Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Externí display pro embedded zařízení

Plzeň, 2020

Martin Forejt

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 7. května 2020

Martin Forejt

Abstract

This bachelor thesis describes the creation and usage of a mobile application for the Android operating system, which allows connection to embedded devices and systems (using Wi-Fi, Bluetooth and Bluetooth Low Energy), their control and display of received data, using HTML pages displayed inside this application. Part of this application is a library in JavaScript, which allows the connection between HTML pages and the Android environment and thanks to that the actual communication with the device is moved to the displayed HTML pages, allowing the application to connect to almost any device regardless of its protocol, provided HTML pages are created tailored to each device.

Abstrakt

Tato práce popisuje vytvoření a použití mobilní aplikace pro operační systém Android, která umožňuje připojení k vestavěným zařízením a systémům (pomocí Wi-Fi, Bluetooth a Bluetooth Low Energy), jejich ovládání a zobrazování přijatých dat a to za pomoci HTML stránek zobrazených uvnitř této aplikace. Součástí této aplikace je knihovna v JavaScriptu, která umožňuje propojení mezi HTML stránkami a prostředím Androidu a díky tomu je vlastní komunikace se zařízením přesunuta právě do zobrazovaných HTML stránek, umožňující tak aplikaci připojení k téměř jakémukoli zařízení nezávisle na jeho protokolu, pod podmínkou vytvoření HTML stránek každému zařízení na míru.

Obsah

1 Úvod **4**

2 Vestavěné systémy **5**

2.1 Definice vestavěného systému 5

2.2 Historie 5

2.3 Rozdělení 5

2.3.1 Dle funkčních požadavků 5

2.3.2 Dle výkonu mikrokontroléru 6

2.4 Příklady použití 6

3 Externí komunikace **7**

3.1 Důvody pro externí komunikaci 7

3.2 Způsoby externí komunikace 7

3.2.1 Datová komunikace 7

3.2.2 Vzdálená obrazovka 8

3.3 Bezpečnost komunikace 8

3.4 Vybrané protokoly 8

3.4.1 Wi-Fi 8

3.4.2 Bluetooth 9

3.4.3 Bluetooth Low Energy 9

4 Existující řešení **10**

4.1 IoT MQTT Dashboard 10

4.2 MQTT Dash 10

4.3 IoT MQTT Panel 10

4.4 Freeboard.io 10

4.5 Thingsboard.io 10

4.6 Bluetooth Electronics 10

4.7 Virtuino 10

4.8 WiFi Controller ESP8266 10

4.9 BlueTooth Serial Controller 11

4.10 Cayenne 11

4.11 RemoteXY: Arduino control 11

4.12 Blynk – IoT 11

5 Android **12**

5.1 Verze 12

5.2 Vývoj aplikací pro Android 12

5.3 Ovládání vestavěných zařízení 13

5.4 Bezdrátová komunikace 13
6 Návrh aplikace 14
6.1 Požadavky na aplikaci 14
6.1.1 Kompatibilita 14
6.1.2 Použitelnost 14
6.1.3 Univerzálnost 14
6.1.4 Bezpečnost 14
6.1.5 Přizpůsobitelnost 14
6.1.6 Srozumitelnost 14
6.2 Popis návrhu 14
6.3 Ověření návrhu 15
7 Použité technologie 17
7.1 Nástroj pro správu projektu 17
7.2 Technologie pro knihovnu EEDC.js 17
7.3 Technologie pro Android aplikaci 17
7.3.1 Sestavení aplikace 18
8 Architektura aplikace 19
8.1 Základní principy 19
8.2 Clean architecture 21
8.2.1 Doménová vrstva 22
8.2.2 Prezentační vrstva 22
8.2.3 Datová vrstva 24
8.2.4 Modul app 24
8.3 Adresářová struktura 26
9 Implementace knihovny EEDC.js 27
9.1 Řídící soubor 27
9.2 eedc-core.js 28
9.3 API 29
9.3.1 Funkce pro pužití ve skriptech 29
9.3.2 Funkce pro použití v Android rozhraní 30
9.3.3 Události 31
9.4 Použití API 31
10 Realizace Android aplikace 33
10.1 Uživatelské rozhraní 33
10.2 Funkce aplikace 33
10.2.1 Seznam zařízení 34
10.2.2 Přidat zařízení 35

10.2.3 Detail zařízení 36
10.2.4 Upravit zařízení 37
10.2.5 Ovládání zařízení 38
10.3 Aplikační komponenty 38
10.4 Databáze 44
10.5 Řídící soubor 44
10.5.1 Nahrání souboru 45
10.5.2 Zobrazení souboru 45
10.5.3 JavaScript rozhraní 47
10.6 Komunikace se zařízeními 49
10.6.1 Wi-Fi TCP 49
10.6.2 Wi-Fi UDP 50
10.6.3 Bluetooth 51
10.6.4 Bluetooth Low Energy 52
11 Ověření funkčnosti a testování 54
11.1 Způsoby testování 54
11.2 Jednotkové testy 54
11.3 Integrační a UI testy 55
11.4 Ruční testování 57
11.5 Detekce chyb v produkci 57
11.6 Testovací aplikace 58
11.6.1 Wi-Fi 58
11.6.2 Bluetooth 60
11.6.3 Bluetooth Low Energy 62
11.7 Publikace aplikace 63
12 Závěr 64

1 Úvod

Účelem této práce je prozkoumání existujících řešení a problematiky vzdáleného zobrazování dat na mobilních telefonech a návrh vhodného protokolu umožňující vzdálené zobrazování/zadávání dat na obrazovce telefonu vhodný pro vestavěná zařízení a to tak, aby dle navrženého protokolu bylo možné implementovat Android aplikaci komunikující s vestavěným zařízením pomocí Bluetooth nebo Wi-Fi. Výsledkem této práci by tak měla být Android aplikace, pomocí které bude možné vzdáleně komunikovat s jinými (vestavěnými) zařízeními.

V úvodu je čtenář seznámen s problematikou vestavěných zařízení a vestavěných systémů, jejich definicí, historií, rozdělením a příkladů použití.

V další části práce jsou probrány možnosti externí (vzdálené, bezdrátové) komunikace, důvody proč se vůbec externí komunikací zabývat a jsou zde uvedené možné způsoby (a protokoly) externí komunikace. Detailněji jsou rozebrány tři vybrané protokoly (Wi-Fi, Bluetooth a Bluetooth Low Energy), které by měla podporovat výsledná aplikace.

Následuje představení vybraných existujících aplikací pro externí komunikaci s vestavěnými zařízeními, převážně pro operační systém Android a jsou krátce probrány specifikace externí komunikace s vestavěnými zařízeními týkající se mobilních telefonů s operačním systémem Android.

Poté jsou vzneseny požadavky na aplikaci a její hrubý návrh, dle kterých jsou popsané vybrané technologie a architektura aplikace. Zároveň je popsána vlastní realizace aplikace.

V poslední části práce je ověřena funkčnost aplikace, popis jejího testování a tvorba testovacích aplikací pro každý typ z vybraných protokolů (Wi-Fi, Bluetooth a Bluetooth Low Energy).

2 Vestavěné systémy

Tato kapitola je úvodem do problematiky vestavěných zařízení a vestavěných systémů, popisuje, co jsou to vestavěné systémy a kde se můžeme setkat s jejich aplikací. Dále stručně shrnuje historii a rozdělení vestavěných systémů.

2.1 Definice vestavěného systému

Existuje mnoho definicí pro vestavěný systém, jedna z nich může být, že vestavěný systém je kombinace počítačového softwaru a hardwaru, která je buď pevná nebo programovatelná a je určena pro konkrétní funkci nebo skupinu funkcí ve vetším systému [1]. Uživatel si může přizpůsobit zařízení, ale nelze změnit funkčnost přidáním nebo nahrazením softwaru. Vestavěný systém je tedy navržen tak, aby vykonával jen konkrétní úkol, ale s možností různého nastavení.

2.2 Historie

Dle [1] je v historii vestavěných systémů několik důležitých milníků. V roce 1960 byl vestavěný systém poprvé použit pro vývoj naváděcího systému Apollo Charles Stark Draperem na MIT. V roce 1965 vyvinula společnost Autonetics D-17B počítač používaný v systému navádění raket. V roce 1968 byl poprvé použit vestavěný systém ve vozidle. V roce 1971 vyvinula společnost Texas Instruments první mikrokontrolér. V roce 1987 byl firmou Wind River představen první vestavěný operační systém VxWorks. V roce 1996 byl představen operační systém Windows CE od firmy Microsoft. Později v 90. letech se objevuje první vestavěný operační systém postavený na Linuxu. V roce 2013 dosáhl trh vestavěných systémů 140 miliard USD.

2.3 Rozdělení

Dle [2] můžeme vestavěné systém dělit dle dvou kritérií. Zaprvné dle funkčních požadavků a zadruhé dle výkonu mikrokontroléru.

2.3.1 Dle funkčních požadavků

Na základě funkčních požadavků můžeme vestavěné systémy dělit na samostatné vestavěné systémy, vestavěné systémy v reálném čase, síťové vestavěné systémy a mobilní vestavěné systémy.

Samostatné vestavěné systémy nevyžadují hostitelský systém – fungují samostatně. Dle vstupu na vstupních portech udělají požadovaný úkol, nebo výpočet a výsledek poskytují dalším připojeným zařízením, které mohou také ovládat. Příkladem může být mikrovlnná trouba nebo systém pro měření teploty.

Vestavěné systémy v reálném čase, jsou systémy, které musí reagovat na požadavek v daném čase. Dále se mohou dělit na hard systémy, u kterých je zpožděná odpověď neakceptovatelné a soft systémy, u kterých je zpoždění nějakým způsobem tolerované.

Síťové vestavěné systémy jsou zapojeny do sítě, ze které mají přístup ke zdrojům. Síť může být LAN, WAN nebo internet a připojení může být kabelové nebo bezdrátové. Příkladem může být domácí bezpečnostní systém, jehož senzory jsou připojeny v síti a komunikují pomocí TCP/IP protokolu.

Mobilní vestavěné systémy se používají v přenosných zařízeních jako jsou mobilní telefony, fotoaparáty, mp3 přehrávače apod.

2.3.2 Dle výkonu mikrokontroléru

Na základě výkonu mikrokontroléru můžeme vestavěné systémy dělit na malé, střední a sofistikované vestavěné systémy.

Malé vestavěné systémy jsou navržený s jedním 8 nebo 16bitovým mikrokontrolérem. Ten může být dokonce aktivován pouze baterií. Pro vývoj se většinou používá assembler.

Střední vestavěné systémy jsou navrženy s jedním 16 nebo 32bitovým mikrokontrolérem, RISC nebo DSP. Pro vývoj se obvykle používá C, C++ nebo Java.

Sofistikované vestavěné systémy mají oproti předchozím dvěma obrovské jak hardwarové tak softwarové požadavky.

2.4 Příklady použití

S vestavěnými systémy se setkáme takřka v každém odvětví. V robotických vědách to mohou být různá robotická vozidla, drony a průmyslové roboty, v lékařství dialyzační stroj, infuzní čerpadla nebo srdeční monitor. V automobilovém průmyslu kontrola motoru, zapalovací systém, brzdový systém, bezpečnostní systémy, airbagy, zámky dveří nebo parkovací asistent. V počítačových sítích routery, huby, switche. Dále se s nimi setkáme také v bankovnictví, mobilních komunikacích, jaderném průmyslu nebo ve vesmírném průmyslu.

3 Externí komunikace

Tato kapitola nejprve uvádí důvody pro externí (vzdálenou, bezdrátovou) komunikaci s vestavěnými zařízeními, dále představuje různé možnosti externí komunikace a vybrané existující protokoly používající se právě pro vestavěné zařízení.

3.1 Důvody pro externí komunikaci

Důvody pro externí komunikaci jsou zřejmé. Vestavěné zařízení může být zapojeno do celého systému a potom veškerá komunikace zařízení probíhá s tímto systémem (přes jeho vstupy a výstupy). V tomto případě není moc důvodů pro přímou komunikaci s tímto jedním zařízením, která obvykle probíhá na úrovni celého systému. Pokud ale bude zařízení samostatné (viz rozdělení systémů dle funkčních požadavků) nebo budeme na celý systém nahlížet jako na samostatné vestavěné zařízení, může se již vyskytnout potřeba se zařízením komunikovat. Komunikací je myšleno jak nastavování zařízení (zadávání dat), tak čtení dat.

Externí (vzdálená) komunikace nám umožní nastavovat/monitorovat vestavěné zařízení bez nutnosti mít u něho ovládací panel (display, klávesnice), jehož pořizovací náklady (zvláště pokud by u každého zařízení měl být vlastní ovládací panel) by přesahovaly náklady na software, přes který by se mohlo ovládat více zařízení. Další viditelnou výhodou externí komunikace je, že není nutné mít fyzický přístup k zařízení. Ať už při komunikaci na kratší vzdálenost, kdy můžeme ovládat např. venkovní nebo těžko přístupná zařízení, nebo při komunikaci přes internet, kdy můžeme zařízení ovládat prakticky odkudkoliv.

3.2 Způsoby externí komunikace

Je několik možností externí komunikace s vestavěnými systémy. Tato práce je zaměřena na datovou komunikaci, dále uvádí ještě jeden typ komunikace a to vzdálený přenos obrazu.

3.2.1 Datová komunikace

Datovou komunikací je myšlena komunikace, kdy data (formátována dle protokolu) mohou popisovat aktuální stav zařízení, data ze senzorů nebo akce, které má vykonat jedno z komunikujících zařízení.

Pro bezdrátovou datovou komunikaci s vestavěnými systémy se používají tyto protokoly [3]:

- **Wi-Fi**
  Komunikační standard IEEE 802.11, k přenosu dat používá vysokofrekvenční rádiové vlny. Pro propojení klientů a dalších sítí slouží přístupový bod.
- **Wi-Fi direct**
  Wi-Fi direct, původně nazývaný Wi-Fi P2P, je standard Wi-Fi, který umožňuje zařízením se navzájem spojovat bez nutnosti přístupového bodu.
- **Bluetooth**
  Otevřený standard IEEE 802.15.1 pro bezdrátovou komunikaci. Vyznačuje se nízkou cenou, nízkým výkonem a krátkým dosahem. Pro vícebodovou komunikaci využívá topologii *scatternet*, která se skládá ze dvou nebo více piconetů, který obsahuje jedno zařízení typu master a několika zařízení typu slave.
- **Bluetooth Low Energy**
  Je součástí specifikace Bluetooth 4.0, hlavním cílem je úspora energie a podpora zařízení napájených malými bateriemi.
- **Zigbee**
  Bezdrátová technologie, založena na standardu IEEE 802.15.4, navržena pro nízkou spotřebu energie, což umožňuje dlouhou výdrž baterie. Používá se především pro domácí sítě nebo průmyslová zařízení.

- **Z wawe**
  Bezdrátový komunikační protokol používaný především pro domácí automatizaci, který využívá nízkoenergetických rádiových vln.
- **6LowPAN**
  6LowPAN je zkratka pro IPv6 přes bezdrátové osobní sítě s nízkým výkonem. Umožňuje méně výkonným zařízením přenášet informace bezdrátově pomocí internetového protokolu.
- **GPRS/3G/LTE**
  Mobilní telekomunikační technologie umožňující mobilním telefonům (nebo jiným zařízením) bezdrátový přenos dat a připojení k internetu.
- **NFC**
  Zkratka pro Near Field Communication. Jedná se o vysokofrekvenční bezdrátovou technologii krátkého dosahu (cca 4 cm). Používá se např. u bezkontaktních platebních karet nebo při platbě mobilním telefonem.

3.2.2 Vzdálená obrazovka

Při komunikaci pomocí vzdálené obrazovky ovládá uživatel vzdálené zařízení prostřednictvím počítačové sítě přes protokol RDP[1].

3.3 Bezpečnost komunikace

Komunikace mezi zařízeními by měla být zabezpečená tak, aby neautorizovaná třetí strana nemohla zprávy odposlouchávat či falšovat a tak, aby vestavěné zařízení mohlo jednoznačně identifikovat druhé zařízení a zamezit tak jeho neoprávněnému nastavování. Bezpečnost komunikace bezdrátových sítí proto můžeme rozdělit na šifrování a autorizaci.

Šifrování zabezpečuje přenášená data před odposloucháváním či falšováním zpráv. Pro šifrování zpráv je vhodné použít TLS[2], nebo jiný kryptografický protokol.

Pro autorizaci, jednoznačnou identifikaci zařízení, je možné použít např. sériových a výrobních čísel, nebo použít asymetrickou kryptografii na principu veřejného a privátního klíče [6].

Bezpečnost vybraných protokolů je popsána v následující podkapitole.

3.4 Vybrané protokoly

Tato podkapitola popisuje použití vybraných protokolů použitých v této práci pro potřeby komunikace s vestavěnými systémy. Protokoly, které byly vybrány jsou Wi-Fi, Bluetooth a Bluetooth Low Energy a to z důvodu jejich dostupnosti jak v mobilních telefonech tak ve vestavěných zařízeních. Následuje stručný popis vybraných protokolů s jejich výhodami a nevýhodami.

3.4.1 Wi-Fi

Při použití Wi-Fi máme tři možnosti jak uskutečnit komunikaci. První možnost je, že obě komunikující zařízení budou připojené v internetu. Potom stačí aby vestavěné zařízení mělo svojí veřejnou IP adresu, ke které se druhé zařízení (např. mobilní telefon) připojí např. pomocí TCP protokolu.

Druhou možností je, že obě zařízení budou ve stejné síti. Potom již není potřeba veřejné IP adresy, ale stále potřebujeme znát IP adresu zařízení. Komunikace potom probíhá stejně jako v prvním případě.

---

[1] RDP - Remote Desktop Protocol [4]
[2] TLS – Transport Layer Security

Třetí možností je, že vestavěné zařízení vytvoří bezdrátovou ad hoc síť. Potom zařízení vystupuje jako přístupový bod, ke kterému se připojí další zařízení. Komunikace potom probíhá na stejném principu jako v předchozích případech.

Šifrování komunikace u Wi-Fi sítí je zajištěno šifrovacími protokoly (algoritmy) WEP, WPA nebo WPA2. Pro autorizaci je použit protokol IEEE 802.1X, který pro autentizaci uživatele požaduje např. uživatelské jméno a heslo. Pokud se uživatel připojí na přístupový bod má blokovanou veškerou komunikaci kromě protokolu EAP, který zajišťuje autentizaci.

Výhody: spolehlivost, vysoká rychlost

Nevýhody: vyšší spotřeba energie, omezený dosah

Použití: přístup na internet, hotspoty, adhoc sítě

3.4.2 Bluetooth

Otevřený standard pro tvorbu osobních sítí (PAN). Mezi jeho vlastnosti patří nízká cena, nízký výkon a krátký dosah, obvykle asi 10 metrů [3]. Při komunikaci jedno zařízení vystupuje jako server, druhé jako klient. Klient vyhledá server a připojí se k němu pomocí jeho mac adresy.

Bluetooth využívá proces *párování*, díky kterému je vyřešena kontrola zařízení, které se mohou připojit. Spárované zařízení mohou navázat spojení kdykoliv budou v dosahu bez nutnosti zásahu uživatele. Při párování je generován klíč, pomocí kterého je následně zašifrována komunikace.

Výhody: cena, snadná instalace, nižší spotřeba energie (oproti Wi-Fi)

Nevýhody: nižší bezpečnost, omezená vzdálenost

Použití: přenos souborů, nositelná elektronika, zařízení pro záznam dat

3.4.3 Bluetooth Low Energy

Na rozdíl od klasického Bluetooth, Bluetooth Low Energy (BLE) je komunikační standard vhodný pro zařízení, u nichž je požadována malá spotřeba energie. To dovoluje komunikaci zařízením, která mají striktní požadavky na spotřebu energie jako jsou proximity sensory nebo monitory srdeční frekvence a další fitness zařízení. [10]

Při komunikaci jedno zařízení vystupuje jako GATT[3] server a druhé jako GATT klient. Server poskytuje jednu nebo více service. Každá service obsahuje jednu nebo více charakteristik, které obsahují vlastní hodnotu (např. srdeční frekvence) a žádný nebo více deskriptorů, který popisují hodnotu dané charakteristiky. Klient potom čte data z vybrané charakteristiky.

Výhody: nízká spotřeba energie

Nevýhody: nižší bezpečnost, omezená vzdálenost

Použití: proximity sensory, fitness zařízení, nositelná elektronika

[3] GATT – Generic Attribute Profile [10]

4 Existující řešení

Tato kapitovala představuje vybrané existující aplikace pro externí komunikaci s vestavěnými zařízeními.

