Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Aplikace pro oblastní charity v ČR

Diplomová práce

Autor: Bc. Daniel Vondra
Studijní obor: Aplikovaná Informatika

Vedoucí práce: Ing. Barbora Tesařová Ph.D.
KIKM

Hradec Králové 04 2022

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

……

V Hradci Králové dne 29.4.2022 Bc. Daniel Vondra

Poděkování:

Děkuji vedoucí diplomové práce, Ing. Barboře Tesařové Ph.D., za metodické vedení práce a své rodině, přátelům a celému akademickému sboru, kteří mi při diplomové práci velice pomohli a podporovali mě.

Anotace

Diplomová práce se zaměřuje na vytvoření webové a mobilní aplikace, specializující se na výpomoc v dobrovolnických centrech. Účelem obou aplikací je umožnit zrychlené získání a snadnější přenos informací o provedených činnostech organizací. Sdružovat dobrovolníky pod jednotlivé oblastní charity a mít kontrolu nad jejich prací, pomocí vlastních koordinátorů, modernějším způsobem. Backendová část webové aplikace je psána pomocí jazyka PHP a frontendová část je psána pomocí latte. Zatímco mobilní aplikace je psána pomocí jazyka Java a vzhled pomocí XML zápisu. Mobilní aplikace pak vyžívá pomocí API požadavků backend webové aplikace.

Annotation

Title: Applications for regional charities in the Czech Republic

The diploma thesis focuses on the creation of web and mobile application that should be helping in volunteer centers. The purpose of the applications is to enable faster acquisition and easier transfer of information about the performed volunteer activities. They make it possible to associate volunteers under individual volunteer organizations and to have their coordinators control their activities in a more modern way. The backend part of the web application is written by using PHP and the frontend part is written by using latte. While the mobile application is written by using Java and the appearance is written by using XML notation. The mobile application then uses the backend requirements of the web application.

Obsah

Seznam obrázků

1 Úvod

Tato diplomová práce na téma „Aplikace pro oblastní charity v ČR“, svým obsahem navazuje, na již existující práci, která byla vytvořena jako součást závěrečné zkoušky na střední škole VOŠ a SPŠ Jičín a podstatným způsobem tuto práci přepracovává a rozvíjí.

V rámci této práce byl vytvořen program, jehož účelem je usnadnit rutinní činnost koordinátorů v místní jičínské charitě s možným dalším rozšířením mezi další oblastní charity.

Pro snazší pochopení důvodů pro vznik, respektive nutnosti vzniku této aplikace, je nezbytné popsat dosavadní formu fungování agendy jednotlivých charit, respektive jejich pracovníků – koordinátorů a dobrovolníků. Veškeré aktivity byly, do doby vzniku aplikace, prováděny za pomoci papírových formulářů, které zachycovaly popis činností a výsledků jak koordinátorů, tak jednotlivých dobrovolníků. Zpracování těchto formulářů bylo jak časově, tak administrativně poměrně náročné a zdlouhavé.

Z výše uvedeného je zřejmé, že mezi klíčové potřeby, respektive požadavky na aplikaci byla možnost, respektive schopnost aplikace propojit koordinátory a jednotlivé dobrovolníky za účelem usnadnění vzájemné komunikace, řízení a dohledu nad činností dobrovolníků, plánování charitativních aktivit, jejich následného reportingu, dokládání provedené činnosti a další.

Během jednotlivých projektových fází byly nejprve se zástupci oblastní charity diskutovány tyto potřeby a očekávání, které byly následně transformovány do formálního zadání aplikace, přesněji do podoby souboru funkčních požadavků. Nefunkční požadavky nebyly, vzhledem k jednoduchosti poptané aplikace, prakticky vůbec předmětem diskuse, nicméně minimální set nefunkčních požadavků zaměřených na bezpečnost aplikace, byl do zadání zahrnut. V následující

fázi byla aplikace úspěšně vytvořena, otestována, a nakonec dodána oblastní charitě k používání.

Vzhledem k datu vzniku této první verze aplikace, technologickému pokroku, a i nutnosti aplikaci rozšířit mezi další oblastní charity, které nový postup chtěly používat, bylo zřejmé, i přes fakt, že aplikace prošla dílčími vylepšeními a aktualizacemi, že existuje řada nedostatků a s tím související prostor pro možné vylepšení či rozšíření o nové funkčnosti, které řeší nová verze aplikace, jenž je zároveň předmětem této diplomové práce.

2 Aplikace pro oblastní charitu – první verze

První verze aplikace vznikla ve spolupráci s oblastní charitou Jičín již v roce 2016. Jednalo se o jednoduchou webovou aplikaci napsanou v jazyce PHP, bez využití dalších technologií, nicméně splňující funkční a technická kritéria a požadavky, které v té době byly na aplikaci kladeny.

Hlavním motivem pro vznik aplikace byla nutnost digitalizace procesu a zároveň vzrůstající množství agendy, která byla v oblastní charitě nedílně spojena s prováděnou dobrovolnickou činností, kdy množství agendy, časová náročnost jejího zpracování a zároveň nutnost následné digitalizace těchto záznamů, přestala být dále udržitelné. Do okamžiku vzniku této aplikace byla veškerá agenda spravována výhradě formou papírových formulářů, které bylo nutno vyzvednout v sídle oblastní charity, dobrovolníci, vykonávající charitativní činnost pro danou charitu, do těchto formulářů zapsali informace o prováděné činnosti, termínu práce a další detaily. Tyto formuláře byly následně odevzdány zpět do sídla oblastní charity, kde je osoba v roli koordinátora zpracovala formou manuálního přepisu do tabulky v MS Excel. Z těchto tabulek byla následně generována data pro potřeby reportingu o prováděné činnosti. Veškeré výše uvedené dokumenty musely být pečlivě archivovány a uchovávány ve fyzické podobě po dobu minimálně 10 let.

Nově vzniklá aplikace umožnila oblastním charitám veškerou agendu plně digitalizovat, byla odstraněna nutnost fyzicky docházet do sídla oblastní charity za účelem vyzvedávání či odevzdávání papírové dokumentace. Veškerá agenda byla díky této aplikaci zpřístupněna online. Digitalizace zároveň do značné míry zjednodušila a zefektivnila dříve rutinní činnosti koordinačních pracovníků, kteří již dále nebyli nuceni data manuálně přepisovat, veškerá data měli k dispozici prakticky ihned a připravena k dalšímu automatizovanému zpracování a reportingu.

Každá aplikace je však životaschopná a konkurence schopná pouze za předpokladu, že probíhá její kontinuální rozvoj a údržba. Tyto aktivity jsou však možné pouze za předpokladu, že aplikace respektuje základní pravidla, architektonické vzory a že náklady či úsilí na její údržbu a rozvoj nepřevyšují rozumnou mez. Pokud žádný z těchto předpokladů neplatí, je zpravidla vhodnější původní řešení opustit a aplikaci vybudovat zcela novou s využitím aktuálních technologií, principů a možností.

Při zpětné analýze původní verze charitativní aplikace byla identifikována řada nedostatků a komplikací. Mezi hlavní bylo možné bezesporu zařadit formu zápisu zdrojového kódu, který odpovídal definici takzvaného „špagetového kódu“, který je pro svou nevhodnou strukturu velmi těžko spravovatelný, rozšiřitelný a pro méně zkušené programátory, či programátory, kteří nepřišli s aplikací dříve do styku, i velmi obtížně pochopitelný. Tento typ zápisu kódu v principu ignoruje modely tříd, neodpovídá architektonickému typu MVC či dalším. S uvážením nevhodné architektury a dalších zásadních nedostatků původního řešení, bylo po dohodě s oblastní charitou rozhodnuto nepokračovat v rozvoji či optimalizacích tohoto řešení, ale nahradit toto řešení aplikací zcela novou, reflektující aktuální požadavky, potřeb a v neposlední řadě i technologické a uživatelské trendy.

3 Cíl práce

Práce na návrhu a implementaci druhé generace aplikace pro oblastní charitu započaly v roce 2021. Ač šlo o individuální činnost, lze z pohledu projektového řízení identifikovat a popsat jednotlivé fáze, které dodávce finální aplikace předcházely:

- Analýza požadavků, specifikace zadání
- Implementace
- Testování
- Nasazení do provozu
- Provoz a rozvoj

3.1 Analýza požadavků

Fázi analýzy požadavků lze rozdělit na fázi definice takzvaných business požadavků, které popisují věcnou podstatu aplikace – její účel a dále fázi technických požadavků, definujících nároky na použité technologie, výkonnost a další. Tato fáze je pro vývoj aplikace zcela klíčová a chyby ve specifikaci, zejména pak ty odhalené až v průběhu využívání aplikace, je velmi složité a časově náročné odstranit.

3.1.1 Business požadavky

Pro správnou analýzu požadavků je nezbytné velmi dobře definovat a popsat procesy, které má aplikace podporovat, identifikovat aktéry, role a uživatele, a samozřejmě jejich odpovědnosti. Proces je velmi náročný pro obě strany, tedy jak pro tvůrce aplikace, tak i pro zákazníka.

Ač lze předpokládat, že zákazník podstatu svého businessu velmi dobře zná, nemusí platit, že ji je schopen potřebným způsobem popsat a vysvětlit. Některé

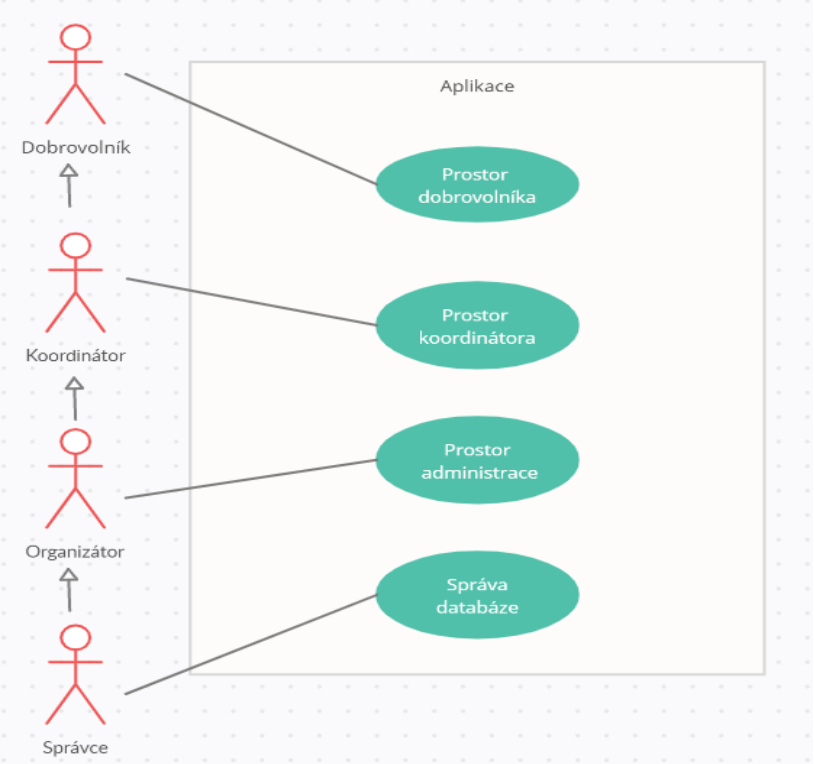
aspekty mu mohou připadat naprosto samozřejmé a v průběhu definice požadavků je tedy nepovažuje za podstatné zmínit, ale zároveň automaticky očekává, že budou součástí finálního řešení. Druhým problematickým bodem může být přílišné zaměření se na nějaký detail či trvání na určité formě řešení takzvaně ze setrvačnosti. Důsledkem pak zpravidla bývá funkčnost, kterou bylo možné řešit výrazně efektivněji nebo snadněji.

Pozice tvůrce aplikace je v této fázi dvojí. Business objednatele nemusí tvůrce znát, což mu situaci komplikuje a je nezbytné, aby byl svými otázkami schopen zákazníka vyzpovídat a zjistit veškeré potřebné informace, tyto informace si zasadil do kontextu a byl schopen dívat se na budoucí řešení jako na celek. Samotná neznalost businessu, může být svým způsobem pro dodavatele paradoxně i výhodou, neb mu umožňuje přemýšlet nad diskutovaným řešením nezaujatě a výrazně kreativněji.

Z diskusí se zadavatelem pak byly identifikovány základní uživatelské role a požadavky. V následujícím obrázku Obr. 1 Rozdělení rolí je vidět jednoduchá struktura, kam v aplikaci mohou jednotlivé role přistupovat. Jedná se o jednoduchý model, kde každá následující role dědí práva a možnosti té předcházející.

Role:

- Koordinátor
- Dobrovolník
- Správce
- Organizátor



Obr. 1 Rozdělení rolí
Zdroj: vlastní tvorba

Požadavky:

- Dostupnost na PC + možnost využití aplikace v terénu (mobilní aplikace)
- Podpora uživatelských profilů a organizací (využití více společnostmi), oddělené pracovní prostory
- Evidence záznamů o dobrovolnické činnosti (založení, správa)
- Reporting

Detailněji se business požadavkům věnuje kapitola číslo **4. Business požadavky.**

3.1.2 Technické požadavky

Fáze definice technických požadavků je, na rozdíl od fáze business analýzy, pro zákazníka, zejména v situaci, kdy není dostatečně odborně vybaven, výrazně

komplikovanější a většina odpovědnosti přechází na osobu která daný prvek, program, vytváří. Ten musí, s uvážením z pravidla velmi obecných požadavků zákazníka, být schopen klientovi doporučit vhodné a efektivní řešení, a to jak z pohledu technického, tak cenového.

V případě zde popisované aplikace byla jako vhodná forma řešení identifikována webová aplikace, a to pro svoji snadnou přístupnost, do značné míry technologickou nezávislost a nenáročnost na straně uživatele, snadnou udržovatelnost a rozšiřitelnost. Značným benefitem je nativní možnost přistupovat k webové aplikace nejen z klasického PC, prostřednictvím webového prohlížeče, ale s uvážením výše uvedených business požadavků i možnost aplikaci používat prakticky z libovolného mobilního zařízení – toto řešení má samozřejmě své limity a negativa v podobě značně omezeného komfortu používání – i z tohoto důvodu vznikla, společně s webovou aplikací, i aplikace mobilní, která je pro využití na mobilních zařízeních uzpůsobena.

Z technických požadavků lze jako nejzásadnější uvést:

- Webová aplikace – PHP, Excel, API
- Mobilní aplikace – Android (kompatibilita s verzí Android 8.0 a vyšší)

3.2 Implementace

Během implementační fáze byly definované požadavky z předchozí fáze převedeny do podoby zdrojového kódu. Ač je fáze implementace jádrem samotného vývoje, možnost teoretického popisu činnosti jako takové, je poměrně omezená. Technické stránce, s důrazem na popis zvolených technologií a jejich konkrétní implementaci v aplikaci, se podrobně zabývá kapitola **5. Použité technologie**.

3.3 Testování

Fáze testování se částečně prolíná s fází implementační, kdy dochází k průběžnému ověřování funkčnosti aktuálně vytvářených částí vznikající aplikace či jejich dopadů do již existujícího řešení. Tuto fázi lze pojmenovat jako vývojářské testování. Ze své podstaty jde o testování poměrně povrchní – vývojář zpravidla testuje pouze bezprostřední dopady aktuálně vytvářeného kódu na ty části aplikace, do kterých zasahuje, neřeší ale testování komplexně.

Za komplexní otestování aplikace je odpovědná fáze takzvaného interního testování – FAT (factory acceptance test). Zcela zásadní informací je, že fáze FAT nemá za úkol pouze ověřit, zda vyrobená aplikace funguje, tedy takzvaně nepadá z důvodu chyb v kódu nebo neošetřených výjimek, ale že je aplikace i kompletní z pohledu funkcionalit, které byly ve fázi analýzy a specifikace se zadavatelem domluveny. V ideální situaci je fáze interního testování prováděna jinou osobou či skupinou osob, než která aplikaci programovala. Důvodem je zajištění nezávislého pohledu na testovanou aplikaci, kdy vývojář může mít, i bez zjevného úmyslu, tendenci testovat primárně a zejména ty testovací scénáře, u kterých ví, že budou úspěšné. Nezávislá testovací skupina je schopna zpravidla i jiného úhlu pohledu, respektive očekává se její schopnost podívat se na aplikaci očima koncového uživatele.

Po úspěšném dokončení interního testování, navazuje chronologicky fáze testování externího – UAT (user acceptance test). V této fázi je aplikace již předána zákazníkovi, zde tedy oblastní charitě a dochází k principiálně totožné kontrole funkčnosti a kompletnosti dodávky, kterou prováděl vývojář, v lepším případě tester, ve fázi interního testování.

V případě této aplikace je pak jak externí, tak i interní testování prováděno zákazníkem. Důvodem je to že je schopen nejlépe určit a otestovat, zdali daný prvek v aplikaci je opravdu to, co původně požadoval.

3.4 Nasazení do provozu

Pokud je dodaná aplikace shledána funkční, může být započato její využívaní koncovými uživateli – koordinátory a dobrovolníky. V případě této webové aplikace, vzhledem k absenci integrace na jiné okolní systémy, či nulovou nutnost aplikaci fyzicky instalovat jednotlivým uživatelům do jejich PC, jde o fázi poměrně jednoduchou a skládající se z nahrání zdrojových kódů aplikace na zvolený webový server, resp. hosting. Tímto krokem je, pro každého uživatele, který disponuje URL adresou, na které je webová aplikace nasazena, fáze nasazení do provozu hotová a může aplikaci začít plnohodnotně využívat.

