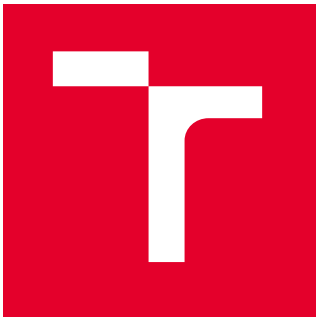


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

MOBILNÍ APLIKACE PRO PRVKY CHYTRÉ BUDOVY

MOBILE APPLICATION FOR SMART BUILDING COMPONENTS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — **Štefan Olenočin**

AUTHOR

VEDOUCÍ PRÁCE — **Ing. Lukáš Jablončík**

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Štefan Olenočin ***ID:*** 231262

Ročník: 3 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Mobilní aplikace pro prvky chytré budovy

POKYNY PRO VYPRACOVÁNÍ:

Cílem bakalářské práce je navrhnout mobilní aplikaci pro ovládání a správu prvků chytré budovy, jako jsou např. dobíjecí stanice, elektroměry, chytré zásuvky apod. V rámci teoretické části se student seznámí s návrhem mobilních aplikací v prostředí Flutter. Zaměří se na biometrické funkce při autentizaci uživatele a princip REST API pro komunikaci s prvky chytré budovy (např. EVSE, elektroměr, chytrá zásuvka). Dále student provede analýzu dostupných aplikací a navrhne princip práv pro přístup k zařízením. V rámci praktické části navrhne aplikaci, které by měla umožňovat vytvořit uživatele a nastavit přihlašování pomocí biometrických metod. Dále bude mít uživatel možnost přidat zařízení, nastavit práva k přístupu, ovládat zařízení a vizualizovat data. Důraz bude kladen na bezpečnost jak z pohledu autentizace uživatele, tak z pohledu zneužití prvků dalšími osobami.

DOPORUČENÁ LITERATURA:

[1] Zaccagnino, C. (2020). Programming flutter: Native, cross-platform apps the easy way. ISBN: 9781680506952.

[2] Martin, R. C. (2009). Clean code: A Handbook of Agile Software Craftsmanship. Prentice Hall. ISBN: 9780132350884.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 26.5.2023

Vedoucí práce: Ing. Lukáš Jablončík

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Bakalárska práca sa venuje návrhu mobilnej aplikácie na interakciu s prvkami chytrých budov, so zameraním na biometrickú autentizáciu. V teoretickej časti sa práca venuje všeobecnému návrhu softvéru, sú priblížené špecifické platformy a návrhu softvéru pomocou frameworku Flutter. Ďalej sa práca zameriava na technológiu REST API, na biometrické systémy, ich štruktúru, výhody, nevýhody a bližšie rozoberá niektoré biometrické metódy používané v mobilných aplikáciach. Na konci teoretickej časti sa práca venuje analýze dostupných riešení a ich porovnaniu. Na začiatku praktickej časti sú vypracované návrhy princípov práv prístupu s využitím služby Thingsboard, ktorá je tiež vysvetlená. Ďalej sú vysvetlené kľúčové procesy v aplikácií. Na koniec sa v práci nachádza popis samotnej aplikácie a jej funkcionality.

KĽÚČOVÉ SLOVÁ

android, biometrická autentizácia, chytrá budova, ios, iot, mobilná aplikácia, inteligentné zariadenie

ABSTRACT

The bachelor thesis is devoted to the design of a mobile application for interaction with elements of smart buildings, focusing on biometric authentication. In the theoretical part, the thesis is devoted to general software design, specific platforms and software design using the Flutter framework are approached. Furthermore, the thesis focuses on REST API technology, biometric systems, their structure, advantages, disadvantages and discusses in more detail some biometric methods used in mobile applications. At the end of the theoretical part, the thesis is devoted to the analysis of available solutions and their comparison. At the beginning of the practical part, proposals are made for the principles of access rights using the Thingsboard service, which is also explained. The key processes in the application are explained below. At the end of the work there is a description of the application itself and its functionality.

KEYWORDS

android, biometric authentication, flutter, ios, iot, mobile application, smart building, smart device

Vysadené pomocou balíčku `thesis` verzie 4.07; `http://latex.feec.vutbr.cz`

OLENOČIN, Štefan. *Mobilná aplikácia pre prvky inteligentných budov*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačných technologií, Ústav telekomunikací, 2023, 62 s. Bakalárska práca. Vedúci práce: Ing. Lukáš Jablončík

Vyhlásenie autora o pôvodnosti diela

Meno a priezvisko autora:	Štefan Olenočin
VUT ID autora:	231262
Typ práce:	Bakalárska práca
Akademický rok:	2022/23
Téma závěrečnej práce:	Mobilná aplikácia pre prvky inteligentných budov

Vyhlasujem, že svoju záverečnú prácu som vypracoval samostatne pod vedením vedúcej/cého záverečnej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podpisuje iba v tlačenej verzii.

POĎAKOVANIE

Rád by som poďakoval vedúcemu bakalárskej práce pánovi Ing. Lukášovi Jablončíkovi za odborné vedenie, trpezlivosť a vecné konzultácie k práci. Ďalej by som rád poďakoval rodine za trpezlivosť a podporu.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov

Úvod

Táto práca sa venuje oblasti vývoja mobilných aplikácií pre prvky chytrých budov a kladie dôraz na využitie biometrického systému pre vyššiu bezpečnosť pri autentizácii používateľa.

Hlavné ciele bakalárskej práce sú zoznámenie sa s návrhom mobilných aplikácií pomocou frameworku Flutter, využitie biometrického systému pri autentizácii a komunikácia s prvkami chytrej budovy prostredníctvom REST API. Ďalej bude vykonaná analýza dostupných riešení, navrhnuté princípy práv na prístup k zariadeniam. Bude vytvorená aplikácia, ktorá umožňuje ovládať zariadenia, vizualizuje dáta zo zariadení a bude umožňovať biometrickú autentizáciu.

Ciele práce boli úspešne dosiahnuté a bola vytvorená aplikácia poskytujúca vyššie popísanú funkcionalitu.

V teoretickej časti sa čitateľ dočíta o spôsobe vývoja aplikácií celkovo, ako aj pre jednotlivé platformy, bude mu priblížený Flutter UI framework, fungovanie a delenie biometrických systémov a príncip REST API. V praktickej časti nájde analýzu dostupných riešení, jednotlivé návrhy fungovania aplikácie a samotný popis aplikácie.

1 Vývoj mobilných aplikácií

V tejto časti je objasnená metodika vývoja nielen mobilných aplikácií, ale softvéru ako celku, prostriedky pre vývoj a niektoré ich plusy a mínusy. Metodika sa väčšinou skladá z týchto jednoduchých bodov:

- komunikácia a následná dohoda o požadovanej funkcionalite,
- návrhu štruktúry aplikácie,
- programové vyhotovenie samotnej aplikácie,
- testovanie aplikácie,
- vyhotovenie dokumentácie a poskytovanie podpory,
- údržba aplikácie, poprípade sa môže cyklus opakovať znova. [1]

V prvej fáze vývoja sa určia požiadavky na daný softvér a naplánuje sa návrh, vývoj a ciele aplikácie. Preto je tento krok jeden z najdôležitejších a musí sa dbať na jeho správne a detailné vykonanie. Komunikácia so zákazníkom by mala byť realistická a zákazník by mal byť informovaný o základných aspektoch vývoja ako sú napr. technické možnosti aplikácie alebo doba vývoja. [1]

Návrh aplikácie by mal viac priblížiť rôznym vývojárom ako má konkrétna aplikácia fungovať a čo má vykonávať. Tento krok je tiež veľmi dôležitý, pretože ak sa vo vývojárskom tíme náchadza viacero vývojárov, je pravdepodobné, že nebudú komunikovať rovnakým jazykom, poprípade každý využíva pre vývoj iný programovací jazyk alebo prostredie. Tento jav sa najviac vyskytuje vo väčších firmách, kde je tím medzinárodný a každý člen tímu má na starosti inú časť projektu napr. vývoj aplikácie pre operačný systém Android a vývoj aplikácie pre operačný systém iOS. [1]

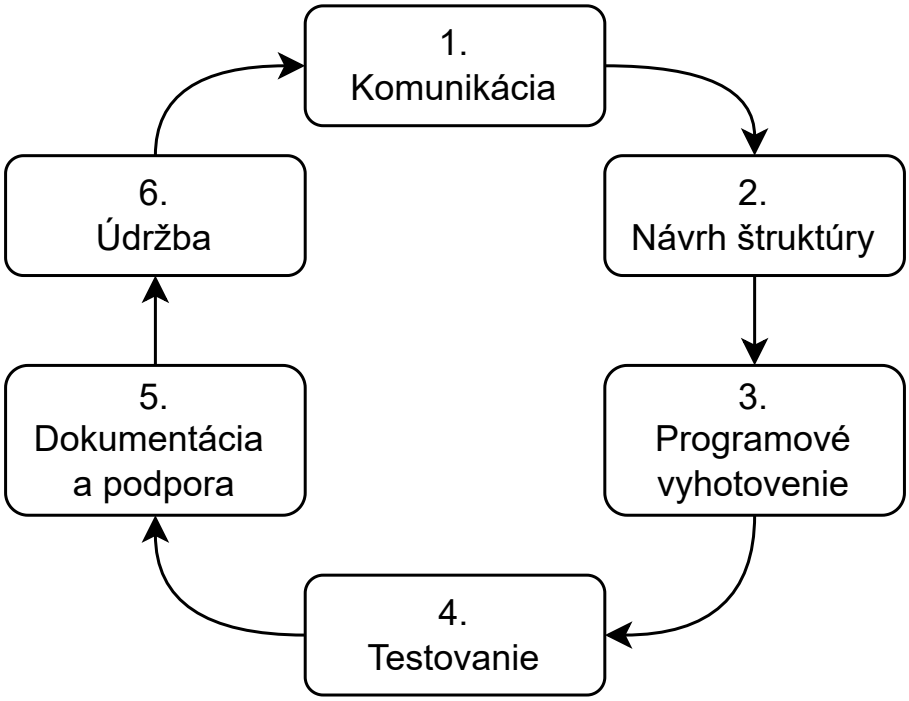
Programové vyhotovenie aplikácie je z pohľadu vývojára najdôležitejší krok. Vývoj by mal prebiehať organizovane, kód by mal byť čitateľný a udržiavateľný. Jednotlivé prvky funkcionality aplikácie by mali byť rozdelené do jednoduchých častí. Aby bol kód škálovateľný a jednoducho udržiavateľný, využívajú sa myšlienky a princípy tzv. čistej architektúry. [1, 2]

Testovanie aplikácie dokáže ešte pred vydaním softvéru na trh odhaliť neočakávané problémy a tým pádom uľahčiť ladenie aplikácie. Testovanie by malo prebiehať automatizovanými testami a internými testermi aplikácie. Výhoda tohto prístupu je, že pri rýchlom vývoji sa niektoré programové chyby dajú jednoducho odhaliť pár testami pričom interný testeri testujú aplikáciu z hľadiska jej celkovej funkčnosti, ovládania a výzoru. [1]

Ďalší krok je vyhotovenie dokumentácie pre softvér. Tento krok je dôležitý ako pre originálnych vývojárov tak aj pre zákazníkov poprípade širokú verejnosť. Ak sa tento krok vynechá, zvyčajne dôjde k pomalému rozpadnutiu softvéru. Vývojársky tím jednoducho nebude mať prostriedky nato, aby sa pokračovalo vo vývoji alebo

údržbe. [1]

Ak aplikácia splňuje požiadavky zákazníka, nasleduje vypustenie aplikácie širokej verejnosti či už v podobe súkromnej aplikácie k nejakému produktu, alebo verejnej aplikácie umiestnenej na niektorom z dostupných riešení napr. Google Play Store. Ďalej nasleduje údržba aplikácie, poprípade jej postupné vylepšovanie ak si to zákazník dohodol. [1]



Obr. 1.1: Životný cyklus vývoja softvéru

Prostredie pre vývoj je celkom rozdielne v závislosti na platforme, pre ktorú je aplikácia vyvíjaná. Pre vývoj na platforme Android, sa využíva programovací jazyk Java. Pre platformu iOS zasa programovací jazyk Swift. Existujú však aj frameworky, ktoré sú zamerané na tzv. multiplatformový vývoj, čiže spravidla z jedného zdrojového kódu dokážu vytvoriť kód pre rôzne platformy. Príkladom takéhoto frameworku je aj Flutter UI framework od Googlu, ktorý je spolu s vývojom v Jave a Swifte viac priblížený v následujúcich sekciách. [3, 4]

1.1 Operačný systém Android

Platforma Android alebo operačný systém Android je operačný systém založený na Linuxe. Je vyvíjaný spoločnosťou Google, ale je *open source* softvér, takže na jeho vývoji sa podieľa aj komunita a samotný kód operačného systému si môže pozrieť každý. [3]

Mobilné aplikácie pre platformu Android sú vyvíjané pomocou programovacieho jazyka Java a bežia vo virtualizovanom prostredí *Android Runtime* prípadne v *Dalvik*. Je tomu tak pretože, týmto spôsobom je možné zabezpečiť stabilnejšie vykonávanie aplikačného kódu, jeho bezpečnosť, keďže sa jedná o virtualizované prostredie a kompatibilitu s viacerými zariadeniami. Kompatibilitou sa myslí vydávanie nových verzií a ich inštalovanie. Keďže sa zdrojový kód v jave kompiluje pre Android Runtime a nie pre jedno konkrétne zariadenie je jednoduchšie a lacnejšie kompilovať aplikáciu len raz. [5]

Tento spôsob, ktorý využíva virtualizované prostredie a aplikačný kód nebeží natívne je veľmi rozšírený, ale samozrejme sa aplikácie môžu vyvíjať aj natívne a to s pomocou programovacieho jazyka C alebo C++. Tento spôsob sa však nepoužíva pre bežné aplikácie a je skôr určený pre vývojárov hardvérových ovládačov a pod. [3]

1.1.1 Programovací jazyk Java

Programovací jazyk Java je „high level“ objektovo orientovaný jazyk. To znamená, že je vytvorená vrstva abstrakcie nad systémom a vývojár nemusí napr. manažovať pamäť. Toto je veľkým plusom pre jazyk, pretože sa odstráni potreba debugovať kód kvôli pamäťovým chybám, ktoré sa obvykle hľadajú veľmi ťažko. Ďalšou výhodou javy, ktorá bola spomenutá aj v kapitole 1.1 je, že beží vo virtualizovanom prostredí. Toto umožňuje jave byť tzv. multiplatformovým jazykom, keďže nezáleží na kompilácií pre jednotlivé architektúry. [3]

