Webová aplikace pro výdejnu zdravotnických pomůcek pro diabetiky

Anna Hana Pilková

Bakalářská práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Anna Hana Pilková**
Osobní číslo:	**A20524**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Webová aplikace pro výdejnu zdravotnických pomůcek pro diabetiky**
Téma práce anglicky:	**Web Application for Dispensary of Medical Supplies for Diabetics**

Zásady pro vypracování

1. Vypracujte literární rešerši na zadané téma.
2. V rešerši proveďte také analýzu současných řešení podobných webových aplikací.
3. Navrhněte webovou aplikaci pro výdejnu zdravotnických pomůcek dle požadavků společnosti DiaTop.
4. Vytvořte webovou aplikaci dle návrhu.
5. Věnujte pozornost zabezpečení aplikace.
6. Implementaci webové aplikace vhodně popište.

Forma zpracování bakalářské práce: **tištěná/elektronická**
Jazyk zpracování: **Slovenština**

Seznam doporučené literatury:

1. WILLIAMS, Nicholas S. Professional Java for Web Applications. Hoboken, USA: John Wiley, 2014. ISBN 978-1-118--90931-7.
2. DARWIN, Ian F. Java Cookbook. 4. Sebastopol, USA: O'Reilly Media, 2020. ISBN 978-1-492-07258-4.
3. ALISKAN, Mert, Kenan SEVINDIK, Rod JOHNSON a Jürgen HÖLLER. Beginning Spring. Hoboken, USA: John Wiley, 2015. ISBN 978-1-118-89311-1.
4. FREEMAN, Adam. Pro Angular: Build Powerful and Dynamic Web Apps. Berkeley, USA: Apress, 2022. ISBN 978-1--4842-8176-5.
5. STUTTARD, Dafydd a Marcus PINTO. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. 2. Hoboken, USA: John Wiley, 2011. ISBN 978-1-118-17524-8.
6. MILLETT, Scott a Nick TUNE. Patterns, Principles, and Practices of Domain-Driven Design. Sebastopol, USA: John Wiley, 2015. ISBN 978-1-118-71469-0.

Vedoucí bakalářské práce: **Ing. Tomáš Vogeltanz, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **2. prosince 2022**
Termín odevzdání bakalářské práce: **26. května 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářské práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne ..... Anna Hana Pilková, v.r. .....

podpis studenta

ABSTRAKT

Bakalárska práca sa zaoberá návrhom a tvorbou webovej aplikácie pre výdajňu zdravotníckym pomôcok pre diabetikov. Je zameraná na analýzu podobných riešení, návrh a samotnú implementáciu webovej aplikácie. V teoretickej časti sa sústreďuje na technológie, ktoré sú použité pri imlementácii a porovnanie kandidátov technológií, medzi ktorými bolo vyberané. V praktickej časti sú podrobne analyzované podobné riešenia webových aplikácií, ich funkcionalita a výzor. Tie sú brané do úvahy pri návrhu samotnej webovej aplikácie. Návrh zohľadňuje požiadavky zadávateľa, ktoré následne spracováva. Súčasťou je implementácia podľa návrhu a jej podrobný popis.

Klíčová slova: Spring Boot, WebFlux, SMTP, MySQL, Redis, Angular, Angular Material Design, webová aplikácia

ABSTRACT

This bachelor thesis examines the design and development of a web application for a healthcare supplier for diabetic patients. It focuses on the analysis of similar solutions, their design and implementation of the web applications. The theoretical part discusses and compares the candidate technologies used in the implementation part. The practical part deeply analyses similar solutions, their functionality and layout. These, as well as the healthcare supplier's requirements, are then taken into consideration in the implementation itself. The detailed description of the implementation is also included.

Keywords: Spring Boot, WebFlux, SMTP, MySQL, Redis, Angular, Angular Material Design, web application

Chcela by som sa poďakovať spoločnosti Diatop za možnosť pracovať na tomto projekte. Taktiež poďakovanie patrí môjmu vedúcemu bakalárskej práce Ing. Tomášovi Vogeltanzovi, Ph.D. za venovaný čas a pochopenie. Veľká vďaka patrí môjmu priateľovi za podporu a pomoc pri tvorbe tejto bakalárskej práce.

OBSAH

ÚVOD

Pre udržanie zákazníka a kroku s dobou, firmy siahajú po moderných riešeniach, práve z oblasti IT. Komunikácia so zákazníkom, či objednávanie tovaru pomocou webových aplikácií, je v súčasnosti už zaužívaným spôsobom, ktorý firmy ponúkajú. Rovnako to platí aj pre spoločnosti zamerané na výdaj liekov, či zdravotníckych pomôcok. Tento spôsob uľahčuje komunikáciu so zákazníkom, ako aj poskytnutie príjemnejšieho zážitku spojeného s objednávaním tovaru. Či už vďaka možnosti objednať tovar z pohodlia domova, alebo ponuky ďalších rozšírených služieb prostredníctvom webovej aplikácie.

Cieľom tejto bakalárskej práce je vytvoriť webovú aplikáciu pre objednávanie zdravotníckych pomôcok pre diabetikov, ktorá zároveň ponúka možnosť čítania článkov práve z tejto oblasti. Jej úlohou bude umožniť zákazníkovi výdajne registrovať sa a vďaka tomu objednávať materiál, ktorý mu bol predpísaný jeho ošetrujúcim lekárom vo forme receptu. V rámci tejto webovej aplikácie prihlásený užívateľ môže čítať najnovšie články pridané výdajňou o trendoch, či kategorizácii zdravotníckych pomôcok pre diabetikov. Zároveň umožňuje pracovníkom výdajne nahrávať tieto dokumenty priamo v prostredí webovej aplikácie.

Práca je rozdelená do dvoch základných častí. Prvá časť je zameraná na teoretickú rovinu bakalárskej práce. Sú tu rozobrané webové technológie použité v tejto práci, ako aj ich porovnanie, či zhodnotenie ich výhod, prípadne nevýhod.

Druhá časť je orientovaná na praktický aspekt bakalárskej práce. Analyzuje podobné riešenia webových aplikácií a výsledky analýzy zhodnocuje v samotnom návrhu webovej aplikácie. Ten spĺňa požiadavky zadané zákazníkom a predstavuje riešenie webovej aplikácie z pohľadu modelov a diagramov. V záverečnej časti je venovaná pozornosť samotnej implementácii webovej aplikácie, ako aj ukážkam z jej užívateľského rozhrania.

I. TEORETICKÁ ČÁST

1 WEBOVÉ TECHNOLÓGIE

1.1 Spring

Spring, celým názvom **Spring Framework**, je softwarovou knižnicou založenou na vkladaní závislostí [1]. *Vznikol v roku 2003 a jeho autorom je Rod Johnson* [2]. Zameriava sa na tvorbu Java aplikácií a práve pomocou vkladania závislostí umožňuje organizovať veľké Java projekty tým, že externalizuje vzťahy medzi objektami v rámci aplikácie. Veľkou výhodou je jeho pokrokovosť a neustála aktuálnosť. Spring tím sleduje najnovšie trendy a tomu prispôsobuje a vyvíja ďalšie knižnice. [1] Spring je rýchly, flexibilný, produktívny, bezpečný a ponúka obrovskú podporu. Má množstvo návodov, ako začať pre úplných začiatočníkov, ale aj zložitejšie návody, komunitu a otvorené repozitáre, kde je možné sledovať najnovšie zmeny a diskusie k nim. [3]

Spring ponúka mnoho možností a knižníc, ktoré je možné pri tvorbe aplikácie využiť. Spring predkladá skutočne mnoho možností, hlavnými však sú **mikroslužby**, **asynchrónny neblokujúci prístup**, **webové aplikácie**, či **Cloud**. [3]

Jednoducho, Spring robí vývoj v Java rýchlejší, jednoduchší a bezpečnejší [3]. Práve preto bol vybraný ako základ webovej aplikácie tejto bakalárskej práce. Jeho flexibilita a zároveň množstvo predpripravených knižníc z neho robia základný stavebný kameň na tvorbu modernej, zabezpečenej a elegantnej webovej aplikácie.

1.2 Spring Boot

Spring Boot je novým ponímaním Spring (Spring framework) [1]. Je softwarovou knižnicou, ktorá ponúka programátorom základ na vybudovanie webovej aplikácie založenej na Spring a Java. [4] S použitím závislostí je možné využiť už implementované konfigurácie, metódy, či triedy, bez nutnosti definovať časti kódu odznova a bez zmeny. Základnými, jednými z najdôležitejších a najpoužívanejších prvkov Spring Boot sú Spring Boot starters či Spring Boot autoconfiguration.

Spring Boot starters je označením pre balíček modulov, s ktorými je možné jednoducho a rýchlo začať tvoriť funkčnú aplikáciu. Obsahujú predkonfigurované závislosti na najpoužívanejšie knižnice.

Spring Boot autoconfiguration disponuje komplexnou konfiguráciou, ktorá je nutná pre správnu funkčnosť Spring aplikácie. Automaticky nastavuje konfiguráciu napríklad podľa závislostí použitých v triedach.

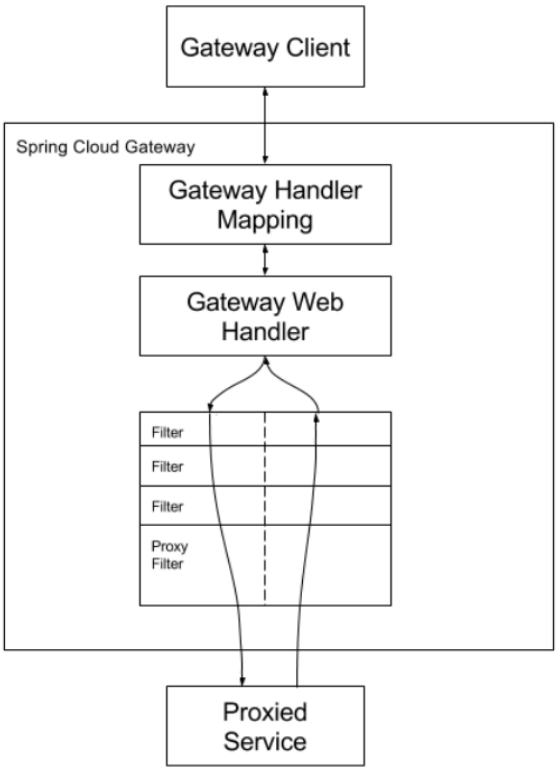
Ďalšími prvkami sú konfigurácie na jednoduchšie nasadenie aplikácie do produkcie, alebo konfigurovanie vlastností aplikácie, či jednotlivých jej častí. [4]

1.2.1 Spring Cloud Gateway

Pri tvorbe webovej aplikácie založenej na Spring s architektúrou mikroslužieb, Spring ponúka vlastné riešenie gateway, ktoré je priamo určené na tvorbu takéhoto typu aplikácií.

Gateway je takzvanou "bránou" do webovej aplikácie. Umožňuje kontrolu vstupu užívateľa do rôznych častí aplikácie ako sú služby, biznis logika, funkcionalita, či ďalšie servery nazývané aj mikroslužby. Taktiež poskytuje filtrovanie, transformáciu a zabezpečenie požiadaviek. Rovnako môže poskytovať autentifikáciu užívateľa, alebo prenechať túto starosť inej časti aplikácie, na ktorú požiadavku len presmeruje.

Spring Boot Cloud Gateway je API Gateway, prijíma API volania a s použitím služieb (*service*) vráti požadovanú odpoveď, výsledok na dané volanie. [5] Po odoslaní požiadavky klientom na Spring Cloud Gateway sa spustí reťazec reakcií zobrazený na obrázku 1.1.



Obr. 1.1 *Proces spracovania požiadavky od klienta v Spring Cloud Gateway* [6]

Prvá vec, ktorá sa vykoná je overenie, či daná požiadavka od klienta zodpovedá jednej z definovaných trás. O túto časť sa stará Gateway Handler Mapping a to tak, že priradí prichádzajúcu požiadavku na smerovací kľúč. Smerovací kľúč odpovedá kombinácii hostiteľa, portu a cesty. Tento smerovací kľúč je následne použitý k určeniu cieľovej

mikroslužby, kam bude požiadavka presmerovaná. Pokiaľ požiadavka spĺňa kritéria, je ďalej odovzdaná na spracovanie, ktoré zabezpečuje Gateway Web Handler. Ten požiadavku nechá prejsť pre ňu špecifickými filtrami. Tieto filtre môžu vykonávať logiku pred a po odoslaní požiadavky. Najskôr sa vykonáva takzvaná "pred" filtrová logika. Potom sa vytvára proxy požiadavka. Po vytvorení proxy požiadavky sa spúšťa takzvaná "po" filtrová logika.

Jednou z hlavných výhod a zároveň nevýhod Spring Cloud Gateway je jej využitie Netty servera a Spring WebFlux, ktoré umožňujú neblokované, asynchrónne spracovanie požiadaviek od klienta. To má za následok, že spolu so Spring Cloud Gateway, nie je možné využiť blokované synchrónne knižnice.
Obecne Spring Cloud Gateway funguje ako vstupná brána pre mikroslužby, ktorá poskytuje možnosti smerovania, filtrovania, transformácie a zabezpečenia požiadavky. [6]

1.2.2 Spring WebFlux

Pokiaľ je aplikácia reaktívna, či je nutné nastaviť napríklad zabezpečenie v Spring Cloud Gateway, Spring uviedol samostatnú softwarovú knižnicu, ktorá pokrýva práve tieto požiadavky.

Spring Webflux, v minulosti súčasťou Spring MVC, je alternatívou práve k spomínanému Spring MVC. Kým Spring MVC sa používa na tvorbu aplikácií, ktoré sú blokované a synchrónne, Spring Webflux bol vyvinutý na vývoj aplikácií, ktoré sú plne neblokované, asynchrónne - reaktívne. Funkcionálne API bolo pridané spolu s lambda výrazmi v Java verzia 8.

Reaktívne programovanie znamená model programovania s reaktívnymi tokmi dát, ktorý reaguje na zmenu. Zmena môže prísť zo vstupu a výstupu, alebo iných častí aplikácie. [7] Udalosti, správy a podobne, sú prenášané dátovým tokom, ktorý je sledovaný a pokiaľ bola emitovaná zmena, aplikácia na ňu reaguje. [8] Pojem "reaktívny" znamená neblokujúci, aplikácia nečaká na vykonanie jednej akcie aby spustila ďalšiu, ale akcie môžu bežať paralelne, pokiaľ sa vzájomne neobmedzujú.

Tým, že Spring WebFlux je reaktívny, využíva Reactor, reaktívnu knižnicu - Reactive Streams. Ako už plynie z názvu samotného Reactor, čiže reaktívna knižnica, všetky operátory v tejto knižnici podporujú neblokujúci prístup. Pre prácu so sekvenciou dát poskytuje dva typy API, Mono a Flux s množstvom operátorov. [7]

Funkcionálne reaktívne programovanie je vhodné použiť na tvorbu reaktívnej aplikácie, pokiaľ vyvíjaná aplikácia bude zaťažovaná väčším množstvom zákazníkov v jeden moment, či bude pracovať s veľkým množstvom dát. Spring WebFlux je funkcionálna, reaktívna knižnica, ktorá je navrhnutá práve na naplnenie vyššie spomenutých kritérií.

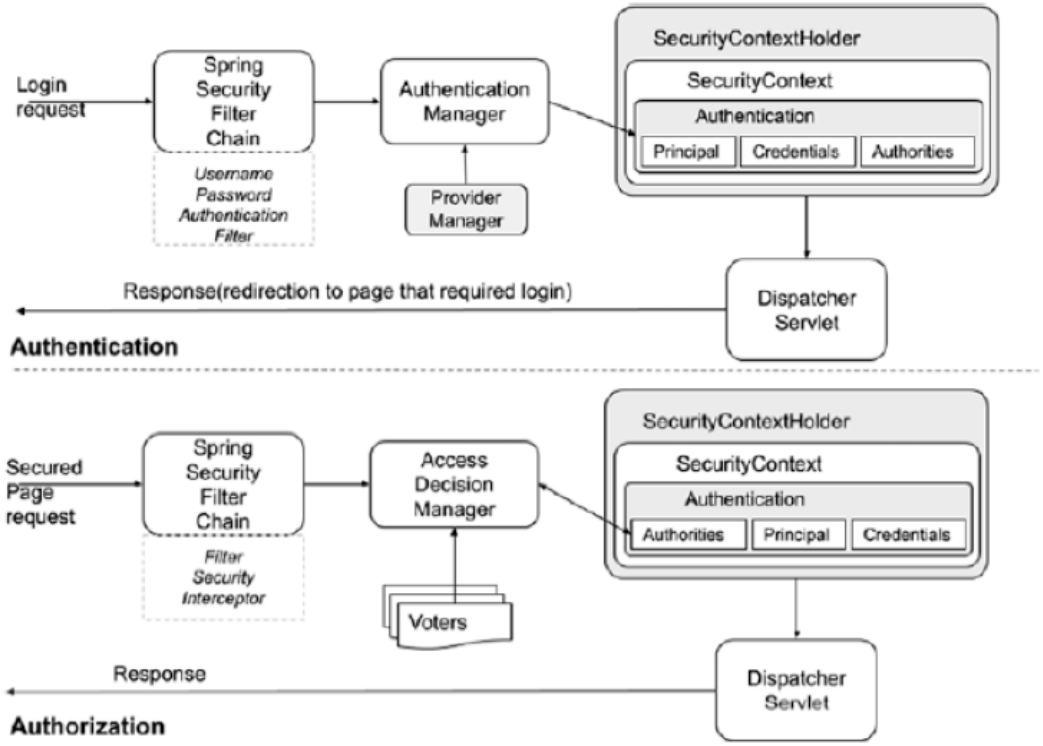
1.2.3 Spring Security

Spring Security je jednou z hlavných možností pre zabezpečenie modernej webovej aplikácie založenej na Spring. Je to softvérová knižnica ponúkajúca zabezpečenie Spring aplikácie. Poskytuje autentifikáciu, autorizáciu a ochranu voči častým útokom na webové aplikácie, či už imperatívne alebo reaktívne. [9] Je flexibilná. Napríklad autentifikáciu je možné prispôsobiť vlastným kritériam, a obsahuje integráciu iných knižníc pre zjednodušenie používania.

Zabezpečenie webovej aplikácie má niekoľko kľúčových pojmov. Každá entita, ktorá má umožnený vstup do aplikácie, kde môže vykonávať nejaké akcie (napríklad užívateľ), sa nazýva **principal**. **Credentials** reprezentujú identifikátory, ktoré sa využívajú na **autentifikáciu**. Počas autentifikácie sú predložené credentials a tie sú porovnané s dátami v databáze. Najčastejšie sa využíva meno užívateľa a jeho heslo ako takzvaná **jednofaktorová autentifikácia**. Prihlásený užívateľ však nemusí mať prístup do každej časti aplikácie alebo používať/vykonávať všetky funkcie aplikácie. Pre tento účel sa používajú **roly**. Jeden užívateľ môže mať od jednej až po n rolí. [10] Po prihlásení užívateľa, úspešnej autentifikácii, webová aplikácia pošle informáciu do webového prehliadača, ktorý vytvorí textový súbor, nazývaný **cookie**. Cookie, cookie prehliadača, či sledovací cookie sa nachádza v adresároch prehliadača a môže obsahovať napríklad zvolený jazyk na danej webovej stránke. [11] Nepermanentný cookie sa nazýva **session cookie**. Nepermanentný preto, pretože jeho životnosť je daná HTTP session užívateľa. Užívateľova HTTP session začína prihlásením a končí odhlásením/zatvorením webovej stránky, čiže čas strávený užívateľom na danej webovej stránke. V tomto type cookie môžme uložiť napríklad produkt pridaný do nákupného košíka, čo je však dôležitejšie, informáciu, že užívateľ je prihlásený a jeho identita bola potvrdená. Inými slovami, uchováva credentials užívateľa, respektíve principal. [10]

Jednou z hlavných zraniteľností je **CSRF**, extrahovanie URL, ktorá vyvoláva nejakú akciu na špecifickom serveri. Následne jej znovupoužitie z vonkajšieho prostredia, čiže mimo servera, na ktorom je užívateľ. Spolieha sa na to, že užívateľ je prihlásený, takže má plný prístup do systému a k väčšine dát. Pokiaľ nie je vykonaná kontrola originality URL, môže byť vyvolaná akcia, ktorá je skrytá práve v tejto požiadavke a samozrejme, je závadná. Zväčša tieto akcie vyvolávajú zmenu dát v systéme. Na riešenie a predchádzanie tohto nebezpečenstva sa využíva **CSRF token** - validácia originality požiadavky alebo **CORS** limitácie. **CSRF token** sa vkladá do hlavičky, napríklad HTTP GET a aplikácia je nastavená tak, že príjma len požiadavky s hlavičkami obsahujúcimi CSRF token, pokiaľ tam nie je, požiadavka, akcia sa nevykoná. **CORS** takisto funguje s hlavičkami, umožňuje nám volania medzi rozličnými doménami určením, ktoré domény sú povolené a aké detaily môžu byť zdieľané.

Spring Security je dôležitou a neoddeliteľnou súčasťou vývoja moderných Spring webových aplikácií, ktorá zjednodušuje implementáciu bezpečnosti webovej aplikácie. Ponúka základ na vystavanie bezpečnej webovej aplikácie a možnosť prispôsobiť riešenie autentifikácie, autorizácie, a ochrany voči útokom. [12] Proces autentifikácie a autorizácie v Spring Security je možné vidieť na obrázku 1.2.



