený kontrolér je přes API tedy dostupný pouze přihlášenému uživateli a pouze s požadovaným oprávněním. Spousta operací na databázi poté vyžaduje Id aktuálního přihlášeného uživatele. Kontroléry tedy obsahují informace o jednotlivých přihlášených uživatelích. Není tedy nejlepší volbou mít tyto informace přímo přístupné v byznys logice. Může se díky chybě stát, že uživatel dostane přístup k informacím, na které nemá oprávnění. Díky rozdělení je šance na takovou chybu menší. Navíc není takto byznys logika přímo vystavená na API. Je skrytá za další vrstvou.
- **Rozdělení zodpovědnosti** - byznys logika může být hodně robustní, pokud by se aplikace dostala do pokročilejšího stádia. Je tedy zbytečné míchat tuto logiku s kontrolérem, který má primárně jen autorizovat, autentizovat uživatele a předávat požadavky dále.

Kontrolér se tedy stará o autorizaci a autentizaci uživatelů. Po jejich ověření deleguje požadovanou operaci na doménovou službu podle toho, které API klient zavolal. Výjimku zde tvoří pouze přihlašování uživatelů. Pro autorizaci a autentizaci uživatelů se v aplikaci využívá ASP.NET Identity. ASP.NET Identity validuje uživatele vůči databázi (zmíněných 7 tabulek, které nejsou v ER diagramu). Pro tento přístup k databázi využívá Entity Framework, v této aplikaci konkrétně Entity framework core. Jedná se o nedostatek, který se mi bohužel nepodařilo odstranit. V mojí architektuře je totiž přímé volání databáze z kontroléru nepřípustné.

Doménová služka se pak stará o samotnou byznys logiku. Z doménové služby je povoleno volat libovolný počet repozitářů. Repozitáře poskládají data získaná z mapovací vrstvy do

složitějších objektů, doménová služba poté získá tyto složitější objekty a provádí s nimi byznys operace. Pro byznys operace totiž můžeme potřebovat data z rozdílných modelů. Určit doménovou službu, která bude za danou operaci zodpovědná, může být někdy hodně náročné. Ne vždy je totiž hranice jasně daná. Doménová služba provádí například výpočty založené na určitých vstupních podmínkách. Tyto výpočty by bylo možné třeba provádět již na repozitáři, ale repozitář je součástí datové vrstvy, a datová vrstva má být kompletně odstíněna od problematiky byznysu.

4.3.4 Prezentační vrstva

Prezentační vrstva je jednolitá a stará se pouze o zobrazování výsledků uživateli. Samotná prezentační vrstva je psaná pomocí již zmíněného framework Blazor a WebAssembly. Na prezentační vrstvě jsou dostupné všechny formuláře, které následně pomocí odesílání dat na server dotváří samotnou aplikaci. Prezentační vrstva by neměla obsahovat žádnou byznys logiku. Obsahuje tedy pouze nezbytnou logiku a funkce. Jedná se o funkce, které vytvářejí například zabezpečení na úrovni uživatelského rozhraní. Není například možné odeslat formulář bez vyplněných povinných údajů. Toto ověření by bylo možné mít i na straně serveru. Bohužel se ale nejedná o čistou byznys logiku. Z hlediska byznysu můžeme požadovat například některá pole povinná, ale validace, že je dané pole vyplněné, už není starost byznysu.

Prezentační vrstva může v této aplikaci komunikovat se serverem dvěma způsoby. První způsob je provolávání API rozhraní na straně kontroléru. Toto provolávání se používá pro následné získávání dat ze serveru nebo pro modifikaci již existujících dat. A to jak z databáze, tak ze statického kontextu serveru. API rozhraní je ze strany klienta voláno pomocí HTTP dotazů. Druhý způsob využívá SignalR pro komunikaci se SignalR hub. Hub je vstupní či výstupní bod spojení více uživatelů mezi sebou a mezi serverem. Hub se nachází na straně serveru a jednotliví uživatelé jsou na straně klienta. Komunikace tedy taky prochází přes HTTP dotazy. O vytváření konkrétních dotazů se nestaráme my, ale přímo knihovna.

Kapitola 5

Implementace a testování

V této kapitole bude popsáno technické řešení implementace aplikace pro řízení projektů a následné testování aplikace. Nejprve se zaměříme na popis implementace serverové části, následovat pak bude popis implementace klientské části a uživatelského rozhraní. Kapitolu uzavřeme detaily z testování. Implementace serverové části nemá žádná specifika, jedná se jen o čtení dat, manipulaci s daty a jejich ukládání do databáze. Při popisu serverové části se tedy zaměříme na jednotlivé vrstvy a jejich specifika z pohledu implementace architektury a rychlého vývoje nových funkcionalit. Ukážeme si, jak nejlépe nachystat serverovou část pro aplikaci tohoto typu. Všechny zde prezentované informace budou vycházet z mé osobní zkušenosti z praxe, nejedná se tedy o návod na dokonalý server.

5.1 Implementace datové vrstvy

Implementace serverové části probíhala podle zásad popsaných v předchozích kapitolách. Ze začátku implementace bylo potřeba nastavit projekt na potřebné knihovny a framework. Následovalo vytvoření prázdné databáze pro projekt. Databáze byla vytvořena prázdná, jelikož napsání vlastního framework nad Dapper jsem zamýšlel od samotného počátku. Jako další bylo tedy potřeba vytvořit samotný framework, který se bude starat o interakci s databází.