4.1 IoT MQTT Dashboard

Volně dostupná aplikace pro zařízení s operačním systémem Android pro komunikaci se zařízeními přes protokol MQTT. Ke každému zařízení umožňuje vytvořit seznam widgetů pro každý senzor (MQTT topic) bez možnosti úpravy jejich vzhledu. Komunikace probíhá přes internet (TCP/IP).
https://play.google.com/store/apps/details?id=com.thn.iotmqttdashboard

4.2 MQTT Dash

Volně dostupná aplikace pro Android podobná předchozí, umožňuje úpravu vzhledu jednotlivých widgetů na úrovni volby ikon, prefixů a postfixů.
https://play.google.com/store/apps/details?id=net.routix.mqttdash

4.3 IoT MQTT Panel

Volně dostupná aplikace pro Android podobné předchozím. Kromě TCP podporuje připojení přes protokol websocket. Nabízí možnost importovat/exportovat konfiguraci.
https://play.google.com/store/apps/details?id=snr.lab.iotmqttpanel.prod

4.4 Freeboard.io

Webová platforma, která umožňuje vytváření panelů pro ovládání zařízení komunikujících přes protokol HTTP. Nabízí pokročilejší funkce jako jsou alarmy a automatické spouštění akcí. Míra kustomizace vzhledu je omezená. K dispozici je 30 denní zkušební verze zdarma.
https://thingsboard.io

4.5 Thingsboard.io

Webová aplikace podobná předchozí, ale navíc podporuje i protokoly MQTT a CoAP. K dispozici je verze zdarma.
https://freeboard.io

4.6 Bluetooth Electronics

Volně dostupná aplikace pro Android pro komunikaci se zařízeními přes Bluetooth nebo USB. Nabízí spoustu widgetů a je možné upravovat jejich vzhled a rozmístění. U každého widgetu se nastavuje, jaké příkazy se mají posílat do zařízení.
https://play.google.com/store/apps/details?id=com.keuwl.arduinobluetooth

4.7 Virtuino

Aplikace pro Android podobná předchozí, ale umožňuje komunikovat přes Bluetooth nebo Wi-Fi.
https://play.google.com/store/apps/details?id=com.virtuino_automations.virtuino

4.8 WiFi Controller ESP8266

Aplikace pro Android podobná té předchozí s podporou komunikace pouze přes Wi-Fi. Bez možnosti upravovat vzhled a rozmístění widgetů.
https://play.google.com/store/apps/details?id=com.mightyit.gops.wificontroller

4.9 BlueTooth Serial Controller

Aplikace pro Android pro komunikaci se zařízeními přes Bluetooth. Bez možnosti úpravy vzhledu widgetů, nabízí pouze tlačítka a každé posílá nějaký příkaz.
https://play.google.com/store/apps/details?id=nextprototypes.BTSerialController

4.10 Cayenne

Aplikace pro Android komunikující se zařízeními pomocí zasílání příkazů přes Wi-fi, dále přes MQTT protokol a přes síť LoRa. Nabízí úpravu vzhledu widgetů a nastavování automatických akcí.
https://play.google.com/store/apps/details?id=com.mydevices.cayenne

4.11 RemoteXY: Arduino control

Aplikace pro Android speciálně pro ovládání mikrokontrolerů Arduinu. Komunikuje přes Bluetooth, Wi-Fi, internet a USB. Nabízí několik upravitelných widgetů.
https://play.google.com/store/apps/details?id=com.shevauto.remotexy.free

4.12 Blynk – IoT

Android aplikace od firmy Blynk, která nabízí kompletní řešení pro technologie internetu věcí. Tato aplikace je speciálně pro mikrokontrolery Arduino, Raspberry Pi a další, na kterých je nutné implementovat dodávanou knihovnu. Nabízí spoustu widgetů s možností úprav.
https://play.google.com/store/apps/details?id=cc.blynk

5 Android

Tato kapitola popisuje bezdrátovou komunikaci s vestavěnými zařízeními pomocí chytrých telefonů v prostředí operačního systému Android. Dále uvádí změny v komunikačních protokolech dle jednotlivých verzí [7].

5.1 Verze

Zde je seznam vybraných verzí operačního systému Android, ve kterých došlo ke změnám v komunikačních protokolech Bluetooth a Wi-Fi.

- **Android 1.0 (API 1)**
  Podpora Wi-Fi a Bluetooth.
- **Android 1.5 (API 3)**
  Automatické párování Bluetooth.
- **Android 2.0 (API 5)**
  Podpora Bluetooth 2.1
- **Android 2.2 (API 8)**
  Funkce Wi-Fi hotspot
- **Android 4.0 (API 14)**
  Wi-Fi Direct (Wi-Fi P2P)
- **Android 4.3 (API 18)**
  Funkce pro Wi-Fi skenování
  Podpora Bluetooth 4.0 a Bluetooth Low Energy
- **Android 8.1 (API 27)**
  Podpora Bluetooth 5

5.2 Vývoj aplikací pro Android

Aplikace pro zařízení s operačním systémem Android lze vyvíjet několika způsoby.

Prvním způsobem jsou nativní aplikace, které jsou vytvořeny pro specifickou platformu a jsou napsány v programovacím jazyce, který daná platforma podporuje. Pro nativní Android aplikace to je Java a Kotlin, pro nativní iOS aplikace to je Swift a Objective-C. Pro vývojáře nativních aplikací jsou k dispozici vývojové nástroje a SDK[4]. Výhodami je především výkon nativních aplikací, který se s dalšími způsoby nedá srovnávat, dále jednotnost grafického rozhraní a tím pádem intuitivnost takového rozhraní pro uživatele dané platformy. Nativní vývoj dovoluje vývojářům přístup ke všem funkcím daného operačního systému. Nevýhodami nativních aplikací je, že pro každou platformu se musí psát celá aplikace zvlášť a proto je nativní vývoj dražší, pokud cílíme na více operačních systémů. Naše aplikace bude vyvíjena tímto způsobem, protože potřebujeme mít zajištěn přístup k hardwaru pro bezdrátovou komunikaci přes Wi-Fi a Bluetooth, a požadujeme vysoký výkon aplikace.

Druhým způsobem jsou webové aplikace, které jsou načtené buď v nějakém externím webovém prohlížeči nainstalovaném v zařízení. Nebo jsou obalené aplikací vytvořenou jedním z dalších způsobů a webová aplikace je načtena přímo v nich pomocí webview. Webové aplikace se typicky vytvářejí pomocí HTML, CSS, JavaScriptu, PHP a dalších webových technologií. Výhodami webových aplikací je rychlost vývoje, nezávislost na platformě (Android, iOS, Windows,..) a hardwaru (mobil, tablet, pc) a to, že změny v takovýchto aplikacích se projeví ihned bez nutnosti aktualizovat aplikaci z marketu (Google Play, App Store). Nevýhodami je nutnost připojení k internetu, webové aplikace

[4] SDK – Software Development Kit, je sada vývojových nástrojů

jsou pomalejší než nativní aplikace a nemají přístup k hardwaru zařízení. Bezdrátová komunikace s jinými zařízení pomocí Wi-Fi nebo Bluetooth by pomocí webových aplikací byla proto nemožná.

Posledním způsobem jsou nástroje, které nám umožní z jednoho kódu zkompilovat aplikace pro více platforem. Známými zástupci těchto nástrojů jsou *Xamarin* a *React Native*. Takovýto vývoj umožňuje jeden kód psaný v případě Xamarinu v jazyce C# v případě React Native v jazyce JavaScript, zkompilovat do nativní aplikace pro Android i iOS. (V případě Xamarinu i pro Windows Mobile). Výhodami těchto nástrojů je to, že máme společný kód pro obě platformy a tím pádem je nižší cena za vývoj. Nevýhodami je, že se jedná o nadstavbu nad nativním kódem, která může obsahovat chyby a na každé platformě se může chovat trochu jinak a to, že stále se musí nějaké části psát v nativním kódu, a potřebujete tak vývojáře minimálně pro 3 programovací jazyky.

5.3 Ovládání vestavěných zařízení

Chytré mobilní telefony, jako je iPhone a Android, představují alternativní způsob komunikace s vestavěnými systémy. Zařízení, která vyžadují interakci na místě, mají svá specifická hardwarová rozhraní - jejich přepínače, tlačítka, displaye a dotykové panely. Zařízení, která lze ovládat na dálku, vyžadují investování do účelového hardwarového rozhraní (dálkové ovládání), používání programu v počítači nebo připojení k vestavěnému webovému rozhraní. S dostupností chytrých telefonů lze vlastní hardware nahradit mobilní aplikací pro interaktivní obsluhu zařízení [14].

5.4 Bezdrátová komunikace

Android SDK nabízí velké množství nástrojů pro bezdrátovou komunikaci pro zmíněné vybrané protokoly (Wi-Fi, Bluetooth a Bluetooth Low Energy) [5]. Základem je balíček *java.net*, který poskytuje třídy pro implementaci síťových aplikací. Dále využijeme nástroje z balíčků *android.net.wifi, android.bluetooth* a *android.bluetooth.le*.

6 Návrh aplikace

V této části jsou uvedeny požadavky na aplikaci, její hrubý návrh a jeho ověření. V další části je pojmem zařízení myšleno vestavěné zařízení (či vestavěný systém) a pojmem aplikace je myšlena vytvářená Android aplikace, která bude s tímto zařízením komunikovat.

6.1 Požadavky na aplikaci

Před návrhem a vývojem aplikace byly určeny cíle a požadavky, které by aplikace měla splňovat. Mezi tyto cíle patří kompatibilita, použitelnost, univerzálnost, bezpečnost, přizpůsobitelnost a srozumitelnost.

6.1.1 Kompatibilita

Kompatibilitou aplikace je myšlena kompatibilita s různými verzemi Androidu. V současné době je k dispozici Android API[5] 29 (Android 10) a je jasné, že s každou novou verzí nelze zajistit zpětnou kompatibilitu se staršími verzemi. Proto byla pro aplikaci zvolena minimální podporovaná verze API 19, která (nyní v době vývoje) pokryje 96,2 % všech zařízení [8] a oproti starším verzím obsahuje webview (které bude základem vytvořené aplikace) založené na chromiu[6]. Díky tomu obsahuje aktualizovanou verzi enginu JavaScript V8 a podporu moderních webových standardů dříve chybějících ve starších verzích [9].

6.1.2 Použitelnost

Aplikace by měla být vytvořena tak, aby její uživatelské rozhraní bylo uživatelsky přívětivé a byla v něm snadná orientace. Proto by měly být použity systémové komponenty, které uživatel zná z prostředí Android a neměl by mít problém je používat.

6.1.3 Univerzálnost

Aplikace by měla být univerzální a nezávislá na použitém protokolu, měla by umožňovat komunikaci jak s jednoduchými zařízení ovládanými např. sekvencí bitů, tak s výkonnějšími zařízeními se sofistikovanějšími protokoly jako je např. protokol MQTT.

6.1.4 Bezpečnost

Měla by být možnost zabezpečit veškerou komunikaci mezi zařízeními. Dále by měla aplikace bezpečně ukládat a zacházet s daty, které do ní uživatel vkládá.

6.1.5 Přizpůsobitelnost

Grafické rozhraní aplikace by mělo být snadno přizpůsobitelné jakýmkoliv požadavkům.

6.1.6 Srozumitelnost

Pro použití aplikace vývojáři by měla být dostupná srozumitelné dokumentace.

6.2 Popis návrhu

Android aplikace bude podporovat komunikaci přes Wi-Fi (v případě veřejné IP adresy i pomocí mobilního internetu), Bluetooth a Bluetooth Low Energy. V nastavení aplikace se vybere příslušný protokol (z těchto tří) a provede se připojení (vyhledání zařízení nebo zadání adresy).

Dále záleží na tom, zda bude zařízení dostatečně výkonné na to, aby mohlo poslat tzv. řídící soubor (control file). Řídící soubor bude zip soubor, který bude obsahovat soubor

[5] API – aplikační programové rozhraní (Application Programming Interface)
[6] Chromium – open source webový prohlížeč https://www.chromium.org

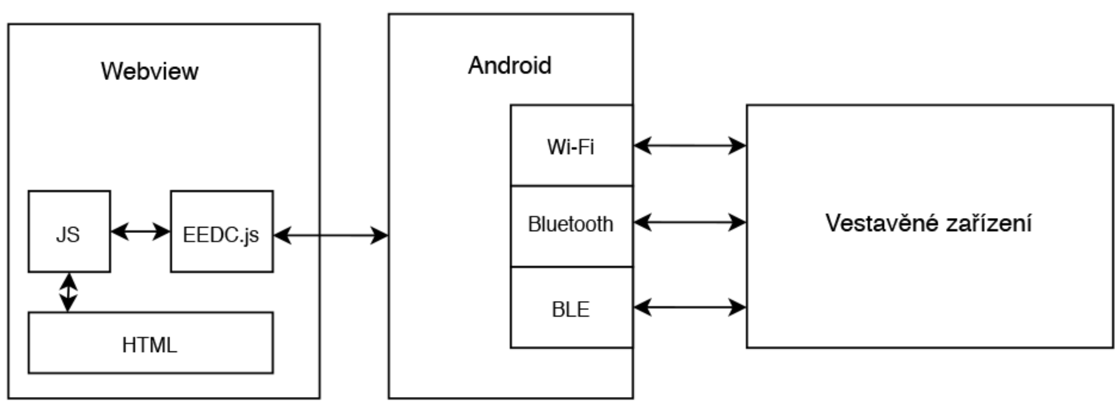
index.html a další klasické webové soubory jako jsou JavaScript soubory a CSS soubory. Pokud bude zařízení dostatečně výkonné, ihned po připojení poskytne aplikaci řídící soubor obsahují JavaScript viz dále.

Pokud zařízení nebude dostatečně výkonné, řídící soubor nahraje uživatel ručně při nastavování připojení k zařízení.

K aplikaci bude knihovna pro JavaScript (*EEDC.js* na obrázku 1), která bude obsahovat funkce pro samotnou komunikaci se zařízením. Například funkce pro poslání pole bytů, řetězce apod. a funkce informující o přijatých datech ze zařízení. Dále bude obsahovat funkce pro informování o stavu připojení a funkce pro manipulaci vzhledu aplikace (např. fullscreen mód). Samotné vykreslování dat a čtení vstupu od uživatele (přes webview) bude mít na starost skript v JavaScriptu, který bude součástí poskytnutého HTML souboru a který bude využívat zmíněnou knihovnu.

Pokud například uživatel klikne na tlačítko a ve skriptu bude obsluha odesílat textový řetězec, zavolá se funkce z JavaScript knihovny (např. *send(string)*). Knihovna zajistí komunikaci s webview a nad webview bude implementované rozhraní, které provede odeslání řetězce do zařízení např. přes Bluetooth.

Díky tomuto způsobu bude aplikace univerzální (nezávislá na protokolu) a vlastní formát zpráv (případně i jejich šifrování), kterému bude rozumět zařízení, bude řešit skript v JavaScriptu, který bude dodán speciálně ke každému zařízení.



Obrázek 1 - Návrh aplikace

6.3 Ověření návrhu

Pro ověření návrhu, před začátkem samotné implementace, byl vytvořen jednoduchý prototyp, který testuje funkčnost mostu mezi webovým prohlížečem a prostředím Androidu. Pro jeho potřeby byla vytvořena jednoduchá HTML stránka, která obsahuje:

- **tlačítko „Send action 1“**
  Pošle řetězec „1“ z JavaScriptu do prostředí Android, který zobrazí dialog
- **textové pole s tlačítkem „Send input“**
  Provede stejnou akci jako předchozí tlačítko, ale pošle obsah textového pole.
- **tlačítko „Receive now“**
  Vyžádá si data od prostředí Android a zobrazí je v poli pod ním. Android vždy odesílá řetězec obsahují aktuální datum a čas.

Tato webová stránka je zobrazena ve webview, pod kterým se ještě nachází nativní tlačítko „Send“, které pošle data JavaScriptu (opět aktuální datum a čas), který je zobrazí znovu v posledním poli.

Tímto prototypem (jehož výsledná podoba je vidět na obrázku 2) byla ověřena komunikace mezi funkcemi JavaScriptu a prostředím Android a nic nebrání vlastní implementaci aplikace.

Obrázek 2- Prototyp

7 Použité technologie

V této kapitole je popis použitých technologií, jak na straně webové části (knihovna *EEDC.js* a vlastní uživatelské HTML a JavaScript soubory) tak na straně samotné Android aplikace.

7.1 Nástroj pro správu projektu

Pro verzování projektu je použit GIT[7], pro správu git repositáře je použita webová aplikace GitLab[8], která nabízí spoustu dalších funkcí pro správu projektu. Za zmínku stojí přehledné procházení změn, vedení úkolů, chyb a dokumentace.

7.2 Technologie pro knihovnu EEDC.js

Knihovna *EEDC.js* bude napsána v JavaScriptu, konkrétně v ES6[9], který obsahuje Typed Arrays API pro práci s binárními daty. Toto API bude využito pro manipulaci s přenášenými daty mezi vestavěným zařízením a vlastním uživatelským JavaScriptem ve formě posloupnosti bytů.

Při tvorbě řídících souborů budou využity klasické webové technologie. Hlavním souborem, který se bude načítat do webview bude HTML soubor *index.html*. Ostatní technologie závisí pouze na preferencích uživatele (tvůrce řídícího souboru) a možnostech webview (respektive možnostech všech moderních prohlížečů). Uživatel může například vkládat vlastní kaskádové styly, obrázky, atd.

Při vývoji knihovny *EEDC.js* bude využito vývojové prostředí *PhpStorm* (verze 2017.2) od společnosti JetBrains. Toto vývojové prostředí je určeno převážně pro tvorbu webových aplikací v jazyce PHP, ale obsahuje podporu i pro další webové soubory jako jsou (pro nás potřebné) HTML, JavaScript a CSS.

7.3 Technologie pro Android aplikaci

Pro nativní vývoj Android aplikací je potřeba Android SDK. Jednotlivé verze a potřebné části Android SDK, stáhneme pomocí aplikace Android SDK Manager, která je součástí vývojové prostředí *Android Studio* (použita verze 3.6 RC 2), od společnosti Google.

Při vývoji bude také použit *Android Jetpack*[10], což je sada knihoven, nástrojů a návodů, které pomáhají vývojářům snadněji psát vysoce kvalitní aplikace. Android Jetpack se skládá ze 4 základních částí a to Foundation, Architecture, Behaviour a UI. Při vývoji naší aplikace využijeme například:

- **Android KTX** z Foundation pro psaní stručnějšího kódu v Kotlinu
- **AppComapt** z Foundation pro podporu nových standardů grafického uživatelského rozhraní u starších verzí Androidu.
- **Test** z Foundation pro psaní jednotkových testů a testů uživatelského rozhraní
- **Data Binding** z Architecture deklarativní propojení pozorovatelných dat k prvkům v uživatelském rozhraní
- **Lifecycle** z Architecture pro správu životního cyklu aktivit a fragmentů
- **LiveData** z Architecture pro notifikaci rozhraní při změně dat
- **Room** z Architecture pro plynulý přístup k SQLite databázi

[7] GIT - https://www.git-scm.com/
[8] GitLab - https://about.gitlab.com/
[9] ES6 – ECMAScript 6
[10] Android Jetpack - https://developer.android.com/jetpack

- **ViewModel** z Architecture pro správu dat souvisejícím s uživatelským rozhraním způsobem zohledňující životní cyklus aplikace (aktivity, fragmentu).
- **Preferences** z Behaviour pro tvorbu obrazovky nastavení
- **Fragment** z UI jako základní jednotku uživatelského rozhraní
- **Layout** z UI pro rozvržení uživatelského rozhraní
- **WebView** z UI pro zobrazování webových aplikací jako součást aplikace

Z kapitoly 5.2 víme, že nativní aplikace pro Android lze psát buď v programovacím jazyce Java, nebo Kotlin. Aplikace bude napsána v jazyce Kotlin a to z důvodu oblíbenosti a zkušenosti autora s tímto jazykem.

Pro sestavování aplikace bude použit nástroj Gradle (verze 5.6.4), který má plnou podporu v Android Studiu. Gradle nám umožní rozdělit aplikaci na samostatné moduly, které pak propojíme pomocí jasně daných závislostí s jasně definovanými pravidly, který modul má přístup k funkcím jiného modulu. Více o modulech v dalších kapitolách.

Za zmínku v této kapitole ještě stojí framework Koin[11], což je framework pro vkládání závislostí (dependency injection) pro Kotlin. Všechny další použité nástroje a knihovny jsou uvedeny v *build.gradle* souborech.

7.3.1 Sestavení aplikace

Aplikaci sestavíme pomocí gradle tasku *assembleRelease*, který vytvoří instalovatelný soubor apk ve složce *app/build/outputs/apk/release*. (Toto apk je nepodepsané a nelze publikovat v Google Play.)

[11] Koin - https://insert-koin.io/

8 Architektura aplikace

V této kapitole je popsána architektura aplikace. Nejprve jsou rozebrány základní principy ve vývoji Android aplikací z pohledu její architektury, poté je detailněji popsána implementovaná architektura a rozdělení adresářové struktury celého projektu.