1.2 Operačný systém iOS

Operačný systém iOS je mobilný operačný systém vyvíjaný spoločnosťou Apple pre zariadenia Apple. Je podobne ako Android založený na Linuxe avšak jeho zdrojový kód je uzavretý. [3]

Aplikácie sú vyvíjané pomocou programovacích jazykov *Objective-C* alebo *Swift*. Swift je špeciálne vyvinutý pre vývoj aplikácií pre iOS, preto sa vo väčšine prípadoch využíva práve tento jazyk. [3]

1.2.1 Programovací jazyk Swift

Swift je kompilovaný programovací jazyk vytvorený spoločnosťou Apple pre vývoj aplikácií a komunikáciu s *Cocoa Touch* frameworkom, ktorý sa stará o užívateľské rozhranie. [3]

Bol dizajnovaný, aby bol bezpečnejší ako Objective-C, ale je postavený na jeho prostredí. Týmto sa dosiahne možnosti využívať nielen Swift, ale aj C, C++ a samozrejme aj samotný Objective-C. [3]

1.3 Flutter framework

Flutter je multi platformový *UI* (User Interface) framework vyvíjaný spoločnosťou Google. Umožňuje vývoj aplikácií pre viacero operačných systémov vrátane iOS a Android, pričom na vývoj aplikácií vo frameworku Flutter sa využíva programovací jazyk Dart. [6, 4]

Aby bol Flutter multi platformový, využíva kompiláciu priamo do strojového kódu danej platformy. Avšak niekedy je potrebné využiť aj natívne prostriedky platformy. Flutter toto umožňuje tak, že Flutter aplikácie vníma operačný systém platformy ako natívne aplikácie. Na každej platforme má Flutter svoj špecifický vstupný bod, ktorý sa môže využiť buď ako prídavný modul k nejakej už existujúcej natívnej aplikácií alebo ako samostatná Flutter aplikácia. Tieto vstupné body sú napísané v jazyku špecifickom pre danú platformu a vývojár Flutter aplikácií sa týmto už nemusí zaoberať. [6]

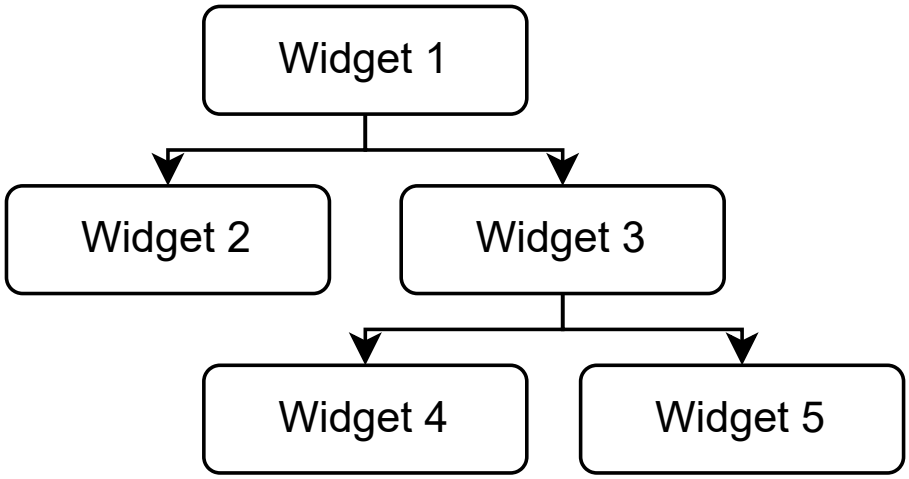
Jadrom Flutteru je *Flutter engine*. Ten zabezpečuje vykresľovanie grafických prvkov, prístup k súborovému systému, sieťovej funkcionalite a pod. Typicky, vývojári Flutter aplikácií nevyužívajú priamo Flutter engine, ale využívajú Flutter framework, ktorý je abstrakciou nad týmto jadrom. [6]

Flutter framework je celkom malý a veľa funkcionalít sa rieši prostredníctvom balíčkov. Tomuto prístupu pomáha aj fakt, že Flutter framework je open source softvér. Tým pádom existuje veľa komunitných balíčkov, ktoré dodávajú ostatnú potrebnú funkcionalitu napr. animácie, grafy alebo mapy. [6]

Štruktúra používateľského prostredia (UI) sa skladá z widgetov usporiadaných do stromovej štruktúry viď obr. 1.2. Tieto widgety môžu meniť svoj stav a Flutter sa už ďalej stará o zmeny, ktoré sa majú premietať do UI a ktoré widgety treba znova vykresliť. Tento prístup je veľmi efektívny a ušetrí sa výpočtový výkon na widgetoch, ktoré svoj stav nezmenili. [6]

Ako bolo spomenuté vyššie, widgety môžu meniť svoj stav, k tomuto Flutter využíva dva typy widgetov, *stavové* a *bezstavové*. Ako vyplýva z názvu bezstavové widgety nemenia svoj stav počas behu aplikácie, môžu to byť napr. ikony. Stavové widgety menia svoj stav v reakcii na nejaký podnet napr. tlačidlo on/off, ktoré má dva stavy, sa zmení po kliknutí. [6]

Keďže widgety majú stromovú štruktúru a každý môže mať svoj stav, tak je potrebný nejaký spôsob, ako k ich stavom pristupovať. Flutter disponuje spôsobom



Obr. 1.2: Stromová štruktúra widgetov

ako môžu widgety komunikovať a zdielať svoj stav medzi sebou. Tento spôsob sa realizuje pomocou *Inherited widget*. [6]

Inherited widget funguje na princípe, že zdieľa svoj stav všetkým svojím potomkom v stromovej štruktúre. Toto umožňuje vývojárom jednoducho pristupovať k stavu jednotlivých widgetov, ktoré zdieľajú spoločný rodičovský Inherited widget, napr. ako je na obr. 1.3. [6]

Tento manažment stavov sa však vie rýchlo stať problémom pri veľkom počte widgetov, a preto boli komunitou vyvinuté iné metódy manažmentu stavov, ako napr. „Redux store“, ktorý bude priblížený v neskôr. [6]

1.3.1 Programovací jazyk Dart

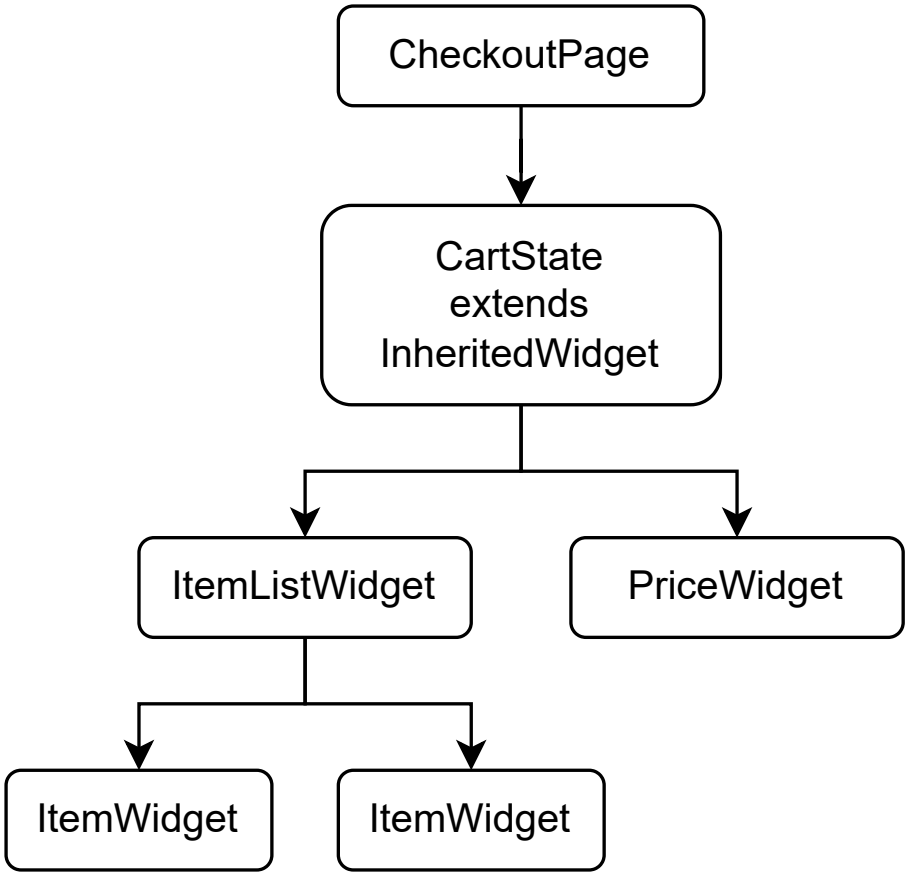
Dart je objektovo orientovaný programovací jazyk, vyvinutý spoločnosťou Google pre použitie s Flutter frameworkom. Je to voliteľne typovaný jazyk. Avšak Dart využíva *type soundness*, aby sa predišlo nedefinovaným stavom a tým pádom aj problémom s datovými typmi. Využíva sa na to statická kontrola typov pri kompilácií v spojení s dynamickou kontrolou počas behu programu. [7]

Tento jazyk je veľmi podobný Jave poprípade C# avšak pridáva nejaké nové funkcie, ktorými ostatné jazyky nedisponujú ako napr. string interpolation. [7]

1.4 Manažment stavov pomocou Redux store

Keďže požiadavky na mobilné aplikácie sú čoraz komplikovanejšie, ich kód musí zvládať viacero stavov. Môže sa jednať o stavy ako:

- aplikácia je v „kľudovom“ stave,



Obr. 1.3: Využitie Inherited widgetu

- aplikácia odoslala požiadavku a čaká na odpoveď od serveru,
- aplikácia dostala odpoveď a je potrebná zmena UI.

Tieto tri stavy, ktoré však nemusia byť jediné, je celkom problém manažovať. Obzvlášť, ak je prechod medzi stavmi asynchrónny. To znamená, že sa napr. odošle požiadavka na server a čaká sa na odpoveď, poprípade sa načítajú dáta z disku a pod. Taktiež nastáva problém ak by skončilo zároveň viacero asynchrónnych funkcií a vytvorila by sa tzv „race condition“, kedy by sa musel obnoviť model viacerými funkciami zároveň. Keďže model, ktorý bol aktualizovaný, chce spravidla zmeniť svoje zobrazenie, poprípade môže model aktualizovať ďalší model a ten bude chcieť aktualizovať svoje zobrazenie alebo znovu ďalší model atď. Rýchlo sa stane, že sa v stave aplikácie vývojár môže stratiť a nerozumie, kde, prečo a kedy sa niečo deje. Toto môže spôsobiť, že aplikácia bude neprehľadná, bude ťažké hladať a reprodukovať chyby alebo pridávať nové funkcie. [8]

Redux store alebo len Redux, je spôsob ako manažovať stav aplikácie. Dosahuje sa to pomocou tzv „jednosmerného toku dát“. Je to príncip, pri ktorom aplikácia disponuje minimálne troma prvkami, a to:

- akcie,

- zobrazenie
- a stav.

Proces zmeny stavu sa začína pri zobrazení. Tu používateľ môže napr. kliknúť na tlačidlo, zmeniť nejaké textové pole a pod. Tieto zmeny sa pomocou „event handlerov“ ďalej propagujú ako akcie. Akcie sú potom jednotlivé objekty, ktoré špecifikujú, aká zmena stavu sa má vykonať. Na konci sa stav aplikácie zmení na základe vyvolanej akcie a obnoví sa zobrazenie. Tento kolobeh sa potom deje znova.

Štruktúra Redux sa skladá zo:

- store,
- stav,
- reducer
- a akcie.

Store je v podstate model, ktorý uchováva hlavný stav aplikácie. Mal by byť dostupný pre všetky widgety v aplikácií, ktoré si k nemu vyžiadajú prístup. Správa sa ako „zdroj jedinej pravdy“, čo znamená, že všetky potrebné dáta z nejakého stavu sú uložené len tu a nikde inde.

Stav je už samotný stav aplikácie. Je to dart objekt, ktorý obsahuje nejaké parametre alebo dáta a je nezmeniteľný. Týmto sa zabezpečí, že zmena stavu môže nastať len vytvorením nového stavu a nahradenie starého novým. Tým pádom sa nemusí vývojár starať o race condition a pod. Ak sa stav aplikácie zmení, widgety, ktoré používajú dáta z tohto stavu, dostanu signál že sa stav zmenil a aby obnovili svoje zobrazenie.

Reducer je časť Redux štruktúry, ktorá má za úlohu na základe vyvolanej akcie vykonať nejaký programový kód alebo funkciu, čiže zmeniť stav aplikácie. Je to jednoduchá funkcia, ktorá sa volá synchrónne a vracia nový stav aplikácie.

Akcie sú dart objekty, ktoré špecifikujú, čo za udalosť nastala. Akcie sa predávajú funkciám reducer a tie na základe typu objektu rozhodujú čo sa ma stať so stavom aplikácie. [8] [9] [10]

Asynchrónne akcie sa vykonávajú pomocou tzv „middleware“ funkcií. Sú to špeciálne funkcie, ktoré sa vykonávajú ešte predtým, ako sa vytvorené akcie dostanú do reducer funkcie. V týchto funkciách sa môže vykonávať napr. odoslanie requestu serveru a čakanie na jeho odpoveď. Ak by odpoveď prišla resp. by nastal timeout, tak asynchrónna funkcia vyvolá akciu korenšpondujúcu s jej stavom. Až keď sa toto všetko stane, tak sa odošle táto nová akcia do reduceru a zmení stav aplikácie. [11]

1.5 RESTful API

API alebo *application programming interface* je súbor pravidiel, ktoré definujú spôsoby ako môžu rôzne aplikácie, služby alebo zariadenia komunikovať medzi sebou.