Výjimkou je pak samozřejmě mobilní aplikace, kterou je třeba formou instalačního balíčku stáhnout a nainstalovat do podporovaných mobilních zařízení koncových uživatelů.

3.5 Provoz a údržba

Provoz a údržba je fáze, která, na rozdíl od fází dříve uvedených, nemusí být nedílnou součástí dodávky aplikace, ale má-li aplikace fungovat ke spokojenosti jejich uživatelů, je nezbytná. Z praktického lze tuto fázi velmi zjednodušeně rozdělit, stejně jako u analýzy, na část technickou a businessovou.

V technické části je sledován provoz aplikace po stránce dostupnosti, stability, využívání systémových zdrojů, rychlosti odezev na jednotlivé uživatelské požadavky a tak dále. V části businessové jsou zpracovávána případná chybová hlášení uživatelů, tedy stavy, kdy fungování aplikace neodpovídá jejímu zadání, resp. specifikaci, případně pak požadavky na změny či rozšíření aplikace – na rozdíl od chyby jde o situaci, kdy aplikace funguje z pohledu specifikace korektně, ale v průběhu užívání jsou identifikována místa, která nebyla v průběhu analytické fáze řešena a to ať už z důvodu opomenutí na straně zadavatele nebo např. z důvodu omezujícího termínu dodání aplikace či nedostatečného rozpočtu, případně řešena

byla, ale po započetí jejich praktického využití byl shledán prostor pro jejich vylepšení či optimalizaci.

4 Business požadavky

Cílem kapitoly je ve větším detailu popsat požadavky, které byly během analytické fáze specifikovány a které tvoří základní pilíře logiky vzniklé aplikace.

V první části se zaměříme na aktéry, tedy typové role a s nimi související funkčnosti:

- Dobrovolník
    - správa vlastního uživatelského účtu
    - založení a evidence docházky
- Koordinátor
    - rozsah funkčností role Dobrovolník
    - správa docházky dobrovolníků
    - správa uživatelů – dobrovolníků
    - reporting
- Správce
    - rozsah funkčností role Koordinátor
    - správa uživatelů – koordinátorů
    - správa organizací
    - migrace DB dat mezi verzemi
- Organizátor
    - rozsah funkčností role Správce s omezením přístupu do DB

Základní požadované funkcionality

Základní požadavek je zejména kladen na jednoduché a přehledné uživatelské rozhraní, ať už se jedná o mobilní nebo webovou aplikaci. Důvodem je to, že obě aplikace budou využívány lidmi různých věkových kategorií, a tak je potřeba, aby byli schopni aplikace používat.

Další rozšiřující funkčnosti je pak samotná správa dalších organizací, oblastních charit, které mohou do aplikace přistupovat. Tato možnost poté tedy

umožní, aby jednu aplikaci mohlo využívat vícero charit, aniž by muselo existovat několik duplicitních verzí jak mobilní, tak i webových aplikací. S tím je samozřejmě spojena nutnost správné správy uživatelských profilů.

Aby mohla výše zmíněná funkčnost pro vícero organizací fungovat správně, je potřeba vytvořit funkcionalitu, které oddělí prostory jednotlivých charit při využití jedné aplikace.

Stěžejním prvkem pak je samozřejmě možnost vytvářet dobrovolnické činnosti a s nimi spojenou jejich správu. Ať už se jedná o jednoduché potvrzení, zamítnutí, či editaci.

Aby poté bylo možné snáze pracovat s těmito vytvořenými záznamy, je potřeba reportovací systém, který umožní uložená data snadno převést do formátu, jako je například MS Excel, čímž umožní snadnou znovu použitelnost a udržitelnost záznamů.

5 Použité technologie

V následujících kapitolách a podkapitolách budou popsány veškeré technologie, které jsou použity v obou aplikacích. Ať se jedná o webovou či mobilní aplikaci, budou zde přiblíženy jejich podstatné prvky a technologie potřebné k vývoji aplikace.

5.1 Webová aplikace

Původní aplikace nesplňovala žádný požadavek moderních webových aplikací i přesto, že byla psána v jazyce PHP, nebylo využito technologie MVC (model, view, controler) a aplikace se různě propojovala a prolínala. Proto je nová aplikace psána za pomocí frameworku, který MVC zaručuje.

V následujících několika blocích budou vysvětleny veškeré technologie, které jsou na aplikaci vázány anebo v ní přímo využity.

5.1.1 PHP

Jedná se o skriptovací programovací jazyk. Tento jazyk je nejčastěji využit pro vytváření dynamických internetových stránek anebo webových aplikací s nejčastěji propojeným formátem HTML. PHP však není vázán pouze na prostor internetu, dají se s ním vytvářet i desktopové aplikace, a k tomu pak slouží jeho kompilovaná verze. U webových aplikací se poté skript provádí na straně serveru a uživatel pak vidí až výsledek jejich činnosti, například v podobě vypsané tabulky dat nebo v podobě zpracovaného obsahu stránky. K jednotlivým skriptům se poté můžeme dostat několika metodami, jako je například pomocí příkazového řádku, http dotazu nebo pomocí webových služeb. PHP je nejrozšířenějším jazykem pro web k červnu 2019. [1]

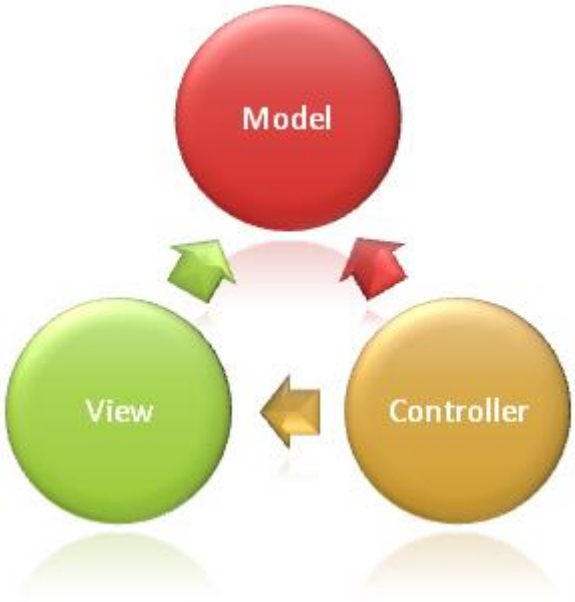
5.1.2 Framework

Aby mohlo být využito správné MVC struktury, bylo zapotřebí využít framework. Pro tyto účely byl navržen nový framework, který bude jednouchý, ale zároveň bude schopen splňovat všechny náležitosti. Celý tento framework stojí na již existujícím frameworku, který je velice populární a rozšířený. Jedná se o framework Nette. Nette díky své struktuře eliminuje velké množství bezpečnostních rizik a díky využití komponent nám umožňuje dosáhnout velké „znovu-použitelnosti“. Díky využití struktury Nette a vlastních prvků nám vznikl framework, který nám dobře dovoluje využívat prvků MVC, AJAX a komunikace s databází. [2]

Jedním s rozšiřujících prvků je struktura využívající komunikaci s databází. Díky rozdělení na jednotlivé části, kterými jsou Manager a Fasáda, jsme schopni snadno komunikovat s jednotlivými části databáze, vznikne přehlednější kód a snižuje se redundance kódu. Díky této struktuře pak každá tabulka má svého Managera a Fasádu. Dalším přidaným prvkem poté je funkční Container, který se nám stará o komunikaci napříč aplikací a také nám umožňuje snadno vyhledávat třídy a reprezentovat je v Prezenterech, které poskytuje samotné Nette.

5.1.3 Schéma

Jedná se o architektonický vzor pro tvorbu (nejčastěji) webových aplikací. Díky jeho navržené struktuře lze pak snadno vyhotovit složitější aplikace, ale přitom dokáže zachovat jasnou strukturu a přehlednost kódu. Jeho základní myšlenkou je poté oddělení logiky od výstupu. Tento přístup nám pak zajistí to, že odstraníme takzvaný „Špagety kód“, kde máme v jednom souboru logické operace a renderování výstupu. Aplikace je poté, jak název naznačuje, rozdělena na Model, View a Controller. [3]



Obr. 2 MVC diagram
Zdroj: [4]

Na obrázku Obr. 2 MVC diagram, je vidět diagram, který reprezentuje základní strukturu a komunikaci mezi jednotlivými prvky v MVC modelu, těmi pak jsou znázorněné kruhy Model, View, Controller.

Model

Veškerá logika, která je potřeba ve webové aplikaci, ať se jedná například o výpočet nebo zpracování dat, je prováděna pomocí model části MVC modelu. Může se jednat jak o dotazy na databázi, výpočty, validace, ale také o spoustu dalších prvků. Model tedy pouze dostane data, která dále zpracuje, a vrátí výsledek. Může se jednat o informace o uživatele, ale Model jako takový neví, od koho data dostal nebo jak budou vypadat na výstupu, pouze zpracuje požadavek a předá data dál. [3]

View

Samotné zobrazení prvků uživateli se všemi náležitostmi je pak prováděno pomocí části MVC, která je nazývána view. Nejčastěji se poté jedná o vykreslenou stránku v html s využitím tagů značkovacího jazyka, pomocí něhož můžeme vypsat ve stránce proměnné. V případě PHP pak můžeme využít samotné PHP anebo

nějakou nadstavbu html, jako je například Latte. View jako takové pak obsahuje naprosté minimum logiky. [3]

Controller

Prostředník, který spojuje dohromady veškeré prvky a umožňuje jim komunikovat, je pak nazýván právě controller. S controllerem také komunikuje uživatel, který do webové aplikace nahlíží a zadává jí data, ať už pomocí formuláře anebo prostřednictvím URL. [3]

5.1.4 Databáze

Pro uskladnění dat bylo potřeba zvolit správný databázový prvek. V souvislosti s tím, že je využito jazyka PHP, bylo zvoleno MySQL jako nejlepší volba. Jedná se o otevřený systém řízení báze dat. MySQL je multiplatformní databáze, která funguje na relačním databázovém modelu. Jako u spousty databází, i u MySQL probíhá komunikace mezi aplikací a databází pomocí jazyka SQL. Aby pak bylo dosaženo co největší bezpečnosti a komfortnosti, jako mezistupeň mezi samotnou databází a aplikace byla využita knihovna Dibi. [5]

Dibi je databázový layer, který nám umožňuje snadno komunikovat se zvolenou databází, ale přidává nám několik dalších výhod. Krom zabezpečené komunikaci a odstínění html prvků při posílání dotazů nám také umožňuje využít dibiFluent. Díky tomuto SQL zápisu pak máme lepší kontrolu na dotazu a větší „znovupoužitelnost“. Samotné SQL dotaz se totiž poté dá psát formou funkcí, jež se dají pouze pospojovat. Díky tomu lze snadno využít například podmínek anebo i cyklů tak, abychom dostali vždy co nejoptimálnější výsledek. Díky struktuře Dibi a našeho frameworku pak lze vytvořit moduly, které vytvářejí automatizované prvky. Těm poté stačí pouze zadat data a následně se funkce v dibiFluent propojí a vznikne dotaz, který je přímo specifikovaný na každou tabulku zvlášť. Řídí se však zadanými daty, a to jen pomocí jednoho předdefinovaného dotazu. [5][6]

Fluent

K SQL dotazům se dá pak dá přistupovat sofistikovaněji, s větší škálou úprav a modifikací, a to právě pomocí fluent způsobu zápisu. Nejedná se o klasický způsob, jak jsme zvyklí u SQL, fluent využívá takzvané tekuté SQL příkazy. Tento přístup nám dovoluje velice snadno modifikovat SQL dotaz tak, abychom dostali co nejlepší výsledky. Dobrým příkladem je pak snadné rozdělení dotazu na několik sekcí, které lze spojovat například pomocí volání funkce, nebo pak doplňování podmínek, aniž bychom využívali SQL. Díky tomu poté lze snadno kontrolovat třeba filtry a na jejich základě pak snadno měnit SQL. Fluent se dá ale také použít například ve for-cyklu, aby se snáze opakoval jeden dotaz, který má pouze pár jiných parametrů. Tyto dotazy poté fluent správně spojí. Jediné, co je u fluentu potřeba vždy dodržet, je základní pořadí. Vždy musí být první alespoň jeden sloupec vybraný pomocí select, následovaný vybráním tabulky pomocí from. Následně pak lze libovolně a klidně i na přeskáčku skládat dotaz. Fluent jej na konci sám složí a vrátí výsledek. Fluent lze dále používat i pro univerzální vkládání nebo upravování záznamů jen pomocí pole, které obsahuje názvy sloupců a jejich hodnoty. [7]

```
$query = $this->db->select( ...args: '*');
$query->from( table: 'userTable');
$query->select( ...field: 'userID')
    ->select( ...field: self::PRIMARY);

if (!is_null($id))
{
    $query->where( ...cond: 'userID = %i', $id);
}

$query->where( ...cond: "userName LIKE '%user%'");
$query->orderBy( ...field: 'userID');
if ($isDesc)
{
    $query->desc();
}

if($needAddress)
{
    $query->leftJoin( ...table: 'adressTable')->as( ...field: 'at')->on( ...cond: 'at.%n = %sql', 'addressID',
        $this->db->select( ...args: 'pt.addressID')
        ->from( table: 'personTable')->as( ...field: 'pt')
        ->leftJoin( ...table: 'userTable')->as( ...field: 'ut')->on( ...cond: 'pt.FK_userID = ut.userID')
        ->where( ...cond: 'ut.userID = %n', 'userID'));
}
```

Obr. 3 Dibi fluent
Zdroj: vlastní tvorba

Ve výše uvedeném obrázku **Obr. 3 Dibi fluent** můžeme vidět strukturu fluentího zápisu. Na prvních dvou řádcích je vidět povinná struktura. Jedná se o zmíněný SELECT a FROM, který musí vždy být po sobě a nesmí být přerušen jiným typem příkazu. Následně pak můžeme vidět různé ovlivňující podmínky, například pro doplnění podmínky, celého dalšího kusu SQL či řazení. Jak lze vidět, je možné vkládat i celé nové SQL do podmínky, a to i za příkazem ORDER. Fluent zápis si před zpracováním celého SQL příkazy správně seřadí a následně sestaví klasické SQL.

5.1.5 HTML a CSS

Pro vykreslování samostatných stránek bylo využito, tak jak tomu často u webových stránek bývá, html. Aby však bylo možné využít co nejvíce prvků z PHP a zachovat strukturu MVC, byl využit šablonového systém pro PHP, Latte. Přesto, že PHP lze psát přímo do html stránky, není to nijak komfortní a často to nezpřehledňuje kód samotného html. Latte za nás pak řeší spoustu věcí a díky jeho struktuře lze psát PHP prvky snadněji a zkráceněji. Krom zmíněných prvků nám umožňuje také vytvářet nové funkce, které lze poté přímo volat z html. Dokáže také například vytvářet jednotlivé sekce a díky nim pak dědičnost mezi jednotlivými Latte soubory. Aby pak stránka vypadala stylově hezky, je využito CSS. [8]

Less

Více než CSS je v naší aplikaci využit LESS. Jedná se o zpětně kompatibilní jazykové rozšíření pro CSS. Díky LESS pak můžeme snadno využívat například předdefinované proměnné anebo cykly přímo uvnitř CSS. Dále nám poté také umožňuje vytvářet lepší dědičnost a tím snižovat i redundanci kódu při samotném vývoji. Později pak však musí být do samotného html souboru napojen CSS soubor, který je z LESS překompilován. [9]

Pro finální doladění vzhledu poté bylo využito CSS knihovny bootstrap. Jedná se o jednoduchou sadu kaskádových a javascriptových souborů, které umožňují snadno stylovat obsah webové stránky. [10]

5.1.6 Ajax

AJAX je ve skutečnosti zkratka – Asynchronous JavaScript and XML. Jak název naznačuje, tento přístup nám umožňuje vytvářet takzvané asynchronní webové aplikace. Takové aplikace nám pak dovoluje načítat data na pozadí, což způsobí, že nám zůstane webová stránka stále stejná, aniž by došlo k jejímu obnovení, a mění se pouze viditelná data. AJAX funguje velice jednoduše. Uživatel provede změnu, například vyplní filtr pro hledání v tabulce dat. JavaScript zaznamená uživatelskou změnu, informace se předá pomocí AJAX funkce, ať už pomocí POST nebo GET volání, pošle data na server, ten zpracuje požadavek, vrátí data zpátky JavaScript funkci a ta nová data zobrazí místo původních dat. Díky tomu neproběhne již zmíněná obnova stránky a web je pak pro uživatele o něco přívětivější. [11]

5.1.7 PhpSpreadsheet

PhpSpreadsheet je knihovna, která je napsána v (a přímo pro) PHP. Pomocí této knihovny jsme pak schopni vytvářet soubory MS excel jakéhokoli rozpoložení. Mohou být úplně jednoduché, jednolistové s jednou tabulkou, tak i složité, například několikalistové s grafy a různými výpočty. PhpSpreadsheet nám také umožňuje vytvářet soubory pro LiberOffice Calc a další podobné tabulkové programy. Tuto knihovnu lze nainstalovat například pomocí Composer přímo do PHP aplikace a následně si pomocí *autoload.php* načíst její potřebné prvky. Následně pak pomocí jednoduchých funkcí jsme schopni naformátovat každou buňku ve výstupním excelu, nahrát do něj data a popřípadě využívat přímo i funkcí, které má v sobě Excel již zabudovaný, jako je například SUMA, MAX, MIN a mnoho dalších. Díky tomu lze snadněji pracovat s daty a lze tak vytvořit i jednodušší a přehlednější struktury, jež se mohou dynamicky měnit podle množství dat. [12]

5.2 Mobilní aplikace

Aby nový přístup aplikace lépe splňoval požadavky moderního světa a mladších lidí, jejichž je ve skupině dobrovolníků stále velký počet, bylo domluveno, že vznikne mobilní aplikace. Tato aplikace nebude nijak složitá a bude pouze reflektovat vlastnosti webové aplikace. Přístup do této aplikace bude umožněn pouze dobrovolníkům a dovolí jím snáze, rychleji a lépe ukládat svoji docházku. Díky této aplikaci tak vznikne další usnadnění celého chodu dobrovolnických organizací.

