VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

INFORMAČNÍ SYSTÉM PRO LÉKAŘE S MOBILNÍ APLIKACÍ PRO PACIENTY

INFORMATION SYSTEM FOR A DOCTOR WITH MOBILE APPLICATION FOR PATIENTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DAVID VLASÁK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. VLADIMÍR BARTÍK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Vlasák David**
Program: Informační technologie
Název: **Informační systém pro lékaře s mobilní aplikací pro pacienty**
Information System for a Doctor with a Mobile Application for Patients
Kategorie: Informační systémy

Zadání:

1. Seznamte se s problematikou tvorby webových a mobilních aplikací a různými vývojovými prostředími.
2. Prostudujte problematiku optického rozpoznávání znaků (OCR) a vyhledejte použitelná existující řešení.
3. Analyzujte požadavky a navrhněte informační systém pro evidenci objednávek u lékaře spolu s mobilní aplikací pro pacienty, která bude umožňovat evidenci pacientů včetně jejich objednávání. Vyšetření se budou ukládat jako události do Google kalendáře. Pacient se bude moci registrovat pomocí OCR načtením kartičky pojištěnce.
4. Navržený informační systém implementujte a otestujte jeho funkčnost na vhodném vzorku dat a uživatelů.
5. Zhodnoťte dosažené výsledky a další možnosti pokračování tohoto projektu.

Literatura:

- MacDonald, M., Freeman, A., Szpuszta, M.: ASP.NET 4 a C# 2010: tvorba dynamických stránek profesionálně. Brno: Zoner Press, 2011. ISBN 978-80-7413-131-8.
- Nagy, G., Nartker, T., Rice, S.: Optical character recognition: an illustrated guide to the frontier. Proceedings of SPIE - The International Society for Optical Engineering. p. 58-69, 1999.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1-3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Bartík Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 20. října 2021

Abstrakt

Tato práce se zabývá vývojem informačního systému pro lékaře a pacienty. Lékař využívá webovou aplikaci, která je implementována pomocí ASP.NET MVC. Pacient využívá aplikaci mobilní, která je napsána pomocí technologie Xamarin.Forms. Tato aplikace využívá pro registraci pacienta optické rozpoznávání znaků. Nejdůležitějšími funkcemi tohoto systému je správa pacientů, jejich objednávek a také možnost pacientů žádat o přeobjednání. Systém byl úspěšně navržen, implementován a otestován potenciálními uživateli.

Abstract

This thesis deals with the development of an information system for a doctor and patients. Doctor uses a web application, which is implemented using ASP.NET MVC. Patient uses a mobile application, which is written using Xamarin.Forms technology. This application uses optical character recognition for patient registration. The most important functions of the system is to manage patients, their orders and also for patients the possibility to request a reorder. The system was successfully designed, implemented and tested by potential users.

Klíčová slova

informační systém, mobilní aplikace, ASP.NET, Entity Framework, Xamarin.Forms, SQLite, OCR, Azure Cognitive Service, Google Calendar API

Keywords

information system, mobile application, ASP.NET, Entity Framework, Xamarin.Forms, SQLite, OCR, Azure Cognitive Service, Google Calendar API

Citace

VLASÁK, David. *Informační systém pro lékaře s mobilní aplikací pro pacienty.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vladimír Bartík, Ph.D.

Informační systém pro lékaře s mobilní aplikací pro pacienty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vladimíra Bartíka. Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Vlasák
2. května 2022

Poděkování

Tímto bych rád poděkoval panu Ing. Vladimíru Bartíkovi, Ph.D. za odborné vedení mé bakalářské práce, věcné připomínky a vstřícnost při konzultacích.

Obsah

Kapitola 1

Úvod

Každý z nás někdy zažil situaci, kdy k lékaři nedorazil včas. Případně jsme na vyšetření u lékaře zcela zapomněli. Lékař pak zbytečně vyčkává a v jeho nabitém kalendáři vznikají mezery, které již není schopen zaplnit. Pokud si pacient vyšetření někam nezaznamená, s největší pravděpodobností na něj zapomene. Proč by si ale měl datum vyšetření poznamenávat znovu, když jej lékař/sestra do systému zapíše? A nemohl by pacient sdělit informaci, že nepřijde, dříve? Tento problém řeší také vybraná oční klinika, pro kterou byla tato problematika řešena.

Hlavním cílem této práce je vytvořit informační systém pro lékaře a zároveň také pro pacienty. Lékař, případně sestra, bude využívat webovou aplikaci pro spravování objednávek jednotlivých pacientů. Pro pacienta bude vytvořena aplikace mobilní, do které provede registraci pomocí načtení své kartičky pojištěnce. V aplikaci bude moci zjistit svá vyšetření a případně požádat o libovolnou změnu dané objednávky (tím může také sdělit informaci, že nepřijde na objednané vyšetření). Aplikace také na blížící se objednání pacienta upozorní.

Obsah této práce je rozdělen do několika kapitol, které popisují vývoj od teoretických znalostí, přes analýzu a návrh systému, až po implementaci a závěrečné testování.

Kapitola 2 pojednává o principech tvorby webových a mobilních aplikací. Seznamuje čtenáře s aktuálně používanými technologiemi a nejoblíbenějšími vývojovými prostředími pro tvorbu webových aplikací. Následně popisuje způsoby vývoje mobilních aplikací s jednotlivými vhodnými technologiemi.

Další teoretické znalosti jsou uvedeny v kapitole 3. Ta řeší problematiku optického rozpoznávání znaků neboli OCR. Čtenář je seznámen s principem této technologie a následně jsou mu představeny již existující použitelná řešení.

Další kapitolou je kapitola 4, která se zabývá analýzou a specifikací požadavků. Specifikace jsou rozděleny zvlášť pro webovou a mobilní aplikaci. Čtenář se v této kapitole dozví, jaké funkce by jednotlivé aplikace měly uživatelům poskytovat.

S touto kapitolou úzce souvisí kapitola 5, která obsahuje návrh systému. Ten vychází právě z kapitoly předešlé. V této kapitole je popsána nejen zvolená architektura webové a mobilní aplikace, ale také návrh uživatelského rozhraní.

Následující kapitola 6 popisuje samotnou implementaci informačního systému. Kapitola nejdříve seznámí se strukturou celého projektu a následně stručně popíše jednotlivé části architektury, se kterou byl čtenář seznámen v kapitole předchozí.

Kapitola 7 pak uzavírá celý proces vývoje pojednáním o prováděném testování. Čtenář zjistí, jakým způsobem bylo prováděno testování nejen webové, ale také mobilní aplikace, a jakých výsledků bylo dosaženo.

Závěrečná kapitola 8 analyzuje a hodnotí získané výsledky, kterých bylo dosaženo při vývoji webové a mobilní aplikace. Následně jsou představena možná budoucí rozšíření tohoto systému.

Kapitola 2

Tvorba webových a mobilních aplikací

Díky vývoji IP spojení v roce 1988, kdy bylo uskutečněno první IP spojení mezi Evropou a Amerikou, mohla započít diskuse o možnosti webových systémů. Na začátku 90. let pak byl publikován návrh na vybudování projektu s názvem World Wide Web jako síť hypertextových dokumentů. [1] Následoval vznik prvního webového serveru a prvního webového prohlížeče. Tento web se skládal ze sady statických dokumentů, které se odkazovaly mezi sebou. Brzy ale softwaroví inženýři přišli na to, že architektura klient-server umožňuje prohlížeči být univerzálním uživatelským rozhraním pro aplikace, které budou běžet právě na serveru. [2]

Zvětšující se všudypřítomnost a popularita chytrých mobilních telefonů začala přitahovat pozornost vývojářů softwaru. Vývoj nativních aplikací, které běží přímo na operačním systému zařízení, se začínají stávat oblíbenější než aplikace webové. Nativní aplikace totiž mohou využívat různé nativní funkce zařízení (např. fotoaparát, senzory, ...). [3]

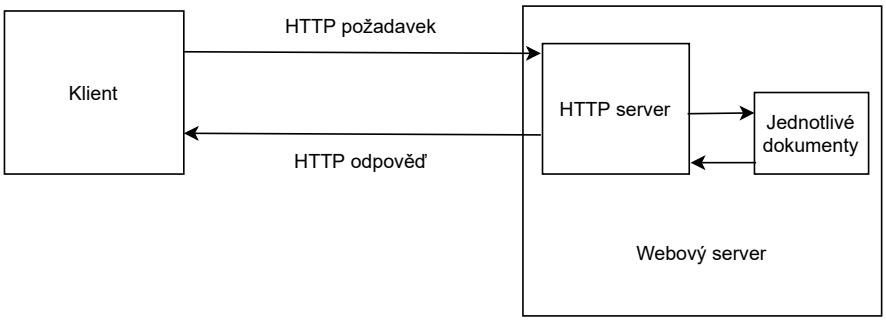
2.1 Webové aplikace

Jako webovou aplikaci můžeme označit aplikaci, která běží na místním nebo vzdáleném serveru. Kdy daný server klientovi, typicky webovému prohlížeči, odpovídá na HTTP[1] dotazy pomocí HTTP odpovědi ve formátu HTML dokumentu, případně serializovaných dat. V nynější době se ovšem pro komunikaci mezi klientem a serverem používá protokol HTTPS, což je obdobný protokol, který navíc provádí veškerou komunikaci v šifrované podobě. Kvůli tomu je před samotným požadavkem o daný dokument prováděn tzv. *handshake*, což je prvotní komunikace v nezašifrované podobě mezi klientem a serverem, kdy si vzájemně vyměňují konfigurační data k navázání bezpečného spojení. V dnešní době se na tento druh spojení klade velký důraz, a tak hraje způsob komunikace zásadní roli v SEO[2].

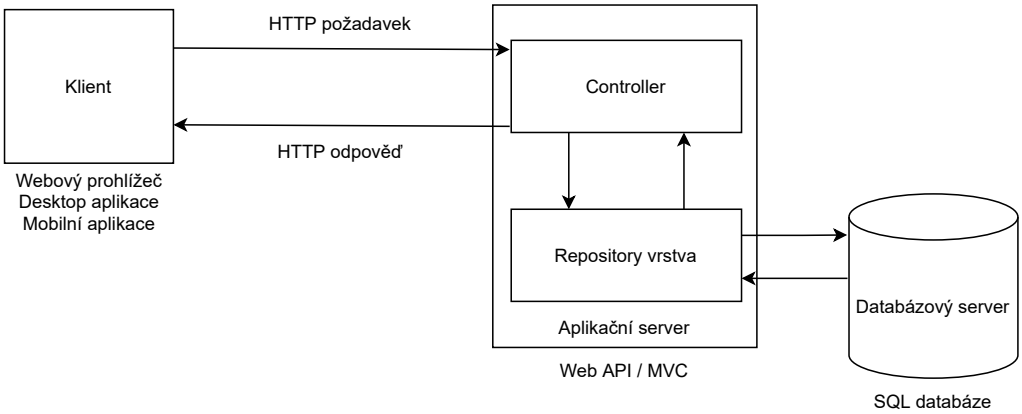
[1] HTTP je aplikační protokol pro komunikaci mezi klientem a serverem.
[2] SEO je optimalizace pro vyhledávače, kdy se stránka zobrazuje na předních místech vyhledávačů.

2.1.1 Princip

Rozdíl webové aplikace oproti běžné webové stránce je v tom, že běžný webový server vrací klientovi vždy stejný dokument pro danou URL[3] adresu. Zatímco webová aplikace získává dokument dynamicky, což znamená, že při dotaze na stejnou URL adresu mohou získat různí klienti různou odpověď. Toto je zapříčiněno tím, že si aplikační server (tj. webový server pro aplikace) ukládá vnitřní informace o daném klientu a mění tak svůj vnitřní stav.



Obrázek 2.1: Scénář komunikace statické webové stránky



Obrázek 2.2: Scénář komunikace s aplikačním serverem (ASP.NET)

Tento aplikační server pak danému klientovi předává data v takové podobě, které jsou pro daného uživatele informativní a relevantní. Server data získává z databázového serveru. Uživatel tedy požádá ve své klientské webové aplikaci o zobrazení libovolných informací aplikační server, který získá všechna potřebná data z databázového serveru a předá je klientovi v takové podobě, ve které jsou pro daného uživatele relevantní.

Zmíněný přístup odpovídá třívrstvé architektuře, která v dnešní době u webových aplikací převládá (třívrstvá architektura je dále popsána v 2.1.1). Dříve byla ovšem používána architektura klient-server.

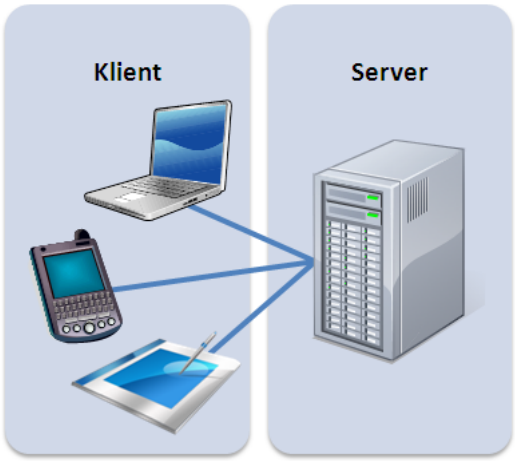
[3]URL je řetězec znaků, pomocí kterého je možné specifikovat přesné umístění zdroje informace na internetu.

Architektura klient-server

Tato architektura se skládá ze dvou částí: klient a server. Z tohoto důvodu se také často označuje jako *dvouvrstvá architektura*. Klient je žadatelem o nějaká data/službu a připojuje se k serveru (pro HTTP se jedná o databázový server), který mu může data/službu poskytnout. Veškerou aplikační logiku obsahuje klient, jehož úkolem je uživatelský požadavek převést do takové podoby, aby byl srozumitelný pro server. Následně také musí přeložit odpověď serveru. Tato architektura byla používána dříve ve webových aplikacích, kde byl klient označován za *tlustého klienta*, jelikož zajišťoval služby uživatelského rozhraní a také služby budoucího aplikačního serveru, který známe z architektury třívrstvé (2.1.1). Ta se později z této architektury vyvinula. [4]

Výhodou této architektury je rychlost, jelikož klient přímo komunikuje s databázovým serverem. Ovšem při používání vícero uživateli se výkon začne znatelně snižovat. Problém nastává také s aktualizací na straně klienta a vysokými požadavky na výkonnost klientova zařízení. Zásadním problémem je ale bezpečnost, kdy klient komunikuje přímo s databázovým serverem, který může obsahovat citlivá data. [5]

Ve své době se tato architektura dočkala velkého úspěchu, kdy byla používána pro HTTP komunikaci, která nyní využívá architekturu třívrstvou. Dále se s touto architekturou můžeme stále setkat u File Transfer Protocol (FTP) nebo Simple Mail Transfer Protocol (SMTP). [5]



Obrázek 2.3: Architektura klient-server (převzato z [4])

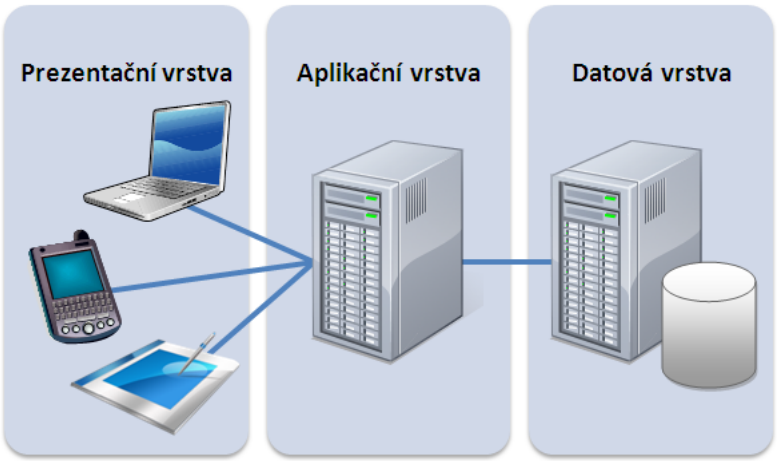
Třívrstvá architektura

Tato architektura je nyní nejpoužívanější architekturou webových aplikací, která organizuje aplikace do tří logických a fyzických počítačových vrstev. Hlavní výhodou je to, že každá vrstva běží na své vlastní infrastruktuře, takže každá vrstva může být vyvíjena samostatným vývojovým týmem a může být dále upravována dle potřeby, aniž by to mělo dopad na ostatní vrstvy. Výhodou této architektury je rychlejší vývoj, vyšší spolehlivost a bezpečnost, protože prezentační vrstva nemůže komunikovat přímo s vrstvou datovou, tudíž by nemělo být příliš možné zaútočit na databázový server. [6] Klient se už tedy stará pouze o grafické uživatelské rozhraní – jedná se tedy o *tenkého klienta*.

Tato architektura se skládá ze tří vrstev [7]:

- Prezentační vrstva – část, kterou vidí uživatel, většinou grafické uživatelské rozhraní. Kontroluje zadávané vstupy. Může se různit podle typu zařízení.
- Aplikační vrstva – část, která provádí výpočty a zpracování dat.
- Datová vrstva – nejčastěji formou databáze. Zajišťuje práci s daty (ukládání, výběr, agregace, ...).

Způsob komunikace je popsán v 2.1.1.



Obrázek 2.4: Třívrstvá architektura (převzato z [7])

Po vzniku třívrstvé architektury, která využívá služeb tenkého klienta, se v dnešní době označením tenký/tlustý klient rozlišuje typ klientské strany. Tenkým klientem rozumíme prohlížeč, který zobrazuje HTML stránku, kterou získal zasláním požadavku na server. Tlustým klientem pak označujeme webovou nebo mobilní aplikaci, která volá aplikační rozhraní tzv. *API*, které vrací pouze serializovaná data. Tato data jsou následně zobrazena pomocí některé front-end technologie (např. JavaScript).

2.1.2 Technologie webových aplikací

Díky velké popularitě webových aplikací v dnešní době existuje mnoho technologií pro vývoj. Z pohledu třívrstvé architektury (2.1.1) můžeme vybrat různé technologie v jednotlivých vrstvách.

Prezentační vrstva

Tato vrstva může mít několik typů z pohledu třívrstvé architektury. V této podsekci uvažujeme prezentační vrstvu jako webovou stránku v prohlížeči. Následující technologie jsou základními webovými technologiemi, které se vzájemně kombinují a doplňují.

HTML

Značkovací jazyk HTML byl primárně navržen jako jazyk pro sémantický popis vědeckých dokumentů. Jeho vývoj vede mezinárodní konsorcium W3C, které vyvíjí webové standardy společně se skupinou WHATWG. [8]

V dnešní době se používá primárně pro statické webové stránky. V případě webových aplikací se používá pro vytváření struktury dané stránky, jelikož World Wide Web očekává popis stránky právě pomocí HTML jazyka. [2]

CSS

Jazyk CSS neboli kaskádové styly je popis způsobu zobrazení elementů napsaných v jazyce HTML. Umožňuje tedy stylovat (font, velikost, barvu, pozici) jednotlivé prvky HTML stránky. V jeho počátcích se styl jednotlivých elementů psal jako atribut daného elementu. Tento způsob byl značně neefektivní, jelikož se stejné informace zobrazovaly na více místech. Později se přešlo k definování stylů do hlavičky HTML dokumentu a od CSS 3 je možné rozdělit styly do jednotlivých modulů. [9]

V dnešní době se velké popularitě dostávají mnohé CSS frameworky, které dodávají již nadefinované styly a vývojář může použít tyto styly podle potřeby. Mezi nejpoužívanější CSS frameworky můžeme zařadit Tailwind CSS, Bootstrap, Materialize CSS, Bulma, nebo také Foundation.

JavaScript

Díky jazyku JavaScript se ze statických webových stránek stávají stránky dynamické, které mohou měnit svůj obsah za běhu, aniž by se dotázaly serveru. Jedná se o lehký, interpretovaný jazyk. [10] V poslední dekádě patří mezi top 10 nejpoužívanějších programovacích jazyků. [11] JavaScript je nejpoužívanější skriptovací jazyk pro webové stránky, avšak je použit i v jiných oblastech, jako je Node.js, Apache CouchDB nebo Adobe Acrobat. [10] Největší popularitě se mu dostává za pomocí různých frameworků. Tyto frameworky poskytují předpřipravené funkce a usnadňují tak práci vývojářům.

