Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Mobilní aplikace pro pracovníky pečovatelské služby

2020

Petr Janiš

Vedoucí práce: Mgr. Janoštík Radek

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. Janoštík Radek
Počet stran:	19
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic 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. Janoštík 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 LaTeXu. Zdrojový text v LaTeXu 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 LaTeX style for the text. The source text in 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

Seznam obrázků

Seznam tabulek

Seznam vět

Seznam zdrojových kódů

1 Úvod

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. 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 jejich úspěšnosti na trhu. Co se týče mobilního softwaru vyvíjeného za účelem korporátního použití, je to otázka finanční náročnosti 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 jakým účelem bude používána. Tato specifikace společně s platformou, na kterou by měl být vývoj cílen, jsou rozhodujícími kriterii 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 pravě 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

Při této metodě je aplikace tvořena odděleně pro každou platformu zvlášť. Stejný kód je tedy potřeba napsat dvakrát v různých programovacích jazycích, 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 jim 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 platforem. 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 bezpečností aspekty za vývojáře, do určité míry, je 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 používaní knihoven a frameworků třetích stran.
- Neklade se důraz na rychlost vydaní 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 vývojáři 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 oficialně 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, určovaní 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 množství rychle vstupujících dat. Celkově tato metoda v porovnaní 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 platforem. I přesto, že je tímto způsobem možné vyvíjet aplikaci na obě platformy za použití počítače s jakýmkoliv operační 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 pomocí kterého se přistupuje k API konkretní platformy. Zároveň popisuje vzhled a chování uživatelského rozhraní za použití komponent Reactu. [**3**] *(volný překlad)*

Důležitou vlastností je to, že kód je kompilován do nativního jazyku používaného 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 pomoci něj vytvořeny aplikace jako Facebook a Instagram, které denně používají milióny uživatelů.

2.2.2 Xamarin

Xamarin je opět platforma, která umožňuje multiplatformní vývoj, tentokrát ale s rozdílem, že společně s mobilní 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 volaní systémového API velice podobná nativnímu vývoji, pouze za použití jiného jazyka.

2.2.3 Capacitor

Nástroj, který umožňuje spuštění webových aplikací jako nativních. 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 přidané 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 1: [**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 potencionálně mohlo na pozadí generovat veliké množství DOM elementů. Možnost vytvořit, z kterékoliv webové aplikace 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 užívaní na úroveň blížící 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

uživatele. [**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)* Instalací se v tomto ohledu myslí přidání ikony na domovskou obrazovku zařízení. Po spuštění přes tuto ikonu se sice otevře výchozí webový prohlížeč a v něm teprve stránka, ale o tomto procesu uživatel vůbec neví. Prohlížeč totiž nemá žádnou ze svých běžných nástrojových lišt a aplikace je otevřena přes celou obrazovku zařízení.

Přístup k periferiím zařízení je potom umožněn různou kombinací funkcí 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. Buď z důvodu získání dat nebo komunikace s jinými instancemi aplikace. Server by měl vykonávat většinu výpočetních operací a posílat pouze nezbytně nutná data, aby co nejméně 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 použití 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 konkretních technologií a programovacích jazyků jsem se zamýšlel nad způsobem, jak výsledný produkt předat potencionálnímu zákazníkovi. Rozhodnutí nakonec bylo, tento problém kompletně eliminovat, celý systém kontejnerizovat pomocí nástroje Docker, kromě výsledných nativních aplikací, u kterých to postrádá smysl. Při výběru technologií jsem tak nebyl vázán na prostředí, ve kterém systém poběží.

3.1 Docker

TODO

3.2 Klientské části

Základním a nejdůležitějším cílem bylo vytvořit aplikaci, která bude funkční v 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 nepřipadal 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 použít framework 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 aplikace. [**8**] *(volný překlad)*

Jednostránkové klientské aplikace fungují tak, že při prvním přístupu na stránku se do klientova zařízení stáhnou všechny soubory potřebné k běhu aplikace. Při jakékoliv další akci uživatele se stránkou už se nikdy nepřistupuje na webový server. Tento mechanizmus značně ulehčuje service workeru práci s ca-

chovaním 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 asynchroního programování. Webové prohlížeče ovšem tento jazyk neumí zpracovat a proto je potřeba provést transpilaci. Transpilace je proces, při kterém se všechny soubory projektu přeloží do souborů s příponou .js, které se navzájem referencují. A dále do 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 directivy, 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 Dependecy injection a všechny služby jsou typu Singleton. Architektura založená na používaní komponent usnadňuje znovupoužitelnost 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

Vzhledem ke zvolené technologii pro aplikaci bylo lepší, co nejvíce zodpovědnosti za operace s daty předat serveru a odlehčit tak zařízením, na kterých poběží. Zároveň jsem chtěl mít plnou kontrolu, nad tím jak serverová část zpracovává data a v jakém formátu je posílá klientovi. Proto jsem se rozhodl kód serveru psát 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 přihlédnutí 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, podpora 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 Corem 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 a 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. Pro stranu 1 je simulována kombinací atributu pro uložení cizího klíče a referencí na instanci třídy reprezentující jinou tabulku(navigační atribut), (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