Než se dostaneme k samotné technologii, je dobré si říci něco k Android vývoji. Je potřeba si uvědomit, že mobilní zařízení jako takové, ať už má Android nebo Mac operační systém, má malou obrazovku, ale zároveň je potřeba vše přizpůsobovat tak, aby bylo možné vše uvnitř aplikace používat. Hlavní zásada tedy zní: není podstatné všechno nahromadit na jednu obrazovku, vše lze rozprostřít do většího prostoru, ale se zachováním přehlednosti. Možná to budí zdání toho, že si to protiřečí, ale při vývoji mobilní aplikace je především potřeba myslet na ovladatelnost a přehlednost, aby uživatel vaši aplikaci chtěl používat. Aby se pak vše dalo ovládat co nejlépe, není dobré využívat fixních velikostí a prvků; díky tomu bude aplikace moci být využita jak na malém, tak i na větším zařízení. [13]

Dalšími prvky, na něž je dobré se zaměřit při vývoji, jsou oprávnění. Dnešní telefony mají veliké množství prvků, ke kterým může aplikace přistupovat. Ať už se jedná o pouhý fotoaparát, přístup k wifi nebo dokonce k biometrickému zařízení. Je potřeba uvažovat tom které prvky aplikace opravdu potřebuje a které naopak ne. Příkladem jsou pak aplikace, které pracují například s fotoaparátem, ale chtějí přístup ke kontaktům. Každá aplikace by měla mít přístup pouze k prvkům, které nezbytně potřebuje. Tím nebude uživatel překvapen a bude vznikat také méně bezpečnostních rizik jak pro uživatele, tak i pro vývojáře. [13]

5.2.1 Java

Jedná se o programovací jazyk, který funguje na principu objektového programování. Zjednodušeně řečeno jsou jednotlivé kusy a prvky kódu, například metody, zapouzdřeny v objektech. Díky tomuto způsobu je pak velice snadné přenášet jednotlivé kusy kódu mezi projekty. Dobrým příkladem tohoto přenosu je například přihlašování, respektive registrace uživatelů. Takto stačí daný princip vytvořit pouze jednou na jednom projektu a následně objekty, kusy kódu, které jsou s touto funkcionalitou spojeny, jednoduchým kopírováním přenášet mezi jinými aplikacemi, které jsou také psané v Javě.

Samotný jazyk je dále vynikající v tom, že umožňuje vytvářet programy nebo aplikace pro různé systémy. Ať už se jedná o zmíněnou mobilní aplikaci nebo webovou či počítačovou aplikaci, tento programovací jazyk nemá problém prakticky s žádným systémem. Dalšími výhodami programovacího jazyka Java pak je například jeho distribuovanost, která nám umožňuje využívat různé úrovně síťování, přístupy k vzdáleným souborům nebo vytvářet aplikace na bázi klient-server. Další výhodou (která však může být i nevýhodou) je pak nezávislost základních datových typů. Díky tomuto přístupu jsou kódy java aplikací snadno přenositelné, neboť všechny základní datové typy jako je int, string, float a další, jsou jednoznačně definované a je nutné jejich využití, pokud se pracuje s libovolnou datovou strukturou. Toto však může být i v určitých případech nevýhoda, třeba v případě, že chceme například snadno přepínat v rámci jedné proměnné mezi různými datovými typy. [14]

Java má spoustu výhod, avšak je zde i pár nevýhod. Například u větších systému při vývoji znatelně déle trvá testování, neboť je potřeba takzvaně „sestavit kód“, který následně testujeme například v grafickém prostředí. Další velikou chybou je pak Null pointer exception, které může dělat veliké problémy. Tato chyba se spouští například ve chvíli, kdy pracujeme s proměnou, jež se z nějakého důvodu

nenaplnila daty, a my s ní pracujeme, jako by data měla. I přesto, že tento stav může být chtěný, se pro Javu jedná o fatální chybu a nenechá vás pokračovat. [14]

5.2.2 XML

XML je ve své podstatě velice podobný HTML. Stejně jako právě HTML má také tagy, v nichž lze různě definovat a skládat libovolné prvky, také se jedná o značkovací jazyk. XML skýtá jednu výhodu – jeho tagy (nebo značky) nejsou předdefinované, a tak si uživatel může definovat vlastní a tím si například vytvořit i snadné soubory pro odesílání, kde bude vždy stejná struktura právě díky vlastním tagům, ale i různá těla těchto prvků. Jedná se tedy o velice mocný nástroj pro například ukládání dat, která se pak snadno hledají i ukládají, například jako text, anebo se i díky této podobě snadno sdílí. Další výhodou pak je to, že můžete XML posílat a využívat napříč systémy a výsledek bude vždy shodný, za což můžeme vděčit standardizaci XML. [15]

Právě díky své možnosti tvorby vlastních XML tagů se snadno používá například při vytváření vzhledů, prvků a layoutů při tvorbě Android aplikací. Prakticky každá mobilní aplikace, která využívá nějakou šablonu, má v sobě XML strukturu. Díky těmto párovým a nepárovým tagům lze vytvořit naprosto jednoduchý layout, který obaluje celou hlavní strukturu zobrazené stránky. Jedná se o párový tag, například LinearLayout. Do tohoto párového tagu pak můžeme dát libovolný počet nepárových, například TextView, button, a párových tagů, například jiné layouty, nebo ListView. [16]

Aby se dalo s tagy lépe pracovat, obsahují předdefinované atributy – ty přenášejí informace pro tag tak, aby byl co nejvíce personalizován. Díky atributům můžeme tagu říci, jak má přesně vypadat, jaký text má zobrazit, nebo id, s jehož pomocí k němu můžeme přistupovat. Android má spoustu vlastních atributů definovaných pomocí slova android. Následuje dvojtečka a název atributu, pro příklad *android:id*, je atribut id. Tomuto atributu pak lze nastavit jakoukoli hodnotu.

Můžeme využít klasický text anebo přistoupit k hodnotám Androidu, a to pomocí zavináče a poté přístupu k id hodnotám pomocí id. Výsledek by pak vypadal takto: „*android:id=“@+id/my_button*“. Tímto říkáme, že daný tag, například tlačítko, má atribut id s hodnotou my_button. [16]

5.2.3 Aktivita

Aktivita je třída, která se stará o jeden daný úkon nebo soubor úkonů, spojených právě s jedním prvkem. Jednoduše řečeno se jedná o jednosměrně zaměřenou věc, kterou může uživatel dělat. Aktivita jako taková je často reprezentovaná právě jednou obrazovkou, kterou uživatel aktuálně vidí. Každé aktivitě pak odpovídá jeden layout, který se zobrazuje. Jedná se tedy o formulář přihlášení, registraci nebo jakékoli jiné jediné okno. Existují pak společné metody shodné pro všechny aktivity. Jednou takovou je onCreate, která představuje místo, kde je daná aktivita inicializována. Kromě samotného vytvoření a zobrazení layoutu se také může starat o všechny akce spojené s daným layoutem. Může se starat o odposlouchávání stisků tlačítka, o akce nad jednotlivými poli a spousty dalších úkonů. Je to tedy taková třída, která se stará o životní cyklus jedné obrazovky. [17]

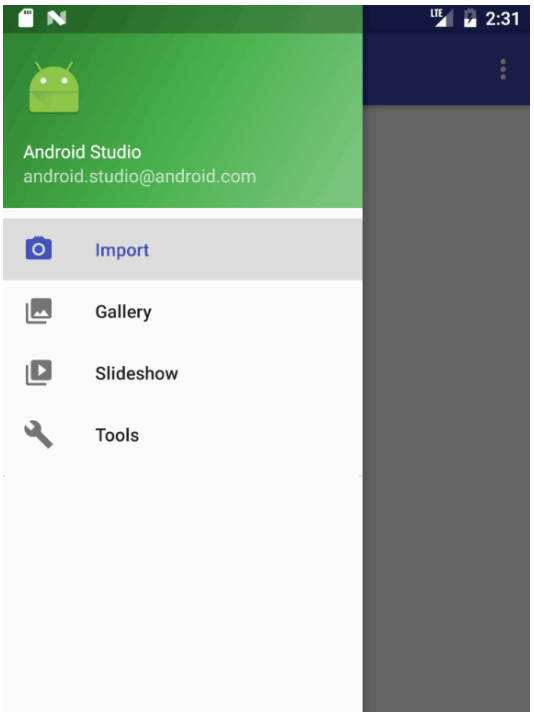
5.2.4 Fragment

Fragment je nutno přímo spojit s aktivitou a toto propojení je nejčastěji použito a aktivitou Drawer, neboli tahací menu. Drawer jedná se tedy o typ aktivity, který se stará o zobrazení tahacího menu na straně telefonu, nejčastěji na levé. Místo tahem se však častěji zobrazuje pomocí stisku tlačítka, reprezentující menu, proto je název trochu zavádějící. Podstatným prvkem však není samotný drawer, ale možnost využití fragmentů místo aktivit. Fragment je ve své podstatě znovupoužitelná část uživatelského rozhraní. Lze ji tedy jednou navrhnout a pak vícekrát volat a využít. Fragment je schopen (stejně jako aktivita) spravovat své vlastní vstupy a výstupy, také své rozvržení nebo i celý životní cyklus. Fragment ale

nemůže existovat sám o sobě. Jedná se vlastně o „parazitický“ prvek, který nemůže žít sám o sobě a potřebuje k životu právě aktivitu, na niž se váže. Právě toto je častým využitím při použitím drawru, menu, kde je menu jako hlavní aktivita a fragmenty vykreslují jednotlivé prvky menu. [18]

Výhodou fragmentů je pak například modularita. Ta vám dovoluje díky svému rozdělení do jednotlivých fragmentů zobrazovat různé prvky a fragmenty v různých zobrazení na různých velikostech obrazovky a tím dosáhnout velké přizpůsobivosti. Díky fragmentům se pak řízení obrazovek i oddělí a proces je tak lépe řiditelný. [18]

Následující obrázek **Obr. 4 Menu pomocí fragmentu s využití draweru,** nám ukazuje, jak vypadá základní zobrazení menu při automatickém vygenerování kódu pro tento typ menu. Menu samotné překrývá obrazovku na levé části, zatímco ztmavený kus obrázku je samotný fragment.



Obr. 4 Menu pomocí fragmentu s využitím draweru
Zdroj: [19]

5.2.5 Volley

Volley je http knihovna, která se primárně stará o ulehčení práce s http požadavky v rámci Android aplikací. Mezi jeho velké výhody pak patří například to, že je rychlejší než běžný přístup, dovoluje vytvářet automatické požadavky, může mít i vícero souběžných připojení a spousty dalších vlastností, které se velice hodí při práci s požadavky na vzdálené servery. Jeho předností jsou pak menší operace, co se týče velikosti stahovaných nebo streamovaných dat, a to proto, že si veškeré odpovědi od serveru ukládá do paměti. Volley má však i velikou podporu pro získávání dat typu string, JSON anebo i obrázek, a proto je vhodný při jednoduchých request response úkonech, kde se přenáší malý počet informací. [20]

```
final TextView textView = (TextView) findViewById(R.id.text);
// ...

// Instantiate the RequestQueue.
RequestQueue queue = Volley.newRequestQueue(this);
String url = "https://www.google.com";

// Request a string response from the provided URL.
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
            new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // Display the first 500 characters of the response string.
        textView.setText("Response is: " + response.substring(0,500));
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
        textView.setText("That didn't work!");
    }
});

// Add the request to the RequestQueue.
queue.add(stringRequest);
```

Obr. 5 Požadavek s Volley knihovnou
Zdroj: [21]

Obrázek **Obr.5 Požadavek s Volley knihovnou**, nám ukazuje, jak vypadá kód, který zpracovává požadavek na danou URL adresu. Znázorňuje, jak se pracovat s přijatou odpovědí a jak samotný požadavek vyslat.

6 Vývoj

Následující podkapitoly pojednávají o samotném vývoji jednotlivých aplikací. Nacházejí se zde například informace o tom, jak byly technologie implementovány anebo jak vypadají struktury aplikací.

6.1 Webová aplikace

V této části bude podrobně popsán vývoj webové aplikace včetně podstatných prvků, jako je databázový návrh nebo struktura tříd v kódu.

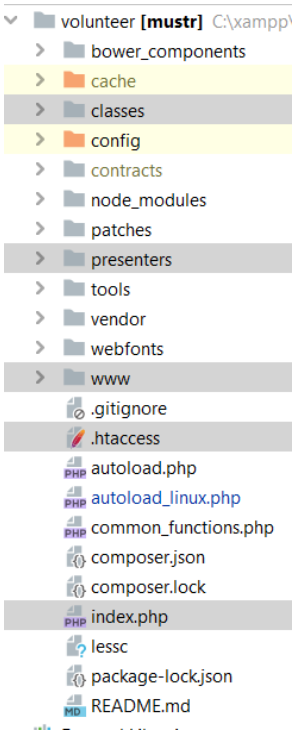
Jak již bylo zmíněno v předchozích kapitolách, aplikace je rozdělena do několika bloků, kterými jsou třídy, prezentery a layouty. V následujících několika blocích budou tyto jednotlivé bloky důkladně popsány i s jejich strukturou.

6.1.1 Podstatné prvky aplikace

Aby aplikace byla schopna pracovat se všemi výše zmíněnými bloky, je potřeba, aby byla schopna rozpoznat, který prezenter má použít. K tomu nám slouží dva základní soubory a několik funkcí. Nejprve je zde *.htaccess*, který nám umožňuje využívat čistou URL adresu tak, aby se v ní nemuseli objevovat přípony php nebo latte ze šablon. Dalším souborem, který musí mít každá webová aplikace, je *index.php*. Pro naši aplikaci je tento soubor stěžejní, neboť dělá několik podstatných operací.

Jako první, co tento index dělá, je, že si načte jeden ze dvou autoload souborů, jeden je pro Windows server a druhý pro Linux. V tomto případě musí být dva, protože každý server pracuje jinak s názvy souborů a s cestami k daným souborům. V těchto souborech se tedy nachází registrační funkce, která pomocí jmenných prostorů v jednotlivých třídách načte tuto třídu v moment, kdy je potřeba, aniž by

bylo zapotřebí je pokaždé vkládat. Následně se pak v indexu vytvoří spojení s databází. Po dokončení tohoto kroku se načte první třída, kontejner. Zde se uchovávají vazby na nezbytné třídy, jako jsou managery a fasády. Kontejner je pak pomocí instanční funkce nahrán do prezenteru potřebné managery a fasády a díky tomu není potřeba jednotlivé soubory připojovat pokaždé, když se z nich volá nějaká funkce. Po dokončení těchto několika bodů se index odkáže pomocí URL na první prezenter, pokud není v URL specifikován, bere se základní, jímž je home s první zobrazovací třídou home.



Obr. 6 Struktura
Zdroj: vlastní tvorba

Výše znázorněný obrázek **Obr. 6 Struktura**, ukazuje základní rozložení webové aplikace se zvýrazněnými podstatnými prvky, které jsou nezbytné pro fungování aplikace.

6.1.2 Třídy

Třídy jsou takto označeny proto, že v této složce se nacházejí veškeré třídy, které nejsou prezentery. Jsou zde tedy zanořeni jednotliví manageři, jejich fasády, utils třídy, generátory, statické třídy a několik dalších.

V tomto bloku tedy budou popsány podstatné třídy, které se týkají celé aplikace, a nejen některých bloků. Jednou z těchto tříd je třída defaultClass. Jedná se o třídu, která v sobě drží veškeré funkce, které pak dědí jednotliví prezenteři. Je zde například funkce init, která je schopna spouštět jednotlivé zobrazovací funkce v daných prezenterech. Dále jsou zde funkce spravující layouty pro vykreslení, nahrávání základních dat do těchto souborů, kontroly oprávnění pro jednotlivé prezentery, nastavování základních hodnot, které se dědí ve všech prezenterech stejně, kontroly přihlášení a další funkce podstatné pro vykreslování prvků na obrazovku.

Velice podstatnou třídou v naší struktuře je pak staticFunctions, což je statická třída, která má pouze statické metody, a to z toho důvodu, aby se daly používat kdekoli napříč celým projektem. Protože to jsou statické funkce, jsou zde metody například na zprávu session hodnot, zpracování layout vykreslení pro AJAX volání, generování hash kódu, nebo například správa oprávnění.