Tab. 1.1: Priradenie HTTP metód k databázovým operáciam

HTTP	DB systém	Funkcia
GET	Read	Získanie nejakého zdroja zo servera
POST	Create	Vytvorenie nového zdroja poprípade úprava zdroja
PUT	Update	Úprava existujúceho zdroja na serveri
DELETE	Delete	Zmazanie existujúceho zdroja zo servera

Tento spôsob umožňuje prístup k rôznym funkciám aplikácie alebo jej dátam bez znalosti implementácie danej aplikácie. Aplikácia, ktorá pristupuje k API sa volá klient a aplikácia, ktorá poskytuje API sa volá server. [12, 13]

REST API alebo RESTful API je druh API, ktorý spĺňa dizajnové princípy *representional transfer state* architektúry. Medzi tieto princípy patrí *jednotné rozhranie* pre viacero requestov, ktoré pristupujú k tomu istému zdroju. Ďalší princip je oddelenie klienta a serveru tak, že sú na sebe nezávislé a jedinú znalosť, ktorou má klient disponovať je adresa daného zdroju ku ktorému pristupuje. *Bezstavovosť* je ďalším princípom a ten stanovuje, že requesty obsahujú všetky potrebné informácie na ich spracovanie a nie sú závislé na sebe. Ak to je možné zdroje by mali byť *uložiteľné do pamäte*, aby sa zvýšila výkonnosť klientskej aplikácie poprípade škálovateľnosť serveru. A posledným princípom je *vrstvenie*, ktoré stanovuje, že aplikácie sú dizajnované tak, aby sa klient nepripájal priamo k serveru, ale k nejakému medzibodu, ktorý sprostredkúva komunikáciu so serverom. [12, 13]

REST API funguje prostredníctvom HTTP metód *POST, GET, PUT* a *DELETE*. Tieto metódy poskytujú spôsob spracovania a možnosti manipulácie so zdrojom uloženým na serveri. Spôsob je celkom podobný databázovým systémom a ich operáciam *CREATE, READ, UPDATE* a *DELETE* viď tab. 1.1. [13, 12, 14, 15, 16]

Stav daného zdroja môže byť doručený klientovi vo viacerých datových formátoch. Najviac využívaný je *JavaScript Object Notation* (JSON) viď výpis 1.1 pretože je zároveň čitateľný ľuďmi a počítačmi, ale používajú sa aj iné napr. normálny text. Ďalej sa využívajú hlavičky a telá v HTTP požiadavkách a odpovediach na bližšiu špecifikáciu napr. sa môže v hlavičke požiadavky posielať autorizačný token, v tele požiadavky môže byť filter na základe ktorého server pripraví odpoveď. [12, 13, 17, 18]

Výpis 1.1: Ukážka JSON

```
1 {
2   "zákazníci": [
3     {
4       "meno": "Ondrej",
```

```
5         "priezvisko": "Malý",
6         "vek": 32,
7         "vozidla": [
8           "BMW", "Audi", "Mercedes"
9         ]
10      }
11    ]
12  }
```

2 Biometrické systémy

Biometrický systém je systém, ktorý je schopný detekovať jedinečné vzory človeka, taktiež nazývané ako biometrická charakteristika. Tento systém zvyčajne slúži na rozpoznanie jednotlivých osôb pomôcou rôznych metód detekcie vzorov, ktorými daná osoba disponuje napr. odtlačok prsta. Biometrické systémy sa väčšinou používajú v dvoch prípadoch a to pri autentifikácii a pri identifikácii. [19]

Autentifikácia funguje na princípe *one to one*, kde sa pomôcou biometrických metód získajú potrebné biometrické charakteristiky, a tieto sa neskôr porovnávajú so záznamom v databáze, ktorý tam bol uložený pri registrácií biometrickej charakteristiky. Zisťuje sa či je daná osoba skutočne osoba, ktorá má prístup k nejakým aktívam. [19]

Pri identifikácii sa rovnako, ako pri autentifikácii získajú biometrické charakteristiky, tie sa ale neporovnávajú s jedným konkrétnym záznamom, ale s viacerými. Tento spôsob teda funguje v relácií *one to many* a týmto spôsobom sa pokúša identifikovať danú osobu. [19]

Existuje veľa rôznych biometrických metód, ako získať od osoby jej jedinečné črty ako napr. otlačok prsta, črty tváre a pod. Niektoré z nich sú priblížené v nasledujúcej sekcii. [19, 20]

Keďže ide o systém v reálnom svete, uvažuje sa, že aj v tomto prípade môžu nastať chyby. Tie sa zvyčajne rozlišujú na dve, a to:

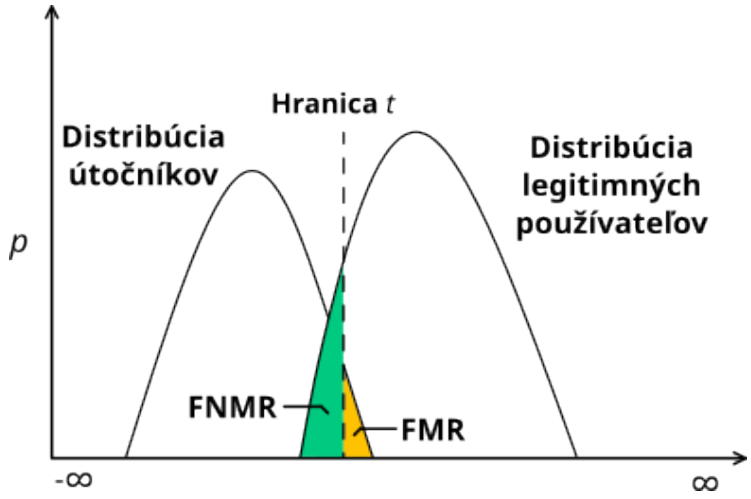
- false match rate (FMR),
- false non-match rate (FNMR).

FMR určuje, ako často systém vyhodnotí aktuálne biometrické charakteristiky zo vstupu systému ako správne, pričom správne nie sú. FNMR zase určuje chybu, ako často systém vyhodnotí, že sa nejedná o správnu biometrickú charakteristiku keď je v skutočnosti správna. [19, 20]

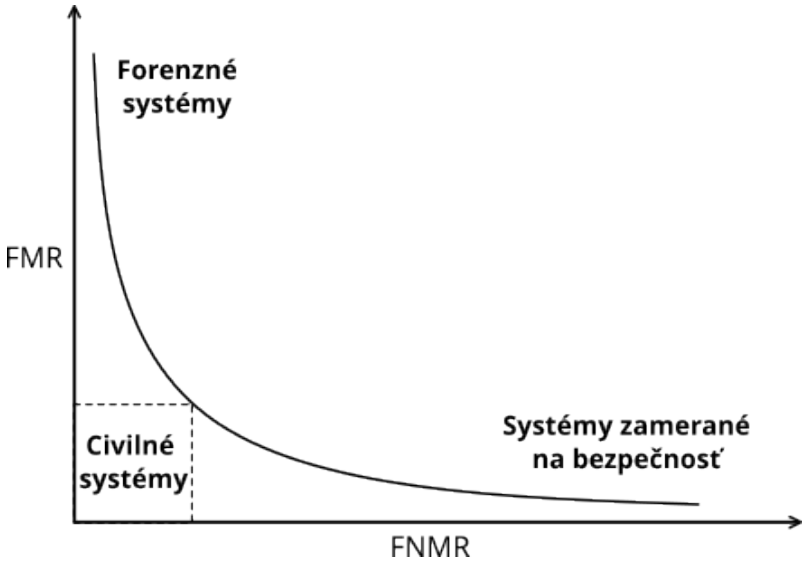
Pri dizajnovaní biometrických systémov musia vývojári počítať s týmito chybami a systém prispôsobiť tak, aby bol vyvážený a neexistovala výrazna prevaha niektorej z chýb viď obr. 2.1. Avšak existujú výnimky z tohto nepísaného pravidla. Aplikácie upredňostňujúce bezpečnosť vyžadujú vyššie FNMR, aby sa predišlo neoprávnenému prístupu k aktívam, pričom niektoré aplikácie môžu upredňostňovať FMR, aby sa zvýšilo pohodlie využívania biometrického systému človekom poprípade sa systém využil pre forenzné účely viď obr. 2.2. [19]

Biometrické systémy a ich štruktúra môže byť popísana pomocou týchto štyroch modulov:

- senzorový modul,
- modul na extrakciu biometrických charakteristík,
- modul overovania zhody,



Obr. 2.1: Distribúcia užívateľov a chybovosti. [19]



Obr. 2.2: Chybovosť v jednotlivých systémoch. [19]

- databázový modul.

Senzorový modul, ako z názvu vyplýva, sa stará o zosnímanie biometrických dát pomocou rôznych druhov senzorov. Tento modul potom posunie zosnímane dáta na spracovanie do modulu na extrakciu biometrických charakteristík z poskytnutých dát. Ďalej modul overovania zhody v spolupráci s databázovým modulom, ktorý obsahuje záznamy biometrických charakteristík vyhodnotí, či sa jedná o zhodu alebo nie. [19]

Ďalej sa môžu biometrické systémy rozlišovať na základe spôsobu, ktorým zbierajú biometrické charakteristiky a ako s nimi pracujú. Konkrétne, ich rozdelenie je na *unimodálne* a *multimodálne*. [19]

2.1 Unimodálne biometrické systémy

Unimodálne biometrické systémy sú systémy zbierajúce jednú vzorku biometrickej charakteristiky. Ide o najbežnejšie biometrické systémy. Ich hlavnou výhodou je pohodlnosť ich používania pre človeka. Avšak tieto systémy majú pár nevýhod:

- rušenie v zosnímanom vzorku,
- vnútorné variácie,
- rozlíšiteľnosť,
- univerzálnosť,
- útoky na systém. [19, 21]

Rušenie v zosnímanom vzorku je jeden z najviac vyskytujúcich sa problémov pri využívaní unimodálnych systémov. Ide primárne o „znečistenie“ vzorku nejakým externým vplyvom napr. špinavé prsty pri rozpoznávaní odtlačku prstu, alebo zlé svetelné podmienky pri rozpoznávaní tváre. [19, 21]

Vnútorné variácie sú spôsobené napr. výmenou senzorov, ktoré boli použité na zozbieranie prvotných biometrických charakteristík za iné. Toto vplýva na zozbierané vzorky, a tým pádom môžu negatívne ovplyvniť proces overovania vzorkou. Niekedy sa ale môže jednať o zle používanie systému ako napr. zlý uhol pri rozpoznávaní tváre a pod. [19, 21]

Rozlíšiteľnosť biometrických charakteristík je v závislosti na použitej biometrickej metóde rôzna a v reálnom systéme sa uvažuje, že každá metóda má svoju hornú hranicu rozlíšiteľnosti, podľa ktorej je schopná odlíšiť biometrické charakteristiky. [19, 21]

Čo sa týka univerzálnosti, tak každý človek alebo skupina nemusí mať danú biometrickú charakteristiku napr. ľudia, ktorí vykonávajú manuálnu prácu môžu mať poškodené odtlačky prstov. [19, 21]

Posledným problémom pri unimodálnych systémoch, ktorým sa práca zaoberá sú útoky na systémy. Pri tomto probléme sa hlavne musí pozerať na typ biometrickej charakteristiky. Ak je typ charakteristiky *behavoriálny*, čiže sa jedná o biometrickú charakteristiku, ktorá vyplýva z jedinečnej činnosti človeka napr. spôsob chôdze, podpis, je jednoduchšie realizovať úspešný útok. V prípade *fyzických* biometrických charakteristík je technicky náročnejšie zaútočiť na systém, ale aj pri týchto charakteristikách boli úspešne realizované útoky, avšak bola vyžadovaná spolupráca osoby s danou charakteristikou a odborné znalosti na realizovanie napr. vytvorenie falošných odtlačkov prsta. [19, 21]

2.2 Multimodálne biometrické systémy

Problémy pri unimodálnych biometrických systémoch sa môžu do nejakej hĺbky riešiť týmito systémami. Ide o biometrický systém, ktorý pracuje s viacerými vzorkami, biometrickými charakteristikami, alebo nejako inak kombinuje biometrické charakteristiky. Existuje viacero spôsobov, ako realizovať tento systém, medzi pár z nich patrí:

- využitie viacerých senzorov,
- využitie viacerých biometrických charakteristík,
- využitie rôznych vzorkov jednej biometrickej charakteristiky,
- využitie viacerých vzorkov jednej biometrickej charakteristiky,
- využitie rôznych algoritmov na realizovanie zhody. [19, 20, 21, 22]

Pri použití viacerých senzorov sa jedna biometrická charakteristika zosníma pomocou viacerých senzorov a výsledky sú neskôr spojené dokopy napr. jeden a ten istý odtlačok prsta sa zosníma na dvoch senzoroch. [19, 20, 21, 22]

Viaceré biometrické charakteristiky zabezpečujú dva alebo viacero stupňov ochrany, pretože sa kombinujú rôzne biometrické metódy, ako napr. rozpoznávanie tváre a odtlačok prsta. Tieto systémy zvyčajne využívajú rýchlu a menej spoľahlivú metódu na určenie N rôznych zhôd a potom sa využije pomalšia metóda na určenie konkrétnej zhody z N predchádzajúcich.[19, 20, 21, 22]

Rôzne vzorky jednej biometrickej charakteristiky vedia zabezpečiť lepšie a presnejšie určenie zhody. Môžu to byť napr. rôzne prsty pri snímaní odtlačkov. [19, 20, 21, 22]

Podobne, ako pri predchádzajúcom spôsobe sa zbiera viacero vzorkov, avšak pri tomto spôsobe sa zbierajú rovnaké vzorky jednej biometrickej charakteristiky napr. viacero odtlačkov jedného prstu. [19, 20, 21, 22]

Rôzne algoritmy na určovanie biometrickej charakteristiky a zisťovanie zhody z jedného a toho istého vzorku vedia vylepšiť určovanie zhody a ich kombináciou sa zvýši spoľahlivosť systému napr. rôzne algoritmy na určovanie biometrickej charakteristiky odtlačku prsta. [19, 20, 21, 22]

Využitie niektorého z týchto spôsobov môže značne ovplyvniť spoľahlivosť systému a jeho presnosť obzvlášť pri spôsobe kombinácie viacerých biometrických charakteristík. Spôsobmi, ako sa kombinujú a spájajú vyšie spomínané možnosti sa táto práca nebude zaoberať. [19, 20, 21, 22]

2.3 Biometrické metódy

Existuje mnoho biometrických metód, každá má svoje výhody a nevýhody. V tomto prípade sa ale najviac rieši, aby daná biometrická metóda spĺňala tieto podmienky:

- univerzálnosť,
- rozlíšiteľnosť,
- trvalosť,
- zbierateľnosť. [19, 20]

Univerzálnosť stanovuje, že každá osoba by mala mať danú biometrickú charakteristiku a nemala by byť zameraná len na jedincov poprípade skupinu. Rozlíšiteľnosť by mala zabezpečiť, aby dve rozdielne osoby nemali rovnakú biometrickú charakteristiku a teda nemohlo dôjsť k ich zámene. Trvalosť biometrickej metódy zaisťuje, že daná biometrická charakteristika sa časom nemení. V poslednom rade zbierateľnosť hovorí, že biometrickú charakteristiku je možné kvantitatívne zbierať a merať. [19, 20]

Z hľadiska reálneho biometrického systému by mala metóda taktiež mať tieto vlastnosti:

- výkonnosť,
- akceptovateľnosť,
- odolnosť voči možnému zneužitiu. [19, 20]

Z pohľadu reálneho biometrického systému záleží na výkonnosti a rýchlosti systému. Jeho akceptovateľnosť ukazuje, ako veľmi sú ľudia ochotní danú charakteristiku využívať a odolnosť určuje ochranu, poprípade odolnosť biometrického systému proti zneužitiu alebo nesprávnemu používaniu. [19, 20]

Porovnanie niektorých vybraných biometrických metód je znázornené tabuľkou 2.1. Keďže týchto metód je veľké množstvo a v mobilných aplikáciach sa nedajú všetky využiť, zameranie bude len na metódu rozpoznanie tváre a odtlačok prsta.