Obr. 1.2 *Proces autentifikácie a autorizácie v Spring Security* [10]

1.3 Spring Session a Redis Server

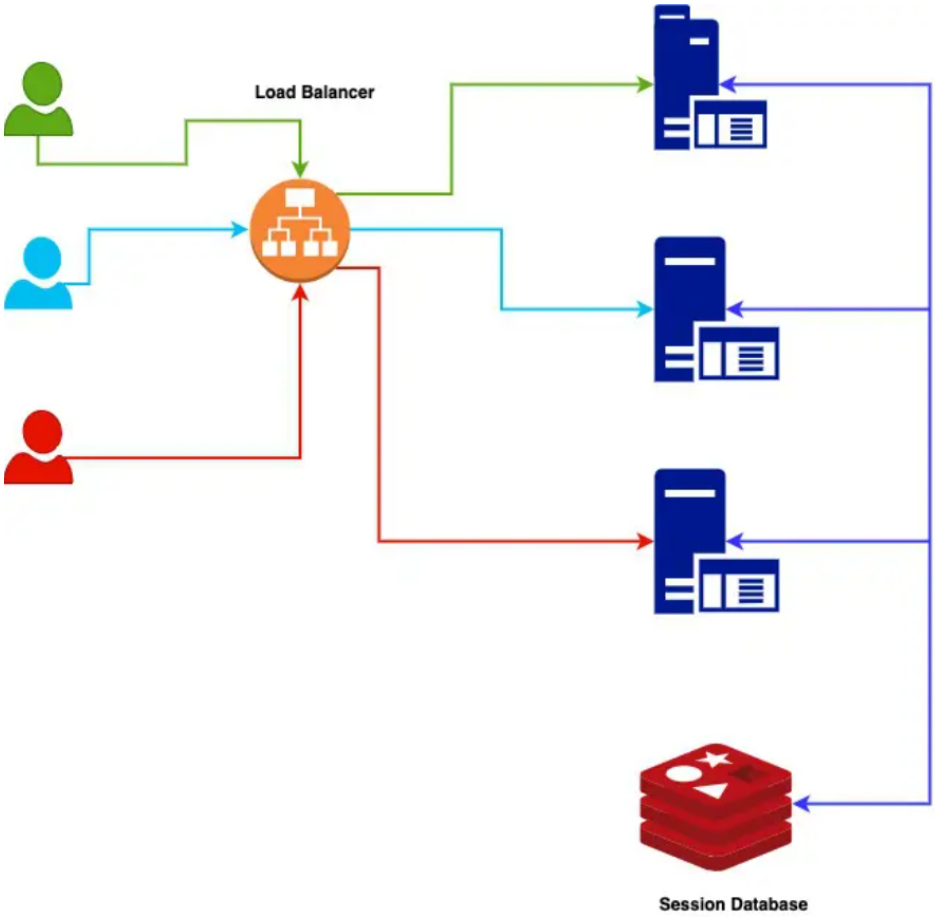
Session je neoddeliteľnou súčasťou zabezpečenej webovej aplikácie. Spring rozlišuje medzi niekoľkými druhmi *session* a ich použitia v rôznych Spring knižniciach.

HttpSession identifikuje užívateľa naprieč webovými stránkami jednej webovej aplikácie. Pomocou nej servlet kontajner, napríklad Tomcat Server, vytvorí *session* medzi HTTP klientom a HTTP serverom. Po každom vstupe užívateľa na webovú stránku rovnakej webovej aplikácie je tento užívateľ rozpoznaný práve pomocou HttpSession, ktorej životnosť je špecifický časový úsek. [13]

WebSession je reprezentáciou HTTP session na strane servera. Je súčasťou Spring WebFlux, čo znamená, že rovnako ako HttpSession ukladá špecifické dáta o užívateľovi, ale je určená pre prácu v reaktívnom prostredí. [14]

Spring Session *ponúka API a implementáciu pre riadenie informácií o užívateľovej session a podporu združených session bez závislosti na špecifickom riešení použi-*

tého úložiska v aplikácii. Obsahuje integráciu s ***HttpSession, WebSocket*** *a* ***WebSession*** [15]. Na ukladanie stavu *session* nie je použitá pamäť, ale separátne externé úložisko ako **Redis Server**, MongoDB, Hazelcast a iné. Práve vďaka tomuto externému úložisku vznikajú výhody ako škálovateľnosť, expiračné možnosti úložiska dát, či viacero užívateľských profilov. Avšak, jednou z najdôležitejších výhod je schopnosť zdieľať stav *session* medzi rôznymi mikroslužbami. [16] Ak je jedna mikroslužba, či server mimo prevádzky, ostatné mikroslužby, servery majú stále dostupné informácie o danej *session*. Každý server aplikácie teda transparentne načíta z daného externého úložiska informácie o *session*.



Obr. 1.3 *Externé úložisko session* [17]

Na obrázku 1.3 je zobrazené riadenie HTTPSession pomocou externého úložiska využitím Redis Server. Obnovenie *session* počas znovunačítania aplikácie, čo znamená, že *session* je prevzatá z Redis Server namiesto z pamäte, či disku, ak je daný server aplikácie reštartovaný. Vďaka externému úložisku je server plne bezstavový.

Použitím Spring Session je možné jednoducho spravovať užívateľovu *session*, predávať ju medzi jednotlivými mikroslužbami a poskytovať *session ID* v hlavičkách Restful API volania. Nie je nutné pridávať žiadny špeciálny kód, len nastaviť predpripravené konfigurácie a pridať Spring Session ako závislosť do projektu. [17]

1.4 MySql verzus PostgreSql Databáza

MySql a PostgreSql sú dve najpoužívanejšie databázy v súčasných moderných webových aplikáciach. Obe sú označené ako relačné databázy, využívajú tabuľky na ukladanie dát a jazyk SQL. Síce obe využívajú jazyk SQL, majú však svoje špecifické dialekty, čo zahŕňa napríklad ďalšie funkcie. MySql je momentálne najpopulárnejšou databázou, na druhej strane PostgreSql je voľne dostupná a objektovo orientovaná relačná databáza. [18]

MySql je relačná databáza, ktorú spravuje a vlastní firma Oracle Corp. Je voľne dostupná, najobľúbenejšia a momentálne ju využíva mnoho spoločností v riešeniach svojich aplikácií. Je jednoducho nastaviteľná, má výbornú podporu bezpečnosti, je dostupná a ľahko ovládateľná i pre menej skúsených vývojárov. Má za sebou obrovskú komunitu s množstvom návodov a dobrou dokumentáciou. Je flexibilná a práve preto sa stáva prvou voľbou pre programátorov webových aplikácií. [18]

Medzi kľúčové benefity patrí práve to, že operuje na viacero platformách, tým je softvér prenosnejší, čo je hlavne využité u webových aplikácií. Podporuje operačné systémy ako Windows, Linux, či Solaris. Takisto podporuje niekoľko programovacích jazykov ako C, Java, PHP, či Python. Spoľahlivý prenos dát v MySql je zabezpečený podporou UNIX a TCP soketov. Poskytuje nepretržité pripojenie ako aj ochranu a integritu údajov zo serverov. Jej ďalšou kľúčovou vlastnosťou v oblasti bezpečnosti, primárne vo webových aplikáciach, je použitý šifrovací algoritmus a bezpečnostný mechanizmus softvéru, ktorý chráni senzitívne informácie.

Na druhej strane medzi jej hlavné nedostatky patrí práve nedostatočná podpora pre hromadné spracovanie databázy, čo je nevýhodou pri webových aplikáciach. Medzi pokročilými funkciami nie je dostatočne zahrnutá možnosť prispôsobiť si MySql vlastným potrebám. [19]

PostgreSql, taktiež nazývaná Postgres, je databázou s otvoreným zdrojovým kódom. Využívajú ju hlavne firmy pracujúce s veľkým množstvom dát na ktorých vykonávajú zložité operácie. [18] Spoločnosti, vývojári si ju vyberajú pretože je bezplatná pre programátorov a vďaka jej flexibilite je umožnené personalizovať funkcionalitu. Primárne využitie má vo vývoji softvéru, ako aj operačných systémov, či programovacích jazykov. [19]

Jednou z jej mnohých výhod je zvládanie viacero úloh efektívne a súbežne. Využíva MVCC, riadenie súbežnosti viacerých verzí, čo znamená, že viacero užívateľov upravujúcich, či len čítajúcich z databázy, môže pracovať v rovnakom čase. [18] Práve flexibilita je kľúčovým faktorom, ktorý dáva spoločnostiam možnosť naplniť obchodné podmienky a to rozmanitým spôsobom práce PostgreSql so systémami správy databáz. Taktiež dovoľuje vývojárom po celom svete podieľať sa na vývoji a ponúkať riešenia správy

databáz. Vyčnieva taktiež možnosťou zostrojiť databázy podľa svojich potrieb.

Samozrejme, Postgres má aj nedostatky. Pri navrhovaní kompatibility aplikácií sú nutné úpravy rýchlosti čítania, aby bola zabezpečená optimálna funkčnosť. Nie je podporovaná v mnohých aplikáciach s otvoreným zdrojovým kódom, čo môže byť limitujúce. [19]

V nasledujúcej tabuľke 1.2 je môžné vidieť porovnanie základných parametrov oboch databáz.

Tab. 1.1 *Postgre verzus MySql* [18]

	PostgreSQL	**MySQL**
Zhoda s SQL	vysoká, spĺňa takmer všetky hlavné funkcie SQL štandardu	čiastočná, neimplementuje všetky funkcie SQL štandardu
Podporované platformy	Solaris, Windows OS, Linux, OS X, Unix-OS a Hp-UX OS	Solaris, Windows OS, Linux, OS X, a FreeBSD OS
Bezpečnosť	natívna podpora SSL pre pripojenia na šifrovanie	vysoko zabezpečená, zabudované bezpečnostné prvky
Výkon	pre veľké systémy, kde je dôležitá rýchlosť čítania a písania	webové projekty, s jednoduchými dátovými tranzakciami
Dátové typy	pokročilé dátové typy, užívateľsky definované typy	štandardné SQL dátové typy

Pri rozhodovaní, ktorú z týchto databáz použiť, je nutné zobrať do úvahy znalosti, veľkosť aplikácie, jej využitie, množstvo dát, ktoré budú v databáze uložené. Rozhodnutie by malo vychádzať od požiadaviek aplikácie, skúseností, robustnosti a potrebnej škálovateľnosti, či spoľahlivosti funkcií správy dát. [19]

Pre webovú aplikáciu tejto bakalárskej práce bola zvolená databáza **MySQL** práve pre jej vyššie spomenuté výhody a jednoduchosť v porovnaní s PostgteSQL. Taktiež, pretože je viac vhodná pre webové aplikácie, má lepšiu dokumentáciu a väčšina výhod PostgreSQL oproti MySQL nie je potrebná pre daný projekt.

1.5 Spring Data R2DBC

Spring Data R2DBC slúži na používanie R2DBC ovládačov so základom v Spring. Poskytuje abstrakciu pre ukladanie a prehľadávanie riadkov databázových tabuliek. [20]

R2DBC, konektivita reaktívnej relačnej databázy, poskytuje reaktívne API do relačnej databázy. Je založené na špecifikácii reaktívnych prúdov a vďaka tomu dodáva plne reaktívne, neblokujúce API. Pracuje s relačnými databázami, v porovnaní s JDBC, ktoré je blokujúce, podporuje prácu s SQL databázami ako MySQL, PostgreSQL s využitím reaktívneho API. Je škálovateľným riešením práce s databázou. [21]

Samozrejme, použitie reaktívneho prístupu má aj svoje nevýhody. Nepodporuje vzťahy medzi dátovými tabuľkami ako *one-to-one* či *one-to-many*, takže narozdiel od

JDBC, vývojár je nútený riešiť vzťahy sám pomocou ďalšieho kódu. Dôvod, ktorý vývojári Spring Data R2DBC uvádzajú je, že tieto **vzťahy - napájanie tabuliek**, sú blokujúcou operáciou, takže tieto prvky momentálne nie je možné do Spring Data R2DBC doplniť.

V aplikácii založenej na Spring, ktorá je reaktívna, neblokujúca, napríklad používajúca Spring WebFlux, pri práci s databázou musí byť využité reaktívne, neblokujúce riešenie na konektivitu s databázou. Práve preto je Spring Data R2DBC dobrou voľbou, je však nutné zvážiť nevýhody, ktoré so sebou prináša.

1.6 Posielanie emailov použitím Spring Boot a SMTP Server

Spring Boot ponúka závislosť *spring-boot-starter-mail*, ktorá je abstrakciou pre posielanie emailov. Využíva *JavaMailSender* rozhranie a pomocou Spring Boot automatickej konfigurácie, je možné pár nastaveniami vytvoriť SMTP server, ktorý posiela emaily. [22]

SMTP je TCP/IP protokol na prenos mailov, ktorý je najčastejšie využívaný emailovými klientami ako Gmail, či Outlook. Posiela a prijíma emaily, ale je limitovaný svojimi funkciami ako radenie prichádzajúcich emailov na koniec. Práve preto sa emaily z SMTP servera periodicky sťahujú a iné protokoly ako POP3, IMAP sa starajú o uchovávanie emailov v serverovej schránke správ.

SMTP server môže byť aplikáciou, či počítačom. Email server použije SMTP na poslanie emailu od email klienta do ďalšieho email servera. Taktiež na prenos emailu z email servera do ďalšieho email servera, či prijatie emailu z iného email servera. Na prijatie správy email server využíva email klienta, aby prichádzajúcu správu stiahol napríklad pomocou IMAP protokolu a vložil ju do schránky príjemcu. [23]

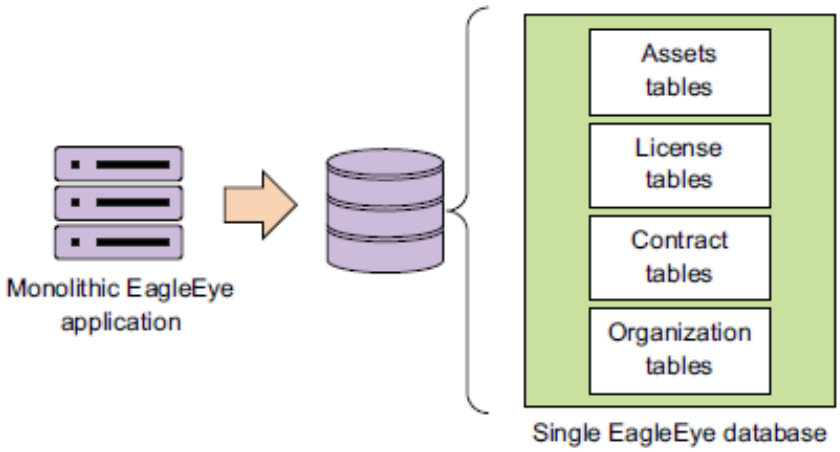
Posielanie emailov pomocou Spring Boot je veľmi jednoduché. Stačí použiť správnu závislosť a nastaviť zopár *properties*, napríklad ako TCP port, cez ktorý bude SMTP server pripojený na odoslanie emailu. Samozrejme, je nutné nastaviť názov host, užívateľské meno - email odosielateľa a heslo.

1.7 Architektúra mikroslužieb

Architektúra mikroslužieb je návrhový vzor aplikácie. Tento architektonický štýl je založený na myšlienke kolekcií služieb, narozdiel od jednej monolitickej aplikácie. [24]

Dôvodov prečo vznikla táto architektúra je hneď niekoľko. Pri monolitickej aplikácii dochádza k niekoľkým problémom. Prvým z nich je pevne prepojená implementačná logika. Takže vyvolanie akejkoľvek obchodnej logiky sa deje úrovňovo, čo má za následok, že aj malá zmena môže vyvolať veľké problémy v aplikácii a zaviesť ďalšie chyby, ktoré sú ťažšie odhaliteľné. Ďalším častým problémom je presah interných dát jednej

časti aplikácie do celej aplikácie. Dáta, ktoré majú medzi sebou prirodzené hranice, sú uchovávané v rovnakom dátovom modeli. To vedie k tomu, že jednotlivé vývojárske tímy v klasickej spoločnosti majú prístup k danému modelu. Síce jeden môže zmeniť len nejakú časť, ktorou sa zaoberá, to však má za následok vytvorenie skrytých závislostí medzi dátami a preniknutie daných zmien, ktoré sa týkajú len interných komponent daného tímu, do celej aplikácie. A v neposlednom rade nastáva monolitický problém. Väčšina komponent v tradičných aplikáciach má základ kódu zdieľaný naprieč celou aplikáciou, to má za následok, že i malá zmena kódu ovplyvní celú aplikáciu. Tá musí byť znovu preložená a otestovaná, čo berie čas aj peniaze. Príklad monolitickej aplikácie je možné vidieť na obrázku 1.4.



Obr. 1.4 *Príklad modelu monolitickej aplikácie* [1]

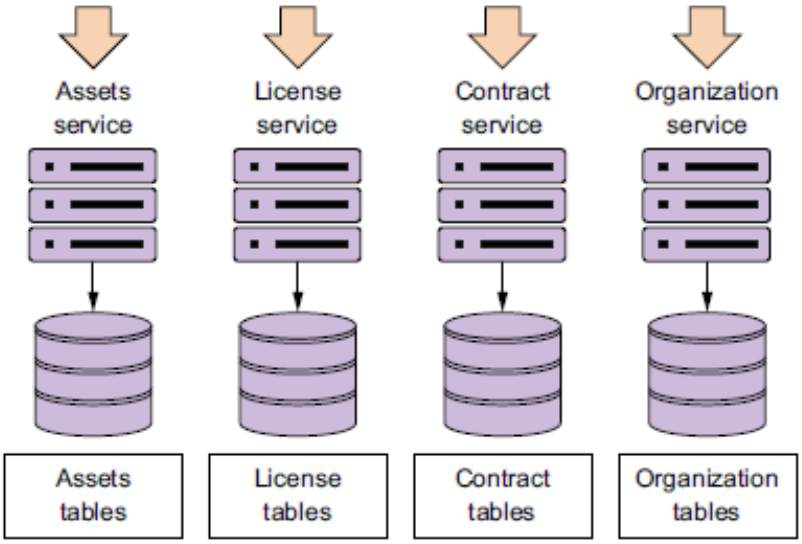
Riešením týchto problémov sa venuje architektúra mikroslužieb. Jej charakteristickými vlastnosťami sú **obmedzenosť**, **voľná viazanosť**, **abstrakcia**, **nezávislosť** a **organizovanie podľa biznisových schopností/potrieb**.

Obmedzenosť je práve v tom, že každá mikroslužba má na starosti malú množinu zodpovedností a tým je teda aj dôkladná v staraní sa o ne. Výhodou je, že pri týchto malých množinách je jednoduchšie zaobstarať skutočne správne fungujúce riešenie problému/biznis požiadavky. Jeden tím zväčša implementuje a udržuje jednu či dve takéto mikroslužby.

V architektúre mikroslužieb sú mikroslužby zväčša prepojené pomocou neimplementačne špecifickým rozhraním a teda je medzi nimi **voľná viazanosť**. To poskytuje väčšiu voľnosť programátorom implementujúcim dané mikroslužby, pokiaľ sa rozhranie, ktoré prepojuje služby, nezmení.

Abstrakcia dátovej štruktúry a dátových zdrojov je vďaka tomu, že každá mikroslužba vlastní svoje dáta a má k nim prístup. Iba daná mikroslužba vie upravovať, meniť dáta svojho dátového modelu. Dokonca aj prístup k dátam v databáze danej mikroslužby je možné uzamknúť tak, aby len daná mikroslužba k nim mala prístup.

Každá mikroslužba môže byť **nezávisle** kompilovaná a dodaná zákazníkovi. Už nie je nutné preložiť, otestovať a dodať zákazníkovi celú aplikáciu, pokiaľ sa zmenila len jej malá časť. [1]



Obr. 1.5 *Príklad modelu aplikácie s architektúrou mikroslužieb* [1]

Na obrázku 1.5 je znázornené rozloženie monolitickej aplikácie z obrázku 1.4 na jednotlivé, samostatné mikroslužby. Narozdiel od monolitického spôsobu má každá mikroslužba na starosti dátové tabuľky súvisiace s jej zameraním a s nimi spojené úkony. To neznamená, že každá mikroslužba má vlastnú databázu, ale má prístup k dátam databázy, ktoré sú pre ňu relevantné.

Samozrejme, aj táto architektúra nie je vždy tým správnym riešením. Tým, že každá mikroslužba beží v produkcii na vlastnom serveri, môže byť niekedy veľmi nákladné udržiavať väčšie množstvo mikroslužieb. Taktiež je na zváženie použiť túto architektúru pokiaľ sa jedná o veľmi malú aplikáciu s obmedzeným množstvom biznisovej a implementačnej logiky a zároveň sa neráta s jej ďalším rozširovaním.

Použitá architektúra riešenia tejto práce je práve **architektúra mikroslužieb**. I keď je zatiaľ aplikácia pomerne malá, ráta sa s jej postupným rozširovaním, kde by mohol nastať bod zlomu a hromadenie problémov pri monolitickom riešení architektúry.

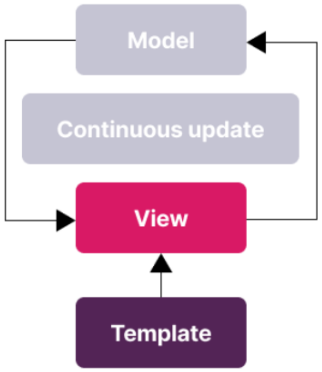
1.8 Angular verzus React

Angular a **React** sú knižnice na tvorbu klientskej časti aplikácie. Obe sú jednými z najpoužívanejších a najobľubenejších knižníc na svete. Síce sa zaoberajú rovnakým odvetvím a ich značná podstata je rovnaká, majú aj mnoho odlišností a výber medzi nimi určuje nie len preferencia vývojára, ale aj kľúčové rozdiely a typ aplikácie, kde majú byť použité.