8.1 Základní principy

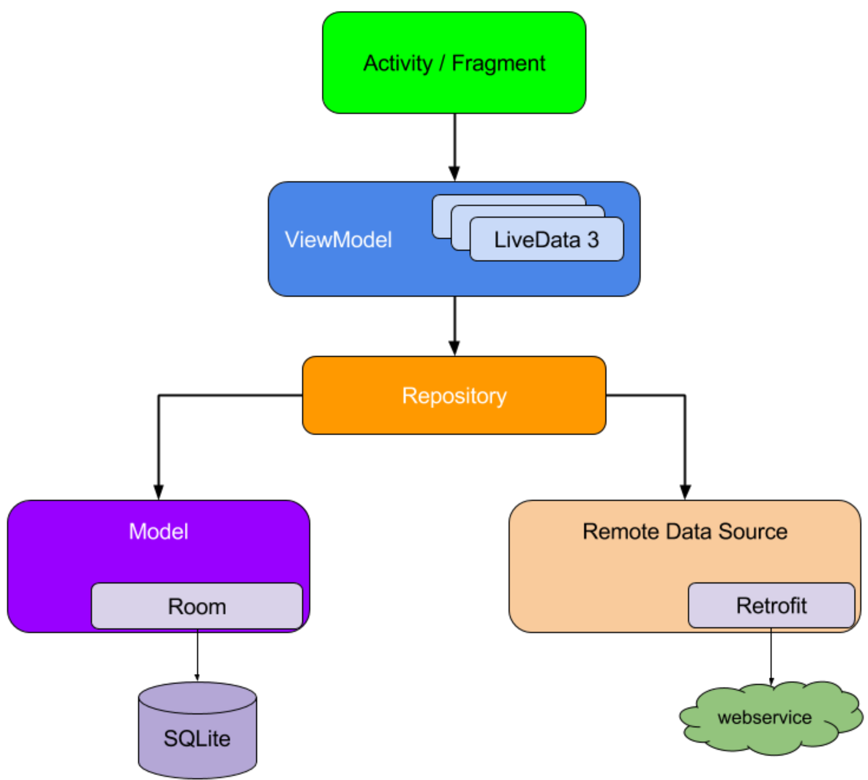
Android aplikace jsou komplexní struktury, které se skládají z několika základních aplikačních komponent. Mezi tyto komponenty patří aplikace (Application), aktivity (Activities), fragmenty (Fragments), služby (Services) poskytovatelé obsahu (Content providers) a Broadcast receivery. Tyto komponenty jsou deklarovány v tzv. aplikačním manifestu, souboru *AndroidManifest.xml*, který musí obsahovat každý Android modul.

Mobilní zařízení mají omezené zdroje, takže operační systém může proces naší aplikace ukončit, aby udělal místo pro jiný. Například pokud při používání naší aplikace přijde příchozí hovor, bude aplikace přerušena a po vyřízení hovoru opět obnovena. Z důvodu takového prostředí operačního systému Android, je možné, že aplikační komponenty budou spouštěny individuálně a bez určeného pořadí a systém nebo uživatel je může kdykoliv ukončit [11]. Protože tyto akce nejsou pod naší kontrolou, neměli bychom ukládat žádná aplikační data, nebo stavy v aplikačních komponentách a naše aplikační komponenty by na sobě neměli být závislé.

Nejdůležitějším principem, který je potřeba dodržovat je princip oddělení zodpovědností [11]. Tento princip nám říká, že různé části aplikace by se z hlediska funkcionality měli co nejméně překrývat. Je běžnou chybou napsat celý kód do aktivity nebo fragmentu [11]. Tyto třídy jsou založené na uživatelském rozhraní a měli by obsahovat pouze logiku, která zpracovává interakce uživatelského rozhraní a operačního systému. Je důležité mít na paměti, že implementace aktivit a fragmentů jsou již hotové a my využíváme pouze jejich služeb. Operační systém je může kdykoli zničit na základě interakce s uživatelem nebo kvůli systémovým podmínkám jako je nízká paměť.

Dalším důležitým principem je, že bychom měli řídit uživatelské rozhraní z modelu [11]. Modely jsou komponenty, které jsou zodpovědné za zpracování dat pro aplikaci. Jsou nezávislé na objektech View uživatelského rozhraní a aplikačních komponentách aplikace, takže nejsou ovlivněny životním cyklem aplikace a s tím souvisejícími problémy.

Na obrázku 3, je vidět diagram doporučované architektury dle [11]. Diagram ukazuje, jak by měli všechny části architektury vzájemně spolupracovat. Jak je vidět z diagramu, komponenta typu aktivita nebo fragment závisí pouze na ViewModelu. ViewModel je objekt, který poskytuje data dané komponentě uživatelského rozhraní jako je aktivita nebo fragment. ViewModel může například pomocí dalších komponent načíst seznam položek z databáze, které potom aktivita/fragment zobrazí v uživatelském rozhraní. ViewModel tak neví o komponentách uživatelského rozhraní (aktivitě a fragmentu). Na diagramu obsahuje ViewModel LiveData. LiveData jsou tzv. pozorovatelné objekty, obsahující jiné objekty. Další komponenty mohou tyto LiveData pozorovat (a případně měnit). Ukázka pozorování LiveData objektu je ve zdrojovém kódu 1. Tento kód bude vykonán ve fragmentu, kde se pozorují LiveData *devices* z ViewModelu. Při každém obnovení obsahu těchto dat je zavolán předaný observer, který (v tomto případě) obnoví data v nějakém adaptéru.



Obrázek 3 - Finalní Architektura [11]

```
viewModel.devices.observe(viewLifecycleOwner, Observer { adapter.bind(it) })
```

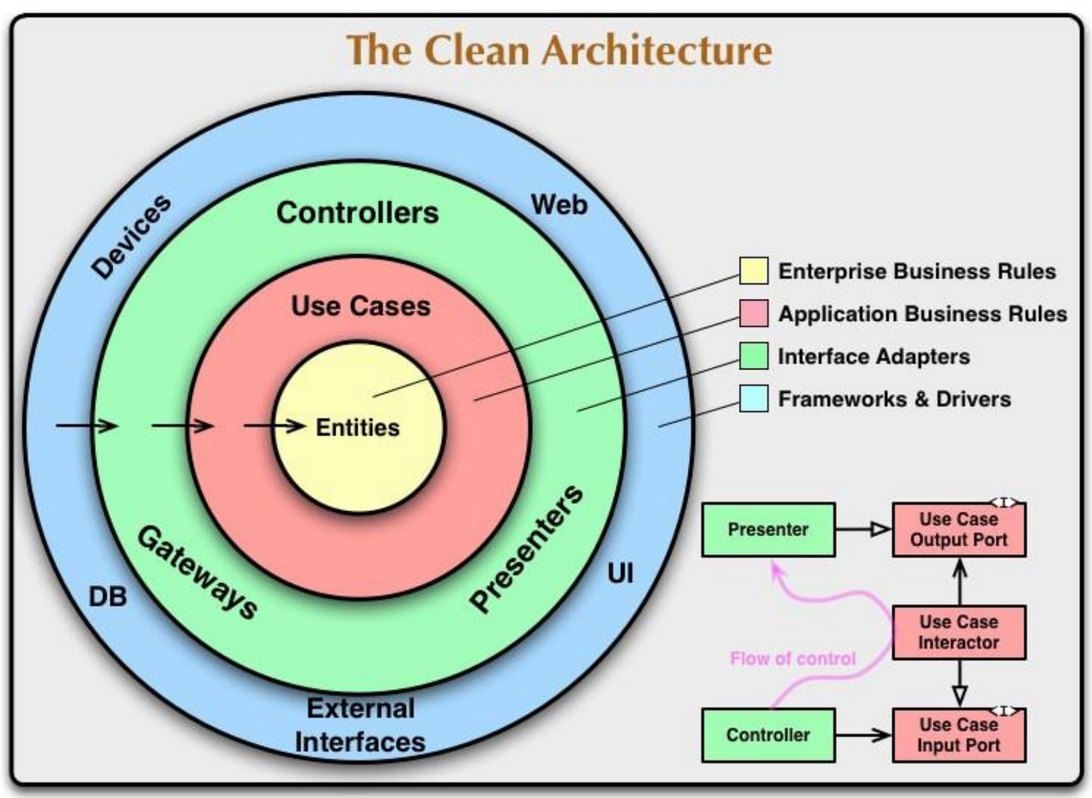
Zdrojový kód 1 - LiveData observer
(presentation/features/device/view/DevicesListFragment.kt)

Další částí v diagramu je repository. Repository se starají o operace s daty. Poskytují čisté API, tak aby jiné komponenty aplikace mohly snadno získat data. Repository ví, odkud a jak mají data získat, například z databáze. K tomu ovšem repository potřebuje model který bude zajišťovat komunikaci s databází viz model s objektem Room na diagramu. Repository by mohla jednoduše vytvořit daný model, ale k tomu potřebuje znát všechny další závislosti daného modelu a repository pravděpodobně nebude jedinou třídou, která daný model potřebuje. Tato situaci by vyžadovala duplicitní kód ve všech modelech, které by vytvářely instanci daného modelu. Tento problém lze vyřešit metodou vkládání závislostí (dependency injection), která dovolí třídám definovat její závislosti bez nustnosti vytvářet jejich instance. Jak již bylo zmíněno v kapitole 7.3, v projektu bude použit framework Koin, což je framework pro vkládání závislostí v Kotlinu.

8.2 Clean architecture

Existuje spousta pohledů na architekturu mobilních aplikací. Existují architektury jako MVC (model-view-controller), MVP (model-view-presenter) nebo MVVM (model-view-viewmodel). V zásadě všechny pracují na stejném principu rozdělení aplikace do 3 vrstev, kde jedna vrstva obsahuje aplikační (business) logiku, druhá vrstva obsahuje uživatelské rozhraní a třetí vrstva funguje jako prostředník mezi těmito dvěma vrstvami (a případně i uživatelem).

Dalším přístupem k architektuře (nejen) mobilních aplikací je tzv. Clean architecture, kterou navrhl v roce 2012 Rober C. Martin (Uncl Bob) a publikoval na svém blogu *The Clean Code Blog* [12] a která bude použita pro implementaci naší aplikace.



Obrázek 4 - The Clean Architecture

Základem této architektury je rozdělení softwaru do vrstev viz 4 vrstvy na obrázku 4. Důvody proč použít tuto architekturu jsou:

- Rozdělení kódu do vrstev s jasně danými závislostmi usnadní pozdější úpravy.
- Kód je více oddělený a snadněji testovatelný.
- Snadný přechod na jinou platformu, kde vrstva s aplikační logikou zůstane beze změny.

Naše aplikace bude rozdělena do 3 základních vrstev a to vrstvy doménové (která v sobě zahrnuje 2 vnitřní vrstvy z obrázku 4), prezentační a datové. Pro dodržení definovaných závislostí vrstev využijeme toho, že používáme sestavovací nástroj Gradle a každá vrstva bude vystupovat jako samostatný gradle modul. Navíc přidáme ještě modul app viz dále.

8.2.1 Doménová vrstva

Doménová vrstva obsahuje tzv. business nebo aplikační logiku a není závislá na žádných jiných vrstvách. Pokud se koukneme na obrázek 4 se schématem Clean Architektury, naše doménová vrstva bude obsahovat entity a usecasy. Dále bude definovat rozhraní adaptérů, jejichž implementace bude ale již v jiných vrstvách. Entity jsou v doménové vrstvě typicky datové třídy (data class) s definovanými atributy dané entity.

Usecasy jsou tzv. spouštěče logiky aplikace. Každá funkce aplikace může mít svůj usecase. Každý usecase má definovaný vstup a výstup a rozšiřuje abstraktní třídu UseCase viz implementace v kapitole 10.3. Ve zdrojovém kódu 2 je vidět implementace třídy *GetDeviceByIdUseCase*. Vstupem do tohoto usecasu je id zařízení (Int) a výstupem je entita zařízení (DeviceDO). Tento usecase má závislost na repository *DevicesRepository,* pomocí které získá požadovanou instanci *DeviceDO*. *DevicesRepository* je rozhraní definující funkce pro práci (jak název napovídá) se zařízeními. O implementaci tohoto rozhraní ale doménová vrstva nic neví, ta se nachází (v tomto případě) v datové vrstvě (kde má repository přístup k databázi).

```
class GetDeviceByIdUseCase(private val devicesRepository: DevicesRepository)
: UseCase<DeviceDO?, Int>() {

    /**
     * Execute usecase data flow using coroutines
     *
     * @param params params
     * @return result
     */
    override suspend fun executeSync(params: Int): DeviceDO? {
        return devicesRepository.getDeviceById(params)
    }

}
```

Zdrojový kód 2 - Ukázkový UseCase
(domain/features/device/usecase/DevicesListFragment.kt)

8.2.2 Prezentační vrstva

Prezentační vrstva obsahuje implementaci grafického uživatelského rozhraní aplikace a je závislá na doménové vrstvě. Tato závislost je definována v souboru *build.gradle* v modulu *presentation* viz zdrojový kód 3. Díky tomu má modul *presentation* přístup ke všemu z modulu *domain*.

```
dependencies {
    ...
    implementation project(':domain')
    ...
}
```

Zdrojový kód 3 - Presentation závislosti
(presentation/build.gradle)

Prezentační vrstva obsahuje všechny aplikační komponenty uživatelského rozhraní, tedy všechny aktivity a všechny fragmenty. Tím pádem obsahuje také všechny definice uživatelského rozhraní tzv. layout xml soubory. Pokud chce aktivita nebo fragment spouštět

usecasy z doménové vrstvy musí tak učinit přes ViewModel. ViewModely jsou tedy také součástí prezentační vrstvy a mohou být sdíleny napříč jednotlivými aktivitami a fragmenty. Pokud například chceme uživateli ve fragmentu zobrazit detail zařízení dle id zařízení budeme potřebovat výše zmíněný usecase *GetDeviceByIdUseCase*.

Nyní narazíme na problém, jak ve ViewModelu vytvořit daný usecase, když nemáme přístup k datové vrstvě a tedy k implementaci rozhraní *DevicesRepository*. Tento problém vyřešíme elegantně pomocí vkládání závislostí a modulu app (viz dále), který má závislost na všech vrstvách aplikace. Námi používáný framework pro vkládání závislostí – Koin, definuje tzv. moduly (dále Koin moduly). Proto každá naše vrstva (=gradle modul) bude definovat i Koin modul. Námi vyžadovaný usecase *GetDeviceByIdUseCase* proto vložíme do ViewModelu pomocí Koinu viz zdrojový kód 4.

```
private val getDeviceByIdUseCase by di<GetDeviceByIdUseCase>()
```

Zdrojový kód 4 - Vložení usecasu do ViewModelu
(presentation/features/control/viewmodel/ControlDeviceViewModel.kt)

Při vytváření instance pro třídu *GetDeviceByIdUseCase* projde Koin všechny Koin moduly a pokud najde definici pro tuto třídu, vytvoří ji dle dané definice. Existují dva způsoby definice pomocí klíčových slov *factory* a *single*. Factory vytvoří vždy novou instanci, single vytvoří pouze jednu instanci, kterou použije na všech místech, tzv. singleton. Ve zdrojovém kódu 5, je vidět část Koin modulu pro doménovou vrstvu, která definuje, že pro vytváření třídy *GetDeviceByIdUseCase* se použije způsob factory. Místo předání instance třídy implementující rozhraní *DevicesRepository* je použit příkaz z Koin frameworku - *get()*. Tento příkaz říká frameworku, že definice pro vytvoření této instance je uvedena jinde (i v jiném Koin modulu). Jinou možnost tady ani nemáme, protože jak víme, doménová vrstva nemá přístup k vrstvě datové, kde je toto rozhraní implementované. Koin tedy začne hledat definici pro *DevicesRepository*, které je v Koin modulu datové vrstvy viz zdrojový kód 6.

```
val domainModule = module {
    ...
    factory {
        GetDeviceByIdUseCase(
                devicesRepository = get()
        )
    }
    ...
}
```

Zdrojový kód 5 - Domain Koin modul
(domain/core/di/DomainModule.kt)

```
val dataModule = module {
    ...
    single<cz.martinforejt.eedc.domain...DevicesRepository> {
        DevicesRepository(
                database = get()
        )
    }
    ...
}
```

Zdrojový kód 6 - Data Koin modul (data/core/di/DataModule.kt)

8.2.3 Datová vrstva

Datová vrstva obsahuje implementaci rozhraní doménové vrstvy a poskytuje tak data jak doménové, tak prezentační vrstvě. Má, stejně jako prezentační vrstva, závislost pouze na vrstvě doménové. Datová vrstva má na starosti komunikaci se zdrojem dat, ať už se jedná o databázi, soubor, nebo v našem případě zásadní komunikaci s dalšími zařízeními.

Prvním úkolem datové vrstvy tedy bude komunikace s databází. K tomu bude sloužit jeden model (viz model na obrázku 3), který budou moci využívat jednotlivé repository. Jak již bylo uvedeno v kapitole 7.3, pro přístup k sqlite databázi bude použit nástroj Room z Android Jetpack. Úkolem datové vrstvy je definice entit, které budou v databázi ukládány a jejich mapování na entity z doménové vrstvy. Pokud tedy bude potřeba uložit do databáze nové zařízení (DeviceDO), nejprve se provede mapování na datovou entitu (DeviceEntity), která se uloží do databáze. Mapování je potřeba z důvodu přizpůsobení datových typů pro účely ukládání v databázi.

Dalším úkolem datové vrstvy bude všechna práce se soubory. Ať už se jedné o načítání našich řídících souborů (viz návrh aplikace) z lokální úložiště, vytváření a sdílení obrázků s QR kódy, nebo stahování souborů z internetu.

Dále bude tato vrstva obalovat komunikaci s *SharedPreferences*, které slouží k ukládání malých dat typu klíč-hodnota. Pomocí tohoto úložiště může být uloženo například nastavení aplikace apod.

Posledním a hlavním úkolem datové vrstvy bude komunikace s vestavěnými zařízeními. Budou zde tedy jednak funkce pro připojení zařízení přes Wi-Fi, Bluetooth a Bluetooth Low Energy, funkce pro čtení a zápis dat do těchto zařízení a také další pomocné funkce jako například zapnutí/vypnutí Bluetooth, vyhledávání zařízení, zjištění stavu sítě atd.

8.2.4 Modul app

Modul app bude spojovat předešlé tři vrstvy dohromady (má tedy závislost na všech třech modulech) a bude obsahovat pouze jednu aplikační komponentu a to třídu *EEDCApplication* dědící od *android.app.Application*. Aby se použila naše vlastní implementace aplikace, musíme ji uvést v aplikačním manifestu viz zdrojový kód 7.

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          xmlns:tools="http://schemas.android.com/tools"
          package="cz.martinforejt.eedc">

    <application
        android:name=".EEDCApplication">
        ...
    </application>

</manifest>
```

Zdrojový kód 7 – Aplikační manifest
(app/AndroidManifest.xml)

Hlavní věcí, kterou v naší aplikační třídě musíme udělat, je přepsat metodu *onCreate* a spustit framework Koin se všemi Koin moduly což provedeme příkazem *startKoin* viz zdrojový kód 8.

```kotlin
startKoin {
    androidContext(this@EEDCApplication)
    modules(
            domainModule,
            dataModule,
            presentationModule
    )
}
```

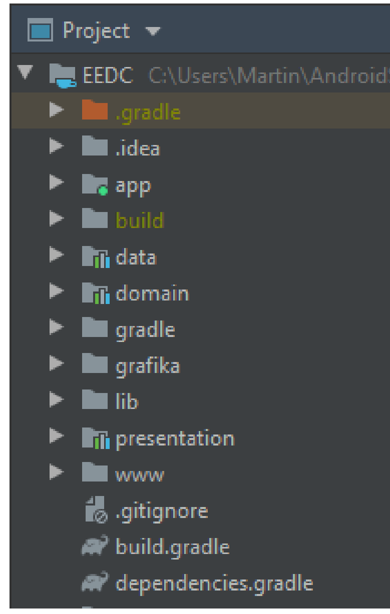
Zdrojový kód 8 - Spuštění Koinu
(app/EEDCApplication.kt)

8.3 Adresářová struktura

Na obrázku 5 je vidět adresářová struktura celého projektu. Každý modul má vlastní adresář, vidíme tu proto adresáře app, data, domain a presentation. Každý z těchto adresářů obsahuje vlastní složku src se zdrojovými kódy, aplikační manifest, zdroje (obrázky, texty,…) a soubor *build.gradle*. Pro gradle se v kořenovém adresáři nachází soubor *build.gradle* pro sestavení celé aplikace.

Dále je zde složka lib, které obsahuje implementaci JavaScriptové knihovny *EEDC.js* a další s tím spojené soubory.

Složka www, obsahuje podpůrné webové stránky, s dokumentací, které jsou dostupné na adrese http://eedc.martinforejt.cz.



Obrázek 5 - Adresářová struktura projektu

9 Implementace knihovny EEDC.js

Tato kapitola popisuje implementaci JavaScriptové knihovny *EEDC.js*, její poskytované funkce a tvorbu a obsah řídících souborů.

9.1 Řídící soubor

Řídící soubor (control file) je archiv ve formátu zip, který musí povinně obsahovat tyto 2 soubory:

- **index.html**, který je zobrazen ve webview a představuje displej pro ovládání zařízení a zobrazování přijatých dat nebo stavu zařízení.
- **eedc-core.js**, který představuje knihovnu *EEDC.js* a je k dispozici ke stažení na zmíněné webové stránce s dokumentací. Tento soubor musí být v HTML přidán v hlavičce viz ukázka *index.html* ve zdrojovém kódu 9.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>...</title>
    <link rel="icon" href="data:;base64,=">
    <link rel="stylesheet" href="style.css">
    <script src="eedc-core.js" type="module"></script>
    <script src="custom.js" type="module"></script>
</head>
<body>
...
</body>
</html>
```

Zdrojový kód 9 - Ukázka index.html

Archiv řídícího souboru může samozřejmě obsahovat další libovolné soubor jako jsou:

- **vlastní JavaScriptové soubor**, viz *custom.js* přilinkovaný v ukázkovém HTML souboru výše.
- **soubory s kaskádovými styly (CSS)**
- **obrázky**
- **a další..**

Takto vytvořený řídící soubor musíme dostat do telefonu, aby mohl být zobrazen ve webview. Tomu se bude věnovat další kapitola, ale ve zkratce máme 3 možnosti. První nejjednodušší možností je nahrát soubor v aplikaci ručně. Další možností je mít soubor uložený na volně přístupném serveru, odkud si ho aplikaci stáhne. V aplikaci se uvede adresa tohoto souboru v internetu. Třetí možností je, že soubor bude uložen v zařízení, které chceme ovládat (případně může být v tomto zařízení dynamicky generován) a pošle se přes vybraný protokol (Wi-Fi, Bluetooth) do telefonu. Při tomto způsobu se v aplikaci zadá příkaz, který se pošle do zařízení jako požadavek na stažení souboru. Více informací jak implementovat odesílání souborů v zařízení je v následující kapitole.