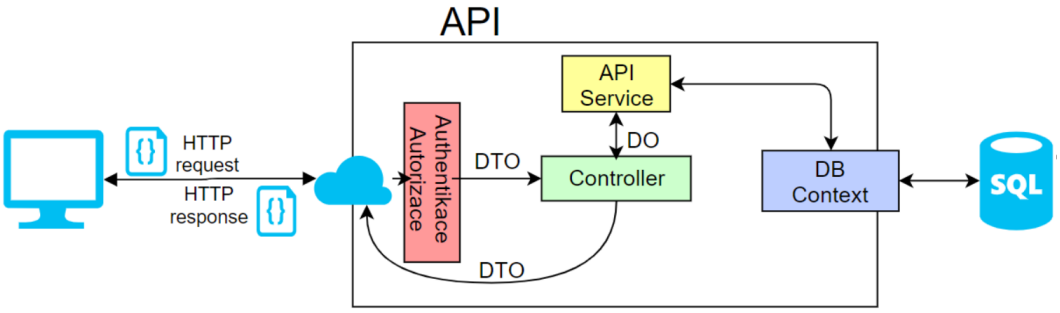
Datové třídy jsou spojeny ve třídě DbContext, jejíž instance reprezentuje spojení s databází. Na úrovni této třídy je zároveň řešeno, jaký databázový poskytovatel bude použit. Dle toho je potřeba integrovat správnou knihovnu. Připojení k serverům s jiným poskytovatelem je možné libovolně měnit s minimem 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 rozšířením, které umožňuje dotazování za pomocí lamba 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 pro tvorbu oné struktury do aktuálně připojené instance databáze. Tento přístup se nazývá Code first.

Pro načítaní hodnot do navigačních atributů objektů představující 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 na jejich místě vytvořili proxy objekty.

4.2 Struktura API

TODO úvod



Obrázek 2: Diagram struktury API

4.2.1 API controllery

Pomocí controllerů se definuje celá struktura API. Obsahují metody, které jsou volány pomoci HTTP požadavků.

Proces životního cyklu požadavku z pohledu API začíná ve chvíli, kdy je přijat nějakým z endpointů. To jaký controller by ho měl zpracovat, záleží na použité směrovací strategii a URI, na kterou byl zaslán. S použitím výchozího

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

Zdrojový kód 2: Ukázka kódu controlleru

směrování, jako jsem použil já, a definice cesty v anotaci (ref. code 2) řádek 1 by controller v ukázce (ref. code 2) 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 řádku 14.

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

- Návratový 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 conrollerem 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 dodržováni 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 Dependecy Injection(DI). Toho jsem využil pro distribuci DbContextu do API služeb a ty zase do controllerů. Výsledek procesu je vidět na (ref. code 2) řádcích 6-11. Controller závisí na *EmployeeApiService*, která je reprezentována svým rozhraním. Při volání tohoto konstruktoru DI kontejner automaticky poskytne její instanci jako parametr.

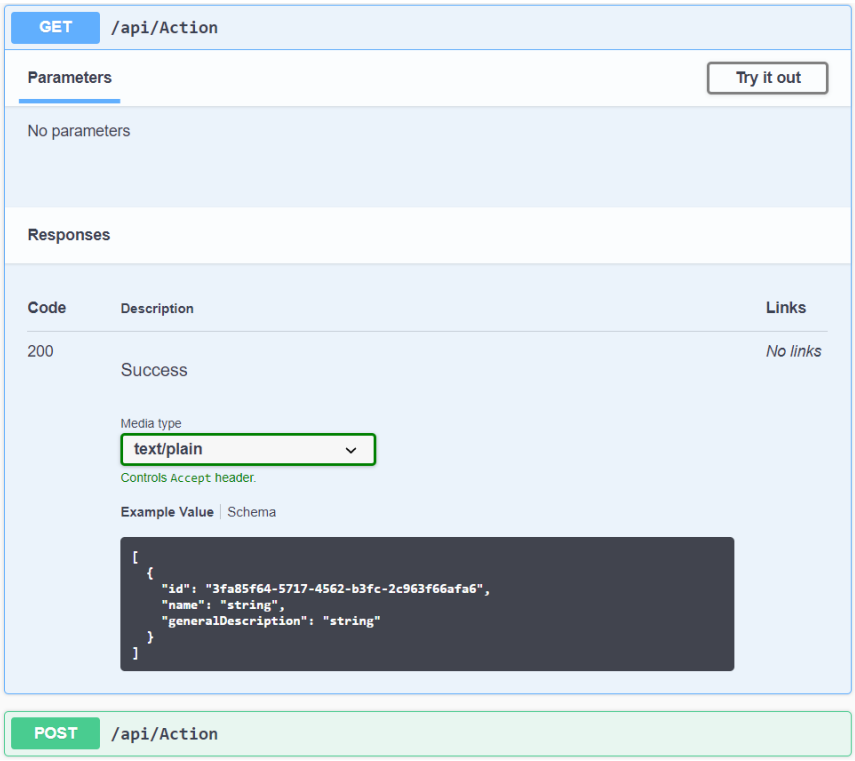
ASP .NET Core má komplexní systém možností konfigurace, který zvládne konzumovat konfiguraci z mnoha různých zdrojů najednou, upřednostnit parametry s větší prioritou a následně je sloučit. Tohoto je využito například při nastavování proměnných prostředí při spouštění Docker kontejneru.