Angular je frontend knižnica s voľne dostupným zdrojovým kódom využívajúca TypeScript. TypeScript je objektovo orientovaný programovací jazyk postavený na JavaScript programovacom jazyku. Je svetovo obľúbenou a rozšírenou knižnicou na strane klienta na tvorbu škálovateľných a vysokovýkonných mobilných, či webových aplikácii. [25] Škálovateľnosť predstavuje možnosť vytvoriť ako malú aplikáciu, tak aj mohutnú aplikáciu pre veľké spoločnosti s vysokými nárokmi na funkcionalitu. [26]

Angular je plne **MVC** knižnicou. MVC predstavuje dátový model - **M**odel, užívateľské rozhranie - **V**iew a riadiacu logiku - **C**ontroller. Práve vďaka tomuto vzoru je možné s využitím Angular implementovať UI, dátový model aj kontrolu logiky. Nasledovaním MVC vzoru je vývojár schopný nasledovať a implementovať filozofiu "oddelenie zodpovedností" (separation of concerns). Nielenže zjednodušuje a zlepšuje údržbu kódu, ale aj jasne rozdeľuje prácu. V modeli sa aplikácia aktualizuje, aby zobrazovala pridanú/odobranú položku. Užívateľské rozhranie zobrazuje to, čo užívateľ vidí (tlačítka, text a podobne) a na backende sa nachádza riadiaca logika. [26] To logicky usporiadava kód, ktorý je potom prehľadnejší a jednoduchšie sa v ňom orientuje.

Angular podporuje **obojsmernú väzbu dát**. Väzba medzi dátami vytvára komunikáciu medzi UI a biznis logikou, viď obrázok 1.6.



Obr. 1.6 *Obojsmerná dátová väzba* [27]

Obojsmerná dátová väzba ponúka najlepšiu synchronizáciu medzi dvoma komponentami. Pokiaľ komponenta A predá dáta komponente B, komponenta B má vždy aktuálne dáta a zároveň pri zmene dát v komponente B, sú aktualizované dáta aj v komponente A. Práve vďaka tomu Angular môže aktualizovať dve vrstvy súčasne a zabezpečiť, aby si zachovali rovnaké údaje. Pomáha jednoduchšie tvoriť interaktívne užívateľské rozhranie a predísť častým spätným volaniam na zabezpečenie kompatibility dát, čo stojí vývojára čas a úsilie. To umožňuje vytvárať dátovo náročné a rozsiahle aplikácie. Práve táto schopnosť radí Angular medzi hlavný technologický základ robust-

ných aplikácií. [26]

Angular je postavený na tradičnej DOM štruktúre. ***DOM*** *definuje spôsob, akým sa k súboru pristupuje a ako sa s ním manipuluje.* [26] Keďže webové stránky sú tvorené HTML alebo XML dokumentami, sú statické. DOM je API implementované prehliadačom na to, aby spravil zo statických webových stránok funkčné. Je schopné meniť štruktúru dokumentu, štýl, či obsah stránky. [28] DOM je reprezentovaný stromovou štruktúrou a to tak, že koreňovým uzlom je samotný dokument (HTML, XML dokument) a každý ďalší uzol (potomok) je objekt reprezentujúci časť dokumentu. Pri aktualizácii kódu - zmene sa vezme uzol - zmenený element a každý jeho potomok sa vykreslí znovu. Je zrejmé, že táto operácia je časovo náročnejšia a v súhrne, DOM implementácia je pomalá. Angular má však funkciu detekcie zmien a *zones* na umožnenie automatickej detekcie zmien v reakcii na zmeny vykonané asynchrónnymi úlohami. Tým je urýchlený výkon aplikácie. [26]

Bezpečnosť je dôležitou stránkou vývoja moderných aplikácií. Rovnako by mala byť zahrnutá aj do rozhodovania pri výbere UI knižnice. Angular má niekoľko bezpečnostných prvkov. Obsahuje prevenciu voči skriptovaniu medzi stránkami (XSS). Angular považuje všetky hodnoty ako nedôveryhodné, keď je hodnota vložená do DOM, Angular vyčistí vstup a vynechá nedôveryhodné hodnoty. Angular šablóny sú považované za bezpečné a pracuje sa s nimi ako so spustiteľným kódom. Ďalšou bezpečnostnou funkciou je AOT šablónový prekladač, ktorý má na starosť prevenciu voči útokom založených na vložení závadného kódu do šablóny. Značne zvyšuje výkon aplikácie a je primárnym prekladačom, ktorý využívajú Angular CLI aplikácie. V neposlednom rade má Angular vstavanú podporu ochrany voči CSRF a XSSI do HTTP klienta. [29]

Na druhej strane, Angular môže byť mätúci a komplikovaný pre začiatočníkov, práve preto je jeho krivka učenia veľmi strmá. Integrácia tretích strán môže byť zložitá a časovo náročná. V niektorých prípadoch, keď stránky obsahujú interaktívne prvky, môže byť Angular pomalý. Problémy sa môžu objaviť pri aktualizácii zo starších verzií na novšie. [27]

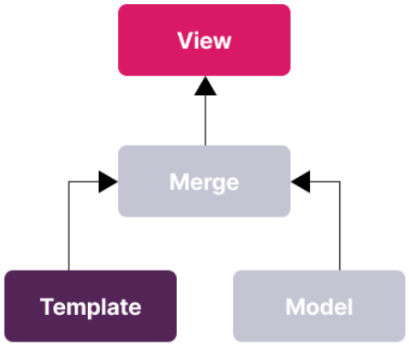
Po Angular je vhodné siahnuť pokiaľ bude aplikácia zložitejšia a komplexnejšia. Takisto je vhodný pre aplikácie, ktoré vyžadujú progresívny vývoj, plnú nativitu a jednostránkové operácie. [26] Prípadne aplikácie s dynamickým obsahom, či aplikácie v reálnom čase. [27]

React je v dnešnej dobe rýchlo rastúcou knižnicou založenou na JavaScript, ktorá sa stáva čím ďalej tým viac populárnejšou v tvorbe UI. Ponúka voľne dostupný zdrojový kód a pomocou neho je možné vytvoriť prispôsobiteľnú a rýchlu aplikáciu. [25]

React je založený na architektúre postavenej na komponentách a to má niekoľko výhod. Komponenty sú ľahko udržovateľné, logicky súdržné a znovupoužiteľné. To

znamená, že jednu komponentu je možné použiť na viacerých miestach aplikácie, takže nevzniká duplicitný kód. Tým je vývoj omnoho rýchlejší, keďže nie je nutné písať ten istý kód niekoľkokrát. Samozrejme, z toho plynie, že komponenty sa ľahko udržujú a je jednoduchšie a rýchlejšie zmeniť jednu komponentu, kde daná zmena sa prejaví na všetkých miestach, ako túto zmenu robiť opakovane pre všetky použitia. [27] Komponenty sú tvorené ako zlúčenie UI a definícií správania danej komponenty, čo znamená, že jeden kód určuje výzor aj chovanie komponenty a preto môžu fungovať v izolácii. [26]

React používa **jednosmernú väzbu dát**, viď obrázok 1.7. Využíva sa na synchronizáciu dát, napríklad medzi dvoma komponentami.[26]



Obr. 1.7 *Jednosmerná dátová väzba* [27]

Pokiaľ komponenta A predáva dáta komponente B pomocou jednosmernej dátovej väzby, komponenta A vždy predá aktuálne dáta, čiže aj keď nastane zmena, komponenta B má vždy zmenené dáta. Pokiaľ však zmena nastane v komponente B, do ktorej sú dáta predávané komponentou A, komponenta A nikdy aktuálne dáta po zmene nedostane. Toto môžme brať ako nevýhodu, aj ako výhodu. *Jednou z výhod tejto techniky viazania dát je, že vývojári sa nemusia obávať zložitých opatrení na spracovanie údajov. Navyše, v jednosmernej dátovej väzbe je ladenie aplikácií plynulejšie a prebieha prirodzene. Táto vlastnosť je dôležitá pre vytváranie veľkých aplikácií.* [26]

Virtuálny DOM je ďalšou kľúčovou vlastnosťou tejto knižnice. *Virtuálna reprezentácia UI aplikácie je uložená v pamäti, kedy virtuálna časť je synchronizovaná s klasickým DOM pomocou knižnice* [26]. Výhodou tohto spôsobu je, že zmena kódu a aktualizácia virtuálneho DOM je omnoho rýchlejšia v porovnaní s klasickým DOM a to preto, pretože mení konkrétny zmenený element a nie celú stromovú štruktúru. [25]

React zahŕňa určité **bezpečnostné** prvky. *Štandardne kóduje takmer všetky hodnoty údajov pri vytváraní prvkov DOM. Aby používatelia mohli vložiť obsah HTML do DOM, React je vybavený funkciou, ktorá jasne vyjadruje nebezpečenstvo jej používania* [30].

O ďalšie zabezpečenie sa musí postarať vývojár sám. Jedná sa hlavne o HTML linky ako href a React komponenty spracúvajúce vstup, ktorý je poskytnutý užívateľom. Vykreslenie komponent môže predstavovať riziko, pokiaľ sa škodlivý vstup používateľa vloží tak, ako je, do kontextu JavaScriptu bez toho, aby bol správne zakódovaný. Taktiež neobsahuje HTTP klienta, takže neposkytuje CSRF ochranu a práve preto, že React sa identifikuje ako minimalistická knižnica, otázku ochrany musí zabezpečiť programátor. [30]

Značnými nevýhodami React knižnice je jeho pochopiteľnosť, primárne v prvotnom štádiu zoznamovania sa s knižnicou sú niektoré jeho koncepty ťažšie uchopiteľné. Udržovanie dokumentácie môže byť taktiež výzvou, keďže React komponenty sa môžu rýchlo meniť. Na vývoj iných častí ako len užívateľského rozhrania - vrstvy zobrazenia aplikácie sú nutné ďalšie knižnice a technológie. [27]

Zvolenie React samozrejme záleží aj na osobných preferenciách programátora, no jedným z hlavných faktorov pri rozhodovaní je dobrá znalosť JavaScript a JSX, potreba aplikácie, ktorá sa bude upravovať podľa požiadaviek zákazníka a nutnosť zdieľaných elementov v aplikácii. [25]

Tab. 1.2 *Angular verzus React* [25]

	Angular	**React**
Typ	Plná softvérová knižnica	JavaScript knižnica
Programovací jazyk	TypeScript	JavaScript
MVC architektúra	Áno	Nie
Dátová väzba	Obojsmerná	Jednosmerná
UI vykreľovanie	Klient/Server strana	Klient/Server strana
Vkladanie závislostí	Podporované	Nepodporované
Typ DOM	Skutočný	Virtuálny
Testovanie	Stačí jeden nástroj	Nutnosť viacerých nástrojov
Krivka učenia	Strmá	Mierna

Nie je možné jednoznačne určiť, ktorá z týchto dvoch frontendových knižníc je lepšia. Každá má svoje výhody a nevýhody a je len na vývojárovi, spoločnosti, požiadavkách aplikácie, či osobných preferenciách, skúsenostiach, ktorú knižnicu pre daný projekt, aplikáciu vybrať.

Pre webovú aplikáciu tejto bakalárskej práce bol zvolený Angular, práve kvôli jeho bezpečnostným prvkom, MVC architektúre, obojsmernej dátovej väzbe, TypeScript jazyku a osobným skúsenostiam s touto knižnicou.

1.8.1 Angular CLI a Angular Material Design

Angular CLI a Angular Material Design sú doplnkami k Angular. Ponúkajú rýchlejšiu možnosť vývoja a predpripravené funkcie či komponenty.

Angular CLI *je rozhranie príkazového riadku, ktorý sa používa na inicializáciu, vývoj a údržbu Angular aplikácií* [31]. Každý príkaz používaný v príkazovom riadku začína slovom *ng*, napríklad *ng serve* na preloženie a lokálne zobrazenie aplikácie, *ng new* vytvorí Angular priečinok pracovného prostredia, či *ng generate* na vytvorenie nových súborov pre Angular komponentu alebo službu. [31] Skracuje čas potrebný na inštalovanie, konfigurovanie a inicializáciu a zároveň ponúka základ novej aplikácie bez nutnosti, aby vývojár písal zakaždým rovnaký kód. [32]

Angular Material Design poskytuje už vytvorené komponenty, ktoré je možné jednoducho použiť v aplikácii pri tvorbe UI. Bol vytvorený priamo Angular tímom na jednoduché integrovanie do Angular aplikácií, je univerzálny a ponúka vývojárom podporu na tvorbu vlastných komponent, ktoré majú štandardné chovanie. [33] Ponúka vyše tridsať komponent, ktoré je jednoduché integrovať, meniť im farebné prevedenie, či typ. Ponúka na výber zo štyroch paliet farieb, alebo možnosť nastavenia vlastných, bez nutnosti preštýlovať každú komponentu zvlášť. Predkladá uľahčenie a urýchlenie vývoja moderných aplikácií so solídnym štýlom a štandardnou funkcionalitou komponent a zároveň ich jednoduchou integráciou.

II. PRAKTICKÁ ČÁST

2 ANALÝZA SÚČASTNÝCH RIEŠENÍ PODOBNÝCH WEBOVÝCH APLIKÁCIÍ

2.1 Webová aplikácia mojlekar

Webová aplikácia *www.mojlekar.eu* [34] je určená pre pacientov a lekárov zo Slovenskej republiky. Jej hlavným zameraním je umožniť pacientom nahlásiť si termín u svojho lekára a zároveň doktorom zabezpečiť jednoduchšie objednávanie ich klientov. Ďalej ponúka blog aj pre neprihlásených pacientov, kontakt, či možnosť nájsť zamestnanie práve v spoločnosti, ktorá túto webovú aplikáciu vyvíja a vlastní.

Pokiaľ chce užívateľ plne využívať služby *mojlekar*, musí sa prihlásiť a teda registrovať, ak nemá vytvorený účet. Aplikácia ponúka dva druhy registrácie podľa toho, ako má byť ďalej využívaná. Jedným je registrácia pacienta a druhým registrácia lekára.

Registrácia pacienta vyžaduje vyplnenie krátkeho formulára, ktorý obsahuje *Titul, Meno, Priezvisko, Mesto, Dátum narodenia, Názov zdravotnej poisťovne, Email, Telefón, Heslo, Zopakovanie hesla, opísanie potvrdzovacieho kódu a súhlas s obchodnými podmienkami.* Pokiaľ už účet pod danou emailovou adresou existuje, registrácia neprebehne a stránka upozorní užívateľa na danú skutočnosť. Taktiež mu ponúkne možnosť prihlásenia či obnovy hesla, pokiaľ ho užívateľ zabudol. Ak účet s rovnakou emailovou adresou neexistuje, registrácia je úspešná a užívateľovi je vytvorený jeho účet.

Užívateľským menom sa stáva emailová adresa zadaná pri registrácii a heslo je práve tým, ktoré bolo vyplnené taktiež pri registrácii. Webová stránka s prihlásením má predvyplnené políčko na rozlíšenie lekárov a pacientov, pole pre zadanie užívateľského mena a hesla. Taktiež sú v tejto časti zahrnuté možnosti pre obnovenie zabudnutého hesla, registrácia ako pacient, či lekár, viď obrázok 2.1.

Obr. 2.1 *Prihlásenie do webovej aplikácie mojlekar* [34]

Po prihlásení má užívateľ niekoľko možností. V hlavnom menu sa nachádzajú možnosti *Hľadaj ambulanciu a zadaj objednávku na vyšetrenie, Výber mojich lekárov, Moje objednávky u lekárov, Moje objednávky receptov, Archív starých objednávok*. Pod hlavným menu nasleduje sekcia *Administrácia*, ktorá obsahuje *Zmena mojich údajov, Zmena hesla* na správu účtu užívateľa. Po kliknutí na jeden z týchto uvedených textov je užívateľ presmerovaný na novú podadresu URL. V neposlednom rade je tlačítko *Odhlásiť* ako prostriedok na odhlásenie užívateľa.

V časti *Hľadaj ambulanciu a zadaj objednávku na vyšetrenie* je možné nájsť lekára a jeho ambulanciu vyplnením polí *Mesto, Špecializácia a Meno lekára / ambulacie / kliniky*. Pokiaľ je ambulancia/lekár úspešne nájdený a teda sa nachádza v databáze webovej aplikácie, je ponúknutá možnosť prihlásenia sa na termín, či pridania tejto ambulancie do zoznamu užívateľových lekárov. Avšak, ak ambulancia v databáze nie je, čiže lekár nepoužíva *mojlekar*, užívateľ je na túto skutočnosť upozornený hláškou.

V sekcii *Výber mojich lekárov* je tabuľka lekárov, ktorých si už užívateľ pridal a zároveň každého lekára, či ambulanciu je možné odstrániť. Ďalej je v tabuľke tlačítko *Zisti termín*, ktoré presmeruje užívateľa na náhľad voľných termínov v danom a nasledujúcich mesiacoch. Taktiež je možné v daný deň požiadať o recept na liek bez nutnosti objednávať sa na konkrétnu hodinu.

Moje objednávky u lekárov, Moje objednávky receptov zobrazujú plánované návštevy lekárov a žiadosti o recepty na lieky. *Archív starých objednávok* zobrazuje tabuľku minulých objednávok k lekárom a žiadosti o recepty na lieky, ktoré už boli predpísané.

V *Zmena mojich údajov, Zmena hesla* vieme zmeniť údaje zadané pri registrácii, či heslo.

Webová aplikácia *mojlekar* je vsadená do šablóny, ktorá sa naprieč jednotlivými sekciami nemení. Používanie aplikácie je intuitívne, obsahuje všetky potrebné úkony a informácie. Dizajn je už pomerne zastaralý a síce aplikácia obsahuje všetko potrebné, výber úkonov je na hlavnej stránke a použitá šablóna nemá plnú výšku a šírku, tým je priestor na interakciu užívateľa s aplikáciou zmenšený. Taktiež zmenšený priestor znižuje prehľadnosť stránky, hlavne v prípade zobrazenia tabuliek. Pri dlhšom nepoužívaní, kedy aplikácia je stále otvorená a užívateľ je prihlásený, dochádza k spomaleniu, dlhému načítaniu, či dokonca úplnému zaseknutiu stránky, kedy je nutné ju obnoviť alebo zavrieť a prihlásiť sa znovu.

Táto webová aplikácia spĺňa svoj účel, ale bolo by potrebné obnoviť dizajn a vyriešiť občasné spomalenie, či úplné zaseknutie.

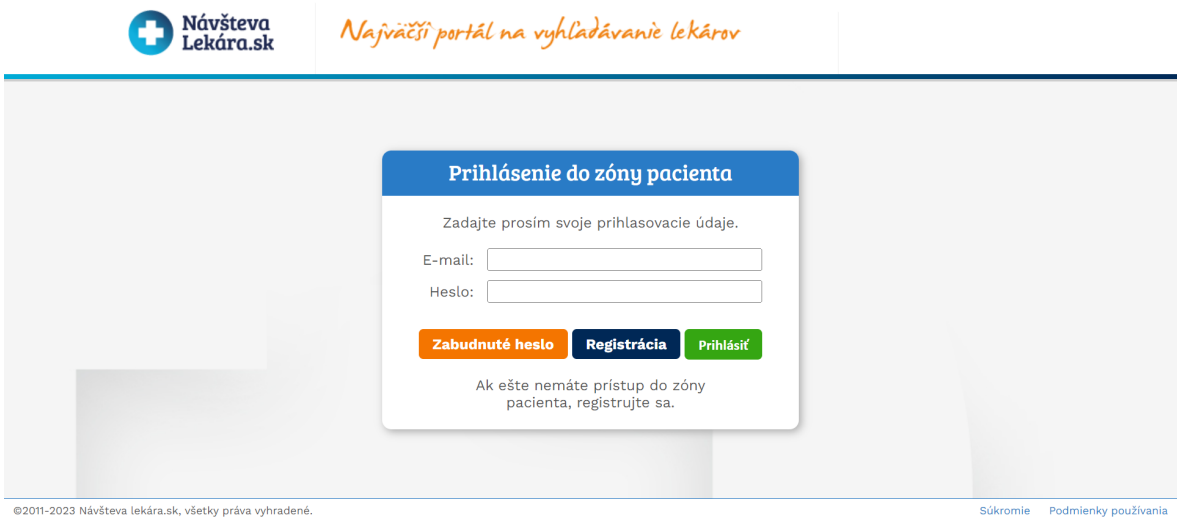
2.2 Webová aplikácia navstevalekara

Webová aplikácia *www.navstevalekara.sk* [35] je taktiež zameraná na online objednávanie pacientov k lekárom. Má dve verzie, verziu pre Slovenskú republiku a verziu pre Českú republiku. Obe verzie ponúkajú v svojej podstate to isté, až na jemné rozdiely v zobrazení ponúkaných možností. Užitočnou časťou je vyhľadávanie lekárskych pohotovostí, kde je možné nájsť pohotovosť v danej krajine (Česko, Slovensko) a informácie o otváracích hodinách, či kontaktoch. Má oddelené zóny pre lekárov a pacientov, ponúka *blog, kontakty, základné informácie o portáli, články, videá, partnerov, články o tomto projekte a sekciu Časté otázky.*

Na hlavnej stránke, bez nutnosti registrácie, je možné vyhľadať lekára podľa zamerania a prihlásiť sa na termín. Pre každý deň v mesiaci je nutné vybrať konkrétny čas a po kliknutí stačí vyplniť formulár s osobnými údajmi a prihlásiť sa na vyšetrenie. Takisto aplikácia zobrazuje podrobný popis ambulancie a lekára, adresu, kontakt, či presnú mapu kde je sídlo ambulancie. Ak bol doktor ohodnotený a bola naňho v rámci portálu uverejnená recenzia, je možné ju v tejto časti nájsť. Hlavná stránka obsahuje stručný návod ako použiť vyhľadávanie doktora, ambulancie. V neposlednom rade obsahuje krátky prehľad doktorov registrovaných na tomto portáli.

Zóna lekárov obsahuje dve prihlásenia. Jedno prihlásenie je do staršej verzie s oznamom, že každý prihlásený lekár má možnosť bezplatne prejsť na novú verziu. Druhé prihlásenie je do spomínanej novej verzie.