Dle [12] jsou nejpopulárnějšími JavaScriptovými frameworky React, Vue.js, Angular a Django.

React je JavaScriptová knihovna, která je vyvíjena Facebookem jako open-source. Je vhodná pro moderní jednostránkové aplikace různých velikostí. Hlavní výhodou je vykreslování aplikace na straně serveru a využití virtuálního DOM[4], které vede k efektivnější aktualizaci aplikace. Nevýhodou je, že React neimplementuje návrhový vzor MVC. [12]

Vue.js je systematický framework pro vývoj uživatelských prostředí. Díky své funkčnosti, výkonnosti, rychlosti a jednoduchosti k naučení je jeden z nejvíce označovaných JavaScriptových frameworků na síti GitHub. Hlavní předností je jednoduchost a tím i rychlost, jelikož výrazně nezatěžuje klientský počítač. Nevýhodou může být malé zastoupení na komerčním trhu. [12]

Jako poslední podrobněji popsaný framework v této práci je **Angular**. Angular je oproti již zmíněnému Reactu a Vue.js složitější a komplexnější framework. Výhodou je, že při práci se stránkou se neaktualizuje celá stránka, ale pouze její část. Využívá návrhového vzoru MVVM, který umožňuje vývojářům pracovat s daty samostatně ve stejné aplikaci. Nevýhodou je složitost a tím pádem i délka učení. Tomuto problému nepřidává ani fakt, že jsou často prováděny velké aktualizace, které mění základní struktury celého frameworku. [12]

[4]DOM neboli Document Object Model je objektově orientovaná reprezentace dokumentů.

Aplikační vrstva

Jedná se o vrstvu, která provádí výpočty a zpracování dat. Je to tedy aplikační server, na který prezentační vrstva posílá dotazy. V této vrstvě se většinou uvažují pouze 3 možné technologie, přičemž 1 technologie silně dominuje.

PHP

Tato technologie je nejpoužívanější technologií v této oblasti. PHP využívá na serverové straně až 78 % webových aplikací podle [13]. Na rozdíl od jiných technologií se jedná pouze o PHP modul pro aplikační server, který typicky představuje Apache. PHP kód je tedy interpretován na serveru a generuje HTML soubory, které jsou následně zaslány uživateli. [14]

V dnešní době se dostává velké popularitě využití různých PHP aplikačních frameworků, které ulehčují následný vývoj aplikace. Vývojář již nemusí řešit nastavování různých HTTP hlaviček nebo zpracování požadavků, toto řeší právě daný framework pomocí svého API. Tím už vývojář řeší pouze vývoj své vlastní webové aplikace. Tyto frameworky implementují návrhový vzor MVC. Mezi nejpoužívanější frameworky můžeme zařadit Laravel, který je výkonný a také snadný na naučení. Dalším používaným frameworkem je Symfony, které využívá více paměti a má větší dobu odezvy, navzdory tomu je také složitější a vývojáři tak musí mít větší znalosti. Nesmíme opomíjet také CodeIgniter, který poskytuje jen základní abstrakci a pro vývojáře je tak velmi flexibilní. Není ovšem vhodný pro větší projekty. [15] V Česku a na Slovensku je navíc velmi populární také framework Nette.

Java

Dalším používaným jazykem na straně serveru podle [13] je Java se 4 %. V dnešní době se tedy nejedná o tolik populární prostředek k vývoji aplikačních serverů jako dříve. Na rozdíl od PHP pracuje na jiném principu, kdy jsou interpretovány značky a generuje se požadovaný obsah, například voláním různých značek a tzv. *beanů*. Výsledky pak odešle zpět ve formě HTML, případně XML, prohlížeči. Logika je tedy rozdělena a zapouzdřena do různých značek a beanů reprezentující jednotlivé operace na straně serveru. [16]

ASP.NET

Poslední nejpoužívanější technologií je ASP.NET z dílny Microsoft, která se používá u zhruba 8 % webových aplikací podle [13]. Tato technologie je součásti .NET technologií, která samotný .NET rozšiřuje o nástroje a knihovny pro vytváření webových aplikací. Samotný aplikační server může být psán v C# případně F# a může využívat návrhový vzor MVC. [17]

Nejdříve je každý požadavek od klienta zpracován pomocí IIS, což je webový server ASP.NET. Ten přesměruje daný požadavek na skriptovací stroj, který spouští skript na straně serveru. Po provedení skriptu odešle svou odpověď zpět serveru IIS. Ten přepošle odpověď klientovi, který požadavek vytvořil. [18]

Datová vrstva

Tato vrstva zajišťuje práci s daty. Nejčastěji se jedná o relační databázi, která ukládá jednotlivá data v tabulkách. Mezi nejpoužívanější databázové systémy můžeme zařadit MySQL, PostgreSQL či Microsoft SQL Server.

MySQL

Jedná se o open-source SQL databázi, která je vyvíjena společností Oracle. Tuto technologii využívají organizace typu Facebook, Twitter nebo Booking.com. Jedná se o vícevláknový server SQL, který podporuje několik klientských programů, knihoven, nástrojů pro správu a API. [19]

PostgreSQL

PostgreSQL je objektově relační databázový systém pod licencí open-source. Využívá, a dokonce i rozšiřuje jazyk SQL mnoha funkcemi, které mají pomoci vývojářům vytvářet aplikace. Jedná se tedy o výkonný a robustní databázový systém. [20]

Microsoft SQL Server

Jedná se o systém pro správu relačních databází vyvinutý společností Microsoft. Obsahuje řadu intuitivních funkcí, flexibilitu použití jazyka a platformy. Taktéž se jedná o open-source. [21] Tuto technologii využívá například síť StackOverflow.

2.1.3 Vývojová prostředí

Pro vývoj webových aplikací není nutné využívat speciální vývojová prostředí. V extrémním případě mohou být psána v Notepadu. Pro větší komfort vývojáře se však webové aplikace takto nevyvíjí. Vývojáři preferují vývojová prostředí, která vyznačují syntaxi daného jazyka, napovídají možné příkazy a případně umožňují ladění aplikace. Jiné požadavky pro tato vývojová prostředí ovšem nejsou, jelikož napsaný kód se v těchto vývojových prostředích následně spouští na lokálních, případně vzdálených, serverech. V následujících odstavcích jsou popsána 3 nejpoužívanější vývojová prostředí pro vývoj webových aplikací podle průzkumu StackOverflow v roce 2019 [22].

Visual Studio Code

Mezi nejoblíbenější a nejpoužívanější vývojové prostředí se uvádí Visual Studio Code. Jedná se pouze o editor zdrojového kódu, avšak s různými rozšířeními umožňuje některé zdrojové kódy také překládat, sestavovat a ladit. Tento editor je pro vývojáře přívětivý nejen tím, že se nejedná o náročnou a komplexní aplikaci, ale také tím, že lze doinstalovat libovolná rozšíření pro zvolený jazyk. Pokud tedy vývojář chce vyvíjet webovou aplikaci v PHP frameworku, stačí doinstalovat balíček pro PHP, případně i HTML, aby se barevně vyznačovala syntaxe daného zdrojové textu a editor podle kontextu napovídal možné příkazy.

Visual Studio

Dalším velmi používaným, ovšem často i neoblíbeným, vývojovým prostředím je mnohem mocnější prostředí Visual Studio. Toto vývojové prostředí primárně podporuje všechny technologie společnosti Microsoft, včetně tvorby GUI, různých ladění, a mnoho dalších balíčků. Každý uživatel si může nadefinovat, které balíčky chce používat, avšak tyto balíčky nejsou pouze pro zvýraznění syntaxe, ale pro kompletní práci s danou technologií. Tímto se stává z Visual Studia velice komplexní systém, který kvůli tomu často naráží po stránce výkonu a různých chyb. I proto někteří vývojáři toto prostředí zásadně nepoužívají.

Notepad ++

Třetím nejoblíbenějším vývojovým prostředím můžeme označit Notepad++. Tento editor je velmi podobný běžnému notepadu, avšak má zásadní vylepšení, která stojí za jeho popularitou. Hlavní předností můžeme označit podporu tzv. panelů, díky kterým lze upravovat více souborů najednou. Druhou zásadní výhodou oproti běžnému notepadu je podpora několika programovacích a skriptovacích jazyků, kdy Notepad++ zvýrazňuje syntaxi a napovídá možné příkazy podle kontextu. Dalším specifikem tohoto vývojového prostředí je automatická detekce jazyka. [23]

Podle průzkumu [22] jsou dalšími oblíbenými vývojovými prostředími pro vývoj webových aplikací například IntelliJ, Vim nebo Sublime Text.

2.2 Mobilní aplikace

S rostoucím počtem aktivních uživatelů mobilních telefonů vzniká větší poptávka po vývoji mobilních aplikací. Mobilní telefon má uživatel neustále při ruce, a tak může využívat danou mobilní aplikaci takřka kdykoliv a kdekoliv. Navíc tato aplikace může využít speciálních funkcí, kterými běžný stolní počítač nedisponuje, jako kamera a mikrofon, gyroskopy a GPS, ...

Mobilní aplikace lze obecně klasifikovat do 2 různých skupin:

- Nativní aplikace – jsou navrženy přímo pro zadanou platformu, kde jsou perfektně vyladěny.
- Multi-platformní aplikace – jsou navrženy pro různé platformy, ale nevyužívají jejich plný potenciál.

2.2.1 Nativní aplikace

Prvním krokem při vývoji nativní aplikace je výběr platformy. V dnešní době běží na 64 % všech mobilních zařízení v Evropě operační systém Android podle [24]. Systém iOS od společnosti Apple pak běží na 35 % zařízeních. Často náš výběr ale nezávisí pouze na podílu trhu. Vývoj mobilní aplikace pro systém iOS je rozhodně finančně náročnější než vývoj Android aplikace. To je také zapříčiněno tím, že vyvíjet aplikace pro iOS je možné pouze na zařízeních značky Apple, kde si musí vývojář zřídit speciální účet, který je zpoplatněn.

Proč tedy jít takto složitou cestou, když lze využít multiplatformní nástroj pro vývoj mobilní aplikace? Výhod je hned několik. Nativní aplikace jsou navrženy tak, aby bezchybně fungovaly na dané platformě. Mají vyšší vykreslovací výkon procesoru. Nativní aplikace také

lépe spolupracují se zařízením a funkce jako GPS, kamera, nebo různé další senzory fungují naprosto bezproblémově. Tyto aplikace také lépe a efektivněji vykreslují už tak špičkovou grafiku. Také nasazení do daného obchodu s aplikacemi je v případě využití nativní aplikace snazší. [25]

Nevýhodou je ale hlavně cena a čas vývoje. Místo 1 zdrojového kódu je nutné napsat 2 zdrojové kódy v různých jazycích. Jakákoliv oprava nebo aktualizace musí být provedena na 2 různých projektech, a to je rozhodně dražší. [25]

Pokud chce tedy zákazník perfektně odladěnou aplikaci, která má silný potenciál, a chce cílit pouze na 1 platformu, pak rozhodně využije nativní vývoj. A to i v případě, že požaduje vytvoření aplikace pro obě platformy, nemá vysoké časové požadavky, a navíc oplývá dostatečnými finančními prostředky.

Technologie nativních aplikací

Volba technologie závisí výhradně na zvolené platformě. Pro každou platformu dominuje vždy 1 technologie.

Android

Pokud se vývojář nebo zákazník rozhodne pro nativní aplikaci pro Android, vývojář s největší pravděpodobností využije jednu z těchto technologií.

Java je výchozím jazykem pro vyvíjení aplikací pro Android od samého počátku. Jedná se o objektově orientovaný programovací jazyk, který byl vyvinut společností Sun Microsystems. Dnes jej vlastní společnost Oracle. Java se kompiluje do tzv. bytecodu, který je za běhu interpretován základním Java Virtual Machine, který běží na operačním systému. [26]

Nejznámější aplikace napsané v Javě jsou: Spotify, Twitter nebo OperaMini.

Kotlin byl v roce 2017 uznán Googlem jako alternativní prvotřídní jazyk pro programování pro Android. Kotlin je interoperabilní s Javou a všechny Java knihovny lze v Kotlinu využít. Dá se tedy říci, že Kotlin je čistší forma Javy. Kotlin se kompiluje do Java Bytecodu. Od roku 2019 je podle Google preferovanější právě jazyk Kotlin. [26]

Nejznámější aplikace napsané v Kotlinu jsou: Airbnb, Duolingo, Uber nebo Netflix.

iOS

Chce-li vývojář vyvíjet aplikace pro Apple zařízení, je nutné, aby vlastnil zařízení Mac a měl účet vývojáře Apple. Následně si může vybrat ze 2 nejpoužívanějších technologií.

Objective-C byl prvním jazykem, který byl podporovaný společností Apple pro vývoj mobilních aplikací pro iOS. Jedná se o objektově orientovaný jazyk, který je syntaxí podobný jazyku C nebo SmallTalk. Největším problémem tohoto jazyka je syntaxe, protože obsahuje nespočet závorek, které mohou často způsobit chybu. [26]

Swift byl představen společností Apple v roce 2014. Tento jazyk se neustále výrazně vyvíjí, což je zatím stěžejní problém. Po zásadní revizi v roce 2016 překonal Swift starší programovací jazyk Objective-C jako jazyk pro aplikace iOS. Od jeho představení tak popularita Objective-C výrazně klesá. Ačkoli Swift a Objective-C spolu mohou koexistovat a jsou vzájemně kompatibilní, Apple dává jasně najevo, že Swift je tou správnou cestou pro vývoj iOS aplikací. [26]

V dnešní době jsou veškeré zásadní aplikace pro iOS napsány právě v jazyce Swift.

2.2.2 Multi-platformní aplikace

V případě, že chceme oslovit co nejvíce uživatelů a zákazník nemá dostatečné finanční prostředky, můžeme zvolit vývoj multiplatformní aplikace. Ty nám umožní použít jazyk nebo technologii, které jsou běžné při vývoji aplikací na jiné platformy (desktop, web, ...). Vývojáři se tedy nemusí učit nový jazyk a mohou pracovat s technologií, kterou již dobře znají. Nevýhodou tohoto řešení je nemožnost využít veškeré nativní funkce dané platformy a také ne tolik vyladěný běh aplikace. [25]

V dnešní době různé frameworky a nástroje pro vývoj multiplatformních aplikací využívají několik postupů a technik, které se snaží tyto nevýhody eliminovat.

Technologie multiplatformních aplikací

Různé komunity nebo společnosti přišly s frameworky, které mají blízko k běžným vývojářským technikám. Tyto frameworky přicházejí s vlastními ekosystémy nástrojů, díky nimž jsou pro vývojáře přívětivé. [26]

PhoneGap

Jedná se o framework web-to-native, který převádí HTML kód a JavaScript do speciálního webového rozhraní, které umožňuje využít nativní funkce zařízení. Pro svůj běh využívá framework JavaScriptu tzv. Ionic, který je postaven na Angularu. Jedná se o relativně mladou technologii, jelikož byla představena v roce 2013. [27]

Jeho HTML kód je součástí samotné aplikace, která je nainstalována na mobilním zařízení. Hlavní nevýhodou tohoto řešení je příliš pomalá odezva způsobená JavaScriptem, který běží ve webovém rozhraní. [26]

Xamarin

Xamarin je překladač zdrojového kódu, který je napsán v jazyce C#. Tento způsob vývoje je velice používaný, jelikož vývojář nepotřebuje žádné speciální znalosti k vývoji mobilních aplikací. Kód v jazyce C# je kompilován do nativní podoby podle typu platformy. Tyto aplikace pak působí jako nativní a mohou využívat mnoho nativních funkcí. Hlavní nevýhodou tohoto řešení je, že některé části kódu se pro jednotlivé platformy liší, a vznikají tak 2 různé zdrojové kódy. [27]

React Native

React Native je open-source technologie vyvinuta společností Facebook. Aplikace jsou tvořeny pomocí JavaScriptu a jsou téměř k nerozeznání od nativních aplikací. Tato technologie je postavena na frameworku ReactJs, jehož nezbytností je DOM. [27] Tento kód je následně interpretován za běhu a spuštěn pomocí tzv. bridge, který umožňuje využít nativní funkce zařízení. Jelikož React Native využívá k vykreslování komponent nativní knihovnu dané platformy, je jeho uživatelské rozhraní stejné jako u nativních aplikací. [26]

Flutter a Dart

Flutter je open-source framework pro tvorbu UI od společnosti Google, který navíc zvyšuje i výkon aplikace na různých platformách. Kód aplikace je pak napsán v jazyce Dart, což je programovací jazyk společnosti Google. Tento kód je kompilován do nativního kódu. [25]

2.2.3 Vývojová prostředí

Na rozdíl od vývoje webových aplikací, který nutně nevyžaduje speciální vývojové prostředí (2.1.3), jsou tato prostředí nezbytnou součástí vývoje. Umožňují totiž vývojáři provádět integrované ladění na fyzickém, případně emulovaném, mobilním zařízení. Výběr prostředí pak úzce souvisí s vybranou technologií pro daný vývoj. V následujících odstavcích jsou popsána 4 nejčastěji používaná vývojová prostředí pro vývoj mobilních aplikací podle [22].

Android Studio

Jak již název napovídá jedná se o vývojové prostředí určené pro vývoj mobilních aplikací na platformu Android. Toto prostředí je vydáno společností Google a jedná se o nejběžnější prostředí pro Android, jelikož obsahuje všechny podstatné funkce (profiling, podepisování, návrh UI, emulátor, ...). Umožňuje psát aplikace v jazyce Java a Kotlin (2.2.1), tedy nativní aplikace. Navíc je dostupný jak na Linux, tak i na Windows a Mac OS X.

Xcode

Xcode je naopak vývojové prostředí pro vývoj iOS aplikací. Prostředí vydává nepřekvapivě společnost Apple. V tomto prostředí se dají vyvíjet aplikace nejen pro platformu iOS, ale také Mac OS. Součástí je kompilátor, debugger, emulátor, návrhář UI a různé nástroje pro testování a analýzu výkonu aplikace. Xcode podporuje vývoj aplikací nejen v jazycích Objective-C a Swift, což jsou jazyky pro vývoj iOS aplikací, ale také C++, Java, Python, či Ruby. Nevýhodou tohoto prostředí je, že je určený pouze pro platformu Mac OS. Avšak všechna ostatní vývojová prostředí umožňují vývoj iOS aplikací pouze na zařízeních Apple. [28]

Visual Studio Code

Jak již bylo zmíněno v 2.1.3, VS Code je v současné době opravdu oblíbené. To potvrzuje i fakt, že je VS Code uváděno jako 2. nejpoužívanější vývojové prostředí i pro vývoj mobilních aplikací. Jak již bylo zmíněno, do tohoto prostředí lze přidat jednotlivé balíčky, které umožňuji editovat, překládat, spouštět a ladit jednotlivé programy. Těchto balíčků je mnoho právě i pro vývoj nativních aplikací na platformy Android, či iOS (pouze na zařízeních Mac OS), ale také pro vývoj multiplatformních aplikací. Balíčky obsahují také emulátory a je tedy možné s nadsázkou říci, že je možné vyvíjet jak webové, tak nativní aplikace pro obě platformy, za použití jednoho prostředí, a to VS Code.

Visual Studio

Posledním uvedeným prostředím v této části je Visual Studio. To bylo již dříve popsáno v 2.1.3. Toto prostředí umožňuje vývoj multiplatformních mobilních aplikací pomocí technologie Xamarin. Taktéž obsahuje emulátor jak pro Android, tak iOS (pouze na zařízeních Apple). V tomto prostředí tedy není možné vyvíjet nativní aplikace v Kotlinu, nebo Swiftu.