5.1.1 Implementace rozšíření Dapper

Jelikož Dapper nám poskytuje základní sadu operací nad databází, tak bylo možné se zaměřit na samotný framework. Cíl toho framework byl předem známý a celkem jasný. Méně jasné bylo samotné řešení. Bylo potřeba zachovat minimální časovou zátěž, ale zároveň implementovat většinu potřebných funkcí. Bezesporu se jednalo o moji největší osobní výzvu v rámci celé aplikace.

Úplně jako první vznikly dva databázové dotazy, které nám umožňují získat a prozkoumat celou strukturu konkrétní databáze. Jednalo se o dotazy na získání schématu databáze, které obsahuje výpis tabulek v databázi. Pak šlo o získání názvů sloupců v jednotlivých tabulkách. S těmito informacemi jsme byli schopní porovnávat strukturu aplikace a databáze.

Pro získání struktury aplikace bylo potřeba prohledat celý projekt na třídy označené značkovacím rozhraním. Pomocí reflexe jsme tedy načetli všechny exportované typy (jednotlivé třídy) a následně je porovnali na existenci našeho rozhraní. Po získání všech tříd určených k mapování do databáze jsme iterováním přes jejich kolekci mohli začít vytvá-

řet seznam jmen sloupců a datových typů. Pomocí reflexe jsme pro každou třídu vytvořili generickou metodu s jejím typem a následně ji zavolali. O mapování každé třídy se stará algoritmus 5.1.

```
private void AutoMapping(){
    var interfaces = typeof(T).GetInterfaces();
    _keyTypeSet = new KeyTypeSet();
    List<PropertyInfo> properties;
    var sql = string.Empty;
    if (interfaces.Any(i => i == typeof(IAutomaticMapper)))
    {
        properties = typeof(T).GetProperties().Where(p =>
            !p.IsDefined(typeof(MapperIgnore), false) && p.GetSetMethod() !=
                null &&
            !(p.PropertyType.IsClass && p.PropertyType !=
                typeof(string))).ToList();

        if (!_tables.Contains(typeof(T).Name))
        {
            properties.ForEach(GetKeyType);
            sql = MapTable<T>(properties, true);
            _tables.Add(typeof(T).Name);
        }
        else
        {
            var columns = GetTableColumns(typeof(T).Name);
            var notInProperties = columns.Where(c => !properties.Select(p =>
                p.Name).Contains(c)).ToList();
            var notInDatabase = properties.Where(p =>
                !columns.Contains(p.Name)).ToList();

            if (notInDatabase.Any())
            {
                notInDatabase.ForEach(GetKeyType);
                sql = MapTable<T>(notInDatabase, false);
            }

            if (notInProperties.Any())
            {
                notInProperties.ForEach(c => _keyTypeSet.DeleteNames.Add(c));
                sql = MapTable<T>(properties, false, true);
            }
        }
    }

    if (sql != string.Empty)
    {
        DbConnection.Open();
        DbConnection.Execute(sql);
        DbConnection.Close();
    }
}
```

Výpis 5.1: automatické mapování atributů

Metoda MapTable<T> se pak postará o vytvoření SQL skriptu, který vytvoří nebo aktualizuje tabulku. Při vytváření skriptu se získávají dodatečné informace, jako třeba datový typ, zda se jedná o cizí nebo primární klíč atd. V algoritmu jsou vidět ještě další použité metody. Všechny tyto metody jsou psány genericky a pomocí reflexe. Nebudu je zde již uvádět, jsou dostupné v přiloženém zdrojovém kódu. Tento algoritmus měl za cíl pouze ukázat možnosti reflexe, kdy s pomocí pár řádků kódu jsme schopní získat spoustu informací o vnitřním stavu aplikace za běhu.

Základní třídy

Jak již bylo mnohokrát zmíněno, serverová část aplikace byla psána tak, aby umožňovala rychlý vývoj nových funkcionalit. Rychlý vývoj je možný díky malému počtu řádků napsaného kódu. S větším počtem řádků kódu roste riziko, že se nám v tomto kódu objeví chyba. Pro snížení počtu řádků kódu jsem se rozhodl využívat základní třídy (base class), abstraktní třídy (abstract class) a rozhraní.

Abstraktní třídy obsahují základní implementace operací. Jedná se o operace, které jsou potřeba u většiny modelů. Všechny metody jsou označeny jako virtuální, v případě potřeby jejich rozšíření je můžeme přepsat. Jelikož se jedná o webovou aplikaci, pak většinu metod potřebujeme ve dvou verzích, synchronní a asynchronní. Na úrovni mapovací vrstvy (označena jako mapper) se nachází abstraktní třída s následujícími metodami:

- Insert
- InsertAsync
- Delete
- DeleteAsync
- Update
- UpdateAsync
- GetAll
- GetById

Můžeme si povšimnout, že se jedná o metody CRUD (Create, Read, Update, Delete). Všechny tyto metody volají základní třídu. Základní třída již obsahuje konkrétní implementaci jednotlivých metod.

Na úrovni repozitáře se jedná o stejný výčet CRUD metod. Každá z těchto metod pouze volá stejnou metodu přes mapper. Mapper je vždy pro konkrétní třídu, tato třída je nastavena jako generický parametr.

Mapper i repozitář mají pro své abstraktní třídy vytvořené rozhraní. Toto rozhraní slouží k dědičnosti. Pokud vznikne nová třída, kterou chceme mít v databázi jako tabulku, tak vytvoříme mapper a repozitář, který bude dědit z daného generického rozhraní pro konkrétní třídu. V tuto chvíli máme k dispozici všechny CRUD metody nad danou tabulkou.