Tab. 2.1: Vybrané metódy a ich vlastnosti

Biometrická charakteristika	Univerzálnosť	Rozlíšiteľnosť	Trvalosť	Zbierateľnosť	Výkonnosť	Akceptovateľnosť	Odolnosť systému
DNA	vysoká	vysoká	vysoká	nízka	vysoká	nízka	nízka
Tvar ucha	stredná	stredná	vysoká	stredná	stredná	vysoká	stredná
Iris oka	vysoká	vysoká	vysoká	stredná	vysoká	nízka	nízka
Retina oka	vysoká	vysoká	stredná	nízka	vysoká	nízka	nízka
Tvár	vysoká	nízka	stredná	vysoká	nízka	vysoká	vysoká
Odtlačok prsta	stredná	vysoká	vysoká	stredná	vysoká	stredná	stredná
Spôsob chôdze	stredná	nízka	nízka	vysoká	nízka	vysoká	stredná
Hlas	stredná	nízka	nízka	stredná	nízka	vysoká	vysoká

2.3.1 Rozpoznávanie tváre

Táto metóda využíva rozpoznanie tváre. Biometrické charakteristiky, s ktorými táto metóda zvyčajne pracuje sú rôzne tvary a vzdialenosti, ako napr. vzdialenosť očí, tvar nosu, tvar obočia, pier alebo brady. Tieto rysy sú samozrejme len podmnožinou a existujú aj iné charakteristiky, ktoré sa môžu zbierať, poprípade sa kombinuje viacero charakteristík dokopy a získa sa tak obecný tvar tváre osoby. [19, 20]

Aby táto metóda fungovala, mali by sa v biometrickom systéme automaticky vykonávať tieto tri kroky:

- zisťovanie, či sa tvár nachádza v obraze,
- zisťovanie, kde v obraze sa tvár nachádza, ak bol prvý krok úspešný,
- rozpoznanie tváre. [19, 20]

Pretože je táto metóda v dnešnej dobe technicky nenáročná a náklady na jej integráciu sú nižšie, ako pri iných metódach využíva sa u chytrých telefónov, laptopov a pod. Spôsoby využitia sú rôzne. Od využitia na statickú detekciu (fotografie), detekciu v kontrolovanom prostredí (snímky väzňov), ale aj na dynamickú detekciu v rušnom prostredí s nejednotným pozadím (detekcie v reálnom čase na telefónoch, laptopoch). [19, 20]

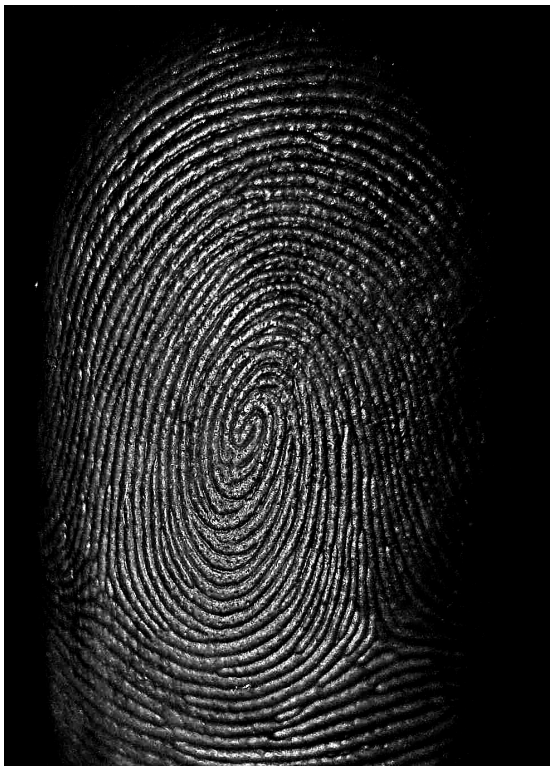
Problém, ktorý sa môže vyskytnúť pri používaní tejto metódy je uhol snímanej tváre, rôzne svetelné podmienky, ale aj jednoduché veci ako napr. kusy oblečenia, okuliare a pod. Toto je ťažké zabezpečiť, ak sa nepoužíva kontrolované prostredie, a pri dynamickej detekcii je to vidieť najviac. Avšak výhody tejto metódy prevyšujú jej nevýhody, a tak sa veľmi často využíva. [19, 20]

2.3.2 Odtlačok prsta

Táto metóda je presná a spoľahlivá, a využíva sa už niekoľko dekád. Odtlačok prsta sa skladá z hran a priehĺbin (papilárne línie), ktoré sa ustália už počas vývoja pred narodením človeka. Tieto hrany a priehĺbiny tvoria jedinečné vzory a z týchto vzorov sa extrahujú poznávacie body odtlačku, ktoré neskôr spracováva biometrický systém. Avšak niektoré skupiny ľudí nemusia disponovať spracovateľnými odtlačkami prstov. Jedná sa hlavne o manuálne pracujúcich, ktorý ich môžu mať poškodené. [19, 20, 23]

Medzi niektoré jedinečné vzory, ktoré sa hľadajú v odtlačku prsta patria konce hrán, trojuholníky, ostrovčeky, spojenie hrán a ich relatívna pozícia. Tieto jednotlivé vzory sú zobrazené na obr. 2.4 kde:

- A - sú konce hrán,
- B - sú spojenia hrán,
- C - sú trojuholníky,
- D - sú ostrovčeky. [19, 20, 23]



Obr. 2.3: Odtlačok prsta

Obr. 2.4: Jedinečné vzory

3 Analýza dostupných riešení

Pri vývoji aplikácie je vhodné sa pozerať na už existujúce riešenia a vyskúšať si ako fungujú, ako vyzerajú a ako dobre sa ovládajú. V nasledujúcich sekciách sú popísané jednotlivé aplikácie, nad ktorými bola spravená analýza. Kritéria analýzy sú následovné:

- jednoduchosť používania,
- funkčnosť,
- rozdielne formy prihlásenia.

3.1 Aplikácia iCOOL4

Táto aplikácia je vyvíjaná českou spoločnosťou ICT EXPERT s.r.o. a je dostupná pre android a iOS. Čo sa týka jednoduchosti používania, aplikácia je trochu nedopracovaná. Malé chyby, ktoré síce nemajú vplyv na funkčnosť, ale príjemné používanie aplikácie sú napr. absencia tlačidla na zobrazenie hesla pri prihlasovaní, pri nastavení českého jazyka aplikácie sú niektoré prvky v angličtine viď obr. 3.1 alebo niektoré prvky pridané používateľom nemajú ošetrený vzhľad a majú neprirodzenú veľkosť.

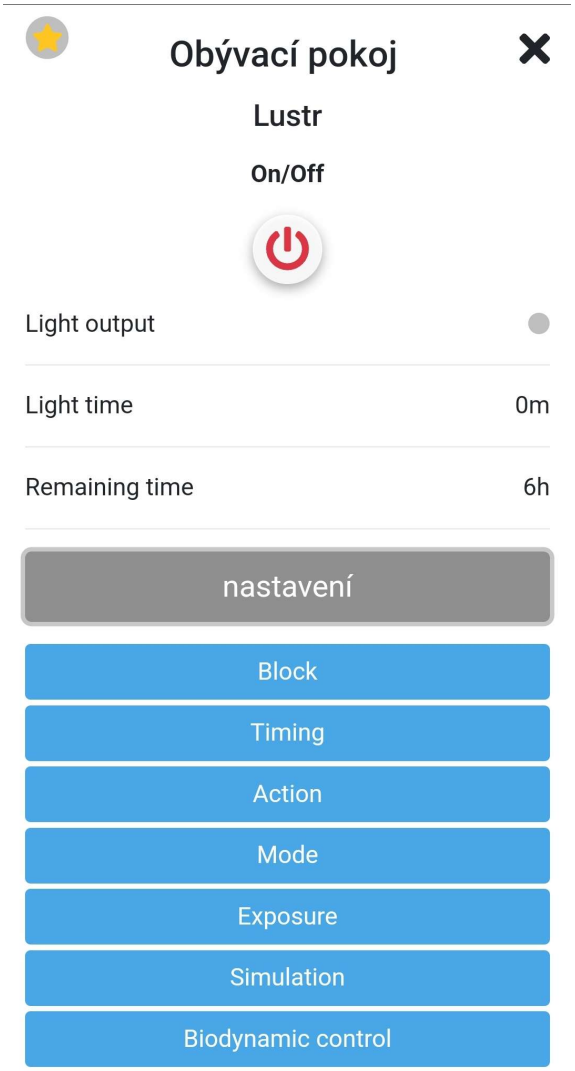
V rámci funkčnosti niektoré nastavenia alebo grafické prvky sa prekrývajú, alebo majú neprirodzenú polohu, ktorá im niekedy bráni vo viditeľnosti. Plusom je však vzhľad aplikácie a niektoré položky aplikácie sú spravené kvalitne a užívateľsky prívetivo. Aplikácia však obsahuje veľmi veľa funkcionality, a preto sú takéto chyby očakávané.

Formy prihlásenia sú dve. Prvé je klasické prihlásenie pomocou emailu a hesla a druhé je automatické prihlásenie. Čo sa týka bezpečnosti, bolo by vhodné pridať autentizáciu pri automatickom prihlasovaní, poprípade možnosť pre používateľa si túto funkcionalitu aktivovať.

3.2 Aplikácia VeSync

VeSync združuje viacero firiem a spája ich produkty prostredníctvom aplikácie VeSync do inteligentnej domácnosti. Je dostupná pre iOS a android. Samotná aplikácia sa veľmi jednoducho používa a je nenáročná pre prvotného používateľa. Je jednoduchá a minimalistická viď obr. 3.3.

Z hľadiska funkčnosti vyzerá byť aplikácia funkčná, avšak k plnému otestovaniu aplikácie je potrebné vlastniť inteligentné zariadenia špecifické pre VeSync, nebolo možné hlbšie otestovať funkčnosť aplikácie.



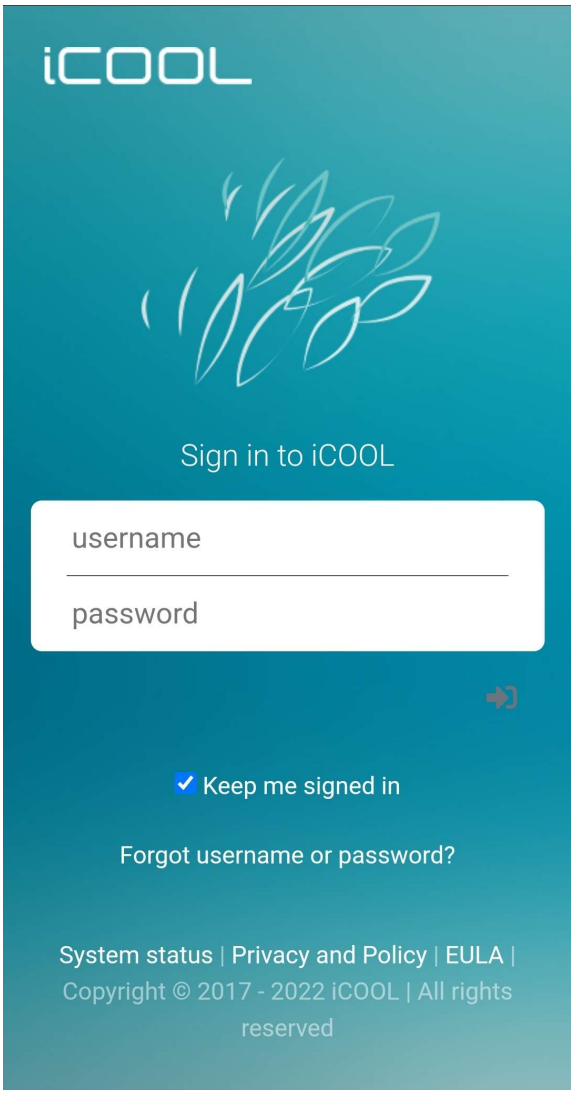
Obr. 3.1: Jazykové nezhody aplikácie iCOOL4.

Prihlasovanie do aplikácie je tiež klasické viď obr. 3.4, čiže pomocou emailu a hesla. Pri opätovnom prihlásení, rovnako ako pri aplikácií iCOOL4, by bolo vhodné pridať dodatočnú autentizáciu, pretože aplikácia sa rovno prihlási pri spustení.

3.3 Aplikácia Smart Life

Aplikácia Smart Life je hlavne zameraná na prepojenie chytrej domácnosti s Google Home alebo Amazon Echo. Aplikácia je jednoduchá, intuitívna a dobre sa používa.