Dále se zde nacházejí třídy jako například paginator, která se stará o stránkování ve všech vykreslených tabulkách, Util třída pro datum, která rozšiřuje již existující základní třídu DateTime, a prvky jako je základní formátování nebo správu měsíců.

6.1.3 Manageři

Jedná se o specifickou třídu, která spravuje veškeré tabulky v databázi. Pro každou tabulku, jež je vytvořena v databázi, vznikne samostatná složka, v níž se

nachází příslušný manager. Veškeří manageři v aplikaci mají stejný název – manager. Liší se pak tedy pouze názvem složky, ve které se nacházejí, a s tím spojeným jmenným prostorem. Díky tomuto zanoření máme poté jednotlivé managery správně odděleny a můžeme s nimi pracovat. Každá manager je pak volaný přes kontejner, kde se mu automaticky s instancí posílá přístup do databáze, a tak není potřeba mít konstruktor anebo vytvářet nějaké vkládání souboru.

Manageři

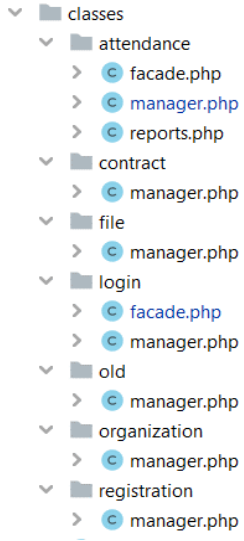
V managerech se pak nachází několik prvků, které jsou specifické pro každý manager. Každá tato třída má konstanty reprezentující primární klíč, název tabulky a jednotlivé sloupečky. V každém manageru pak musí být shodná konstanta table a primary, díky které poté můžeme volat automatizované funkce pro databázi. Manager jako takový je pouze jednoduchá třída, která drží funkce, v nichž se pracuje s databází. Tyto funkce by neměly být nijak složité, respektive by zde neměly být žádné operace, jako jsou třeba cykly. Ty se pak aplikují na získaná data ve fasádách. V těchto funkcích se tedy nacházejí pouze jednotlivé SQL příkazy, které jsou psány pomocí fluent příkazů.

Rozdělení managerů

Pro rozdělení dotazů do patřičných managerů existuje jednoduché pravidlo: tabulka, která je hned za FROM, je tabulka určující, který manager bude zvolen. Proto pokud je například dotaz, který má *FROM tLogin*, který je následován například několika LEFT JOINy, tento dotaz půjde do managera v složce login, neboť je primární tabulka z tohoto managera. Díky tomu pak můžeme snadněji vyhledávat dotazy podle názvů managerů. Dalším způsobem, jak snadněji vyhledat dotazy, je hledání pomocí konstant. Každá tabulka má své hodnoty napsané v konstantách, ty obsahují název tabulky a názvy sloupců a samozřejmě i primární a cizí klíče. Tyto konstanty se poté používají ve všech dotazech tak, aby vzniklo propojení právě na daného managera. Díky tomu pak lze vidět u každé konstanty její reference na všechny dotazy a místa, kde je daná konstanta využita.

Aby bylo možné manager volat z prezenterů nebo z fasády, je potřeba vytvořit jeho instanci; ta je pak, jak již bylo zmíněno, vytvořena ve třidě kontejner, která udržuje napojení na managery, fasády a popřípadě i reporty tak, aby byly snadno přístupné z prezenterů.

Obrázek **Obr.7 Manageři a fasády**, nám zobrazuje veškeré fasády a managery, které jsou použity v aplikaci i s jejich rozdělení dle názvů tabulek pro které jsou definováni.



Obr. 7 Manageři a fasády
Zdroj: vlastní tvorba

6.1.4 Fasády

Jak již bylo zmíněno, každá fasáda spadá do složky s případným managerem, díky tomu pak vznikne jednoznačné rozdělení a sníží se duplicita kódu. Jak lze vidět na obrázku výše, ne každý manager nutně potřebuje fasádu. Důvodem je to, že spousta věcí se dá řešit pouze pomocí SQL dotazu, a není proto potřeba přímo vytvářet fasádu, která by daná data z dotazu zpracovávala. Fasáda předně slouží hlavně k tomu, aby buď předzpracovávala data před uložením do databáze,

zpracovávala data pro výstup anebo doplňovala potřebné informace na základě dat získaných z databáze.

Když se zaměříme na jednotlivé fasády viditelné na obrázku výše Manageři a Fasády, tak jsou zde dvě podstatné: pro uživatele login a pro docházku attendance. Fasáda pro uživatele se zejména stará o všechny informace týkající se uživatelů, ať už se jedná o vytváření nových uživatelů nebo jejich přihlášení. Dále pak například zajišťuje výpis na základě oprávnění a také třeba změnu přístupu uživatelů. Jak je z popisu fasády pro uživatele zřejmé, fasáda pro docházku se zaměřuje na zprávu dat pro docházku. Kromě samotného výpisu, ukládání nebo upravování dat pro jednotlivé docházky zajišťuje také tato fasáda zpracování dat pro reporty.

Důvod pro fasády

Fasády jsou velice podstatnými prvky v celé aplikaci, neboť nám rozdělují kód na drobnější a snáze použitelné funkce tak, aby prezentery mohly být co nejkratší. Díky tomu nám vzniká velká znovu-použitelnost a nízká redundance kódu. Stejně jako tomu je u managerů, tak i volání fasád je z kontejneru.

Tvorba fasád

Fasáda pak oproti managerovi má vlastní konstruktor, který je pro ni nezbytný. Díky němu se specifikuje, s jakými managery, popřípadě jakými dalšími třídami může pracovat. Poté nám vznikne jasně specifikované rozmezí dosahu dané fasády. Tento konstruktor je pak vytvořen, respektive zavolán, z kontejneru při volání potřebné funkce z fasády. Pro konstruktor existují pouze dvě pravidla, a to, že vždy musí obsahovat managera dané složky, kde se nachází, a nesmí volat jiné fasády.

Na níže uvedeném obrázku **Obr. 8 Vytvoření instance pro managera a fasádu** můžete vidět strukturu funkcí, které se starají o inicializaci managerů a fasád. Přesto, že manager nemá konstruktor a fasáda ano, jsou oba volány pomocí jedné funkce *getInstance*. A to proto, že každý manažer dědí ze třídy Dibi, která má vlastní konstruktor, jenž požaduje instanci připojení do databáze, proto je možno

použít tuto strukturu. Samotné parametry ve funkci *getInstance* jsou pak jednotlivé parametry v konstruktoru jednotlivé třídy.

```
public function getAttendanceManager() : \Classes\Attendance\Manager
{
    return $this->getInstance($this->getDibi());
}

public function getAttendanceFacade() : \Classes\Attendance\Facade
{
    return $this->getInstance($this->getAttendanceManager(), $this->getPaginator());
}
```

Obr. 8 Vytvoření instance pro managera a fasádu
Zdroj: vlastní tvorba

6.1.5 Reporty

Jedná se o specifickou třídu, která je stejně jako manager nebo fasáda vázaná na svou složku, která specifikuje, pro jaký daný typ dat se jedná. V našem případě máme pouze jednu hlavní reportovací třídu a ta se týká docházky. To ale neznamená, že není možné vytvořit report třídy i pro jiné složky, ba naopak, avšak v našem případě to není potřebné.

Report třídy, podobně jako fasády, mají také vlastní konstruktor, do kterého se posílají manageři a další třídy, s nimiž může pracovat. Oproti fasádě, která může mít v konstruktoru pouze managery, může report mít v sobě i fasádu, ale musí jít pouze o fasádu ve stejné složce, kde se nachází report. Pro report je pak nutností, aby v konstruktoru byla zařazena třída ExcelGenerator.

Rozšíření PhpSpreadsheet

Toto je třída, která rozšiřuje již výše zmíněný PhpSpreadsheet, díky kterému lze vytvářet a jednoduše spravovat excel soubory. Krom jednoduchého rozšíření však tato třída ještě doplňuje vlastní funkce. Rozšíření se pak uplatňuje například v případě ukládání souborů, kdy na základě jednoduchých parametrů buď vezme

vytvořený excelový soubor a ten uloží na server, nebo automaticky tento soubor uloží uživateli do počítače. Dále zde také probíhá jednoduché pojmenování tohoto souboru. Krom tohoto rozšíření pak zde můžeme najít například funkce definující základní hlavičku v excelu, respektive řádky jako jsou třeba nadpisy, nebo výpisy filtrů. Pak jsou zde také některé funkce, které stylují určité typy buněk, jako jsou například časy. Samozřejmostí jsou pak metody, které obsahují pole s předdefinovanými styly, aby každý excel mohl být stylován stejně tak, jak si zákazník, v našem případě Oblastní charita, představuje. Tyto funkce se jednoduše zavolají a aplikují se na potřebné datové prvky, například hlavičky anebo těla tabulek, které se vypisují v Excelu.

Zpracování dat

Samotné report soubory by se pak už neměly starat o zpracovávání dat, toto zpracování by mělo proběhnout na straně fasády, odkud se potřebná data berou. Pokud jsou potřebné některé prvky, jako je jednoduché sčítání v průběhu výpisu, tak podobně jako u manageru to může být zde řešeno, avšak v našem případě jsou veškerá data spravována ve fasádě a report třídy se stará pouze a jen o samotný výpis dat.

Generování

Výpis pak probíhá pomocí PhpSpreadsheet, kde se vytvoří první list v Excelu, se kterým se pracuje. Poté s odkazem na tento list nám PhpSpreadsheet povolí přistupovat k jednotlivým buňkám tohoto sešitu, kam přímo vypisujeme data. Samozřejmostí poté je, že kromě vypsání dat lze, jak již bylo zmíněno, na jednotlivé buňky aplikovat i styl. Ten se aplikuje buď pomocí pole, nebo funkcí přímo přes PhpSpredsheet, které dovolují přes jednoduché konstanty nastavovat potřebné styly. Příkladem pak může být třeba zarovnání, které se snáze vytvoří právě pomocí těchto metod.

Příloha číslo 1 odkazuje na excelový soubor, který obsahuje a znázorňuje, jak vypadá report pro testovací data, vygenerovaný z aplikace. Jedná se o report pro docházku, nikoli pro odpracované hodiny. Data jsou převážně smyšlená a fiktivní

6.1.6 Prezentery

Jedná se o specifickou třídu, která se stará o zobrazovaní dat na stránce, respektive o jejich přípravu dat a následné předání patřičnému layoutu, jenž je zobrazí uživateli. Prezentery se volají na základě parametrů v URL adrese, jak již bylo zmíněno dříve. URL adresa se zpracuje přímo v index souboru a následně se přepošle do funkce moduleLoader, kde se pomocí několika příkazů, pracujících s PHP třídou ReflectionMethod, adresa zpracuje a zavolá se patřičná funkce v daném prezenteru. Pro toto zpracování se používají veškeré prvky, které se nachází za koncem hlavní části URL adresy, v našem případě tedy za „koncovkou" cz. Avšak může nastat i okamžik, kdy za „koncovkou" není žádný text, v tom případě se bere základní pravidlo.

Logika volání prezenterů

Pravidla pro zpracování URL adresy jsou velice jednoduché. Platí, že pokud je samotná URL adresa, tedy adresa končící na cz, vezmeme základní třídu i základní metodu, jejichž oba názvy jsou reprezentovány slovem home. Veškeré metody, které se mají zobrazovat pomocí prezenterů, nebo pomocí ajax metod, mají na začátku svého názvu slovo render, které reprezentuje, že se jedná o třídu, jež bude něco zobrazovat nebo vracet stránce. Pro příklad základního pravidla pak máme tedy třídu s názvem Home a metodu s názvem *renderHome*.

Pravidla pro správné volání

Další pravidla se pak aplikují na text za již zmíněnou „koncovkou". Tento text je nejprve rozdělen na všechny části, a to pomocí oddělovače „/". Díky tomu pak získáme veškeré potřebné parametry pro následné zpracování adresy pro zavolání potřebné metody. Platí tedy: po rozkouskování tohoto textu získáme pouze jedno slovo (přesněji – za „koncovkou" je pouze jedno slovo, například attendance). Toto nám říká, že se zavolá třída attendance ze složky prezenterů. Protože však zde není druhý parametr, který by nám řekl, jakou zobrazovací (render) funkci máme volat,

aplikuje se upřesňující pravidlo. To nám říká, že pokud zde není parametr upřesňující metodu, zavolá se základní zobrazovací metoda pro toto pravidlo a tou je metoda s názvem Default. Pro náš případ, kde url adresa je *cz/attendance*, bude tedy zavolána třída Attendance a v ní metoda renderDefault.

Následně nám zůstává poslední pravidlo: jedná se o pravidlo, kde jsou po rozkouskování parametry dva. První parametr nám říká, o jakou třídu se jedná, a druhý parametr nám sděluje druh zobrazovací funkce. Pro lepší znázornění si například vezměme URL adresu *cz/user/login/*. Z této adresy, respektive z jejího konce, je patrné, že se přesuneme do prezenteru, třídy User a zavoláme zobrazovací metodu renderLogin.

Ať se jedná o jakékoli z předchozích pravidel, jsou zde další prvky, kterými se prezenteři řídí. Jedním z nich je, že na konci URL adresy nemusí být „/“. Dalším jsou pak parametry, které posíláme například z formuláře. Prezenteři jako takoví nevědí, zdali dostali data z POST nebo z GET metody, neboť využívají sjednocující funkci getParameter, která sjednotí obě pole z POST I GET a prezenteři s tím pak pracují jako s jednotným polem. Zmiňuji to kvůli URL adrese, kde je metoda GET zobrazena za znakem „?“; ten může být přítomen, ale na vyhledávání zobrazovacích funkcí nebo tříd nebude a nemá žádný vliv.

Struktura prezenterů

Všechny metody v prezenterech, které něco zobrazují uživateli na webové stránce, krom několika výjimek, jako je například zmíněná metoda *renderLogin* ve třídě User, která po zpracování dat uživatele přesměruje tam, kam je potřeba na základě jeho dat, mají velice podobnou strukturu. Protože, jak bude níže zmíněno, používáme pro zobrazování latte layouty, jako první je vytvoření latte proměnné.

Při jejím vytváření, krom namapování Latte třídy na proměnou jsou provedeny další prvky. Dobrým příkladem je nahrání základních uživatelských dat ze SESSION, kam ukládáme data po přihlášení, nebo data v průběhu práce na stránce. Dalším příkladem pak může být například kontrola zpráv, které se zobrazují uživateli. Po

tomto kroku nastane kontrola, zdali je uživatel přihlášen; tento krok je až zde právě kvůli již zmíněným zprávám, aby se mohla zobrazit uživateli např. zpráva, že nebyl přihlášen. V tomto kroku také nastává kontrola oprávnění, a to, aby uživatel nemohl projít tam, kam nemá přístup. Dalším krokem, který je v zobrazovacích metodách použit, je nastavení menu, respektive pozice, kde se uživatel nachází, tak, aby bylo možné tuto informaci sdělit uživateli.

Vykreslení

Po dokončení všech výše zmíněných krocích nastává vykreslení stránky pomocí metody *renderLayout*. Této metodě se předají data, která patří pro danou zobrazovací funkci, dále pak jméno potřebného layoutu. Samozřejmostí je proměnná, která drží latte třídu, a pak několik parametrů, jež určují, kde se layout nachází anebo jak naložit se zprávou pro uživatele.

```php
public function renderDefault()
{
    $latte = $this->setLatte();
    $this->isLoggedIn( checkIfAdmin: false);
    $this->setActiveMenu( menuID: self::MENU_HOME);

    $this->addData( key: 'programs',  data: \Classes\Attendance\Manager::PROGRAM_DICTIONARY);

    $this->renderLayout($latte,  file: 'index', $this->getData(),  specificLatte: self::LATTE_USER,  refreshMessage: true);
}
```

Obr. 9 Základní struktura render metody
Zdroj: vlastní tvorba

Na obrázku **Obr. 9 Základní struktura render metody**, lze vidět jednotlivé prvky potřebné pro správné fungování a následné vykreslení u render metod. Postupně shora dolu se nejprve definuje funkce, nastaví se základní prvky pro latte šablonu, kontrola přihlášení, nastavení, které má být aktivní, následné nepovinné přidání dat, které se vypisují a úplně na konci je volání vykreslovací metody, kde se definuje, jaký latte soubor se má použít

Specifičtí prezenteři

V prezenterech se pak nacházejí dvě specifické (již zmíněné) třídy, kterými jsou Ajax a Container. Ajax třída zpracovává veškeré požadavky, které jdou ze

stránky pomocí JavaScriptu. Zde použitá URL adresa má vždy stejné znění */ajax/názevMetody*. Protože je Ajax třída specifickým poddruhem prezenteru, pak jsou zde všechny veřejné metody zobrazovací, a tudíž mají ve svém názvu na začátku slovo render. Metody v této třídě se mohou chovat jako datové nebo zobrazovací.

Datové funkce zpracují požadavek a následně vrátí JSON, který si sám JavaScript zpracuje a následně tyto data vypíše. Příkladem je například metoda *renderGeneratePassword*, která na podnět od uživatele vygeneruje náhodný string, který pak následně JavaScript pouze zobrazí v patřičném inputu.