Kapitola 3

Optické rozpoznávání znaků

Optické rozpoznávání znaků (OCR) se zabývá elektronickou konverzí z ručně psaného nebo tištěného textu na odpovídající alfanumerické nebo jiné znaky. Tato technologie umožní snížit náklady a vynaložený čas pro převod velkého množství papírových dokumentů do elektronické podoby. [29]

V posledním desetiletí se jedná o jednu z nejčastějších oblastí výzkumu v rozpoznávání vzorů pro jeho velký aplikační potenciál. OCR patří do skupiny technik, které provádějí automatickou identifikaci. Tím tedy systém sám, bez zásahu člověka, automaticky identifikuje objekty a převede jej na data, které přímo předá do počítačového systému. [29]

3.1 Historie

Počátky výzkumu OCR lze nalézt již v 70. letech 19. století, kdy americký vynálezce Charles R. Carey představil přenosný systém s mozaikou fotobuněk. Samotný výzkum OCR započal až na začátku 30. let 20. století, rok poté, co si německý průkopník informačních technologií Gustav Tauschek zaregistroval patent pro zařízení s foto-senzorem. [29]

V rámci rychlého vývoje informačních věd a informačních systémů, se v 50. letech 20. století zvětšovala snaha vytvořit technologii, která by umožňovala efektivněji zpracovávat čím dál větší množství dat. Tento počátek vědecko-technické revoluce vedl ke vzniku mnoha neobvyklých informačních systémů. V roce 1954 bylo v magazínu Reader´s Digest instalováno zařízení, které umožňovalo převod strojově psaných zpráv o prodeji na děrné štítky, které mohly být dále zpracovávány v počítači. [29] Tímto byl vytvořen první čtecí stroj.

Komerční systémy, které se objevovaly v letech 1960-1965, jsou nazývány jako OCR 1. generace. Pro tyto stroje je typické omezení použití pouze na předem definované znaky. Následně se začaly objevovat stroje, které umožňovaly zpracovávat více fontů, avšak počet písmen byl stále omezen.

V letech 1965 až počátkem 70. let se objevily stroje druhé generace. Tato generace se vyznačovala tím, že stroje byly schopny rozpoznat nejen běžné strojově tištěné znaky, ale také texty psané. Jejich znaková sada však byla omezena pouze na základní znaky a číslice.

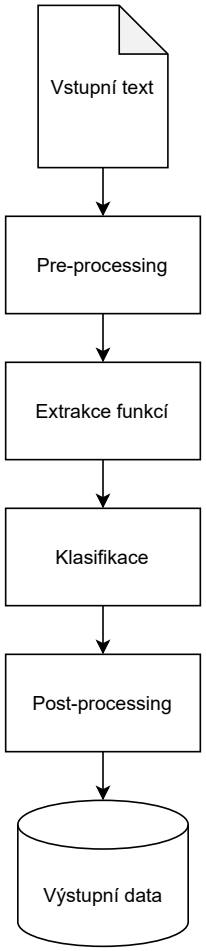
Třetí generace se objevila v polovině 70. let 20. století. Tato generace již nepotřebovala speciální fonty, což umožňovalo čtení psaných textů (např. z psacích strojů). Tohoto úspěchu bylo dosaženo díky drastickému pokroku v hardwaru.

Všechny tyto studie trpěly nedostatkem výpočetního výkonu, a tak byla pořizovací cena systémů značně vysoká. Když ceny hardwaru, dostatečnému této problematice, začaly

klesat, začaly být systémy OCR dostupné jako softwarové balíčky, čímž se vývoj a prodej opět urychlil. [30]

3.2 Techniky systémů OCR

Typický systém OCR je složen z několika komponent. Nejdříve je nutné provést předzpracování, jelikož vstupní skener nikdy není dokonalý. To znamená, že se eliminuje šum, provede se vhodné natočení apod., aby byla extrakce prvků přesnější. Předzpracované symboly se následně na základě vlastností porovnají s popisy tříd. Tyto informace jsou poté použity k rekonstrukci původního textu. Celý systém zpracování je znázorněn na obr. 3.1. Toto schéma se velice často liší v různých literaturách podle podrobnosti popisu jednotlivých komponent. Často jsou uváděny také fáze před samotným předzpracováním, jako optické skenování či segmentace umístění. Také fáze klasifikace bývá často rozdělována do dalších fází kvůli své složitosti.



Obrázek 3.1: Schéma systému OCR

3.2.1 Předzpracování

První položkou zmíněnou v této práci (v literatuře [29] až 3. položkou po již zmíněném optickém skenování, či segmentaci umístění) je předzpracování neboli pre-processing. Obraz, který je výsledkem skenování, může v závislosti na skeneru obsahovat nežádoucí množství šumu. Navíc mohou být znaky například rozmazané, což závisí na rozlišení daného skeneru. Některé z těchto problémů, které by mohly v budoucnu narušit přesnost rozpoznání, lze eliminovat právě pomocí pre-processingu. Tato komponenta se tak snaží produkovat data, která umožní systému OCR snadno a přesně provést klasifikaci znaku. Za hlavní cíle pre-processingu můžeme tedy označit [31]: snížení šumu, normalizaci dat a kompresi množství informací.

Samotný šum může způsobit rozdělení segmentů čar, nerovnosti a mezery v řádcích apod. Tyto neduhy lze odstranit různými technikami redukce šumu, které se typicky dělí do 3 skupin podle [31]:

- Filtrování – jeho cílem je odstranit šum a zmenšit rušivé body, které vznikají nerovným povrchem. Hlavní myšlenkou je otočit předdefinovanou masku s obrázkem a přiřadit hodnotu pixelu jako funkci hodnot šedi sousedních pixelů.
- Morfologické operace – tyto operace byly navrženy proto, aby spojovaly přerušené tahy, rozpojovaly spojené tahy, ztenčovaly znaky, extrahovaly hranice aj. Základní myšlenkou je filtrovat znak a nahradit konvoluce za logické operace.
- Noise modeling – pokud lze šum modelovat, lze jej odstranit kalibračními technikami. Modelování hluku ovšem typicky není možné. Této problematice se věnují další literatury uvedené v [29].

Normalizace dat řeší problém, kdy OCR systém očekává standardizovanou velikost vstupu. Pokud by tak nebylo učiněno, mohla by být klasifikace nesprávná. Proto je potřeba získat standardní rozměr obrazu právě pomocí normalizace. Mezi nejpoužívanější metody patří [31]:

- Normalizace zkosení – řeší naklonění a zakřivení písma. Tyto problémy mohou být způsobeny kvůli nepřesnostem v procesu skenování, případně stylu psaní. Používanými metodami jsou: použití projekčního profilu obrazu, shlukování nejbližších sousedů a také Houghuova transformace.
- Normalizace sklonu – zde se jedná o problém různých stylů rukopisů. Pomocí této normalizace se normalizují všechny znaky do standardního tvaru.
- Normalizace velikosti – je používána k úpravě velikosti na definovaný standard. Každý znak je rozdělen do několika zón, kde každá zóna je samostatně škálována.

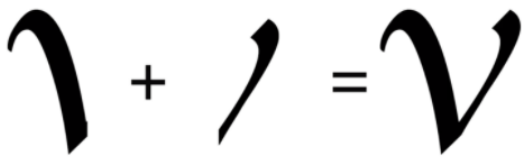
Obrázek 3.2: Postupná normalizace velikosti znaku (převzato z [29])

Komprese snižuje datovou a výpočetní náročnost. Klasické komprese obrazu by obraz transformovaly do nevhodné oblasti domén pro rozpoznávání. Používané kompresní techniky tedy jsou:

- Prahování – obrázky ve stupních šedi nebo barevné obrázky se převedou na binární obrázky výběrem prahové hodnoty. Vybere se buď globální práh, tedy práh stejný pro celý obrázek, který je založen na odhadu úrovně pozadí pomocí histogramu. Nebo se použije místní neboli adaptivní prahování, které používá různé hodnoty pro každý pixel podle informací o dané oblasti (tj. nejbližšího okolí/sousedů).
- Ztenčování – sníží velikost dat, ale extrahuje informace o tvaru znaků. Základní přístupy se zakládají na ztenčování po pixelech a ztenčování bez pixelů.

3.2.2 Extrakce funkcí

Cílem extrakce je zachytit základní charakteristiky symbolů, které upřesní výslednou klasifikaci. Zjednodušeně lze říci, že pokud máme 2 rovnice úseček, které se setkávají dole, může se jednat o písmeno V. Těchto informací se pak využívá podle typu klasifikace. Přesná klasifikace je pak popsána v kapitole 3.2.3.



Obrázek 3.3: Příklad extrakce funkcí (převzato z [32])

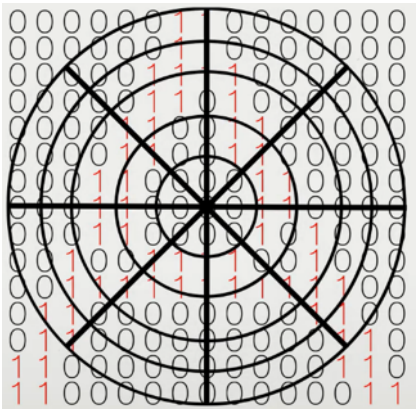
3.2.3 Klasifikace

Tato fáze se zabývá správným klasifikováním znaku a jeho následným zařazením do dané třídy. Za nejdůležitější třídu znaků můžeme považovat třídu, která určí, zda se jedná o číslo, písmeno, interpunkci, speciální znak nebo znak jiný. Mezi další třídy patří například určení jazyku, nebo fontu. Po přiřazení správné třídy znaku dochází k jeho identifikaci. Klasifikace se dá rozdělit na:

- Klasická klasifikace
- Soft computing

Klasická klasifikace

V této klasifikaci je využito charakteristik, které se nazývají *deskriptory*. Tyto deskriptory popisují vlastnosti jednotlivých znaků a podle jednotlivých vlastností jsou následně znaky zařazeny. Zařazení je pak prováděno maticovým porovnáním, kde se využívá minimální vzdálenosti v prostoru deskriptorů mezi znakem a jeho vzorem, případně vztahy mezi jednotlivými tahy. [32]



Obrázek 3.4: Maticové porovnání znaku (převzato z [32])

Pokud tedy máme znak popsaný z kapitoly 3.2.2, použijeme deskriptory pro rozhodování, zda opravdu patří do očekávané třídy či nikoliv. K tomu ovšem potřebujeme také znát vztah mezi danými úsečkami.

Klasické klasifikace lze provést podle dalších charakteristik, jako jsou barvy pixelů, kde je použit Gaussův filtr a maticové porovnání kritériem nejmenší vzdálenosti. Další možností je využít charakteristiku vertikální polohy v řádku, kde nás zajímá minimální a maximální y-ová souřadnice. Tímto můžeme například rozlišit písmeno g a číslici 9.

Tato metoda již není příliš používaná, a to kvůli omezenému množství znaků, které umí rozeznat. Pro každý znak musíme definovat deskriptory, což se velice komplikuje kvůli stylům písma v ručně psaném textu. Proto se této metodě říká také často teoretické rozhodování. [29]

Soft computing

Tato technika je naopak nejmodernější a nejpoužívanější metoda. Jedná se o matematický model, který se snaží co nejvíce přiblížit lidskému mozku a chápání. Soft computing je důležitým matematickým aspektem v oblasti rozpoznávání vzorů a byl aplikován téměř do všech oborů inženýrství a věd. [33] Soft computing dokáže na základě výsledků, které získal z experimentálních dat, generovat přibližné výstupy pro neznámé vstupy. Celý výpočetní proces lze popsat jako transformaci modelovaného prostoru M do prostoru řešení S a následně do rozhodovacího prostoru D. [33]

$$M \rightarrow S \rightarrow D$$

kde $S \rightarrow D$ je rozhodovací funkce a D jsou rozhodnutí.

Mezi nejdůležitější části soft computingu patří [29]:

- Fuzzy množiny – jelikož lidský mozek interpretuje nepřesné a neúplné smyslové informace, je potřeba se s takovými informacemi lingvisticky vypořádat. Fuzzy množiny pak provádí numerické výpočty pomocí jazykových označení.
- Umělé neuronové sítě – neboli ANN, napodobují lidský nervový systém. V OCR využíváme schopnosti perceptronu neboli umělého neuronu, který je schopen se naučit klasifikovat jednotlivé vstupy pomocí učení.
- Genetické algoritmy
- Rough sets

Nyní už tedy víme, že soft computing nám umožní rozpoznávat zcela nové znaky za využití učení perceptronu.

Učení může být prováděno bez učitele, kde jsou systému přiváděna neoznačená data, která byla vygenerována algoritmem. Výhodou řešení je, že trénovací data, na kterých se ANN učí, může generovat algoritmus. Ten může vytvořit mnohem větší množství dat, než jaké bychom byli schopni zadat ručně. Toto řešení má ovšem i nevýhodu, a to požadavek na vysoký výkon. Proto se učení bez učitele používá pro např. počítačové vidění nebo zpracování dat. Může být tedy využito také pro rozpoznávání textu. [29]

Druhý způsob učení je učení s učitelem, kde jsou předávána trénovací data, která jsou označená informacemi o tom, do které třídy mají náležet a jaký reprezentují znak. K tomuto učení se využívá stochastických gradientních metod (chcete-li náhodného provádění Hill-climbingu, což je informovaná metoda prohledávání stavového prostoru) v umělé neuronové síti. [29]

3.2.4 Následné zpracování

Po provedení klasifikace a rozpoznání textu se musí provést tzv. post-processing. Tato komponenta provádí korekci výstupu z důvodu nedokonalého rozpoznání znaků. Před samotnou detekcí a opravou chyb je potřeba provést seskupování.

Výsledkem klasifikace jsou pouze jednotlivé symboly, které neobsahují dostatek informací. Tyto symboly jsou vzájemně spojeny, aby tvořily slova a kompletní čísla. Seskupování symbolů do řetězců je založeno na umístění symbolů v dokumentu. Díky tomu lze opravit některé chyby s použitím kontextu. Avšak ani nejlepší rozpoznávací systémy neposkytnou 100% správnou identifikaci všech znaků [34].

Mezi nejčastější důvody vzniku chyb patří [35]:

- Vady zobrazení – tlusté/slabé písmo, nerovné řádky
- Podobné symboly
- Interpunkce

Pro opravu chyb můžeme použít lexikální nebo kontextovou korekci.

Lexikální korekce

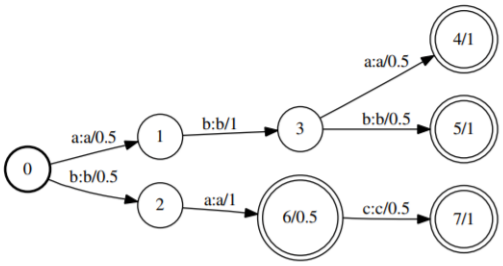
Lexikální korekce používá 2 různé přístupy. První z nich využívá pravidel definujících syntaxi slova. Pro různé jazyky vypočítáme pravděpodobnost, že se 2 nebo více znaků objeví společně za sebou. Pokud se zjistí, že pravděpodobnost nalezeného znaku je nulová, je téměř jisté, že nastala chyba. Tato chyba se opraví nahrazením chybného písmena za podobné písmeno s největší pravděpodobností.

Další přístup je použití slovníků, což je účinnější. Pokud systém OCR rozpozná slovo, které není ve slovníku, je nahrazeno slovem podobným. Nevýhodou této metody je, že opravené slovo stále nemusí být tím správným hledaným slovem. To se v tomto případě již nedá zjistit. Další nevýhodou je doba trvání vyhledávání a porovnávání slov ve slovníku. [29]

Kontextová korekce

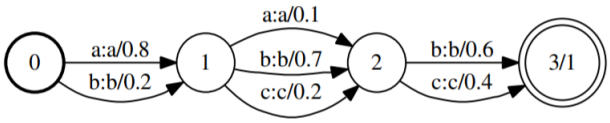
Tato korekce bere v úvahu kontext celého souvislého textu. Informace o daném jazyce lze zachytit v jazykovém modelu. V tomto případě se jedná o formální nebo stochastický jazykový model. Dalším méně používaným modelem je model chybový. [36]

Formální jazykový model neboli LM, používá gramatický inferenční algoritmus k sestavení stochastického konečného stroje. Tento stroj pak zjišťuje, zda skladba věty vyhovuje pravidlům gramatiky daného jazyka. [36]



Obrázek 3.5: Příklad použití formálního jazykového modelu (převzato z [36])

Dalším jazykovým modelem je stochastický jazykový model. Ten vychází ze sekvence vektorů, které určují pravděpodobnost výskytu daného slova. [36]



Obrázek 3.6: Příklad použití stochastického jazykového modelu (převzato z [36])

3.3 Dostupné knihovny

Díky velkému hardwarovému pokroku se systémy OCR mohly ze složitých hardwarových strojů přenést do softwarových balíčků (3.1). V dnešní době tak má vývojář k dispozici již několik OCR knihoven pro jeho aplikaci. Mezi přední společnosti v této oblasti patří Google, Microsoft a Adobe.

3.3.1 Tesseract

Jako první engine uvádím Tesseract, jehož vývoj nyní vede Google jako open-source. Tento engine původně vyvinula společnost Hewlett-Packard, kdy v roce 2005 byla vydána jako open-source a začala být mocně sponzorována právě společností Google. [37]

Podle zdroje [38] má Tesseract průměrnou úspěšnost 43 %. Zdroj [39] uvádí průměrnou úspěšnost 70-78 %. V obou případech jde o jeden z nejhorších výsledků v porovnání s konkurencí. Hlavní výhodou je podpora více než 100 jazyků [40] a také licence open-source. Díky tomu se stává Tesseract nejpoužívanějším enginem vůbec.

3.3.2 Google Cloud Vision AI

Tento engine je provozován v cloudu společnosti Google. Jedná se o jeden z nejlepších a nejpřesnějších OCR systémů dnešních dob. Jeho úspěšnost se pohybuje okolo 90-96 %. [39] Podporuje více než 200 jazyků a díky tomu se jedná o velmi populární balíček. Hlavní nevýhodou je nejen licence nesvobodného softwaru, kdy celý vývoj řídí Google, ale také vysoká cena.

3.3.3 Azure Cognitive Services

Balíček vydávaný společností Microsoft se dá považovat za celkem zdařilý. Nejedná se o licenci open-source, avšak pro základní užití s limitovaným počtem dotazů za měsíc je klíč k této API zcela zdarma (pokud jste vlastníkem Azure účtu). Jeho úspěšnost je značně vyšší než engine Tesseract (3.3.1) a to 76 % podle zdroje [38]. Hlavní nevýhodou je již zmíněná licence nesvobodného softwaru, kdy alespoň Microsoft umožňuje limitované použití zcela zdarma, na rozdíl od API společnosti Google.

3.3.4 Anyline

V tomto případě se nejedná o pouhé API jako u předchozích, ale o celé SDK pro mobilní zařízení. Jedná se o placené SDK, které lze pro testovací účely použít zdarma po určitou dobu.

3.3.5 ABBYY

Taktéž velmi oblíbený SDK cloud OCR systém, který má vysokou úspěšnost pro zpracování PDF dokumentů, přesněji 94 %. Celkově pak jeho úspěšnost rozpoznání je odhadována na 77-94 % podle testu [39]. Počet různých jazyků, které umí zpracovat by se měl přibližovat ke 200. Hlavní nevýhoda je opět cena, která je u tohoto SDK nejvyšší ze všech předcházejících.

Existuje mnoho dalších enginů nebo SDK. Levnější řešení mají zpravidla horší úspěšnost rozpoznání než API nebo SDK za větší poplatek. Tomuto tvrzení se ovšem vyjímá API Tesseract (3.3.1), kde sice jeho úspěšnost není nejvyšší, avšak v některých případech dosahuje lepších hodnot než jiné nesvobodné nástroje. Proto se dá považovat za jedno z nejlepších řešení. Mnoho dalších knihoven, které vydávají menší společnosti staví právě na Tesseractu. V případě, že vývojář hledá co nejpřesnější OCR systém, nemělo by mu uniknout Google Cloud Vision AI. V případě vlastnictví Azure účtu je ideální volba využít právě jejich OCR systém.

Kapitola 4

Analýza a specifikace požadavků

Při vývoji software je fáze analýzy a specifikace požadavků zásadní. Zákazník nejdříve uvede své požadavky. Při komunikaci je potřeba se zákazníka opakovaně doptávat, jelikož ani zákazník sám neví, co přesně požaduje. Jakmile sdělí všechny své požadavky, je nutné zjistit, zda jsou jeho představy reálné a splnitelné. Teprve poté se může přejít k návrhu systému. V průběhu vývoje zákazník požadavky upřesňuje nebo dokonce mění, což může celý vývoj prodloužit.