9.2 eedc-core.js

Jak již bylo zmíněno v kapitole 7.2, tento soubor je napsán v JavaScriptu ES6. Díky tomu můžeme použít systém modulů a vytvořit naší knihovnu jako ES6 modul. Soubor obsahuje celkem tři třídy (a dvě další pomocné funkce):

- třída **EEDC**
- třída **DispatcherEvent**
- třída **Mock**

Třída *EEDC* je z tohoto modulu exportována a obsahuje proto klíčová slova *export default*. Všechny její funkce jsou statické a díky tomu lze jednoduše importovat a volat její metody, představují aplikační rozhraní (API) knihovny.

```
let events = {};

export default class EEDC {
    ...
}

class Mock {
    ...
}

class DispatcherEvent {
    constructor(eventName) {
        this.eventName = eventName;
        this.callbacks = [];
    }

    registerCallback(callback) {
        this.callbacks.push(callback);
    }

    unregisterCallback(callback) {
        const index = this.callbacks.indexOf(callback);
        if (index > -1) {
            this.callbacks.splice(index, 1);
        }
    }

    fire(data) {
        const callbacks = this.callbacks.slice(0);
        callbacks.forEach((callback) => {
            callback(data);
        });
    }
}

//const dev = Mock;          // for mock
const dev = EEDCAndroid;     // for android interface
```

Zdrojový kód 10 – DispatcherEvent
(lib/eedc-core.js)

Třída *DispatcherEvent* slouží k poslouchání a zasílání událostí. V souboru existuje globální proměnná *events* typu pole prvků typu *DispatcherEvent*. Pokud chce uživatel poslouchat nějakou událost, zavolá funkci *on* z API a předá jí 2 parametry – typ události a callback, který

se zavolá při vzniku této události. Odhlášení tohoto callbacku je možné přes funkci *off* se stejnými parametry. Funkce *on* (popsána v další podkapitole), vytvoří instanci typu *DispatcherEvent* s daným jménem události a zaregistruje callback pomocí funkce *registerCallback*. Tato instance je přidána do pole *events*. Poté při vzniku dané události se projde pole *events* a všechny callbacky pro danou událost se spustí pomocí funkce *fire* s daty dané události. Implemetace třídy *DispatherEvent* je vidět ve zdrojovém kódu 10.

Třída *Mock* slouží pouze pro testování mimo aplikaci a představuje JavaScriptové rozhraní ve webview viz kapitola 10. Přepnutí mezi rozhraním v aplikaci a tímto mockem provedeme jednoduše zakomentováním dané řádky na konci souboru *eedc-core.js* viz zdrojový kód 10.

9.3 API

Všechny funkce třídy *EEDC* jsou přístupné zvenčí a jsou statické. Celkem 15 funkcí slouží pro použí API ve vlastních scriptech tvůrce řídícího souboru. Další 5 funkcí slouží pro použití z Android rozhraní a neměli by být volány ve scriptech.

9.3.1 Funkce pro pužití ve skriptech

Tyto funkce slouží k použití v JavaScriptech v řídícím souboru. Následuje seznam funkcí s krátkým popisem účelu dané funkce, předávaných parametrů a případné návratové hodnoty.

- **`static`** ***`init`*** **`(callback)`**
  Tato funkce provede inicializaci propojení mezi řídícím souborem a Android rozhraním. Musí být volána jako první, před použitím jiné funkce API a měla by být volána pouze jednou, ideálně ihned po načtení webové stránky. Funkce přijímá jeden parametr, kterým je *callback,* což je funkce, která je volána po prvním připojení k ovládanému (vestavěnému) zařízení.

- **`static`** ***`on`*** **`(eventName, callback)`**
  Již zmiňovaná funkce *on*, slouží k přidání callbacku k poslouchání události. První parametr *eventName* je název událost viz kapitola 9.3.3, druhý parametr *callback* je funkce, která se provede po vzniku dané události.

- **`static`** ***`off`*** **`(eventName, callback)`**
  Funkce *off* slouží k odhlášení callbacku od události, který byl předtím zaregistrován pomocí funkce *on*. První parametr *eventName* je název události, druhý parametr *callback* je stejná funkce, která byla předána při registraci pomocí funkce *on*.

- **`static`** ***`sendString`*** **`(data)`**
  Funkce *sendString* odešle řetězec předaný v parametru *data* do ovládaného zařízení.

- **`static`** ***`sendBytes`*** **`(data, ch1, ch2)`**
  Funkce *sendBytes* odešle bytová data do ovládaného zařízení. První parametr *data* je typu *Uint8Array* a představuje pole bytů, které se mají odeslat do zařízení. Parametry *ch1* a *ch2* jsou nepovinné a slouží pro použití s Bluetooth Low Energy (ble), *ch1* představuje identifikaci (UUID) pro ble service, *ch2* představuje UUID pro ble charakteristiku z dané service, do které se mají *data* zapsat.

- **`static`** ***`listenLines`*** **`()`**
  Funkce *listenLines* slouží k poslouchání (čtení) řetězcových dat z ovládaného zařízení řádek po řádku. Po přijetí nových dat, respektive nové řádky, vzniká událost *new_line* viz kapitola 9.3.3.

- **static *listenBytes*(bufferSize, ch1, ch2, ch3)**
  Funkce *listenBytes* je obdoba předchozí funkce, ale čte jednotlivé byty. Po přijetí nových dat vzniká událost *new_bytes*. První parametr *bufferSize*, udává velikost použitého bufferu při čtení bytů. Parametry *ch1, ch2* a *ch3* jsou opět nepovinné a slouží pro použití s ble. *Ch1* představuje UUID pro ble service, *ch2* UUID pro ble charakteristiku z dané service a *ch3* případný ble deskriptor dané charakteristiky.

- **static *stopListenLines*()**
  Funkce *stopListenLines* přeruší čtení dat, které bylo předtím inicializováno pomocí funkce *listenLines*.

- **static *stopListenBytes*(ch1, ch2, ch3)**
  Funkce *stopListenBytes* přeruší čtení dat, které bylo předtím inicializováno pomocí funcke *listenBytes* (se stejnými parametry *ch1, ch2, ch3*).

- **static *readSingleLine*()**
  Funkce *readSingleLine* je obdoba funkce *listenLines*, ale čeká na přijetí pouze první řádky a další už nečte. Opět vzniká událost *new_line*.

- **static *readSingleBytes*(bufferSize, ch1, ch2)**
  Funkce *readSingleBytes*, při použití bez *ch1* a *ch2* je obdoba funkce *listenBytes*, ale čte do bufferu o velikosti *bufferSize* pouze jednou. Opět vzniká událost *new_bytes*. Při použití s ble zařízením a tedy s parametry *ch1* (UUID pro ble service) a *ch2* (UUID pro ble charakteristiku) se nenastavuje notifikace u této charakteristiky jako v případě funkce *listenBytes*, ale pouze se přečte aktuální hodnota dané charakteristiky (opět vzniká událost *new_bytes*).

- **static *getDeviceInfo*(): string**
  Funkce *getDeviceInfo* vrací informace o aktuálním zařízení na kterém běží aplikace (telefonu) a o ovládaném zařízení (které zadal uživatel při vytváření připojení k zařízení). Informace jsou vráceny jako json řetězec. Seznam všech hodnot, které tato funkce vrací je k dispozici v API dokumentaci:
  http://eedc.martinforejt.cz/api.html

- **static *setFullScreen*(fullScreen)**
  Funkce *setFullScreen* slouží k zapnutí nebo vypnutí fullscreen režimu aplikace. Při fullscreen režimu je schována notifikační lišta (status bar) a aplikační lišta (action bar). Zapnutí nebo vypnutí je specifikováno pomocí parametru *fullscreen* typu boolean, kde hodnota true představuje zapnutí a hodnota false vypnutí fullscreen režimu.

- **static *isFullScreen*(): boolean**
  Funkce *isFullScreen* slouží pro zjištění aktuálního stavu fullscreen režimu. Aktuální stav je vrácen jako boolean hodnota, kde true představuje zapnutý fullscreen režim a hodnota false vypnutý fullscreen režim.

- **static *exit*()**
  Funkce exit provede odpojení od ovládaného zařízení a vypnutí obrazovky s webview (s řídícím souborem).

9.3.2 Funkce pro použití v Android rozhraní

Tyto funkce slouží k použití v Android rozhraní. Následuje seznam funkcí s krátkým popisem účelu dané funkce a předávaných parametrů.

- **`static _onConnected(data)`**
  Funkce *_onConnected* je volána z Android rozhraní v okamžiku, kdy je zařízení prvně připojeno. (V dalších případech po případném výpadku spojení a jeho opětovném obnovení se použije funkce *_onStateChange*). Parametr *data* je nyní nevyužit, slouží pro případné pozdější použití v budoucnosti (posílá se prázdný řetězec).
- **`static _onStateChange(data)`**
  Funkce *_onStateChange* je volána z Android rozhraní při každé změně stavu připojení a nový stav je předán v textové formě v parametru *data*. Existují 3 možné stavy, které mohou být pomocí této funkce předány a to: *CONNECTING* v případě probíhajícího pokusu o připojení, *CONNECTED* při úspěšném připojení a *DISCONNECTED* při odpojení od zařízení po výpadku spojení.
- **`static _onReceiveLine(data)`**
  Tato funkce je volána z Android rozhraní při přijetí nové řádky, která je předána v parametru *data*. Funkce vyvolá událost *new_line*.
- **`static _onReceiveBytes(data, ch1, ch2, ch3)`**
  Tato funkce je volána z Android rozhraní při přijetí nových bytů, které jsou předány v parametru *data*. V případě komunikace s ble zařízením jsou použity parametry *ch1* pro UUID ble service, *ch2* pro UUID ble charakteristiky a případně *ch3* pro UUID deskriptoru. Funkce vyvolá událost *new_bytes*.
- **`static dispatch(eventName, data)`**
  Tato funkce není přímo volána z Android rozhraní, ale je volána ve výše uvedených funkcích a slouží k vyvolání události jejíž název je předán v parametru *eventName* s daty v parametru *data*.

9.3.3 Události

Jak již bylo zmíněno výše, pomocí funkce *on* a *off* lze zaregistrovat respektive odhlásit callback ke/od vzniku události. API používá následující 4 události:

- **connected** – je vyvolána při prvotním úspěšném připojení k zařízení. (V dalších případech po případném výpadku spojení a jeho opětovném obnovení se použije událost *state_change*). Událost nemá žádná data.
- **state_change** – je vyvolána při změně stavu zařízení. Existují 3 možné stavy, které mohou být pomocí této události předány jako její data a to: *CONNECTING* v případě v probíhajícího pokusu o připojení, *CONNECTED* při úspěšném připojení a *DISCONNECTED* při odpojení od zařízení po výpadku spojení.
- **new_line** – je vyvolána při přijetí nové řádky, která je předána jako data této událost.
- **new_bytes** – je vyvolána při přijetí nových bytů. Data této události představuje objekt v JavaScriptu s hodnotami *data*, což jsou vlastní přijaté byty, *ch1, ch2 a ch3,* které představují již výše zmíněné UUID při použití s ble zařízeními.

9.4 Použití API

Příklad použití API představuje ukázkový soubor *custom.js* uvedený ve zdrojovém kódu 11, který by byl přidán do HTML viz zdrojový kód 9. V tomto souboru se nejprve importuje *EEDC.js* knihovna. Následně se volá funkce *init* a v předaném callbacku (který se provede po prvním úspěšném připojení k zařízení) se nejprve zaregistruje callback pro událost *state_change*, poté se zaregistruje callback pro událost *new_line*. Po registraci těchto callbacků k událostem se začnou poslouchat přijaté řádky ze zařízení pomocí funkce

listenLines. Poslední příkaz získá informace o aktuálním zařízení kde běží aplikace (telefonu) a o ovládaném zařízení pomocí funkce *getDeviceInfo*.

```
import EEDC from './eedc-core.js'; // import eedc library

EEDC.init(function () {
    // device is first time connected

    EEDC.on("state_change", function (data) {
        if (data === 'CONNECTED') {
            // device is again connected
        }
    });

    EEDC.on("new_line", function (line) {
        // new line received
    });

    EEDC.listenLines(); // start listen lines
    let info = EEDC.getDeviceInfo(); // get device info
});
```

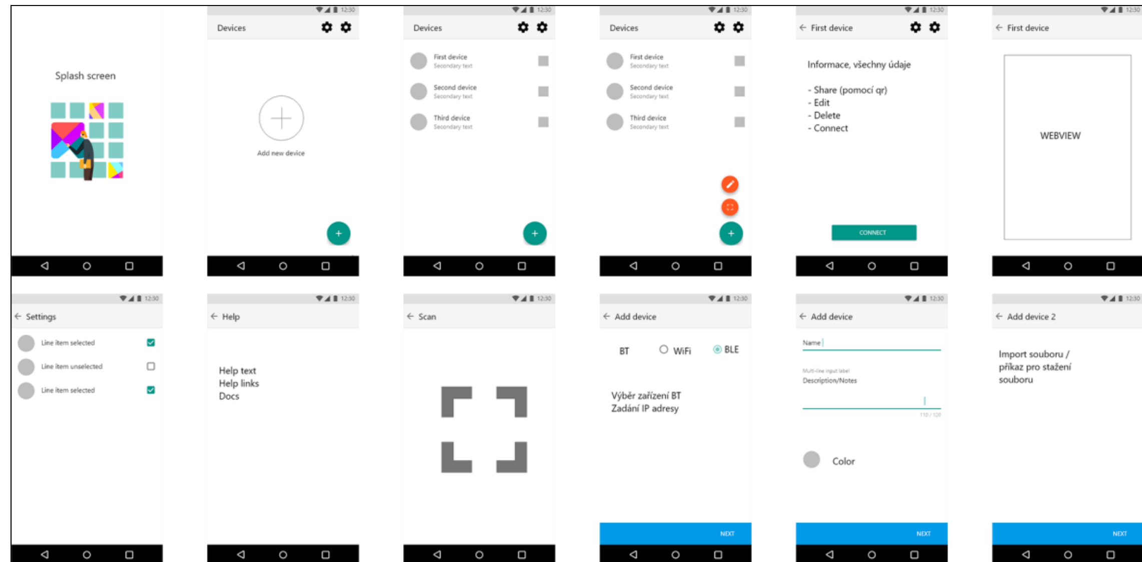
Zdrojový kód 11 - Ukázka custom.js

10 Realizace Android aplikace

V této kapitole je nejprve popsáno grafické uživatelské rozhraní aplikace a všechny hlavní funkce, které aplikace nabízí včetně popisu jednotlivých obrazovek. Poté následuje popis nejdůležitějších komponent a tříd a jejich interakce. Dále funkce řídícího souboru z pohledu Android aplikace, způsoby stažení, uložení řídících souborů v paměti zařízení a jejich zobrazení a komunikace s nimi skrz rozhraní JavaScriptu.

10.1 Uživatelské rozhraní

Před začátkem vývoje bylo navrženo jednoduché uživatelské rozhraní v programu Adobe XD[12]. Tento nástroj umožňuje navrhovat ux/ui a tvořit jednoduché prototypy. Výsledný hrubý návrh, je vidět na obrázku 6. Uživatelské rozhraní bude využívat převážně systémové komponenty, jak je vyžadováno v kapitole 6.1, aby bylo pro uživatele Androidu známé a srozumitelné a bude se tedy co nejvíce držet tzv. material designu, který je součástí Androidu od verze 5.1 (se zpětnou kompatibilitou viz AppComapt z kapitoly 7.3).



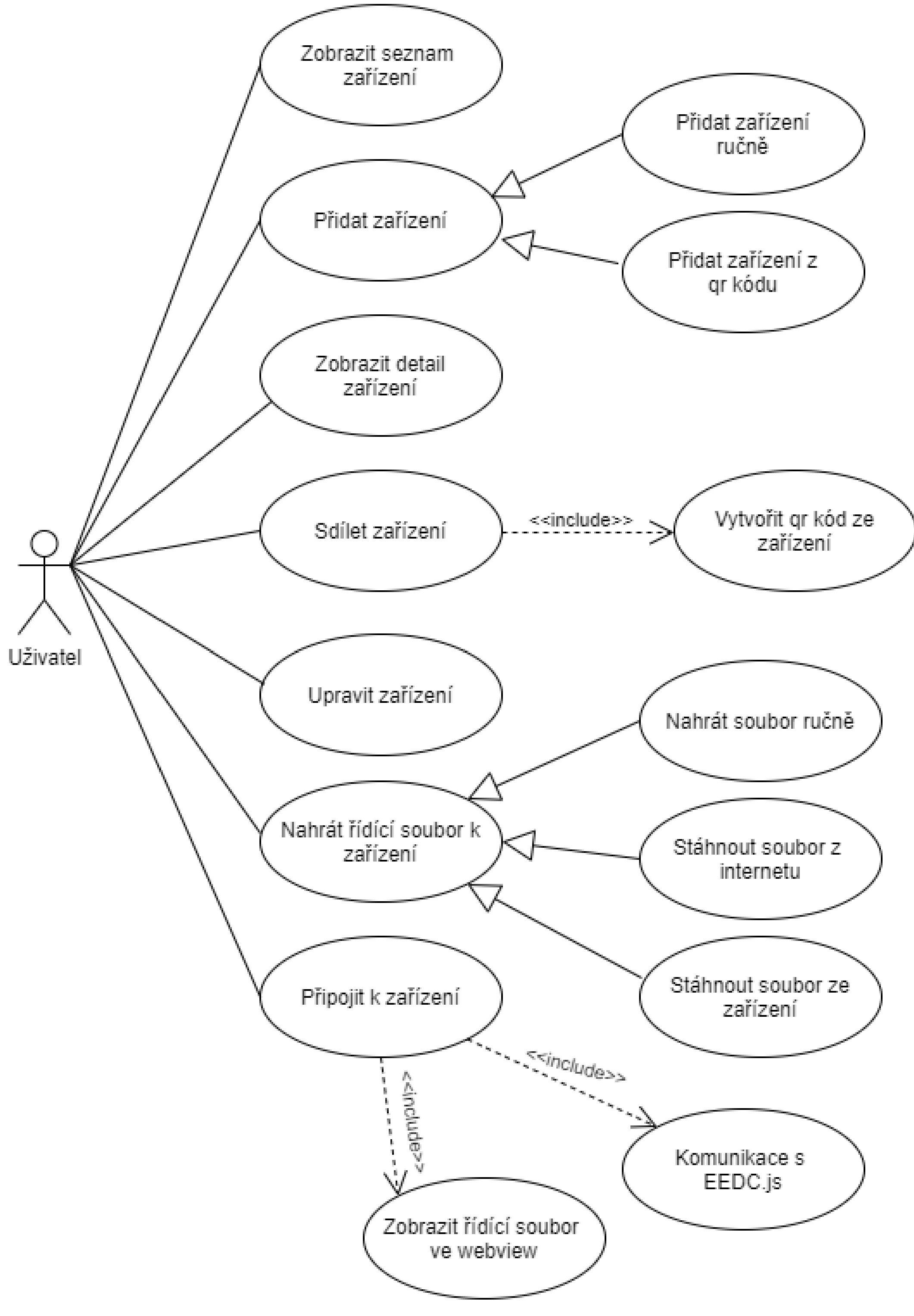
Obrázek 6 - Návrh gui

10.2 Funkce aplikace

Hlavní funkcí aplikace je bezpochyby ovládání vestavěných zařízení pomocí Wi-Fi, Bluetooth a nebo Bluetooth Low Enegy. Aplikace obsahuje další řadu podpůrných funkcí, které uživatelům práci s ní usnadňují, jako je možnost vytvořit více zařízení a vybrání, ke kterému se chci zrovna připojit, editace záznamu zařízení nebo sdílení zařízení pomocí QR kódu a následný import do jiného zařízení pomocí tohoto QR kódu.

Hlavní funkce aplikace byly zaneseny do diagramu případů užití (obrázek 7) a níže následuje popis těchto hlavních funkcí včetně screenshotů z aplikace.