Po kliknutí na *Zóna pacienta* je užívateľ presmerovaný na prihlásenie. Pokiaľ nemá vytvorený účet, je mu ponúkaná možnosť prejsť na registráciu. Taktiež je tu aj riešenie pri zabudnutí hesla, viď obrázok 2.2.



Obr. 2.2 *Prihlásenie do webovej aplikácie navstevalekara* [35]

Registrácia je vo forme jednoduchého formulára. Stači len vyplniť polia *Meno, Priezvisko, Email, Telefónne číslo* a zaškrtnúť súhlas s podmienkami používania a so spracovaním osobných údajov. Pokiaľ profil s danou emailovou adresou existuje, užívateľ je na to upozornený a registrácia nie je dokončená. Po úspešnej registrácii sa zobrazí tlačítko presmerovania na stránku prihlásenia, kde je možné použiť práve vytvorený účet. Heslo k účtu je vygenerované a zaslané na emailovú adresu zadanú pri registrácii.

Po prihlásení sa užívateľ ocitne na hlavnej stránke, tá obsahuje registrácie vytvorené užívateľom. Zahŕňa registrácie vytvorené na danú emailovú adresu ešte pred vytvorením účtu v aplikácii. Prihlásenie sa k doktorovi je možné len na hlavnej stránke, po prihlásení tu táto možnosť nie je. Ďalej sú tu možnosti *Odhlásiť sa a Profil*.

V sekcii *Profil* si môže užívateľ zmeniť údaje zadané pri registrácii a heslo. Po kliknutí na tlačítko *Zmeniť*, sú údaje uložené a teda zmenené.

Je možné prejsť zo sekcie prihláseného užívateľa na hlavnú stránku kliknutím na logo. Nie je nutné sa opätovne prihlasovať pokiaľ sa chce užívateľ vrátiť, zostáva prihlásený po určitú dobu.

Webová aplikácia *navstevalekara* je komplexnejšia a účet vytvorený registráciou slúži len na prehľad, nie na samotné vykonávanie objednávok. Dizajn je chaotickejší a stránka obsahuje veľa funkcií, tlačítiek a textu pri sebe, či v nelogických zhlukoch. Chýba prehľadné menu funkcií a sekcií, ktorého účel substituuje spodná lišta s rôznymi odkazmi. Na druhej strane možnosť objednania k lekárovi bez nutnosti registrácie je pre mnoho ľudí veľkou výhodou. Rovnako ako spracovanie informácií o doktoroch, či portál pre dve krajiny. Avšak, po kliknutí na *Zóna pacienta* v sekcii pre Českú republiku, stránka nefunguje a zobrazí len chybovú hlášku zo servera.

2.3 Webová aplikácia medevio

Webová aplikácia *medevio* [36] je určená na objednávanie pacienta k svojim lekárom. Okrem webovej aplikácie ponúka aj mobilnú verziu. Na hlavnej stránke ponúka výber z dvoch jazykov, slovenčinu a češtinu, prehľad, čo všetko užívateľovi ponúka, *sekciu pre pacienta a lekára, prihlásenie a založenie testovacej ordinácie*.

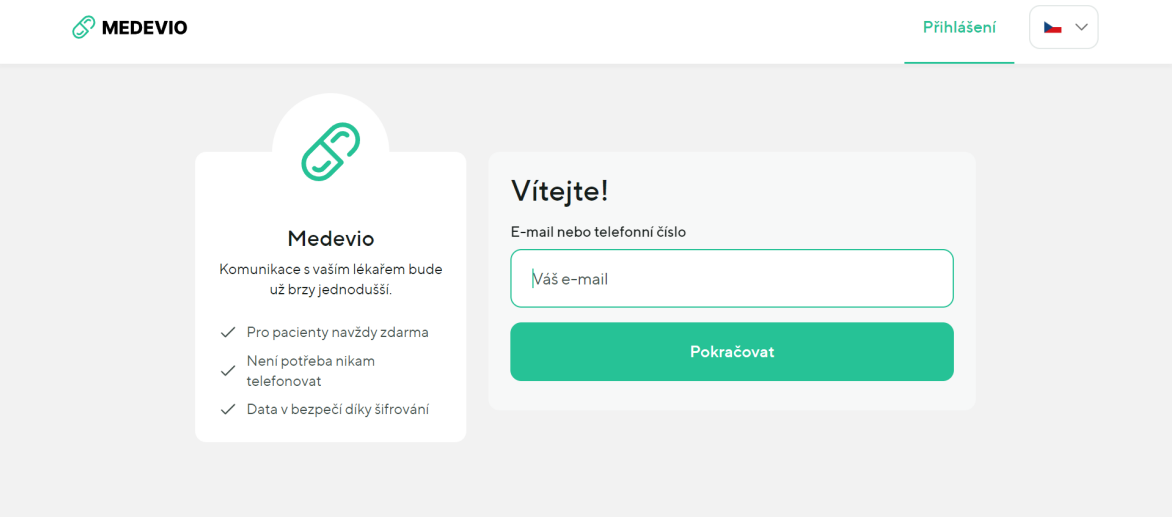
Na úvodnej stránke sa nachádza možnosť *Vyhľadať svojho lekára*, ktorá presmeruje na stránku, kde je možné vyhľadať lekára, či terapeuta. Pokiaľ sa lekár nenachádza v databáze, Medevio ponúka spôsob, vo forme kontaktovania spoločnosti, ktorá daného doktora osloví. Avšak, ak je lekár nájdený, užívateľ vidí, či má dovolenku, aké vyšetrenia ponúka a kde sa nachádza jeho ambulancia.

Zóna pacienta má na začiatku opäť tlačítko na vyhľadanie lekára, stručný prehľad ako aplikácia funguje, najčastejšie kladené otázky, partnerov a formulár, kde je možné odporučiť doktora, aby ho spoločnosť mohla osloviť a pacient sa cez aplikáciu prihlásil

na vyšetrenie.

Zóna pre lekárov obsahuje tlačítko na vyskúšanie aplikácie ako lekár, podrobný popis ako aplikácia funguje a jej výhody pre doktora. Ponuka tvorby webových stránok zameraných na lekársku činnosť a cenník. V poslednej časti je formulár, ktorý doktor môže vyplniť a spoločnosť sa mu ozve a vytvorí mu ponuku. Taktiež sú tu recenzie spokojných užívateľov - lekárov.

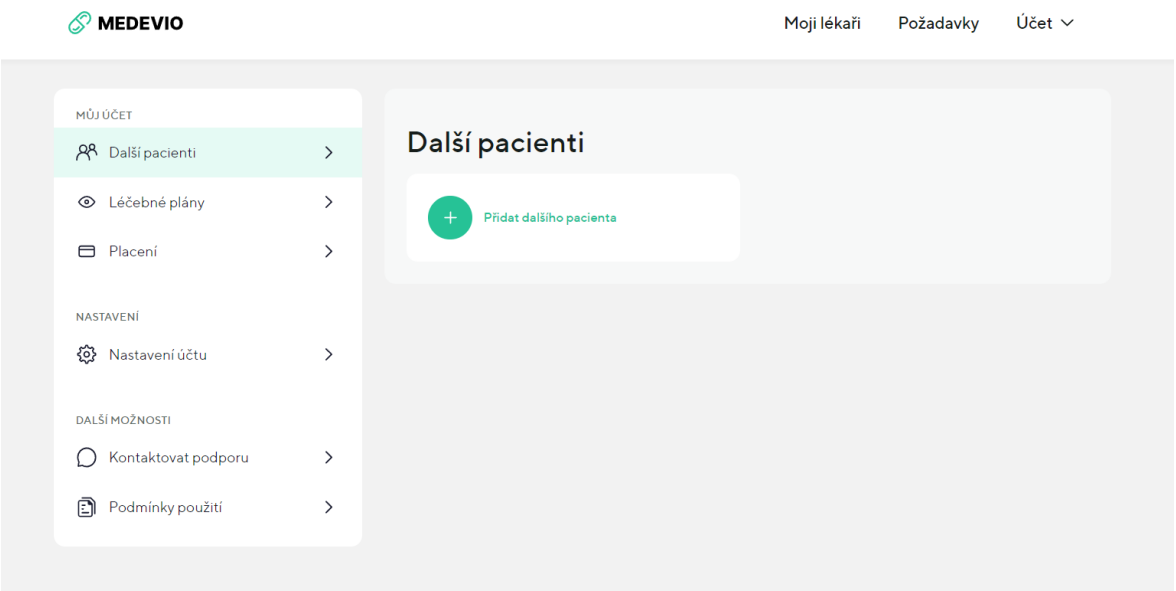
Pokiaľ sa užívateľ chce objednať k vybranému lekárovi, musí sa prihlásiť. Ak nemá vytvorený účet, je nutné zadať svoju emailovú adresu alebo telefónne číslo a skúsiť sa prihlásiť. Aplikácia mu ponúkne registráciu na danú emailovú adresu, telefónne číslo, alebo možnosť ísť o krok späť a skúsiť sa prihlásiť inou adresou, číslom. Pokiaľ sa užívateľ rozhodne registrovať, emailová adresa, telefónne číslo je predvyplnené, ale dá sa zmeniť. Aplikácia vyžaduje len zadanie hesla a potvrdenie registrácie tlačítkom, viď obrázok 2.3. V danom momente je užívateľ presmerovaný na svoj užívateľský účet. Na zadanej emailovej adrese, telefónnom čísle musí potvrdiť registráciu, čiže overiť ich platnosť. A teda, i keď užívateľ nepotvrdil svoju emailovú adresu, telefónne číslo, má prístup do svojho účtu, bezprostredne po registrácii. Pri ďalšom prihlásení už musí byť emailová adrasa/telefónne číslo overené.



Obr. 2.3 *Prihlásenie do webovej aplikácie medevio* [36]

Účet užívateľa je minimalistický. Na stránke opäť dominuje vyhľadávanie lekára, ktoré funguje rovnako, ako na hlavnej stránke. Nachádza sa tu lišta s tlačítkami *Moji lekári, Požiadavky* a menu s tlačítkom *Účet*. Po kliknutí na *Moji lekári* je zobrazený zoznam lekárov užívateľa a opätovne, možnosťou vyhľadať ďalšieho doktora. Sekcia *Požiadavky* zobrazuje *Aktívne požadiavky*, ktoré ešte neboli vybavené a *Vybavené požiadavky*, čiže tie, o ktoré už je úspešne postarané, napríklad prebehla objednaná náv-števa u lekára. Táto časť teda zobrazuje dosavadnú históriu, ale aj možnosť vytvoriť požia-

davku vrátením sa do sekcie *Požiadavky*, či úvodná stránka, teda umožňuje prihlásiť sa na kontrolu, či vyhľadať lekára. Posledná možnosť je správa účtu cez menu tlačítko *Účet*. Ponúka dve možnosti, *Môj účet a Odhlásiť sa*. V sekcii *Môj účet* je variácia možností ako pridanie ďalšieho pacienta, liečebné plány, platenie, nastavenie účtu, kontakt na podporu a podmienky používania aplikácie, viď obrázok 2.4.



Obr. 2.4 *Užívateľský účet* [36]

Medevio je modernou, dobre spracovanou, prehľadnou webovou aplikáciou, ktorá ponúka prehľadné objednávanie k doktorom. Na druhej strane, stránky mimo sekcie, kedy je užívateľ prihlásený, sú roztiahnuté a je nutné dlho prechádzať, než sa človek dostane na ich koniec. V nižších častiach sú zaujímavé informácie a funkcionalita, ktorá, kvôli ich umiestneniu, môže byť ľahko prehliadnutá. V užívateľskej sekcii sú tlačítka, ktorých funkcionalita je rovnaká a vlastne presmerovávajú na tú istú časť aplikácie. Celkovo je táto aplikácia skutočne moderným riešením so zaujímavými, užitočnými funckiami a prehľadným dizajnom.

2.4 Zhrnutie analýzy súčasných riešení podobných webových aplikácií

Webové aplikácie *mojlekar* [34], *navstevalekara* [35] a *medevio* [36] poskytujú v súhrne rovnaké služby, vyhľadanie doktora a objednanie sa k lekárovi. Aplikácia *mojlekar* je dizajnom najzastaralejšou a najmenej prívetivou na používanie. Aj ked je manipulácia jednoduchá, vyskytujú sa tam problémy. Rovnako *navstevalekara* patrí dizajnom a spôsobom prevedenia, zobrazením ponúkaných služieb medzi zastaralejšie webové aplikácie. Takisto má zopár problémov, na ktoré nie je ťažké naraziť. Webová aplikácia *medevio* je najnovšou a najmodernejšou aplikáciou. Je prehľadná, ponúka zaujímavé doplnkové služby a možnosti pre užívateľa, či lekára.

3 NÁVRH WEBOVEJ APLIKÁCIE

Návrh webovej aplikácie prebiehal v niekoľkých krokoch. Najprv boli zvolené funkčné požiadavky, architektonický model aplikácie a architektúra systému.

3.1 Funkčné požiadavky aplikácie

Funkčné požiadavky webovej aplikácie boli rozvrhnuté do aktérov, modelu prípadov použitia a ich špecifikácie. Zohľadňujú požiadavky zadávateľa a zobrazujú, akí sú účastníci v systéme a ako v ňom pracujú. V rámci spracovania funkčných požiadaviek bol taktiež zohľadnený výsledok analýzy riešení podobných webových aplikácií.

Funkčné požiadavky sa odvíjajú od požiadaviek spoločnosti Diatop na túto aplikáciu. A teda funkčnými požiadavkami sú:

- registrácia,
- prihlásenie užívateľa,
- možnosť zmeny údajov užívateľom zadaných pri registrácii,
- vytvorenie objednávky užívateľom,
- poslanie objednávky vytvorenej užívateľom systémom vo forme emailu výdajni Diatop,
- zobrazenie prehľadu doposiaľ vytvorených objednávok užívateľovi,
- administrátorská časť s možnosťou nahrania dokumentu,
- zobrazenie registrovaných užívateľov administrátorovi,
- zobrazenie nahraných súborov - článkov užívateľovi k čítaniu, nie však k stiahnutiu, či ku kopírovaniu,
- odhlásenie.

Tieto požiadavky sú spracované do modelu prípadov použitia spolu s aktérmi.

3.1.1 Aktéri

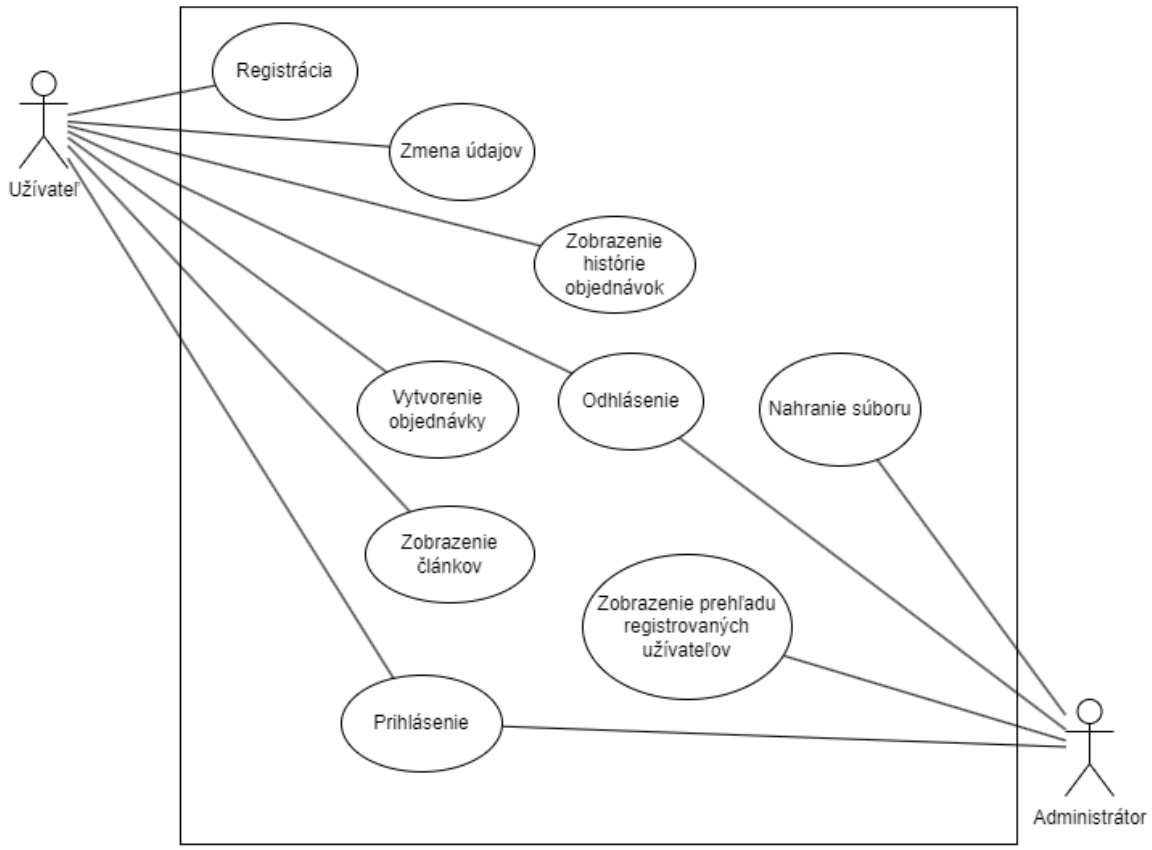
Vo webovej aplikácii pre výdajňu zdravotníckych pomôcok pre diabetikov vystupujú dvaja aktéri, a to užívateľ a administrátor. Administrátor je v tomto ponímaní človek, ktorý má prístup k dátam užívateľov a vie spravovať určité časti systému.

Užívateľ je zákazníkom výdajne zdravotníckych pomôcok. Zohráva jednu z najdôležitejších rolí v systéme. Má vytvorený účet, v ktorom sa pohybuje a využíva jednotlivé funkcie systému. Na to, aby mohol používať aplikáciu, sa musí registrovať a následne prihlásiť. Po prihlásení môže čítať články, spravovať svoj účet a hlavne, objednávať si dodávky pomôcok.

Administrátorom je osoba zamestnaná vo výdajni. Má oddelenú časť aplikácie, v ktorej sa pohybuje. V tejto sekcii môže vidieť registrovaných užívateľov a základné informácie o nich. Avšak, jeho primárnou funkciou je nahrávanie článkov, ktoré sú zobrazené užívateľovi na čítanie.

3.1.2 Model prípadov použitia

Model prípadov použitia reprezentuje interakcie aktérov so systémom. Postupnosť činností nie je nutná pre tento diagram. Každý aktér, **užívateľ** a **administrátor** využíva systém určitými spôsobmi, má k dispozícii iný balíček funkcií.



Obr. 3.1 Model prípadov použitia

Užívateľ má možnosť využívať viac funkcií systému, ako je vyobrazené na obrázku 3.1. Samozrejme, prvým a najdôležitejším krokom je **registrácia**. Vďaka nej je registrovaný v systéme a môže využívať ďalšie časti aplikácie. Po registrácii prichádza

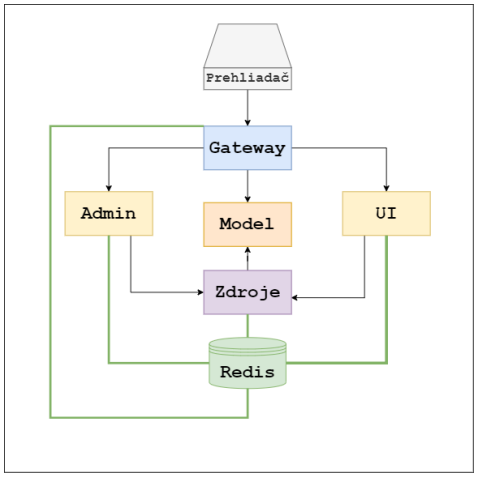
možnosť **prihlásenia**. Prihlásenie je realizované užívateľským menom a heslom, ktoré boli zadané pri registrácii. Následne, prihlásený užívateľ prichádza do aplikácie, kde má niekoľko možností. V prvom rade sú mu **zobrazené články**, hlavná funkcia a teda jadro aplikácie je **vytvorenie objednávky** na doručenie pomôcok výdajňou. Neodlúčiteľnou funkciou je **zmena údajov**, ktoré zadal pri registrácii a **zobrazenie histórie objednávok**. V neposlednom rade je možné **odhlásiť sa** z aplikácie.

Administrátor má obmedzenejší balíček funkcií, s ktorými pracuje so systémom, keďže jeho postavenie slúži na administráciu. Rovnako ako užívateľ sa **prihlási** do aplikácie. Je mu umožnené **zobraziť prehľad registrovaných užívateľov**. Hlavnou funkciou je **nahrávanie súborov** a teda článkov, ktoré si užívateľ môže prečítať vo svojej sekcii. Rovnako je mu poskytnutá možnosť **odhlásiť sa** zo systému.

3.2 Architektonický model aplikácie

Architektonický model webovej aplikácie pre výdajňu zdravotníckych pomôcok pre diabetikov sa skladá z piatich základných modulov a *Redis servera*. Každá časť je *Maven modul*, pracuje ako samostatná oddelená jednotka - mikroslužba. Mikroslužby medzi sebou komunikujú a tak tvoria ucelený celok, webovú aplikáciu.

Každá časť architektonického modelu má v réžii určitú funkcionalitu. Jedná sa či už o UI prevedenie, komunikáciu s UI - predávanie dát, ale aj zabezpečenie daného modulu, mikroslužby, prácu s dátami z databázy a ich spracovanie a predanie do UI sekcie. Každý modul je samostatná jednotka. Oddelením je zabezpečená jednoduchá škálovateľnosť a teda aj rozšíriteľnosť celej aplikácie. Práve preto bola zvolená daná architektúra, pretože je v pláne, že klient bude mať ďalšie požiadavky na fungovanie systému, ako napríklad pridanie nových funkcionalít.