Prvotním požadavkem pro tento systém je evidence objednávek jednotlivých pacientů. Pro základní evidenci pacientů bude nutné znát osobní údaje (jméno, příjmení, RČ, datum narození a bydliště), které lékař, případně sestra, vyplní při registraci pacienta. Systém bude lékaři přehledně zobrazovat nadcházející objednávky pro daný den, nebo pro zvolené časové období. Pro každého pacienta bude možné vytvářet objednávky nové, případně upravovat či mazat, již existující. Ke každé objednávce pak bude možné přidat další informace o vyšetření.

Hlavní důvod tvorby tohoto systému je umožnit pacientům sledovat a upravovat své nadcházející objednávky. Pacient bude moci použít mobilní aplikaci, která bude napojena na tento informační systém. Jestliže byl pacient zaregistrován u lékaře, budou mu po zadání nebo načtení osobních údajů do mobilní aplikace zobrazeny nadcházející, ale i minulé objednávky u lékaře. Nadcházející objednávky mu budou přidány do Google kalendáře a den před návštěvou lékaře aplikace vytvoří notifikaci jako upomínku. Pacient bude také moci případně požádat o přeobjednání.

4.1 Informační systém pro lékaře

Se samotným informačním systémem, který bude vytvořen jako webová aplikace, bude moci pracovat lékař, případně zdravotní sestra.

Před samotným použitím webové aplikace bude nutná autentizace pomocí přihlašovacího jména a hesla. Samotné přihlašovací údaje bude vytvářet administrátor webové aplikace.

Administrátor webové aplikace

Tento uživatel bude moci spravovat přihlašovací údaje ostatních uživatelů. Administrátorovi bude umožněno vytvářet, upravovat a mazat jednotlivé uživatele, tedy účty do webové aplikace. Administrátor taktéž může být i lékařem a může provádět veškeré další operace, jako běžný uživatel.

Správa ordinační doby

Každý uživatel bude mít možnost si určit svou ordinační dobu, na kterou pak bude brán zřetel při tvorbě objednávek. Taktéž ji budou moci zjistit ostatní uživatelé a v neposlední řadě také pacienti, kteří budou moci žádat o přeobjednání apod.

Informace o lékaři

Pro zjištění informací o daném lékaři, může libovolný uživatel využít hledání v seznamu lékařů. Následně je schopen zjistit základní osobní údaje o lékaři, dále jeho ordinační dobu a také si zobrazit jeho nadcházející objednávky v kalendáři, kde si jednotlivé objednávky může zobrazit pouze pro čtení.

Registrace pacientů

Samotnou tvorbu pacientů budou moci provádět všichni lékaři. Při registraci bude potřeba vyplnit základní údaje (jméno, příjmení, RČ, datum narození, telefonní číslo a bydliště). Tyto údaje budou řádně zkontrolovány, aby nemohlo dojít k chybnému zadání. Dále se budou u jednotlivých pacientů zapisovat anamnézy a různé poznámky. Takto registrovaný pacient již bude dostupný pro všechny lékaře.

Úprava pacientů

Také úprava pacientů bude dostupná pouze ve webové aplikaci. Úpravu bude možné provádět jak při návštěvě daného pacienta, tak po vyhledání pacienta v seznamu všech pacientů (4.1). Pacienty v systému nebude možné mazat, aby byly jejich anamnézy apod. neustále k dohledání.

Seznam všech pacientů

Pacienti, kteří právě nebudou na vyšetření a nebude tedy možné přejít na kartu pacienta (4.1) pomocí kliknutí na objednávku v zobrazení objednávek, bude možné vyhledat v seznamu všech pacientů. Pacienty bude možné vyhledat podle příjmení nebo rodného čísla.

Karta pacienta

Tato záložka bude zobrazovat osobní údaje pacienta s poznámkami k danému pacientovi. Hlavní funkcí této karty bude možnost objednat daného pacienta (4.1). Dále zde bude možnost zobrazit všechna objednání tohoto pacienta (jak nadcházející, tak minulé), upravit anamnézu, či upravit osobní údaje (4.1).

Tvorba objednávky

Vytvořit novou objednávku bude možné při zobrazení karty pacienta (4.1), zobrazení detailu objednávky (4.1), či tvorby úplně nové objednávky z přehledu objednávek (při této možnosti bude jako 1. krok požadován výběr pacienta). Následně lékař zvolí, jaké úkony budou prováděny. Jednotlivé úkony bude možno přidávat se zvolenou odhadovanou dobou provádění. Následně bude vybrán čas objednání, přičemž systém upozorní na pokus o vytvoření kolidujících objednávek, případně objednávek mimo pracovní dobu. O této skutečnosti však pouze uživatele zřetelně upozorní, jelikož v některých případech je tvorba kolidujících objednávek akceptovatelná. Délka objednávky bude navrhnuta podle vybraných operací

a následně může být změněna lékařem. Ke každé objednávce musí být možné přidat také libovolný text. Každý lékař může objednávat pacienty pouze pro svou ordinaci.

Úprava a mazání objednávek

Upravit, či dokonce smazat objednávku, bude možné provést kdykoliv. Tuto akci nicméně může provádět pouze ošetřující lékař. Každá změna objednávky, v případě že se nejedná o změnu vyvolanou žádostí pacienta, musí mít evidovaný důvod změny. Po smazání objednávky budou všechny nezpracované žádosti o změnu této objednávky považovány za vyřešené.

Seznam všech objednávek

Při zobrazení všech objednávek se uživateli primárně zobrazí nejstarší nedokončené objednávky pro přihlášeného uživatele. U každé objednávky bude zobrazen počet nevyřízených žádostí k dané objednávce. Uživatel může hledat objednávky jednotlivých pacientů, případně ve zvoleném období. Taktéž je mu umožněno zobrazit objednávky i ostatních lékařů, nebo objednávky již dokončené.

Detail objednávky

Po vybrání objednávky z kalendáře, případně seznamu, budou zobrazeny základní informace o pacientovi, objednávce, o naplánovaných vyšetřeních apod. Také budou vypsány změny související s touto objednávkou a veškeré (nezpracované i zpracované) žádosti o změnu. V této sekci bude možné přidat závěrečnou zprávu z vyšetření a danou objednávku dokončit. Musí být možné objednávku i znovu otevřít. Dále bude možné přejít na kartu daného pacienta, a také znovu objednat pacienta z této objednávky. Veškeré úpravy a akce s danou objednávkou bude moci provádět pouze ošetřující lékař.

Zpracování žádostí

Žádost o přeobjednání/objednání, kterou pacient vytvoří skrze mobilní aplikaci, bude zobrazena jako nová zpráva. Lékař si danou zprávu zobrazí a provede případné přeobjednání/objednání, čímž bude žádost vyřízena. V přehledu žádostí pak lékař uvidí jednotlivé žádosti o objednání, přeobjednání, zrušení objednání a také již zpracované žádosti. Ke každé žádosti může lékař přidat odpověď pacientovi, která mu bude sdělena v mobilní aplikaci.

Zobrazení objednávek v daném dni

Na domovské stránce aplikace bude časová osa, na které budou vypsány jednotlivé objednávky pro daný den. Po kliknutí na libovolnou objednávku, bude uživatel přesunut na informace o aktuální objednávce (4.1).

4.2 Mobilní aplikace pro pacienty

Pacient bude mít možnost používat mobilní aplikaci, kde mu budou zobrazeny objednávky, které vytvořil lékař ve webové aplikaci pro jeho osobu. Navíc bude moci požádat o změnu některé již existující objednávky, nebo o vytvoření objednávky zcela nové.

Přihlášení do aplikace

Před prvním použitím aplikace bude uživatel nucen vyplnit své osobní údaje pro identifikaci v rámci informačního systému. Pro ulehčení vyplňování údajů bude mít uživatel možnost načíst údaje ze své kartičky pojištěnce, pomocí technologie OCR (tato technologie je blíže popsána a vysvětlena v 3). Uživatel si následně zvolí přihlašovací údaje do mobilní aplikace. Po úspěšném provedení zůstane pacient v aplikaci přihlášen, dokud se sám neodhlásí (4.2).

Nastavení účtu

V nastavení si bude moci uživatel změnit heslo účtu. Taktéž bude možné přidat svůj Google účet, který bude po autentizaci použit pro synchronizaci objednávek do Google kalendáře. Po přihlášení budou automaticky všechny nadcházející objednávky přidány do jeho kalendáře. Uživatel také bude mít možnost se z Google účtu odhlásit, čímž již nebudou vkládány další nadcházející objednávky tohoto pacienta do jeho kalendáře. V této sekci se bude moci pacient také z aplikace odhlásit.

Zobrazení objednávek

Na domovské stránce aplikace budou vypsané nadcházející objednávky seřazeny podle času. V další kartě pak bude možnost sledovat historii objednávek, které již proběhly. Jednotlivé objednávky si bude moci uživatel dále zobrazit, kde uvidí historii změn a žádostí. Taktéž bude moci z karty objednávky vytvořit novou žádost pro danou objednávku.

Zobrazení žádostí

V sekci žádostí si pacient bude moci zobrazit nezpracované, ale také již zpracované žádosti. Po rozkliknutí detailu zjistí stav žádosti, případně vyjádření lékaře, a také odkaz na danou objednávku.

Vytvoření žádosti o přeobjednání/objednání

Pacient může v rámci mobilní aplikace požádat lékaře o přeobjednání/zrušení stávající objednávky, nebo vytvoření objednávky nové. Pacient sdělí lékaři jeho preferovaný čas a odešle svou žádost. Jakmile bude žádost zpracována, oznámí se tato skutečnost pacientovi a případná změna objednávky bude zobrazena v přehledu objednávek.

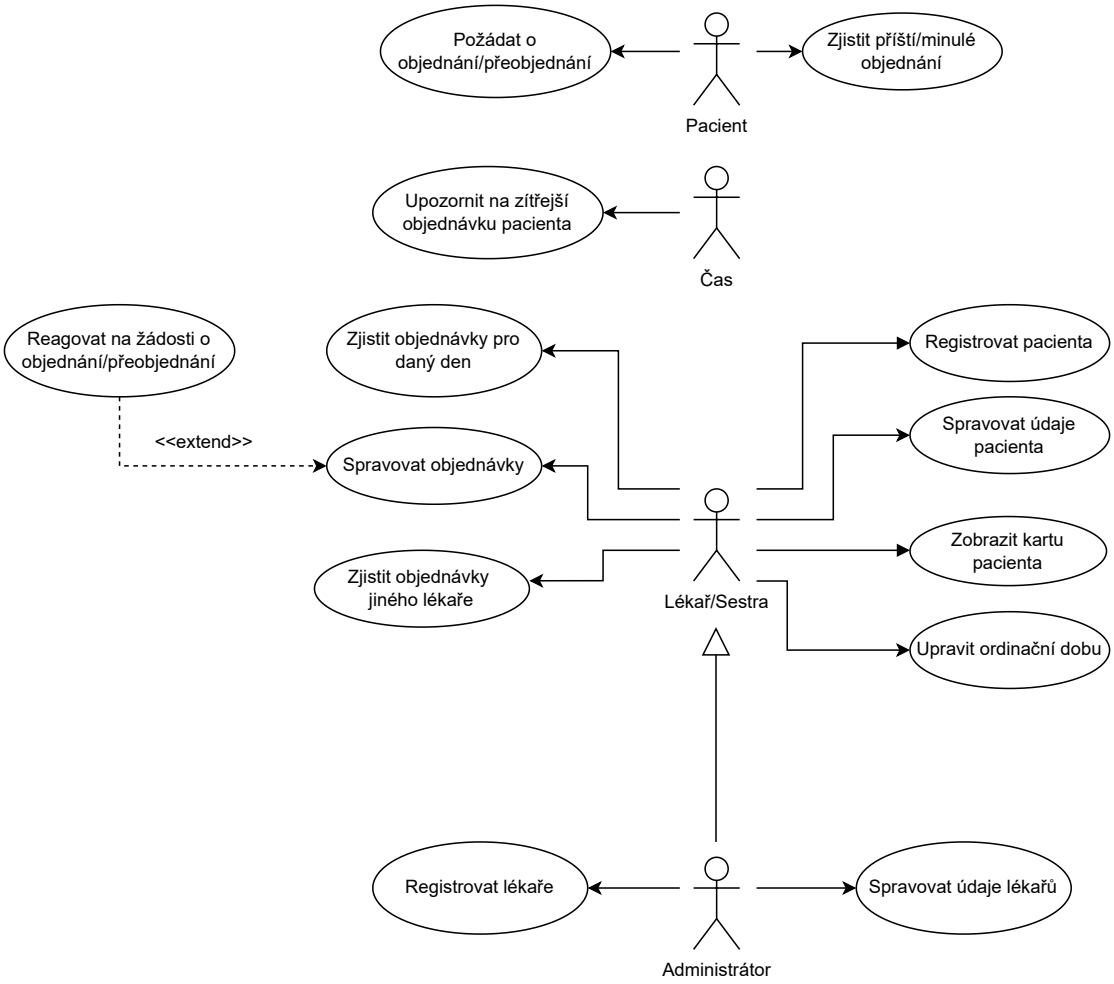
Upozornění na následující objednávku

Den před plánovanou objednávkou bude uživateli zobrazena notifikace v mobilu na připomenutí nadcházejícího vyšetření.

4.3 Diagram případů užití

Hlavním účelem tohoto diagramu je vizualizace jednotlivých použití v systému. Diagram případů užití nezobrazuje pouze jeden případ užití, ale více případů užití, které lze s daným systémem provést. Při jednotlivých případech užitích nám diagram sděluje, co lze provést, nikoliv jak to bude provedeno. [41]

V tomto systému budou figurovat 3 různé role (administrátor systému (viz podkapitola 4.1), uživatel systému (lékař/sestra) a pacient, který bude používat mobilní aplikaci (viz podkapitola 4.2)).



Obrázek 4.1: Diagram případů užití informačního systému

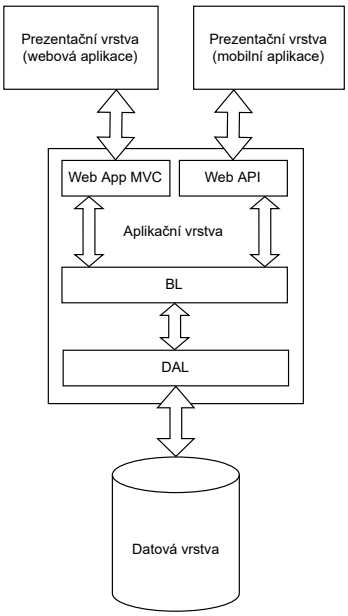
Kapitola 5

Návrh systému

Před samotnou implementací je nutné navrhnout systém podle zjištěných požadavků zákazníka. Veškeré požadavky byly specifikovány v kapitole 4. V této kapitole je tedy představena architektura informačního systému, návrh uživatelského rozhraní a datový návrh v podobě ER diagramu.

5.1 Architektura

Pro tvorbu informačního systému byla zvolena třívrstvá architektura (popsána v 2.1.1). V rámci této architektury budou vytvořeny 2 prezentační vrstvy. První prezentační vrstvou bude webová aplikace, druhou pak aplikace mobilní. Obě prezentační vrstvy budou komunikovat s aplikační vrstvou implementovanou pomocí frameworku ASP.NET. Přičemž webová aplikace bude využívat Web App MVC (5.1.1), zatímco mobilní aplikace Web API (5.1.2). Veškerá obchodní logika systému bude zapouzdřena do vrstvy BL. Aplikační server pak bude komunikovat s vrstvou datovou pomocí vrstvy DAL (Data access layer). Ta bude obsahovat objektově-relační mapování.



Obrázek 5.1: Architektura informačního systému

5.1.1 Webová aplikace

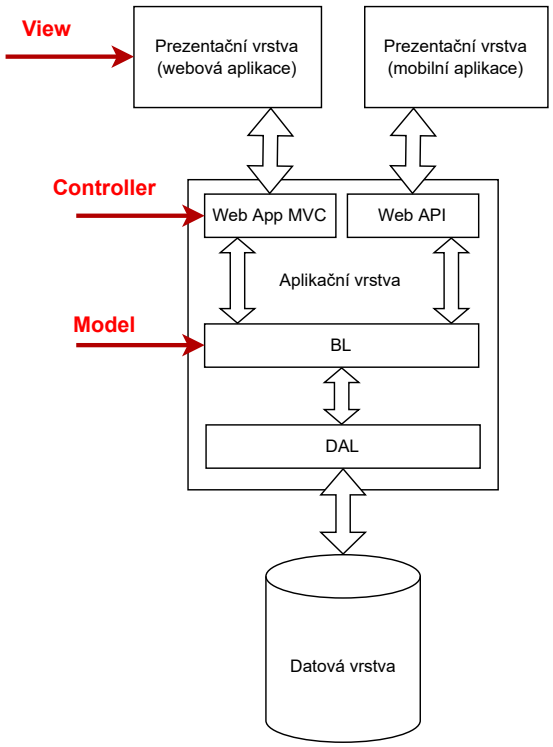
Jak již bylo zmíněno, webová aplikace bude implementována pomocí frameworku ASP.NET Web App MVC. Tento způsob implementace, jak již název napovídá, implementuje návrhový vzor MVC.

Model-View-Controller

Hlavním důvodem zavedení tohoto návrhového vzoru je oddělení logiky aplikace od uživatelského rozhraní. Celý kód je pak rozdělen do 3 částí: [42]

- Model – data aplikace. Komunikuje s databází (vkládá, upravuje, ...) a odpovídá na jednotlivé požadavky Controlleru.
- View – prezentace dat (co vidí uživatel). Vytváří jej Controller a naplňuje jej daty (tedy modely).
- Controller – propojuje Model a View. Zajišťuje jejich komunikaci a určuje, který View použít, následně jej naplní Modely.

Ve zvolené architektuře, pak MVC lze vidět zde:



Obrázek 5.2: MVC v architektuře informačního systému

Vrstva DAL na obrázku 5.2 využívá ORM, neboli objektově-relační mapování, kde jednotlivé entity odpovídají jednotlivým entitám ze sekce 5.2.

5.1.2 Mobilní aplikace

Tato prezentační vrstva nebude tenký klient jako v případě webové aplikace, ale klient tlustý. Bude totiž obsahovat veškerou funkcionalitu a získávat data z datové vrstvy pomocí Web API.

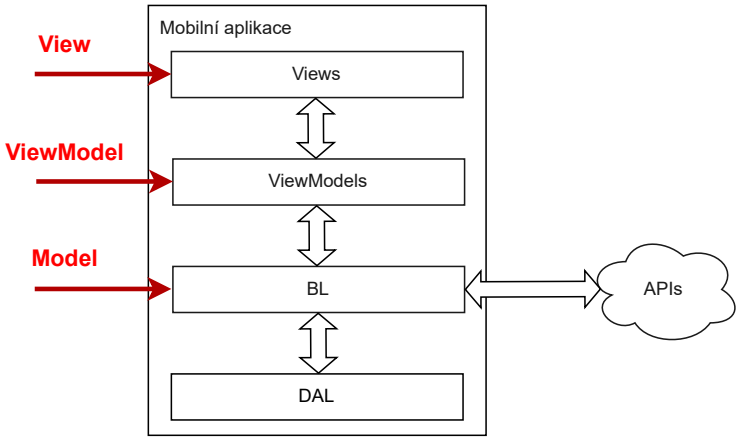
Jelikož se jedná o tlustého klienta, je vhodné taktéž oddělit logiku od uživatelského rozhraní, jako v případě webové aplikace. Pro implementaci mobilní aplikace byla vybrána technologie Xamarin.Forms (viz. 2.2.2), která umožňuje současný vývoj jak pro Android, tak iOS. Pro tuto technologii je nejvhodnější použít návrhový vzor MVVM.

Model-View-ViewModel

Tento návrhový vzor taktéž slouží k oddělení logiky aplikace od uživatelského rozhraní jako vzor MVC. Rozdíl oproti MVC je v tom, že ke každému View odpovídá pouze 1 ViewModel. Kód je tedy rozdělen do 3 částí: [43]

- Model – data aplikace. Komunikuje s databází nebo API a odpovídá na jednotlivé požadavky ViewModelu.
- View – prezentace dat (co vidí uživatel). Stav, data i uživatelskou interakci zpracovává ViewModel.
- ViewModel – předává Modely do View a upozorňuje View na změny stavu. Provádí přesměrování na ostatní ViewModely.