V případě, že vyžadujeme pokročilejší operaci, tak přidáme její hlavičku do rozhraní a implementaci do konkrétní třídy. Tato implementace má na úrovni mapper k dispozici databázové operace.

5.2 Implementace aplikační vrstvy

Přístup k implementaci aplikační vrstvy byl stejný jako pro datovou vrstvu. Pro doménové služby (domain service) obsahuje abstraktní a základní třídy. Tyto třídy opět obsahují základní CRUD operace a to jak synchronní, tak asynchronní. Většina těchto metod ale byla na úrovni domény přepsána, bylo totiž potřeba přidávat dodatečné informace či výpočty. Doménová služba vždy volá pouze repozitář s daty. Na úrovni domény je ještě potřeba konvertovat data mezi databázovým modelem a aplikačním modelem. Pro tyto účely je zde použita knihovna AutoMapper. Tato knihovna nám umožňuje konvertovat třídy mezi sebou, pokud pro ni vytvoříme mapovací funkci. Mapovací funkce obsahuje informaci o tom, který atribut z jedné třídy se mapuje na jiný atribut třídy druhé. Pro zjednodušení práce jsem mezi databázovými a aplikačními modely dodržoval stejné pojmenování atributů. Díky tomu bylo možné napsat další základní třídu, která provede automatické mapování bez nutnosti vytvářet mapovací funkci.

Na úrovni kontroléru je implementována autorizace. Tato autorizace je společná pro celý kontrolér. Pokud vyžadujeme povolit jinou autorizaci pro konkrétní API metodu, tak je možné přepsat pravidla kontroléru pomocí atributu metody. Většina operací požaduje informace o aktuálně přihlášeném uživateli, který danou operaci spustil. Pro tyto účely jsem napsal další základní třídu, která provede extrakci těchto informací z poskytnutého JWT tokenu. Tento token se totiž z klienta posílá při každém volání API.

Poslední částí aplikační vrstvy je globální metoda, která se zavádí při spuštění aplikace jako první. Tato metoda provádí všechny nezbytné operace pro chod aplikace. První součástí je vytvoření základních rolí v systému. Následuje vytvoření kontextu pro chat. Tento kontext je udržován na úrovni aplikace a následně synchronizován s databází. V rámci aplikace je držen jako statický kontext. Tento kontext je zde kvůli tomu, aby nedocházelo ke zbytečnému zpoždění zápisem do databáze. Z tohoto kontextu je navíc možné získávat všechny informace o aktuálních uživatelích připojených k chatu. Poslední částí je **vkládání závislostí**(Dependency injection - DI).

5.2.1 Vkládání závislostí

Vkládání závislostí neboli Dependency injection se řadí mezi návrhové vzory. V tomto návrhovém vzoru se místo instance třídy používá její rozhraní. Můžeme tedy vytvářet závislosti mezi komponentami, aniž by bylo potřeba mít na danou komponentu referenci během překladu. Tato technika má široké využití. Například pokud v rámci třídy potřebujeme instance jiné třídy, tak je můžeme předat přes konstruktor pomocí jejich rozhraní. Případně za běhu můžeme vyvolat kontejner, který obsahuje všechna rozhraní, a získat konkrétní instanci. Výhodou je, že se v celé aplikaci postavené na vkládání závislostí nevyskytuje konkrétní implementace třídy. Na žádném místě totiž není volán konstruktor některé z tříd.

Vkládání závislostí jsem se rozhodl použít z toho důvodu, že nám umožňuje lehce měnit konkrétní implementace tříd, pokud zachováme rozhraní. Takže kdybych se například rozhodl změnit mapper na mapper spolupracující s Oracle databází na místo MSSQL databáze, tak stačí změnit provázání mezi rozhraním a konkrétní implementací třídy. Každé rozhraní je totiž na začátku běhu aplikace vloženo do kontejneru a provázáno s definovanou implementací třídy. K tomuto účelu mi posloužil vestavěný kontejner v .NETCore framework.

Všechny konstruktory v aplikaci přijímají pouze rozhraní, které definuje metody. Za běhu se místo rozhraní vloží do konstruktoru provázaná implementace a metody se volají

nad touto implementací. Částečnou nevýhodou je nutnost udržovat seznam metod v rozhraní a ve třídě stejný.

5.3 Implementace klientské části

Po napsání většiny serveru jsem mohl začít psát klientskou část. Většina navržených a potřebných funkcí byla na serveru napsána a otestována. Bylo tedy již jen potřeba je provázat s klientem. Ale implementace klientské části aplikace byla pro mě velkou neznámou. Netušil jsem, co mám očekávat, ani jak to bude náročné. Prvním důvodem bylo, že nikdy jsem nedělal front-end aplikace. V rámci každého projektu jsem pracoval pouze jako back-end vývojář, protože back-end technologie jsou mi daleko bližší než HTML, Javascript a CSS. Navíc mé grafické cítění je nedostatečné. Druhým důvodem bylo, že v době začátků psaní byl framework Blazor zároveň na úplném začátku svého vývoje. Kromě strohé dokumentace nebyly dostupné téměř žádné materiály. Nejednou jsem byl nucen položit dotaz na StackOverflow, abych nalezl odpověď i na triviální otázku.