Obr. 3.2 Architektonický model

Na obrázku 3.2 je zobrazený architektonický model s jednotlivými časťami a spô-

sobom komunikácie, kde zelené čiary reprezentujú dorozumievacie cesty medzi *Redis serverom* a jednotlivými *Maven modulmi*. Čierne šípky vyobrazujú komunikačné cesty medzi jednotlivými *Maven modulmi*.

Gateway je vstupnou bránou do aplikácie. Každá požiadavka, či komunikácia prehliadača s aplikáciou prechádza cez Gateway. Jeho úlohou je predať úlohy mikroslužbe (*Maven modulu*), ktorá za ňu zodpovedá. Má na starosti autentifikáciu a autorizáciu užívateľa.

Admin mikroslužba zodpovedá za administrátorskú časť aplikácie. Využíva ho len administrátor. Obsahuje UI časť v ktorej sa administrátor pohybuje a vykonáva činnosti spojené s jeho rolou. Rovnako obsahuje zabezpečenie tejto časti aplikácie.

UI modul reprezentuje hlavnú užívateľskú časť, teda UI kde sa užívateľ pohybuje - čo vidí. Tento priestor UI je určený užívateľom - zákazníkom výdajne. Rovnako ako Admin, obsahuje zabezpečenie danej časti aplikácie.

Zdroje mikroslužba sa stará o predávanie a prácu s dátami. Úzko spolupracuje s **Model** mikroslužbou. Obsahuje SMTP server, ktorý posiela emaily. Rovnako ako ostatné moduly, má konfiguráciu bezpečnosti.

Model obsahuje reprezentáciu databázových dát. Triedy, ktoré sa využívajú na prácu s databázovými dátami a rozhrania repozitárov pre prácu s týmito dátami. Je to časť aplikácie, kde sa nachádzajú všetky potrebné triedy, repozitáre a podobne, na prácu s dátami uloženými v databáze.

Redis je server využitý na ukladanie *session*. Každý modul, do ktorého sa pristupuje z prehliadača, sa odkazuje práve na túto *session* a teda, či má užívateľ prístup - je prihlásený v aplikácii.

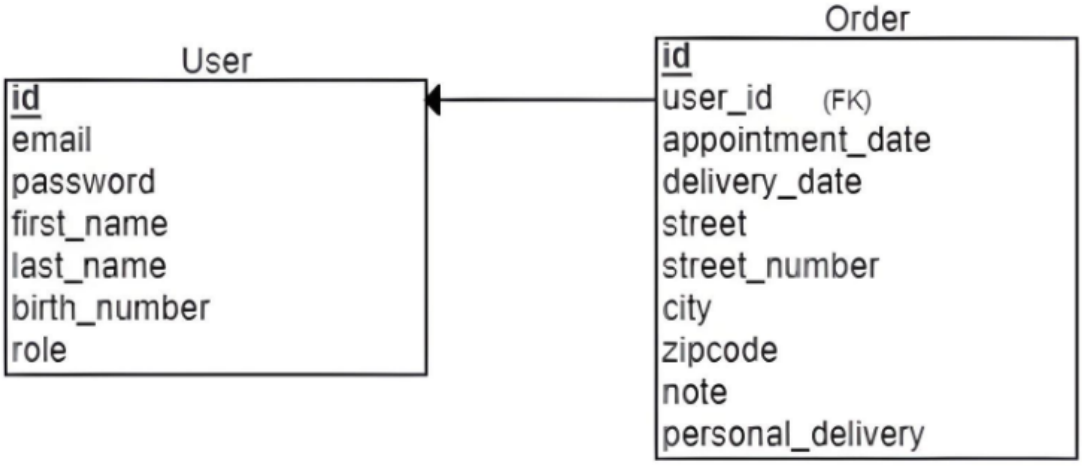
Komunikácia medzi mikroslužbami má dve podoby. Prebieha medzi mikroslužbou a Redis serverom, pýta sa na *session*. Gateway predáva požiadavky Admin časti a UI časti, čiže požiadavky od užívateľa, administrátora. Je to reakcia na akciu užívateľa, administrátora v prehliadači. Rovnako používa triedy, rozhrania repozitárov z Model modulu. Admin a UI mikroslužby posielajú žiadosti do Zdroje modulu, či už v podobe toho, aby odoslal email, alebo poskytol nejaké dáta. Modul Zdroje komunikuje s Modelom a teda používa jeho triedy, rozhrania repozitárov.

3.3 Architektúra systému

Architektúra systému predstavuje dátový model, čiže model dát uložených v databáze a model tried. Model tried reprezentuje objekty s ktorými sa pracuje v rámci implementácie aplikácie. Dátový model sa nachádza v mikroslužbe Model, rovnako ako väčšina tried aplikácie.

3.3.1 Dátový model

Dátový model aplikácie sa skladá z *User, Document* a *Order*.



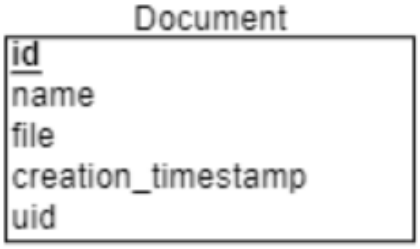
Obr. 3.3 Dátový model *User* a *Order*

User reprezentuje informácie o užívateľovi uložené v databáze, ktorého model je vyobrazený na obrázku 3.3 . Obsahuje *id* ako primárny kľúč, je jedinečné pre každého užívateľa a na základe neho je užívateľ identifikovaný. Užívateľským menom sa stáva *email*, ktorým sa užívateľ prihlasuje do aplikácie spolu s *password*, čiže heslom. Ďalšími dôležitými informáciami o užívateľovi sú *first_ name, last_ name*, teda meno a priezvisko. Rodné číslo *birth_ number* musí byť uložené pre každého užívateľa, pretože na základe neho výdajňa dokáže stiahnuť recept daného klienta. Posledným atribútom je *role*, ktorá reprezentuje rolu užívateľa.

Order obsahuje informácie o objednávkach. Vytvorenie objednávok je kľúčovou funkciou webovej aplikácie. Objednávka má svoje unikátne *id*, ktorým sa identifikuje v systéme. Cudzím kľúčom je *user_ id* a teda hovorí o tom, komu objednávka patrí. Tento vzťah a model je zobrazený na obrázku 3.3. Atribút *appointment_ date* nosí informáciu kedy klient navštívil doktora. Táto informácia je dôležitá práve pre výdajňu, pretože niektoré poukazy na pomôcky majú veľmi krátku dobu platnosti. To znamená, že užívateľ môže zadať dátum doručenia *delivery_ date* o mesiac, ale poukaz, ktorý mu predpísal lekár na kontrole, už nemusí byť platný. Preto výdajňa potrebuje vedieť, kedy daný poukaz spracovať. Adresa je uložená v *street, street_ number, city a zipcode*. Klient môže výdajni počas objednávky zanechať nejakú poznámku, ktorá sa ukladá do *note*. Je možné zvoliť osobné doručenie, táto informácia je uchovaná v *personal_ delivery*.

Document reprezentuje súbor, ktorý sa nahráva v administrátorskej časti a zobrazuje v užívateľskej časti. Ako je ukázané na obrázku 3.4 má pridelený jedinečný

identifikátor *id*, *name* - meno, časovú známku, kedy bol nahratý do systému *creation_ timestamp*, samotný súbor ako pole bytov *file* a *uid*.



Obr. 3.4 Dátový model *Document*

3.3.2 Model tried

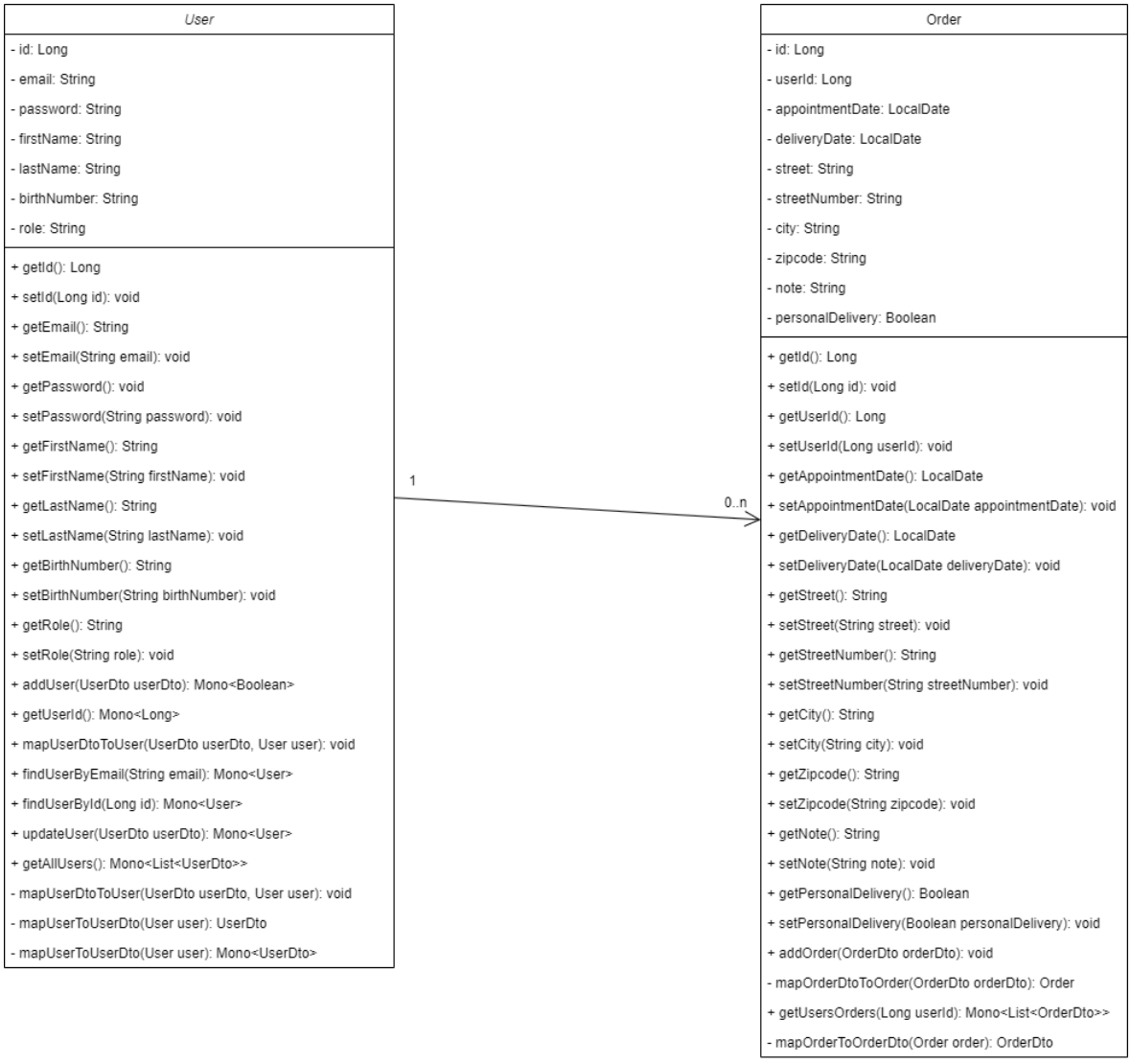
Model tried zobrazuje ako sa používajú dáta z databázy a aké metódy sú na to použité. Je to teda model implementácie, dátový model je náčrtom systému, zatiaľ čo model tried je úplný a jeho použitím zabezpečujeme, že aplikácia funguje.

Model tried sa nachádza v *Maven module Model*. Obsahuje triedy *User, Order* a *Document*. Rovnako obsahuje pomocné triedy *UserDto, OrderDto* a *DocumentDto*, ktoré sú kópiou predchádzajúcich a slúžia na prácu s modelom. Vznikli, aby bolo možné v budúcnosti pridať atribúty, ktoré nie je nutné ukladať do databázy a aby sa v systéme nepracovalo s databázovými triedami.

Trieda ***User*** reprezentuje užívateľa. Na obrázku 3.5 je vidno, že obsahuje rovnaké atribúty ako v databáze a teda *id, email, password, firstName, lastName, birthNumber* a *role*. Zahŕňa *public* - verejné *getter, setter* pre každý atribút, pretože všetky atribúty triedy sú *private* - privátne. Metóda *Mono<Boolean> addUser(UserDto userDto)* pridáva užívateľa do databázy. Vracia *Mono<Boolean>*, *Mono* pretože operácia v databáze je asynchrónna a *Boolean* zobrazuje, či sa operácia pridania podarila. *Mono<Long> getUserId()* vráti *id* prihláseného užívateľa. Metóda *void mapUserDtoToUser(UserDto userDto, User user)* príjma atribúty triedy *UserDto* a vloží ich do atribútov triedy *User*. *Mono<User> findUserByEmail(String email)* nájde a vráti užívateľa z databázy, ktorý má rovnaký email ako vstupný email. Metóda *Mono<User> findUserById(Long id)* nájde v databáze užívateľa podľa zadaného *id*. Úpravu užívateľových údajov a teda ich uloženie do databáze vykonáva metóda *Mono<User> updateUser(UserDto userDto)*. Všetkých užívateľov s rolou *USER* uložených v databáze vráti metóda *Mono<List<UserDto>> getAllUsers()*. Metóda *Mono<UserDto> getUser()* vráti práve prihláseného užívateľa. Privátnymi metódami sú *UserDto mapUserToUserDto(User user)*, *void mapUserDtoToUser(UserDto userDto, User user)* a *Mono<UserDto> mapUserToUserDto(User user)*, ktoré mapujú atribúty triedy *User*

do pomocnej triedy *UserDto* a opačne.

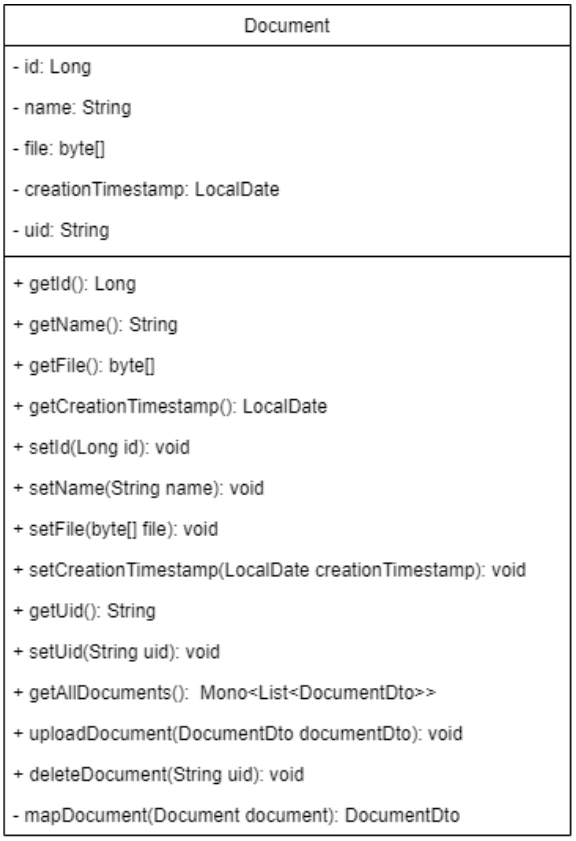
Trieda *Order* predstavuje objednávku užívateľa. Taktiež má rovnaké atribúty (*id, userId, appointmentDate, deliveryDate, street, streetNumber, city, zipcode, note* a *personalDelivery*) ako v databáze, ktoré je možné vidieť na obrázku 3.5. Takisto obsahuje *getter, setter* pre každý atribút. Metóda *void addOrder(OrderDto orderDto)* uloží objednávku do databázy a *Order mapOrderDtoToOrder(OrderDto orderDto)* vloží atribúty z inštancie *OrderDto orderDto* do triedy *Order*. Pre vyhľadanie a vrátenie objednávok konkrétneho užívateľa slúži metóda *Mono<List<OrderDto>> getUsersOrders(Long userId)*. Podľa parametru *userId*, teda identifikačného čísla užívateľa vyhľadá všetky jeho objednávky. Privátna metóda *OrderDto mapOrderToOrderDto(Order order)* namapuje atribúty z triedy *Order* do triedy *OrderDto*.



Obr. 3.5 Model tried *User, Order*

Model triedy ***Document***, na obrázku 3.6, predstavuje dokumenty, ktoré administrátor nahrá v administrátorskej časti aplikácie. Zobrazia sa potom v užívateľskej časti.

Rovnako ako v databázovom modeli, trieda obsahuje atribúty *id, name, file, creationTimestamp* a *uid*. Každý atribút má verejnú metódu *get* a *set*. Na prácu s dokumentami v rámci aplikácie slúžia verejné metódy *void uploadDocument(DocumentDto documentDto)*, ktorá uloží dokument do databázy. Vymazanie z databázy umožňuje metóda *void deleteDocument(String uid)* a *Mono<List<DocumentDto>> getAllDocuments()* vráti všetky uložené dokumenty. Privátna metóda *DocumentDto mapDocument(Document document)* slúži na naplnenie triedy *DocumentDto* atribútmi triedy *Document*.



Obr. 3.6 Model triedy *Document*

4 POPIS IMPLEMENTÁCIE WEBOVEJ APLIKÁCIE

Implementácia webovej aplikácie pre výdajňu zdravotníckych pomôcok pre diabetikov prebiehala v niekoľkých fázach. Ako prostredie pre implementáciu projektu bolo vybrané *IntelliJ IDEA* od firmy *JetBrains*. Mikroslužby boli implementované postupne, ako samostatné *Maven* moduly a teda reprezentujú jednotlivé fázy implementácie.

Všetky mikroslužby sú takzvane obalené jedným *Maven* modulom, ktorý obsahuje *pom.xml* - rodičovský *pom*. Ten obsahuje rodiča, teda *spring-boot-starter-parent*, ktorý určuje, že daný modul a jeho potomkovia využívajú Spring Boot v určitej verzii.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.8</version>
    <relativePath/>
</parent>
```

Určuje identifikátor pre celý modul *groupId* a identifikátor pre daný *Maven* modul *artifactId*.

```
<groupId>com.diatop</groupId>
<artifactId>diatop-web-app</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>pom</packaging>
<name>diatop-web-app</name>
<description>diatop web app</description>
```

Na koniec definuje *Maven* moduly, ktoré sa v danom rodičovi nachádzajú. Sú to *diatop* (UI mikroslužba), *resource* (mikroslužba zdroje), *gateway*, *admin* a *model*.

```
<modules>
    <module>diatop</module>
    <module>resource</module>
    <module>gateway</module>
    <module>admin</module>
    <module>model</module>
</modules>
```

4.1 Gateway

Gateway je vstupným modulom do aplikácie. Je *Maven* modulom, ktorý obsahuje *pom.xml* definujúci závislosti.

4.1.1 Spring Boot aplikácia Gateway

Aplikácia Gateway je samostatne spustiteľná aplikácia. To, že je Spring Boot aplikáciou určuje anotácia *@SpringBootApplication* pred názvom triedy.

```
@SpringBootApplication
@EnableRedisWebSession
public class GatewayApplication { }
```

Anotácia *@EnableRedisWebSession* hovorí, že daná aplikácia využíva *Redis* server ako správcu *session*.

V rámci triedy *GatewayApplication* je riešený *routing* požiadaviek. Každá požiadavka z prehliadača, ktorá príde do Gateway je priradená konkrétnej mikroslužbe.

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
   return builder.routes()
      .route("resource", p -> p.path("/resource/**")
         .filters(f -> f.rewritePath("/resource/(?<segment>.*)", "/$segment"))
         .uri("http://localhost:9000"))
      .route("ui", p -> p.path("/ui/**")
         .filters(f -> f.rewritePath("/ui/(?<segment>.*)", "/$segment"))
         .uri("http://localhost:8081"))
      .route("admin", p -> p.path("/admin/**")
         .filters(f -> f.rewritePath("/admin/(?<segment>.*)", "/$segment"))
         .uri("http://localhost:8082"))
      .build();
}
```

Každá požiadavka, ktorá má na začiatku URL adresy */resource/* je presmerovaná na mikroslužbu *Zdroje* a */resource/* časť je z URL adresy odňatá. Rovnaký postup je zvolený aj pre požiadavky na *UI* a *Admin* mikroslužby.

4.1.2 Konfigurácia

Konfigurácia Gateway je riešená pomocou dvoch tried *RedisConfiguration* a *SecurityConfiguration*. Aby Spring Boot vedel, že daná trieda je konfiguráciou, využíva sa anotácia *@Configuration*.

Konfigurácia *Redis* slúži na to, aby Gateway bolo schopné využívať a komunikovať s *Redis* serverom. Aby vôbec mikroslužba vedela používať anotácie a funkcionalitu *Redis*, musí byť definovaný v *pom.xml* pre Gateway.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.session</groupId>
   <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

Pomocou závislosti v *pom.xml* pre *Lettuce* je možné nakonfigurovať *Redis* v konfigurácii.

```
<dependency>
   <groupId>io.lettuce</groupId>
   <artifactId>lettuce-core</artifactId>
</dependency>
```

Konfigurácia pre *Redis* následne vytvorí *LettuceConnectionFactory*, čiže možnosť pripojenia sa na *Redis* v rámci *Gateway*. Toto pripojenie funguje cez určitý port, ten je nutné nastaviť. Nastavenie pre pripojenie *Redis* je v rámci *application.properties*.