[12] Adobe XD - https://www.adobe.com/cz/products/xd.html

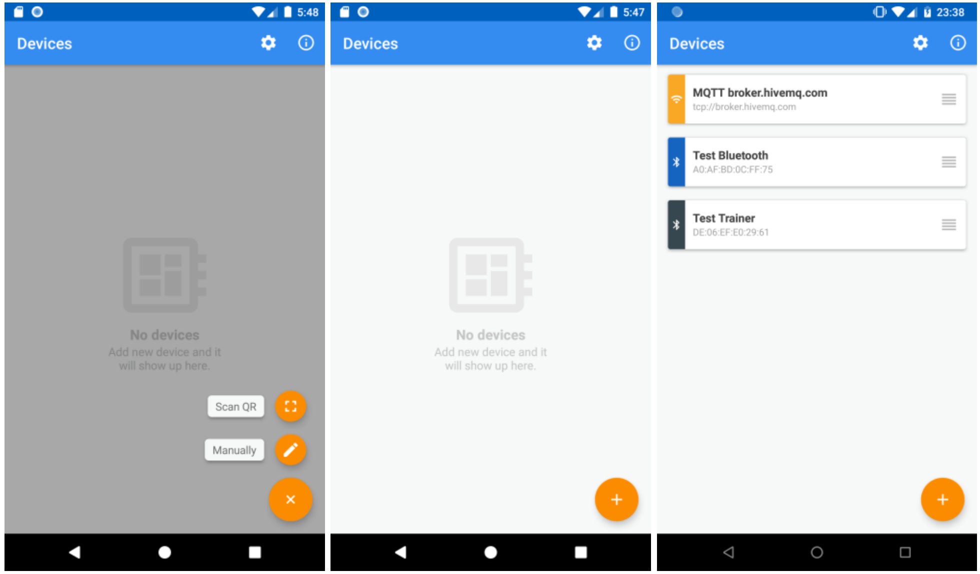


Obrázek 7 – Diagram případů užití

10.2.1 Seznam zařízení

Seznam zařízení je obrazovka, která se zobrazí ihned po spuštění aplikace (pokud nebereme v potaz spouštěcí obrazovku, tzv. splash screen, která se zobrazí na několik málo vteřin a nemá žádnou jinou funkčnost). Pokud v databázi ještě není žádné zařízení zobrazí se prázdný seznam jak je vidět na první a druhé části obrázku 8. Pokud již databáze obsahuje nějaké záznamy zobrazí se seznam těchto záznamů (viz poslední část obrázku 8), kde každá položka obsahuje název zařízení a adresu pro připojení. Vlevo je ikonka reprezentující

způsob připojení (Wi-Fi x Bluetooth), která je podbarvena barvou zařízení (viz další kapitola). V pravé části každé položky je ještě ikonka pro posun položky v seznamu, aby si uživatel mohl zařízení seřadit podle sebe. Výchozí řazení je podle data přidání, kde později přidané položky se přidávají na konec seznamu.

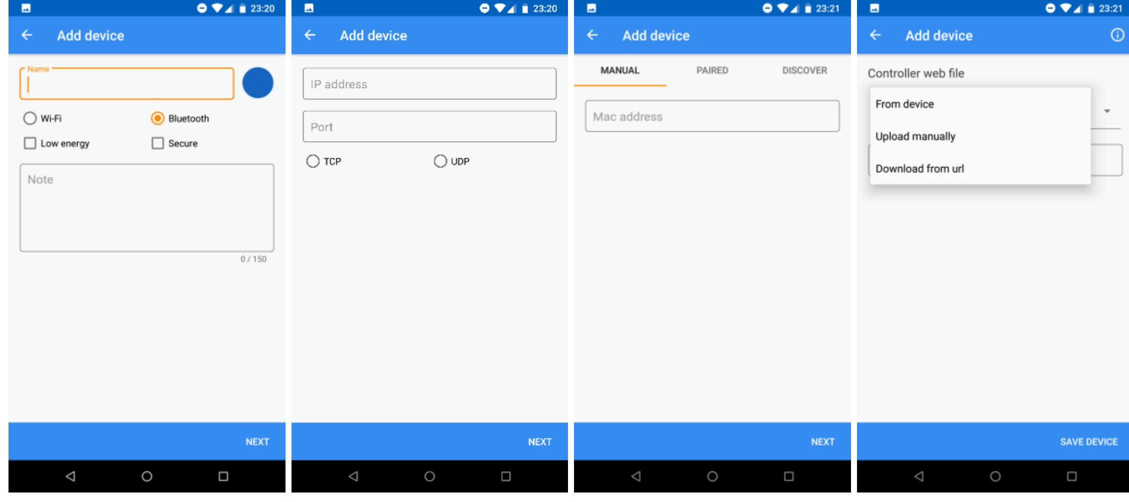


Obrázek 8 - Seznam zařízení

V horní liště jsou dvě tlačítka pro přechod do nastavení aplikace a pro přechod na obrazovku obsahující informace o aplikaci. Spodní tlačítko se symbolem + je probráno dále.

10.2.2 Přidat zařízení

Přidání zařízení se provede z hlavní obrazovky (obrazovky seznamu zařízení) kliknutím na tlačítko se symbolem + v dolním pravém rohu obrazovky. Tím se nám ukáže kontextové menu s dvěma možnostmi jak zařízení přidat. První možností je zařízení přidat manuálně, druhou možností je přidat zařízení pomocí QR kódu.



Obrázek 9 - Přidat zařízení manuálně

Pokud se rozhodneme přidat zařízení manuálně (obrázek 9), nejprve zadáme obecné informace o zařízení, společné pro všechny typy připojení a informace o připojení. První

zadávanou hodnotou je jméno zařízení, které slouží jen pro nás a orientaci v případě více zařízení. Dále vybereme barvu zařízení pomocí modrého kolečka vedle jména, které zobrazí dialog s nabídkou několika barev. Poté vybereme způsob připojení a to Wi-Fi nebo Bluetooth a v případě Bluetooth máme na výběr ještě dva checkboxy. První *low energy* zaškrtneme pokud se jedná o Bluetooth Low Energy zařízení (respektive připojení), druhý *secure* je pro zabezpečené připojení, před kterým musí být zařízení spárována. Proces párování je automaticky inicializován při prvním pokusu o připojení k zařízení. Poslední věc kterou můžeme zadat, je krátká poznámka o maximálně 150 znacích.

Poté následuje zadání informací relevantních pro daný typ připojení. V případě Wi-Fi se jedná o IP adresu (nebo název serveru), port a typ protokolu – TCP nebo UDP. V případe Bluettoth máme možnost buď zadat ručně mac adresu zařízení, nebo vybrat zařízení ze seznamu spárovaných zařízení nebo zařízení vyhledat.

V posledním kroku při manuálním přidávání je přidání řídícího souboru. Máme na výběr 3 způsoby jak stáhnout řídící soubor do zařízení. První způsob je ruční přidání, kdy soubor vybere uživatel z lokálního souborového systému. Druhý způsob je stažení souboru z internetu a třetí možnost je stažení souboru přímo ze zařízení. Detailněji bude přidávání souborů probráno v kapitole 10.5.

Druhou možností jak přidat zařízení, je pomocí QR kódu. QR kód zařízení lze vygenerovat v aplikaci v detailu zařízení viz další část. Po kliknutí na položku skenování v kontextovém menu pro přidání zařízení na obrazovce seznam zařízení se zobrazí kamera, která ihned začíná skenovat. Jakmile skener detekuje QR kód, který je validní přidá se zařízení do databáze a nastane návrat na obrazovku se seznamem zařízení, kde již bude i nově přidané zařízení. Pokud máme QR kód uložený v telefonu ve formě obrázku, můžeme použít ikonku pro nahrání QR kódu v horní liště na pravo (obrázek 10). Po kliknutí se zobrazí dialog pro vybraní souboru s QR kódem a poté se pokračuje stejně jako v případě jeho detekce kamerou.

Obrázek 10 - Skenování QR kódu

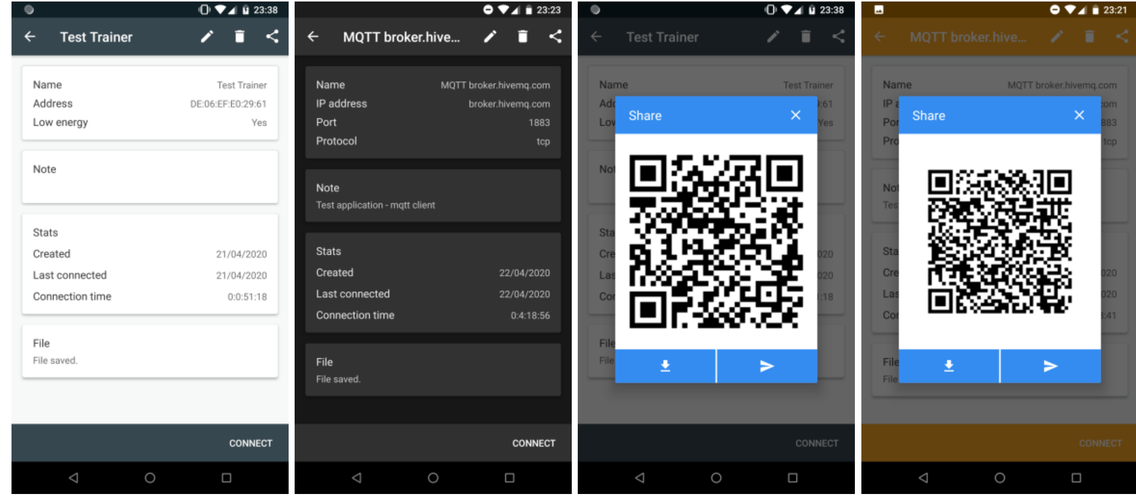
10.2.3 Detail zařízení

Do detailu zařízení se dostaneme ze seznamu zařízení kliknutím na položku ze seznamu. Detail zařízení je vidět na obrázku 11 a jeho obsah je rozdělen do 4 částí. V první části jsou obecné informace, které jsme zadávali při vytváření zařízení, jako název zařízení, adresa, port, protokol nebo zda se jedná o ble zařízení. V druhé části je textová poznámka. Ve třetí

části jsou statistiky, které obsahují datum vytvoření zařízení, datum posledního připojení k zařízení a celková suma času připojení k zařízení ve formátu *dny:hodiny:minuty:sekundy*. Poslední část informuje zda je k zařízení uložen řídící soubor. Informování je textové v podobě „File saved.“ nebo „No file.“.

Ve spodní části obrazovky je tlačítko *connect* pro připojení k zařízení.

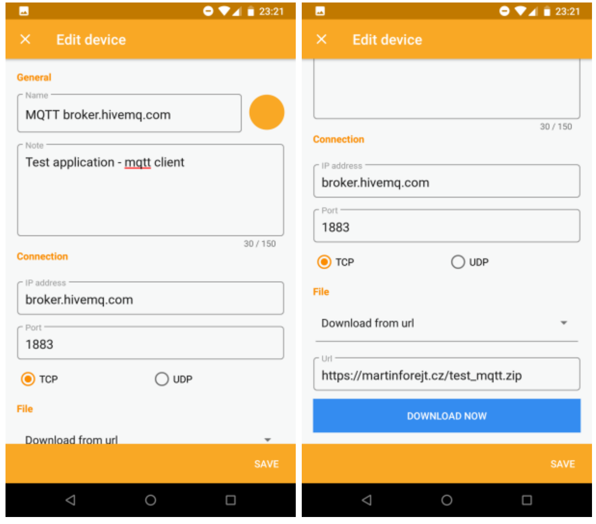
Horní lišta obsahuje 3 ikony (položky menu). První zleva slouží k editaci zařízení viz dále, druhá slouží ke smazání zařízení a třetí ke sdílení zařízení. Při sdílení zařízení se zobrazí dialog, který je vidět v pravé části na obrázku 11 a který obsahuje velký obrázek s QR kódem. Tento QR kód lze přímo načíst při přidávání zařízení viz předchozí funkce, nebo lze použít jedno ze dvou tlačítek ve spodní části dialogu. První slouží k uložení QR kódu ve formě obrázku do zařízení (do složky *downloads*). Druhé slouží ke sdílení QR kódu ve formě obrázku pomocí jiných aplikací v telefonu (např. přiložením do emailu).



Obrázek 11 - Detail zařízení

10.2.4 Upravit zařízení

V úpravě zařízení můžeme změnit de facto stejné informace, které jsme zadávali při vytváření zařízení, kromě změny typu připojení (Wi-Fi x Bluetooth). Důležitá je poslední část, kde lze stáhnout řídící soubor viz obrázek 12.



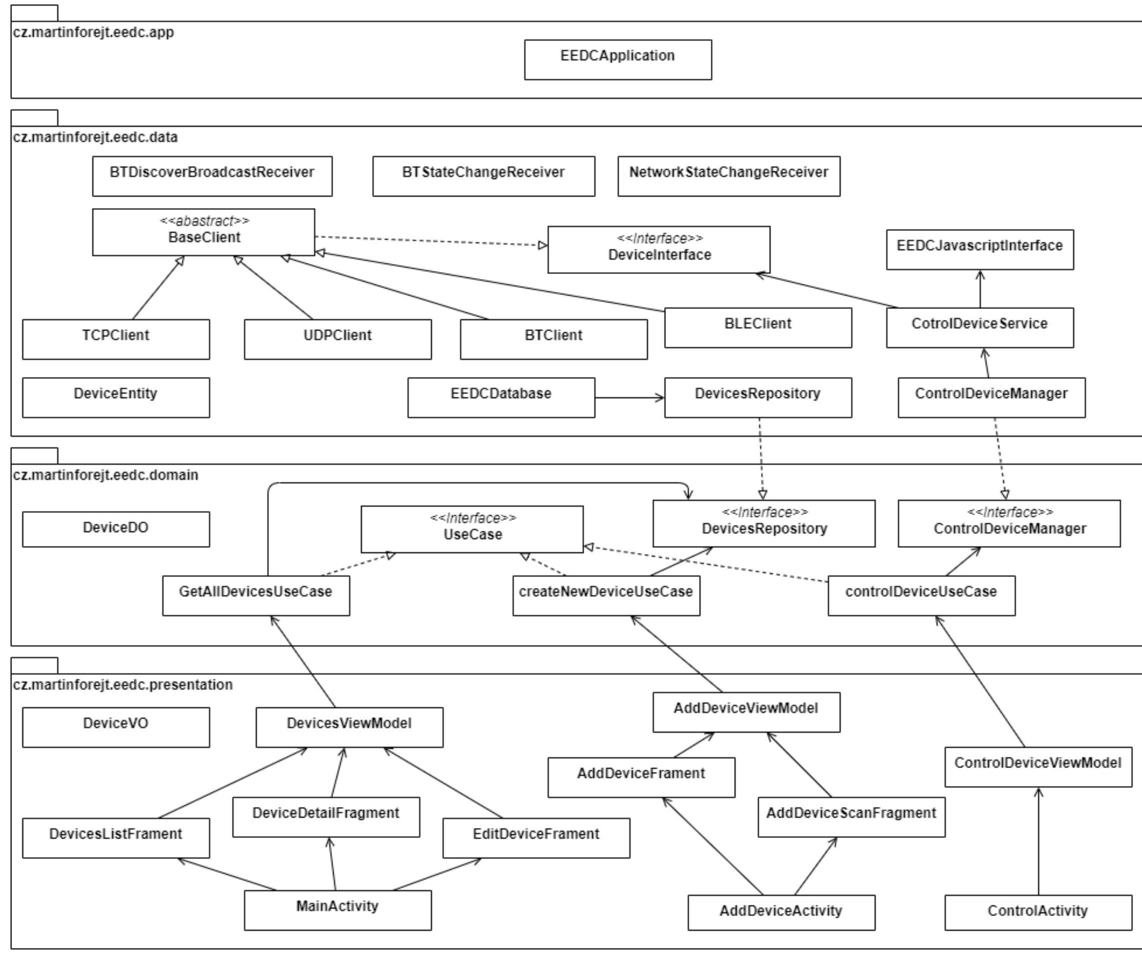
Obrázek 12 - Upravit zařízení

10.2.5 Ovládání zařízení

Pokud v detailu zařízení klikneme na dolní tlačítko *connect*, zobrazí se obrazovka pro ovládání zařízení, která obsahuje pouze webview přes celou svou plochu a ve webview je načten soubor *index.html* z řídícího souboru. Komunikace s *EEDC.js* probíhá pomocí rozhraní JavaScriptu.

10.3 Aplikační komponenty

V této části jsou popsány nejdůležitější aplikační komponenty a další třídy v aplikaci. Na obrázku 13 je vidět diagram tříd, který obsahuje pouze ty nejdůležitější komponenty nutné k pochopení principu struktury aplikace. Komponenty jsou v diagramu rozděleny dle modulů a je tak krásně vidět oddělení jednotlivých vrstev a jejich závislosti. Dále následuje popis jednotlivých komponent.



Obrázek 13 – Diagram tříd

EEDCApplication

Třída *EEDCApplication* z balíku *app*, představuje aplikační třídu celé Android aplikace. Je proto uvedena v manifestu v atributu *name* u xml tagu *application*. Jediným účelem aplikace je spuštění Koinu a inicializace Sentry (viz kapitola 11.5). Spuštění Koinu již bylo probráno v kapitole 8 (a je vidět ve zdrojovém kódu 8).

MainActivity

Hlavní aktivita, uvnitř ní jsou zobrazovány fragment *DevicesListFragment, DeviceDetailFragment, EditDeviceFragment* a další méně důležité fragmenty jako fragment pro obrazovku nastavení nebo informace o aplikaci.

AddDeviceActivity

Aktivita pro přidávání zařízení, zobrazuje fragmenty *AddDeviceFragment*, který představuje první obrazovku pro ruční přidávání zařízení a stejně tak fragmenty pro všechny další kroky, tedy *AddDeviceBTFragment, AddDeviceWiFiFragment, AddDeviceFileFragment* a fragment pro skenování QR kódu *AddDeviceScanFragment*.

ControlActivity

Aktivita pro vlastní ovládání externího zařízení. Zobrazuje v sobě webview (které je vytvořeno v *ControlDeviceManager* viz dále). Vlastní ViewModel *ControlDeviceViewModel*, přes který začne (a případně ukončí) komunikaci a dle kterého řídí ui, jako například zobrazení dialogu při odpojení od zařízení, nastavení fullscreen módu a podobně.

DeviceDO

Doménový objekt zařízení. Jedná se o datovou třídu (data class), která si drží všechny informace o zařízení jako je id, jméno, popis, typ připojení, adresa, port, atd.

DeviceVO

Prezentační (view) verze třídy *DeviceDO* vhodnější pro účely gui. Pro převod z třídy *DeviceDO* do *DeviceVO* slouží tzv. mapovací funkce.

DeviceEntity

Datová verze třídy *DeviceDO* vhodnější pro účely uložení v databázi. Pro převod z (a do) třídy *DeviceDO* slouží také mapovací funkce. Více o databázi je v kapitole 10.4.

DevicesListFragment

Fragment, jehož hlavním úkolem je zobrazení seznamu zařízení viz funkce seznam zařízení v kapitole 10.2.1. Fragment má ViewModel *DevicesViewModel*, z kterého získá LiveData se seznamem zařízení (typu *DeviceVO*). Fragment tento seznam zobrazuje pomocí komponenty uživatelského rozhraní recyclerview. Dále má na starost přechod na obrazovku detailu zařízení při kliku na položku v recyclerview a přechod na obrazovku přidání zařízení (ručně nebo skenováním QR kódu). Při navigaci do detailu předává jako argument index z tohoto seznamu.

DeviceDetailFragment

Fragment zobrazující detail zařízení, má také ViewModel *DevicesViewModel*, který sdílí s dalšími fragmenty jako je předchozí *DevicesListFragment.* Při spuštění dostává jako argument index v seznamu zařízení v LiveData proměnné z ViewModelu. Tím získá aktuální zařízení (*DeviceVO*) a zobrazí ho (pomocí databindingu). Před přechodem na obrazovku ovládání zařízení zjistí (pomocí funkcí ViewModelu), zda je dostupná síť nebo zapnuté Bluetooth a pokud ne, zobrazí uživateli dialog s výzvou.

EditDeviceFragment

Fragment pro editaci zařízení, dostává index v seznamu zařízení ve sdíleném ViewModelu, stejně jako předchozí fragment. Zobrazuje formulář pro editaci zařízení. Má ViewModel *EditDeviceViewModel* pomocí kterého validuje vstupy a ukládá zařízení.

DevicesViewModel

ViewModel, který obsahuje LiveData se seznamem zařízení (*DeviceVO)*. Má několik usecasů (*UseCase*), ke kterým umožňuje přístup přes funkce, a jsou to např.: *GetAllDeviceUseCase* pro získání již zmíněných LiveDat se seznamem zařízení, *UpdateDevicesUseCase* pro uložení upravených zařízení do databáze, *DeleteDeviceUseCase* pro odstranění zařízení, *IsBtEnabledUseCase* pro ověření zda je zapnutý Bluetooth, *EnableBtUseCase* pro zapnutí Bluetooth a *IsNetworkAvailable* pro zjištění dostupnosti sítě.

AddDeviceFragment

Fragment představující první krok při manuálním přidávání zařízení (zde uveden jako zástupce dalších fragmentů, kterými jsou *AddDeviceBTFragment, AddDeviceWifiFragment* a *AddDeviceFileFragment*). Obsahuje sdílený ViewModel *AddDeviceViewModel*, který sdílí s uvedenými fragmenty a který si udržuje model vytvářeného zařízení.