Výsledná mobilní aplikace s MVVM:



Obrázek 5.3: Architektura mobilní aplikace

Vrstva DAL na obrázku 5.3 využívá stejně jako v případě aplikačního serveru ORM. Ve vrstvě BL je mimo jiné také mnoho služeb, které mohou komunikovat s různými API. V tomto případě se bude jednat o Web API implementovaného informačního systému (5.1) a API pro Google Calendar.

Aplikační programovací rozhraní

Aplikační programovací rozhraní neboli API je sada definic a protokolů pro vytváření a integraci aplikačního softwaru. [44] Zjednodušuje vývoj softwaru a úpravy tím, že aplikacím umožňuje snadno a bezpečně vyměňovat data a funkce. Aplikační rozhraní umožní jiné službě nebo produktu komunikovat mezi sebou a vzájemně si vyměňovat svá data. Přičemž vývojář nepotřebuje vědět, jak je API implementováno, jednoduše použije toto rozhraní ke komunikaci s jiným produktem. [45]

V tomto případě se bude jednat o webové aplikační rozhraní, které bude veškeré dotazy dostávat na předem definované URL neboli *endpointy*. [46] Tyto endpointy budou vracet serializovaná data (5.1.2) klientovi, případně provedou patřičné operace pro úpravu databáze (např. vložení, aktualizace hodnot, smazání). Pro webová API vzniklo mnoho standardů jako XML-RPC, SOAP, REST, GraphQl. Tyto standardy definují různé syntaxe dotazů, syntaxe dat apod.

Serializovaná data

Serializovaná data jsou data, která jsou v takovém formátu, který umožňuje přenášení datových struktur nebo objektů přes internetovou síť. Existuje několik serializačních formátů, které se liší svojí syntaxí. [47]

XML je původně značkovací jazyk, který je snadno rozšiřitelný pro další aplikace. Byl jedním z prvních jazyků pro reprezentaci dat. [47]

JSON je člověku čitelný jazyk, který byl navržen především pro svou jednoduchost a univerzálnost. JSON data mohou být jednoduše zpracována v JavaScriptu voláním vestavěné funkce. [48]

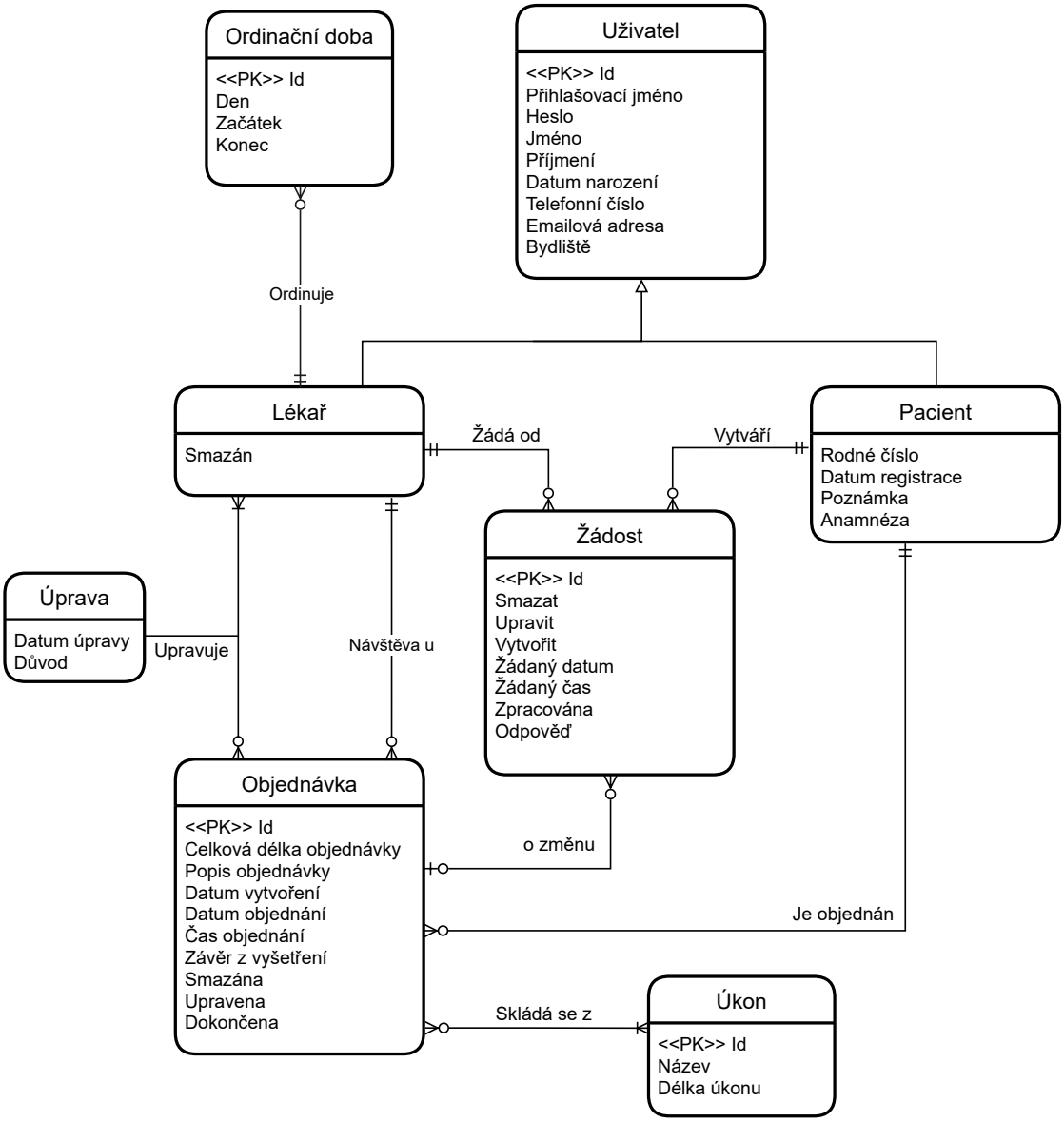
YAML je pro vývojáře více přívětivý svou čitelností, avšak je komplexnější. Dá se říci, že se jedná o nadmnožinu JSON. [48]

Podle porovnání a studie [49] je formát XML pro mobilní aplikace považován za nejhorší ze všech porovnávaných formátů. Z tohoto důvodu byl vybrán JSON.

5.2 ER diagram

Entity-relationship diagram neboli ERD, je konceptuální model, který se používá k popisu základních pojmů konkrétní domény, jejich struktury a vztahy mezi nimi. [50] Základními prvky ERD jsou:

- Entity – představují atomické koncepty nebo objekty v reálném světě.
- Atributy – představují jednotlivé prvky konkrétních entit.
- Vztahy – používají se k vyjádření asociace mezi jednotlivými entitami.
- Mohutnost/kardinalita – určuje počet instancí daného vztahu.



Obrázek 5.4: Datový model navrhovaného systému (ER diagram)

5.3 Návrh uživatelského rozhraní

Návrhu uživatelského rozhraní se často nevěnuje dostatečná pozornost. Právě tento návrh je ale důležitý pro to, aby uživatel považoval systém za uživatelsky přívětivý a celý systém tak rád používal.

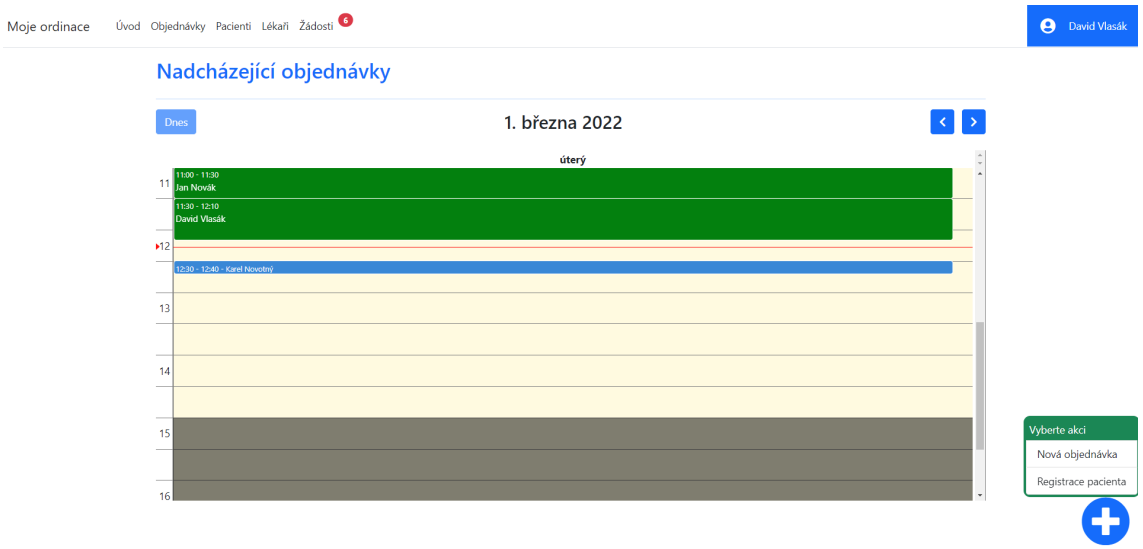
V této části se zaměříme jak na uživatelské rozhraní webové aplikace, tak na rozhraní aplikace mobilní.

5.3.1 Webová aplikace

Tato aplikace bude sloužit výhradně pro lékaře/sestry. Aby bylo vyhověno některým zvyklostem v oblasti vzhledu jednotlivých komponent, jsou použity předdefinované styly z knihovny Bootstrap[1].

Pro celou webovou aplikaci je navigační menu umístěné v horní části obrazovky. V této navigaci může uživatel pokračovat do jednotlivých sekcí.

U sekce žádosti se v navigaci zobrazuje počet nevyřízených žádostí. Předpokládá se totiž, že lékař bude pracovat primárně s úvodní obrazovkou, kde jsou zobrazena jednotlivá vyšetření, která se liší barvou podle stavu (viz. obr 5.5)[2]. Díky počtu upozornění tak lékař nemusí přerušovat svoji práci a sám si explicitně zjišťovat, zda nějaký pacient nepožádal o přeobjednání. Na úvodní obrazovce také může využít panelu rychlých akcí pro nejčastější operace. Tento panel se zobrazí až po kliknutí na tlačítko v pravém dolním rohu.



Obrázek 5.5: Náhled úvodní obrazovky

Navigace dále obsahuje další submenu, které se zobrazí po kliknutí na uživatelský účet. Na něm najde uživatel už nepříliš časté akce, jako úpravu svého účtu a správu své ordinační doby. Taktéž se v tomto submenu lze odhlásit ze systému.

[1] Bootstrap je knihovna již vytvořených HTML komponent, nadefinovaných CSS stylů a s tím i souvisejících JS funkcí. Více informací o této knihovně: getbootstrap.com

[2] Nedokončené objednávky jsou vyplněny modrou barvou, zelená pak znázorňuje vyšetření dokončená. Tyto barvy byly zvoleny po konzultaci se zaměstnanci již zmíněné kliniky.

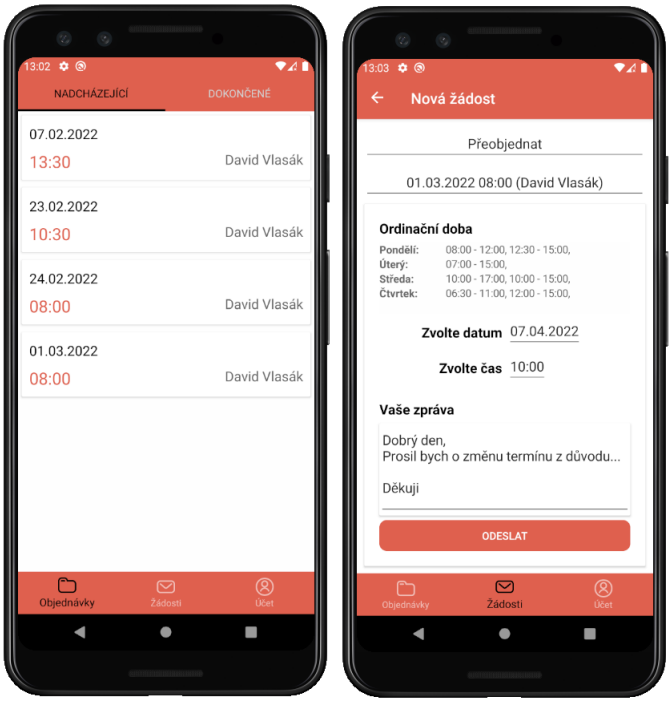
5.3.2 Mobilní aplikace

Tuto aplikaci naopak budou používat pacienti. Je tedy nutné dodržovat veškeré zvyklosti uživatelů mobilních aplikací, protože uživatelé nebudou nikterak seznámeni s ovládáním této aplikace někým jiným. Kvůli tomu tedy nelze například rozlišit objednávky podle barev jako u webové aplikace pro lékaře.

Pro zachování zvyklostí a snadnější orientaci je využita knihovna Material Design[3], ta je již součástí technologie Xamarin.

Mobilní aplikace má globální navigaci v dolní části obrazovky. Položky `Objednávky` a `Žádosti` obsahují navíc také místní navigaci, která odlišuje nadcházející a dokončené objednávky, respektive nevyřízené a vyřízené žádosti. Samotný přehled objednávek s místní navigací lze vidět na obr 5.6.

Druhý pohled na obrázku 5.6 je tvorba žádosti, což je jedna z nejdůležitějších funkcí mobilní aplikace. Uživatel si nejdříve zvolí, o jakou žádost se jedná, a následně k čemu se váže. Uživatel pak připojí preferovaný čas a případně i volitelnou zprávu.



Obrázek 5.6: Náhled mobilní aplikace

[3]Material Design je knihovna od společnosti Google, která přidává a stylizuje jednotlivé komponenty, upravuje rozložení obsahu a přidává různé animace a přechody. Více informací: material.io

Kapitola 6

Implementace

Tato kapitola pojednává o způsobu implementace informačního systému a o nejdůležitějších částech kódu. Implementace vychází z analýzy a specifikace požadavků, která je popsána v kapitole 4, a z návrhu systému z kapitoly 5.

6.1 Struktura projektu

Pro snazší pochopení celé implementace zde udávám i strukturu celého projektu, který byl vyvíjen pomocí prostředí Visual Studio 2022.

- **WebApp** – Projekt reprezentující aplikační vrstvu spolu s webovou aplikací. Je implementován technologií .NET 6.0 v jazyce C#. Skládá se z dalších projektů (vrstev).
  - **WebApp.DAL** – Projekt implementující ORM (objektově-relační mapování). Jedná se o entity s vazbami, které odpovídají datovému modelu z kapitoly 5.2.
  - **WebApp.BL** – Tento projekt obsahuje veškerou business logiku. Skládá se z Repository části, která provádí různé operace nad databází pomocí DAL entit. Dále z Modelů, které používají vrstvy vyšší, a v neposlední řadě z Mapperů, které převádějí jednotlivé entity na modely a opačně.
  - **WebApp.App** – Zde je již implementace samotné webové aplikace, která obsahuje jednotlivé Controllery a Views. Veškerá data získává a ukládá pouze pomocí vrstvy BL. Tato část navíc využívá frontendové technologie jako HTML, CSS a JavaScript.
  - **WebApp.API** – Vrstva, která implementuje webové aplikační rozhraní, na které se následně dotazuje mobilní aplikace. Tento projekt obsahuje své vlastní Controllery, které tvoří endpointy. Dále obsahuje své vlastní Modely, které jsou následně převáděny do JSON notace.
- **MobileApp** – Projekt, který implementuje mobilní aplikaci. Je implementován technologií Xamarin.Forms. Tento kód je sdílený pro obě platformy (iOS i Android). Tato aplikace obsahuje i vlastní lokální databázi. Díky tomu může aplikace zobrazovat údaje i při nedostupném internetu. Ty však nemusí byt aktuální.
  - **MobileApp.DAL** – Projekt implementující ORM pro lokální databázi. Taktéž se jedná o entity, jako v případě WebApp.DAL, nicméně bez vazeb, jelikož to daná technologie z důvodu výkonnosti nepodporuje.

  - **MobileApp.BL** – Vrstva zastřešující veškerou logiku mobilní aplikace. Obsahuje Database část, která provádí různé operace nad databází pomocí DAL Entit. Dále obsahuje modely a mappery obdobně jako v projektu WebApp.BL. Navíc tato část obsahuje Služby, které komunikují s jednotlivými aplikačními rozhraními (informační systém, Google API).
  - **MobileApp.App** – Vrstva, která z největší části implementuje návrhový vzor MVVM. Obsahuje totiž jednotlivé Views a ViewModely, navíc také různé sdílené balíčky pro Xamarin, jako Autentizaci pro Google OAuth, implementaci služeb na pozadí apod.

- **MobileApp.Android** – Tento projekt je již nativní aplikace pro Android, která využívá výše zmíněný sdílený kód. V této části se nachází inicializace a spuštění jednotlivých částí aplikace. Navíc obsahuje nativní funkce, jako zobrazení přihlašovacího okna pro Google přihlášení, nebo aktivaci služeb na pozadí.
- **MobileApp.iOS** – Tento projekt je naopak nativní aplikace pro iOS. Taktéž využívá sdílený kód a obsahuje své vlastní konstrukce aplikace. Tato část nicméně není implementována z důvodu nemožného vývoje iOS aplikací na OS Linux a Windows.

6.2 Aplikační server

Tato podkapitola popisuje část aplikačního serveru, kterou využívá jak webová aplikace, tak webové aplikační rozhraní. Zde jsou implementovány veškeré operace s databází, čímž dochází k zásadnímu zapouzdření.

6.2.1 DAL – Data Access Layer

V této části je implementováno objektově-relační mapování, které nám umožní automatickou konverzi mezi relační databází a objekty v tomto informačním systému. Jednotlivé implementované entity odpovídají již dříve zmíněnému ER diagramu (5.2). Dále je zde třída `DbContext`, ve které se ujasní jednotlivé vztahy mezi entitami.

Pro objektově-relační mapování byl využit balíček `Entity Framework`[1], kdy byl uplatněn přístup `Code-First` (inicializace databáze podle nadefinovaných tříd).

Entity

Každá entita dědí z abstraktní třídy `EntityBase`, která obsahuje identifikátor a sděluje Entity Frameworku, že se jedná o primární klíč, který si generuje databáze.

Příklad implementace třídy `OfficeHours`, která popisuje entitu `Ordinační doba` z 5.2.

```
public class OfficeHours : EntityBase
{
    public int Day { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
    public int DoctorId { get; set; }
    public Doctor Doctor { get; set; }
}
```

[1] Další informace zde: docs.microsoft.com/ef

Jednotlivé vazby mezi entitami obsahují vždy atribut značící cizí klíč (pro vkládání) a danou entitu (pro získání odkazované instance pomocí klíčového slova `include`).

Inicializace databáze

Pro inicializaci databáze, jak již bylo zmíněno v 6.2.1, je implementována třída `DbContext`, která v metodě `OnModelCreating(ModelBuilder)` upřesňuje jednotlivé vazby mezi entitami.

Z předešlého příkladu entita `OfficeHours` je pak definována takto:

```
modelBuilder.Entity<OfficeHours>(entity =>
{
    entity.HasOne(o => o.Doctor).WithMany(d => d.OfficeHours);
});
```

Následně byla vytvořena migrace příkazem `AddMigration`, čímž bylo vytvořeno schéma databáze zapsané C# notací. Pro použití schématu na databázový server je implementována třída `DesignTimeDbContextFactory`, kde se vybírá, o jaký SQL server se jedná. V tomto případě byl vybrán SQL server s připojovacím řetězcem ze souboru `appsettings.json`.

6.2.2 BL – Business Logic

V této části je popsána implementace vrstvy, která zapouzdřuje veškeré business operace. Obsahuje závislosti na vrstvu DAL, a naopak vyšší vrstvy obsahují závislosti už pouze na vrstvu BL.

Modely

Jelikož uživatel nepotřebuje znát veškeré informace o dané entitě z databáze a některé údaje jsou tak pro něj zbytečné, jsou implementovány modely, které obsahují už pouze podstatné informace.