Zabezpečenie Gateway je riešené v triede *SecurityConfiguration* rovnako s anotáciou *@Configuration*. *Bean SecurityWebFilterChain* ponúka definovanie bezpečnostných pravidiel. Jedným z týchto pravidiel je určenie URL adries, ktoré majú prístup do Gateway bez nutnosti prihlásenia.

```
.authorizeExchange()
.pathMatchers("/", "/index.html", "/registration", "/login", "/*.js", "/*.css", "/favicon.ico", "/assets/diatop.svg", "/*.map").permitAll()
.anyExchange().authenticated()
```

Pre autentifikáciu užívateľa je nutný *ReactiveAuthenticationManager*. Ten slúži na predávanie informácií o užívateľovi ako užívateľské meno a heslo. Tie sú kontrolované cez službu *ReactiveUserDetailsService*. Táto služba môže byť upravená podľa potrieb aplikácie tým, že sa vytvorí služba dediaca z nej. Novou službou, dediacou z *ReactiveUserDetailsService* je *UserDetailsService*. Rovnako je nutné definovať spôsob kódovania hesla užívateľa. Na tieto účely bola zvolená trieda *BCryptPasswordEncoder*.

```
@Bean
public ReactiveAuthenticationManager authenticationManager() {
    UserDetailsRepositoryReactiveAuthenticationManager authenticationManager =
        new UserDetailsRepositoryReactiveAuthenticationManager(userDetailsService);
    authenticationManager.setPasswordEncoder(passwordEncoder());
    return authenticationManager;
}
```

Jeho definícia je použitá v *SecurityWebFilterChain*. V rámci bezpečnosti je nutné nastaviť spôsob práce so *session* v *SecurityWebFilterChain*. Na tento účel slúži trieda *WebSessionServerSecurityContextRepository*. Tá sa nastavuje do *securityContextRepository*.

```
.securityContextRepository(webSessionServerSecurityContextRepository());
```

4.1.3 Služby

V Gateway sú dve služby a to *UserService* a *UserDetailsService*. Obe majú anotáciu *@Service*, aby Spring Boot vedel rozlíšiť, že sa jedná o službu a nie triedu.

UserService má vložené závislosti *UserRepository* a *PasswordEncoder* pomocou anotácie *@Autowired*. Definuje tri verejné a dve privátne metódy slúžiace na prácu s triedou *User* a dátami užívateľa v databáze. Verejnými metódami sú *public Mono<Boolean> addUser(UserDto userDto)*, *public Mono<Long> getUserId()* a *public Mono<UserDto> getUser()*. Pre pridanie užívateľa, teda uloženie užívateľa do databázy slúži metóda *addUser*.

```
public Mono<Boolean> addUser(UserDto userDto) {
    return userRepository.findUserByEmail(userDto.getEmail())
        .hasElement()
        .flatMap(userExists -> {
            if (userExists) {
                return Mono.just(false);
```

```
        } else {
            User user = new User();
            mapUserDtoToUser(userDto, user);
            return userRepository.save(user).map(createdUser -> true);
        }
    });
}
```

Metóda *addUser* nájde užívateľa v databáze pomocou emailovej adresy, ak užívateľ v databáze existuje, metóda vráti *false*, teda nový užívateľ nebol uložený do databázy, pretože už užívateľ s danou emailovou adresou existuje. Ak neexistuje, vráti *true*, čiže užívateľ bol úspešne uložený do databázy, čo znamená, že bol registrovaný. Na zistenie identifikačného čísla *id* práve prihláseného užívateľa sa používa metóda *getUserId*.

```
public Mono<Long> getUserId() {
    return ReactiveSecurityContextHolder.getContext()
        .map(SecurityContext::getAuthentication)
        .map(Principal::getName)
        .flatMap(username -> {
            if (username != null) {
                return userRepository.findUserByEmail(username)
                    .flatMap(user -> Mono.justOrEmpty(user.getId()));
            }
            return Mono.empty();
        });
}
```

Podobnou metódou je *getUser*, ktorá vráti prihláseného užívateľa, ktorého atribúty sú namapované do pomocnej triedy *UserDto*.

UserDetailsService dedí z triedy *ReactiveUserDetailsService*. Má vložené dve závislosti, *UserRepository* a *PasswordEncoder* pomocou anotácie *@Autowired*, ktorá vloží závislosť a jej metódy je možné použiť v rámci implementácie. Služba má jednu metódu *public Mono<UserDetails> findByUsername(String username)*, ktorá je deklarovaná v triede *ReactiveUserDetailsService* a teda definovaná v *UserService*.

```
@Override
public Mono<UserDetails> findByUsername(String username) {
    return userRepository.findUserByEmail(username).map(user -> User.builder()
```

```
.username(user.getEmail())
.password(user.getPassword())
.roles(user.getRole())
.build()); }
```

Táto metóda nájde pomocou *UserRepository findByEmail* užívateľa na základe jeho užívateľského mena - emailovej adresy a nastaví atribúty *username, password, role* (užívateľské meno, heslo a rolu) triedy *UserDetails*. Práve vďaka konfigurácii bezpečnosti a *ReactiveAuthenticationManager* Spring Boot dokáže autentifikovať užívateľa a povoliť mu prístup do aplikácie.

4.1.4 Spracovanie REST požiadaviek

O REST požiadavky sa stará trieda *GatewayController*, ktorej úlohu spozná Spring Boot vďaka anotácii *@RestController*. Má vloženú závislosť služby *UserService* pomocou anotácie *@Autowired*. Metóda *public UserPrincipal user(Principal user)* je GET požiadavkou a vracia hlavné informácie o užívateľovi *UserPrincipal*, pokiaľ je užívateľ registrovaný v aplikácii.

```
@RequestMapping("/user")
@ResponseBody
public UserPrincipal user(Principal user) {
   UserPrincipal userPrincipal = new UserPrincipal();
   userPrincipal.setName(user.getName());
   userPrincipal.setRoles(AuthorityUtils.authorityListToSet(((Authentication) user)
      .getAuthorities()));
   return userPrincipal;
}
```

Metóda *getUserId()* vráti identifikačné číslo *id* práve prihláseného užívateľa a metóda *public Mono<Long> getUserId()* vráti prihláseného užívateľa. Pridanie užívateľa (registrácia) je realizované pomocou REST požiadavky *addUser*. Je to POST požiadavka, ktorá v tele požiadavky posiela triedu *UserDto*, ktorá obsahuje informácie o užívateľovi a vracia odpoveď, či bol užívateľ úspešne vytvorený, alebo už existuje.

```
@PostMapping("/addUser")
@ResponseBody
public Mono<Boolean> addUser(@RequestBody UserDto user) {
   return userService.addUser(user); }
```

4.1.5 Užívateľské rozhranie

UI Gateway obsahuje stránku prihlásenia a registrácie. Po prihlásení presmeruje užívateľa buď do časti pre administrátora, alebo pre užívateľa. Hlavnou stránkou je *index.hmtl*, ktorý následne dynamicky zobrazuje ďalšie stránky bez nutnosti znovunačítania. Koreňovou komponentou je *AppComponent*. Využíva presmerovanie *routing* definovaný v *app-routing.module.ts* pocou Angular *Route*, ktorý reprezentuje konfiguráciu presmerovacej služby. Na to, aby presmerovanie fungovalo, musí *app.component.html* obsahovať direktívu

<router-outlet></router-outlet>

ktorá vraví o tom, kde sa budú zobrazovať komponenty, ktoré sa navigujú cez router. Týmito komponentami sú *LoginComponent* a *RegistrationComponent*. Tie sú deklarované v *app.module.ts* spolu s importami modulov, ktoré komponenty deklarované v tomto module využívajú. Pod komponentu *AppComponent* spadá *app.component.ts*, ktorý definuje selektor komponenty (pod akým názvom vieme komponentu zavolať v html súbore, aby sa na danom mieste zobrazila), URL šablóny (html súboru) a samotnú definíciu komponenty, čiže všetky jej metódy.

```
@Component({
   selector: 'app-root',
   templateUrl: 'app.component.html'
})
export class AppComponent {
   constructor(private matIconRegistry: MatIconRegistry,
      private domSanitizer: DomSanitizer){
   this.matIconRegistry.addSvgIcon('diatop',
   this.domSanitizer.bypassSecurityTrustResourceUrl(
      '$window.location.origin/assets/diatop.svg'));
   }
}
```

Na koniec registruje ikonu *diatop.svg* ako *MatIcon* pomocou *MatIconRegistry*. Po tomto je možné pomocou direktívy *mat-icon* napísať v html len *diatop* a ikona sa zobrazí.

Komponenta zabezpečujúca prihlásenie sa nazýva *LoginComponent*. Skladá sa z *login.component.ts*, *login.component.html*, *login.component.scss* a *login.service.ts*. Typescript súbor *login.component.ts* definuje komponentu, selektor a šablónu. V konštruk-

tore sa nachádza služba *LoginService* ako *app* a *FormBuilder* ako *formBuilder*. Kontrola pomocou *app.authenticated* zistí, či už je užívateľ prihlásený. Ak áno, potom je užívateľ priamo presmerovaný cez definovanie URL do */ui/* pokiaľ je užívateľ, alebo */admin/* ak je užívateľ administrátorom. Pri inicializácii nastáva inicializácia *form*, ktorá je zobrazená v *login.component.html*.

```
ngOnInit(): void {
   this.formLogin = this.formBuilder.group({
      username: new FormControl('', [Validators.required, Validators.email]),
      password: new FormControl('', Validators.required)
   });
}
```

Validators.required znamená, že *form* vie, že toto pole je nutné vyplniť a bez toho sa užívateľ neprihlási. *Validators.email* zabezpečí, že toto vstupné pole musí mať tvar emailovej adresy. Po kliknutí na tlačítko *Prihlásiť* sa zavolá metóda *login()*. Tá vezme naplnené atribúty *formLogin* a naplní ich do atribútov triedy *Credentials*. Zavolaním metódy *authenticate()* zo služby *LoginService* prebehne pokus o prihlásenie. Ak je úspešný, presmeruje podľa roly užívateľa do odpovedajúcej časti aplikácie.

```
login() {
   if (this.formLogin.invalid) {
      return;
   }
   this.credentials.username = this.formLogin.get('username')?.value;
   this.credentials.password = this.formLogin.get('password')?.value;
   this.app.authenticate(this.credentials, () => {
      this.app.isAdmin ? window.location.href = '/admin/'
         : window.location.href='/ui/';
   });
   return false;
}
```

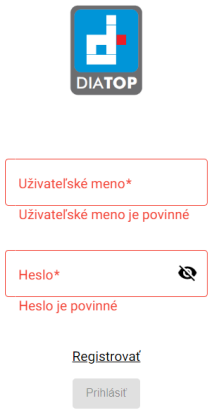
Služba *login.service.ts* zaobstaráva poslanie prihlasovacích údajov užívateľa do Spring Boot časti. Realizácia je cez *HttpClient*.

```
this.http.get<UserPrincipal>('/user', {headers: headers})
```

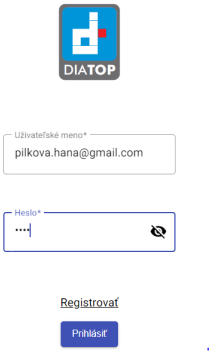
Hlavičky *headers* obsahujú autorizáciu *authorization*, ktorá zakóduje prihlasovacie údaje pomocou *Base64*.

```
const headers = new HttpHeaders(credentials ? {
    authorization : 'Basic ' + btoa(credentials.username + ':' + credentials.password)
} : {});
```

Šablóna *login.component.html* je tvorená pomocou Angular Material Design. Na obrázku 4.1 je ukázaný pokus užívateľa prihlásiť sa s nesprávnym formátom prihlasovacích údajov. Opakom je obrázok 4.2, ktorý zobrazuje pokus o prihlásenie so správnymi údajmi.



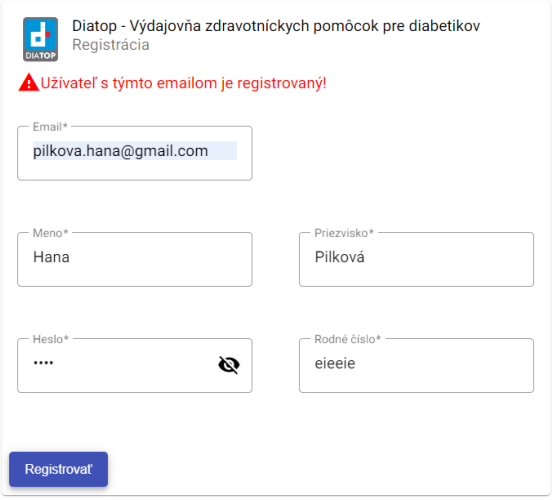
Obr. 4.1 Stránka prihlasovania s chybovými hláškami



Obr. 4.2 Stránka prihlasovania

Registrácia je vykonávaná po kliknutí užívateľa na link *Registrovať* v časti prihlásenia. Skladá sa z *registration.component.ts*, *registration.component.html* a *registration.co-*

mponent.scss. Pri inicializácii v *registration.component.ts* nastáva prvotné nastavenie a vytvorenie *formRegistration* pomocou *FormBuilder*. Validátor určuje validačné pravidlá ako povinné vstupné pole a užívateľské meno musí byť v tvare emailovej adresy. Po kliknutí na tlačítko *Registrovať*, pošle registračné údaje do Spring Boot časti aplikácie pomocou HTTP POST */addUser*. Pokiaľ je užívateľ úspešne registrovaný, je presmerovaný na */login*. Ak užívateľ s rovankým emailom už existuje, zobrazí sa varovná hláška, ktorú je možné vidieť na obrázku 4.3.



Obr. 4.3 Neúspešná registrácia

4.2 Admin

Admin je *Maven* modulom, ktorý sa skladá zo Spring Boot časti aplikácie a UI časti, určenej pre užívateľa s rolou *ADMIN*, čiže administrátor.

Rovnako ako Gateway obsahuje *pom.xml*, ten udáva jeho identifikáciu a závislosti. Aplikácia *AdminApplication* podporuje *Redis session* anotáciou *@EnableRedisWebSession* a je definovaná ako Spring Boot Aplikácia anotáciou *@SpringBootApplication*.

4.2.1 Konfigurácia

Konfigurácie v Admin mikroslužbe sú *RedisConfiguration*, rovnako ako v Gateway obsahuje anotácie *@Configuration, @EnableRedisWebSession* a *@Bean public LettuceConnectionFactory redisConnectionFactory()* pre tvorbu pripojenia na Redis server. Bezpečnostná konfigurácia *SecurityConfiguration* je rovnako s anotáciou *@Configuration* a podporou Redis *session @EnableWebFluxSecurity*. Bezpečnosť je riešená v *@Bean public SecurityWebFilterChain filterChain(ServerHttpSecurity http)*.

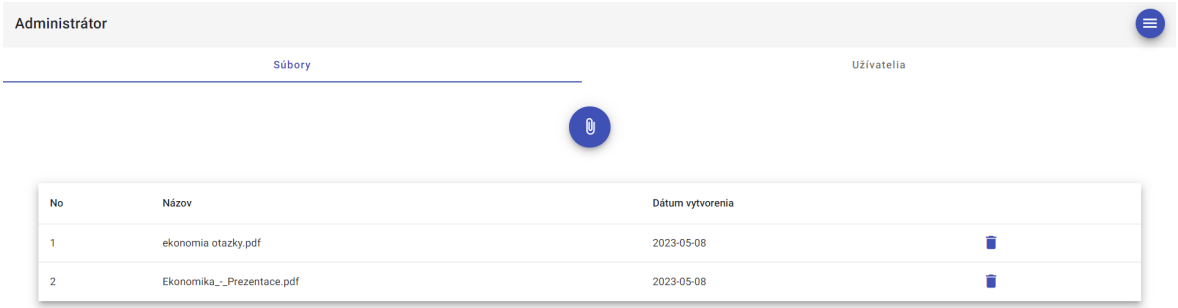
```
@Bean
public SecurityWebFilterChain filterChain(ServerHttpSecurity http) {
    http
        .authorizeExchange()
        .pathMatchers("/**", "/admin/**").hasRole("ADMIN")
        .and()
        .csrf()
        .csrfTokenRepository(CookieServerCsrfTokenRepository.withHttpOnlyFalse());
    return http.build();
}
```

Každú URL adresu, ktorá sa zhoduje s *pathMatchers()* a rola užívateľa je *ADMIN*, nie je nutné autorizovať. Prenos atribútu požiadavky *XSRF-TOKEN* do súboru *cookie* zabezpečuje *csrfTokenRepository(CookieServerCsrfTokenRepository.withHttpOnlyFalse())*.

4.2.2 Užívateľské rozhranie

Stavebným kameňom užívateľského rozhrania je *index.html*. Nachádzajú sa tu komponenty *AppComponent, HomeComponent, DocumentsComponent* a *UsersPreviewComponent*. Komponenta *AppComponent* len používa direktívu *router-outlet* na dynamické zobrazovanie komponent pomocou *routing* presmerovania.

Hlavnou komponentou je *HomeComponent*, ktorá sa skladá z *home.component.ts* a *home.component.html*. Typescriptový súbor len definuje selektor, šablónu a rieši odhlásenie. V *home.component.html* je definované rozvrhnutie výzoru aplikácie. Obsahuje horný panel s tlačítkom pre menu a dve záložky zobrazujúce dokumenty a užívateľov. Spomenuté časti je možné vidieť na obrázku 4.4.



Obr. 4.4 Časť pre administrátora

Komponenta *DocumentsComponent* zobrazuje dokumenty, ktoré už boli nahrané, ponúka možnosť nahrať súbor pomocou tlačítka a vymazať už uložený súbor. V *documents.component.ts* sa pri inicializácii pomocou GET získajú všetky doposiaľ uložené

súbory.

```
private initDocuments() {
   this.httpClient.get<Document[]>('/resource/getAllDocuments')
      .subscribe(response => {
         this.documentsToDisplay.data = response
            .map((document: Document, index) =>
         this.mapDocument(index, document));
   })
}
```

Pomocou *private mapDocument(index: number, document: Document)* sú potom atribúty každého dokumentu triedy *Document* namapované do pomocnej triedy *DocumentToDisplay* deklarovanej v *document.component.ts*.

```
export class DocumentToDisplay extends Document {
   position: number = 1;
}
```

Vymazanie dokumentu zabezpečuje metóda *onDeleteDocument(doc: DocumentToDisplay)* zavolaná po kliknutí na tlačítko s ikonou koša. Funguje tak, že zavolá POST s identifikačným číslom dokumentu *uid*. Pokiaľ prebehlo vymazanie bez chyby, teda dokument bol úspešne vymazaný, potom sa v spodnej lište zobrazí *snackbar* s touto informáciou. Ak vymazanie nenastalo, ale POST vráti chybu, ten sa taktiež zobrazí v *snackbar*.

```
onDeleteDocument(doc: DocumentToDisplay) {
   this.httpClient.post('/resource/deleteDocument', doc.uid).subscribe({
      error: error => {
         this.openSnackBar(error);
      },
      complete: () => {
         this.documentsToDisplay.data.splice(this.documentsToDisplay.data
         .indexOf(doc),1)
         this.documentsToDisplay.data = this.documentsToDisplay.data;
         this.openSnackBar('Súbor s názvom ' + doc.name + ' bol úspešne vymazaný.')
      }
   }) }
```

Nahrávanie súboru rovnako funguje cez POST v metóde *onFileUpload(event: Event)*, ktorá sa zavolá po kliknutí na tlačítko pre nahranie súboru. Ak nahranie súboru bolo úspešné, alebo skončilo s nejakou chybou, obe informácie sa zobrazia pomocou *snackbar*. Samotný súbor je súčasťou *Event event*, ktorý je parametrom metódy *onFileUpload*. Všetky súbory, ktoré užívateľ vybral že chce nahrať, sú v *HTMLInputElement* atribúte *files*.

```
const element = event.currentTarget as HTMLInputElement;
let files: FileList | null = element.files;
```

Nahráva sa prvý súbor z *files* a pokiaľ existuje pomocou triedy *FormData* je poslaný cez POST do Spring Boot časti.

```
if (file) {
   const formData = new FormData();
   formData.append('file', file);
   this.httpClient.post<Document>('/resource/uploadDocument', formData)
}
```

Dokumenty sú zobrazené v tabuľke použitím *mat-table*. Tabuľke z Angular Material Design je predaný pomocou *one-way-binding* zdroj dát, čiže *documentsToDisplay*.

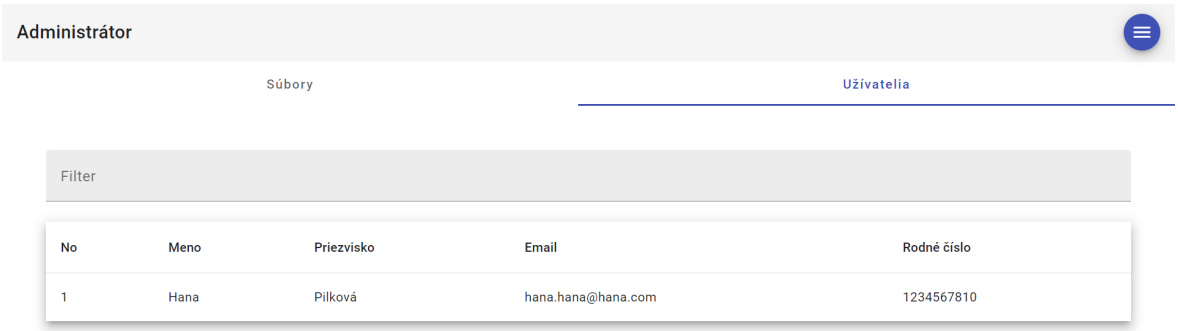
```
<table mat-table [dataSource]="documentsToDisplay"class="mat-elevation-z8">
```

Jeden stĺpec tabuľky je tvorený *matColumnDef* - definícia stĺpca tabuľky, *mat-header-cell* názov stĺpca a *mat-cell* bunka tabuľky, kde sa zobrazia dáta.

```
<ng-container matColumnDef="position">
   <th mat-header-cell *matHeaderCellDef> No </th>
   <td mat-cell *matCellDef="let element"> {{element.position}} </td>
</ng-container>
```

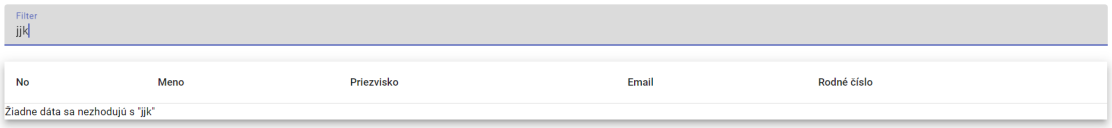
Všetci registrovaní užívatelia sú zobrazení v záložke *Užívatelia* pomocou *user-preview .component.ts*, *user-preview.component.html* a *user-preview.component.scss*. V typescript súbore pri inicializácii je volaný GET */resource/getAllUsers*, ktorý vráti všetkých užívateľov s rolou *USER*. Napĺňajú sa tu dáta pre tabuľku v html, práve týmito užívateľmi. Tabuľka je *mat-table* s filtrom na filtrovanie užívateľov, ktorú je možné spolu s filtrom vidieť na obrázku 4.5. Tí sa filtrujú metódou *applyFilter(event: Event)*.

```
applyFilter(event: Event) {
    const filterValue = (event.target as HTMLInputElement).value;
    this.usersToDisplay.filter = filterValue.trim().toLowerCase();
}
```



Obr. 4.5 Časť pre administrátora, zobrazenie užívateľov

Pokiaľ filter nenájde zhodu v *usersToDisplay* zobrazí hlášku v tabuľke, viď obrázok 4.6.