AddDeviceScanFragment

Fragment pro skenování QR kódu (nebo ruční vybrání obrázku s QR kódem). Obsahuje sdílený ViewModel *AddDeviceViewModel*. Využívá volně dostupného nástroje *ZBar*[13] z volně dostupné knihovny pro Android, která tento nástroj obaluje – *dm77/barcodescanner*[14]. V layoutu má jednu komponentu grafického rozhraní a to vlastní *EEDCScannerView*, který rozšiřuje *ZbarScannerView* z uvedené knihovny. Při detekci QR kódu předá jeho textovou hodnotu ViewModelu k dalšímu zpracování.

AddDeviceViewModel

ViewModel pro přidávání zařízení, má tyto usecasy, ke kterým poskytuje přístup přes funkce: *DecodeStringToDeviceUseCase* pro vytvoření objektu *DeviceDO* z řetězce (z QR kódu), *CreateNewDeviceUseCase* pro uložení nového zařízení do databáze, *DeleteDeviceUseCase* pro odstranění zařízení, *GetLocalFileInfoUseCase* pro získání informací o souboru v lokálním souborovém systému (informace jako název souboru a velikost souboru) z cesty k tomuto souboru, *UpdateDevicesUseCase* pro uložení upravených zařízení a *ImportDeviceFromLocalFileUseCase* pro uložení zařízení ze souboru (obrázku s QR kódem). Dále ViewModel obsahuje instanci typu *DeviceForm*, která představuje aktuálně vytvářené zařízení a jehož atributy jdou editovat a z které lze vytvořit *DeviceDO* pro následné uložení do databáze.

ControlDeviceViewModel

ViewModel pro řízení (vestavěného) zařízení, který využívá aktivita *ControlActivity*. Má usecase *ControlDeviceUseCase*, který vrací objekt typu *DeviceController* přes který může ViewModel komunikovat s *ControlDeviceManager*(em).

DeviceController obsahuje funkce *getState* pro získání aktuálního stavu zařízení (LiveData), *webPageStatus* pro získání stavu načítání řídícího souboru (LiveData), *getFullScreen* pro

---

[13] Zbar - http://zbar.sourceforge.net/
[14] dm77/barcodescanner - https://github.com/dm77/barcodescanner

získání aktuálního požadavku na fullscreen režim (zapnuto/vypnuto) (také LiveData), *connect* pro obnovení připojení k zařízení, *disconnect* pro odpojení od zařízen a funkce *goBackground* a *goForeground* pro informování manageru o viditelnosti aktivity.

UseCase

Abstraktní třída *UseCase* (viz zdrojový kód 12), která představuje usecase a všechny ostatní usecasy ho musí implementovat. Jedná se o generické rozhraní s dvěma generickými typy. První určuje typ výstupu z tohoto usecasu, druhý určuje typ vstupního parametru u funkcí usecasu. Usecase má 3 varianty funkce pro jeho spuštění. První *execute* se provede synchronně a může být volána odkudkoliv, druhá *executeSync*, která je označena klíčovým slovem *suspend* se provede také synchronně, ale musí být volána uvnitř korutiny. Třetí *executeAsync* se provede asynchronně v korutině a výsledek vrátí pomocí předaného callbacku. Každý usecase může přepsat libovolné z těchto funkcí, pokud ale některou nepřepíše a bude pomocí ní spuštěn dojde k vyhození výjimky *UseCaseMethodNotImplementedException*.

```
abstract class UseCase<out R, in P> {

    open fun execute(params: P): R {
        throw UseCaseMethodNotImplementedException("...not implemented!")
    }

    open suspend fun executeSync(params: P): R {
        throw UseCaseMethodNotImplementedException("...not implemented!")
    }

    fun executeAsync(params: P, callback: (R) -> Unit) {
        GlobalScope.launch {
            callback(executeSync(params))
        }
    }
}
```

Zdrojový kód 12 – UseCase
(domain/core/usecase/UseCase.kt)

GetAllDevicesUseCase

Usecase pro získání seznamu všech zařízení v databázi (jako LiveData). Ve svém konstruktoru má parametr typu *DevicesRepository*, přes který získá seznam zařízení pomocí funkce *getAllDevices*. Vstupem usecasu není nic (*Unit*) a výstupem je seznam zařízení (*LiveData<List<DeviceDO>>*).

CreateNewDeviceUseCase

Stejně jako předchozí usecase má přístup k *DevicesRepository* a pomocí její funkce *createNewDevice* vytvoří nové zařízení. Vstupem usecasu je zařízení (*DeviceDO*) a výstupem to samé zařízení, ale již s přiřazeném id, nebo null pokud je zařízení nevalidní (*DeviceDO?)*.

ControlDeviceUseCase

Usecase pro získání controlleru (*DeviceController*) pro komunikaci s *ControlDeviceManager* a webview s načteným řídícím souborem. Usecase má přístup k *ControlDeviceManager* nad kterým vyvolá funkci *startControl*. Vstupem usecasu je třída *Params*, která obsahuje id

zařízení a callback, který se vyvolá po spuštění service *ControlDeviceService*. Výstupem je zmiňovaný controller a webview obalené ve třídě *StartControlResult*.

DevicesRepository (domain)

Rozhraní *DevicesRepository* z modulu *domain* definuje funkce pro získávání a editaci zařízení (*DeviceDO*): *getAllDevices*, která vrací LiveData se seznamem všech zařízení (*LiveData<List<DeviceDO>>*), *createNewDevice*, která uloží nové zařízení (*DeviceDO*), *getDeviceById*, které vrátí zařízení (*DeviceDO*) dle jeho id (*Int*), *updateDevice*, která uloží změny v zařízení (*DeviceDO*), *updateAll*, která uloží seznam změněných zařízení a *deleteDeviceById*, která vymaže zařízení z databáze.

DevicesRepository (data)

Třída *DevicesRepository* z modulu *data* implementuje rozhraní *DevicesRepository* z modulu *domain* uvedené výše. Ve svém konstruktoru dostává instanci třídy *EEDCDatabase* pro přístup k databázi.

ControlDeviceManager (domain)

Rozhraní *ControlDeviceManager* z modulu *domain* definuje funkce pro zahájení řízení zařízení *startControl*, která přijímá id zařízení a callback, který je volán po startu service *ControlDeviceService* a vrací controller (*DeviceController*) a instanci třídy *WebView*. Dále definuje funkci *getControlledDeviceId*, která vrací id aktuálně řízeného zařízení nebo null, pokud není žádné zařízení řízeno.

ControlDeviceManager (data)

Třída *ControlDeviceManager* z modulu *data* implementuje rozhraní *ControlDeviceManager* z modulu *domain* uvedené výše. Při zavolání funkce *startControl* spustí service *ControlDeviceService* a vytvoří webview do kterého načte řídící soubor zařízení. Cestu řídícího souboru získá pomocí usecasu *GetControlFileByDeviceUseCase*. Dále vytvoří instanci *EEDCJavascriptInterface*, kterou přidá do webview pomocí funkce *addJavascriptInterface*. Manager řídí service *ControlDeviceService* a přijatá data předává do webview přes *EEDCJavascriptInterface*.

EEDCDatabase

Třída *EEDCDatabase* rozšiřuje třídu *RoomDatabase* a má abstraktní funkci pro získání jediného dao (Data access object) a to *DeviceDao*. Statická funkce *getDatabase* vrací statickou proměnnou *instance* a pokud je *instance* null nejprve ji vytvoří pomocí builderu *Room.databaseBuilder()…build()*

ControlDeviceService

ControlDeviceService je aplikační komponenta service. Běží na pozadí i když jsou všechny aktivity ukončené a udržuje připojení k zařízení. Ke komunikaci se zařízením využívá rozhraní *DeviceInterface*. Má 3 LiveData proměnné do kterých ukládá nově přijatá data (jedna pro String druhá pro ByteArray) a stav připojení (*DeviceState*). Aby bylo zajištěno, že service nebude ukončen operačním systémem, je definován jako *foreground service*, a pomocí funkce *startForeground* zobrazuje notifikaci v notifikační liště. V této notifikaci je také informace o stavu připojení a počet nepřečtených dat za dobu co byla aktivita pozastavená (nebo ukončená).

DeviceInterface

Rozhraní *DeviceInterface* definuje přístup k zařízení pomocí funkcí*:*

- ***open*(): OpenStatus**
  Otevře připojení k zařízení a vrací hodnotu výčtového typu *OpenStatus* (CONNECTING, CONNECTED nebo ERROR)

- ***close*(): Boolean**
  Ukočí připojení, vrací boolean hodnotu zda proběhlo ukončení v pořádku.

- ***send*(data: String): Boolean**
  Odešle řetězec *data* do zařízení a vrací boolean hodnotu zda proběhlo odeslání v pořádku.

- ***send*(data: ByteArray, ch1: Int?, ch2: Int?, ch3: Int?): Boolean**
  Odešle pole bytů *data* do zařízení. Parametry *ch1, ch2* a *ch3* jsou nepovinné a slouží pro použití s ble zařízením. *Ch1* představuje UUID pro ble service, *ch2* UUID pro ble charakteristiku z dané service a *ch3* případný ble deskriptor dané charakteristiky. Vrací hodnotu boolean zda proběhlo odeslání v pořádku.

- ***readLine*(ch1: Int?, ch2: Int?)**
  Přečtě (asynchronně) jednu řádku ze zařízení. Parametry *ch1* a *ch2* mají stejný účel jako u předchozí funkce.

- ***readBytes*(bufferSize: Int, ch1: Int?, ch2: Int?)**
  Přečte (asynchronně) data ve formě bytů ze zařízení do bufferu o velikosti *bufferSize*. Parametry *ch1* a *ch2* viz výše.

- ***listenLines*(ch1: Int?, ch2: Int?, ch3: Int?)**
  Začne číst (asynchronně) všechny řádky ze zařízení. Parametry *ch1. ch2* a *ch3* viz výše.

- ***listenBytes*(bufferSize: Int, ch1: Int?, ch2: Int?, ch3: Int?)**
  Začne číst (asynchronně) všechny byty ze zařízení do bufferu o velikosti *bufferSize*. Parametry *ch1, ch2* a *ch3* viz výše.

- ***stopListenLines*(ch1: Int?, ch2: Int?, ch3: Int?)**
  Zruší čtení inicializováno funkcí *listenLines*. *Ch1, ch2* a *ch3* viz výše.

- ***stopListenBytes*(ch1: Int?, ch2: Int?, ch3: Int?)**
  Zruší čtení inicializováno funkcí *listenBytes. Ch1*, *ch2* a *ch3* viz výše.

BaseClient

Abstraktní třída implementující rozhraní *DeviceInterface*, neimplementuje ale žádné z jeho funkcí a nechává to na svých potomcích. Definuje rozhraní *ReadListener*, má atribut jeho typu a funkci *setReadListener*. *ReadListener* má funkce *onNewLineData* při přijetí nové řádky, *onNewByteData* při přijetí nových bytových dat, *onError* při vzniku chyby v připojení a *onConnected* při úspěšném připojení.

TCPClient

Rozšiřuje *BaseClient* a slouží k připojení k zařízením přes Wi-Fi a protokol TCP viz kapitola 10.6.1.

UDPClient

Rozšiřuje *BaseClient* a slouží k připojení k zařízením přes Wi-Fi a protokol UDP viz kapitola 10.6.1.

BTClient

Rozšiřuje *BaseClient* a slouží k připojení k zařízením přes klasický Bluetooth viz kapitola 10.6.2.

BLEClient

Rozšiřuje *BaseClient* a slouží k připojení k zařízením přes Bluetooth Low Energy viz kapitola 10.6.3.

EEDCJavascriptInterface

Třída, která má funkce označené anotací *android.webkit.JavascriptInterface.* Třída bude podrobněji rozebrána v kapitole 10.5.3.

BTDiscoverBroadcastReceiver

Aplikační komponenta Broadcast Receiver, která slouží k detekci nalezení nového Bluetooth zařízení při procesu hledání zařízení v okolí. Je zaregistrován na akce *BluetoothDevice.ACTION_FOUND,* *BluetoothAdapter.ACTION_DISCOVERY_STARTED* a *BluetoothAdapter.DISCOVERY_FINISHED*. Obsahuje dvě LiveData proměnné, jednu se seznamem nalezených zařízení, druhou se stavem hledání.

BTStateChangeReceiver

Aplikační komponenta Broadcast Receiver, která slouží k detekci změny stavu Bluetooth (zapnuto nebo vypnuto). Obsahuje jednu LiveData proměnou se stavem Bluetooth. Je zaregistrován na akci *BluetoothAdapter.ACTION_STATE_CHANGED*.

NetworkStateChangeReceiver

Aplikační komponenta Broadcast Receiver, která slouží k detekci změny stavu sítě. Obsahuje jednu LiveData proměnou se stavem sítě. Je zaregistrován na akce *WifiManager.NETWORK_STATE_CHAGED_ACTION*
a *WifiManager.WIFI_STATE_CHANGE_ACTION.*

10.4 Databáze

Aplikace pro svoji práci využívá sqlite databázi. Pro přístup je využit nástroj *Room*, jak již bylo zmíněno v kapitole 7.3, který umožňuje ukládat data ve formě objektů (ORM). V databázi je pouze jedna tabulka pro entitu *DeviceEntity*.

10.5 Řídící soubor

V této kapitole je popsán proces nahrávání řídícího souboru k zařízení, zobrazení souboru ve webview a komunikace s JavaScript knihovnou *EEDC.js* přes JavaScript rozhraní.

10.5.1 Nahrání souboru

Jak již bylo zmíněno, řídící soubor lze nahrát do aplikace buď v posledním kroku při manuálním přidávání zařízení nebo v editaci již vytvořeného zařízení a existují 3 způsoby jak soubor nahrát. První způsob je ruční nahrání, kdy uživatel vybere soubor z lokálního souborového systému, druhý způsob je stažení souboru z internetu a třetí způsob je stažení souboru přímo ze zařízení. Následuje popis jednotlivých způsobů získávání souboru.

Ruční nahrání souboru

Při ručním nahrání je zobrazen systémový dialog pro vybrání souboru z lokálního souborového systému telefonu pomocí akce *Intent.ACTION_GET_CONTENT* a funkce *startActivityForResult*. Výsledek výběru poté získáme klasicky ve fragmentu v mětodě *onActivityResult*. Pokud je výběr v pořádku zjistíme cestu k souboru. Nejprve se ověří, zda se jedná o soubor zip a pokud ano, extrahuje se do složky aplikace a podsložky /control_files/id_zařízení. To má na starosti třída *EEDCFileManager* a její funkce *downloadLocalFile*, která má parametry zařízení (*DeviceDO*), cesta k souboru a název souboru.

Stažení souboru z internetu

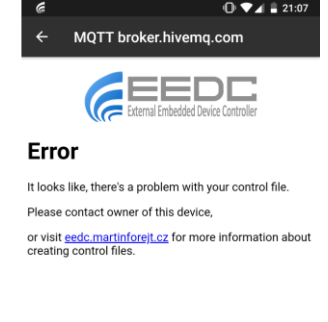
Při stažení souboru z internetu se musí zadat url adresa souboru. Stažení se provede opět ve třídě *EEDCFileManager*, konkrétně ve funkci *downloadFromUrl*, s parametry zařízení (*DeviceDO*) a url adresou. Soubor se stáhne do složky aplikace a podsložky /control_files/id_zařízení, kde se poté extrahuje.

Stažení souboru ze zařízení

Při stažení souboru přímo ze zařízení musí být zařízení k tomu uzpůsobeno. Uživatel zadává tzv. příkaz, což je řetězec, který se pošle do zařízení jako požadavek na stažení souboru. Toto stažení je opět provedeno ve třídě *EEDCFileManager*, konkrétně ve funkci *downloadFromDevice* s parametry zařízení (*DeviceDO*) a příkaz (*String*). V této funkci se vytoří instance třídy *BaseClient*, konrétní implementace závisí na způsobu připojení (*TCPCLient, UDPClient, BTClient* nebo *BLEClient*) a odešle se příkaz pomocí funkce *send*. K příkazu se ještě přidá znak nové řádky (\n). Připojované zařízení je zodpovědné za to, že jako odpověď na tento požadavek nejprve odešle 8 bytů, které obsahují číslo udávající velikost souboru v bytech (v pořadí big-endian). Po těchto 8 bytech ihned začíná posílat samotný byty souboru (počet dále odeslaných bytů musí souhlasit s odeslanou velikostí v prvních 8 bytech). Soubor je opět uložen do složky aplikace a podsložky /contro_files/id_zařízení, kde se poté extrahuje.

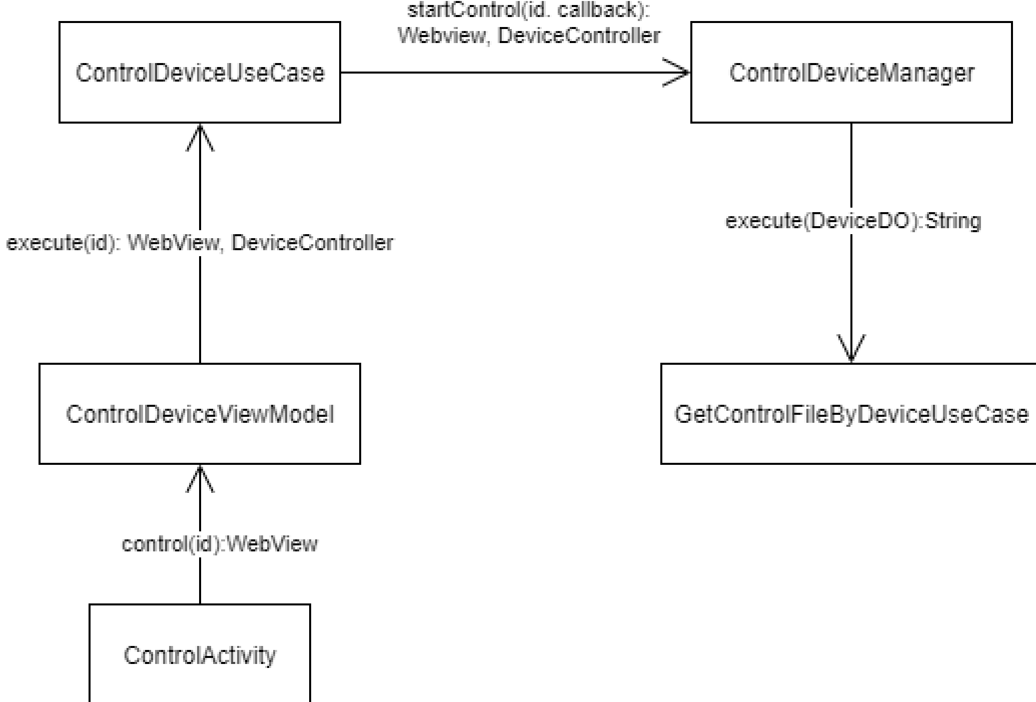
10.5.2 Zobrazení souboru

Jak bylo zmíněno výše, každé zařízení (pokud má již řídící soubor staženo) má uloženo svůj řídící soubor ve své vlastní složce nacházející se v privátní složce zařízení a podsložce /control_files/id_zařízení. Tato složka musí nutně obsahovat soubor *index.html*, který bude zobrazen ve webview. Pokud složka soubor *index.html* neobsahuje, nebo některý ze souborů obsahuje chybu je zobrazena hláška viz obrázek 14.



Obrázek 14 - Chyba řídícího souboru

Způsob zobrazení souboru je následovný. Aktivita *ControlActivity*, zavolá funkci *control* ViewModelu *ControlDeviceViewModel*, jehož návratová hodnota je *WebView* a toto webview je poté zobrazeno v aktivitě. *ControlDeviceViewModel* využije usecase *ControlDeviceUsecase*, který spustí pomocí id zařízení a jako výsledek získá jednak zmiňované webview a také *DeviceController* po pozdější komunikaci s *ControlDeviceManager*. UseCase *ControlDeviceUseCase* pouze volá funkci *startControl* z *ControlDeviceManager*. V této funkci je vytvořeno webview (kterému se přidá JavaScript interface viz dále) a do kterého je načten soubor *index.html* ze řídícího souboru daného zařízení. Cesta k tomuto souboru je získána pomocí usecasu *GetControlFileByDeviceUseCase*.



Obrázek 15 - Postup získání webview

Postup získání (a zobrazení) webview s načteným řídícím souborem je znázorněn na diagramu na obrázku 15, kde je vidět která komponenta volá jakou funkci (včetně parametrů a návratových hodnot) jiné komponenty.

Webview je zobrazeno v aktivitě, ale *ControlDeviceManager* si jeho instanci uchovává u sebe. Pokud je aktivita *ControlActivity* ukončena bez odpojení od zařízení, zůstává připojení dále aktivní a knihovna *EEDC.js* včetně dalších scriptů se zařízením stále komunikuje. Pokud je potom aktivita znovu vytvořena a znovu si vyžádá webview stejným způsobem, již se nevytváří nová instance ale vrátí se uložená instance s již načteným řídícím souborem.

10.5.3 JavaScript rozhraní

JavaScript rozhraní pro komunikace mezi JavaScriptem v řídícím souboru načteném ve webview a Android aplikací, respektive třídou *ControlDeviceManager,* zajišťuje třída *EEDCJavascriptInterface*. Pokud chceme nějakou z metod této třídy zpřístupnit pro JavaScript, označíme ji anotací *android.webkit.JavascriptInterface*. Poté stačí nad webview zavolat funkci *addJavascriptInterface* s instancí třídy *EEDCJavascriptInterface* a jménem objektu pod kterým bude tato instance dostupná v JavaScriptu. My použijeme jako jméno „EEDCAndroid“ viz zdrojový kód 10 v kapitole 9.2.