Z hľadiska funkčnosti aplikácia ponúka viacero fukcií ako napr. nastavenie časového spustenia, manuálne vytvorenie a spustenie nejakej akcie alebo inej funkcie a automatizáciu domácnosti. Rovnako ako pri aplikácii VeSync, nebolo možné otestovať aplikáciu do jej plného potenciálu z dôvodu absencie inteligentných zariadení



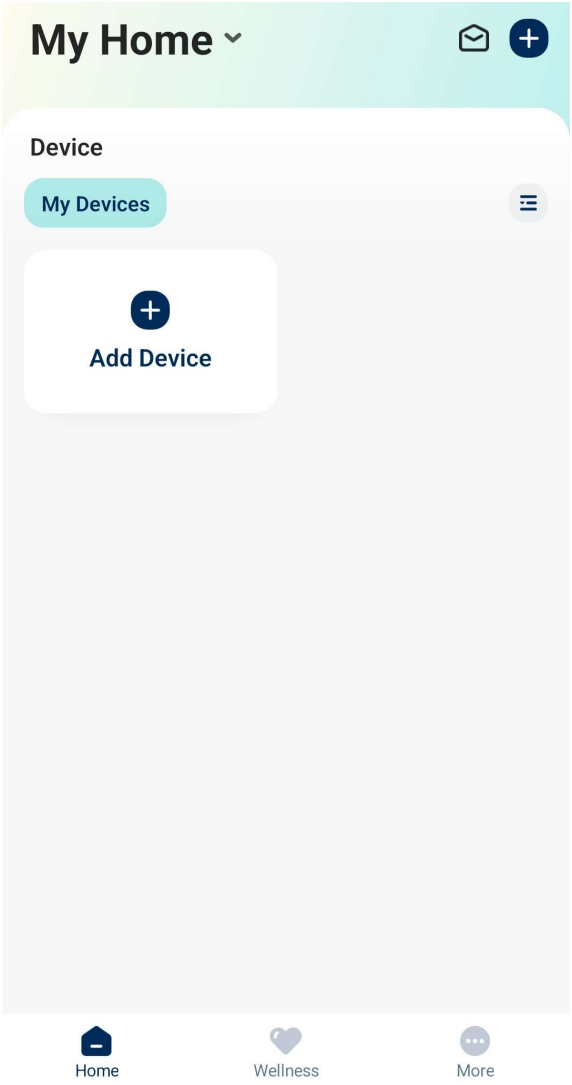
Obr. 3.2: Prihlasovacia obrazovka aplikácie iCOOL4.

špecifických pre ňu.

Metódy prihlásenia ponúka táto aplikácia viacero viď obr. 3.5. Okrem klasického ponúka aj prihlásenie pomocou odtlačku prstu viď obr. 3.6. a prihlásenie pomocou kresleného vzoru.

3.4 Finálna analýza a porovnanie

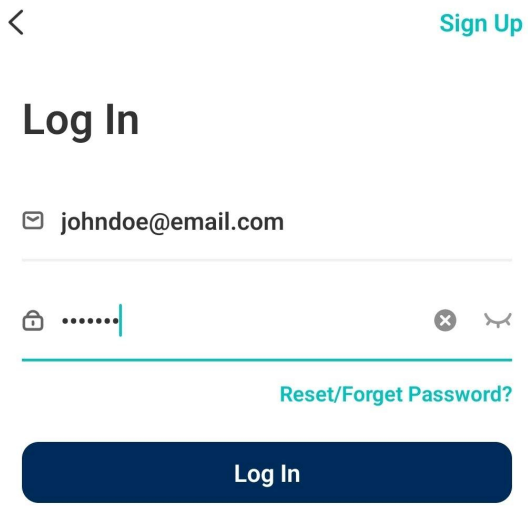
Výsledky analýzy aplikácií a ich zjednodušené zobrazenie sa nachádza v tabuľke 3.1. Z výsledkov je zrejmé, že aplikácia Smart Life disponuje jednoduchým UI, je funkčná a obsahuje dodatočné metódy prihlásenia.



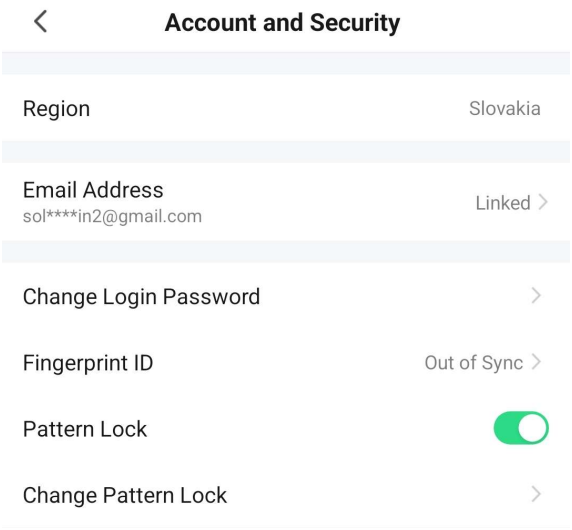
Obr. 3.3: Domovská obrazovka aplikácie VeSync.

Tab. 3.1: Tabuľka výsledkov analýzy.

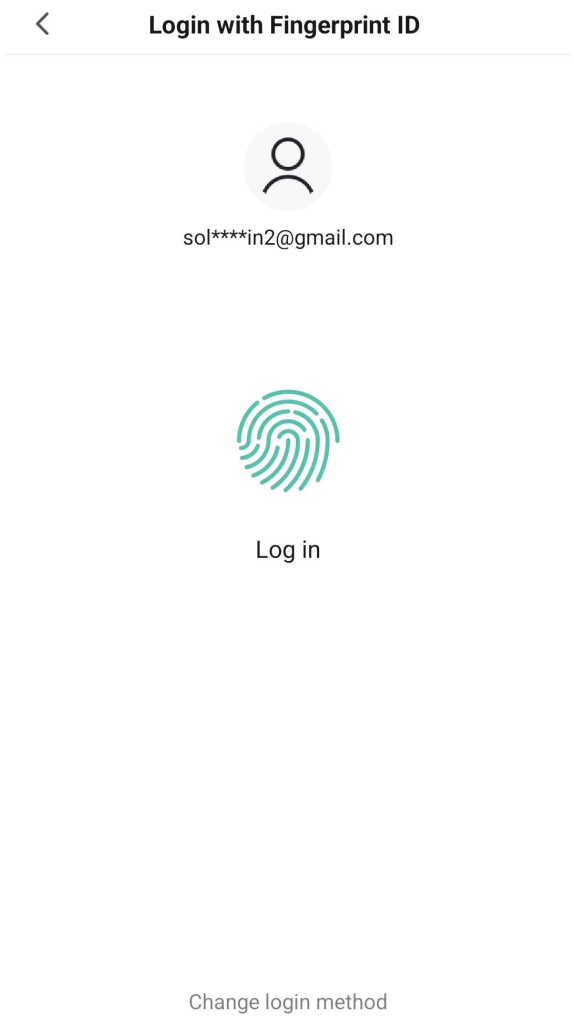
Aplikácia	Jednoduchosť	Funkčnosť	Prihlasovacie metódy
iCOOL4	Zložité UI	Malé chyby	Email + heslo
VeSync	Jednoduché UI	Funkčná	Email + heslo
Smart Life	Jednoduché UI	Funkčná	Email + heslo, biometrika, vzor



Obr. 3.4: Prihlasovacia obrazovka aplikácie VeSync.



Obr. 3.5: Dostupné prihlasovacie metódy



Obr. 3.6: Prihlasovacia obrazovka aplikácie Smart Life.

4 Výsledná aplikácia

Výsledkom praktickej práce je mobilná aplikácia umožňujúca prihlásenie používateľa na server klasickým spôsobom, ale aj s využitím biometrického systému. Ďalej aplikácia umožňuje voľbu metódy opätovného prihlásenia, ovládanie zariadení a vizualizáciu dát z rôznych typov inteligentných zariadení získaných pomocou REST API zo serveru. Aplikácia taktiež umožňuje pridávať zariadenia poprípade vytvorenie nového používateľa.

4.1 Služba Thingsboard

Aby aplikácia mohla fungovať potrebuje prístup ku zdrojom na serveri. Toto je zabezpečené prostredníctvom služby *Thingsboard*, ktorá beží na serveri, a stará sa o manažovanie zariadení, rôznych skupín a ich práv, spravuje jednotlivých používateľov.

Zariadenia sú pripojené k serveru pomocou MQTT (MQ Telemetry Transport) protokolu a sú priradené jednotlivým používateľom. Tento fakt sa neskôr využíva pri rozdelení práv prístupu k zariadeniam aby používateľ mal prístup len k jeho zariadeniam a nevedel získať prístup k ostatným zariadeniam a zneužiť ich. Prideľovanie zariadení, vytváranie používateľov a pod. je možné vykonať v manažment sekcii webovej aplikácie, ktorú poskytuje Thingsboard server.

Ďalej môže mať každá entita serveru, teda používateľ, customer, zariadenie a pod. vlastné atribúty. Tieto atribúty môžu byť:

- klientské,
- serverové,
- a zdieľané.

Klienstké atribúty sa využívajú pri zariadeniach a slúžia na zobrazovanie stavu o zariadení resp. aké dáta zariadenie posiela serveru. Tieto atribúty môže meniť len konkrétne zariadenie. Serverové atribúty zase slúžia na rôznu konfiguráciu entít. Tieto môžu byť menené pomocou REST API ale zariadenia k nim nemajú prístup. Posledným typom atribútov sú zdieľané atribúty. Tieto môžu byť menené ako zariadením tak aj pomocou REST API a slúžia primárne na konfiguráciu zariadenia. Zariadenie si periodicky vyčítava hodnoty v zdieľaných atribútoch a aktualizuje si ich vo svojej pamäti.

Thingsboard server ďalej poskytuje REST API pre prístup a manipuláciu so zariadeniami, používateľmi a pod. Dokumentácia poskytovanej REST API sa nachádza na tejto[1] adrese. Príklad komunikácie prostredníctvom REST API, konkrétne ide o

[1] https://thingsboard.cloud/swagger-ui/

požiadavku, ktorá má za úlohu overiť prihlasovacie údaje používateľa, je znázornený na výpise 4.1. Po odoslaní tejto požiadavky sa očakáva odpoveď od serveru. V hlavičke odpovedi v poli „status“ sa nachádza číselný kód, ktorý je využívaný v HTTP metodách a neskôr sa využíva na zhodnotenie výsledku poslanej požiadavky a telo odpovedi s JSON štruktúrou viď výpis 4.2. [24]

Výpis 4.1: Ukážka kódu, ktorý komunikuje s REST API.

```
Future<http.Response> login(
  String email, String passwd
) async {
  Uri url = Uri.https(_serverUrl, "/api/auth/login");
  return await _httpPost(
    url,
    headers: <String, String>{
      'Content-Type': 'application/json; charset=UTF-8',
    },
    body: jsonEncode(<String, String>{
      "username": email,
      "password": passwd,
    }),
  );
}
```

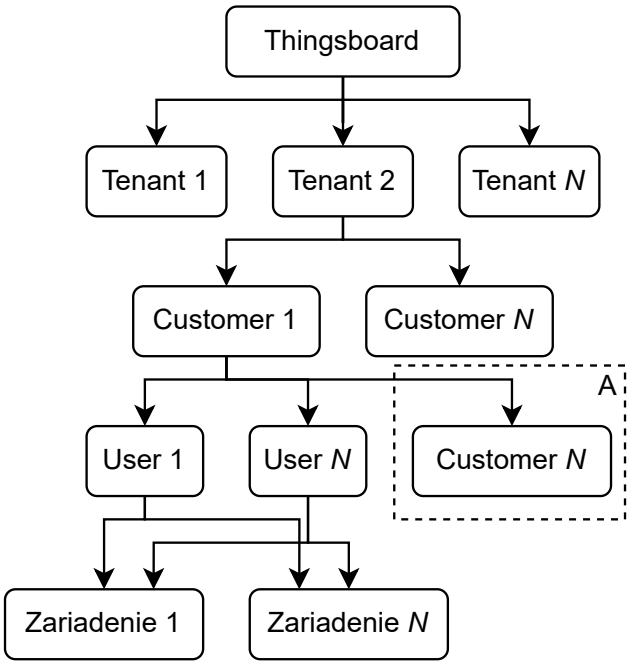
Výpis 4.2: Ukážka tela odpovedi prijatej od serveru.

```
{
  "message": "User account is not active",
  "errorCode": 10,
  "status": 401,
  "timestamp": "2022-11-18T10:08:23.154+00:00"
}
```

4.2 Princíp práv prístupu

Práva prístupu k daným zariadeniam sú vyriešené pomocou služby Thingsboard. Keďže sa táto služba automaticky stará o manažment používateľov a ich práv kde a k akým zariadeniam majú prístup, je vhodné využiť túto funkcionalitu. Hierarchia používateľov v službe Thingsboard je následovná:

- Tenant - prenajímateľ,



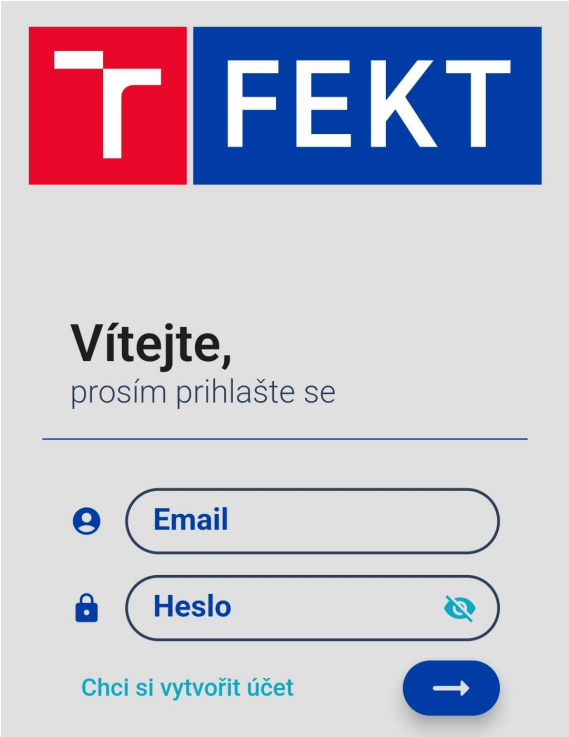
Obr. 4.1: Štruktúra práv v službe Thingsboard

- Customer - zákazník,
- User - používateľský účet.

Každý tenant môže mať N customerov. Tenant poskytuje zariadenia a prideľuje ich jednotlivým customerom. Customer zasa môže mať ďalších N customerov, ale táto funkcionalita nebude využitá v tejto práci. Ak má customer priradené zariadenia, jednotlivý používatelia k nim majú prístup a môžu s nimi komunikovať. Jednotlivý používatelia sa môžu prihlasovať pomocou používateľských účtov, ktoré sa vytvárajú osobitne pre každého Customera. Toto je znázornené na obr. 4.1, kde úsek A reprezentuje časť kedy môže mať jeden customer ďalších customerov. [25]

Samotný princíp je teda postavený na tom, že sa používateľ nachádza v skupine customer a ten má pridelené nejaké zariadenia. Používateľ tak vidí len zariadenia, ktoré mu boli pridelené. Preto sa v procese prihlásenia získa aj *customerID* (ID zákazníka) a to sa neskôr využíva na filtrovanie zariadení, ktoré patria len aktuálnemu používateľovi. Toto je bližšie popísané v nasledujúcej sekcii.

Ďalej sa prostredníctvom serverových atribútov používateľa zisťuje aké typy zariadení mu boli pridelené. Keďže sa na domovskej stránke nachádza viacero typov zariadení, tak sa týmto spôsobom obmedzi prístup len k tým, ktoré mu sú povolené.