Každý model dědí z abstraktní třídy `ModelBase`, ve které je identifikátor. Tento model ale již neobsahuje informace pro EF, že se jedná o primární klíč, to řeší vrstva nižší.

Uživatel informačního systému tedy pracuje s modely (vidí je, upravuje apod.), ty jsou následně mapovány na odpovídající entity, aby s nimi mohl EF pracovat. Pro mapování byl využit balíček `AutoMapper`[2].

AutoMapper

AutoMapper je přidán do závislostí aplikace při spuštění a je možné jej použít k mapování určitého objektu na objekt třídy `T` pomocí generické metody `Mapper.Map<T>(object)`. Pro konfiguraci mapování jsou použity příkazy `CreateMap<Tsource,Tdestination>()` ve třídě vycházející z abstraktní třídy `Profile`.

Repozitáře

Všechny potřebné operace s databází jsou implementovány v repozitářích. Ke každému modelu je vytvořen zvláštní repozitář, který obsahuje CRUD operace[3] pro danou entitu daného modelu.

[2] Další informace zde: docs.automapper.org

[3] Čtyři základní operace – create, read, update, delete

Metody pro čtení jsou často implementovány jako `GetBy`, aby dotaz selekce obsahoval SQL klauzuli `WHERE` při dotazu do databáze. Tím se zvyšuje výkonnost dotazů, jelikož BL vrstva předává pouze potřebné záznamy. Implementace jednotlivých dotazů je pak prováděna pomocí jazyka LINQ, který je integrován v .NET a umožňuje efektivní dotazování na data.

Kód metody `OfficeHoursRespository.GetByDoctorId()` je pak díky využití LINQ velice krátký a ideálně optimalizovaný.

```
public IEnumerable<OfficeHoursModel> GetByDoctorId(int id)
{
    using var dbContext = _dbContextFactory.CreateDbContext();
    return dbContext.OfficeHours
                    .Where(o => o.DoctorId == id)
                    .ToList()
                    .Select(_mapper.Map<OfficeHoursModel>);
}
```

V případě, že se do databáze vkládá (`Add()`), aktualizuje hodnota (`Update()`), nebo maže záznam (`Remove()`), jsou tyto změny uloženy pomocí metody `SaveChanges()`. Pro vkládání, úpravu a mazání je nutný parametr, který obsahuje instanci dané entity. Mazání záznamu v tabulce tedy nejdříve obsahuje získání záznamu z databáze a až poté jeho smazání.

Při použití metod `First()`, `Where()`, `Select()` apod. se do parametru přidá LINQ výraz. Ten bere jako parametr daný objekt, čímž lze přistupovat k jednotlivým atributům objektu.

6.3 Webová aplikace

Samotná webová aplikace je závislá na vrstvě BL. Využívá její modely a repozitáře pro práci s databází. Aby se mohla BL vrstva dotazovat do databáze, je nutné vytvořit objekt `DbContext` pro celou aplikaci.

Všechny instance objektů, které mají být dostupné pro celou webovou aplikaci (jako repozitáře, AutoMapper, DbContext apod.) jsou vytvářeny v souboru `Program.cs`, který obsahuje aplikačního buildera. Tomu je předán například `DbContext`, který využívá připojovacího řetězce ze souboru `appsettings.json`, což je konfigurační soubor aplikace. Dále je mu předán `AutoMapper`, jak již bylo zmíněno v 6.2.2, nebo také jednotlivé repozitáře. V neposlední řadě je zde vytvořena session, které je přiřazen timeout 12 hodin.

6.3.1 Autorizace

Za účelem autorizace byl vytvořen autorizační filtr. Ten musí povinně obsahovat implementaci metody `OnAuthorization()`, která zjistí, zda má daný uživatel právo k přístupu.

Metoda se podívá do session, zda obsahuje záznam s klíčem `user`, pomocí příkazu `Context.Session.GetString("user")`. V případě kladné odpovědi, jsou získaná data předána do `Context.Items`. Tato data se například používají k zobrazení navigace, kde je vypsáno jméno aktuálně přihlášeného lékaře. Pokud nastala negativní odpověď, je uživatel přesměrován na stránku s přihlašovacím formulářem.

Pomocí `[TypeFilter(typeof(AuthorizationLoggedInFilter))]` se filtr použije. Tato značka je vložena před třídy jednotlivých controllerů, což způsobí, že veškeré dotazy na tyto

controllery vyžadují autentizaci. V případě controlleru, který zpracovává přihlašovací formulář, je filtr vložen pouze před metody, které nezpracovávají dotaz na přihlášení. Kód filtru se vykonává vždy před provedením metody, na kterou dotaz směřuje.

Metody, které jsou dostupné pouze pro administrátora aplikace, jsou chráněny filtrem `AuthorizationAdministratorFilter`. Ten ze session zjistí, zda se jedná o administrátora.

6.3.2 Controllery

Každý controller ve svém konstruktoru získává instance jednotlivých repozitářů, které byly vytvořeny v `Program.cs`. Názvy veřejných metod pak odpovídají cestě dotazu (případně jsou upraveny pomocí `[ActionName("nazev")]`, kde parametry dotazu jsou parametry metod. Implicitně se jedná o dotazy typu GET. Pro jiné typy dotazů je metoda označena například pomocí `[HttpPost]` značící metodu POST.

K získání dat z těla dotazu je použit odpovídající model, který odpovídá schématu dat. Tyto modely jsou uloženy ve `WebApp.App/Model`. Například metoda pro tvorbu objednávky, která získává data z těla dotazu:

```
[HttpPost]
public JsonResult Create([FromBody] CreateOrder data)
```

Jednotlivé metody pak vrací `View` s libovolným modelem. K předávání dalších dat slouží `ViewBagy` a `ViewData`, jelikož nelze předávat více modelů najednou. Metody ale nevrací vždy `View`, mohou také vracet různá serializovaná data (typicky JSON).

Nyní následuje stručný popis jednotlivých controllerů:

- **DoctorsController** – Dotazy spojené s lékařem (např. seznam všech lékařů, správa lékařů a jejich účtů, objednávky ostatních lékařů, apod.).
- **HomeController** – Pro přihlašování a zobrazení domovské obrazovky.
- **OfficeHoursController** – Správa ordinačních hodin jednotlivých lékařů.
- **OperationsController** – Správa operací v objednávce (tvorba).
- **OrdersController** – Zobrazení seznamu objednávek, správa objednávek (tvorba, úprava, mazání, včetně reakce na žádosti).
- **PatientsController** – Správa pacientů.
- **RequestsController** – Zobrazení jednotlivých žádostí a získání počtu nezpracovaných žádostí zobrazeného v navigaci.
- **SchedulerController** – Získávání dat pro plánovací kalendář.

6.3.3 Views

Jednotlivá Views jsou seskupena do složek podle různých controllerů. Všechny tyto složky jsou ve složce `Views`. Každý view se skládá ze 2 částí: `Shared/_layout.cshtml` a jeho těla, který určí controller. View `_layout.cshtml` obsahuje veškeré importy knihoven v sekci `<head>` a také zobrazení horního menu. Tohle se netýka View pro přihlášení, to je definováno zcela samostatně, včetně všech importů pro danou stránku, jelikož není žádoucí zobrazovat menu.

Formuláře

Kód formulářů je obsažen v jednotlivých Views. Pomocí atributu `asp-action=""` se určí, jaká akce se na daném controlleru má provést. Následně díky využití modelů, jejichž třída je ve View nastavena pomocí příkazu `@model T`, lze dané vlastnosti modelu nastavovat přímo pomocí formuláře. Tím se jedná o formulář, který zašle metodu POST.

Příklad přihlašovacího formuláře:

```
<form asp-action="Login">
   <input asp-for="UserName" required/>
   <label asp-for="UserName">Uživatelské jméno</label>
   <input asp-for="Password" type="password" required/>
   <label asp-for="Password">Heslo</label>
   <input type="submit" value="Přihlásit"/>
</form>
```

V případě, že se do vstupních polí formuláře použije atribut `id=""`, jedná se již o formulář s metodou GET a jednotlivá data lze získat pomocí parametrů dané funkce. Těchto formulářů je využito například při filtrování objednávek, pacientů apod.

6.3.4 Důležité části kódu

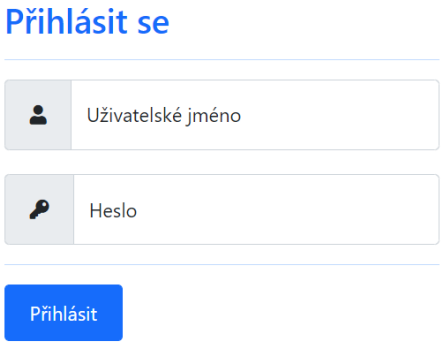
V této sekci jsou znázorněny a vysvětleny nejdůležitější části implementace webové aplikace.

Přihlášení uživatele

Jak již bylo zmíněno v 6.3.1, veškeré funkce webové aplikace jsou dostupné pro autorizované uživatele. Nejdříve tedy musí dojít k autentizaci uživatele. Ta využívá metodu `Login()` v controlleru `HomeController`. Tato metoda nejdříve zjistí, zda uživatel s daným uživatelským jménem existuje a následně porovná šifry hesel.

Při registraci uživatele není uloženo heslo v textové podobě, ale je vypočítána šifra ve funkci `Encrypt(string)` ze statické třídy `Security` ve vrstvě BL. Tato funkce vygeneruje tajný klíč, kterým se dané heslo zašifruje. Tato šifra je poté uložena do databáze. Při autentizaci uživatele se zadané heslo opět zašifruje a porovná se s šifrou z databáze. Stejný princip i metodu využívá také webové aplikační rozhraní pro účty pacientů.

Při úspěšné autentizaci se následně uloží údaje o uživateli (id, uživatelské jméno, jméno a příjmení) do aktuálního sezení. Na obrázku 6.1 je znázorněno přihlašovací okno.



Obrázek 6.1: Přihlašovací okno

Objednání pacienta

Jelikož se jedná o jednu z nejdůležitějších a nejsložitějších operací, je celá akce pomocí JavaScriptu (dále JS) rozdělena do několika kroků. V jednotlivých krocích se postupně vyplňují vstupní pole, která jsou následně odeslána controlleru. Kód je obsažen ve View `Orders/Create.cshtml`.

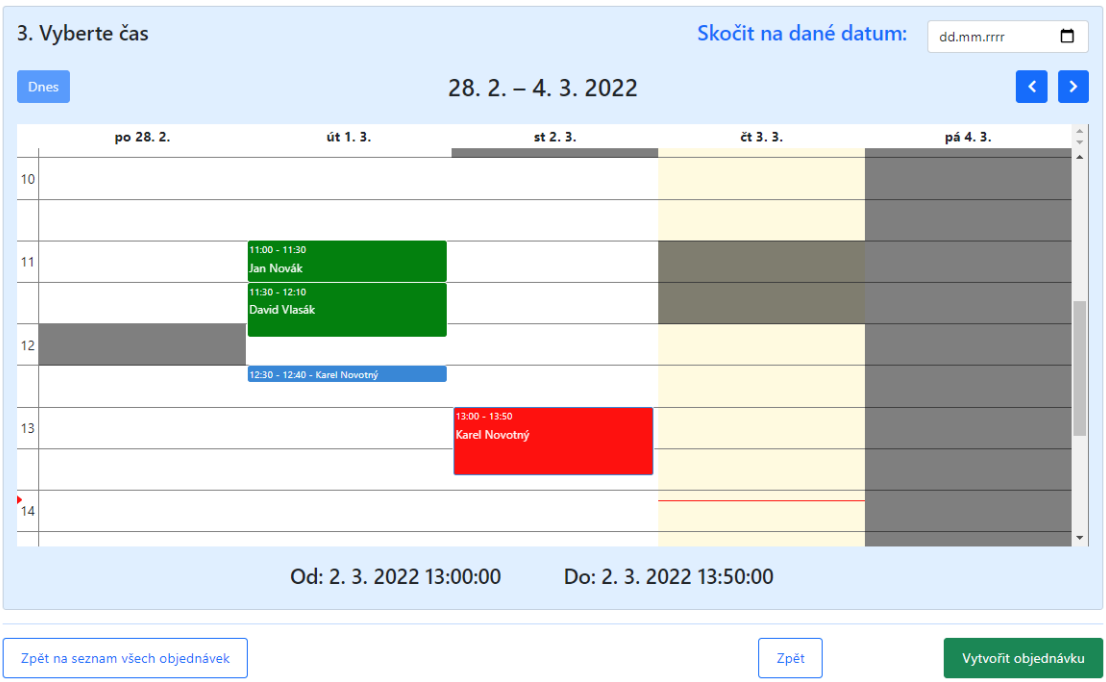
V prvním kroku je uživatel nucen vybrat pacienta. Pacient je automaticky vybrán, JS funkcí `GetPatient()`, pokud bylo zahájeno objednání z karty pacienta. Pro výběr pacienta slouží komponenta z knihovny `bootstrap-select`[4], která umožňuje v běžnému selectu navíc vyhledávání.

V dalším kroku uživatel vybere plánované operace, které budou provedeny. Navíc může vytvořit operaci novou, která bude pomocí asynchronního volání `ajax` vytvořena v databázi a přidána do aktuální objednávky. Po výběru alespoň 1 operace může uživatel přejít k vybrání časového slotu z kalendáře. Popis kalendáře je v části 6.3.4.

Po kliknutí na dané políčko v kalendáři je v modálním okně zobrazen vybraný čas a odhadovaná doba trvání. Po potvrzení výběru je možné objednávku vytvořit pomocí JS funkce `SaveOrder()`. Ta zašle asynchronní dotaz controlleru, který nejdříve uloží získaná data do sezení. Následně zjistí možné kolize objednávky (kolize s jinou objednávkou, mimo ordinační dobu, ...) pomocí privátní funkce `CheckCollision(OrderModel, int?)`.

V případě nepřítomnosti žádné kolize je volána funkce `CreateOk()`, která získá uložená data ze session a uloží je do databáze pomocí `OrderRepository`. Pokud byla nalezena kolize, je tato skutečnost oznámena uživateli. Uživatel i přesto může objednávku vytvořit pomocí JS funkce `CreateOk()`, která vyvolá stejně pojmenovanou funkci v controlleru. Ta vytvoří objednávku, která byla uložena do session před kontrolou kolizí.

Obrázek 6.2 znázorňuje výběr časového slotu v kalendáři.



Obrázek 6.2: Plánovací kalendář pro objednání

[4]Více informací zde: boostrap-select

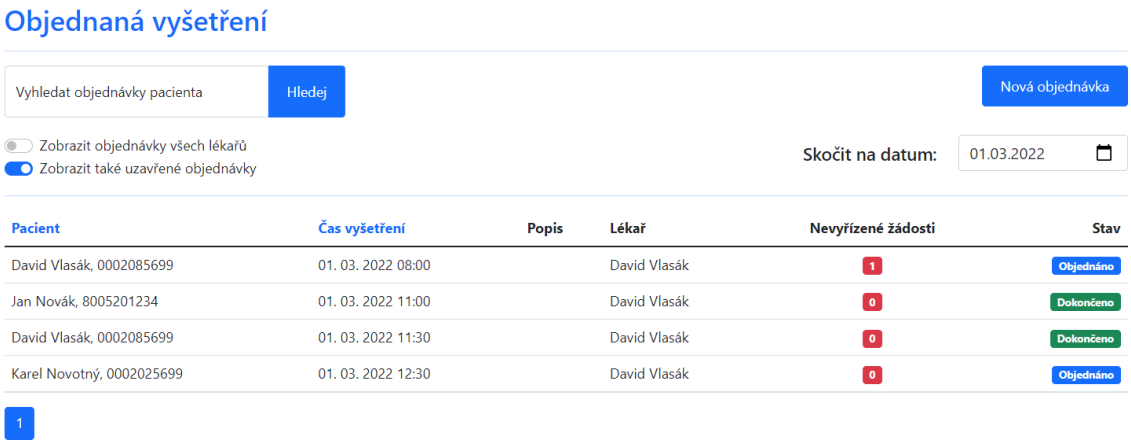
Podobným principem je implementována také úprava objednávky a reakce na žádost, která je při tvorbě/úpravě objednávky neustále zobrazena jako draggable okno. K tomuto posloužila knihovna `jQuery UI`[5].

Seznam objednávek

Seznam objednávek implicitně zobrazí všechna nedokončená vyšetření pro daného lékaře. Ta jsou zobrazena jako řádky tabulky. U každé objednávky je zobrazen čas, zkrácený popis, počet nevyřízených žádostí apod.

Zobrazení vyšetření lze nejrůzněji filtrovat. Zvolené filtry se zapisují jako parametry dotazu GET. Pro zpracování těchto filtrů slouží metoda `Index()`. Ta nejprve získá všechny objednávky a provede filtraci dat pomocí již zmíněných LINQ příkazů. Následně se vyfiltrují podle hledaného výrazu, seřadí se a převedou do číslovaného seznamu. Pro číslovaný seznam byl využit balíček `X.PagedList`[6].

Pro nastavené filtry podle obrázku 6.3 pak adresa vypadá takto:
`Orders?allDoctors=False&allOrders=True&date=2022-03-01`



Obrázek 6.3: Seznam objednávek s aktivovanými filtry

Stejným principem jsou pak implementována i zobrazení všech pacientů a lékařů. Tento způsob byl převzat z dokumentace ASP.NET MVC [51].

Správa žádostí

Jak již bylo zmíněno v 5.3.1, navigace zobrazuje počet nevyřízených žádostí. Zobrazení žádostí pro daného lékaře provádí metoda `RequestController.Index()`, která předá všechny žádosti pro daného lékaře. Samotné rozdělení žádostí podle typu je provedeno až ve View pomocí C# bloku kódu ohraničeného `@{ }`.

Tyto seznamy jsou poté pomocí smyčky `@foreach` zobrazeny v příslušných kategoriích. Každá nedokončená žádost je ohraničena ve značkách pro hypertextový odkaz (`<a>`), který směřuje na stránku pro vyřešení dané žádosti.

[5] Více informací zde: jqueryui.com
[6] Více informací zde: github.com/x.PagedList

Kalendář objednávek

Pro efektivní plánování a zobrazení objednávek (viz. obrázky 5.5 a 6.2) je využito JavaScriptové knihovny `FullCalendar`[7]. Jedná se o open-source knihovnu, která poskytuje rozhraní jak pro čistý JavaScript, tak JS frameworky (React, Vue, Angular).

Tento kalendář je využit při objednávání, úpravě objednání, či zobrazení následujících objednávek. Implementace je tedy ve Views `Doctors/Orders`, `Home/Index`, `Orders/Create` a `Orders/Edit`. Ve všech případech je implementace v JS funkci `InitCalendar()`.

Při tvorbě kalendáře pomocí `new FullCalendar.Calendar()` je do parametru vložen element, kam se má kalendář vložit, a také nastavení kalendáře. To obsahuje nastavení vzhledu, konfigurace triggerů po kliknutí na událost, případně prázdné políčko, a také nastavení relativní cesty pro získání dat do kalendáře. Kalendář pak na danou cestu zasílá asynchronní dotazy k zisku dat pro dané časové období, a následně odpověď také zpracuje a zobrazí.

Pro tuto funkci je implementována metoda `SchedulerController.GetEvent()`. Tato metoda vrací objekty kalendáře třídy `Event`, jejichž implementaci lze nalézt ve složce `WebApp.App/Models`. Třída obsahuje jednotlivé atributy pojmenované tak, jak je vyžadováno v dokumentaci knihovny. Z tohoto důvodu jsou použity vlastní mappery. Během mapování se také například nastavuje barva daného eventu podle toho, zda je již objednávka uzavřena (atribut `IsDone`).

Jestliže se jedná o úpravu nebo vkládání, je nutné danou událost v kalendáři odlišit. Z tohoto důvodu byl zaveden parametr `eventId`, podle kterého se změní barva dané události.

6.4 Webové aplikační rozhraní

Jak již bylo zmíněno dříve, mobilní aplikace získává veškerá data z informačního systému pomocí webového aplikačního rozhraní. Toto API je implementováno v projektu `WebApp.API` pomocí technologie ASP.NET 6.0 Web API. Je implementován standard REST.