Zobrazovací metody jsou pak takové, které zpracují požadavek a namísto JSON vrací pouhý string, který obsahuje HTML, respektive latte strukturu, jíž pak lze následně zobrazit. Toto je dobře využito například při vykreslování docházky, kdy při stránkování není, díky Ajax volání, nutné nahrát celou stránku, pouze se překreslí tabulka na základě nových informací. JavaScript tedy zavolá Ajax zobrazovací metodu s tím, že se změnila stránka, funkce zpracuje požadavek, zavolá si patřičný layout, kam vypíše data. Ten je následně převeden do podoby stringu a následně navrácen zpět onomu JavaScriptovému požadavku, který tento string vykreslí do obrazovky namísto původní tabulky.

Container

Dalším specifickým prezenterem je pak třída Container, která zde byla již mnohokrát popsána. Tato třída je specifická tím, že má v sobě pouze jednu funkční metodu, *getInstance*, a poté pouze instanční třídy, které připravují managery, fasády a jim podobné třídy tak, aby bylo možné jejich použití v prezenterech. V jejich případě se jedná pouze o metody typu GET, kde se specifikují prvky, které daná třída potřebuje, a ty se aplikují. Pro managery je to přidání prvku DibiConnection, díky němuž mohou přistupovat k databázi. Pro ostatní metody se pak jedná o konstruktory, respektive o naplnění proměnných, které si konstruktor žádá, a to nejčastěji voláním ostatních GET funkcí, jakou jsou právě například manageři. Díky tomu poté není nutné, aby metody byly neustále donekonečna instancovány. Kontejner toto „připojení“ souboru provede jednou a pak pouze dovoluje využívání

jejich funkcí prostým zavoláním potřebné GET funkce. Díky tomu pak lze snadno využít jmenných prostorů, namespace, namísto PHP funkcí, jako je například include, nebo require.

Dříve zmíněná metoda *getInstance* je pak velice užitečná metoda, která plní dříve zmíněnou funkcionalitu. V těle GET metod je pak tedy volána funkce *getInstane*, které jsou předány parametry, jejichž využití vyžadují volané třídy. V těle funkce *getInstance* pak dochází k dohledání potřebné třídy, vytvoření spojení a díky tomu umožněno využití jejich funkcí. Krom těchto prvků se *getInstance* také stará o ostatní prezentery. Je to tedy metoda, která z dat, které dostane z URL adresy, zavolá potřebnou zobrazovací metodu v té správné třídě.

6.1.7 Layouty

Aby aplikace mohla co nejlépe využívat modelu MVC a zároveň bylo možné přímo v layoutu psát PHP, nebo pseudo PHP příkazy, je využita latte knihovna namísto obyčejného HTML. Ta nám díky svým specifickým tagům a atributům dovoluje psát do HTML „PHP“ příkazy, díky nimž lze lépe pracovat s daty, které se zobrazují na obrazovce. Díky dalším vlastnostem latte, jako je například dědění mezi soubory, nebo jejich propojování, lze rozdělit layouty do několika základních bloků. V případě této aplikace se pak jedná o rodičovský, základní, podpůrný a tabulkový layout.

Rodičovský layout

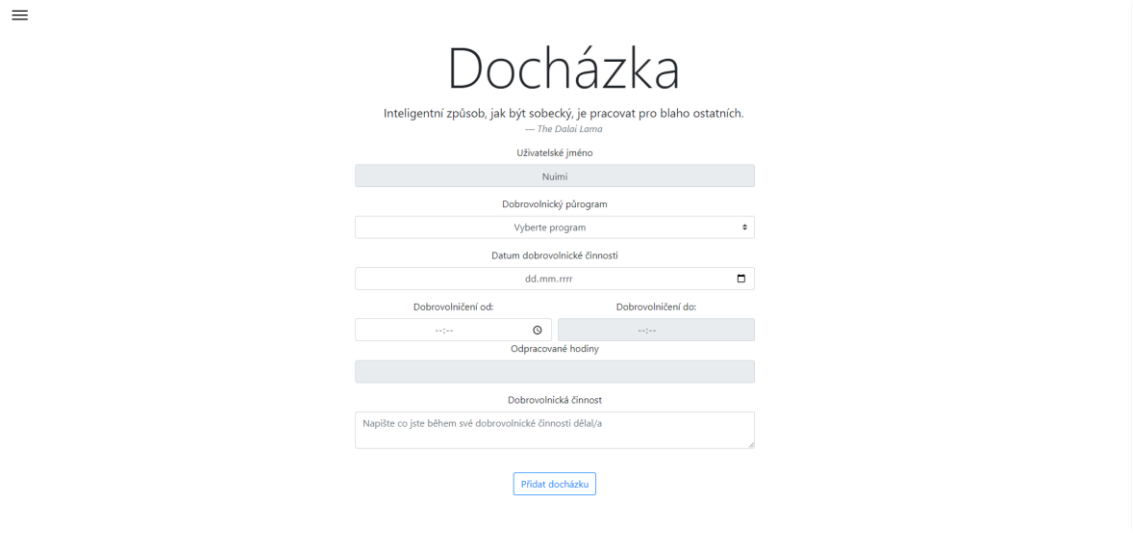
Rodičovský layout je specifický v tom, že obsahuje pouze několik základních prvků, které se dědí, respektive sdílí napříč ostatními layouty. Zde pak můžeme najít základní definici HTML, včetně nadefinování hlavičky i s CSS a JS knihovnami, dále poté základní definici těla stránky, kde se nachází několik tagů, které nastavují stránku tak, aby měla vždy stejné rozložení. Aby pak bylo možné snadno dědit, tento rodičovský layout v sobě obsahuje (na přesně určeném místě) latte tag include, který nám umožňuje, krom vkládání celých souborů, vkládat také bloky. Tento tak

pak vypadá takto: *{include body}*, kde body je název bloku, jenž bude vkládán. Umístění tohoto tagu je v místě ve stránce, které je navržené tak, aby bylo možné nahrazovat tento prostor, aniž by ovlivnil další prvky stránky, jako je například menu nebo hlavička. Bloky jsou poté jednotlivé kusy latte, respektive HTML kódu, které se do těchto míst vkládají. Název takového souboru pak zpravidla zní @layout.

Základní layout

Základní layout je pak tedy již zmíněný blok, který se mění na základě URL adresy, tedy podle zobrazovací metody z prezentera. Názvy těchto souborů se pak liší buď podle toho, co zobrazují, nebo podle prezentera, z něhož jsou volány. Aby však mohl takovýto soubor bát správně vložen do stránky, potřebuje mít specifikovaný blok a také potřebuje určit, do jakého souboru se bude dělit. V našem případě tedy tyto soubory mají na začátku souboru specifikováno, že rozšiřuje rodičovský soubor, tedy *{extends @layout.latte}*. Dále poté nastává párový tag *{block body}*, v němž lze napsat libovolným způsobem HTML kód, který bude následně zobrazen na stránce.

Oba předchozí typy layoutu jsou plnohodnotnými layouty a mohou tedy jeden bez druhého existovat a být samostatně zobrazeny na stránce. Dalším typem, který potřebuje být vypsán v nějakém z předchozích layoutů, je podpůrný typ layoutu. Jedná se pak o latte soubor, který v sobě drží informace, které mohou být sdíleny napříč celou aplikací, popřípadě i vícero rodičovskými layouty. Může tedy jít, jako v našem případě, o layout, který se stará o zobrazovaní zpráv. V tomto souboru je celá logika, jak zobrazovat zprávu, kde ji zobrazovat a co ve zprávě bude vypsáno. Tento soubor tedy je vložen někde nahoře v rodičovském layoutu a jsou do něj zasílána data, na jejichž základě zprávu zobrazí uživateli. Tento postup pak zaručí, že zpráva vypadá stejně napříč celou aplikací. Krom tohoto layoutu na zprávy zde máme layout s modal prvky nebo menu. Modal je pak specificky definován pro určité události, ale všechny tyto modaly mohou být na jednom místě a opět sdíleny napříč celou aplikací. Samotné menu může být uchováno rovnou v rodičovském layoutu, ale protože v našem případě drží jistou část logiky, je vytažen bokem. To způsobuje, že také může být využit v jiném rodičovském layoutu.

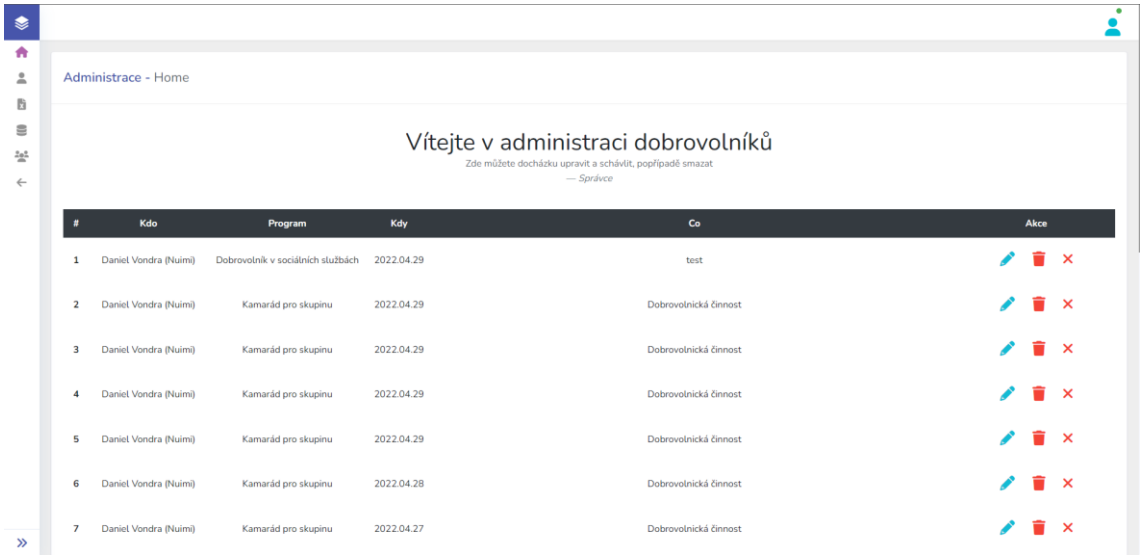


Obr. 10 Základní layout – část pro dobrovolníky
Zdroj: vlastní tvorba

Výše uvedený obrázek **Obr.10 Základní layout – část pro dobrovolníky**, nám znázorňuje, jak vypadá samotná aplikace pro přihlášeného uživatele s oprávněním dobrovolník. Jedná se o základní layout, který obsahuj formulář pro přidání odpracované docházky.

Tabulkový layout

Posledním typem layoutu je pak tabulkový layout, který se rovněž musí vypsat v nějakém souboru. Na rozdíl však od podpůrného typu se tabulkový vypisuje přímo do základního layoutu, neboť je specifický pro daný část stránky. Tabulkové layouty, jak název naznačuje, obsahují pouze tabulky, popřípadě něco, co tabulku připomíná, nebo zobrazuje jakýmkoli způsobem uživatelem žádaná data. Díky tomuto rozdělení poté lze jednotlivé layouty využívat například při Ajax volání. Ajax si zavolá nějakou funkcionalitu, které má zobrazit nová data, například změna filtru. Tato metoda si pouze vezme tabulkový layout, který je již vypsán, naplní do něj data a ten pak vrátí. Díky tomu eliminujeme velkou duplicitu dat a získáme přehlednější a logičtější strukturu kódu.



Obr. 11 Tabulkový layout – část pro adminy
Zdroj: vlastní tvorba

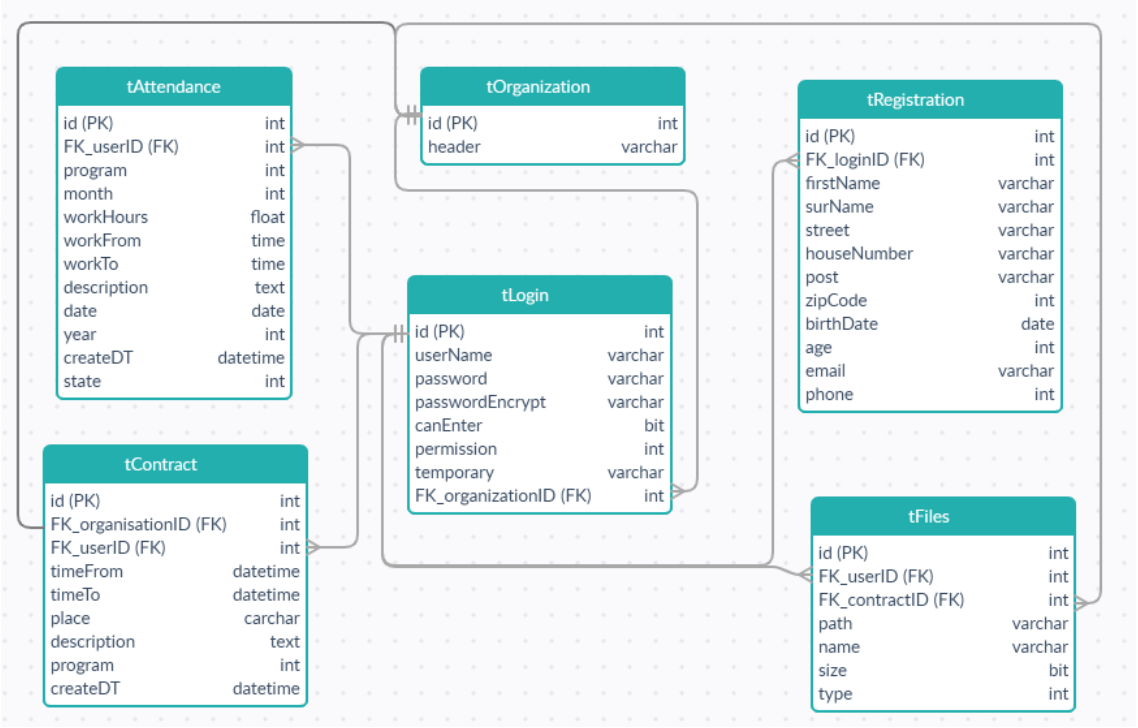
Na uvedeném obrázku **Obr. 11 Tabulkový layout – část pro adminy**, lze vidět tabulku zasazenou v layoutu pro adminy. Tento layout je přístupný pouze pro uživatele s oprávněním vyšší, než je dobrovolník, koordinátor, organizátor, správce. Jedná se o úvodní obrazovku v administraci, kde je vidět výpis provedených, neschválených docházek vykonaných dobrovolníky.

6.1.8 Databáze

Jak již bylo zmíněno v předešlých kapitolách, databáze je uložena v prostředí PhpMyAdmin, ke kterému je přistupováno pomocí SQL. SQL jako takové zde však není úplně použito, neboť je využita knihovna Dibi. Ta nám pak umožňuje využívat fluent zápis, díky kterému můžeme zapisovat složitější dotazy. V celém kódu pak není viditelný jediný SQL zápis. Veškeré prvky jsou dopisovány pomocí konstant, které drží informace o jednotlivých tabulkách samotní manažeři. Díky tomu máme přehled o tom, jak jsou jaké dotazy propojeny, ačkoliv se ztrácí přehlednost samotného SQL.

Samotné tabulky jsou pak spolu propojeny pomocí cizích klíčů. Díky tomu, že primární klíče jsou vždy automaticky generovaná čísla, můžeme snadno propojit jednotlivé tabulky a následně nad nimi provádět různé operace. Příkladem jsou pak automatizované dotazy na úpravu, mazání nebo vkládání, které se na základě managera a pole s prvky automaticky mapují pro daný manager. Pro úpravu nebo mazaní je poté využit primární klíč.

Tyto klíče jsou pak ve všech managerech pojmenovány PRIMARY, a to z toho důvodu, abychom mohli mít různé názvy primárních klíčů v databázi, avšak v kódu byl název vždy shodný, a to proto, abychom mohli snadno vytvářet univerzální dotazy. Rozdíl v přístupu k této konstantě je pak v použitém manageru, který určuje, z jaké tabulky bereme daný primární klíč.



Obr. 12 Struktura databáze
Zdroj: vlastní tvorba

Ve výše uvedeném obrázku, **Obr. 11 Struktura databáze**, lze vidět kompletní strukturu databáze pro novou aplikaci. Nachází se zde jednotlivé tabulky a jejich vazby. Většina vazeb je tvořena pomocí 1:N, je to jeden z nejpoužívanějších typů

vazby a v našem případě i nejvhodnější. Všechny tabulky jsou nějakým způsobem propojeny mezi sebou za pomoci tabulky tLogin, protože pro každou akci je zapotřebí nějaký uživatelský podnět, který se do tabulek propisuje.