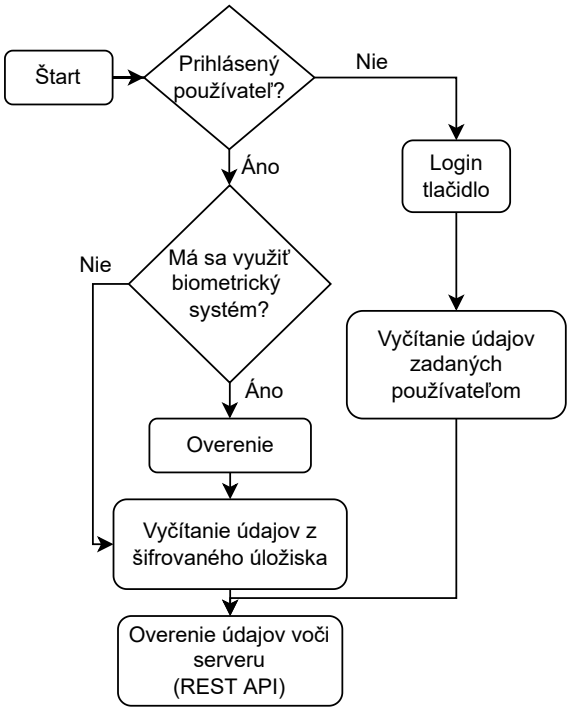
Obr. 4.2: Prihlasovacia obrazovka aplikácie

4.3 Proces prihlásenia používateľa

Po spustení mobilnej aplikácie sa zobrazí úvodná prihlasovacia obrazovka aplikácie, ktorá je znázornená na obr. 4.2, kde používateľ zadáva svoje prihlasovacie údaje.

Počas procesu prihlásenia používateľa sa najprv zisťuje, či je daný používateľ prihlásený. Ak daný používateľ nie je prihlásený, pokračuje sa klasickým procesom, čiže sa čaká pokiaľ používateľ zadá svoje prihlasovacie údaje a stlačí tlačidlo *Prihlásiť sa*. Potom sa tieto údaje overia voči serveru pomocou REST API. Ak je používateľ prihlásený, pokračuje sa získaním jeho nastavení z pamäte telefónu a podľa nich sa rozhoduje, či sa má využiť biometrický systém alebo nie. Ak má používateľ nastavené prihlasovanie pomocou biometrického systému, tak sa autentizuje a prihlasovacie údaje sa vyčítajú zo šifrovaného úložiska a overia sa voči serveru. Posledný prípad je, keď používateľ je prihlásený, ale nechce využiť biometrický systém. V tento moment sa rovno prihlasovacie údaje vyčítajú zo šifrovaného úložiska a overia sa voči serveru. Tento proces je znázornený na obrázku 4.3 a časť kódu zabezpečujúca autentizáciu voči serveru je ukázaná v prílohe A.1.

Ak bolo overenie úspešné, v odpovedi od serveru sa bude nachádzať *JWT Token*, ktorý sa neskôr pridáva do hlavičky ostatných REST API požiadaviek a tým pádom sa používateľ autentizuje a autorizuje na serveri bez nutnosti znova odosielať prihlasovacie údaje. Ďalej sa získajú aj dodatočné informácie o používateľovi,

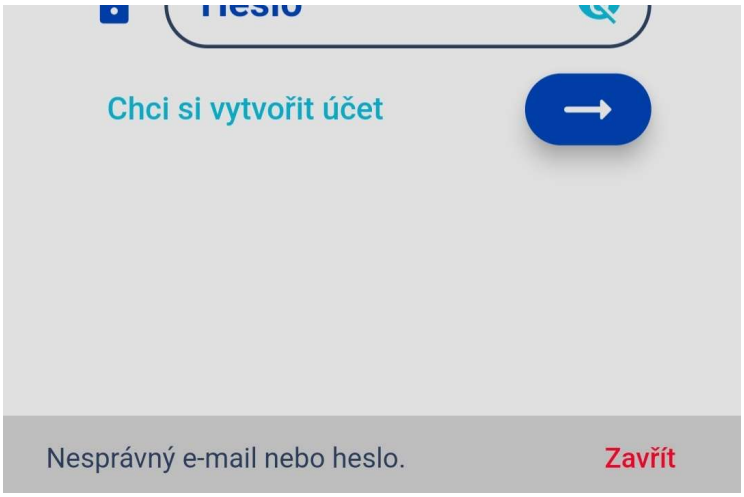


Obr. 4.3: Proces prihlásenia

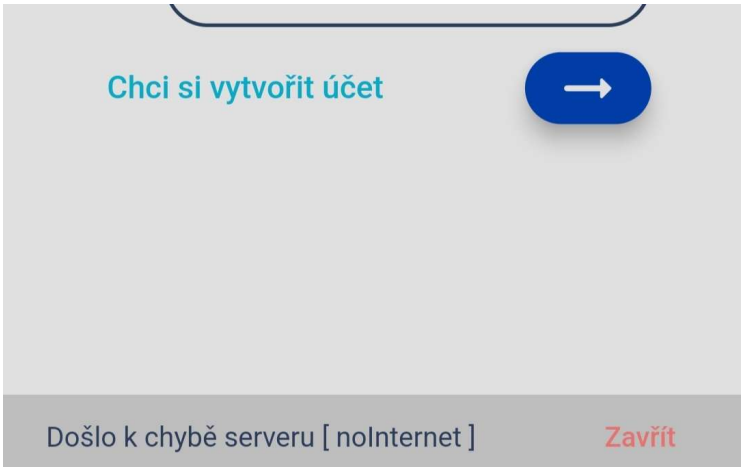
ako napr. jeho meno, priezvisko, zákaznícke ID a používateľské ID. Tieto údaje sa neskôr využívajú na filtráciu zariadení, ktorá je popísaná v ďalšej sekcii. Ak by nastala situácia kedy používateľ zadá nesprávne prihlasovacie údaje, tak aplikácia dostane odpoveď zo serveru s kódom **401** a správou *Invalid username or password* viď výpis 4.3. Odpoveď sa spracuje a používateľovi sa vypíše v novom okne, že zadal nesprávne prihlasovacie údaje viď obr. 4.4. Podobným spôsobom sa spracuje aj prípad, keď server odpovie s kódom iným ako **200** čo je kód naznačujúci úspešný priebeh, a kódom **401**, ktorý je popísaný vyššie. Avšak v tomto prípade bude používateľ informovaný správou *Nastala chyba serveru* a v hranatých zátvorkách bude bližšie popísaný dôvod, čo sa stalo. V ukážke je zobrazený prípad, keď používateľ nemá funkčné pripojenie na internet viď obr. 4.5.

Výpis 4.3: Telo odpovede pri nesprávnych prihlasovacích údajoch

```
{
  "message": "Invalid username or password",
  "errorCode": 10,
  "status": 401,
  "timestamp": "2022-11-18T10:09:49.839+00:00"
}
```



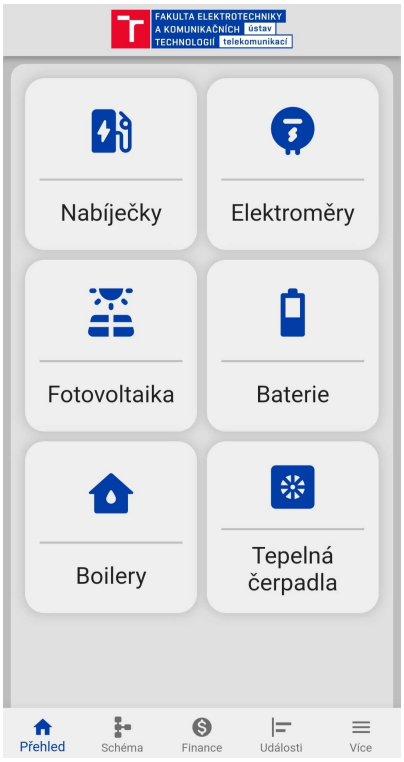
Obr. 4.4: Hláška nesprávnych údajov



Obr. 4.5: Hláška chyby serveru

4.4 Proces spracovania údajov o zariadeniach

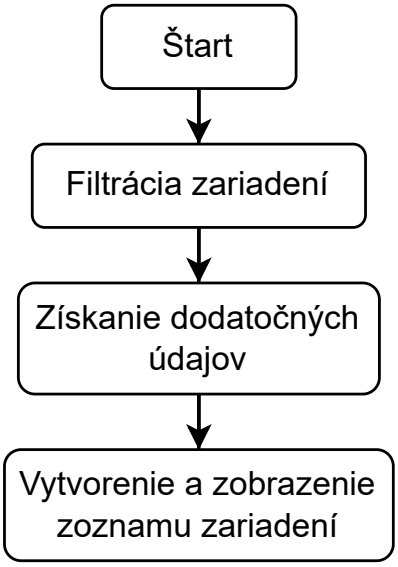
Po úspešnom prihlásení sa zobrazí domovská obrazovka aplikácie, ktorá zobrazí zoznam rôznych typov dostupných zariadení pre daného používateľa viď obr. 4.6. Jednotlivé typy zariadení potom majú svoje zoznamy dostupných zariadení viď obr. 4.7. Tieto zoznamy si používateľ vie manuálne aktualizovať „potiahnutím nadol“.



Obr. 4.6: Domovská obrazovka aplikácie

Proces získavania údajov o zariadeniach je znázornený na obr. 4.8 a príslušný kód sa nachádza v prílohe A.3 a A.4. Po úspešnom prihláseni používateľa sa najprv vezme jeho *zákaznícke ID*, ktoré bolo získané počas prihlasovania a na základe toho sa potom vyfiltrujú zariadenia, ktoré patria len tomuto ID viď výpis A.2. Po úspešnom

Obr. 4.7: Zoznam zariadení



Obr. 4.8: Proces získavania zariadení

získaní zoznamu zariadení sa potom pre každé zariadenie vytvorí interný model a získajú sa detailnejšie údaje ako napr. názov zariadenia, jeho typ a jeho parametre. Parametre, ktoré sa majú získať, sú implementované v samotných modeloch viď výpis 4.4. Tieto sú využité napr. pri vytváraní zoznamu zariadení, kde sa využije názov zariadenia viď obr. 4.7 alebo pri zobrazovaní dát o zariadení ako napr. aktuálne nastavenia nabíjačky.

Výpis 4.4: Získavané parametre modelu nabíjačky

```
@override
Map<String, String> getAttributeKeys() {
    return {
    'SHARED_SCOPE': 'all',
    'CLIENT_SCOPE': 'evse_status_wh_in_this_session,'
        'evse_status_wh_all_time_charged,'
        'evse_status_allowed_charging_current,'
        'evse_status_charging_authorized,'
        'evse_status_state_of_electric_vehicle,'
        'evse_status_state_of_charging',
    'SERVER_SCOPE': 'calculated_watt,'
        'location_lat,'
        'location_lon',
    };
}
```

4.5 Využitie biometrickej autentizácie

Biometrická autentizácia sa využíva hneď po zapnutí mobilnej aplikácie. Ak sa používateľ prihlási po prvý krát, sú do šifrovanej pamäti telefónu uložené jeho prihlasovacie údaje. Tieto tam ostanú až pokiaľ používateľ neklikne na tlačidlo „odhlásiť sa“. Kvôli zvýšeniu bezpečnosti sa prístup k uloženým prihlasovacím údajom môže regulovať pomocou biometrického systému. Toto sa deje pri opätovnom zapnutí aplikácie. Ak si používateľ zvolil metódu opätovného prihlásenia pomocou biometrického systému, tak sa využije dodatočná autentizácia pomocou biometrického systému na prístup k údajom. Ak si používateľ nezvolil túto metódu, tak sa zo šifrovaného úložiska rovno vyčítajú prihlasovacie údaje a aplikácia sa prihlási.

Používateľ si môže zvoliť metódy opätovného prihlásenia pomocou stránky nastavení viď obr. 4.9 do ktorej sa dostane pomocou stránky „Více“ s ostatnými možnosťami viď obr. 4.10. Nastavenie funguje tak, že pri vybratí možnosti sa pomocou manažmentu stavov uloží do šifrovaného úložiska telefónu hodnota či sa má využiť biometrický systém pri prihlasovaní alebo nie, a ktorá je potom pri opätovnom prihlásení overovaná.

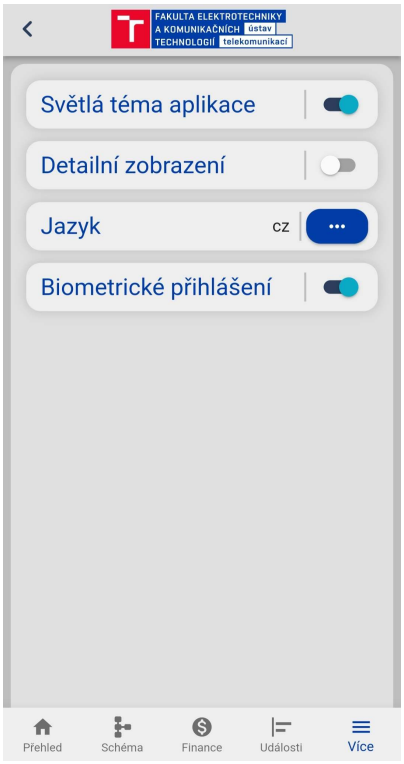
Šifrované úložisko je využité preto, aby sa prihlasovacie údaje mohli bezpečne uložiť a neskôr pri automatickom prihlasovaní využiť. Samozrejme keďže ide o citlivé údaje musia byť zabezpečené aby sa k nim nedalo pristupovať ako k bežným dátam. Toto zabezpečuje knižnica „flutter_secure_storage“, ktorá sa stará o prístup k úložisku telefónu a následné šifrovanie a dešifrovanie.