REST API

Místo posílání požadavků na 1 endpoint, má endpointů více. Každý zdroj má vždy 1 koncový bod. Pro odlišení akce nad zdrojem (resp. endpointem) jsou využity metody HTTP protokolu. Pro získání dat se typicky využívá metoda `GET`, pro vytvoření metoda `POST`, úpravy `PUT` a mazání `DELETE`. Samozřejmě mohou být využity i metody další.

Po zaslání požadavku jsou k odpovědi využívány stavové kódy HTTP (200 Ok, 401 Unauthorized, 404 Not Found apod.). Při předávání dat jsou data serializována. V této implementaci je využita serializace pomocí formátu JSON.

6.4.1 Modely

Pro přenos dat jsou implementovány modely ve složce `WebApp.API/Models`. Tyto modely jsou následně serializovány do JSON a zasílány klientovi. Nejsou využívány modely z BL vrstvy, protože obsahují odkazy na mnoho dalších objektů. Jediným využitým modelem z BL je model `OfficeHours`, ten totiž neobsahuje žádný další objekt.

[7] Více informací zde: fullcalendar.io

6.4.2 Autorizace

Jelikož se jedná o API poskytující citlivé údaje, je nutná autorizace uživatelů. Pro autorizaci se využívá JSON Web Token (JWT)[8]. Ten je po zadání údajů na veřejný endpoint `/Authorize`, metodou POST, vygenerován a zaslán klientovi zpět. Tento endpoint slouží jak pro registraci, tak přihlášení. V těle metody POST musí být obsažena JSON data odpovídající třídě `Patient` s údaji. Dotazem do `PatientRepository` z BL vrstvy se zjistí, zda je uživatel již registrován. Pokud ano, ověří se přihlašovací údaje a v případě úspěšného ověření, je token navrácen. Pakliže uživatel není registrován, registraci provede API a uživateli taktéž vrátí token. Heslo je opět šifrováno pomocí metody `Security.Encrypt()` z kapitoly 6.3.1.

V datech tokenu je uloženo přihlašovací jméno uživatele. Token nemá omezenou časovou platnost. Generování tokenu je prováděno již ve zmíněném controlleru `Authorize` za použití balíčku `Authentication.JwtBearer`. Všechny zabezpečené endpointy/controllery pak obsahují značku `[Authorize]`. O ověření se pak stará použitý balíček.

6.4.3 Controllery

Jednotlivé controllery odpovídají jednotlivým endpointům. Každý endpoint pak pracuje s jedním zdrojem dat. V tomto případě to znamená s jedním repozitářem. Jeho instanci lze získat v konstruktoru controlleru, jelikož byl zaregistrován v souboru `Program.cs`. Každý controller dědí ze třídy `ControllerBase` jako v 6.3.2 a je mu nastavena cesta jeho endpointu.

Jednotlivé dotazy (resp. metody) jsou pak odlišeny podle metody HTTP protokolu, případně podle argumentů. Těla metod jsou krátká a typicky obsahují pouze práci s repozitářem, kde jsou jednotlivé modely mapovány. Mapování je prováděno opět pomocí AutoMapperu, kde jeho konfigurace je ve třídách `WebApp.API/Mappers`.

Následuje krátký popis jednotlivých endpointů podle controllerů:

- **AuthorizeController**
  - POST /Authorize – vygeneruje JWT pokud je úspěšná autentizace
- **DoctorsController**
  - GET /Doctors – vrátí seznam lékařů (včetně ordinační doby)
- **OrdersController**
  - GET /Orders/{id} – vrátí seznam všech objednávek daného pacienta (včetně smazaných, které ještě neproběhly)
- **PatientsController**
  - GET /Patients/personalId?personalId={value} – vrátí uživatele podle RČ
  - GET /Patients/userName?userName={value} – vrátí uživatele podle přihlašovacího jména
  - OPTIONS /Patients – ověří heslo uživatele (při změně hesla)
  - PATCH /Patients – upraví uživatele (změna hesla)

[8] Jedná se o internetový standard pro vytváření dat s volitelným podpisem. JWT obsahuje záhlaví, které identifikuje použitý algoritmus k podpisu. Následují data, která může vývojář svévolně využít, a na závěr obsahuje podpis (tajný klíč). Token pak má syntaxi `xxx.yyy.zzz`.

- **RequestsController**
  - GET /Requests/{id} – vrátí seznam všech žádostí daného pacienta
  - DELETE /Requests/{id} – smaže žádost s daným identifikátorem
  - POST /Requests – vloží novou žádost

6.5 Mobilní aplikace

Pro mobilní aplikaci je využita technologie Xamarin.Forms (popsána v kapitole 2.2.2). Aplikace získává data z webového aplikačního rozhraní. Pakliže mobilní zařízení není připojené k internetu, jsou data získávána z lokální databáze v daném zařízení. To tedy umožňuje omezené použití aplikace. Pro vytvoření nové žádosti je ovšem připojení vyžadováno.

6.5.1 DAL – Data Access Layer

V této části je implementováno objektově-relační mapování, obdobně jako v kapitole 6.2.1. Pro implementaci je využit balíček SQLite. Samotné připojení databáze není prováděno pomocí `DbContext` jako v 6.2.1, ale pomocí vytvoření objektu `Database` při inicializaci aplikace v `App.xaml.cs`. V konstruktoru je poté vytvořeno asynchronní připojení na databázi uložené v aplikačních datech aplikace.

Entity

Do databáze se ukládají pouze entity takové, které jsou nutné pro offline funkci. V lokální databázi jsou tedy uloženy pouze entity: `Order`, `Request`, `Update` a entita `User`, která poskytuje základní informace o přihlášeném uživateli. Jednotlivé entity pak dědí opět ze třídy `ModelBase`, která obsahuje identifikátor a značku primárního klíče pro SQLite `[PrimaryKey]`. Vazby entit jsou provedeny pouze pomocí cizích klíčů, nikoliv pomocí instance dané entity, z důvodu výkonnosti.

6.5.2 BL – Business Logic

Pro zapouzdření business operací je implementována vrstva BL (obdobně jako v 6.2.2). Ta navíc obsahuje i `Services`, které zabezpečují komunikaci s jednotlivými API.

Implementované modely se shodují s modely, které předává webové API. Navíc obsahuje modely pro Google Calendar API ve složce `GoogleCalendar`. Ty odpovídají modelům z [52]. Pro mapování modelů na entity již není použita knihovna AutoMapper, ale jsou implementovány mappery vlastní. Ty navíc často upravují nebo doplňují jednotlivé informace (například textový popis podle bool hodnot). Tyto mappery jsou implementovány v `Mappers`.

Přístup k jednotlivým tabulkám zajišťuje třída `Database`, která obsahuje jednotlivé části databáze odpovídající entitám. Tyto části obsahují CRUD operace, případně pouze operaci čtení. Pro aktualizaci dat z webového aplikačního rozhraní se používají metody `Reload(List<T>)`, které nejdříve tabulky vyčistí, a následně je naplní novými a aktuálními daty.

Services

Komunikaci s jednotlivými API provádějí služby. Ty jsou implementovány v `Services` a následně registrovány v `App.xaml.cs`, odkud jsou z `BaseViewModel` získány. Služby pro komunikaci s webovým API informačního systému dědí ze třídy `ServiceBase`, která udává adresu daného API a vytváří klienta pro HTTP dotazy. Ten je následně využit při každém dotazu, kde je mu nejdříve nastaven JWT token v případě, že se jedná o autorizovaný endpoint. Po přijetí odpovědi s kódem 200[9] se získaná data deserializují pomocí metody `JsonConvert.DeserializeObject<T>(data)`.

Služba pro komunikaci s Google Calendar API je implementována v `GoogleService`. Aplikace pak volá metodu `SyncEvents(string, List<OrderDetailModel>)`, která smaže zrušené objednávky a naopak přidá/aktualizuje objednávky do daného Google kalendáře. Při vkládání/aktualizaci je prováděno přemapování na objekty, které vyžaduje Google API. Operace se provádějí s primárním kalendářem (endpoint `calendars/primary/events`) pod identifikátorem s prefixem `mujlekarappv1orderid`.

Implementace této komunikace byla provedena pomocí dokumentace [52].

6.5.3 Sdílený kód

Projekt `MobileApp` obsahuje sdílený kód pro obě platformy. Při spuštění aplikace se vykoná konstruktor třídy `App.xaml.cs` a metoda `OnStart()`. V těchto metodách dochází k inicializaci databáze, registraci služeb a také tvorbě navigace. Nejdříve je ovšem zjištěno, zda byl uživatel již dříve přihlášen. To je uloženo v zabezpečeném úložišti `SecureStorage`.

Navigace v aplikaci je vytvořena pomocí Shell navigace. Jednotlivé cesty jsou registrovány pomocí `Routing.RegisterRoute()` v `AppShell.xaml.cs`. Navigace je vytvořena v `AppShell.xaml` ve značkách `<TabBar>`. Přesměrování pomocí této navigace je pak prováděno příkazem `Shell.Current.GoToAsync("cesta")`. Před přihlášením do aplikace (registrace apod.) se pro navigaci využívá `NavigationPage`.

ViewModels

Každý ViewModel dědí ze třídy `BaseViewModel`, který získá služby ze závislostí a obsahuje implementaci `PropertyChanged`, což umožní překreslení View po změně dat ve ViewModelu. Jednotlivé property ViewModelů jsou pak pomocí bindingu zobrazeny ve Views. Při navigaci na konkrétní stránku se pomocí bindingu zavolá daný ViewModel a případně se mu předají parametry z URI adresy přesměrování. Tyto parametry jsou pak získány pomocí `[QueryProperty()]`.

Views

Views jsou ve formátu `ContentPage`, kde typicky obsahují `Grid`, který obsahuje data a `ActivityIndicator`. Ten je nabindován na property z BaseViewModelu `IsBusy`. Jednotlivá data jsou pak zobrazena pomocí bindingu a akce vykonávány pomocí Commandů. Code-behind pak obsahuje již zmíněné nastavení BindingContextu, a také v některých případech úpravu metody `OnAppearing()`. Ta se vykoná vždy při zobrazení daného pohledu a zavolá určitou metodu z daného ViewModelu (typicky načtení dat).

[9] Kód značící úspěšný HTTP požadavek.

OCR

Pro implementaci OCR bylo vybráno řešení Azure Cognitive Service. Implementace je provedena v `CameraViewModel`. Pro načtení kartičky pojištěnce je využit `CameraView` z balíčku `Xamarin.CommunityToolkit`. To umožňuje přes živé zachycení kamery zobrazovat další informace (např. rámeček pro kartičku). Po vyfocení je volána metoda `Capture(byte[])`, která pomocí balíčku `Azure.ComputerVision` zašle OCR dotaz na daný endpoint. Odpověď je poté vrácena s typem `OcrResult` [53].

Jelikož jsou data takto předána, je nutné zjistit, kde se nalézají důležité informace. Při opakovaných testech se řádky s názvem políčka zachycují zřídka a není tedy možné určit, o jaké políčko se jedná. Pro získání rodného čísla a data narození posloužily regulární výrazy. Implementace je pak v metodě `ParseText()`, kde je využito LINQ výrazů. Jelikož není možné rozeznat jméno a příjmení podle regulárního výrazu, aplikace se dále dotáže na webové API k získání pacienta podle rodného čísla. Tím se zároveň ověří správnost data narození a také, zda již není pacient v mobilní aplikaci registrován.

Autentizace

Jak již bylo zmíněno v 6.5.3, při spuštění aplikace je nahlédnuto do úložiště `SecureStorage`, zda obsahuje záznam s klíčem `isLogged`. Pokud takový klíč existuje a hodnotu má `1`, znamená to, že uživatel byl již dříve autentizován a zobrazí se úvodní strana s následujícími objednávkami. Pokud klíč není nalezen, je uživatel vyzván k přihlášení, případně registraci pomocí OCR/ručního vypsání dat. Po vyplnění údajů k přihlášení/registraci se aplikace dotáže webového API. Tato komunikace je podrobněji popsána v 6.4.2.

Po úspěšném přihlášení/registraci jsou vymazána všechna data z lokální databáze a je přidán aktuální uživatel (kvůli údajům jako jméno, příjmení, id apod.). Dále se do úložiště `SecureStorage` uloží příznak, že je uživatel přihlášen (`isLogged`), a také se uloží JWT token pro komunikaci s webovým API.

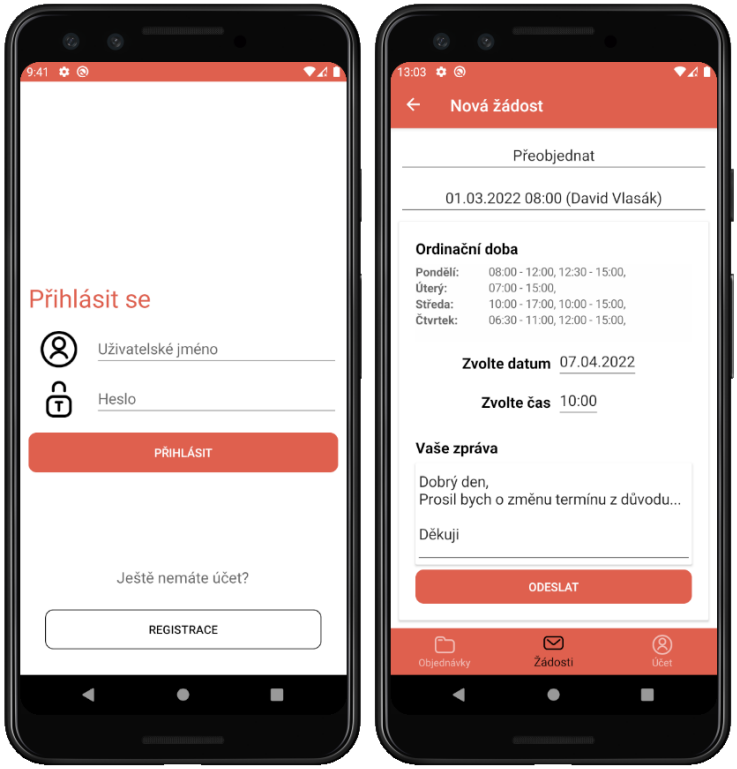
Levý pohled obrázku 6.4 ukazuje přihlášení uživatele do mobilní aplikace.

Nová žádost

Při vytváření nové žádosti se nejdříve zobrazí pouze výběr typu žádosti. Pro výběr typů je vytvořena třída `RequestType`. Po výběru typu se zobrazí příslušené okno výběru (lékař nebo objednávka). Viditelnost těchto prvků je nabindována na vlastnosti ViewModelu (`ShowOrders` a `ShowDoctors`). Pomocí tlačítka `Odeslat` je volána metoda `SendAction()`, která nejdříve zkontroluje, zda je žádost v ordinační době daného lékaře (neplatí při mazání objednávky). Následně je tato skutečnost zobrazena uživateli pomocí `DisplayAlert`, který i přesto může žádost zaslat.

V případě, že uživatel přejde na vytvoření žádosti přes již existující vyšetření, je využito parametru při přesměrování. Pomocí `[QueryProperty(OrderId, OrderId)]` se získá id objednávky a nastaví se bindovatelná property `SelectedOrder`. Tím se vyvolá metoda `OrderSelected()`, která nastaví i lékaře. Díky tomu se nejen po výběru přeobjednání/zrušení vyšetření zobrazí daná objednávka, ale také se při objednání zobrazí lékař, u kterého bylo původní vyšetření objednáno.

Pravý pohled obrázku 6.4 znázorňuje vytváření nové žádosti o přeobjednání.



Obrázek 6.4: Přihlášení uživatele do aplikace a vytvoření nové žádosti

Google služby

Pro komunikaci s Google službami je doporučována oficiální knihovna pro .NET[10]. Tato knihovna nicméně není podporována pro Xamarin a tudíž je nutné veškerou komunikaci implementovat. Samotná komunikace s Google kalendářem je součástí služby ve vrstvě BL (6.5.2). Pro fungování této komunikace je na platformě Google Cloud Platform vytvořen projekt, který umožňuje komunikovat s Google Calendar API. Pro tuto komunikaci je ovšem nutná autorizace uživatele.

Autorizace uživatele do Google služeb je provedena pomocí Google OAuth 2.0, kde je mobilní aplikace (název balíčku a certifikát) zaregistrována jako klient. To umožňuje této aplikaci se dotázat na daný OAuth a tím získat token pro komunikaci s Google Calendar API. Implementace těchto dotazů je v `Auth/Authenticator.cs`, kde v metodě `Auth()` je vytvořen dotaz pro autentizaci pomocí balíčku `Xamarin.OAuth2Authenticator`. Tomu jsou předány adresy a také rozsah práv podle [54]. V tomto případě postačil rozsah `.events`.

Samotné okno pro autentizaci je implementováno nativně (6.5.4). Po dokončení autentizace jsou token, refresh token a doba platnosti uloženy do úložiště aplikace, a při dalším dotazu je volána metoda `GetActiveToken()`. Ta v případě, že platnost tokenu vypršela, požádá o nový token pomocí refresh tokenu a vrátí tak vždy platný token pro komunikaci.

[10]Více informací o Google API pro .NET: github.com/googleapis/google-api-dotnet-client

6.5.4 Nativní funkce

Veškeré nativní funkce musí být implementovány pro obě platformy do daných projektů. V tomto řešení byly implementovány pouze nativní funkce Android platformy, jelikož Apple umožňuje vyvíjet iOS aplikace pouze na zařízeních Apple. Veškeré funkce, které jsou zde zmíněny, jsou tedy funkční pouze pro aplikaci na Android. iOS aplikace tedy bude také fungovat, ale bez těchto dalších funkcionalit.

V rámci nativních funkcí je vytvořeno již zmíněné autentizační okno pro Google Auth. To se aktivuje pomocí spouštěcího filtru, který odpovídá autentizaci Google Auth. Následně zavře okno autentizace a vrátí kontext zpět do mobilní aplikace.

Dále je pomocí nativních funkcí implementováno zobrazení upozornění i v případě, že aplikace neběží. Pro vytváření upozornění je vytvořena služba na pozadí, která se vždy aktivuje po určité časové době. K tomu slouží `AlarmManager`, který má nastavené opakování po 24 hodinách. Toto opakování však není naprosto přesné, aby se minimalizovalo buzení a spotřeba baterie.[11] Maximální garantovaná odchylka je 1 hodina. Kód spuštění časovače je ve třídě `Alarm`. Po uplynutí časového úseku je vyvolán přijímač `AlarmReceiver`, který vytvoří službu na pozadí. Kód této služby je ve sdíleném projektu ve třídě `BackgroundTask`. Ta v případě dostupného připojení k internetu provede na pozadí synchronizaci dat a následně z lokální databáze získá všechna vyšetření, která jsou naplánována na následující den. Tato vyšetření jsou následně uživateli pomocí balíčku `LocalNotification` připomenuta formou notifikace.

[11] AlarmManage nicméně umožňuje použití i naprosto přesných časovačů, avšak Google žádá Android vývojáře, aby přesné časovače používali pouze v nejnutnějších případech (budíky apod.).

Kapitola 7

Testování

Další podstatnou částí vývoje je testování. To zásadně ovlivňuje kvalitu celé vyvíjené aplikace. Díky testování jsou odhalovány nejen různé chyby, ale také nesrovnalosti s očekávaným výstupem zákazníka. V rámci této práce probíhalo testování jak v průběhu vývoje, které prováděl programátor, tak po dokončení vývoje, kterého se účastnili potenciální zákazníci.

7.1 Testování vývojářem

V průběhu implementace byl vyvíjený systém neustále testován. Po přidání nové funkcionality byla nová funkce testována a zdokonalována, dokud nebyla bezchybná. Po doladění více nových funkcionalit byla prováděna konzultace s potenciálním zákazníkem (lékařem), zda tento celek naplňuje jeho představy a očekávání. Následně bylo jeho připomínkám vyhověno a byly implementovány. Typicky se jednalo o různé zvyklosti, či zvýraznění nejčastějších akcí.