Obr. 4.6 Časť pre administrátora, zobrazenie užívateľov s použitím filtra

Pokiaľ však zhodu nájde, v tabuľke sa zobrazia výsledky zhodné s filtrom.

4.3 UI

UI mikroslužba je časť, kde sa pohybuje užívateľ v rámci užívateľského rozhrania. Rovnako ako predošlé mikroslužby je *Maven* modulom s vlastným *pom.xml*. Trieda *DiatopApplication* má takisto *@SpringBootApplication* a *@EnableRedisWebSession* anotácie.

4.3.1 Konfigurácia

Konfigurária UI mikroslužby sa skladá z dvoch tried. Rovnako, ako v predošlých mikroslužbách, obsahuje *RedisConfiguration* s anotáciou *@Configuration* a *@EnableRedisWebSession*. Vytvára pomocou *LettuceConnectionFactory* pripojenie k Redis server a teda využitie Redis ako úložný priestor pre *session*.

```
@Bean
public LettuceConnectionFactory redisConnectionFactory() {
```

return new LettuceConnectionFactory(); }

Druhou konfiguráciou je konfigurácia bezpečnosti. Takisto má anotáciu *@Configuration* a *@EnableWebFluxSecurity*. Vstup do tejto mikroslužby majú len užívatelia s rolou *USER*, kde URL požiadavka splňuje vzor.

*.pathMatchers("/**", "/ui/**").hasRole("USER")*

4.3.2 Užívateľské rozhranie

Užívateľské rozhranie UI mikroslužby je najviac obsiahle, či už množstvom funkcií alebo komponent. Je hlavnou časťou aplikácie, kde sa pohybuje užívateľ a interaguje s celou aplikáciou. Rovnako ako v predchádzajúcich mikroslužbách, jeho stavebným kameňom je *index.html*, v ktorom tele je

<app-root></app-root>

čiže hlavná komponenta *app.component*. Tá slúži na *routing* (dynamické zobrazovanie komponent) a importovanie vlastnej ikony spoločnosti Diatop. Rovnako obsahuje hlavný modul celého užívateľského rozhrania *app.module.ts* s anotáciou *@NgModule*. *Routing* zabezpečuje *app-routing.module.ts*. Tento modul definuje *Routes*, teda podľa URL vzoru Angular vie, akú komponentu má užívateľovi zobraziť a v URL použiť danú príponu.

```
const routes: Routes = [
   { path: '', pathMatch: 'full', redirectTo: '/home'},
   { path: 'home', component: HomeComponent},
   {path: 'account', component: AccountInfoComponent}
];
```

A teda, pokiaľ URL končí */home*, potom je zobrazená komponentna *HomeComponent*, rovnaký princíp je pre komponentu *AccountInfoComponent*.

Hlavnou komponentou, kde sa väčšinu času užívateľ v aplikácii nachádza, je *HomeComponent*. Skladá sa z *home.component.html* (šablóna), *home.component.scss* (štýly) a *home.component.ts* (logika komponenty). Šablóna *home.component.html* vytvára štruktúru výzoru hlavnej časti aplikácie. Obsahuje niekoľko komponent z Angular Material Design, ktoré rozdeľujú stránku na *mat-toolbar* s menu tlačítkom a ikonou firmy Diatop a *mat-tab* ako dve záložky, ktoré zobrazujú dokumenty. Rovnako sa tu nachádza tla-

čítko vo forme zelenej ikonky, ktorá otvorí samostatné okno na vytvorenie objednávky. Konkrétny výzor domovskej stránky je viditeľný na obrázku 4.7.

Obr. 4.7 Domovská stránka

Záložky *Najnovší príspevok* a *Všetky príspevky* zobrazujú dokumenty nahrané do aplikácie v administrátorskom prostredí. *Najnovší príspevok* je vyobrazený komponentou *PdfViewerComponentSingle*, ktorá dedí z abstraktnej triedy *PdfViewerClass*, rovnako ako komponenta *PdfViewerComponentAll*, ktorá zobrazuje všetky dokumenty v záložke *Všetky príspevky*. Týmto komponentám ako *@Input* pomocou *one-way binding* prenesie *HomeComponent* všetky súbory, ktoré sú uložené v databáze. Tie získa pomocou GET, ktorý sa volá pri inicializácii *HomeComponent* a to z toho dôvodu, aby toto volanie nebolo vyvolávané v rámci komponent zobrazujúcich dokumenty. Keďže medzi týmito komponentami sa užívateľ môže ľubovoľne často preklikávať, znamenalo by to, že by sa neustále volal GET na získanie dokumentov, čo je neefektívne a mohlo by viesť k spomaleniu aplikácie.

```
this.http.get<Document[]('/resource/getAllDocuments').subscribe(response => {
    this.documents = response;
})
```

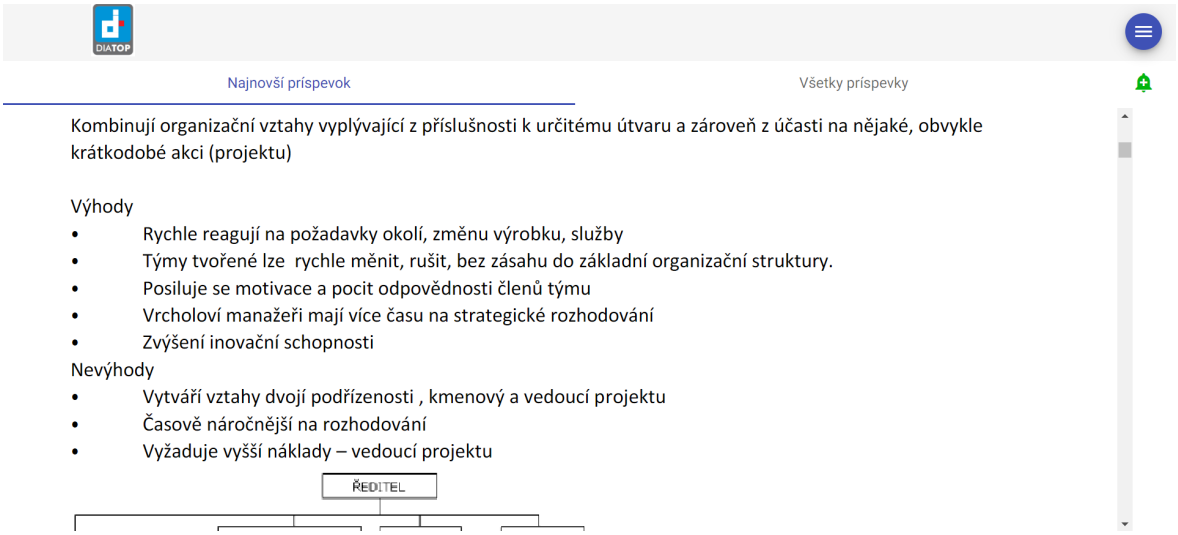
Abstraktná trieda *PdfViewerClass* obsahuje logiku pre obe komponenty, ktoré zobrazujú dokumenty. Má premennú *noPdfToDisplay* typu *boolean*, ktorá reprezentuje, či existuje nejaký dokument, ktorý sa dá zobraziť. Premenná *pdfDocumentToDisplay* typu *PdfDocument* je naplnená prvým dokumentom, ktorý sa zobrazí užívateľovi. Všetky dokumenty, ktoré je možné zobraziť drží premenná *pdfDocuments*, ktorá je typu pole *PdfDocument*. Trieda *PdfDocument* je pomocnou triedou tejto abstraktnej triedy, obsahuje atribúty *name* meno dokumentu a *file* typu *string*, kde reprezentácia dokumentu je v *Base64*. Prípravu dokumentov na samotné zobrazenie zabezpečuje metóda *public prepareDocuments(documents: Document[])*. Táto metóda má ako vstupný parameter pole *Document*. Pokiaľ nie je prázdne, tak namapuje atribúty nutné na zobrazenie dokumentu (meno a *Base64* reprezentáciu súboru) do triedy *PdfDocument* metódou *public mapDocument(document: Document)*, ktorej výsledok dá do premennej *pdfDocuments*. Následne nastaví premennú *pdfDocumentToDisplay* ako prvý dokument v poli *pdfDocuments* a *noPdfToDisplay* na *false*, pretože existuje minimálne jeden dokument na zobrazenie. Pokiaľ však je vstupné pole *Document* prázdne, premenná sa nastaví na *true*, pretože žiadne dokumenty, ktoré by sa dali zobraziť, neexistujú.

```
public prepareDocuments(documents: Document[]) {
    if (documents != null && documents.length > 0) {
        this.pdfDocuments = documents.map(doc => this.mapDocument(doc));
        this.pdfDocumentToDisplay = this.pdfDocuments[0];
        this.noPdfToDisplay = false;
    } else {
        this.noPdfToDisplay = true;
    }
}
```

Táto metóda sa volá z komponent *PdfViewerComponentAll* a *PdfViewerComponentSingle* vždy pri zmene vstupu *documents*.

```
@Input() documents: Document[] = [];
ngOnChanges(): void {
    this.prepareDocuments(this.documents);
}
```

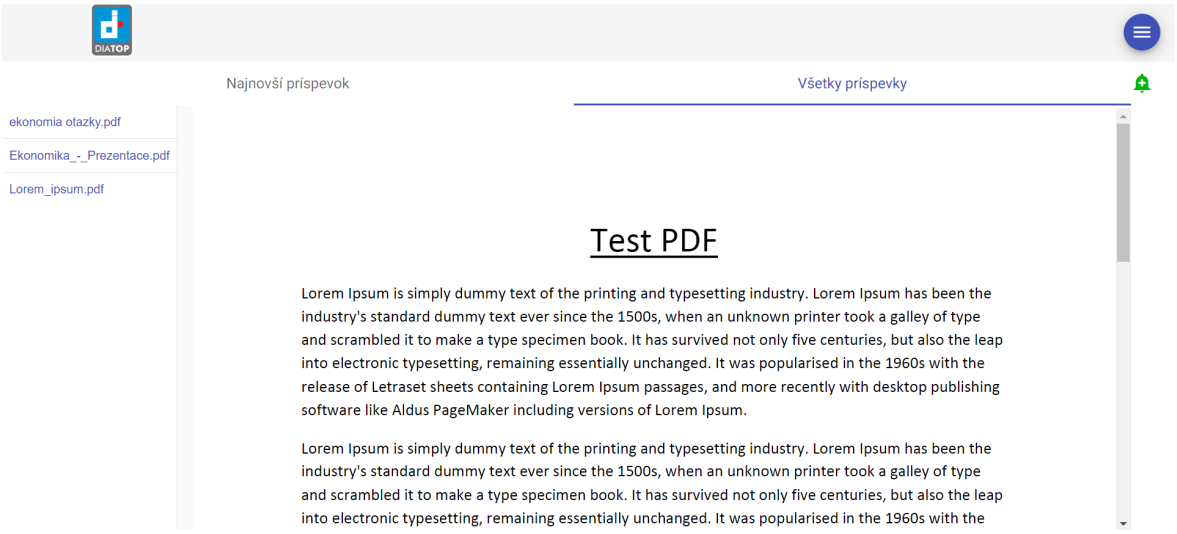
Zobrazenie dokumentu je pomocou externej knižnice *ngx-extended-pdf-viewer*. V šablóne komponenty *PdfViewerComponentSingle* je iba jednoducho použitá. Zobrazí dokument spolu s možnosťou približovať a odďalovať dokument, viď 4.9. Nie je možné text kopírovať, čo bola jedna z požiadaviek firmy Diatop.



Obr. 4.8 Domovská stránka s najnovším dokumentom

V šablóne na zobrazenie všetkých dokumentov *PdfViewerComponentAll* je externá knižnica vsadená do Angular Material Design komponenty *mat-drawer-container*. Ten

ponúka bočný panel, s ktorým je možné interagovať, viditeľný na obrázku 4.9. V tomto prípade sú v bočnom paneli zobrazené názvy dokumentov, medzi ktorými je možné preklikávať.



Obr. 4.9 Domovská stránka so všetkými dokumentami

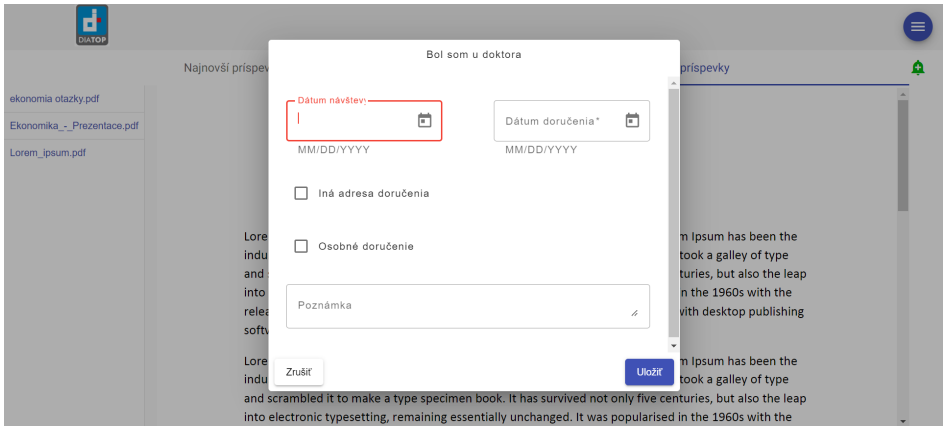
Tlačítko na vytvorenie objednávky otvorí modalové okno a to tak, že sa na kliknutie zavolá v *home.component.html* metóda z *home.component.ts* s názvom *openOrderRequestDialog()*. Dialóg je opäť komponentou z Angular Material Design a musí byť v *app.module.ts* registrovaný ako *entryComponent*. V konštruktore komponenty *HomeComponent* je vložená závislosť na *MatDialog*. Ten sa potom použije na ovládanie dialógu objednávky. Metóda *openOrderRequestDialog()* vytvorí nový *MatDialog* a nastaví parametre ako vstupné dáta. Vytvorí výstup dialógu *dialogOutput* a pomocou metódy v *MatDialog open* otvorí komponentu *OrderRequestDialogComponent*, ktorá predstavuje dialóg s jeho nastavením *dialogConfig*.

```
const dialogConfig = new MatDialogConfig();
dialogConfig.disableClose = true;
dialogConfig.autoFocus = true;
dialogConfig.data = this.userId;
const dialogOutput = this.dialog.open(OrderRequestDialogComponent, dialogConfig);
```

Po zatvorení dialógu pomocou metódy *afterClose*, taktiež z *MatDialog*, prichádza reakcia na vyplnené dáta v dialógu, ktoré sa pomocou POST pošlú do Spring Boot časti aplikácie a sú uložené do databázy.

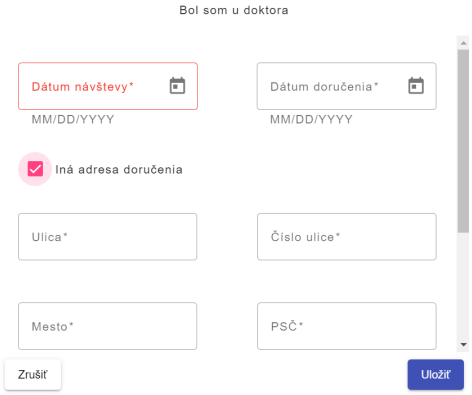
```
dialogOutput.afterClosed().subscribe(order =>
  this.http.post<Order>("/resource/addOrder", order) .subscribe({
    error: error => {
      console.log('Order error', error);
  }
}));
```

Samotná komponenta *OrderRequestDialogComponent* využíva pre prácu s dátami *@Inject(MAT_DIALOG_DATA)*, ktoré boli vložené v *home.component.ts openOrderRequestDialog()*. Obsahuje *mat-dialog-content* a *mat-form-field* ako *form* na zadávanie objednávok. Tento základ dialógu je možné vidieť na obrázku 4.10.



Obr. 4.10 Dialóg pre zadanie objednávky

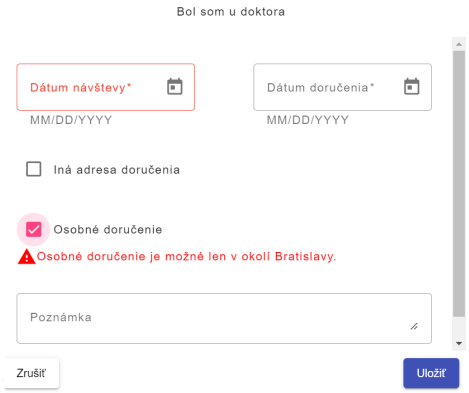
Pokiaľ chce užívateľ zadať inú adresu, ako tá, kde mu štandardne chodí materiál, využije možnosť *Iná adresa doručenia*, kedy sa mu otvorí ďalšia časť dialógu, viď 4.11.



Obr. 4.11 Dialóg pre zadanie objednávky s inou doručovacou adresou

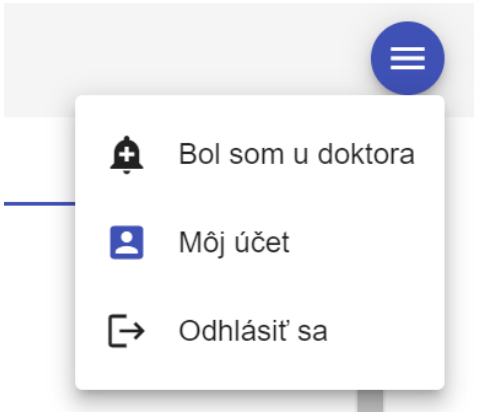
Alebo môže zvoliť možnosť osobnej objednávky, tá je však možná len v okolí Bratislavy,

na čo je užívateľ upozornený. To je možné vidieť na obrázku 4.12.



Obr. 4.12 Dialóg pre zadanie objednávky s osobným doručením

Poslednou dôležitou časťou domovskej stránky je menu tlačítko. To ponúka tri možnosti, *Bol som u doktora*, *Môj účet* a *Odhlásiť sa*, viď obrázok 4.13.



Obr. 4.13 Menu

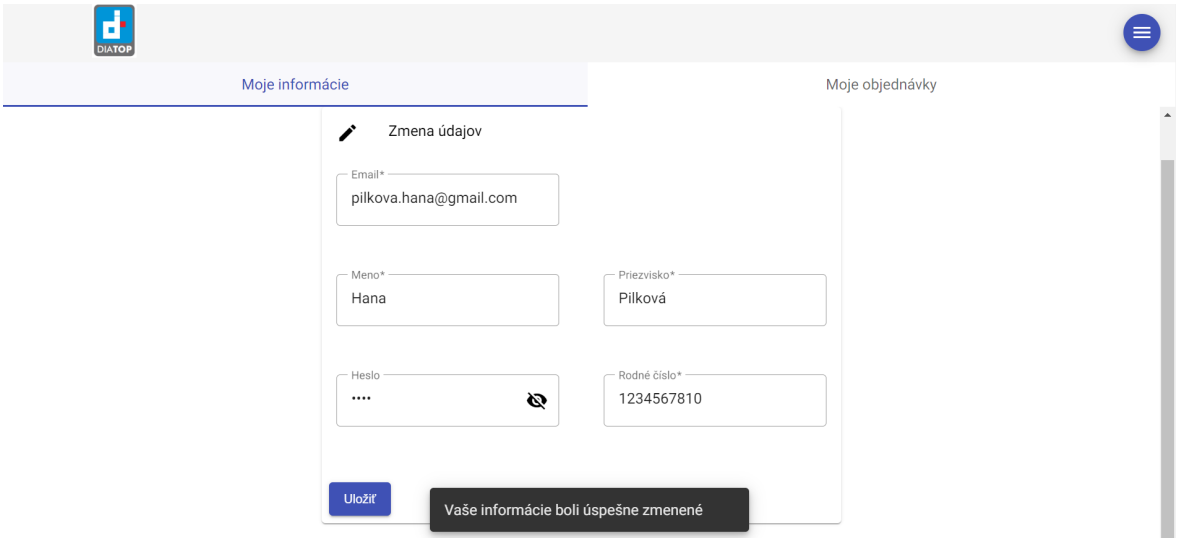
Bol som u doktora otvorí dialóg pre zadávanie objednávky. Možnosť *Odhlásiť sa* odhlási užívateľa a presmeruje na stránku prihlásenia. Poslednou možnosťou je *Môj účet*, ktorý presmeruje užívateľa na správu účtu.