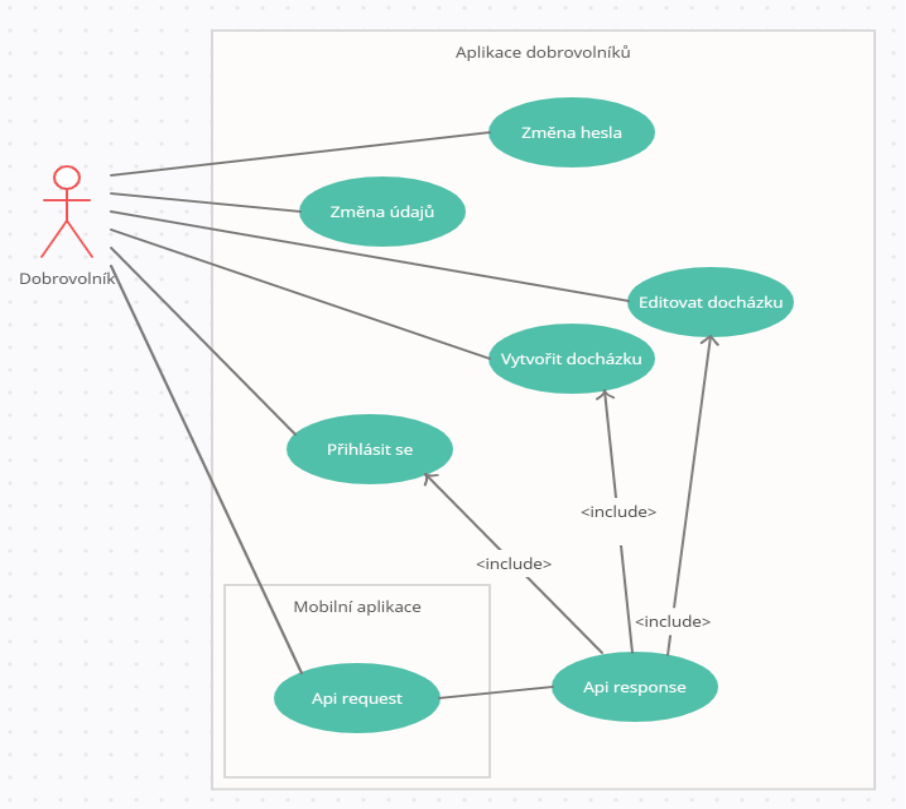
6.1.9 Uživatelské účty (role)

Aplikace je rozlišena do několika jednoduchých úrovní, které jednoznačně rozdělují uživatele do kategorií a následně jim umožňují se ve svém prostoru pohybovat. Díky tomuto rozdělení lze jednoznačně oddělit například koordinátora od dobrovolníka a přesně jim povolit přístup tam, kam mohou, a zakázat přístup tam, kam nemohou. Jak již bylo zmíněno, jedním z oprávnění je koordinátor, nebo admin, dále pak dobrovolník neboli klasický uživatel s prakticky nejnižší pohyblivostí po aplikace a poté správce, který má přístup k celé aplikaci. Dodatečnou úrovní uživatele je pak organizátor, jehož pozice zastupuje správce, dokud nebude patřičná osoba zvolena. Jeho oprávnění oproti koordinátorovi se poté odlišuje pouze v rozdělování dobrovolníků do organizací, charit.

Dobrovolník

Představuje nejzákladnější úroveň. Jeho role je nejlépe srovnatelná s obyčejným uživatelem. Důvodem tohoto srovnání je to, že dobrovolník jako takový nemá práva na to, aby jakýmkoli zásahem ovlivňoval aplikaci, na rozdíl od ostatních rolí. Dobrovolník ve své podstatě má povoleno do aplikace zadávat svoji docházku, popřípadě pak tuto vyplněnou aktivitu změnit, aby nedošlo k tomu, že v aplikaci budou nepřesné nebo neúplné informace. Toto jsou vlastně jediné dvě věci, které v aplikaci může dobrovolník dělat a kterými se odlišuje od ostatních rolí. Stejně jako ostatní role, i dobrovolník může také měnit informace o sobě nebo si změnit heslo. Aby však dobrovolník nebyl úplně znevýhodněn, tak na rozdíl od ostatních uživatelů, kromě správce, smí dobrovolník přistupovat do mobilní aplikace, jejíž fungování a věci, které v ní může dobrovolník dělat, budou popsány v následujících kapitolách. Na obrázku, **Obr. 13 Use case – dobrovolník**, vidíme veškeré akce,

které dobrovolník může v aplikaci dělat. Je zde i zaznamenána mobilní aplikace, která využívá API requestů na webovou aplikaci.



Obr. 13 Use case – dobrovolník
Zdroj: vlastní tvorba

Koordinátor (admin)

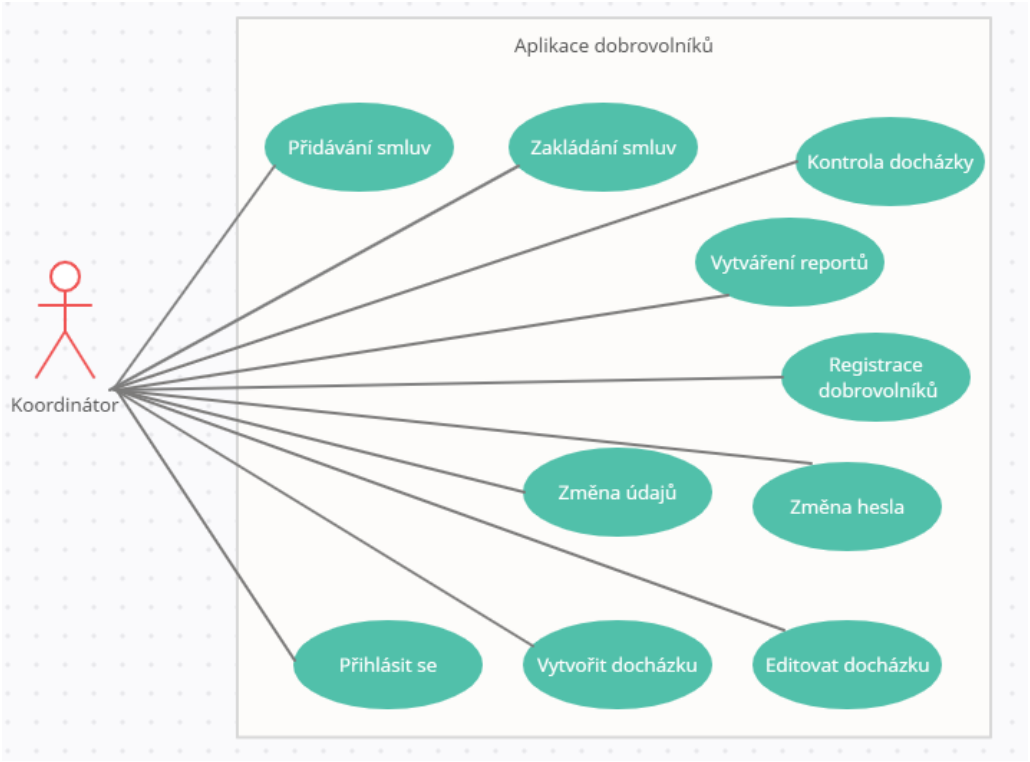
Koordinátor (nebo také admin) je jedním z nejpodstatnějších uživatelů v aplikaci. Krom možnosti zadávat docházku za dobrovolníky (což je někdy nutnost, neboť existují dobrovolníci, kteří buď ze zdravotních, znalostních, nebo kvůli nedostupnosti technologie nejsou schopni docházku sami vyplnit) může koordinátor přistupovat k poměrně velké části administrace. Hlavním a velice podstatným prvkem je pak správa dobrovolníků. Zde poté může koordinátor dobrovolníkovi založit jeho účet, neboť se dobrovolník nemůže sám registrovat, a připojit k němu i smlouvu, aby tak všechny informace byly na jednom místě. Dále má pak přístup do prostoru docházky, kde přehledně vidí veškerou docházku dobrovolníků. Zde ji může poté spravovat, to znamená potvrdit jejich správnost,

popřípadě něco upravit. Z těchto vytvořených docházek si pak může sám vytvářet reporty, které si může libovolně filtrovat. Zde si poté může vybrat z podrobného reportu. Ten obsahuje veškeré informace k docházce. Pak zbývá sumární report, který má v sobě pouze informace ohledně odpracovaného času společně se základními informacemi.

Všechny tyto akce se poté provádí v rámci dané organizace, k níž koordinátor náleží. Díky tomu se tak oddělí prostory jednotlivých charit a koordinátoři se nemusí bát, že by viděli data ostatních charit, nebo že by je mohl upravovat či přidávat. Při vytváření dobrovolníků pak dobrovolníci dědí organizaci od koordinátora. To pak určí, že i dobrovolník přidává docházku pouze pod svoji organizaci, tedy pod svojí charitou.

Díky všem zmíněným operacím, které ovlivňují aplikaci, poté může být koordinátor považován za admina, přestože však jsou jeho zásahy pouze uvnitř své dané organizační oblasti. V následujícím obrázku vidíme veškeré akce, které může tento uživatel provádět. Díky tomu, že koordinátor může doplňovat docházku za dobrovolníka a zároveň je sám uživatelem, mají některé funkce shodné s dobrovolníkem. Rozdělení do vícero obrázků je zde hlavně pro přehlednost.

Níže uvedený obrázek, **Obr. 14 Use case – koordinátor**, znázorňuje akce, které jsou koordinátorovy v aplikaci povoleny. Jak již bylo zmíněno, některé akce má shodné s dobrovolníkem, aby mohl vypomáhat s vyplněním docházky, kdyby bylo potřeba.



Obr. 14 Use case – koordinátor
Zdroj: vlastní tvorba

Správce

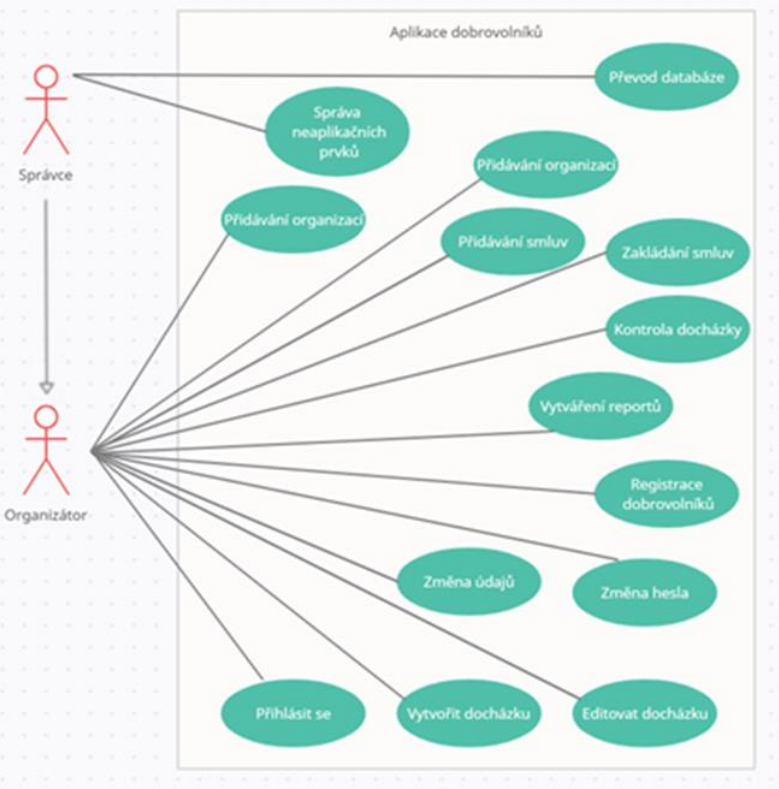
Poslední a nejzásadnější rolí v aplikaci je pak správce. Jeho role, jakožto osoby s nejvyšším oprávněním, je poté udržovat systém v chodu. Krom veškerých předešlých oprávnění, jako je například přidávání docházky, úprava údajů, také pak generování reportů, nebo následná kontrola docházky, má na starost například databázi. Jelikož existují, jak již bylo zmíněno, dvě aplikace, tak správce musí udržovat správnou konzistenci dat, přesněji řečeno převádět nově vzniklá data z původní aplikace do nové. Kvůli jistým převodním vlastnostem, které budou zmíněny v následující kapitole, je potřeba tuto operaci dělat ručně, aby se zamezilo možnosti, že vzniknou chybná nebo duplicitní data.

Správce pak jako jediná osoba, která není v žádné organizaci, vidí do všech organizací. Díky tomu tak může vytvářet koordinátory do jednotlivých organizací, charit. Samozřejmostí je poté vytváření jednotlivých, nově příchozích charit. Díky tomu, že má pak i přístup do databáze, existuje možnost přímo upravovat data,

například pokud je potřeba upravit něco, co nelze dělat z aplikace. Příkladem je třeba samotná editace chybně převedených dat anebo převod dobrovolníků mezi jednotlivými organizacemi.

Organizátor (dodatečný uživatel)

Aby se snáze dodržovala veškerá GDPR pravidla a nedošlo k zneužití uživatelských údajů, byla navržena role organizátor. Toto oprávnění, organizátor, by mělo pak stejná práva jako správce. Mohl by tedy vytvářet koordinátory a s určitými úpravami by pak viděl veškeré záznamy. Neměl by však přístup do editace databáze nebo by nemohl dělat samotný databázový převod. Protože se však zatím nedohodlo a v nejbližší době, kvůli covidu a válce na Ukrajině, kolik a které charity budou novou aplikaci využívat, tak je pouze teoretickým oprávněním a bude použito v platnost v okamžiku, kdy se dohodne, co daná osoba bude přesně moci dělat. Tedy zdali bude pouze vytvářet organizace a koordinátory, nebo bude jakýmsi mezistupněm mezi správcem a koordinátorem.



Obr. 15 Use case – správce
Zdroj: vlastní tvorba¨

Obr. 15 Use case – správce, nám ukazuje, tak jako je tomu u Obr. 14 a Obr. 13, veškeré akce, které může správce nebo i organizátor provádět. Protože se jedná o úzce spjaté role, jsou znázorněny na jednom obrázku.

6.1.10 Převod databáze

Při vytváření nové aplikace kromě vylepšení samotné struktury bylo potřeba vyřešit i novou databázi. Původní aplikace neměla žádné databázové problémy a byla navržena i relativně správně, avšak vyskytly se zde nedostatky i přebytky, které bylo potřeba vyřešit.

Problémy

Když se nejprve zaměříme na přebytky, tak zde najdeme několik tabulek, které byly v původním návrhu a již nejsou vyžadovány. Příklady představují tabulky jako například chat nebo obrázky. V původním návrhu byla možnost, aby dobrovolníci mohli napřímo pomocí chatu spolu komunikovat nebo komunikovat s koordinátorem, což se postupem času neuplatnilo a chat zde zůstal vlastně jen pro komunikaci s vývojářem, když byla potřeba něco poupravit pro individuální dobrovolníky. Většina této komunikace o úpravách se pak týkala přizpůsobování aplikace pro částečně nevidomého dobrovolníka. Další zmíněnou přebytkovou tabulkou pak byla tabulka obrázků. V této tabulce se ukládaly obrázky, které dobrovolníci mohli přidávat k docházce. Nakonec tato funkce přestala být využívána. Proto bylo rozhodnuto, že se přestane používat. V nové databázi je však připravena tabulka tFiles, která v sobě drží převážně smlouvy, avšak je připravena tak, aby zde mohly být i jiné typy souborů.

Zbylé problémy se pak týkají třeba špatného využití primárních a cizích klíčů, které nám moc nedovolí aplikaci rozšiřovat. Příkladem pak je sloupeček jmeno_u, který funguje jako primární klíč a propojuje ostatní tabulky, kde se chová jako cizí klíč. Což se postupně ukázalo jako velice neefektivní, obzvláště pokud bylo potřeba změnit uživatelské jméno, který tento sloupeček reprezentuje. Dalším problémem

pak byla veliká shodnost, neboť se jednalo o jméno povětšinou vygenerované dle jména a příjmení dobrovolníka.

Pokud vezmeme v potaz nové rozšíření aplikace, které umožňuje přístup vícero organizacím, tak by byla rozšiřitelnost velice složitá, a proto bylo potřeba založit kompletně novou databázi, která by reflektovala veškeré nové a podstatné prvky. Přestože se jedná pouze o přidání jedné tabulky a vazebního sloupečku, tato návaznost by byla vzhledem ke stavu původní databáze velice složitá.