Začal jsem tedy prozkoumávat Blazor a jeho funkcionality. Brzy jsem zjistil, že se jedná o relativně lehký framework, pokud už máme nějaké zkušenosti s vývojem v C#. Navíc Blazor funguje jako *Single page application* - *SPA*. SPA značí typ webových aplikací, které používají server pouze jako úložiště dat, a všechny operace se provádějí na straně klienta. U SPA není potřeba vždy stahovat kompletní kód stránky, ale mohou se pouze aktualizovat data, která se změnila. Je zde tedy rychlejší odezva a načítání stránek. Z toho těží hlavně uživatel, kterému se zdá chod webové aplikace plynulý. Aplikace se pouze při prvním pokusu o načtení může načítat delší dobu, jelikož je potřeba stáhnout všechny zdrojové kódy stránky ze serveru na klienta. S těmito informacemi jsem se tedy rozhodl pustit do vývoje. Pro ukázku si nejdříve ukážeme úsek zdrojového kódu v Blazor.

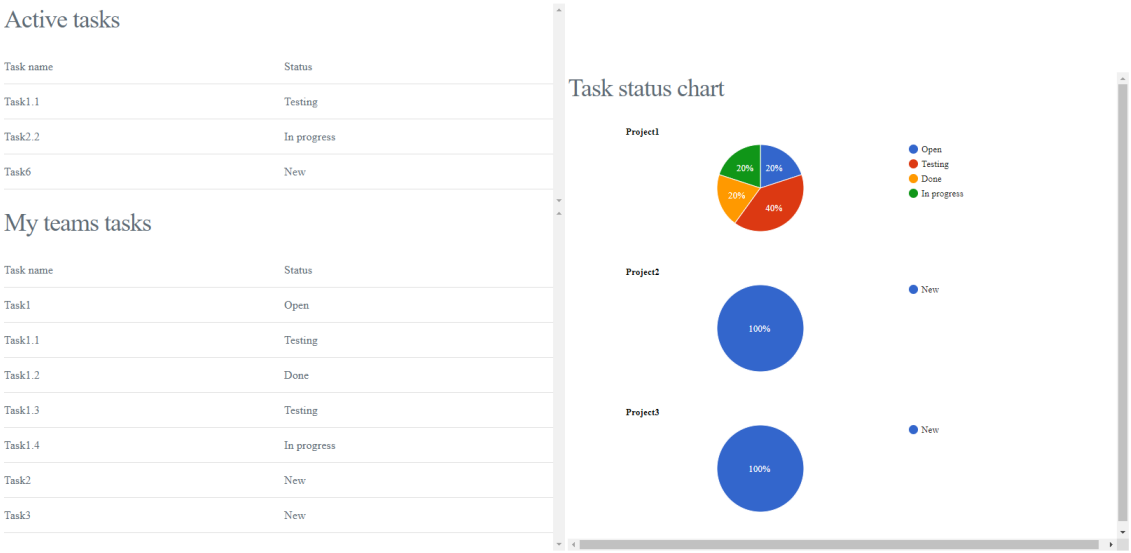
```
<div class="col-md-5">
    <h2>Active tasks</h2>
    <table>
        <thead>
            <tr>
                <td>Task name</td>
                <td>Status</td>
            </tr>
        </thead>
        <tbody>
        @foreach (var assignedTask in _assignedTasks.Where(t => t.StatusId !=
            (int) ProjectTaskStatusEnum.Closed))
        {
            <tr onclick="@(() => ProjectTask_click(assignedTask.Id))"
                style="cursor: pointer">
                <td>@assignedTask.ProjectTaskName</td>
                <td>@assignedTask.StatusDescription</td>
            </tr>
        }
        </tbody>
    </table>
</div>

@functions{
private IList<ProjectDto> _managedProjects = new List<ProjectDto>();
```

```
    private IList<ProjectTaskDto> _assignedTasks = new List<ProjectTaskDto>();

    public void Dispose()
    {
    }

    protected override async Task OnInitAsync()
    {
        await LoadProjects();
        await RenderPieChart();
    }

    private async Task LoadProjects()
    {
        _managedProjects = await
            Http.GetJsonAsync<List<ProjectDto>>("api/project/GetAllProjectsManagedByUser");
        _assignedTasks = await
            Http.GetJsonAsync<List<ProjectTaskDto>>("api/projectTask/GetAllProjectTasksAssignedF
        StateHasChanged();
    }

    private async Task RenderPieChart()
    {
        if (!_managedProjects.Any())
        {
            return;
        }

        var tasks = _managedProjects.Select(p => CreateChart(p.Id));
        await Task.WhenAll(tasks);
    }

    private async Task CreateChart(int projectId)
    {
        var chartData = await
            Http.PostJsonAsync<List<PieChartDataDto>>("api/bi/ProjectTaskStatus",
            projectId);
        if (chartData.Any())
        {
            await JSRuntime.Current.InvokeAsync<bool>("projectTaskStatusChart",
                chartData);
        }
    }

    private void ProjectTask_click(int projectTaskId)
    {
        UriHelper.NavigateTo("taskDetail/" + projectTaskId);
    }
}
```

Výpis 5.2: Ukázka Blazor kódu

Na ukázce5.2 je vidět aktuální algoritmus pro zobrazování přehledu projektů na *Dashboard*. Na začátku můžeme vidět kód připomínající HTML, všimněme si ale odchylek, které v HTML není možné nalézt. Jedná se konkrétně o metody ze C#, vidím zde například

foreach cyklus, kterým generujeme řádky tabulky. Dále je zde máme *onclick* událost, která pomocí lambda výrazu spustí *ProjectTask_click* metodu s parametrem *assignedTask.Id*. Následuje sekce *@functions*, která slouží pro definování funkcí v jazyce C#, a to úplně stejně, jak jsme zvyklí na úrovni serveru. V asynchronní metodě *CreateChart* je ještě umístěno volání Javascript Interop služby pomocí *JSRuntime*. V poslední metodě je vidět, jak je voláno routování mezi stránkami. Výslednou stránku můžeme vidět na obrázku 5.1.