Správa účtu sa skladá rovnako ako domovská stránka z *toolbar* a *mat-tabs*. Záložky v tejto časti sú *Moje informácie* a *Moje objednávky*. Správu účtu má na starosti komponentna *AccountInfoComponent*. Pri jej inicializácii zavolá GET, ktoré vráti práve prihláseného užívateľa s jeho osobnými informáciami. Pokiaľ nie je užívateľ prázdny a teda GET vrátil užívateľa, zavolá sa GET, ktorý vráti všetky objednávky užívateľa. Tie sú potom zobrazené v záložke *Moje objednávky*.

```
private initUser() {
    this.httpClient.get<User>('/getUser').subscribe({
        next: response => {
            this.user = response;
            this.initOrders(response.id);
        },
    })
}

private initOrders(userId: number) {
    const params = new HttpParams() .set('userId', userId);
    this.httpClient.get<Order[]>('/resource/getUsersOrders', {params: params})
    .subscribe(response => {
        this.orders = response;
    })
}
```

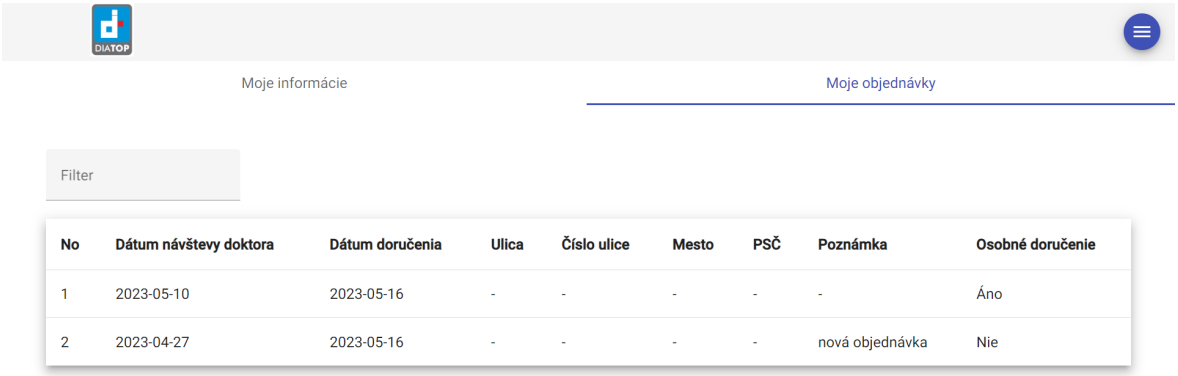
Komponenta zobrazujúca užívateľove údaje *UserFormComponent* využíva *form* s komponentami z Angular Material Design, rovnako ako komponenta obstarajúca registráciu. Keď užívateľ uloží dáta, pokiaľ sa zmenili, do Spring Boot časti aplikácie sa pošlú pomocou POST a uložia sa do databázy. Po tejto akcii, ak bola úspešná, zobrazí sa *MatSnackBar*, ktorý informuje užívateľa, že všetko prebehlo v poriadku ako na obrázku 4.14.



Obr. 4.14 Užívateľské informácie

Objednávky užívateľa prezentuje komponenta *OrdersComponent* s použitím tabuľky

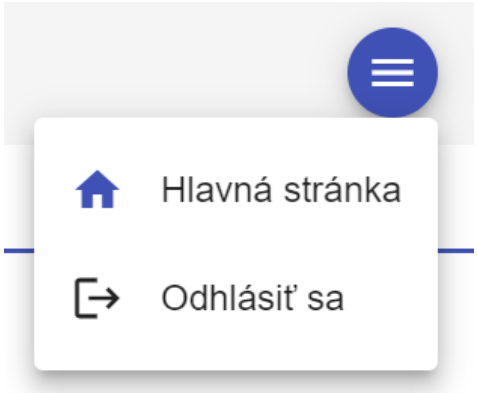
MatTable z Angular Material Design. Objednávky užívateľa sú predané do komponenty pomocou *@Input()*. Potom sú dáta z predaných objednávok zobrazené v tabuľke, viď obrázok 4.15.



Obr. 4.15 Užívateľské informácie

Z tejto časti sa užívateľ môže dostať dvoma spôsobmi. Prvý je kliknutie na ikonu výdajne Diatop, alebo v menu klikne na možnosť *Hlavná stránka*. Spomenuté možnosti je možné vidieť na obrázku 4.16. Túto akciu zabezpečuje *AccountInfoComponent*.

```
onHomeClick() {
    this.router.navigateByUrl('/home');
}
```



Obr. 4.16 Menu v časti Užívateľské informácie

4.4 Zdroje

Mikroslužba Zdroje zabezpečuje zdroje pre jednotlivé mikroslužby. Nemá užívateľské rozhranie.

4.4.1 Konfigurácia

Ako jediná mikroslužba, ktorá prijíma požiadavky od prehliadača, nepoužíva WebFlux, ale MVC, pretože nemá užívateľské rozhranie. Preto komunikácia s Gateway, ktorá nutne WebFlux vyžaduje, je len vo forme presmerovania požiadavky vo forme REST GET a POST. Štandardne obsahuje konfiguráciu pre spojenie s Redis serverom.

Konfigurácia bezpečnosti sa teda, práve kvôli MVC, líši od predchádzajúcich. Obsahuje štandardnú anotáciu konfigurácie *@Configuration*. Pre povolenie bezpečnosti je použitá anotácia *@EnableWebSecurity*. Určenie repozitára databázy pre prístup k modelu a databázovým repozitárom je anotáciou *@EnableR2dbcRepositories*.

@EnableR2dbcRepositories(basePackages = "com.diatop.model")

Samotné nastavenie bezpečnosti je veľmi jednoduché. Metóda *public SecurityFilterChain filterChain(HttpSecurity http) throws Exception* využíva *http builder*, kedy nastaví *csrfTokenRepository* a *session*.

.sessionManagement()
.sessionCreationPolicy(SessionCreationPolicy.NEVER);

Nastavenie *SessionCreationPolicy.NEVER* vraví, že daná mikroslužba **nemá** vytvárať novú *session*, ale pracovať s tou, čo má Redis server uloženú ako aktívnu.

4.4.2 Služby

Pretože Zdroje mikroslužba poskytuje zdroje ostatným častiam aplikácie, obsahuje aj najviac služieb. Týmito službami sú *UserService*, *OrderService*, *DocumentService* a *EmailService*.

UserService poskytuje zdroje, teda informácie o užívateľovi a ich zmenu. Má vloženú závislosť pomocou *@Autowired* na *UserRepository* a *PasswordEncoder*. Obsahuje metódu *public Mono<List<UserDto>> getAllUsers()*, ktorá vráti všetkých užívateľov a informácie o nich. Metóda *public Mono<User> updateUser(UserDto userDto)*, ktorá aktualizuje užívateľove dáta predané metóde ako parameter *UserDto userDto*. Tá nájde užívateľa pomocou metódy z *UserRepository*, aktualizuje jeho údaje a uloží ich, znovu metódou z *UserRepository*. Pokiaľ by bolo heslo prázdne, alebo dokonca *null*,

nezmení ho. To zabezpečuje, aby užívateľ mal vždy heslo, aj keď by tento stav nemal vďaka validáciam v UI časti nastať.

```
public Mono<User> updateUser(UserDto userDto) {
    return userRepository.findUserById(userDto.getId())
        .flatMap(existingUser -> {
            existingUser.setFirstName(userDto.getFirstName());
            existingUser.setLastName(userDto.getLastName());
            existingUser.setEmail(userDto.getEmail());
            existingUser.setBirthNumber(userDto.getBirthNumber());
            if (userDto.getPassword() != null && !userDto.getPassword().isEmpty()
             && !userDto.getPassword().isBlank()) {
                existingUser.setPassword(passwordEncoder()
                    .encode(userDto.getPassword()));
            }
        return userRepository.save(existingUser);
    });
}
```

Služba *OrderService* zaobstaráva pridanie a teda uloženie novej objednávky do databázy a metódu na vrátenie všetkých užívateľových objednávok. Rovnako má vložené závislosti pomocou anotácie *@Autowired* na *OrderRepository* a *EmailService*. Metóda *public void addOrder(OrderDto orderDto)* uloží objednávku do databázy a zároveň, zavolá metódu z *EmailService*, aby odoslala email výdajni, s informáciami o novovzniknutej objednávke.

```
public void addOrder(OrderDto orderDto) {
    Order order = mapOrderDtoToOrder(orderDto);
    if (order != null) {
        orderRepository.save(order).subscribe();
        emailService.sendEmail(order);
        logger.info("Order was saved");
    }
}
```

Metóda *public Mono<List<OrderDto>> getUsersOrders(Long userId)* vráti list všetkých objednávok užívateľa podľa zadaného *userId*, identifikačného čísla užívateľa, zoradených podľa *deliveryDate*.

EmailService využíva SMTP server na posielanie emailov. Tento server je nutné na-

konfigurovať. Konfigurácia je v *application.properties* súbore. Je nutné nastaviť meno *host*, port, užívateľské meno a heslo. Rovnako je nastavená aj autentifikácia SMTP. V službe sú vložené závislosti na *JavaMailSender* a *UserRepository*. Hlavnou metódou tejto služby je *public void sendEmail(Order order)*. Táto metóda vytvorí novú emailovú správu.

SimpleMailMessage message = new SimpleMailMessage();

Následne vyhľadá užívateľa pomocou metódy *private Mono<UserDto> getUser(Long id)*.

Optional<UserDto> userOptional = getUser(order.getUserId()).blockOptional();

Ak užívateľ nie je *null*, potom nastaví do emailovej správy odosielateľa a príjemcu. Ako predmet správy je, od koho je objednávka - metóda *private String getSubject(UserDto user)* a v tele správy následne všetky údaje o objednávke - metóda *private String getMessage(UserDto user, Order order)*.

Poslednou službou mikroslužby Zdroje je *DocumentService*. Poskytuje všetky metódy, ktoré pracujú s dokumentami. Vrátene všetkých doposiaľ nahraných dokumentov, zoradených podľa *creationTimestamp* metódou *public Mono<List<DocumentDto>> getAllDocuments()*. Nahranie *public void uploadDocument(DocumentDto documentDto)* a vymazanie súboru *public void deleteDocument(String uid)* z databázy. Privátna metóda *private DocumentDto mapDocument(Document document)* zakóduje bytovú reprezentáciu súboru do reťazca pomocou *Base64*.

Base64.getEncoder().encodeToString(document.getFile());

4.4.3 Spracovanie REST požiadaviek

Spracovanie REST požiadaviek v Zdroje zabezpečuje *ResourceController* s anotáciou *@RestController*. Má vložené tri závislosti *OrderService*, *DocumentService* a *UserService*.

Rest požiadavky na prácu s objednávkami sú pridanie novej objednávky a vrátenie všetkých objednávok užívateľa. Využívajú metódy z *OrderService*.

```
@PostMapping("/addOrder")
public void addOrder(@RequestBody OrderDto orderDto) {
    orderService.addOrder(orderDto); }
```

Uloží novú objednávku zaslanú v *@RequestBody* ako *orderDto*.

```
@RequestMapping("/getUsersOrders")
@ResponseBody
public Mono<List<OrderDto>> getAllUsersOrders(@RequestParam("userId")
    Long userId) {
    return orderService.getUsersOrders(userId);
}
```

Vráti list všetkých objednávok užívateľa podľa zadaného parametra *userId* v *@RequestParam*.

Nahranie, vymazanie a vrátenie všetkých dokumentov zaobstarávajú tri metódy. Vymazanie má na starosti POST metóda *public void deleteDocument(@RequestBody String uid)*, predá *uid* dokumentu metóde zo služby *DocumentService*. Vrátenie všetkých dokumentov zabezpečuje GET *public Mono<List<DocumentDto>> getAllDocuments()*. Nahranie nového súboru je realizované pomocou metódy POST *public DocumentDto uploadDocument(@RequestBody MultipartFile file) throws IOException*. Tá z *MultipartFile* najprv nájde meno súboru.

```
String fileName = StringUtils.cleanPath(Objects.requireNonNull(file
    .getOriginalFilename()));
```

V *try catch* bloku skúsi nahrať súbor tým, že zavolá metódu z *DocumentService*, ak sa v procese stala chyba, vyhodí sa výnimka *IOException*.

Poslednými metódami sú tie, ktoré pracujú s užívateľom. Metóda, ktorá vráti všetkých užívateľov je GET *public Mono<List<UserDto>> getAllUsers()* a POST *public Mono<Boolean> updateUser(@RequestBody UserDto userDto)* aktualizuje dáta užívateľa poslaného v *@RequestBody*.

4.5 Model

Model je mikroslužba, ktorá obsahuje deklaráciu tried reprezentujúcich dáta v databáze a pomocné triedy s tým spojené. Pre každú triedu databázových dát deklaruje rozhranie repozitára.

Trieda *User* reprezentuje dáta v tabuľke *users*. Má parametrický konštruktor, *toString* metódu a *getter, setter* pre každý atribút. Anotácie pred deklaráciou triedy *@Data* a *@Table(name = "users")* hovoria o tom, že daná trieda prezentuje dáta v databáze. Trieda *UserDto* obsahuje všetky atribúty triedy *User* ako aj konštruktory a *getter,*

setter pre každý atribút. Repozitár *UserRepository* je označený anotáciou *@Repository* a dedí z *ReactiveCrudRepository*. Nie je nutné implementovať dané metódy, *query*, ktoré vyberajú na základe pravidiel pomocou SQL jazyka dáta z databázy, vytvára ich Spring Boot, konkrétne knižnica *spring-boot-starter-data-r2dbc*, ktorej závislosť je vložená v *pom.xml* mikroslužby Model.

```
@Repository
public interface UserRepository extends ReactiveCrudRepository<User, Integer> {
    Mono<User> findUserByEmail(String email);
    Mono<User> findUserById(Long id);
    Flux<User> findAllByRole(String role);
}
```

Podobne ako trieda *User* aj trieda *Order* reprezentuje dáta v tabuľke s rovnakými anotáciami. Taktiež existuje pomocná trieda *OrderDto* a repozitár *OrderRepository* s metódou *Flux<Order> findAllByUserId(Long userId)*.

Poslednou triedou, založenou na rovnakom princípe je *Document*. Pomocná trieda *DocumentDto* obsahuje navyše atribút *encodedFile*, ktorý je napĺňaný konštruktorom alebo svojím *setter* kódovaným súborom pomocou *Base64*. *DocumentRepository* taktiež ako predchádzajúce dva, dedí z *ReactiveCrudRepository* a obsahuje metódu *Mono<Void> deleteByUid(String uid)*.

ZÁVĚR

Počas tvorby tejto bakalárskej práce vznikla webová aplikácia pre výdajňu zdravotníckych pomôcok pre diabetikov. Okrem registrácie, prihlásenia a zadávania objednávok, ktoré sú následne formou emailu posielané priamo výdajni, obsahuje aj priestor na čítanie článkov zverejnených výdajňou Diatop. Táto funkcionalita odlišuje webovú aplikáciu a ponúka doplňujúcu funkcionalitu. Registrovaní užívatelia môžu objednávať materiál a zároveň byť neustále v obraze, čo je nové vo svete diabetu a zdravotníckych pomôcok s ním spojených.

Verzia tejto aplikácie nie je finálna. Výdajňa naďalej plánuje rozširovať jej funkcionalitu, ktorá pomôže jej chodu ako aj samotným zákazníkom. Príkladom je plánované rozšírenie pomocou REST API, ktoré bude informovať výdajňu o momentálnom množstve vydaného materiálu konkrétnemu zákazníkovi. To vznikne v spolupráci so spoločnosťou spravujúcou informačný systém výdajne. Spôsob registrácie administrátora je rovnako ďalším krokom k vývoju.

Tvorba tejto bakalárskej práce ma obohatila o mnoho nových skúseností s vývojom pomocou Spring a Angular, ktoré využijem aj v pracovnom prostredí.

Následným krokom bude zavedenie a pilotné testovanie na vzorke zákazníkov výdajne Diatop. Práve vďaka tomu bude aplikácia upravená podľa podnetov zo strany zákazníkov a tým sa stane ešte viac prívetivejšou pre užívateľov.

SEZNAM POUŽITÉ LITERATURY

[1] Carnell, J.: *Spring Microservices in Action*. Manning Publications Co, první vydání, 2017, ISBN 978-1-61729-398-6.

[2] JavaTpoint : Spring Tutorial [online]. https://www.javatpoint.com/spring-tutorial, [cit. 2023-04-18].

[3] Spring® : Why Spring? [online]. https://spring.io/why-spring, [cit. 2023-04-18].

[4] Reddy, K. S. P.: *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Apress, 2017, ISBN 978-1-4842-2930-9.

[5] Red Hat®: What does an API gateway do? [online]. https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do, 2019, [cit. 2023-04-04].

[6] Spring® : Spring Cloud Gateway [online]. https://cloud.spring.io/spring-cloud-gateway/reference/html/, [cit. 2023-04-04].

[7] Spring® : Web on Reactive Stack [online]. https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html, [cit. 2023-04-08].

[8] Red Hat® : 5 Things to Know About Reactive Programming [online]. https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming, [cit. 2023-04-08].

[9] Spring® : Spring Security [online]. https://docs.spring.io/spring-security/reference/index.html, [cit. 2023-04-08].

[10] K. Siva Prasad Reddy, I. C., Marten Deinum: *Pro Spring MVC with WebFlux web development in Spring Framework 5 and Spring Boot 2*. Apress, 2021, ISBN 978-1-4842-5665-7.

[11] All About Cookies : What are cookies in computers? [online]. https://allaboutcookies.org/, [cit. 2023-04-08].

[12] Spilcă, L.: *Spring Security in Action*. Manning Publications, 2020, ISBN 978-1-6172-9773-1.

[13] Java Development Jounral : Interface HttpSession [online]. https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html, [cit. 2023-04-10].

[14] Java Development Jounral : Interface WebSession [online]. https://www.javadoc.io/doc/org.springframework/spring-web/5.0.5.RELEASE/org/springframework/web/server/WebSession.html, [cit. 2023-04-10].

[15] Spring® : Spring Session [online]. https://docs.spring.io/spring-session/reference/index.html, [cit. 2023-04-10].

[16] Turnquist, G. L.: *Learning Spring Boot 2.0*. Packt Publishing Ltd., druhé vydání, 2017, ISBN 978-1-78646-378-4.

[17] Java Development Jounral : Guide to Spring Session [online]. https://www.javadevjournal.com/spring/spring-session/, [cit. 2023-04-10].

[18] Sagar Bhatia : PostgreSQL vs MySQL: Everything You Need to Know in 2023 [online]. https://hackr.io/blog/postgresql-vs-mysql, [cit. 2023-04-11].

[19] David Manda : PostgreSQL vs. MySQL: Which Is Best? [online]. https://www.databasejournal.com/mysql/postgresql-vs-mysql-which-is-best/, [cit. 2023-04-11].

[20] Spring® : Spring Data R2DBC - Reference Documentation [online]. https://docs.spring.io/spring-data/r2dbc/docs/current/reference/html/, [cit. 2023-04-14].

[21] R2DBC Contributors : R2DBC [online]. https://r2dbc.io/, [cit. 2023-04-14].

[22] Spring® : Sending Email [online]. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/io.email, [cit. 2023-04-15].

[23] Alexander S. Gillis : SMTP (Simple Mail Transfer Protocol) [online]. https://www.techtarget.com/whatis/definition/SMTP-Simple-Mail-Transfer-Protocol, [cit. 2023-04-15].

[24] Chris Richardson : What are microservices? [online]. https://microservices.io/, [cit. 2023-04-15].

[25] Nihar Raval : React vs Angular: Which JS Framework to Pick for Front-end Development? [online]. https://radixweb.com/blog/react-vs-angular, [cit. 2023-04-16].

[26] Inexture Solutions LLP : How to Choose Between React and Angular for your Upcoming Projects [online]. https://www.inexture.com/angular-vs-react/, [cit. 2023-04-16].

[27] Rodion Salnik : Angular vs React. Which JS Framework Is Better in 2022? [online]. https://brocoders.com/blog/angular-vs-react/, [cit. 2023-04-16].

[28] TypeScript : DOM Manipulation [online]. https://www.typescriptlang.org/docs/handbook/d manipulation.html, [cit. 2023-04-16].

[29] Google : Security [online]. https://angular.io/guide/securitysecurity, [cit. 2023-04-16].

[30] Liran Tal : Comparing React and Angular secure coding practices [online]. https://snyk.io/blog/comparing-react-and-angular-secure-coding-practices/, [cit. 2023-04-16].

[31] Google : CLI Overview and Command Reference [online]. https://angular.io/cli, [cit. 2023-04-16].

[32] YourTechDiet : What is Angular CLI? How is it different from AngularJS? [online]. https://yourtechdiet.com/blogs/what-is-angular-cli-how-is-it-different-from-angularjs/, [cit. 2023-04-16].

[33] Google : Angular Material [online]. https://material.angular.io/, [cit. 2023-04-16].

[34] Ellington a.s. : www.mojlekar.eu [online]. https://www.mojlekar.eu/, [cit. 2023-05-02].

[35] PROFILAND s.r.o. : www.navstevalekara.sk [online]. https://www.navstevalekara.sk/, [cit. 2023-05-02].

[36] © 2023 Medevio : www.medevio.cz [online]. https://www.medevio.cz/, [cit. 2023-05-03].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
AOT	Ahead Of Time
CSRF	Cross-site Request Forgery
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	identifikátor
IMAP	Internet Message Access Protocol
IT	Informačné Technológie
JDBC	Java Database Connectivity
JSX	JavaScript XML
MVC	Model-view-controller
MVCC	Multiversion Concurrency Control
OS	Operačný Systém
POP3	Post Office Protocol version 3
R2DBC	Reactive Relational Database Connectivity
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet
UNIX	Universal Interactive Executive
XSS	Cross-Site Scripting
XSSI	Cross-Site Script Inclusion

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I. SD KARTA

Priložená SD karta obsahuje súbory a zložky:

- zložka **diatop-bakalarka** - obsahuje implementáciu bakalárskej práce,
- súbor **bakalarska_praca** - obsahuje bakalársku prácu vo formáte PDF/A,
- súbor **Readme.pdf** - návod na spustenie webovej aplikácie.