Převodník

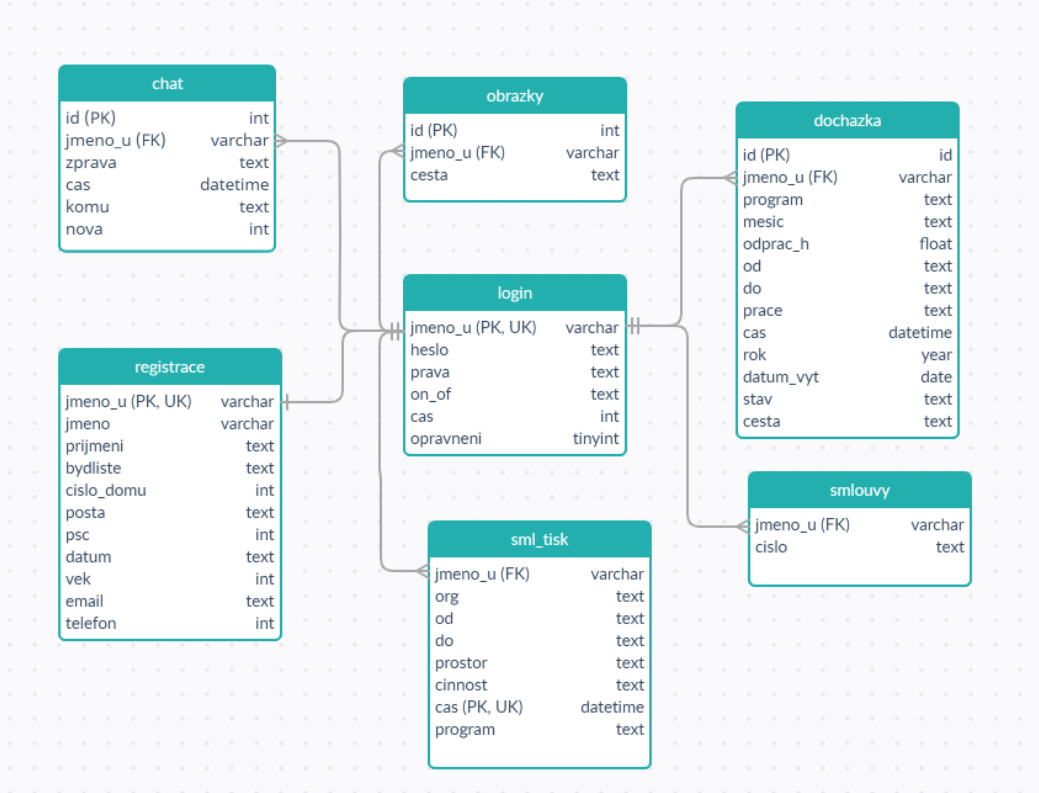
Díky tomu, že nová databáze obsahuje trochu jiné rozložení sloupečků a tabulek, bylo potřeba vytvořit převodník. Tento převodník vezme data tabulku po tabulce a postupně je transformuje tak, aby data byla korektní. Korektnost dat je v tuto chvíli velice důležitá, neboť původní aplikace procházela časem spousty změn, i co se týče jednotlivých názvů, vázaných na programy, které se ukládaly napřímo do databáze. Převodník tedy musel být schopen chybná, zastaralá anebo nekorektní data rozeznat a vložit je do databáze správně.

Největší problém pak tedy vznikal například u již zmíněných programů. Kdy existovala velká škála možností, se kterými bylo potřeba počítat a správně je převést. Původní názvy, jako je například Pět P, Velký kamarád a spousta dalších, se převáděly na kódové označení, kterým poté jsou čísla od 1. Tyto čísla se pak podle aktuálních názvů přepisují na ty korektní. Další problémy pak vznikaly při převodu již zmíněného klíče, jmeno_u. To nyní dostalo vlastní nový sloupeček a jak je vidět v nákresu databáze, tak propojení nyní vzniklo pomocí id, které se automaticky generuje. Výše zmíněné problémy představují pouze ty nejpodstatnější, ale bylo zde i vícero menších problémů, jako je například převod dat a správného přiřazování sloupečků a posloupnost tabulek.

Hesla

Aby se zaručila co největší bezpečnost, má nová databáze také jiný přístup k zakódování hesel a přístupu k nim. V původním návrhu se heslo kódovalo pomocí

jména, kterým se uživatel přihlašoval; nyní se kóduje pomocí speciálního klíče, který se generuje pro každou osobu zvlášť. Aby se tak předešlo problémům s přihlášením, tak se pro uživatele vytvořil speciální sloupeček, který zastává funkci nového hesla. Po převodu uživatel obdrží email s hodnotou tohoto sloupečku a po přihlášení je vyzván ke změně hesla, které již následuje novou strukturu kódování hesla. Po změně hesla je pak toto dočasné heslo smazáno, aby nebylo zneužito.



Obr. 16 Stará databáze
Zdroj: vlastní tvorba

Na obrázku **Obr. 16 Stará databáze**, lze vidět, jak vypadá původní databáze. Z tohoto nákresu chyby nejsou příliš patrné. Ale již ze samotného využití jmeno_u, které má datový typ varchar, jako primárního a cizího klíče je zřejmé, že by bylo při několika organizacích problémové. Můžeme také spatřit podobnost s novou databází, a to hlavně strukturou, kdy se všechny tabulky napojují na tabulku login, a to ze stejného důvodu. Každá činnost je provedena uživatelem a je proto tedy potřeba udělat záznam.

6.1.11 Api požadavky

Aby bylo možné dobře propojit mobilní a webovou aplikaci, bylo zapotřebí vymyslet správný přístup. Propojení těchto dvou jednotlivých samostatných entit bylo zapotřebí hlavně z důvodu potřeby sdílení základních prvků. Těmi pak jsou například uživatelská data pro jasnou identifikaci uživatele nebo kompletní databázová struktura a data pro docházku. Toto propojení je pak možné za pomoci sdílené databáze, kdy obě aplikace přistupují ke stejné databázi, a tak mohou data jednoduše sdílet, anebo využití API požadavků.

Nejprve si specifikujeme, co je API a jak je v naší aplikaci využito a proč zrovna API. Jedná se o rozhraní pro programování aplikací. Může jít o určité množství funkcí, protokolů či celých tříd, které mohou být využívány třetí stranou. V našem případě pak je třetí strana naše mobilní aplikace. Jednoduše řečeno se jedná o přístup, kdy programátor dovolí třetí straně přistupovat k funkcím jeho aplikace, aniž by musel využívat nějaké grafické rozhraní.

Díky celkové struktuře této webové aplikace pak API rozhraní je vlastně jednoduchý prezenter. Tento prezenter, stejně jako ostatní, je přístupný pomocí URL adresy a je schopen přijímat i různé argumenty. Díky tomu, že se jedná o strukturu prezenteru, můžeme jednoduše využívat veškerou funkcionalitu celé aplikace a poskytovat ji třetí straně. Třetí strana pak pouze zašle URL nebo API požadavek, naše aplikace ho ve správné funkci zpracuje a následně vrátí hodnotu pomocí zápisu JSON. Jedná se vlastně jen o pole informací, které je převedeno do jednodušší struktury, která se dá poslat nazpátek jako text a příjemce si pouze tuto informaci převede zpátky na pole a zjistí si informace, které potřebuje.

API bylo vybráno právě z toho důvodu, že může využívat funkcionality celé aplikace. Díky tomuto přístupu není potřeba, aby mobilní aplikace měla nějakou složitou strukturu nebo výpočetní funkce, které by se následně dotazovaly na databázi. Stačí pouze poslat požadavek s parametry, webová aplikace tento

požadavek splní a následně odešle zpátky hodnoty. Ty jsou zpracovány a mobilní aplikace si s nimi nakládá tak, jak potřebuje. Tento přístup nám tedy umožní to, že nebude vznikat velká duplicita kódu, veškerá logika zůstane pouze na jednom místě a obě aplikace mohou využívat své přednosti.

6.2 Mobilní aplikace

Mobilní aplikace jako taková je opravdu jednoduchá. Jediným jejím úkolem je totiž umožnit uživateli snáze přistoupit ke svým docházkám nebo je přidat ve chvíli, kdy ji dokončí. Díky tomu, že se jedná o mobilní aplikaci, umožňuje několik usnadňujících prvků. Jedním z nich je urychlené přihlášení, o kterém budeme hovořit níže, nebo pak samotné rozložení aplikace, které slouží vlastně k jedinému účelu – k možnosti snadného vytváření docházky.

Naše aplikace, jako většina mobilních aplikací, pak potřebuje některá oprávnění. Aby aplikace mohla plně fungovat, potřebuje samozřejmě přístup k internetu, který je automaticky dovolen aplikaci, když si o přístup zažádá. Podobným oprávněním, které pak naše aplikace potřebuje, je oprávnění k biometrickým prvkům telefonu, a to právě k možnosti přihlášení.

6.2.1 Rozložení aplikace

Díky tomu, že je aplikace vlastně pouze jednoúčelová a využívá API požadavků z webové aplikace, je její samotné rozdělení velice jednoduché. Toto rozdělení nám pak říká, že je poté aplikace samozřejmě i přehlednější a snadno se pochopí její jednoznačný účel. V celku se vlastně jedná pouze o tři jednoduché obrazovky, kde se nachází vše, co dobrovolník v této aplikaci potřebuje. Každá tato obrazovka má pak vlastní třídu, která se stará o veškeré její záležitosti. Tyto tři jednoduché obrazovky jsou pak rozděleny do dvou typů. Prvním z nich je aktivita a druhým je fragment. Oba se ve své podstatě liší, ale oba napomáhají vykreslení potřebného objektu, nebo celé stránky a zároveň se také starají o fungování prvků na stránce, jako například akcí volaných po stisku tlačítka. Krom těchto dvou prvků se v aplikaci také nachází layouty a vykreslovací objekty, které jsou psány ve formátu XML.

Aktivita

Díky tomu že aplikace jako taková není složitá a neskládá se z velkého množství oken, tak nám stačí pouze dvě aktivity. První z aktivit se stará o vykreslení přihlašovací stránky, kde se dobrovolník může přihlásit do aplikace pomocí stejných údajů jako do webové aplikace, ale o přihlášení pojednáváme v kapitole o pár řádků níže. Další aktivitou je pak takzvaný drawer, dále jen menu, jehož popis byl zmíněn v kapitole o použitých technologiích. Pro rychlé a obecné shrnutí se jedná o aktivitu, která je schopna vykreslit jednu aktivitu, která se dělí na fragmenty. Hlavním prvkem této aktivity je pak menu, které se zobrazuje na straně obrazovky a dovoluje uživateli snadno se pohybovat mezi obrazovkami. Naše menu poté slouží pro pohyb po celém prostoru aplikace, který se uživateli zobrazí po přihlášení. V menu jako v takovém pak můžeme najít již zmíněné odrážky přidání docházky, seznam docházky anebo odhlášení. Samotné menu pak potřebuje hlavní obrazovku, fragment, který vykreslí jako první, a to je v našem případě přidání docházky.

Fragment

Naše menu pak má pod sebou dvě aktivity, přesněji dva fragmenty. Jedním z nich je zmíněné přidávání docházky a druhým je poté seznam docházky. Jak funguje fragment a k čemu jej používáme, máme již vysvětleno z kapitoly Použité technologie v sekci Android. Pro rychle zopakování se jedná o jakousi znovupoužitelnou část, dalo by se říci, že jde o aktivitu, co se dá znovu použít se vším všudy, od vzhledu až po fungování. Naše menu pak v sobě drží tyto dva fragmenty, mezi nimiž může uživatel snadno přepínat. Nachází se zde další fragment, kterým je odhlášení. Jeho jedinou funkcí pak je možnost dovolit uživateli se z aplikace odhlásit, a tedy přerušit komunikaci s webovou aplikací a smazat přihlášené údaje. Nemá tedy ani vlastní vzhled, pouze funkcionalitu odhlášení.

6.2.2 Přihlášení

Přihlášení do aplikace je rozděleno do dvou částí, které umožní dobrovolníkovi snazší práci s aplikací a rychlejší přístup. Obě části vlastně dělají totéž, avšak k oběma se přistupuje úplně jiným způsobem.

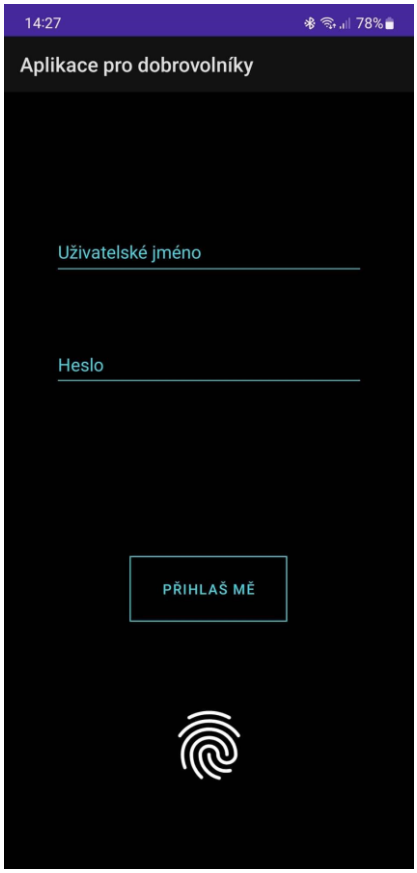
Prvním možným způsobem přihlášení je klasické přihlášení pomocí údajů stejných, jako jsou do aplikace webové. Možnost se přihlásit stejnými údaji a získat i stejná data, aniž by vznikala jejich redundance, nebo se k nim přistupovalo z dvou různých aplikací, je díky již zmíněnému API požadavku.

Žádost o přihlášení

Princip v mobilní aplikaci je pak velice jednoduchý. Uživatel, v našem případě dobrovolník, zadá své přihlašovací údaje a následně aplikaci zažádá o přihlášení. Mobilní aplikace převezme uživatelská data v případě nesplnění pravidel, jako je třeba chybějící heslo, vyzve uživatele k vyplnění. Pokud je však vše v pořádku, vezme tato data a pomocí API požadavku, který je specifikován pomocí URL adresy, vyšle žádost s daty na webovou aplikaci. V okamžik, kdy data dorazí do přesně určené funkce, která je specifikovaná v URL adrese (více jsme si řekli při popisu presenterů v popisu webové aplikaci), se tato uživatelská data zpracují, stejně jako by se uživatel přihlašoval přímo do webové aplikace. Následně se zpátky mobilní aplikací vrátí JSON s výsledkem zpracování dat. Pokud je odpověď na požadavek kladná, mobilní aplikace vytvoří objekt uživatele, kam uloží veškerá jeho potřebná data. Krom vytvoření objektu také uloží data přímo do mobilního zařízení. Následně pak uživatele přesměruje na obrazovku po přihlášení, která je reprezentovaná pomocí menu neboli draweru.

Druhým možným způsobem přihlášení je pak pomocí biometrických údajů. Přesněji pak pomocí otisku prstu. Aby však tato varianta fungovala, je potřeba, aby byl uživatel nejprve alespoň jednou přihlášen klasickým způsobem pomocí svých uživatelských dat pro přihlášení do webové aplikace. Důvod je prostý – po klasickém

přihlášení, krom zmíněného vytvoření uživatelského objektu, se také uloží přihlašovací údaje do telefonu. Aby však nebylo úplně narušena bezpečnost, ukládá se pouze identifikátor a přihlašovací jméno uživatele. Tedy i kdyby se někomu podařilo data získat, nezíská heslo, a tak ani přístup do webové aplikace, kde je vícero úkonů, které lze dělat, jako například editovat informace o přihlášeném uživateli. Po ověření zmíněného otisku prstu pak aplikace automaticky provede proces zmíněný výše v kapitolce žádost o přihlášení. Výhodou tohoto biometrického přihlašování pak je, že uživatel nemusí zadávat stále svoje údaje a stačí mu pouze přiložit prst a přihlásit se.



Obr. 17 Přihlášení do mobilní aplikace
Zdroj: vlastní tvorba

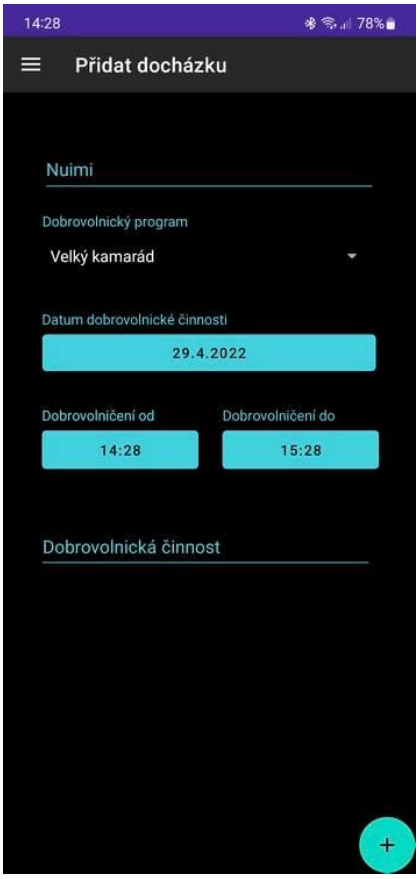
Obrázek Obr. 17 Přihlášení do mobilní aplikace ukazuje, jak vypadá úvodní obrazovka po otevření aplikace. Jedná se o stránku, kde se může dobrovolník přihlásit do aplikace a následně vyplňovat docházku.

6.2.3 Vytváření docházky

Jak již bylo zmiňováno, tak vytváření docházky je jeden z fragmentů našeho menu. Je to vlastně pouze jednoduchý formulář, který funguje na stejném principu, jako ten, jenž je ve webové aplikaci. Dobrovolník naprosto identicky, akorát za pomocí formuláře, který je upraven pro mobilní aplikace, vyplní data stejně jako ve webové aplikaci. Vybere program, vybere den, nastaví čas, od kdy do kdy dobrovolnická akce probíhala, a následně stiskne tlačítko plus, které symbolizuje vytvoření docházky.

Po stisku tlačítka začíná pravý úkon aplikace. Aby vše bylo správně, proběhne kontrola dat, zejména zdali uživatel vyplnil všechna pole. Pokud ne, je uživateli oznámeno, která pole má vyplnit. V moment, kdy je vše z pohledu formuláře správně, tak se provede API požadavek pro vytvoření docházky. Podobně jako u přihlášení se API požadavek stanoví pomocí URL adresy. Danému požadavku se pošlou data, ta se na straně webové aplikace zpracují a uloží se do databáze tak, jako kdyby uživatel vytvářel docházku přímo z webové aplikace. Jediným rozdílem je zde odlišný výpočet data. Je potřeba dopočíst rozdíl mezi odpracovanými hodinami a vytvořit z hodnot od do datový typ DateTime. Následně se provede odeslání zpětné vazby pro uživatele do mobilní aplikace a ta se zobrazí.