Obrázek 5.1: Dashboard stránka

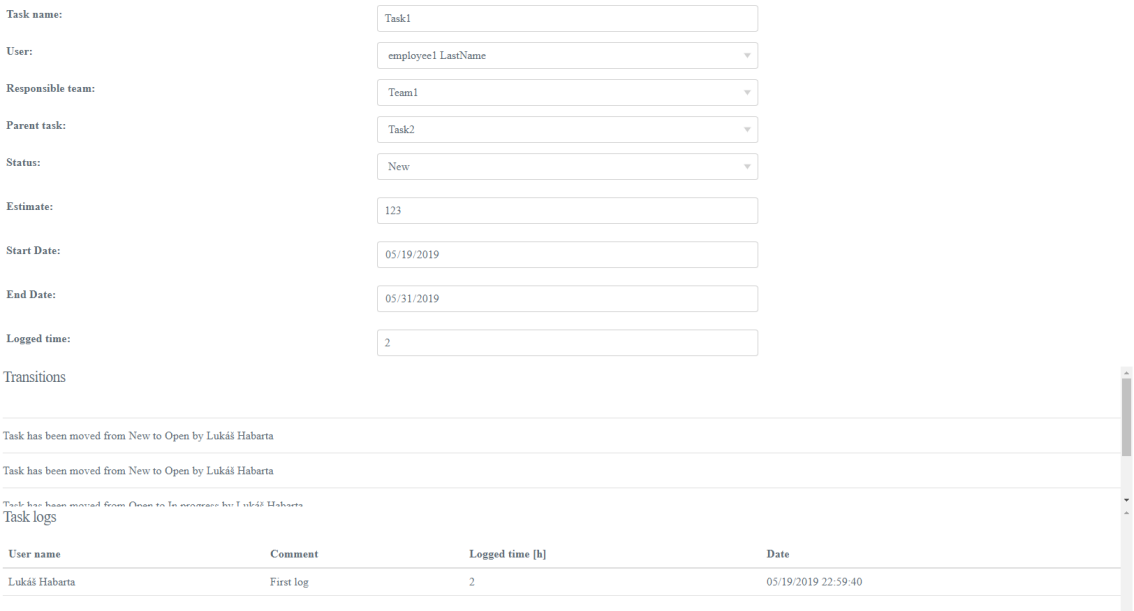
Po prozkoumání kódu můžeme usoudit, že Blazor je opravdu jednoduchý framework, kde pomocí C# a Razor dokážeme vytvořit webovou stránky rychle a hlavně typově bezpečně. Můžeme využívat opravdu všechny metody a funkce ze C#. Výhodou je především využití metod z třídy *Task*, která slouží pro práci s asynchronními operacemi a LINQ. V Javascript je *Promise* částečným ekvivalentem pro třídu *Task*. Třída *Task* je ale daleko pokročilejší ohledně funkcionality a poskytuje nám ještě větší kontrolu nad asynchronními operacemi. Pro LINQ není v Javascript žádný ekvivalent, pokud nepoužijeme dodatečné knihovny, které tyto funkce implementují. LINQ nám poskytuje pokročilé dotazování nad kolekcemi objektů.

Proběhlo vytvoření testovací komponenty se spoustou předpřipravených dat, která sloužila jako místo pro testování funkcionalit před jejich oddělením do samostatné stránky či komponenty. Po otestování funkcionality na této komponentě proběhlo vždy oddělení do samostatné stránky, kde se následně odstranila testovací data a připojila se na server.

Vývoj stránek jsem dělal podle postupné dekompozice systému. Nejdříve tedy proběhlo vytvoření stránek nutných pro běh aplikace. Byla vytvořena přihlašovací stránka, stránka pro správu uživatelů, odhlášení z aplikace. Všechny stránky byly jednoduché, jednalo se totiž jen o čtení a zápis dat z formulářů na server. V tento moment byl nachystán základ pro běh aplikace na klientské úrovni, následovalo tedy vytvoření stránky pro projekt. Vytvoření takové stránky nebyl problém, jelikož stránka obsahuje pouze přehled všech projektů v systému a jednu komponentu pro vytváření nových projektů. Následovalo vytvoření stránky pro správu a vytváření úkolů. Z pohledu klienta se jedná o nejobsáhlejší stránku. Na této stránce bylo potřeba vytvořit přehled všech úkolů přiřazených na přihlášeného uživatele a v případě projektového manažera i přehled úkolů, které jsou součástí jeho projektů,

obrázek 5.2. Vytváření úkolů bylo původně řešeno pomocí komponenty, ale pro účely editace projektů a zobrazení dodatečných informací bylo vhodnější nahradit komponentu vlastní stránkou. Pro přechod na jinou stránku s konkrétními údaji bylo potřeba použít routování stránek. Naštěstí v Blazor se jedná o triviální činnost, stačilo rozšířit stránku o parametr vstupu. Detail úkolu je vidět na obrázku