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 přijímá 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, datum a čas expirace, kdo je konzument, vydavatel tokenu a role. Tyto informace se nazývají claimy a je možné je z tokenu kdykoliv vyčíst. Jediná informace, která zaručuje bezpečnost, je právě soukromý klíč, kterým se ověřuje, že přijatý token byl vystaven právě touto API.

4.2.5 Swagger a automatická dokumentace

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



Obrázek 3: Ukázka grafické dokumentace pomocí nástroje Swagger

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

Testoval jsem metody 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 bude na začátku každého testu obsahovat prázdnou strukturu tabulek a po jeho skončení přestane existovat. Pro dosažení tohoto efektu byl DbContext 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

TODO úvod

5.1 Multiplatformní aplikace

?úvod?

5.1.1 Komunikace se serverem

Díky integraci automatické dokumentace v serverové části jsem získal popis struktury API jako soubor ve formátu JSON. V kombinaci s nástrojem Swagger codegen jsem soubor použil pro vygenerování celého modulu pro komunikaci se serverem. Šablonu, pomocí které se jednotlivé služby generují, bylo potřeba upravit, aby k jednotlivým dotazům do hlavičky přidával token z lokálního úložiště a ty byly tak z pohledu serveru autorizované. Pokud požadavek skončil chybou, je o tom uživatel patřičně informován v podobě notifikace nebo návratem na přihlašovací obrazovku dle povahy chyby.

Problém představoval převod časových údajů z formátu ISO 8601 do datového typu datum v JavaScriptu. ISO 8601 je standardem pro reprezentaci časových údajů ve formátu JSON, který je používán pro přenos dat mezi klientem a serverem. Tento formát na rozdíl od datového typu není závislí na časové zóně. Z tohoto důvodu byl přidán HTTP interceptor, který projde data každého požadavku a odpovědi a upraví v nich data, tak aby jejich hodnota byla korektní dle časové zóny klienta, což ve výsledku ovlivňuje i správnost záznamů v databázi.

5.1.2 Klientská cache

HTTP požadavek na API může být časově náročná operace. Proto jsem se rozhodl, při zapínání aplikace načíst a zpracovat nejčastěji používaná data a následně se na ně na všech stránkách pouze odkazovat. Konkrétně jde o data klientů.

Při tomto načítání se kontroluje, v jakém stavu jsou údaje o klientově adrese. Systém totiž umožňuje uložení bydliště, buď pouze jako souřadnice zeměpisné šířky a délky nebo jako objekt popisující adresu v řetězcích. V různých místech aplikace jsou potřeba oba údaje, proto je případné chybějící udaje třeba doplnit. Za tímto účelem je použita Google Maps Geocoding API.

Zde se poprvé zjišťuje i poloha zařízení. Ve webových prohlížečích je použita standardní HTML Geolokační API a v případě nativní verze geolokační plugin, který čte data z GPS telefonu a internetového připojení pro větší přesnost. Z tohoto důvodu je uživatel při prvním spuštění požádán o oprávnění k určování lokace zařízení.

V objektu obsahujícím data o klientovi je současně uložena jeho aktuální vzdálenost jeho bydliště od zařízení a výpočet této informace je také součástí procesu načítaní. Přišlo mi zbytečné pro tuto operaci opět používat Google API.

V tuto chvíli už je zaručeno, že jsou údaje o zeměpisných souřadnicích adresy a zařízení přítomny a je tak možné použít výpočet pomocí Haversínovy rovnice. Haversínova rovnice (ref. ronice 1) určuje vzdálenost mezi dvěma body ležícími na povrchu koule zadanými zeměpisnou šířkou a délkou.[**12**] *(volný překlad)* TODO rovnice

$$\left\{ \frac{x^2}{y^3} \right\}$$

Tento výpočet se spouští každých 60 vteřin a aktualizuje, tak informace o vzdálenosti v celé aplikaci.

5.2 Managerská webová aplikace

Bibliografie

[1] https://developer.android.com/guide/components/fundamentals

[2] https://en.wikipedia.org/wiki/IOS$_{S}DK$

[3] https://reactnative.dev/docs/intro-react-native-components

[4] https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin

[5] https://ionicframework.com/resources/articles/what-is-hybrid-app-development

[6] https://developers.google.com/web/fundamentals/primers/service-workerswhat$_{i}s_{as}ervice_{w}orker$

[7] https://web.dev/add-manifest/

[8] https://angular.io/guide/architecture

[9] https://en.wikipedia.org/wiki/.NET$_{C}ore$

[10] https://www.tutorialspoint.com/http/http$_{m}ethods.htm$

[11] http://softwaretestingfundamentals.com/unit-testing/

[12] https://en.wikipedia.org/wiki/Haversine$_{f}ormula$

6 Obsah přiloženého CD/DVD