Anotací máme zajištěno, že naše označené metody budou dostupné v JavaScriptu přes *EEDCAndroid.názevMetody.* Pokud ale chceme naopak z rozhraní (z *ControlDeviceManager*) volat funkce v JavaScriptu, musíme použit funkci webview *loadUrl*, kde url bude ve tvaru „javascript:názevFunkce('parametr1', 'parametr2')“. K tomu bylo vytvořena rozšířená funkce pro webview *callJsFunction* viz zdrojový kód 13.

```
fun WebView.callJsFunction(name: String, params: String) {
    this.loadUrl("javascript:EEDC.$name($params);")
}
```

Zdrojový kód 13 – Funkce callJsFunction
(data/features/web/utils.kt)

Rozhraní má 4 funkce, pro volání funkcí v JavaScriptu a to:

- ***publishReceiveLine*(w: WebView, d: ReceiveData<String>)**
  Volá funkci *_onReceiveLine* z *EEDC.js* pomocí funkce *callJsFunction*. Třída receive data obsahuje atribut data generického typu, v tomto případě String a atributy ch1, ch2 a ch3 pro použití s ble zařízeními. Funkci *callJsFunction* se tedy předá pouze *d.data*.

- ***publishReceiveBytes*(w: Webview, d: ReceiveData<ByteArray>)**
  Volá funkci *_onReceiveBytes* z *EEDC.js* pomocí funkce *callJsFunction*. A předává jí všechny atributy z *d*.

- ***publishOnConnected*(w: WebView)**
  Volá funkci *_onConnected* z *EEDC.js* pomocí funkce *callJsFunction*, bez parametru (prázdný řetězec).

- ***publishStateChanged*(w: WebView, state: DeviceState)**
  Volá funkci *_onStateChange* z *EEDC.js* pomocí funkce *callJsFunction*, s jedním parametrem v podobě atributy *name* parametru *state* výčtového typu *DeviceState*.

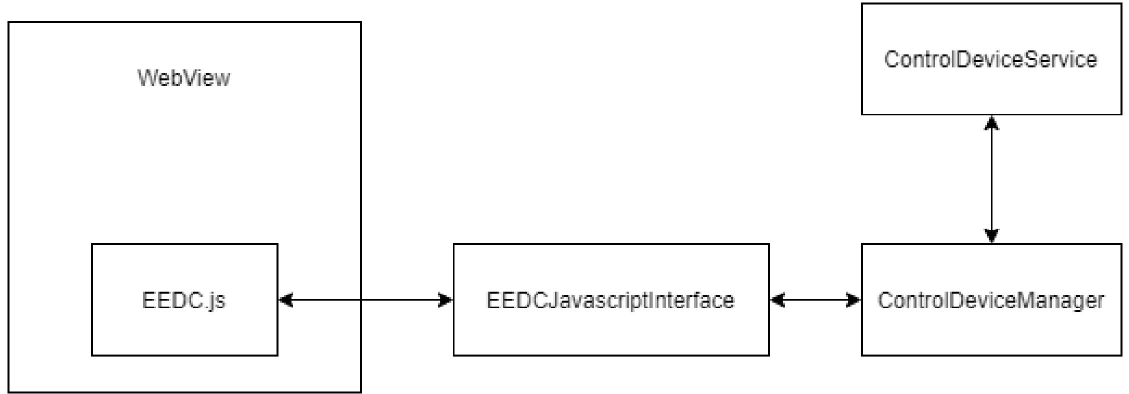
Rozhraní má 18 funkcí, které lze volat z JavaScriptu a jsou označeny anotací *android.webkit.JavascriptInterface*. Všechny funkce obsahují de facto pouze jeden příkaz a to volání funkce z rozhraní *WebExecutor*, jehož instance je předána rozhraní v konstruktoru. Toto rozhraní implementuje *ControlDeviceManager* a při vytváření rozhraní tedy předává v konstruktoru sám sebe. Jedná se o tyto funkce:

- ***initialized*(): Boolean**
  Je voláno z JavaScriptu při inicializaci knihovny *EEDC.js* pomocí funkce *init*. Návratová hodnota je true pokud je zařízení již připojeno, false pokud není.
- ***sendString*(data: String)**
  Odešle *data* do zařízení.
- ***sendBytes*(data: String)**
  Převede *data*, která se posílají jako tzv. hex řetězec na pole bytů a ty předá dál (rozhraní *WebExecutor*) pro odeslání do zařízení.
- ***sendBytesCh*(data: String, ch1: Int, ch2: Int)**
  Stejné jako předchozí funkce, ale s použitím parametrů *ch1* a *ch2* pro použití s ble zařízením. (UUID pro service a charakteristiku).
- ***listenLines*()**
  Začne číst řádky ze zařízení.
- ***listenBytes*(bufferSize: Int)**
  Začne číst byty do bufferu velikosti *bufferSize*.
- ***listenBytesCh*(bufferSize: Int, ch1: Int, ch2: Int,ch3:Int)**
  Stejné jako předchozí funkce, ale s použitím parametrů *ch1, ch2* a *ch3* pro ble.
- ***stoplistenLines*()**
  Přestane číst řádky ze zařízení.
- ***stopListenBytes*()**
  Přestane číst byty.
- ***stopListenBytesCh*(ch1: Int, ch2: Int, ch3: Int)**
  Stejné jako předchozí funkce, ale s použitím parametrů *ch1, ch2* a *ch3* pro ble.
- ***readSingleLine*()**
  Přečte jednu řádku ze zařízení.
- ***readSingleBytes*(bufferSize: Int)**
  Přečte byte data do bufferu o velikosti *bufferSize*.
- ***readSingleByteCh*(bufferSize: Int, ch1: Int, ch2: Int)**
  Stejné jako předchozí funkce, ale s použitím parametrů *ch1* a *ch2* pro ble.
- ***getDeviceInfo*(): String**
  Vrací informace o zařízení (telefon) a připojeném zařízení ve formě json řetězce.
- ***setFullScreen*(fullscreen: Boolean)**
  Podle parametru *fullscreen* zapne nebo vypne fullscreen režim.
- ***isFullScreen*(): Boolean**
  Vrací, zda je fullscreen režim zapnutý nebo vypnutý.

- ***exit*()**
  Odpojí se od zařízení a ukončí aktivitu s webview

10.6 Komunikace se zařízeními

Jak již bylo zmíněno, hlavní funkci při komunikaci se zařízeními mají třídy *ControlDeviceService, ControlDeviceManager* a *EEDCJavascriptInterface*. Manager spustí service a vytvoří interface, které přidá do webview. Pokud je volána funkce z JavaScriptu, interface zavolá odpovídající funkci z rozhraní *WebExecutor* v manageru. Pokud se jedná o funkci pro komunikaci se zařízením (např. *send, listen, read, connect, disconnect*), manager zavolá příslušnou funkci service, který si drží instanci rozhraní *DeviceInterface* popsaného v kapitole 10.3 a pomocí funkce z *DeviceInterface* daný požadavek provede. Při příjmu dat ze zařízení se aktualizuje LiveData proměná *receiveLine* nebo *receiveBytes*, kterou pozoruje manager a tyto data odešle přes rozhraní do JavaScriptu. Komunikace je znázorněna na obrázku 16.



Obrázek 16 - Komunikace se zařízeními

Při vytvoření service *ControlDeviceService* se musí vytvořit instance implementace rozhraní *DeviceInterface*. K tomu slouží pomocný objekt *ClientFactory*, který má pouze jednu funkci *createClient*, která dle předaného zařízení (*DeviceDO*) vytvoří jednu ze 4 implementací rozhraní *ControlDeviceService*, které jsou popsány níže.

10.6.1 Wi-Fi TCP

TCPClient k připojení pomocí protokolu TCP používá třídu *java.net.Socket* z kterého získá vstupní a výstupní stream pro čtení a zápis z a do zařízení. Socket vytvoří pomocí IP adresy (názvu hostitele) a portu, které *TCPClient* dostane ve svém konstruktoru.

Zjednodušený kód třídy *TCPClient* je vidět ve zdrojovém kódu 14, který více přiblíží práci této třídy. Ve funkci *open* (z rozhraní *DeviceInterface*, které implementuje) se vytvoří socket, vstupní a výstupní stream a pokud vše proběhne v pořádku vrátí se *OpenStatus.CONNECTED* jinak *OpenStatus.ERROR*.

Ve funkci *send* pro odeslání řetězce do zařízení, se řetězec *data* převede do pole bytů (funkce *toByteArray* a odešle se do zařízení pomocí výstupního streamu a jeho funkce *write*. Pokud při čtení nastane nějaká chyba (značící problém s připojení), volá se funkce (ze svého předka *BaseClient*) *onError*.

Při čtení řádek pomocí funkce *listenLines* se ze vstupního streamu vytvoří *BufferedReader* a řádky se čtou pomocí funkce *readLine*. Při přečtení nové řádky se volá funkce (z předka) *onNewLineData*, při chybě *onError*.

```
override fun open(): DeviceInterface.OpenStatus {
    socket = try {
        Socket(host, port)
    } catch (e: IOException) {
        return DeviceInterface.OpenStatus.ERROR
    }
    dataIn = socket!!.getInputStream()
    dataOut = socket!!.getOutputStream()
    return DeviceInterface.OpenStatus.CONNECTED
}

override fun send(data: String): Boolean {
    dataOut?.let {
        return try {
            it.write(data.toByteArray())
            it.flush()
            true
        } catch (e: IOException) {
            onError()
            false
        }
    }
    return false
}

override fun listenLines() {
    dataIn?.let {
        val reader = BufferedReader(it.reader())
        try {
            while (true) {
                val l = reader.readLine() ?: break
                onNewLineData(l)
            }
        } catch (e: IOException) {
            onError()
        }
    }
}
```

Zdrojový kód 14 – TCPClient
(data/features/control/client/TCPClient.kt)

10.6.2 Wi-Fi UDP

Třída *UDPClient* slouží k připojení pomocí protokolu UDP a pro svoji práci využívá třídu *java.net.DatagramSocket*, které při vytváření předává instanci třídy *InetSocketAddress*, kterou vytvoří opět pomocí IP adresy a portu, které *UDPClient* dostává v konstruktoru.

Funkčnost třídy *UDPClient* je vidět na ukázce ve zdrojovém kódu 15. Ve funkci *open* vytvoří datagramSocket. Při odesílání dat viz funkce *send*, se vytvoří instance třídy *DatagramPacket* z pole bytů a velikosti tohoto pole a pomocí funkce *send* z datagramSocketu se tento paket odešle. Při chybě se volá funkce *onError*.

Při čtení dat ve funkci *listenBytes* se nejprve vytvoří buffer (pole bytů) o velikosti *bufferSize*. Poté se vytvoří datagramSocket, úplně stejně jako v předchozí funkci a přijmou se data pomocí funkce *receive* z datagramSocketu. Přijatá data se předají funkci *onNewByteData*.

UDPClient neposkytuje implementaci pro funkce pro čtení řádků (*readLine* a *listenLines*).

```
override fun open(): DeviceInterface.OpenStatus {
    socket = try {
        DatagramSocket(address)
    } catch (e: IOException) {
        return DeviceInterface.OpenStatus.ERROR
    }
    return DeviceInterface.OpenStatus.CONNECTED
}

override fun send(data: String): Boolean {
    val byteData = data.toByteArray()
    val dpSend = DatagramPacket(byteData, byteData.size)
    return try {
        socket?.send(dpSend) ?: return false
        true
    } catch (e: IOException) {
        onError()
        false
    }
}

override fun listenBytes(bufferSize: Int) {
    val data = ByteArray(bufferSize)
    try {
        while (true) {
            val dpReceive = DatagramPacket(data, bufferSize)
            socket?.receive(dpReceive) ?: break
            onNewByteData(data)
        }
    } catch (e: IOException) {
        onError()
    }
}
```

Zdrojový kód 15 – UDPClient
(data/features/control/client/UDPClient.kt)

10.6.3 Bluetooth

Třída *BTClient* slouží k připojení k zařízením pomocí klasického Bluetooth. Implementace je hodně podobná třídě *TCPClient*, jen místo třídy *java.net.Socket* se použije *android.bluetooth.BluetoothSocket.* Nejprve je vytvořena instance *BluetoothDevice* pomocí funkce *getRemoteDevice* z *BluetoothAdapter*, které předáme mac adresu zařízení. Poté se vytvoří socket a záleží, zda má být připojení zabezpečené. Pokud ano, použijeme funkci *createRfcommSocketToServiceRecord*, které předáme UUID Bluetooth service, kde vždy použijeme UUID pro service SerialPort, který má definové UUID viz *SPP_UUID* ve zdrojovém kódu 16. Při použití zabezpečeného připojení musí být zařízení spárována, pokud nejsou, je proces párování zahájen automaticky při prvním pokusu o připojení. Pokud chceme použít nezabezpečené připojení použijeme funkci *createInsecureRfcommSocketToServiceRecord* opět se stejným UUID. Ze socketu, který nám tyto 2 funkce vytvoří získáme vstupní a výstupní stream a ostatní věci jsou stejné jako v *TCPClient*.

```kotlin
private const val SPP_UUID = "00001101-0000-1000-8000-00805F9B34FB"

override fun open(): DeviceInterface.OpenStatus {
    val adapter = BluetoothAdapter.getDefaultAdapter()
    val device = adapter.getRemoteDevice(mac)
    socket = try {
        if (secure) {
           device.createRfcommSocketToServiceRecord(
                UUID.fromString(SPP_UUID)
           ).also { it.connect() }
        } else {
            device.createInsecureRfcommSocketToServiceRecord(
                UUID.fromString(SPP_UUID)
            ).also { it.connect() }
        }
    } catch (e: IOException) {
        return DeviceInterface.OpenStatus.ERROR
    }
    return try {
        dataIn = socket!!.inputStream
        dataOut = socket!!.outputStream
        DeviceInterface.OpenStatus.CONNECTED
    } catch (e: IOException) {
        DeviceInterface.OpenStatus.ERROR
    }
}
```

Zdrojový kód 16 – BTClient
(data/features/control/client/BTClient.kt)

10.6.4 Bluetooth Low Energy

Pro komunikaci přes Bluetooth Low Energy slouží třída *BLEClient* viz zdrojový kód 17. Uvnitř funkce *open* se vytvoří *BluetoothDevice* stejně jako ve třídě *BTClient*. Z toho se potom získá instance třídy *BluetoothGatt*, pomocí které se bude komunikovat se zařízením. Při získávání gattu musíme předat *BluetoothGattCallback*, přes který (jeho funkce) budeme informování při změně stavu připojení, vyhledání service, které zařízení poskytuje, přečtení nebo změně charakteristiky. Funkce *open* vrací *OpenStatus.CONNECTING*, značící, že stále probíhá připojování. Poté co je volána funkce *onConnectionChange* v gatt callbacku, volá podle stavu připojení buď funkci *onConnected* nebo *onError*.

Jelikož v jednu chvíli může probíhat pouze jeden zápis do zařízení (není možné např. zároveň zapsání do charakteristiky a do deskriptoru), je vytvořena pomocná třída *BleRequest* obsahující informace o požadované akci a třída *BLEClient* obsahuje frontu těchto requestů a postupně je zpracovává. Při volání funkcí jako je *send, readBytes, listenBytes* se nejprve přidá požadavek do fronty a poté se zavolá funkce *nextRequest* která spustí další request z fronty, až to bude možné.

Ve zdrojovém kódu 17 je vidět ukázka při čtení hodnoty z charakteristiky pomocí funkce *readBytes* z *DeviceInterface*. Při volání této funkce se vytvoří request (*BleRequest*) a přidá se do fronty. Po výběru tohoto requestu z fronty se volá funkce *doRead*, která získá service, z něho charakteristiku a nad gattem volá *readCharacteristic*. Výsledek operace získáme v callbacku ve funkci *onCharacteristicRead* odkud zavoláme funkci *onNewByteData* a vyvoláme další request z fronty pomocí funkce *nextRequest*.

BLEClient neposkytuje implementace pro funkce pro čtení řádků (*readLine* a *listenLines*).

```
override fun open(): DeviceInterface.OpenStatus {
    val adapter = BluetoothAdapter.getDefaultAdapter()
    val device = adapter.getRemoteDevice(mac)
    bluetoothGatt = device.connectGatt(context, true, gattCallback)
    return DeviceInterface.OpenStatus.CONNECTING
}

private fun doRead(request: BleRequest) {
    val service = bluetoothGatt?.getService(request.ch1.toUUID())
    val characteristic = service?.getCharacteristic(request.ch2.toUUID())
    val res = characteristic?.let { bluetoothGatt?.readCharacteristic(it) }
}

private val gattCallback = object : BluetoothGattCallback() {
   override fun onConnectionStateChange(g:BluetoothGatt, s:Int, state:Int){
        super.onConnectionStateChange(g, s, state)
        when (state) {
            BluetoothProfile.STATE_CONNECTED -> {
                onConnected()
                bluetoothGatt?.discoverServices()
            }
            BluetoothProfile.STATE_DISCONNECTED -> {
                onError()
            }
        }
    }

override fun onCharacteristicRead(g: BluetoothGatt, ch:
BluetoothGattCharacteristic, status: Int) {
        super.onCharacteristicRead(g, ch, status)
        onNewByteData(
            chc.value,
            ch.service.uuid.toInt(),
            ch.uuid.toInt()
        )
        nextRequest()
    }
}
```

Zdrojový kód 17 – BLEClient
(data/features/control/client/BLEClient.kt)

11 Ověření funkčnosti a testování

V této kapitole budou popsány způsoby testování aplikace a testovací aplikace pro každý typ připojení (Wi-Fi, Bluetooth a Bluetooth Low Energy).

11.1 Způsoby testování

Testování je nedílnou součástí procesu vývoje (nejen) Android aplikací. Průběžným spouštěním testů můžeme před vydáním aplikace ověřit její správnost, funkční chování a použitelnost [13]. Testovat můžeme jednak psaním a spouštěním automatizovaných testů a jednak ručním testováním dle testovacích scénářů. Android framework nám umožňuje psát 3 druhy automatizovaných testů a to jednotkové testy (Unit testy), které testují jednu konkrétní třídu, integrační testy, které testují interakci mezi jednotlivými komponenty (např. fragment a ViewModel) a testy uživatelského rozhraní (UI testy).

Pro testování naší aplikace byly použity tyto nástroje:

- **JUnit 4** – framework, pro psaní jednotkových testů, ale využití najde i v dalších typech testů
- **Espresso** – framework pro psaní ui testů, obsahuje funkce pro interakci s prvky uživatelského rozhraní.
- **Mockk** – knihovna pro tvorbu tzv. mocků[15] pro Kotlin.
- Další pomocné nástroje z ***androidx.test*** z Android Jetpack

11.2 Jednotkové testy

Jednotkové testy se nacházejí v adresáři *test*, konkrétně *název_modulu/src/test/java.* Jednotkové testy, jsou tzv. lokální testy, které pro svůj běh nepotřebují fyzické zařízení (nebo emulátor). Spuštění všech testů provedeme pomocí gradle tasku *test*, který spustí jednotkové testy v celém projektu a výsledek uloží ve formátu HTML do složky: *název_modulu/build/reports/test*

Pomocí jednotkových testů byly testovány třídy nezávislé na Android frameworku. Příkladem může být test třídy *DecodeStringToDeviceUseCase*, což je usecase pro převod řetězce (např. při načtení QR kódu) do zařízení (DeviceDO), který je otestován ve třídě *DecodeStringToDeviceUseCaseTest* viz zdrojový kód 18. Kód v metodě *before*, která je označena anotací *org.junit.Before* se provede před spuštěním testovacích metod označených anotací *org.junit.Test*. V této metodě se vytvoří instance třídy *DecodeStringToDeviceUseCase*.

Následují jednotlivé testovací metody. První *executeSyncWifi* otestuje dva validní řetězce, zda je z nich vytvořeno Wi-Fi zařízení pomocí funkce *assertNotNull*, která, pokud je jejím parametrem hodnota null, vyvolá výjimku *AssertionError* a test neprojde. Druhá metoda *executeSyncBle* je totožná s předchozí, ale pro řetězce vytvářející Bluetooth zařízení. Třetí a poslední metoda *executeSyncError* testuje nevalidní řetězce pomocí funkce *assertNull*.