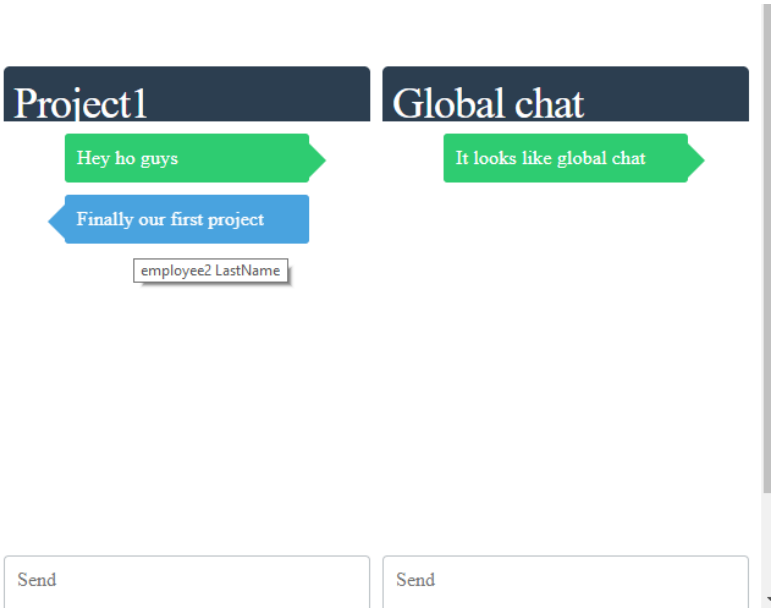
Obrázek 5.2: Přehled úkolů



Obrázek 5.3: Detail úkolu

Po vytvoření stránek pro zobrazování projektů a úkolů jsem se rozhodl splnit první náročnější požadavek. Bylo potřeba vytvořit chatovací rozhraní. Pro tyto účely jsem rozšířil Blazor o SignalR knihovnu a začal vytvářet generování jednotlivých chatovacích skupin. Vznikla tedy jedna globální skupina následována navázáním skupiny na vytvoření nového projektu. Po úspěšném vytvoření těchto skupin zbývalo jen vytvořit grafické zobrazení

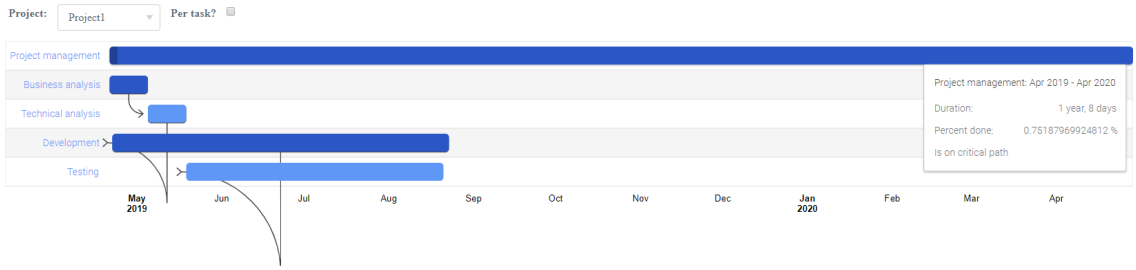
chatu. Inspiroval jsem se tedy populárními aplikacemi a rozhodl jsem se pro samostatná okna zobrazená na stránce jako plovoucí. Bohužel zde jsem narazil na první z problémů. Mé znalosti CSS a HTML nebyly dostatečné na vytvoření plovoucí komponenty napříč aplikací. Vytvořil jsem tedy pro jednotlivá chatová okna vlastní stránku, kde jsou zobrazeny. Chat byl tedy funkční, ale trpěl nedostatkem odesíláním zprávy pomocí tlačítka. Ač se jedná o banální problém, tak do odevzdání práce jsem nebyl schopný nahradit tlačítko za stisk klávesy *Enter*. Vinu bohužel v tomto případě nese Blazor. V této rané fázi vývoje nemá Blazor podporované zachytávání všech událostí na uživatelské úrovni, respektive podporuje většinu událostí, ale neumožňuje číst jejich data. Je tedy možné odchytit události na stisknutí klávesy, ale není možnost jak získat data obsažená v konkrétním textovém poli. Pole jsou totiž generována bez identifikátorů, takže ani vyvolání Javascript Interop služby by nám nebylo nápomocné. Funkcionalita chatu je tedy kompletní, pouze s malým estetickým nedostatkem. Grafická reprezentace chatu je na obrázku 5.4. Je zde zobrazen globální chat a chat v rámci projektu.