Tento druh testování probíhal lokálně. Pro lokální databázi byl využit nástroj `Microsoft SQL Server Management Studio`. Webovou aplikaci umožňoval otestovat vestavěný debugger `Visual Studia`. K testování webové API byl využit open-source framework `Swagger`, a také program `Postman`. Základní funkcionalita mobilní aplikace byla testována pomocí vestavěného emulátoru Android zařízení ve Visual Studiu, OCR funkce pak pomocí ladění na fyzickém zařízení.

7.2 Uživatelské testování

Po dokončení vývoje a řádném otestování vývojářem proběhlo uživatelské testování. Testování se účastnili vždy potenciální zákazníci. V případě webové aplikace bylo testování prováděno na klinice, která tento systém požadovala. Testování se zúčastnili 3 pracovníci této kliniky. Mobilní aplikaci pak testovali 3 potenciální pacienti. Účelem tohoto testování bylo nejen zjistit další skryté chyby v systému, ale hlavně zjistit uživatelskou přívětivost jak webové, tak mobilní aplikace.

Testování probíhalo na uživatelských zařízeních, kdy databáze, webová aplikace a API byly hostovány v cloudu pomocí platformy `Microsoft Azure`, kde bylo využito studentského předplatného. Mobilní aplikace pak byla vydána ve formě balíčku `apk` a zaslána jednotlivým uživatelům.

7.2.1 Webová aplikace

Pro webovou aplikaci byl vytvořen následující testovací scénář:

1. Nastavte si svou ordinační dobu.
2. Proveďte registraci nového pacienta.
3. Upravte anamnézu tohoto pacienta.
4. Pacienta objednejte na vyšetření.
5. Vytvořenou objednávku uzavřete a připište k ní závěr z vyšetření.
6. Zkontrolujte, zda Vám nepřišla nová žádost o změnu vyšetření.
7. Zpracujte tuto žádost – proveďte přeobjednání pacienta na požadovaný termín.
8. Zjistěte aktuální vytíženost lékaře Davida Vlasáka.

Každý uživatel webové aplikace byl před samotným testováním lehce seznámen s aplikací. Několik funkcí nebylo uživatelům vysvětleno vůbec, aby byla zjištěna intuitivnost systému. Uživatelé následně obdrželi testovací scénáře a bez dalšího napomáhání prováděli jednotlivé kroky. Během testování byl sledován čas nutný k provedení daného úkonu a také byly zapsány veškeré poznámky a připomínky.

Z výsledků z tabulky 7.1 si lze všimnout, že nejdéle trvalo objednání/přeobjednání pacienta (úlohy 4. a 7.). Tento výsledek byl očekáván, jelikož se jedná o nejsložitější operaci v tomto systému. Překvapivě déle trvalo nastavení ordinační doby (úloha 1.), kde uživatelé očekávali další potvrzovací tlačítko k uložení.

Úloha\Uživatel	**Uživatel A**	**Uživatel B**	**Uživatel C**
1.	2 min	2 min	1-2 min
2.	1-2 min	1-2 min	1 min
3.	ihned	ihned	1 min
4.	2 min	2 min	2 min
5.	ihned	1 min	ihned
6.	ihned	ihned	ihned
7.	2 min	2 min	2 min
8.	ihned	ihned	1 min

Tabulka 7.1: Tabulka trvání jednotlivých úkonů podle uživatelů.

Uživatelé se shodli, že jim velice vyhovuje způsob objednávání, kdy jsou jednotlivé úkony rozděleny do několika fází. Proces sice trvá o trochu déle než v nynějším systému, avšak díky jeho přehlednosti nevzniká tolik prostoru pro potenciální chyby. Celkově pak systém hodnotili jako velice intuitivní a moderně vypadající. To je zapříčiněno i faktem, že se svým aktuálním systémem jsou velice nespokojeni, jelikož je pro jejich práci příliš složitý, nepřehledný a zastaralý.

Vyjádření uživatelů:

- Uživatel A: Systém působí vizuálně jednoduše, lehký na ovládání, pěkný a čitelný.
- Uživatel B: Způsob objednávání je perfektní, přehledné a moderní. Po 1. použití uživatel umí ihned ovládat.
- Uživatel C: Systém hodnotím jako srozumitelný, přehledný a jednoduchý na ovládání.

7.2.2 Mobilní aplikace

Pro mobilní aplikaci byl vytvořen následující testovací scénář:

1. Proveďte registraci pomocí načtení kartičky pojištěnce (pokud neúspěšně, použijte ruční vypsání údajů).
2. Zjistěte Vaši nejbližší následující objednávku.
3. Požádejte o nové objednání u lékaře.
4. Tuto žádost následně zrušte.
5. Požádejte o přeobjednání Vaší nejbližší následující objednávky.
6. Propojte aplikaci s Vaším Google účtem a zkontrolujte Váš Google kalendář.

Žádný uživatel před testováním nebyl s mobilní aplikací nikterak seznámen. Reálnému uživateli taktéž nebude nikdy nic vysvětlováno. Uživatelé tedy pouze věděli, k čemu mobilní aplikace slouží. Po poskytnutí jejich soukromých údajů kvůli nutnosti registrace uživatele jako nového pacienta, jim byl předán `apk` balíček k nainstalování aplikace, a také již zmíněný testovací scénář.

Podle tabulky 7.2 vidíme, že ovládání aplikace je intuitivní, jelikož nejdelší trvání některého z kroků byla 1 minuta. U operací, které byly provedeny ihned, je značeno, zda provedení bylo úspěšné. Znak `"-"` značí, že tato část testovacího scénáře nebyla provedena z důvodu, že uživatel není vlastníkem Google účtu.

Úloha	**Otázka\Uživatel**	**Uživatel A**	**Uživatel B**	**Uživatel C**
1.	Povedlo se načíst kartičku?	Ano	Ne	Ano
2.	Povedlo se?	Ano	Ano	Ano
3.	Odhadovaná doba?	1 min	20 s	30 s
4.	Povedlo se?	Ano	Ano	Ano
5.	Odhadovaná doba?	1 min	10 s	1 min
6.	Povedlo se?	Ano	–	Ano

Tabulka 7.2: Tabulka vyjádření uživatelů.

Všichni tázaní uživatelé se shodli, že aplikace na ně působí jednoduše a přehledně. Uživatel B, kterému se nepovedlo načíst jeho kartičku pojištěnce, pak udává, že skenování prováděl za velice špatných světelných podmínek a uvítal by tak možnost použití blesku.

Vyjádření uživatelů:

- Uživatel A: Mobilní aplikace se mi velice líbila, oceňuji načtení kartičky pojištěnce a synchronizaci s Google kalendářem, který používám.
- Uživatel B: Kartička mi nešla načíst z důvodu velkého šera, uvítal bych možnost zapnutí blesku. Jinak se mi aplikace líbí.
- Uživatel C: Aplikace působí jednoduše a přehledně, vše jsem viděla a chápala na první pohled. Celkově velice snadné na ovládání.

Kapitola 8

Závěr

Cílem této bakalářské práce bylo navrhnout a následně implementovat informační systém pro lékaře s mobilní aplikací pro pacienty. Tento systém byl navrhnut pro zvolenou oční kliniku. Pro webovou aplikaci bylo důležité, aby bylo jednoduché nejen objednávání pacientů, ale také reagování na jejich žádosti o přeobjednání. U mobilní aplikace pak bylo nutné co nejrychlejším způsobem zobrazit nejbližší následující vyšetření a jednoduše vytvořit nové žádosti o objednání, případně přeobjednání. V rámci mobilní aplikace byl také kladen důraz na vhodné použití optického rozpoznávání znaků pro načtení kartičky pojištěnce. Zmíněné požadavky byly úspěšně implementovány a webová i mobilní aplikace jsou plně funkční.

Nejdříve byly v této technické zprávě popsány principy tvorby webových a mobilních aplikací spolu s nejpoužívanějšími technologiemi. K jednotlivým technologiím pak byla vybrána oblíbená vývojová prostředí. V další části byla studována problematika optického rozpoznávání znaků (OCR), kde byl popsán nejen princip této technologie, ale také použitelné existující nástroje.

V následující kapitole byla práce zaměřena již na samotnou analýzu a specifikaci požadavků systému, kde tato analýza byla prováděna ve spolupráci s již zmíněným očním centrem. Výsledky této analýzy vedly k vytvoření diagramu užití.

Z této analýzy pak byl učiněn návrh systému, který byl popsán v navazující části. Zde byl navržen ER diagram, výsledná architektura webové a mobilní aplikace, a také byl vytvořen návrh uživatelského rozhraní.

Po úspěšném návrhu systému následovala samotná implementace systému. V technické zprávě pak byla nejdříve popsána struktura projektu, aby se čtenář lépe orientoval a následně byly popsány jednotlivé části zvolené architektury. Závěrečnou fází vývoje bylo uživatelské testování, u kterého byly popsány jednotlivé testovací scénáře a získané výsledky.

8.1 Možné pokračování

Dalším možným pokračováním by bylo implementovat nativní funkce i pro mobilní aplikace iOS, jelikož nyní aplikace pro zařízení Apple obsahuje pouze základní funkcionalitu mobilní aplikace. K tomuto kroku by ale bylo nutné použít Apple zařízení. Díky tomu by pak aplikace pro iOS umožňovala použití Google kalendáře a zobrazování notifikací.

Dalším možným vylepšením by byla možnost použití blesku při načítání kartičky pojištěnce, jak zmínil účastník testování. Pro tuto funkcionalitu by bylo nutné opět použít nativní funkce jednotlivých platforem, tudíž by tento vývoj dával smysl pouze v případě, že by byla provedena implementace nativních funkcí také pro iOS.

Velice perspektivní a zajímavou možností dalšího vývoje by mohla být úprava systému pro více oddělení, ordinací a případně klinik. Tím by pacient získal přehled nad všemi svými navštěvovanými lékaři.

Literatura

[1] Berners Lee, T. J. a Cailliau, R. Worldwideweb: Proposal for a hypertext project. 1990.

[2] Jazayeri, M. Some trends in web application development. In: IEEE. *Future of Software Engineering (FOSE'07)*. 2007, s. 199–213.

[3] Joorabchi, M. E., Mesbah, A. a Kruchten, P. Real Challenges in Mobile App Development. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 2013, s. 15–24. DOI: 10.1109/ESEM.2013.9.

[4] ManagementMania.com. *Architektura klient-server (Client–server model)* [online]. [cit. 2021-11-28]. Dostupné z: https://managementmania.com/cs/architektura-klient-server.

[5] Oluwatosin, H. S. Client-server model. *IOSRJ Comput. Eng.* 2014, sv. 16, č. 1, s. 2278–8727.

[6] IBM Cloud Education. *Three-Tier Architecture* [online]. [cit. 2021-11-28]. Dostupné z: https://www.ibm.com/cloud/learn/three-tier-architecture.

[7] ManagementMania.com. *Třívrstvá architektura (Three-tier architecture)* [online]. [cit. 2021-10-17]. Dostupné z: https://managementmania.com/cs/trivrstva-architektura-three-tier-architecture.

[8] W3C. *HTML5 - A vocabulary and associated APIs for HTML and XHTML* [online]. [cit. 2021-11-29]. Dostupné z: https://dev.w3.org/html5/spec-LC/.

[9] W3C. *CSS Snapshot 2020* [online]. [cit. 2021-11-29]. Dostupné z: https://www.w3.org/TR/2020/NOTE-css-2020-20201222/.

[10] MDN contributors. *JavaScript* [online]. [cit. 2021-11-29]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[11] Tiobe. *TIOBE Index for November 2021* [online]. [cit. 2021-11-29]. Dostupné z: https://www.tiobe.com/tiobe-index//.

[12] Hutagikar, V. a Degde, V. Analysis of Front-end Frameworks for Web Applications. *International Research Journal of Engineering and Technology (IRJET)*. 2020, sv. 7, č. 4, s. 3317–3320.

[13] W3Techs. *Usage statistics of server-side programming languages for websites* [online]. [cit. 2021-12-04]. Dostupné z: https://w3techs.com/technologies/overview/programming_language.

[14] Welling, L. a Thomson, L. *PHP and MySQL Web Development*. Sams, 2003. Developer's library. ISBN 9780672325250. Dostupné z: https://books.google.cz/books?id=G4dTRyvpfhoC.

[15] Laaziri, M., Benmoussa, K., Khoulji, S. a Kerkeb, M. L. A Comparative study of PHP frameworks performance. *Procedia Manufacturing*. Elsevier. 2019, sv. 32, s. 864–871.

[16] Oracle. *JavaServer Pages Technology* [online]. [cit. 2021-12-04]. Dostupné z: https://www.oracle.com/java/technologies/javaserver-faq.html.

[17] Microsoft. *What is ASP.NET?* [online]. [cit. 2021-12-04]. Dostupné z: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet.

[18] Rosencrance, L. a Bigelow J., S. *Internet Information Services (IIS)* [online]. [cit. 2022-03-28]. Dostupné z: https://www.techtarget.com/searchwindowsserver/definition/IIS.

[19] MySQL. *Why MySQL?* [online]. [cit. 2021-12-04]. Dostupné z: https://www.mysql.com/why-mysql/.

[20] PostreSQL. *About PostgreSQL* [online]. [cit. 2021-12-04]. Dostupné z: https://www.postgresql.org/about/.

[21] Gorman, K., Hirt, A., Noderer, D., Rowland Jones, J., Sirpal, A. et al. *Introducing Microsoft SQL Server 2019*. Packt Publishing Ltd., 2019.

[22] StackOverflow. *Developer Survey 2019 - Most Popular Development Environments* [online]. [cit. 2021-12-05]. Dostupné z: https://insights.stackoverflow.com/survey/2019#development-environments-and-tools.

[23] Notepad++. *Notepad++ User Manual* [online]. [cit. 2021-11-29]. Dostupné z: https://npp-user-manual.org/docs/getting-started/.

[24] StatCounter. *Mobile Operating System Market Share Europe* [online]. [cit. 2021-12-05]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/europe.

[25] Shah, K., Sinha, H. a Mishra, P. Analysis of Cross-Platform Mobile App Development Tools. In: IEEE. *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*. 2019, s. 1–7.

[26] Chebbi, A. *Choosing the best programming language for mobile app development* [online]. [cit. 2021-12-05]. Dostupné z: https://developer.ibm.com/articles/choosing-the-best-programming-language-for-mobile-app-development/.

[27] Willocx, M., Vossaert, J. a Naessens, V. A quantitative assessment of performance in mobile app development tools. In: IEEE. *2015 IEEE International Conference on Mobile Services*. 2015, s. 454–461.

[28] AppleInsider. *XCode* [online]. [cit. 2021-11-29]. Dostupné z: https://appleinsider.com/inside/xcode.

[29] Chaudhuri, A., Mandaviya, K., Badelia, P. a K Ghosh, S. *Optical Character Recognition Systems for Different Languages with Soft Computing*. Cham: Springer International Publishing AG, 2016. Studies in fuzziness and soft computing. ISBN 3319502514.

[30] Chaudhuri, A. Some Experiments on Optical Character Recognition Systems for different Languages using Soft Computing Techniques. In: *Proceedings of SPIE*. Birla Institute of Technology Mesra, Patna Campus, Indie, 2010.

[31] Arica, N. a Yarman Vural, F. T. An overview of character recognition focused on off-line handwriting. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*. IEEE. 2001, sv. 31, č. 2, s. 216–233.

[32] Karandish, F. *The Comprehensive Guide to Optical Character Recognition (OCR))* [online]. [cit. 2021-11-19]. Dostupné z: https://moov.ai/en/blog/optical-character-recognition-ocr/.

[33] Cheriet, M., Kharma, N., Suen, C. a Liu, C.-L. *Character recognition systems: a guide for students and practitioners*. John Wiley & Sons, 2007.

[34] Chaudhuri, A., Mandaviya, K., Badelia, P. a Ghosh, S. K. Optical character recognition systems. In: *Optical Character Recognition Systems for Different Languages with Soft Computing*. Springer, 2017, s. 9–41.

[35] Nagy, G., Nartker, T. A. a Rice, S. V. Optical character recognition: an illustrated guide to the frontier. In: *Proceedings of SPIE*. SPIE, 1999, sv. 3967, č. 1, s. 58–69. ISBN 0819435856.

[36] Llobet, R., Cerdan Navarro, J.-R., Perez Cortes, J.-C. a Arlandis, J. OCR post-processing using weighted finite-state transducers. In: IEEE. *2010 20th International Conference on Pattern Recognition*. 2010, s. 2021–2024.

[37] Vincent, L. *Announcing Tesseract OCR* [online]. [cit. 2021-11-21]. Dostupné z: http://googlecode.blogspot.com/2006/08/announcing-tesseract-ocr.html.

[38] Malathi, T., Selvamuthukumaran, D., Chandar, C. D., Niranjan, V. a Swashthika, A. An Experimental Performance Analysis on Robotics Process Automation (RPA) With Open Source OCR Engines: Microsoft Ocr And Google Tesseract OCR. In: IOP Publishing. *IOP Conference Series: Materials Science and Engineering*. 2021, sv. 1059, č. 1, s. 012004.

[39] Saitwal, A. *5 Best OCR software for 2021* [online]. [cit. 2021-11-21]. Dostupné z: https://www.klearstack.com/5-best-ocr-softwares.

[40] Weil, S., Mayo, C., Wajer, M., Charles, L., Jake, S. et al. *TESSERACT(1) Manual Page* [online]. [cit. 2021-11-21]. Dostupné z: https://github.com/tesseract-ocr/tesseract/blob/main/doc/tesseract.1.asc.

[41] Siau, K. a Lee, L. Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML. *Requirements engineering*. Springer-Verlag. 2004, sv. 9, č. 4, s. 229–237. ISSN 0947-3602.

[42] Čápka, D. *MVC architektura* [online]. [cit. 2021-01-03]. Dostupné z: https://www.itnetwork.cz/navrh/mvc-architektura-navrhovy-vzor.

[43] Britch, D. *Enterprise Application Patterns using Xamarin. Forms.* Microsoft Press, A Division of Microsoft Corporation, One Microsoft Way . . . , 2017.

[44] RedHat. *What is an API?* [online]. [cit. 2021-11-29]. Dostupné z: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

[45] IBM Cloud Education. *Application Programming Interface (API)* [online]. [cit. 2021-11-29]. Dostupné z: https://www.ibm.com/cloud/learn/api.

[46] Mulloy, B. *Web API design.* Academic Press, 2013.

[47] Burget, R. *Vizualizace a serializace* [přednáška]. 2021.

[48] Eriksson, M. a Hallberg, V. Comparison between JSON and YAML for data serialization. *The School of Computer Science and Engineering Royal Institute of Technology.* Citeseer. 2011, s. 1–25.

[49] Sumaray, A. a Makki, S. K. A comparison of data serialization formats for optimal efficiency on a mobile platform. In: *Proceedings of the 6th international conference on ubiquitous information management and communication.* 2012, s. 1–6.

[50] Della Penna, G., Marco, A. D., Intrigila, B., Melatti, I. a Pierantonio, A. Interoperability mapping from XML schemas to ER diagrams. *Data knowledge engineering.* AMSTERDAM: Elsevier B.V. 2006, sv. 59, č. 1, s. 166–188. ISSN 0169-023X.

[51] Dykstra, T., Vega, D. a Parente, J. *Tutorial: Add sorting, filtering, and paging with the Entity Framework in an ASP.NET MVC application* [online]. [cit. 2022-03-03]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/sorting-filtering-and-paging-with-the-entity-framework-in-an-asp-net-mvc-application.

[52] Google. *API Reference* [online]. [cit. 2022-03-07]. Dostupné z: https://developers.google.com/calendar/api/v3/reference.

[53] Microsoft. *OcrResult Class* [online]. [cit. 2021-03-14]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/api/microsoft.azure.cognitiveservices.vision.computervision.models.ocrresult?

[54] Google. *Authorizing Requests to the Google Calendar API* [online]. [cit. 2022-03-08]. Dostupné z: https://developers.google.com/calendar/api/guides/auth.

Příloha A

Obsah přiloženého paměťového média

- **/xvlasa16/instalace.pdf** – instalační pokyny
- **/xvlasa16/zprava.pdf** – technická zpráva v PDF
- **/xvlasa16/zprava/** – zdrojové soubory technické zprávy
- **/xvlasa16/src/** – implementace informačního systému
  - **WebApp/wwwroot/lib/** – adresář obsahující použité knihovny, které nejsou dílem autora této práce