Na platforme iOS je k tomuto účelu využitá API Keychain. Na platforme Android sa zasa využíva AES protokol. Súkromný kľúč AES je neskôr zašifrovaný pomocou RSA protokolu a nakoniec je súkromný kľúč RSA uložený v KeyStore, ktorý poskytuje operačný systém Android. [26] [27] [28]

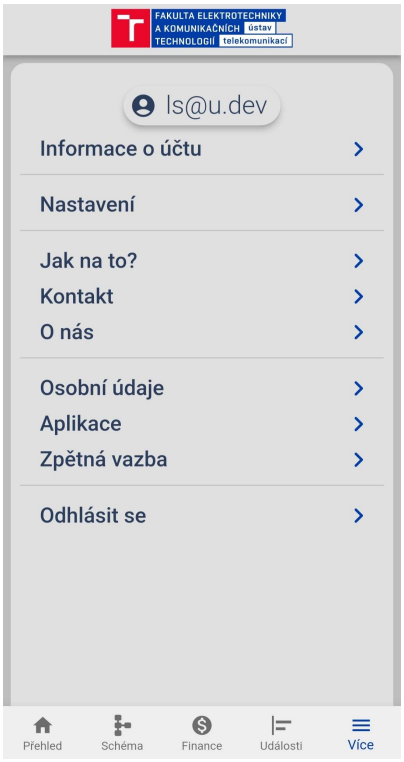
4.6 Registrácia nového používateľa

Ak by nastala situácia, kedy sa používateľ chcel prihlásiť ale ešte nemá vytvorený svoj účet, mobilná aplikácia poskytuje možnosť odosielať požiadavky na vytvorenie nového účtu. Registračný formulár sa nachádza na obrazovke registrácie viď obr. 4.11.

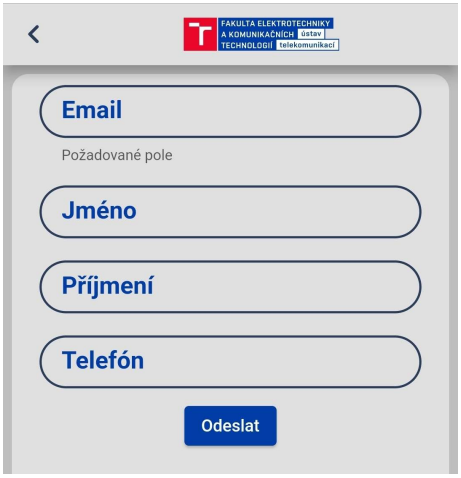
Aplikácia neumožňuje vytvorenie používateľa „automaticky“, to znamená, že sa len pošle email administrátorovi o tom, že sa chce používateľ zaregistrovať a s akými údajmi. Je to tak pretože REST API poskytuje endpoint na vytváranie účtov len používateľom s právami tenanta. Avšak tenant sa nachádza v hierarchii práv úplne navrchu, takže by bolo z hľadiska bezpečnosti nevhodné, aby aplikácia mala v pozadí



Obr. 4.9: Stránka s nastaveniami



Obr. 4.10: Navigácia na stránku s nastaveniami



Obr. 4.11: Stránka s registračným formulárom

prístup aj na tento zabezpečený API endpoint bez interakcie s overeným používateľom.

4.7 Zabezpečenie prenosu dát

Keďže sa v mobilnej aplikácií využívajú a posielajú citlivé dáta ako napr. prihlasovacie údaje alebo osobné údaje cez internet, bolo potrebné zabezpečiť dôvernosť dát. V tomto prípade sa využíva TLS protokol a teda komunikácia so serverom je šifrovaná.

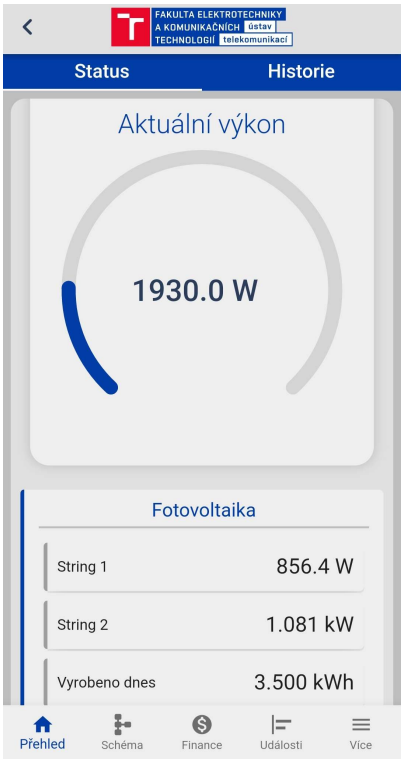
To, že je komunikácia šifrovaná sa v aplikácii využilo hlavne pri autentizácii voči serveru pretože REST API Thingsboard serveru poskytuje prihlasovací endpoint, kde sa v HTTP requeste očakáva e-mail a heslo v čitateľnom formáte, nie hash hesla. Týmto spôsobom sa odstránila jedná veľká zraniteľnosť Thingsboard serveru a to bolo posielanie hesiel v čitateľnom formáte.

4.8 Zobrazovanie dát a ovládanie zariadení

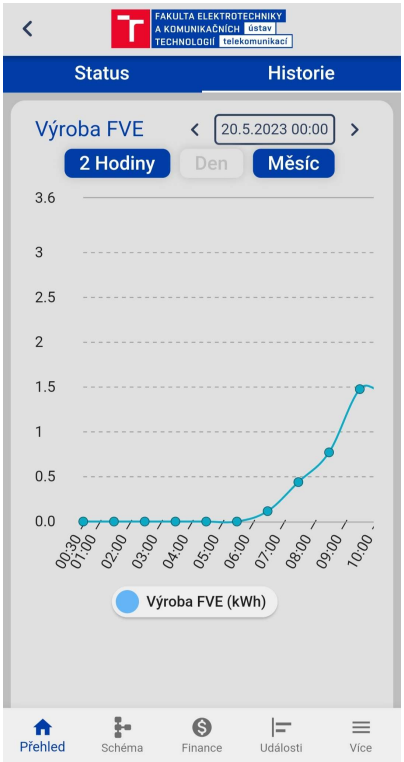
Pre používateľa je vhodné zobrazovať aj údaje zo zariadení. Toto bolo dosiahnuté tromi spôsobmi:

- zobrazovanie nespojitých dát v jednoduchom poli viď obr. 4.12,
- zobrazovanie spojitých dát pomocou grafov viď obr. 4.13,
- a zobrazovanie na stránke „Schéma“ viď obr. 4.14

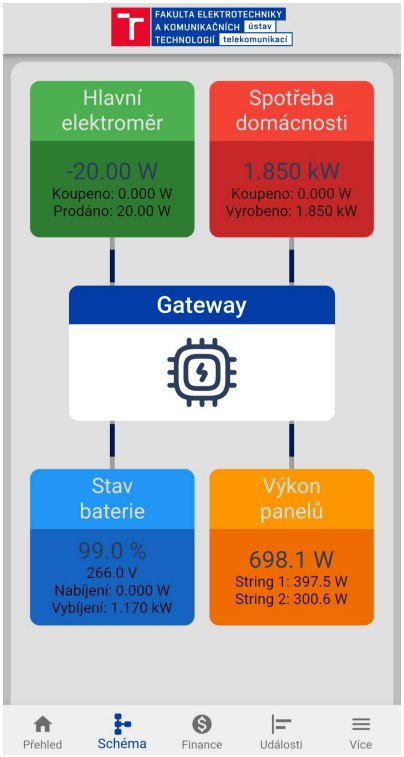
Každý typ zariadenia má svoj unikátny štýl zobrazenia dát na základe toho, čo je potrebné zobraziť. Priebeh získavania dát začína pri výbere typu zariadenia na domovskej stránke aplikácie. Potom nasleduje výber konkretného zariadenia zo



Obr. 4.12: Stránka s jednoduchým zobrazovaním dát



Obr. 4.13: Stránka s grafom



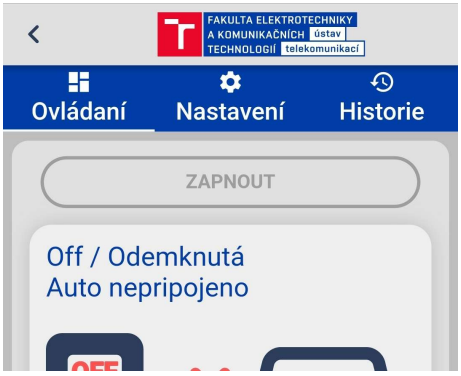
Obr. 4.14: Stránka so schémou

zoznamu dostupných zariadení a následné uloženie zvoleného zariadenia do manažmentu stavu aplikácie. Aplikácia si potom z tohto zvoleného modelu zariadenia vezme jeho unikátne ID a prostredníctvom REST API si požiada Thingsboard server o potrebné dáta. Tie sa potom zobrazia v príslušnom widgetu. Zároveň sa spustí časovač, ktorý periodicky posiela požiadavku na Thingsboard server o nové dáta a tým pádom ich obnovuje.

Prvým typom dát, o ktorý sa môže jednať, sú atribúty zariadenia, ktoré aktualizujú svoj stav len pri zmene ich hodnoty. Tieto atribúty slúžia skôr ako konfiguračné údaje a menia sa menej často napr. stav nabíjania pri nabíjačkách. Ďalším typom dát, ktoré sú zariadenia schopné poskytnúť je telemetria. Telemetria sa aktualizuje pravidelne a slúži na dáta, ktoré sa menia často a sú často spojité, napr. teplota boileru. V aplikácií sa využívajú obidva tieto typy avšak primárne je využívaná telemetria zariadení. Telemetria je potom zobrazovaná podľa konkrétneho typu dát, čiže ak dáta reprezentujú spojitú premennú hodnotu, sú spracované vo forme grafu, ak niesú spojité tak sú reprezentované formou jednoduchého informačného poľa.

Na zobrazovanie nespojitých dát nebolo potrebné využívať žiadne dodatočné UI knižnice. Na zobrazovanie spojitých dát bola využitá knižnica fl_chart[1]. Je to knižnica poskytujúca rôzne grafy a je veľmi flexibílna čo sa týka možností konfigurácie

[1]https://pub.dev/packages/fl_chart



Obr. 4.15: Stránka s ovládaním nabíjačky

grafov. Pri zobrazovaní spojitých dát bolo avšak potrebné vypočítavať začiatočnú časovú značku, teda od kedy sa majú nazbierané dáta poslať, koncovú časovú značku, teda do kedy sa majú nazbierané dáta poslať a agregačný interval, ktorý slúži na agregáciu dát z veľkej množiny dát na menšiu. Po výpočte týchto parametrov sa už môže poslať pomocou REST API požiadavka na server, ktorý vráti dáta z požadovaného časového intervalu a s požadovanovým agregačným intervalom. Samotnú agregáciu vykonáva server. Po prijatí odpovedi zo serveru su dáta vo formáte JSON, ktorý sa ďalej spracuje podľa potreby daného grafu.

Ovládanie zariadení bolo dosiahnuté pri zariadeniach typu nabíjačka, kedy je potrebné ju ovládať a nastavovať rôzne parametre napr. povolený nabíjací prúd. Na ovládanie zariadení Thingsboard využíva zdieľané atribúty, ktoré sa dajú meniť pomocou REST API. Hlavná ovládacia obrazovka nabíjačky poskytuje základné údaje o priebehu nabíjania a ovládacie tlačidlo on/off viď obr. 4.15. Ďalej sa na tejto stránke nachádza záložka s nastaveniami kde sa využívajú zdieľané atribúty viď obr. 4.16. Pri zmene niektorého z nastavení sa pomocou manažmentu stavov pošle HTTP požiadavka na zmenu zdieľaného atribútu na server. Funkcia, ktorá má za úlohu vykonať túto zmenu je zobrazená na výpise 4.5.

Výpis 4.5: Nastavovanie zdieľaných atribútov nabíjačky

```
1 Future<http.Response> changeDeviceAttribute(
2     Device device, String attribute, int value
3 ) async {
4     Uri url = Uri.https(_serverUrl,
5         "api/plugins/telemetry/DEVICE/" +
6             "${device.storableId}/SHARED_SCOPE"
7     );
8     return await _httpPost(
9         url,
```

```
        headers: <String, String>{
            'Content-Type': 'application/json',
            "X-Authorization": "Bearer ${_user.token}",
        },
        body: jsonEncode(<String, int>{
            attribute: value,
        }),
    );
}
```

Obr. 4.16: Stránka s nastaveniami nabíjačky

Záver

Cieľom bakalárskej práce bolo zoznámenie sa s návrhom a vývojom mobilných aplikácií v prostredí Flutter. Dôraz bol kladený aj na biometrické funkcie pri pri autentizácií používateľa a mal byť spracovaný príncip REST API pre komunikáciu s prvkami inteligentnej budovy. Ďalej mala byť spracovaná analýza dostupných riešení, navrhnutý príncip pre prístup k zariadeniam a implementované ovládanie zariadení, vizualizácia ich dát a biometrická autentizácia.

V kapitolách teoretickej časti bol popísaný vývoj mobilných aplikácií, celkový kolobeh vývoja softvéru, jeho dôležité body a boli popísané platformy android a iOS spolu s ich korenšpondujúcimi programovacími jazykmi. Ďalej bol spracovaný a vysvetlený príncip fungovania Flutter UI frameworku, boli popísané základné body a stromová štruktúra widgetov, ktoré tvoria Flutter aplikáciu. Taktiež je spracovaný natívny manažment stavov vo Flutter frameworku. Bol opísaný programovací jazyk Dart, ktorý je využívaný frameworkom Flutter. Následne sa práca zaoberala príncipom API a REST API, bola priblížena ich funkčnosť, a bolo vytvorené porovnanie HTTP metód a ich databázových eqvivalentov, ktoré boli následne popísané. Ďalej sa práca venovala opisu biometrického systému, jeho chybám, štruktúre a rozdeleniu na unimodálne a multimodálne systémy, ktoré boli tiež bližšie popísané. Taktiež boli spracované biometrické metódy a ich porovnanie. Metódy snímania odtlačku prsta a rozpoznania tváre boli viac priblížené.

V praktickej časti práce bola spravená analýza dostupných riešení. Boli popísané jednotlivé aplikácie a bolo vypracované ich porovnanie. Ďalej sa vytvoril príncip práv pre prístup k zariadeniam, bol vytvorený návrh prihlásenia používateľa, získavanie údajov o jednotlivých zariadeniach a bola spracovaná možnosť nastavenia metódy prihlásenia. Výsledkom praktickej časti je aplikácia umožňujúca klasické prihlásenie ale aj autentizáciu pomocou zvolenej biometrickej metódy. Vyzualizácia dát z rôznych typov zariadení je taktiež implementovaná formou jednoduchého zobrazenia ale aj zložitejších grafov. Ďalej aplikácia umožňuje registráciu použivateľa a obmedzuje prístup len na zariadenia patriacie danému účtu, ktoré vie používateľ ovládať ako napr. nabíjačky.

Literatúra

[1] divyanshu_gupta1. Software development life cycle (sdlc). [online], posledná revízia 5.7.2021. [cit. 2022-10-05]. URL: https://www.geeksforgeeks.org/software-development-life-cycle-sdlc/.

[2] Shady Boukhary and Eduardo Colmenares. A clean approach to flutter development through the flutter clean architecture package. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1115–1120, Dec 2019. doi:10.1109/CSCI49370.2019.00211.

[3] Ovidiu Constantin Novac, Mihaela Novac, Cornelia Gordan, Tamas Berczes, and Gyöngyi Bujdosó. Comparative study of google android, apple ios and microsoft windows phone mobile operating systems. In *2017 14th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 154–159, Jun 2017. doi:10.1109/EMES.2017.7980403.

[4] Ovidiu Constantin Novac, Cornelia Mihaela Novac, Bogdan Ciora, Cornelia Emilia Gordan, Mircea Ioan Gordan, and Gyöngyi Bujdosó. The rise of mobile development: a comparison between ionic and flutter. In *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–10, June 2022. doi:10.1109/ECAI54874.2022.9847460.

[5] Android runtime (art) and dalvik. [online], posledná revízia 13.9.2022. [cit. 2022-10-10]. URL: https://source.android.com/docs/core/runtime.

[6] Flutter architectural overview. [online]. [cit. 2022-10-08]. URL: https://docs.flutter.dev/resources/architectural-overview.

[7] Dart programming language specification 5th edition. [online], posledná revízia 9.4.2021. [cit. 2022-10-15]. URL: https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf.

[8] fluttercommunity. Redux motivation. [online], posledná revízia 29.11.2017. [cit. 2023-3-24]. URL: https://github.com/fluttercommunity/redux.dart/blob/master/doc/why.md.

[9] Dan Abramov and the Redux documentation authors. Redux fundamentals, part 1: Redux overview. [online], posledná revízia 28.4.2023. [cit. 2023-3-24]. URL: https://redux.js.org/tutorials/fundamentals/part-1-overview.

[10] Dan Abramov and the Redux documentation authors. Redux fundamentals, part 2: Concepts and data flow. [online], posledná revízia 30.4.2023.

[cit. 2023-3-24]. URL: https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow.

[11] fluttercommunity. Redux middleware. [online], posledná revízia 9.9.2019. [cit. 2023-3-24]. URL: https://github.com/fluttercommunity/redux.dart/blob/master/doc/async.md.

[12] IBM Cloud Education. What is a rest api? [online], posledná revízia 6.4.2021. [cit. 2022-10-18]. URL: https://www.ibm.com/cloud/learn/rest-apis.

[13] What is a rest api? [online], posledná revízia 8.5.2020. [cit. 2022-10-28]. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[14] MDN contributors. Http request methods. [online], posledná revízia 9.9.2022. [cit. 2022-10-20]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[15] MDN contributors. An overview of http. [online], posledná revízia 22.11.2022. [cit. 2022-10-19]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[16] Sourabh Shirhatti and Jimmy Campbell. Crud (create, read, update, delete). [online], posledná revízia 24.1.2019. [cit. 2022-10-20]. URL: https://learn.microsoft.com/en-us/iis-administration/api/crud.

[17] MDN contributors. Http headers. [online], posledná revízia 2.12.2022. [cit. 2022-10-19]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

[18] MDN contributors. Http response status codes. [online], posledná revízia 26.10.2022. [cit. 2022-10-20]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

[19] A.K. Jain, A. Ross, and S. Prabhakar. An introduction to biometric recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1):4–20, Jan 2004. doi:10.1109/TCSVT.2003.818349.

[20] A.K. Jain, A. Ross, and S. Pankanti. Biometrics: a tool for information security. *IEEE Transactions on Information Forensics and Security*, 1(2):125–143, Jun 2006. doi:10.1109/TIFS.2006.873653.

[21] Danny Thakkar. Unimodal biometrics vs. multimodal biometrics. [online]. [cit. 2022-10-22]. URL: https://www.bayometric.com/unimodal-vs-multimodal/.

[22] Waleed Dahea and H.S. Fadewar. Multimodal biometric system: A review. *International Journal of Engineering and Technology*, 4:25–31, Jan 2018. doi:10.13140/RG.2.2.34056.65287.

[23] A.K. Jain, Lin Hong, S. Pankanti, and R. Bolle. An identity-authentication system using fingerprints. *Proceedings of the IEEE*, 85(9):1365–1388, Sep 1997. doi:10.1109/5.628674.

[24] Swagger ui. [online]. [cit. 2022-11-01]. URL: https://thingsboard.cloud/swagger-ui/.

[25] Entities and relations. [online]. [cit. 2022-10-28]. URL: https://thingsboard.io/docs/pe/user-guide/entities-and-relations/.

[26] steenbakker.dev. flutter_secure_storage. [online]. [cit. 2023-3-24]. URL: https://pub.dev/packages/flutter_secure_storage.

[27] Keychain services. [online]. [cit. 2023-3-24]. URL: https://developer.apple.com/documentation/security/keychain_services#//apple_ref/doc/uid/TP30000897-CH203-TP1.

[28] Android keystore system. [online], posledná revízia 7.10.2022. [cit. 2023-3-24]. URL: https://developer.android.com/training/articles/keystore.html.

Zoznam symbolov a skratiek

AES Advanced Encryption Standard protocol

API aplikačné programovacie rozhranie – Aplication Programming Interface

FMR miera falošnej zhody – False Match Rate

FNMR miera falošnej nezhody – False Non-Match Rate

HTTP HyperText Transfer Protocol

JSON zápis javascript objektu – JavaSript Object Notation

MQTT MQ Telemetry Transport

REST reprezentačný stav prevodu – Representional Transfer State

RSA Rivest Shamir Adleman cryptographic protocol

TLS Transport Layer Security

UI užívateľské rozhranie – User Interface

A Časti dôležitého kódu

A.1 Overenie prihlasovacích údajov

Výpis A.1: Časť kódu zabezpečujúca prihlásenie a získanie dát

```
1  // Validate provided credentials on server
2  static Future<ResponseSharedAPI> loginServer(
3    String email, String passwd) async {
4    Response serverResponse = await
5      HttpHandler().login(email, passwd);
6    switch (serverResponse.statusCode) {
7      // Successful login
8      case 200:
9        _user = User(email);
10       _user.token =
11         HttpHandler().body(serverResponse)["token"];
12       HttpHandler().user = _user;
13       // Get user details
14       serverResponse =
15         await HttpHandler().userFilter(email);
16       switch (serverResponse.statusCode) {
17         // Success getting details
18         case 200:
19           var data =
20             HttpHandler().body(serverResponse)['data'];
21           // Permission error
22           if (data.isEmpty)
23             return ResponseSharedAPI.serverError;
24           data = data[0];
25           _user.customerId = data['customerId']['id'];
26           _user.name = data['firstName'];
27           _user.surname = data['lastName'];
28           _user.userId = data['id']['id'];
29           _user.authority = data['authority'];
30           _user.passwd = passwd;
31           Storable? storedUser =
32             await SharedAPI.getStorable(
33               StorableType.user, email);
```

```
34            if (storedUser != null) {
35              _user.loginWithBiometrics =
36                (storedUser as User).loginWithBiometrics;
37            }
38            _user.save();
39            Storage().writeLastLogin(email);
40            ReduxStore().store.dispatch(
41              SetUserAction(_user));
42            return ResponseSharedAPI.valid;
43          default:
44            return ResponseSharedAPI.serverError;
45        }
46      case 401:
47        return ResponseSharedAPI.invalidCredentials;
48      default:
49        return ResponseSharedAPI.serverError;
50    }
51  }
```

A.2 HTTP požiadavka na filtráciu zariadení

Výpis A.2: Vytvorenie HTTP požiadavky

```
1  Future<http.Response> getDevices() async {
2      Uri url;
3      if (_user.authority == 'TENANT_ADMIN') {
4        url = Uri.https(
5          _serverUrl,
6          "api/tenant/deviceInfos", {
7            "pageSize": "100",
8            "page": "0",
9          }
10       );
11     } else {
12       url = Uri.https(
13         _serverUrl,
14         "/api/customer/${_user.customerId}/devices", {
15           "pageSize": "100",
```

```
16            "page": "0",
17          }
18        );
19      }
20      return await _httpGet(
21        url,
22        headers: <String, String>{
23          "accept": "application/json",
24          "X-Authorization": "Bearer ${_user.token}",
25        },
26      );
27    }
```

A.3 Získanie zariadení a ich parametrov

Výpis A.3: Časť kódu zabezpečujúca získanie zariadení

```
1  static Future<List<Device>> getRemoteDevices(
2    DeviceType deviceType
3  ) async {
4    List<Device> result = [];
5    var devices = HttpHandler().body(
6      await HttpHandler().getDevices()
7    )["data"];
8    if (devices == null) return result;
9    for (var device in devices) {
10     switch (deviceType) {
11       case DeviceType.charger:
12         if (
13           device['type'] ==
14             Constants.chargerAttrIdentificator
15         ) {
16           Charger charger = Charger(device['id']['id']);
17           charger.name = device['name'];
18           charger.label = device['label'] ?? '';
19           ReduxStore()
20               .store
21               .dispatch(
```

```
22                updateDeviceAttributes(
23                  charger, 'CLIENT_SCOPE'
24                )
25              );
26          ReduxStore()
27              .store
28              .dispatch(
29                updateDeviceAttributes(
30                  charger, 'SERVER_SCOPE'
31                )
32              );
33          result.add(charger);
34        }
35        break;
36      case DeviceType.elmer:
37        if (
38          device['type'] ==
39            Constants.elmerAttrIdentificator
40        ) {
41          var type = HttpHandler().body(
42            await HttpHandler()
43              .getDeviceAttributes(
44                  device['id']['id'],
45                  'SERVER_SCOPE',
46                  'all'
47              )
48            )['data'];
49          for (var element in type) {
50            if (
51              element['key'] == 'is_main'
52                && element['value']
53            ) {
54              Elmer elmer = Elmer(device['id']['id']);
55              elmer.name = device['name'];
56              elmer.label = device['label'] ?? '';
57              result.add(elmer);
58            } else if (
59              element['key'] == 'is_fve'
60                && element['value']
```

```
61              ) {
62                Elmer elmer = Elmer(device['id']['id']);
63                elmer.name = device['name'];
64                elmer.label = device['label'] ?? '';
65                elmer.isFve = true;
66                result.add(elmer);
67              }
68            }
69          }
70          break;
71        case DeviceType.fve:
72          if (
73            device['type'] ==
74              Constants.fveAttrIdentificator
75          ) {
76            FVE fve = FVE(device['id']['id']);
77            fve.name = device['name'];
78            fve.label = device['label'] ?? '';
79            result.add(fve);
80          }
81          break;
82        case DeviceType.battery:
83          if (
84            device['type'] ==
85              Constants.batteryAttrIdentificator
86          ) {
87            Battery battery = Battery(device['id']['id']);
88            battery.name = device['name'];
89            battery.label = device['label'] ?? '';
90            result.add(battery);
91            break;
92          }
93          break;
94        case DeviceType.boiler:
95          if (
96            device['type'] ==
97              Constants.boilerAttrIdentificator
98          ) {
99            if (
```

```
100             device['name'].toString()
101               .split('_').last != '2'
102           ) {
103             Boiler boiler = Boiler(device['id']['id']);
104             boiler.name = device['name'];
105             boiler.label = device['label'] ?? '';
106             result.add(boiler);
107           }
108           break;
109         }
110         break;
111       case DeviceType.inverter:
112         if (
113           device['type'] ==
114             Constants.inverterAttrIdentificator
115         ) {
116           Inverter inverter = Inverter(
117             device['id']['id']
118           );
119           inverter.name = device['name'];
120           inverter.label = device['label'] ?? '';
121           result.add(inverter);
122           break;
123         }
124         break;
125       case DeviceType.none:
126         break;
127       default:
128         throw Exception(
129           'Tried to get unknown device type: $deviceType'
130         );
131     }
132   }
133   return result;
134 }
```

Výpis A.4: Časť kódu zabezpečujúca získanie parametrov zariadení

```
1 static Future<void> updateDeviceAttributes(
```

```
2   Device device, String scope) async {
3   Response response;
4   response = await HttpHandler().getDeviceAttributes(
5       device.storableId,
6       scope,
7       device.getAttributeKeys()[scope] ?? 'all'
8   );
9   if (response.statusCode == 200) {
10    Map<String, dynamic> data =
11      HttpHandler().body(response);
12    bool proceed = false;
13    proceed = device is Charger
14      || device is FVE ? true : false;
15    if (proceed) {
16      device.updateAttributes(data['data']);
17    }
18  }
19 }
```

B Obsah elektronickej prílohy

Táto príloha obsahuje zdrojový kód aplikácie. Verzie komponentov s ktorými bola aplikácia testovaná:

- Flutter - 3.9.0-11.0.pre.3,
- Dart - 3.0.0.

```
/ ..... koreňový adresár
├── android ..... vygenerované a konfiguračné súbory platformy android
│   ├── local.properties ..... nastavenie verzií
│   └── app/src/main/AndroidManifest.xml ..... nastavenia android aplikácie
├── assets/ ..... ikony a loga využité v aplikácií
├── ios ..... vygenerované a konfiguračné súbory platformy iOS
│   └── Runner/Info.plist ..... nastavenia iOS aplikácie
├── lib/ ..... hlavná zložka obsahujúca zdrojový kód
│   ├── api/ ..... komunikácia s externými zdrojmi
│   │   ├── http_handler.dart ..... HTTP požiadavky
│   │   ├── shared_api.dart ..... knižnica na zjednodušenie používania zdrojov
│   │   ├── storage.dart ..... knižnica na komunikáciu so šifrovaným úložiskom
│   │   └── redux_store.dart .... knižnica na zjednodušenie prístupu k REDUX store
│   ├── error_handling/ ..... spracovanie errorov
│   ├── models/ ..... modely
│   ├── native/ ..... komunikácia s native platformou
│   ├── pages/ ..... stránky aplikácie
│   ├── service/ ..... služby aplikácie ( notifikácie)
│   ├── utils/ ..... rôzne utility funkcie
│   │   ├── actions/ ..... REDUX akcie stavov
│   │   ├── i18n/ ..... preklady a k tomu potrebné funkcie
│   │   ├── states/ ..... REDUX stavy
│   │   ├── constants.dart ..... konštanty
│   │   ├── keys.dart ..... flutter kľúče
│   │   ├── theme.dart ..... téma aplikácie
│   │   └── utils.dart ..... utility funkcie
│   ├── widgety/ ..... widgety rozdelené podľa ich lokácie v aplikácii
│   └── main.dart ..... hlavný súbor aplikácie
├── pubspec.lock ..... vygenerovaný súbor balíčkov
└── pubspec.yaml ..... hlavný konfiguračný súbor
```