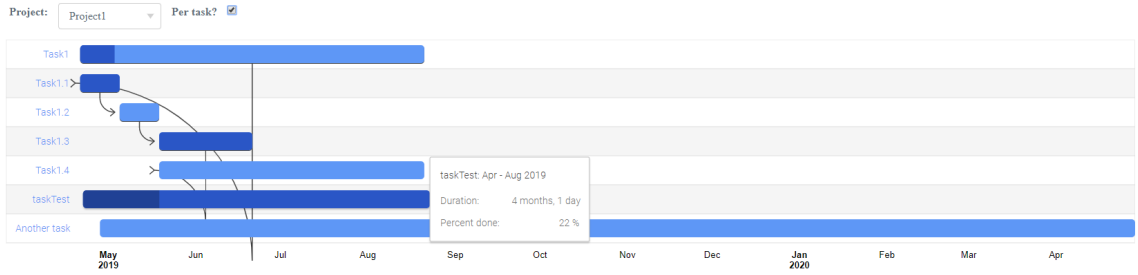
Obrázek 5.4: Chatovací rozhraní

Následovalo vytvoření komplikovanější části aplikace, která bude vyžadovat volání Javascript Interop služby. Bylo již potřeba zkusit vytvořit grafické zobrazení stavu v aplikaci. V první fázi jsem se rozhodl vytvořit jednoduchý koláčový graf pro zobrazení počtu projektů na uživateli. K vykreslování grafu jsem zvolil Google chart API. Toto API umožňuje strukturovaná data převádět na různé druhy grafů. Jednalo se o jednoduché a efektivní řešení tohoto problému. Jediným větším nedostatkem je, že Google Chart API požaduje volání z Javascript. V téhle situaci jsem tedy byl nucen využít mnou zavrhovaný Javascript a k tomu potřebnou Javascript Interop službu. Samotné volání služby nebyl problém, jedná se o jeden jediný řádek kódu v Blazor. Blazor požaduje, aby všechny Javascript funkce byly součástí globální proměnné *window*. Jedná se o pochopitelný požadavek. V Javascript máme pro vnější volání dostupné pouze funkce umístěné v globálním *closure* (český překlad by odpovídal uzavření, ale není moc popisný). Jen umístění této proměnné v rámci aplikace není ideální. Po vytvoření mapovací funkce pro data na straně Javascript se graf povedlo vykreslit podle očekávání. Následovalo tedy vytvoření Ganttova diagramu pomocí stejné

knihovny. Data pro Ganttův diagram jsou přichystána na straně serveru pomocí kombinací dotazu na databázi. Z těchto dat je poskládán objekt, který je přes API předán na stranu klienta a odtud je delegován přes Javascript Interop službu do Javascript funkce. V Javascript funkci jsou data namapována na objekt požadovaný na straně Google Chart API. Objekt je nutné mapovat, jelikož v C# je konvence používat velké počáteční písmeno v názvu atributů tříd, Javascript ale vyžaduje pro atributy malá počáteční písmena. Výsledek Ganttova diagramu můžeme vidět na obrázku 5.5, který zobrazuje Ganttův diagram pro projekt z pohledu závislostí. Na obrázku 5.6 je vidět náhled na stejný projekt jen z pohledu jednotlivých úkolů a jejich návaznosti.



Obrázek 5.5: Ganttův diagram z pohledu závislostí



Obrázek 5.6: Ganttův diagram z pohledu návaznosti jednotlivých úkolů

Poslední důležitou částí bylo vytvoření závislostí pro jednotlivé typy úkolu. Tyto závislosti jsou prezentovány pouze jako tabulka, její obsah je ale velmi důležitý pro vykreslování Ganttova diagramu. Po dokončení závislostí následovalo doladění detailů ohledně obsahu dat v jednotlivých formulářích, základní stylování komponent pomocí CSS a přidání dodatečných omezení na straně klienta.

Samotný vývoj klienta nebyl nikterak náročný. Problematické byly pouze technologie, jelikož byly pro mě nové. Funkcionálně je totiž klient pouze zobrazovač dat vytvořených na serveru. Většina logiky se tedy nachází na serveru. Jelikož se jedná o aplikaci zpracovávající tok dat v čase (vytváření projektu, vytváření úkolů, přechod mezi stavy na projektu, ...), tak kromě vykreslování grafů a chatovacích oken nebylo potřeba data na klientovi modifikovat. Tento přístup má výhodu v minimálním zatížení klienta. Veškerá zátěž se totiž odehrává na serveru. Toto odpovídá typickému popisu Single Page Application.

5.4 Testování

Testování je obecně považováno za jednu z důležitých součástí vývoje. Testování je možné provádět automaticky nebo manuálně. Automatické testy můžeme rozdělit do tří kategorií. Unit testy nám testují samostatné jednotky kódu. Jednotkou je například konkrétní metoda. Dále zde máme integrační testy, tento typ testů se nám snaží pokrýt společné fungování jednotlivých jednotek. Testujeme tedy například celou komponentu či celý modul. Poslední kategorií jsou API testy. Tyto testy jsou velmi podobné integračním, jedná se tedy o komplexní testování aplikace. Simulujeme pomocí testu interakce uživatele se systémem. Posíláme požadavky na API a validujeme odpověď, zda odpovídá očekávaným výsledkům.

Aplikaci bylo během vývoje tedy potřeba postupně testovat. Nepostupoval jsem ovšem podle manuálu testera, nejednalo se tedy o systematické a pravidelné testování všech funkcionalit a jejich provázání. Zvolil jsem testování pouze nově přidaných funkcionalit. Po větším množství vyvinutých funkcionalit následovalo regresní testování všech funkcionalit. Po dokončení vývoje byl udělán rychlý smoke test. Regresní testování představuje otestování celé aplikace od začátku až do konce. Probíhá bez použití předdefinovaných testovacích dat. Jsme tedy nuceni použít každou komponentu v systému. Díky tomu můžeme odhalit dostatek chyb, které mohou vznikat až v návaznosti na několik po sobě jdoucích akcí. Smoke test je označení pro rychlý závěrečný test. Obvykle se provádí před odevzdáním hotové aplikace zákazníkovi. Během smoke testu se provádí testování pouze hlavních komponent systému. Očekává se totiž, že většina chyb je již opravena.

Pro účely testování jsem průběžně přidával testovací data do SQL skriptu. Tento skript obsahuje základní data pro všechny entity, včetně provázání mezi entitami. Skript není nijak závislý na přednastavených identifikátorech na straně databáze. Všechny primární klíče jsou generovány automaticky stejně jako provázání přes cizí klíče.

Jelikož se jedná o aplikaci ve stylu Proof of Concept, rozhodl jsem se ignorovat minoritní chyby, které neovlivňují chod aplikace. Přístup proof-of-concept totiž takové chyby povoluje. Mezi povolené chyby patří například špatný text u pole nebo třeba rozdílné formátování datumů napříč aplikací. Dále nejsou vytvořeny žádné unit testy ani integrační testy. Testování tedy probíhalo pouze ručně. Manuální testování je sice časově náročnější, ale při provázanosti jednotlivých komponent je spolehlivější k odhalení defektu v aplikaci. Nejproblematičtější částí na testování byly právě návaznosti, a to jak v projektech a úkolech, tak v Ganttově diagramu. Bylo potřeba kontrolovat obsah, který je generován pomocí formulářů a nastavení na několika stránkách.

Kapitola 6

Závěr

V rámci této diplomové práce se mi podařilo implementovat systém, který odpovídal zadaným požadavkům. Systém není moc prakticky použitelný bez dalšího vývoje. Aplikace zatím poskytuje pouze základní vzorek funkcionalit, které jsou pro řízení projektu potřeba. Některé funkcionality, jako Ganttův diagram nebo chatovací rozhraní, byly od kolegů hodnoceny kladně. Záporně bylo hodnoceno uživatelské rozhraní, které není dobře graficky provedeno ani příliš intuitivní. Toto ale odpovídá mému očekávání, jedná se totiž o první aplikaci, kde jsem byl nucen pracovat i na vývoji uživatelského rozhraní. Daleko větší jistotu mám při práci na serverové části.

Jak již bylo zmíněno, aplikace poskytuje pouze základní množinu operací. Pokud bych se rozhodl ve vývoji aplikace pokračovat, bylo by nutné přidat velké množství funkcí. Díky propracované serverové části by jejich vývoj byl ovšem značně rychlejší než u nového systému. Aby se tato aplikace mohla rovnat aplikacím jako JIRA nebo Easy project, pak by bylo potřeba několik let vývoje a značně velký vývojářský tým. Rozhodně bych zvážil propracování uživatelského rozhraní. Dále bych se rozhodl pro dynamické nastavení, které umožní přizpůsobovat aplikace konkrétním požadavkům pomocí nastavení. Nebylo by tedy potřeba přidávat jednotlivé funkcionality přímo do kódu, ale každý uživatel by měl možnost si definovat vlastní.

Aplikace ovšem poskytuje základní pohled na řízení agilních projektů. Umožňuje nám spravovat uživatele, vytvářet týmy, vytvářet projekty a úkoly. Uživatele nebo týmy je pak možné přiřazovat jako zodpovědné či vedoucí osoby na jednotlivé projekty a úkoly. Pro uživatele poskytuje přehled aktivních úkolů. Pro projektové manažery zde máme stejný přehled jako pro běžného uživatele, ale navíc rozšířený i o grafické zobrazení stavu jednotlivých projektů. Je zde navíc Ganttův diagram, který nám umožňuje dvě různá zobrazení návazností. Pro všechny uživatele v systému je tu navíc možnost komunikovat v reálném čase pomocí chatu.

Samotné řešení aplikace z hlediska kódu a provedené implementace je rozhodně kvalitní. Obě části, serverová i klientská, nám umožňují pokračovat ve vývoji, aniž by byl potřeba na začátku rozsáhlý zásah do zdrojových kódů. Díky této kvalitní implementaci je systém dlouhodobě udržitelný a je odolný vůči změnám, které by byly potřeba. Výslednou implementaci z hlediska kódu a funkcionality hodnotím kladně a jako životaschopnou aplikaci.

Literatura

[1] Agilemanifesto: *Manifest agilního programování*. [Online; navštíveno 01.04.2019]. URL https://agilemanifesto.org

[2] Doležal, J.; Máchal, P.; Lacko, B.: *Projektový management podle IPMA*. Grada, 2012, ISBN 978-80-247-4275-5.

[3] Google: *Google charts*. [Online; navštíveno 01.04.2019]. URL https://developers.google.com/chart/interactive/docs/gallery/ganttchart

[4] Josef Hajkr and Jan Havlík and Pavel Máchal and Michael Motal and Igor Novák and Zdenko Staníček: *Výkladový slovník pojmů*. [Online; navštíveno 01.04.2019]. URL https://www.ipma.cz

[5] MangementMania kolektiv: *Aktuální informace k řízení projektů*. [Online; navštíveno 01.04.2019]. URL https://managementmania.com

[6] Máchal, P.; Ondrouchová, M.; Krunčíková, I.; aj.: *Mezinárodní standard projektového řízení podle IPMA ICB v.4*. IPMA Česká republika, z.s., 2017, ISBN 978-80-7326-286-0.

[7] Oxford University: *Oxford dictionary*. [Online; navštíveno 01.04.2019]. URL https://en.oxforddictionaries.com/definition/process

[8] Patrick Weaver: *THE ORIGINS OF MODERN PROJECT MANAGEMENT*. [Online; navštíveno 01.04.2019]. URL https://mosaicprojects.com.au/PDF_Papers/P050_Origins_of_Modern_PM.pdf