Obrázek **Obr. 18 Přidání docházky v mobilní aplikaci**, který je uvedený níže, představuje vzhled formuláře, kde dobrovolník, který je do aplikace může přidávat docházku. Jedná se o takřka totožný formulář jako je ve webové aplikaci, akorát je více přizpůsobený pro mobilní zařízení, a tak se s ním snáze pracuje.



Obr. 18 Přidání docházky v mobilní aplikaci
Zdroj: vlastní tvorba

6.2.4 Seznam docházky

Stejně jako tomu je u obrazovky přidávání docházky, i seznam je pouze fragment z menu. Tento fragment je pak specifický tím, že slouží k zobrazení seznamu docházky přihlášeného uživatele, a to pomocí takzvaného List View. Do tohoto zobrazení se pošle jednoduchý layout. Tento layout nám specifikuje, jak se jednotlivé položky našeho pole mají zobrazovat uživateli. Vlastně specifikujeme, jak bude vypadat jeden celý item v daném listu. V našem případě na jednom itemu budeme zobrazovat informace o docházce, jako je například v jakém programu proběhla, kdy proběhla a jaká byla činnost. Protože tyto informace dostaneme v poli, je potřeba je nějak dostat na správné pozice v itemu a ve správném počtu. K tomuto slouží adaptér. Ten můžeme využít předdefinovaný anebo využít vlastní. Díky tomu

že zobrazujeme tolik informací a máme vlastní layout, je lepší využít vlastní adaptér. Tomuto adaptéru se pak specifikuje, kam přesně jakou informaci má vypsat a tolikrát, kolikrát informace dostane, tolik vytvoří itemů, které bude následně vypisovat v listu.

Získání informací pro tento list je pak stejné jako ve všech předchozích případech. Vytvoří se API požadavek na webovou aplikaci, který je přesně specifikovaný pomocí URL adresy. Tentokrát však nemáme žádný formulář ani informace z telefonu, ale máme objekt uživatele, který vznikl při přihlašování. V tomto objektu máme uživatelské jméno, ale také uživatelský identifikátor, který v tomto případě použijeme jako parametr pro API. API převezme uživatelské ID a získá potřebné informace a následně je vrátí v podobě JSON. Protože se však jedná o multidimenzionální pole, je potřeba JSON parsovat na dvakrát. Nejprve se rozšifruje základní struktura, která řekne, kolik docházek, respektive kolik záznamů jsme z databáze obdrželi. Na základě této hodnoty se nastaví velikost polí stringu, které jsou potřeba pro adaptér. Poté for cyklus projde JSON znovu a rozšifruje každý záznam z databáze. Tyto záznamy pak rozpadne podle typu přímo do polí pro ně specifikovaných. Program půjde do pole pro programy, datum pro pole s daty a tak dále. Následně se tyto jednotlivé pole pošlou do adaptéru a ten je zpracuje a vykreslí požadovaný list. Tento proces se pak opakuje pokaždé, když uživatel přijde na daný fragment. Díky své jednoduchosti je však rychlý a nijak uživateli nezpomalí jeho práci v aplikaci. Přece jen se jedná pouze o informační prvek, který uživateli říká, jaké docházky má vytvořené, a také může sloužit jako kontrola, že požadavek na vytvoření docházky v mobilní aplikaci byl opravdu úspěšně splněn.

Obr. 19 parsování JSON pro list view nám ukazuje, jak vypadá zpracování odpovědi na rquest požadavek pro získání docházky přihlášeného dobrovolníka. Jak již bylo zmíněno, jedná se o parsování dat z JSON záznamu a následný převod do adaptéru, který je pak schopen data zobrazit v listu, který je čitelný pro běžného uživatele.

```
JSONObject responseJSON = new JSONObject(response);

int length = responseJSON.getJSONArray( name: "data").length();

programs = new String[length];
descriptions = new String[length];
from = new String[length];
to = new String[length];
date = new String[length];

JSONArray jsonArray;
JSONObject jsonAttendance;
for (int i = 0; i < length; i++)
{
    jsonArray = responseJSON.getJSONArray( name: "data");
    jsonAttendance = new JSONObject(jsonArray.getString(i));
    programs[i] = jsonAttendance.getString( name: "program");
    descriptions[i] = jsonAttendance.getString( name: "description");
    from[i] = jsonAttendance.getString( name: "workFrom");
    to[i] = jsonAttendance.getString( name: "workTo");
    date[i] = jsonAttendance.getString( name: "date");
}

MyAdapter adapter = new MyAdapter(root.getContext(), programs, descriptions, from, to, date);
```

Obr. 19 parsování JSOn pro list view
Zdroj: vlastní tvorba

7 Zpětná vazba

Aby bylo docíleno toho nejlepšího výsledku, byla po celou dobu vývoje poskytována jakási „pracovní verze“ dobrovolnické organizaci Jičín, pro kterou byl vývoj dělán primárně. Tato verze v sobě vždy obsahovala aktuálně řešený vývoj, či změnu. Přesněji zde tedy byla aktualizovaná úprava každé funkcionality. Díky tomu k ní měli dobrovolníci i koordinátoři přístup a mohli si ji pomocí univerzálních přihlašovacích údajů proklikávat a testovat. Každý tento uživatelský přístup pak měl samozřejmě svoje práva podle toho, jak se aplikace vyvíjela, až dosáhli práv, tak jak jsou finálně popsána v kapitole o oprávněních, několik stran výše.

Díky tomu, že byla takto přístupná verze, mohla vznikat snadná zpětná vazba na aktuální stav aplikace, podle které se mohla aplikace lépe přizpůsobit potřebám jak koordinátorům, tak dobrovolníkům. Tato zpětná vazba probíhala primárně prostřednictvím emailové konverzace, a to i z důvodu pandemie covid-19. I přestože se jednalo pouze o emailovou konverzaci, byli jsme schopni vychytat spoustu chyb a připravit aplikaci tak, aby byla snadno srozumitelná pro všechny věkové kategorie a také jak pro dobrovolníky, tak pro koordinátory.

Webová aplikace

Když vyjmenuji některé zpětné vazby, tak šlo prvky týkající se především vzhledu, kde dobrovolníci i koordinátoři požadovali jednoduchost. Aby přesně věděli, kde jsou a co se zde dá dělat. Také aby se snadno dostali na všechny prvky, které potřebují, a to např. na úpravu svých osobních údajů, změnu hesla či aby koordinátor předně viděl docházku dobrovolníků nežli možnost ji sám založit. Jedním z prvků zpětné vazby, co už spíše byl požadavek, který vyplynul z vytváření docházky, byl požadavek na změnu programů. Dobrovolnická centra získala nové akreditace, a tak bylo potřeba přizpůsobit staré programy novým tak, aby se nic neztratilo. Výhradně šlo o zmenšení počtu a vytvoření nových názvů. Prvotní poskytované služby stále zůstávaly, avšak byly převedeny do nového kabátu a seskupeny tak, aby byly přehlednější a srozumitelné. Příkladem je třeba

sjednocení nic neříkajícího programu K2 do nového programu dobrovolník v sociálních službách, který již svým názvem sám napoví, o co se jedná, a dobrovolník si pak snáze vybere, pod kterým projektem chce pracovat, nebo pod který projekt bude v aplikaci vykazovat svoji docházku.

Mobilní aplikace

Výše zmíněné prvky, které byly obdrženy formou zpětné vazby, se však týkaly pouze webové aplikace, a to proto, že aby mohla vzniknout mobilní aplikace, bylo potřeba kvůli API požadavkům nejprve dokončit prvotní aplikaci. Zpětná vazba ohledně mobilní části byla pouze jednorázová a týkala se pouze rychlého ozkoušení, zdali vše funguje a schválení jednoduchého vzhledu a snadného a srozumitelného použití. Tato zpětná vazba byla pouze od jednoho dobrovolníka. Bohužel vývoj mobilní aplikace byl dokončen chvíli před začátkem války na Ukrajině. Z důvodu toho, že je potřeba (nebo by spíše bylo dobré) aplikaci před reálným spuštěním nechat dobrovolníky proklikat, vyzkoušet a následně na základě zpětné vazby případně upravit, byla aplikace hned po dokončení zaslána dobrovolnickému centru. Avšak kvůli zmíněné válce aplikace byla testována pouze zběžně.

Přes všechny nastalé problémy se však pomocí zpětné vazby, ať byla jakákoli, povedlo dotáhnout potřebné prvky aplikace a ta v sobě nese části každého, kdo svojí zpětnou vazbou přispěl.

8 Shrnutí výsledků

Díky již zmíněné zpětné vazbě je zřejmé, že se aplikace povedlo dokončit. Veškeré prvky, které jsou pro obě aplikace potřebné a jsou nezbytné k jejímu fungování, jsou využity a úspěšně implementovány. Jak již bylo zmíněno několikrát v předešlých kapitolách, vznikly aplikace dvě. Jednou je aplikace webová a druhou je aplikace mobilní, pro operační systém Android. Obě tyto aplikace jsou spolu provázány. Samotná webová aplikace pak může žít sama, aniž by ke svému životu potřebovala něco jiného než server, na kterém poběží, a databázi, k níž může přistupovat. Naopak mobilní aplikace potřebuje ke svému životu aplikaci webovou, a to proto, že využívá její prvky.

Obě aplikace pak slouží dobrovolníkům a koordinátorům k urychlení a usnadnění práce. Ať už se bude jednat o základní vytvoření dobrovolnické činnosti, její uskladnění nebo předávání koordinátorovi. Také napomáhají v ukládání dat o dobrovolnících, které drží aktuální díky samotným dobrovolníkům a snadné zprávě. Dále pak umožňuje vytvářet reporty pomocí excelových tabulek, čímž se opět o něco více ulehčí a urychlí práce lidem „pracujícím“ v dobrovolnických organizacích.

Aplikace na rozdíl od původní přináší možnost, aby ji mohlo využívat vícero dobrovolnických organizací. Tato vlastnost byla vyzkoušena a funkčnost potvrzena. Bohužel, kvůli již zmíněné probíhající válce na Ukrajině se zatím další dobrovolnická centra nemohla připojit a aplikaci využívat. Doufejme snad, že k tomu dojde v budoucnu.

Nová aplikace, porovnáme-li ji s tou původní, postrádá několik prvků. Jedním z nich je například možnost vytvořit smlouvu při vytváření uživatelského účtu. Tato možnost nebyla přidána, neboť v minulosti byla málokdy využívána. Zatím zůstává její náhrada, a to ta, že si lze stáhnout aktuální smlouvy rovnou připravené k tisku. Dobrým vylepšením do budoucna by pak byla možnost tyto vytvořené smlouvy do

aplikace nahrát. Popřípadě na jejich základě vytvářet nové reportovací prvky a prakticky přenést celou korespondenční část mazy dobrovolníkem a organizací do aplikace.

9 Závěr a doporučení

Vytvořené aplikace splňují veškeré základní požadavky a jsou tak vhodné pro použití v reálném provozu. Avšak z důvodu již zmíněné války na Ukrajině a některým covidovým opatřením, která během vývoje nastala, se aplikace nezvládla plně otestovat a nasadit se do reálného provozu. I přes pouze drobné testování obě aplikace jsou plně funkční, uživatelsky přívětivé a snadno ovladatelné. Prozatím běží pouze v testovacím prostředí a do budoucna jsou připraveny pro reálné použití i vícero charitativními organizacemi. Obě aplikace budou stále vyvíjeny a vylepšovány tak, aby byly stále aktuální a aby co nejvíce pomáhaly jak dobrovolníkům, tak koordinátorům.

10 Seznam použité literatury

[1] PHP. Wikipedia [online]. naposledy editována 16. 2. 2022 [cit. 2022-04-19]. Dostupné z: https://cs.wikipedia.org/wiki/PHP

[2] Nette. Nette – Pohodlný a bezpečný vývoj webových aplikací v PHP [online]. 2008 [cit. 2022-04-19]. Dostupné z: https://nette.org/cs/

[3] MVC architektura. ITnetwork [online]. 2013 [cit. 2022-04-19]. Dostupné z: https://www.itnetwork.cz/navrh/mvc-architektura-navrhovy-vzor

[4] MVC pattern [online]. Microsoft, 2022 [cit. 2022-04-29]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-6.0

[5] Why MySQL?. MySQL [online]. [cit. 2022-04-19]. Dostupné z: https://www.mysql.com/why-mysql/

[6] Dibi. Dibi tiny 'n' smart database layer [online]. 2008 [cit. 2022-04-19]. Dostupné z: https://dibiphp.com/cs/

[7] DibiFluent. Php Fashion [online]. 2020 [cit. 2022-04-19]. Dostupné z: https://phpfashion.com/dibifluent-tekute-sql-prikazy

[8] Latte. Latte – nejbezpečnější & opravdu intuitivní šablony pro PHP [online]. 2008 [cit. 2022-04-19]. Dostupné z: https://latte.nette.org/cs/

[9] Less. Less [online]. 2010 [cit. 2022-04-19]. Dostupné z: https://lesscss.org/

[10] Bootstrap. Bootstrap 5.1 [online]. 2021 [cit. 2022-04-19]. Dostupné z: https://getbootstrap.com /

[11] Co je AJAX. Strafelda [online]. 2022 [cit. 2022-04-19]. Dostupné z: https://www.strafelda.cz/ajax

[12] PhpSpreadsheet. PhpSpreadsheet Documentation [online]. 2020 [cit. 2022-04-19]. Dostupné z: https://phpspreadsheet.readthedocs.io/en/latest/

[13] Android. ITnetwork [online]. 2018 [cit. 2022-04-19]. Dostupné z: https://www.itnetwork.cz/java/android/zaklady/tutorial-programovani-pro-android-v-jave-uvod

[14] Java (programovací jazyk). Wikipedia [online]. naposledy editována 2. 1. 2022 [cit. 2022-04-19]. Dostupné z: https://cs.wikipedia.org/wiki/Java_(programovac%C3%AD_jazyk)

[15] Layouts. Developer [online]. 2021 [cit. 2022-04-19]. Dostupné z: https://developer.android.com/guide/topics/ui/declaring-layout

[16] XML introduction. Developer [online]. 2022 [cit. 2022-04-19]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction

[17] Activity. Developer [online]. 2022 [cit. 2022-04-19]. Dostupné z: https://developer.android.com/reference/android/app/Activity

[18] Fragments. Developer [online]. 2021 [cit. 2022-04-19]. Dostupné z: https://developer.android.com/guide/fragments

[19] Update UI components with NavigationUI [online]. Android, 2021 [cit. 2022-04-29]. Dostupné z: https://developer.android.com/guide/navigation/navigation-ui

[20] Volley overview. Google github [online]. 2020 [cit. 2022-04-19]. Dostupné z: https://google.github.io/volley/

[21] Send a simple request [online]. 2020 [cit. 2022-04-29]. Dostupné z: https://google.github.io/volley/simple.html

11 Přílohy

Data na přiloženém úložném médiu

1) *2022-04-29_Dochazka.xlsx* – obsahuje ukázkový report pro docházku s fiktivními testovacími daty
2) *webovaAplikace* – obsahuje zdrojové kódy webové aplikace
3) *androidAplikace* – obsahuje zdrojový kód mobilní aplikace
4) *volunteerApp.apk* – jedná se o instalační soubor pro instalaci android aplikace pro přidávání docházky

Oskenované zadání práce

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2020/2021

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Daniel Vondra**
Osobní číslo:	**I1900292**
Adresa:	Doubrava 103, Loukov, 29411 Loukov u Mnichova Hradiště, Česká republika
Téma práce:	Aplikace pro oblastní charity ČR
Téma práce anglicky:	Application for the regional charity of the Czech Republic
Vedoucí práce:	Ing. Barbora Tesařová, Ph.D. Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Cíl: Cílem je vytvořit novou aplikaci pro správu dobrovolníku kteří jsou vedeni pod oblastními charitami po celé České republice. Aplikace se bude skládat ze dvou částí. První bude webová aplikace, která vznikne refaktoringem a předěláním stávající, aplikace a to tak, že bude zmodernizovaná za použití MVC modelu, který zde není. Dále pak bude rozšířena pro všechny oblastní charity v ČR. Druhou částí bude vytvoření mobilní aplikace, která bude fungovat jako docházková aplikace, kde si budou moci dobrovolníci vyplňovat svoji docházku v online i off-line režimu. Obě tyto části budou propojené.

Hlavní body:

1. Analýza aktuálního stavu aplikace
2. Navržení nové databáze
3. Připravení nového vzhledu aplikace
4. Přenesení a upravení stávající funkcionality do MVC modelu
5. Doplnění API rozhraní pro mobilní aplikaci
6. Vytvoření mobilní aplikace
7. Přenesení dat ze staré aplikace

Využité technologie:

- Databáze: MySQL
- Webová aplikace: PHP v objektové podobě verze 7.4 a vyšší
  - Použití vlastního MVC frameworku na bázi nette
  - Dále pak latte (HTML), less (CSS), javascript, Dibi (databáze)
- Mobilní aplikace: java s napojením na MySQL a využitím PHP API

Seznam doporučené literatury:

Books:

1. Lem Java for Android Development
2. MySQL cookbook
3. Programing php

Podpis studenta: 

Datum: 29. 4 2022

Podpis vedoucího práce:

Datum: