

Katedra informatiky  
Přírodovědecká fakulta  
Univerzita Palackého v Olomouci

# BAKALÁŘSKÁ PRÁCE

Mobilní aplikace pro pracovníky pečovatelské služby



2020

Vedoucí práce: Mgr. Janoščík Ra-  
dek

Petr Janiš

Studijní obor: Aplikovaná informatika,  
prezenční forma

## **Bibliografické údaje**

Autor:	Petr Janiš
Název práce:	Mobilní aplikace pro pracovníky pečovatelské služby
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2020
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Janoščík Radek
Počet stran:	19
Přílohy:	1 CD/DVD
Jazyk práce:	český

## **Bibliographic info**

Author:	Petr Janiš
Title:	Mobile application for workers of the day care
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2020
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Janoščík Radek
Page count:	19
Supplements:	1 CD/DVD
Thesis language:	Czech

## Anotace

*Ukázkový text závěrečné práce na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci, který je zároveň dokumentací stylu pro text práce v  $\text{\LaTeX}$ . Zdrojový text v  $\text{\LaTeX}$  je doporučeno použít jako šablonu pro text skutečné závěrečné práce studenta.*

## Synopsis

*Sample text of thesis at the Department of Computer Science, Faculty of Science, Palacký University Olomouc and, at the same time, documentation of the  $\text{\LaTeX}$  style for the text. The source text in  $\text{\LaTeX}$  is recommended to be used as a template for real student's thesis text.*

**Klíčová slova:** styl textu; závěrečná práce; dokumentace; ukázkový text

**Keywords:** text style; thesis; documentation; sample text

Thanks

*Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.*

datum odevzdání práce

podpis autora

# Obsah

<b>1</b>	<b>Styly pro psaní bakalářských a diplomových prací</b>	<b>7</b>
1.1	Požadavky a podprovaná prostředí . . . . .	7
1.2	Přepínače . . . . .	7
1.3	Geometrie stránky . . . . .	7
<b>2</b>	<b>Sazba částí dokumentu</b>	<b>9</b>
2.1	Sazba úvodní strany či obsahu . . . . .	9
2.2	Závěry . . . . .	9
2.3	Matematika . . . . .	9
2.4	Sazba literatury . . . . .	10
2.4.1	Sazba bibliografie přes BIB <sub>TEX</sub> . . . . .	10
2.4.2	Manuální sazba bibliografie . . . . .	10
2.5	Drobná makra . . . . .	11
2.6	Sazba rejstříku . . . . .	11
2.7	Sazba zdrojových kódů . . . . .	11
	<b>Závěr</b>	<b>15</b>
	<b>Conclusions</b>	<b>16</b>
<b>A</b>	<b>První příloha</b>	<b>17</b>
<b>B</b>	<b>Druhá příloha</b>	<b>17</b>
<b>C</b>	<b>Obsah přiloženého CD/DVD</b>	<b>17</b>
	<b>Seznam zkratk</b>	<b>19</b>

## Seznam obrázků

## Seznam tabulek

1	Seznam přepínačů . . . . .	8
2	Seznam přepínačů . . . . .	12

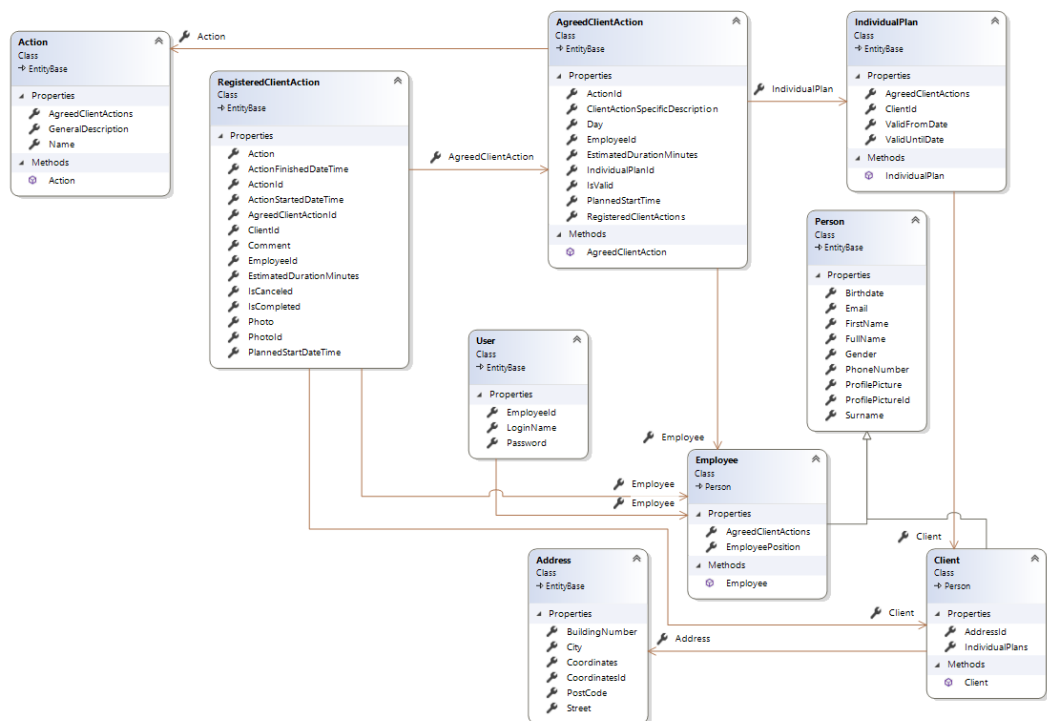
## Seznam vět

1	Definice (Název definice) . . . . .	13
	Důkaz (Název důkazu) . . . . .	13
2	Poznámka (Pumpovací věta) . . . . .	13
3	Příklad (Pumpovací věta) . . . . .	13
4	Lemma (Název definice) . . . . .	13
5	Důsledek (Název důkazu) . . . . .	13
6	Věta (Pumpovací věta) . . . . .	13

## Seznam zdrojových kódů

1	Volání třídy <b>kidiplom</b> . . . . .	9
2	Sazba závěrů . . . . .	10
3	C++ . . . . .	14
4	JS . . . . .	14
5	C# . . . . .	14
6	SQL . . . . .	14
7	TutorialD . . . . .	14

# 1 Úvod



Obrázek 1: Datový model

## 2 Vývoj mobilních aplikací

Mobilní aplikace by uživateli měla co nejjednodušeji a nejrychleji umožnit dosáhnout jeho cíle. Tento fakt je potřeba mít na paměti v průběhu celého procesu vývoje. Proto je důležité, aby aplikace běžela plynule a měla jednoduché uživatelské rozhraní. U aplikací určených pro širokou veřejnost, je dobrý návrh a provedení UI, faktorem rozhodujícím o její úspěšnosti na trhu. Co se týče mobilního softwaru vyvíjeného za účelem korporátního použití je to otázka financí pro firmu, která se ho rozhodne používat. Jednodušší ovládaní znamená méně potřebného času a prostředků vynaložených na zaškolení zaměstnanců.

Pro plynulý běh aplikace je důležité hned na začátku vývoje přesně určit za jakými účely bude používána. Tato specifikace společně s platformou na kterou by měl být vývoj cílen jsou rozhodujícími kritérii při výběru frameworku a způsobu vývoje.

V dnešní době se na trhu s mobilními telefony vyskytují v převážné míře pouze dva operační systémy. Android, který podporuje zařízení od vícero výrobců. A iOS od firmy Apple, který je určen přímo pro hardware od toho výrobce. Ovšem tyto systémy mají spoustu rozdílných verzí, samotná zařízení mají rozdílné velikosti, funkce a vnitřní komponenty. Tohle je právě to, co činí vývoj mobilních aplikací složitým a zdůrazňuje důležitost výběru správné metody. Aby bylo možné toto klíčové rozhodnutí udělat správně je potřeba se seznámit s detaily různých stylů vývoje a frameworky.

### 2.1 Nativní vývoj

Standardní styl vývoje, který byl dříve i jediným možným způsobem. Při této metodě je aplikace tvořena odděleně pro každou platformu zvlášť. Stejný kód je tedy potřeba napsat dvakrát v různých programovacích jazycích a použít různá vývojová prostředí a nástroje. Nehledě na to, že Apple neumožňuje vývoj nativních aplikací na jiném operačním systému než MacOS.

Pokud by tedy výsledný software měl běžet na obou platformách, což je ve většině případů předpokladem. Firma by musela mít dva týmy, každý pro jednu část a poskytnout rozdílné vybavení. Nejenom, že tímto způsobem bude vývoj trvat dlouho, ale zároveň bude i drahý. Na druhou stranu výsledkem bude snadno udržovatelná a výkonná aplikace umožňující využívat všech funkcí zařízení. Knihovny, které jsou použity spravují samotní vlastníci OS. Je tedy možné rychle nebo úplně bez zásahu reagovat na nové verze systémů a změny v designu nativních komponent.

Výsledný produkt je potřeba šířit mezi uživatele. K tomuto účelu slouží takzvaný "Store" a každá platforma má svůj vlastní. Obě firmy se snaží vytvořit bezpečné prostředí pro uživatele a kontrolují aplikace sdílené pomocí této služby. Otevřená povaha Androidu ovšem umožňuje mít více než jedno centrální místo pro získání aplikace a proto se snaží řešit část bezpečnostních rizik už na straně kódu a knihoven. Apple je v tomto ohledu daleko přísnější a kontroluje každou aplikaci, která je do App Storu přidána. Někdy trvá i několik dní než prohlásí



aplikaci nebo aktualizaci za důvěryhodnou. V tomto procesu se kontroluje hlavně jestli produkt opravdu dělá to co vydavatel tvrdí v popisu a splňuje všechny bezpečnostní normy. Tyto bezpečnostní aspekty za vývojáře do určité míry schopno pohlídat nativní vývojové prostředí a kompilátor. Tímto způsobem je tedy možné se vyhnout neschválení aplikace firmou Apple a bezpečnostních nedostatků na jiných platformách.

Tento přístup je tedy vhodný pro specifické typy projektů:

- Tam kde není potřeba, aby výsledná aplikace běžela na obou platformách.
- Je kladen důraz na výkon aplikace, nebo je to z podstaty požadavků nutnost.
- Hraje velkou roli bezpečnost a není žádoucí používání knihoven a frameworků třetích stran.
- Neklade se důraz na rychlost vydání aplikace na trh a cenu produktu.

### 2.1.1 Android

Aplikace pro Android mohou být napsány pomocí jazyků Kotlin, Java a C++. Nástroj Android SDK zkompiluje kód společně s daty a soubory obsahujícími zdrojové kódy do APK, což je archivní soubor s příponou .apk. APK soubor obsahuje všechny komponenty aplikace a zároveň je to soubor, který zařízení s operačním systémem Android používají k instalaci. [1] (*volný překlad*)

K vývoji pro platformu Android je tedy potřeba Android SDK a vhodné vývojové prostředí. Dle dokumentace je doporučovaným vývojovým prostředím Android studio. Velkou výhodou je, že tyto nástroje mohou běžet na jakémkoliv operačním systému.

### 2.1.2 iOS

iOS SDK je balíček nástrojů umožňujících vývoj mobilních aplikací na operační systém iOS. Nástroje umožňují vývojářům přistupovat k různým funkcím a službám iOS zařízení jako jsou hardwarové a softwarové atributy. Obsahuje také iPhone simulátory, které kopírují vzhled a pocit z užívání skutečných zařízení. Aby bylo možné aplikaci testovat a distribuovat ji přes App Store je potřeba zaplatit vývojářskou licenci. SDK společně s vývojovým prostředím Xcode pomáhá psát aplikace pro platformu iOS za použití oficiálně podporovaných jazyků, kterými jsou Swift a Objective-C. [2] (*volný překlad*)

## 2.2 Multiplatformní vývoj

Umožňuje vývoj na obě platformy současně a v některých případech pro ně používá i identický kód. Každý framework toto umožňuje trochu jiným způsobem. Základní myšlenka je ale taková, že na rozdíl od nativních aplikací, které komunikují přímo s operačním systémem a zařízením je u multiplatformních aplikací

přidána abstraktní mezivrstva neboli nativní obálka. Mezivrstva zaručuje to, že se aplikace pro systém telefonu tváří jako nativní a zároveň umožňuje kódu přístup ke všem periferiím zařízení jako jsou kamera, určování polohy, gyroskop atd. Vzhled takto napsaných aplikací bude naprosto totožný s nativní verzí. Tato obálka ovšem zpomaluje chod aplikace, protože každý proces, který vyžaduje komunikaci se systémem přes ní musí projít a mezivrstva jej musí zpracovat. Toto zpomalení je možno považovat za zanedbatelné pokud není cílem vývoje například hra, aplikace s robustním uživatelským rozhraním nebo s velkým množstvím rychle vstupujících a ?vystupujících? dat. Celkově tato metoda v porovnání s klasickým přístupem poskytuje levnější a většinou i rychlejší cestu k vývoji aplikace.

Na druhou stranu projekt může být vystaven riziku zastavení podpory pro zvolený framework, vrstva navíc znamená prostor pro více bezpečnostních nedostatků a každý vývojář musí být obeznámen s detaily fungování obou platform. I přesto, že je tímto způsobem možné vyvíjet aplikaci na obě platformy za použití počítače s jakýmkoliv operačním systémem, tak pro kompilaci a testování kódu bude v případě iOS nutné použít stroj s macOS.

### 2.2.1 React Native

React Native je open-source framework, který umožňuje vývoj aplikací pro Android a iOS za použití Reactu a nativních možností platformy. Používá skriptovací jazyk JavaScript za pomoci kterého se přistupuje k API konkrétní platformy, zároveň popisuje vzhled a chování uživatelského rozhraní za použití komponent Reactu. [3] (*volný překlad*)

Důležitou vlastností je to, že výsledný kód je kompilován do nativního jazyku používaný platformou. Velkou výhodou je, že pokud žádná z poskytovaných knihoven neobsahuje danou funkcionalitu je možné vytvořit modul v některém z nativních jazyků a zaintegrovat jej do projektu. Za kvalitu frameworku mluví to, že je aktuálně nejpoužívanější ve své kategorii a jsou pomocí něj vytvořeny aplikace jako Facebook a Instagram, které denně používají miliony uživatelů.

### 2.2.2 Xamarin

Xamarin je opět platforma, která umožňuje multiplatformní vývoj tentokrát ale s rozdílem, že společně s mobilními verzí je možné tvořit i aplikaci pro Windows. Patří do rodiny nástrojů .NET čili kód je psán v programovacích jazycích C# a F#. Microsoft využívá už nabitých zkušeností vývojářů s jejich předešlým nástrojem WPF a architekturou Model-View-ViewModel. Většina kódu je sdílená, Microsoft uvádí až 90% [4], ale je možné určité části vytvořit speciálně pro jednu platformu. V této části je struktura kódu a volání systémového API velice podobná nativnímu vývoji, pouze za použití jiného jazyka. ?MONO?

### 2.2.3 Capacitor

Nástroj, který umožňuje spuštění webových aplikací jako nativní. Takto vytvořené aplikace jsou také nazývány jako hybridní. Na rozdíl od React Native se kód tvořený HTML, CSS a JavaScriptem nepřekládá. Namísto toho se uvnitř přidane abstraktní vrstvy otevírá takzvané web view. Web view je v podstatě vestavěný webový prohlížeč, který se stará o běh a zobrazení kódu, lze vidět na Obrázku2 !reference!.



Obrázek 2: [5] Architektura hybridních aplikací

Přístup k nativní API je poté zařízen přes pluginy volané z JavaScriptu, které propojují web view s nativní obálkou. Zde už je zpomalení poněkud výraznější a proto se tento způsob doporučuje pouze pro menší až středně velké projekty s nepříliš složitým uživatelským rozhraním, které by potenciálně mohlo na pozadí generovat velké množství DOM elementů. Možnost z kterékoliv webové aplikace vytvořit nativní aplikaci je hlavní výhodou, ale tvoří problém s nekonzistentním designem uživatelského rozhraní a nepřipraveností webu pro běh na mobilním zařízení. Z tohoto důvodu je dobré používat tento nástroj v kombinaci s některým z frameworků, které jsou k tomu uzpůsobeny.

## 2.3 Progresivní webové aplikace(PWA)

Jsou webové aplikace plně využívající možností moderních webových prohlížečů. Posouvají tím možnosti jejich funkcionality a pocitu z užívání na úroveň blízkou se nativním a multiplatformním aplikacím. Mohou být ale řazeny mezi ostatní mobilní aplikace? Pokud jako definující vlastnosti mobilních aplikací označíme přístup k periferiím zařízení a systémové API, spuštění i bez přístupu k internetu a nutnosti zapnutí webového prohlížeče, provádění procesů na pozadí, potom ano.

Prvním stavebním blokem této technologie jsou takzvané service workery. Service worker je skript, který běží na pozadí odděleně od webové stránky a umožňuje tak funkcionalitu, která nepotřebuje webovou stránku nebo interakci

uživatelé. [6] (*volný překlad*) Řeší tedy procesy na pozadí a zároveň ukládají kód stažený z webového serveru do aplikační cache a tím umožňují spustit aplikaci bez přístupu k internetu. Na zařízeních používající iOS jsou některé možnosti těchto skriptů omezeny, ale nejedná se o klíčovou funkcionalitu.

Druhou důležitou součástí jsou webové aplikační manifesty. Je to soubor ve formátu JSON, který určuje jak se bude aplikace chovat a vypadat po "instalaci" na zařízení. [7] (*volný překlad*) Instalaci se v tomto ohledu myslí přidáním ikony na domovskou stránku zařízení. Po spuštění přes tuto ikonu se sice spustí výchozí webový prohlížeč a v něm teprve stránka, ale o tomto procesu uživatel vůbec neví. Prohlížeč totiž nemá žádný ze svých běžných nástrojových lišt a aplikace je otevřena přes celou obrazovku zařízení.

Přístup k periferiím zařízení je potom zařízen různou kombinací možností prohlížeče, HTML5 a JS. Zde je ovšem oproti nativním a multiplatformním aplikacím výběr omezenější.

## 2.4 Backend pro mobilní aplikace

Většina aplikací se neobejde bez serverové části. Buďto z důvodu získání dat nebo komunikace s jinými instancemi aplikace. Server by měl vykonávat většinu výpočetních operací a posílat pouze nezbytně nutná data, aby méně zatěžoval mobilní zařízení. Jako nejlepší postup se tedy může zdát vytvoření vlastní API přímo na míru konkrétní aplikace. Toto řešení zvyšuje výdaje za vývoj i údržbu systému. Zbytečným se stává obzvláště v situacích, kdy by měl server provádět pouze jednoduché operace jako ověření identity uživatele. Pro projekty této povahy je tedy vhodnější vytvářet aplikace bez nutnosti vlastního serveru za pomoci cloudových služeb. Tyto služby umožňují vytvoření databázové struktury a na tomto základě vystavení REST API, které může aplikace konzumovat.

## 3 Výběr technologií

Ještě před výběrem konkrétních technologií a programovacích jazyků jsem se zamýšlel nad způsobem jak výsledný produkt předat potencionálnímu zákazníkovi. Rozhodnutí nakonec bylo tento problém kompletně eliminovat a celý systém konteinerizovat pomocí nástroje Docker. Kromě výsledných nativních aplikací u kterých to postrádá smysl. Při výběru technologií jsem takto nebyl vázán na prostředí ve kterém systém poběží.

### 3.1 Docker

### 3.2 Klientské části

Základním a nejdůležitějším cílem bylo vytvořit aplikaci, která bude funkční na mobilních zařízeních s operačními systémy Android i iOS a zároveň ve webových prohlížečích. Protože jsem byl na vývoj sám a chtěl jsem vytvořit kvalitní aplikaci, nativní vývoj pro každou platformu s následným vytvořením webové verze nepřipadalo v úvahu. Webová aplikace s následným zabalením do nativní obálky je v dané situaci nejvíce vyhovujícím řešením. Jako nástroj pro převod jsem použil výše zmíněný Capacitor.

Posledním krokem bylo zvolit správný přístup k vývoji webové aplikace. Tradiční způsob tvoření webové stránky za použití HTML, CSS a JavaScriptu by bylo zdlouhavé a potencionálně nebezpečné. Aplikace by očividně obsahovala spoustu kódu v čistém JS, který nemá žádnou kontrolu datových typů, což ve větších projektech může způsobovat chyby, které není lehké odhalit. Další problém by nastal se samotným uživatelským rozhraním. Veškeré UI komponenty by museli být vytvořeny ručně a nebyla by zaručena konzistence se zásadami stylů platformy. Upravovat styl stránky pomocí CSS, tak aby fungovala na zařízeních všech velikostí je sice možné, ale zbytečně náročné. Z těchto důvodů jsem se rozhodl pro použití frameworku Angular a platformu Ionic, která je navíc výborně kompatibilní s pluginy Capacitoru.

#### 3.2.1 Angular

Angular je platforma a framework pro vývoj jednostránkových klientských aplikací za použití HTML a TypeScriptu. Samotný Angular je napsaný v TypeScriptu. Implementuje základní a volitelné funkce jako množinu TypeScriptových knihoven, které je možné importovat do aplikací. [8] (*volný překlad*)

Jednostránkové klientské aplikace fungují tak, že při prvním přístupu na stránku se ke klientovi stáhnou všechny soubory potřebné k běhu aplikace. Při jakékoliv akci uživatele se stránkou už se nikdy nepřistupuje na webový server. Tento mechanismus značně ulehčuje service workeru práci s cachováním a následným spouštěním aplikace v offline režimu, v případě, že by bylo potřeba z aplikace vytvořit PWA.

TypeScript je nadmnožina skriptovacího jazyku JavaScript a přidává do něj možnost typování proměnných, parametrů a návratových hodnot funkcí. Je to objektově orientovaný jazyk podporující funkcionální paradigma s prvky asynchronního programování. Webové prohlížeče ovšem tomuto jazyku nerozumí a proto je potřeba provést proces zvaný transpilace, při kterém se všechny soubory projektu přeloží do souborů s příponou .js, které se navzájem referencují a index.html souboru, který je kořenem tohoto stromu závislostí.

Základními stavebními bloky jsou komponenty a služby. Komponenty se skládají z náhledů a kódu na pozadí. Náhled je HTML kód doplněný o takzvané directives, které umožňují například psát cykly a podmínky jako atributy HTML tagů. Kód na pozadí je vždy třída, obsahující metody a atributy, které je možné referencovat v náhledu. Služby obsahují funkcionalitu, která přímo nesouvisí s náhledy a je možné je do komponent "injectovat" jako závislosti. [8] Framework tedy automaticky nabízí podporu návrhového vzoru Dependency injection a všechny služby jsou typu Singleton. Architektura založená na používání komponent velice jednoduše umožňuje znovu použitelnost kódu.

### 3.2.2 Ionic

Platforma, která ulehčuje vývoj webových aplikací cílených na mobilní zařízení. Poskytuje vývojářům množství komponent, které se vzhledově přizpůsobují dle platformy na které běží. Projekty vytvořené pomocí nástrojů, které platforma nabízí jsou plně responzivní. Ionic rozšiřuje základní knihovnu Angularu o nové stavy životních cyklů komponent a navigační strategii, která umožňuje uchovávat si navigační zásobník instancí naposledy otevřených náhledů. Dohromady je díky nim možné implementovat chování navigačních tabů na které jsou uživatelé nativních aplikací zvyklí.

## 3.3 Server

Chtěl jsem mít plnou kontrolu nad tím jak serverová část zpracovává data a v jakém formátu je posílá na klienta. Vzhledem k zvolené technologii pro aplikaci bylo lepší co nejvíce zodpovědnosti za operace s daty předat serveru a odlehčit tak zařízením na kterých poběží. Proto jsem se rozhodl psát kód serveru sám a přímo na míru místo použití cloudových řešení jako Firebase.

Základem Angularu zvoleného pro tvorbu klientských částí je JavaScript. JS umí automaticky vytvářet objekty a kolekce z JSON souborů. Angular má navíc v základní knihovně velice dobře zpracovanou podporu pro vytváření HTTP požadavků. Jasnou volbou tedy bylo REST API, které funguje na základě protokolu HTTP a standardním formátem pro přenos dat je právě JSON. S přihlédnutím na budoucí rozšíření bylo žádoucí, aby vybraná technologie umožňovala případné rozšíření o gRPC služby založené na protokolu HTTP/2 kvůli plně duplexní komunikaci.

Dalšími faktory při výběru byly: dobrý vestavěný framework pro práci s databází, podpora pro ověřování identity uživatele pomocí JSON web tokenů, pod-

pora nástroje Swagger pro dokumentaci, nezávislost na operačním systému a ideálně i vestavěná podpora pro Docker. Konečným výběrem tedy byl ASP .NET Core.

### **3.3.1 .NET Core**

.NET Core je open source framework pro operační systémy Windows, Linux a macOS. Je to multiplatformní nástupce .NET Frameworku. Projekt je primárně vyvíjen firmou Microsoft a vydáván pod MIT Licencí. [9] (*volný překlad*) Při práci s .NET Core jsou používány jazyky C#, F# a VisualBasic.

ASP .NET Core slouží k vývoji webových aplikací a služeb s tímto vývojem spojených.

## 4 Server

### 4.1 Databázový modul

Prvním krokem při vývoji serverové části bylo vytvoření modulu pro práci s databází. Stavebním blokem pro tento modul je Entity Framework Core. Zároveň jsem do tohoto modulu přidal nástroj, který generuje testovací data, který je možno explicitně spustit ve formě konzolové aplikace.

#### 4.1.1 Datové třídy

U tvorby tříd odrážejících strukturu datového modelu (ref obr. 1), bylo potřeba zvolit vhodné datové typy, vyřešit jak simulovat relace mezi jednotlivými tabulkami a rozdíl mezi povinnými a nepovinnými parametry:

- Název tabulky určuje anotace (ref kod.1) na řádku 1.
- Jako identifikátor jsem zvolil datový typ Guid, který umožňuje přímo identifikovat určitý záznam v databázi.
- Atributy s hodnotovými datovými typy jsou označeny jako nullable pokud nejsou povinné.
- K atributům s referenčními datovými typy je přidána anotace Required pokud jsou povinné, (ref kod. 1) řádky 4 a 7.
- V případě relace 1:N. Jsou relace simulovány kombinací atributu pro uložení cizího klíče a referencí na instanci třídy reprezentující jinou tabulku neboli navigační atribut pro stranu relace 1, (ref kod. 1) řádky 10 a 11. Pro stranu N bude navigační atribut kolekce těchto typů.

```
1 [Table("User")]
2 public class User : EntityBase
3 {
4     [Required]
5     public string LoginName { get; set; }
6
7     [Required]
8     public string Password { get; set; }
9
10    public Guid EmployeeId { get; set; }
11
12    public virtual Employee Employee { get; set; }
13 }
```

Zdrojový kód 1: Kód reprezentující entitu.



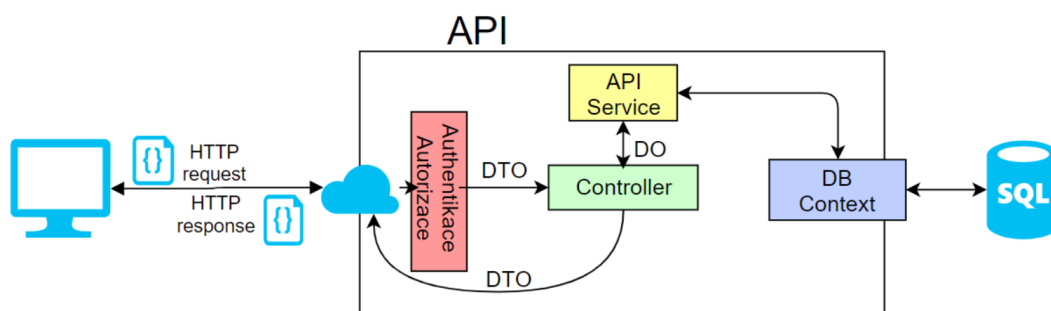
### 4.1.2 Db Context

Datové třídy jsou spojeny ve třídě DbContext jejíž instance reprezentuje spojení s databází. Na úrovni této třídy je zároveň řešeno jaký databázový poskytovatel bude použit. Dle toho je potřeba integrovat správnou knihovnu. Tato integrace je naštěstí triviální a je možné připojení k serverům s jiným poskytovatelem libovolně měnit s minimem úsilí. Právě tato třída umožňuje ostatním modulům číst a zapisovat data do databáze. Zodpovědností modulu přistupujícího k datům je předat contextu connection string, díky kterému je možné spojení navázat. V celém projektu je pro práci s daty použit dotazovací jazyk LINQ s rozšíření, které umožňuje dotazování za pomoci lambda výrazů.

Balíček EFCore Tool umožňuje za použití příkazové řádky vytvářet Migrace. Migrace je automaticky vygenerovaný kód lišící se dle momentálně použitého databázového poskytovatele popisující strukturu databáze danou DbContextem. Tento kód poté může být použit při volání metody Migrate, která do aktuálně připojené instance databáze vytvoří onu strukturu. Tento přístup se nazývá Code first.

Pro načítání hodnot do navigačních atributů do objektů představujících záznamy v databázi je použita metoda Lazy loading. Na rozdíl od ostatních možných přístupů zaručuje, že data budou načtena vždy až to opravdu bude nutné a současně o to není potřeba explicitně žádat. Toho efektu jsem docílil označením všech klíčových atributů jako virtuálních, což umožňuje aby se na jejich místě vytvořili proxy objekty.

## 4.2 Struktura API



Obrázek 3: Diagram struktury API

### 4.2.1 API controllery

Pomocí controlleru se definuje celá struktura API. Obsahují metody, které jsou volány pomocí HTTP requestů.

Proces životního cyklu z pohledu API začíná ve chvíli, kdy na nějaký z endpointů přijde požadavek. Který controller by ho měl zpracovat záleží na použité směrovací strategii a URI na kterou byl zaslán. S použitím výchozího směrování,

```

1  [Route("api/[controller]")]
2  [ApiController]
3  [Authorize]
4  public class EmployeeController : DssBaseController
5  {
6      private readonly IEmployeeApiService _employeeApiService;
7
8      public EmployeeController(IEmployeeApiService
9          employeeApiService)
10     {
11         _employeeApiService = employeeApiService;
12     }
13     [HttpPost]
14     [Authorize(Roles = "Manager")]
15     public EmployeeDetailDTO CreateEmployee(EmployeeDetailDTO dto)
16     {
17         var employee = MapDetailDtoToEmployee(dto);
18         var createdEmployee = _employeeApiService.CreateEmployee(
19             employee);
20         var createdDto = MapEmployeeToDetailDto(createdEmployee);
21
22         return createdDto;
23     }
24     ...

```

Zdrojový kód 2: Ukázka kódu controlleru

jako jsem použil já a definice cesty v anotaci (ref. code 2) řádek 1 by tento controller obsluhoval všechny požadavky na adrese: *[základní\_adresa]/employee/...* Jestli je potřeba, aby byl požadavek opatřen platným tokenem určuje anotace `Authorize`. (ref. code 2) řádkem 3 je tedy dáno, že všechny požadavky, které tímto controllerem budou obslouženy musí být takto zabezpečeny.

HTTP metod je dohromady 8, ale v projektu jsou použity pouze 4 z nich a při jejich použití jsem se držet konvencí:

- **GET** - je používána pro získání informací ze serveru. Měla by být používána pouze pro získání dat a neměla by na data mít žádný jiný efekt. [10] (*volný překlad*)
- **POST** - používá se pro zaslání dat na server.[10] (*volný překlad*)
- **PUT** - Nahrazuje všechny aktuální reprezentace daného zdroje nahrávaným obsahem.[10] (*volný překlad*)
- **DELETE** - Smaže všechny aktuální výskyty zdroje uvedeného v URI.[10] (*volný překlad*)

Metoda *CreateEmployee* bude díky anotaci (ref. code 2) na řádku 13 data ukládat a reagovat pouze na POST requesty.

Posledním aspektem, který je potřeba zmínit je rozdělení rolí. V systému jsou role dvě: pracovník pečovatelské služby a vedoucí pracovník. Každá role má jiná práva a tím pádem může přistupovat k různým endpointům a získávat tak odlišná data. Pracovník pečovatelské služby je základní role s minimálními oprávněními pro vstup do systému, proto metody konzumovány tímto typem účtů není potřeba explicitně kontrolovat. Při přihlášení se do tokenu ukládá informace(claim - viz ref sekce zabezpečení) o této roli a ta je kontrolována, pokud je metoda označena stejně jako (ref. code 2) na řádce 14.

Framework automaticky serializuje a deserializuje objekty z a do formátu JSON. Pro vývojáře je tento proces naprosto transparentní a může rovnou pracovat s objekty jak jsou definovány v kódu. Z důvodu bezpečnosti, znovupoužitelnosti a možné budoucí rozšiřitelnosti kódu jsem se pro práci s objekty a daty rozhodl použít strategii řídicí se následujícími pravidly:

- Návrátový typ metody controlleru je vždy DTO obsahující jen ty nejnutnější údaje.
- DTO objekty existují pouze v rozsahu controlleru(předchází problému kruhových závislostí).

Lze pozorovat, že metoda (ref. code 2) na řádcích 15-22 se tohoto principu drží. Na (ref. obr.2) je toto demonstrováno mezi controllerem a API službou mezi kterými se posílají pouze Domain Objects.

#### 4.2.2 API služby

Všechny operace s databází jsou striktně prováděny pouze v této skupině tříd. Správnost databázových dotazů je totiž jediným místem, kde může dojít k sémantickým chybám. Tyto třídy proto implementují rozhraní obsahující všechny jejich veřejné metody, což činí případné Mockování v následném testování možným. Navíc díky dodržování pravidel o DTO mohou být kdykoliv v budoucnu využity znovu v jiném modulu.

#### 4.2.3 DI a propagace konfigurace

Framework má zabudovanou automatickou podporu pro návrhový vzor Dependency Injection(DI). Toho jsem využil pro distribuci DbContextu do API služeb a tech zase do controlleru. Výsledek procesu je vidět na (ref. code 2) řádcích 6-11. Controller závisí na *EmployeeApiService*, která je reprezentována svým rozhraním a DI kontejner při volání tohoto konstrukturu automaticky poskytne její instanci jako parametr.

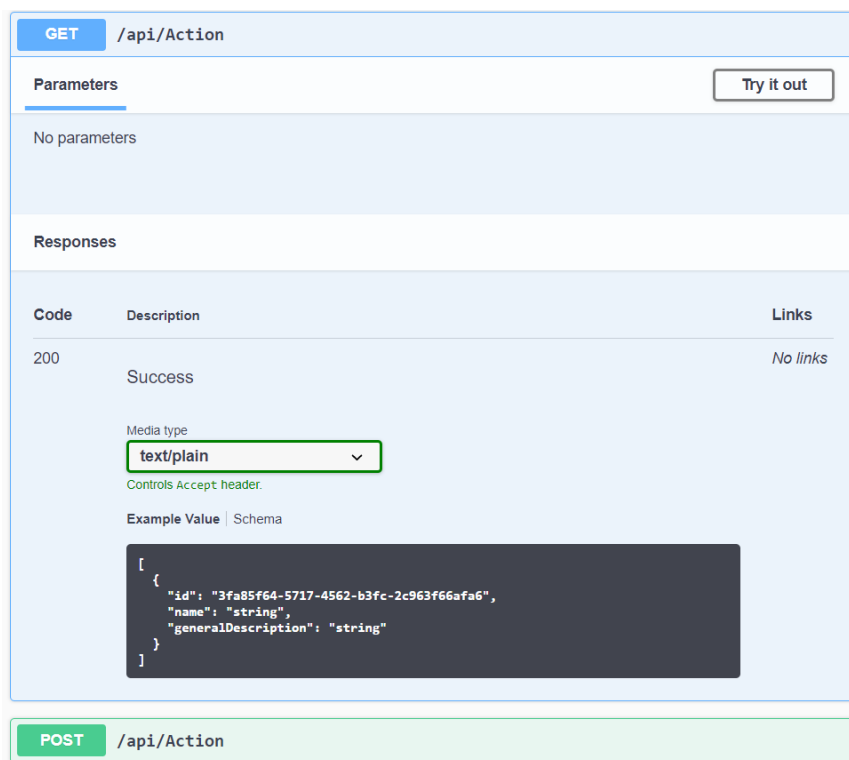
ASP .NET Core má komplexní systém možností konfigurace, která zvládne konzumovat konfiguraci z mnoha různých zdrojů najednou, upřednostnit parametry s větší prioritou a následně je sloučit. Tohoto je využito například při nastavování proměnných prostředí při spouštění Docker kontejneru.

#### 4.2.4 Zabezpečení

Všechny požadavky musí v hlavičce obsahovat platný token, jinak je API odmítne obsloužit. Tyto Json Web Tokeny vystavuje jediný nezabezpečený endpoint, který přímá jako parametry jméno a heslo uživatele. Hesla jsou v databázi uložena v hashované podobě, proto je nutno udělat to stejné a převést heslo pomocí hashovací funkce MD5. Pokud ověření proběhne v pořádku je vytvořen nový token, který je podepsán soukromým klíčem a jsou do něj zakódovány informace jako jméno uživatele, kdo je konzument a vydavatel tokenu a role. Tyto informace se nazývají claimy a je možné je z tokenu pomocí určitých nástrojů vyčíst. Jediná informace, která zaručuje bezpečnost je právě soukromý klíč.

#### 4.2.5 Swagger a automatická dokumentace

Pro účely dokumentace API jsem použil nástroj Swagger, který automaticky zpracuje její strukturu a poskytne jí v podobě uživatelského rozhraní a JSON souboru. V produkci je žádoucí tuto funkci vypnout nebo zabezpečit ověřením identity.



Obrázek 4: Ukázka automatické dokumentace pomocí Swaggeru

### 4.3 Testování částí

Testování částí je úroveň testování softwaru, kde jsou testovány individuální komponenty nebo části. Smyslem je ověřit, zdali se každá část chová jak byla navr-

žena. Část je nejmenší testovatelná jednotka jakéhokoliv softwaru.[11] (*volný překlad*)

Testování částí jsem v systému použil pro testování metod API služeb, které provádí složitější databázové dotazy a operace nad těmito daty. Všechny tyto metody používají DbContext pro přístup do databáze. Tento přístup bylo potřeba nahradit testovací instancí databáze, která bude na začátku každého testu obsahovat prázdnou strukturu tabulek a po jeho skončení přestane existovat. Pro dosažení tohoto efektu byl DbContextu konfigurován pro používání databáze v paměti.

Každý test se skládá z těchto stavů:

- **Arrange** - připraví strukturu dat, tak aby testovala přesnou funkcionalitu testované části.
- **Act** - spustí kód testované části.
- **Assert** - porovnává vrácené výsledky s očekávanými výsledky, pokud se liší vyvolá chybu.

## **5 Klientské části**

### **5.1 Multiplatformní aplikace**

### **5.2 Managerská webová aplikace**

## **6 Nasazení**

### **6.1 Kontejnerizace**

### **6.2 Azure**

### **6.3 CI/CD**

## Bibliografie

- [1] <https://developer.android.com/guide/components/fundamentals>
- [2] [https://en.wikipedia.org/wiki/IOS\\_SDK](https://en.wikipedia.org/wiki/IOS_SDK)[https : //reactnative.dev/docs/intro – react – native – components](https://reactnative.dev/docs/intro-react-native-components)
- [3] <https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin>
- [4] <https://ionicframework.com/resources/articles/what-is-hybrid-app-development>
- [5] <https://developers.google.com/web/fundamentals/primers/service-workers>[https : //web.dev/add – manifest/https : //angular.io/guide/architecture](https://web.dev/add-manifest/)
- [6] [https://en.wikipedia.org/wiki/.NET\\_Core](https://en.wikipedia.org/wiki/.NET_Core)[https : //www.tutorialspoint.com/http/http\\_methods.htm](https://www.tutorialspoint.com/http/http_methods.htm)
- [7] <http://softwaretestingfundamentals.com/unit-testing/>



## 7 Obsah příloženého CD/DVD

Na samotném konci textu práce je uveden stručný popis obsahu příloženého CD/DVD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

### **bin/**

Instalátor `INSTALATOR` programu, popř. program `PROGRAM`, spustitelné přímo z CD/DVD. / Kompletní adresářová struktura webové aplikace `WEBOVKA` (v ZIP archivu) pro zkopírování na webový server. Adresář obsahuje i všechny runtime knihovny a další soubory potřebné pro bezproblémový běh instalátoru a programu z CD/DVD / pro bezproblémový provoz webové aplikace na webovém serveru.

### **doc/**

Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

### **src/**

Kompletní zdrojové texty programu `PROGRAM` / webové aplikace `WEBOVKA` se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu / adresářové struktury pro zkopírování na webový server.

### **readme.txt**

Instrukce pro instalaci a spuštění programu `PROGRAM`, včetně všech požadavků pro jeho bezproblémový provoz. / Instrukce pro nasazení webové aplikace `WEBOVKA` na webový server, včetně všech požadavků pro její bezproblémový provoz, a webová adresa, na které je aplikace nasazena pro účel testování při tvorbě posudků práce a pro účel obhajoby práce.

Navíc CD/DVD obsahuje:

### **data/**

Ukázková a testovací data použitá v práci a pro potřeby testování práce při tvorbě posudků a obhajoby práce.

### **install/**

Instalátory aplikací, runtime knihoven a jiných souborů potřebných pro provoz programu `PROGRAM` / webové aplikace `WEBOVKA`, které nejsou standardní součástí operačního systému určeného pro běh programu / provoz webové aplikace.

### **literature/**

Vybrané položky bibliografie, příp. jiná užitečná literatura vztahující se k práci.

U veškerých cizích převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovoluují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy na CD/DVD, je uveden jejich zdroj (např. webová adresa) v bibliografii nebo textu práce nebo v souboru `readme.txt`.