[15] Mock zastupuje reálný objekt a umožňuje simulovat jeho chování

```kotlin
class DecodeStringToDeviceUseCaseTest {

    private lateinit var usecase : DecodeStringToDeviceUseCase

    @Before
    fun setUp() {
        usecase = DecodeStringToDeviceUseCase()
    }

    @Test
    fun executeSyncWifi() {
        runBlocking {
            assertNotNull(usecase.executeSync(
                "0;go;tcp://89.221.216.98;1111;6;1;file;_"
            ))
            assertNotNull(usecase.executeSync(
                "0;go;udp://89.221.216.98;1111;6;2;url;_"
           ))
        }
   }

   @Test
   fun executeSyncBle() {
       runBlocking {
           assertNotNull(usecase.executeSync("1;go;mac;11;6;1;file;_"))
           assertNotNull(usecase.executeSync("1;go;mac;10;6;1;file;_"))
           assertNotNull(usecase.executeSync("1;go;mac;01;6;1;file;_"))
           assertNotNull(usecase.executeSync("1;go;mac;00;6;1;file;_"))
       }
   }

   @Test
   fun executeSyncError() {
       runBlocking {
           assertNull(usecase.executeSync(""))
           assertNull(usecase.executeSync("a.sd"))
           assertNull(usecase.executeSync(
               "0;go;http://89.221.216.98;1111;6;2;url;_"
           ))
           assertNull(usecase.executeSync("go;mac;00;6;1;file;_"))
       }
   }
}
```

Zdrojový kód 18 – DecodeStringToDeviceUseCaseTest
(domain/features/device/usecase/DecodeStringToDeviceUseCaseTest.kt)

11.3 Integrační a UI testy

Integrační testy a testy uživatelského rozhraní se nacházejí v adresáři *androidTest*, konkrétně *název_modulu/src/androidTest/java* a potřebují pro svůj běh prostředí android reálného zařízení nebo emulátoru. Všechny testy byly spouštěny na emulátoru s Androidem ve verzi 10.0. Spuštění testů provedeme pomocí gradle tasku *connectedDebugAndroidTest*, který spustí všechny testy ze složky *androidTest* na připojeném emulátoru. Výsledky testů se uloží do složky *název_modulu/build/repors/androidTests*.

```kotlin
class DevicesListFragmentTest {
    @get:Rule
    val activityRule: ActivityTestRule<MainActivity> = ActivityTestRule(
         MainActivity::class.java, true, false)
   @RelaxedMockK
   private lateinit var mockViewModel: DevicesViewModel
   @RelaxedMockK
   private lateinit var deviceVO: DeviceVO
   private val mockedDevices = MutableLiveData<List<DeviceVO>>()

   @Before
   fun setUp() {
       MockKAnnotations.init(this)
       every { mockViewModel.devices } returns mockedDevices
       loadKoinModules(module(override = true) {
           viewModel(override = true) { mockViewModel } })
       activityRule.launchActivity(Intent())
   }

   @Test
   fun recyclerTest() {
       val nav = TestNavHostController(
               ApplicationProvider.getApplicationContext())
       nav.setGraph(R.navigation.main_nav_graph)

       val s = launchFragmentInContainer<DevicesListFragment>(...)
       s.onFragment {Navigation.setViewNavController(it.requireView(), nav)}

       onView(withId(R.id.recycler))
           .check(RecyclerViewItemCountAssertion(0))
       mockedDevices.postValue(listOf(deviceVO))
       onView(withId(R.id.recycler))
           .check(RecyclerViewItemCountAssertion(1))

       onView(withId(R.id.recycler)).perform(
       RecyclerViewActions.actionOnItemAtPosition<ViewHolder>(0, click()))

       assertEquals(R.id.deviceDetailFragment,nav.currentDestination?.id)
    }

}
```

Zdrojový kód 19 – DevicesListFragmentTest
(presentation/features/device/view/DevicesListFragmentTest.kt)

Příklad integračního a ui testu je vidět ve zdrojovém kódu 19 výše, který testuje fragment *DevicesListFragment* a jeho interakci s ViewModelem *DevicesViewModel*. V uvedeném příkladu je vytvořeno ActivityTestRule pro spuštění aktivity *MainActivity*, ve které bude testovaný fragment spuštěn. Dále je vytvořen mock pro *DevicesViewModel* a *DeviceVO* které jsou označeny anotací *RelaxedMockK*, která vytvoří tvz. relaxed mock, který vrací nějakou hodnotu pro všechny funkce objektu.

V metodě *setUp* se inicializuje knihovna *mockK* pomocí příkazu *MockKAnnotations.init*. Pro mock *mockViewModel* nastavíme, aby jeho atribut *devices* vždy vracel námi vytvořený objekt *mockedDevices*. Poté ještě musíme spustit Koin a předat mu náš vytvořený ViewModel, aby ho mohl fragment získat pomocí vkládání závislostí. Nakonec spustíme aktivitu.

V metodě *recyclerTest* budeme testovat chování recyleru, který představuje seznam zařízení na úvodní obrazovce aplikace se seznamem zařízení. Abychom mohli také otestovat navigaci musíme vytvořit *TestNavHostController*. Poté spustíme fragment a spojíme ho s navigací. Následují vlastní testy pomocí funkcí espressa *onView, withId, check, perform* a dalších. Nejprve otestujeme, že recycler je prázdný. Poté do našeho mocku *mockViewModel* přidáme mock zařízení pomocí našeho seznamu zařízení a otestujeme, zda se přidal do recycleru, který by měl nyní obsahovat jednu položku. Poté provedeme kliknutí na první položku recycleru a testujeme, zda se změnila destinace navigace a došlo tedy k přechodu na obrazovku detailu zařízení. Tím jsem otestovali jednak komunikaci mezi fragmentem a ViewModelem a také uživatelské rozhraní fragmentu, konkrétně jeho recycler.

Při psaní integračních testů a testů uživatelského rozhraní nastal problém u fragmentů využívající databinding u kterých nebyl načten obsah ihned po vytvoření fragmentu. Tento problém částečně vyřešila pomocná třída *DataBindingIdlingResourceRule* (presentation/androidTest/core), ale vytvořené testy i přesto často selžou s chybou neexistence prvku uživatelského rozhraní nebo nemožnosti kliknout na neexistující prvek i přesto, že tento prvek v rozhraní je. Jindy tyto testy projdou v pořádku. Dalším problémem u těchto testů je, když je vytvořen dialog pro potvrzení udělení oprávnění nebo např. zapnutí Bluetooth. Tyto problémy se nepodařilo jednoduše a v rozumném čase vyřešit, proto většina testování probíhala ručně viz dále.

11.4 Ruční testování

Aplikace byla z větší části testována ručně, kdy se opakovaně během vývoje testovali všechny funkce aplikace, včetně ověření reakce na nevalidní vstupy. Dále bylo např. nutné otestovat konzistentnost dat aplikačních komponent (aktivita nebo fragment), které byly ukončeny a znovu spuštěny. Během testování byla odhalena spousta chyb, které byly opraveny. Testování probíhalo na následujících zařízeních:

- **Emulátor 1** – Android 4.4
- **Emulátor 2** – Android 10.0
- **Nexus 5X** – Android 8.1

Tyto 3 zařízení byly vybrány proto, že Android 4.4 je minimální podporovaná verze, Android 10.0 je aktuálně nejnovější produkční verze a Nexus 5X jako zástupce fyzických zařízení, který se v některých případech mohou chovat jinak než emulátory.

11.5 Detekce chyb v produkci

Je samozřejmé, že se nepodaří odhalit všechny chyby v aplikaci. Jednak kvůli chybám během samotného testování (špatně volené testovací scénáře), ale také z důvodů velkého množství podporovaných zařízení a verzí Androidu, na kterých se může aplikace chovat jinak než na těch otestovaných. Z toho důvodu je nutné detekovat a logovat všechny chyby, které nastanou v produkční verzi tak, aby je bylo možné co nejdříve opravit. K tomu byl použit nástroj Sentry[16], který poskytuje monitorování chyb aplikace v reálném čase. U všech detekovaných chyb poskytuje spoustu informací jako je výpis chyby, verze aplikace, typ zařízení, verze Androidu a další. Jednotlivé chyby lze spravovat, označovat jako opravené, ignorované, atd. Zprovoznění Sentry je jednoduché, stačí si založit bezplatný účet na sentry.io, vložit závislosti na knihovnu pomocí gradlu a Sentry spustit v aplikační třídě (*EEDCApplication*) viz zdrojový kód 20. Jelikož byla aplikace publikována na Google Play,

[16] Sentry - https://sentry.io

které nám nabízí také možnost zobrazení detekovaných chyb, můžeme využít i tuto alternativu. Oproti Sentry, je ale méně přehledná a neposkytuje tolik možností, proto jsme se rozhodli právě pro Sentry.

```
val sentryDsn = getString(R.string.sentry_dsn)
Sentry.init(sentryDsn)
```

Zdrojový kód 20 – Sentry
(presentation/features/device/view/DevicesListFragmentTest.kt)

11.6 Testovací aplikace

Pro každý typ připojení (Wi-Fi, Bluetooth a Bluetooth Low Energy) byla vytvořena jedna testovací aplikace, respektive testovací řídící soubor. Níže jsou popsány všechny vytvořené testovací řídící soubory a ukázky obrazovek přímo z aplikace.

11.6.1 Wi-Fi

Pro otestování připojení přes Wi-Fi (nebo i mobilní internet v tomto případě), byl vytvořen jednoduchý MQTT klient. Jelikož MQTT protokol pracuje nad protokolem TCP, je možné ho použít i v námi vytvořené aplikaci. Pro parsování a generování MQTT paketů, byla použita volně dostupná knihovna pro JavaScript *mqtt-packet* (https://github.com/mqttjs/mqtt-packet).

```
<!DOCTYPE html>
<html lang="en">
<head>
    ...
    <script src="eedc-core.js" type="module"></script>
    <script src="eedc-mqtt.js" type="module"></script>
    <script src="custom.js" type="module"></script>
</head>
<body>
<h1>MQTT test</h1>
<button id="btn_connect">connect</button>
<button id="btn_disconnect">disconnect</button>
<button id="btn_subscribe">subscribe</button>
<button id="btn_unsubscribe">unsubscribe</button>
<button id="btn_publish">publish</button>
<br>
<br>
<div>
        <textarea id="receive" name="receive" ... ></textarea>
</div>
</body>
</html>
```

Zdrojový kód 21 - MQTT klient index.html

Ve zdrojovém kódu 21 je vidět soubor *index.html* z řídícího souboru pro tuto aplikaci. V hlavičce jsou importované 3 JavaScript soubory (z archivu řídícího soubor): *eedc-core.js* dodaný soubor s knihovnou *EEDC.js*, *eedc-mqtt.js*, který obsahuje 2 pomocné funkce pro práci s knihovnou *mqtt-packet* a *custom.js*, jehož zjednodušená implementace je vidět ve zdrojovém kódu 22 níže. V samotném těle HTML souboru (body) je 5 tlačítek a jedno textové pole. První tlačítko provede MQTT příkaz connect, druhé disconnect. Třetí tlačítko provede MQTT příkaz pro odebírání tzv. tématu (topic) s maskou „eedc/#“. Čtvrté tlačítko

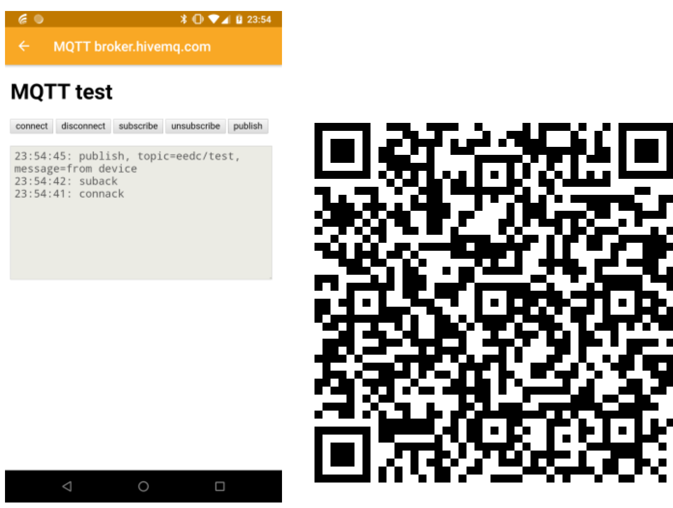
provede odhlášení od tohoto tématu a páté tlačítko odešle zprávu „from device“ do tématu „eedc/test“. Textové pole slouží k logování všech přijatých zpráv, kterými jsou jednak potvrzení odeslaných požadavků a jednak přijaté zprávy z odebíraného tématu.

Soubor *custom.js* (zdrojový kód 22) nejprve importuje knihovnu *EEDC.js* a pomocný soubor *eedc-mqtt.js*. Ve funkci přiřazené k window.onload se provede inicializace knihovny *EEDC.js* a uvnitř předaného callbacku se začnou poslouchat přijaté byty s bufferem o velikost 200 bytů. Dále se zaregistruje callback k události *new_bytes* ve kterém se z přijatých dat vytvoří MQTT paket a ten se zpracuje (zobrazí v textovém poli) ve funkci *parseCallback*. Dvě ukázky funkcí *connect* a *subscribe* se volají po kliknutí na příslušná tlačítka. Uvnitř nich se vytvoří paket, funkcí *MQTT.generatePacket* se převede na byty a odešle funkcí *EEDC.sendBytes*.

```
import EEDC from './eedc-core.js';
import MQTT from './eedc-mqtt.js';

let parseCallback = function (data) {...};

window.onload = function () {
    EEDC.init(function () {
        EEDC.listenBytes(200);
    });
    EEDC.on("new_bytes", function (data) {
        MQTT.parsePacket(data.data, parseCallback);
    });
};

function connect() {
    let data = {
        cmd: 'connect',
        protocolVersion: 4,
        clientId: 'eedc',
        keepAlive: 60,
    };
    let packet = MQTT.generatePacket(data);
    EEDC.sendBytes(packet);
}

function subscribe() {
    let data = {
        cmd: 'subscribe',
        subscriptions: [{
            topic: 'eedc/#',
            qos: 0,
        }]
    };
    EEDC.sendBytes(MQTT.generatePacket(data));
}
```

Zdrojový kód 22 - MQTT klient - custom.js

Ukázka obrazovky této aplikace je vidět na obrázku 17. V textovém poli jsou vidět tři přijaté zprávy. První *connack* jako potvrzení připojení k MQTT serveru, druhý *suback* jako potvrzení odebírání tématu a třetí *publish* přijetí nové zprávy z odebíraného tématu, v tomto případě odeslané přímo ze zařízení pomocí posledního tlačítka publish. Na obrázku 17 je také QR kód pro importování tohoto příkladu přímo do aplikace. Před prvním připojení je samozřejmě nutné přejít nejprve do nastavení zařízení a stáhnout řídící soubor. Ten v tomto případě bude již nastaven na stažení z vyplněné url adresy.



Obrázek 17 - MQTT klient

11.6.2 Bluetooth

Testovací aplikace pro Bluetooth je jednodušší. HTML souboru obsahuje dvě vstupní pole pro nastavení dvou textových hodnot a u každé z nich tlačítko „set“ pro odeslání hodnoty do zařízení a „get“ pro načtení hodnoty ze zařízení. Dále je zde velké textové pole do kterého se načtou informace o zařízení pomocí funkce *EEDC.getDeviceInfo* a tlačítko pro zapnutí a vypnutí fullscreen režimu.

Implementace souboru *custom.js*, který je do HTML souboru přidán stejně jako v předchozí aplikaci, je vidět ve zdrojovém kódu 23. V callbacku na událost *state_change* se do textového pole vloží informace o zařízení pomocí funkce *EEDC.getDeviceInfo*. Po přijetí nové řádky v callbacku na událost *new_line* se přijatý řetězec rozdělí na dvě části spojené středníkem. První část udává o jakou hodnotu se jedná (1 nebo 2) a druhá část je samotná hodnota, která se vloží do příslušného pole. Další je příkaz *EEDC.listenLines*, kterým se začnou poslouchat přijaté řádky.

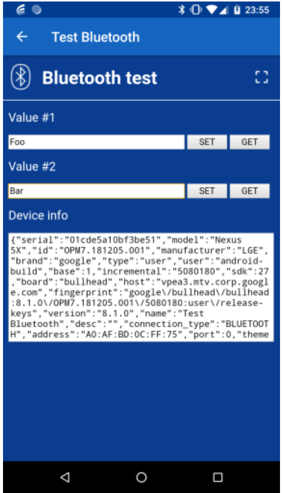
Po kliknutí na tlačítko „set“ se zavolá funkce *set1* respektive *set2*, které odešlou řetězec ve formátu „set;1nebo2;hodnota“. Při kliknutí na tlačítko „get“ se volá funkce *get1* a *get2*, které odešlou řetězce „get;1“ a „get;2“.

Pro tuto aplikaci byl v Javě naprogramován Bluetooth server, který si uchovává hodnoty těchto 2 hodnot a při zavolání požadavku „set“ si danou hodnotu přepíše, pří zavolání požadavku „get“ ji odešle do zařízení. Dále umí tento server reagovat na přijatý řetězec „file“ po kterém začne vysílat řídící soubor. Je tak možné stáhnout řídící soubor přímo z tohoto serveru.

Ukázka z této aplikace je vidět na obrázku 18.

```
import EEDC from './eedc-core.js';

window.onload = function () {
    EEDC.init(function () {
        EEDC.on("state_change", function (data) {
            if (data === 'CONNECTED') {
                document.getElementById('info').value =
                    EEDC.getDeviceInfo();
            }
        });
        EEDC.on("new_line", function (line) {
            let parts = line.split(';');
            if (parts[0] === '1') {
                document.getElementById('val_1').value = parts[1];
            } else if (parts[0] === '2') {
                document.getElementById('val_2').value = parts[1];
            }
        });
        EEDC.listenLines();
    });
};

function set1() {
    EEDC.sendString('set;1;'+document.getElementById('val_1').value
        +'\n')
}

function get1() {
    EEDC.sendString('get;1\n')
}

function fullscreen() {
    EEDC.setFullScreen(!EEDC.isFullScreen())
}
```

Zdrojový kód 23 - Bluetooth custom.js



Obrázek 18 - Bluetooth ukázka

11.6.3 Bluetooth Low Energy

Pro otestování Bluetooth Low Energy byl vytvořen ovladač cyklistického trenažéru (testováno s Tacx Flux 2 Smart), který podporuje ble service Fitness Machine (org.bluetooth.service.fitness_machine, uuid=1826). Tento service obsahuje charakteristiku Indoor Bike Data (org.bluetoot.characteristic.indoor_bike_data, uuid=2AD2). Tato charakteristika nabízí mimo jiné aktuální výkon, kadenci a rychlost, které budeme v naší aplikaci zobrazovat. Dále service obsahuje charakteristiku Fitness Machine Control Point (org.bluetooth.characteristic.fitness_machine_control_point, uuid=2AD9), pomocí které lze nastavit cílový výkon, který bude trenažér držet nezávisle na kadenci.

HTML soubor řídícího souboru je jednoduchý, obsahuje textové prvky se zobrazovanými údaji (výkon, kadence, rychlost), aktuální cílový výkon, tlačítka + a -, které tento výkon upraví a tlačítka pro spuštění a zastavení časovače, který je na trenažéru nezávislí a je řešen pouze v rámci JavaScriptu.

```
import EEDC from './eedc-core.js';

const OP_TARGET = 0x05;

window.onload = function () {
    EEDC.init(function () {
        EEDC.on("new_bytes", function (data) {
            let speedVal = (data.data[2] + data.data[3] * 256)/100.0;
            let cadenceVal = (data.data[4] + data.data[5] * 256)*0.5;
            let powerVal = (data.data[6] + data.data[7] * 256);
            speed.innerHTML = speedVal;
            cadence.innerHTML = cadenceVal;
            power.innerHTML = powerVal;
        });
        EEDC.listenBytes(1, 0x1826, 0x2AD2, 0x2902);
    });
};

function inc() {
    targetValue += 10;
    target.innerHTML = targetValue;
    let val = getInt16Bytes(targetValue);
    let data = new Uint8Array([OP_TARGET, val[1], val[0]]);
    EEDC.sendBytes(data, 0x1826, 0x2AD9);
}
```

Zdrojový kód 24 - BLE custom.js

Ve zdrojovém kódu 24 je vidět část souboru *custom.js* z řídícího souboru této aplikace. Na začátku souboru je opět importována knihovna *EEDC.js*, dále je definována hodnota požadavku, který se bude posílat do zařízení při změně cílového výkonu. V callbacku funkce *EEDC.init* se nastaví callback pro událost *new_bytes*, ve kterém se nastaví text 3 zobrazovaným hodnotám (výkon, kadence a rychlost). Pomocí *EEDC.listenBytes* se nastaví notifikace u service deskriptoru s uuid 2902 u charakteristiky Indoor Bike Data v service Fitness Machine.

Po kliknutí na tlačítko + se volá funkce *inc*, která navýší aktuální cílový výkon o 10 watů převede ho na šestnácti bytovou hodnotu pomocí pomocné funkce *getInt16Bytes* a pomocí funkce *EEDC.sendBytes* zapíše tuto hodnotu uvozenou hodnotou OP_TARGET do charakteristiky Fitness Machine Control Point, tím se odpor trenažéru zvýší o 10 watů.

Ukázka z této aplikace je vidět na obrázku 19. Specifikace charakteristiky Fitness Machine Control Point je dostupná na https://www.bluetooth.com/specifications/gatt

Obrázek 19 - Trenažér

11.7 Publikace aplikace

Aplikace byla publikována ke stažení zdarma na Google Play:
https://play.google.com/store/apps/details?id=cz.martinforejt.eedc

12 Závěr

V rámci této bakalářské práce byla prozkoumána problematika vzdáleného zobrazování dat na mobilních telefonech a byla navržena a implementována aplikace pro operační systém Android, využívající HTML stránek a JavaScriptu ke vzdálené komunikaci s vestavěnými zařízeními přes Wi-Fi, Bluetooth nebo Bluetooth Low Energy.

Ve vytvořené aplikaci lze přidat více zařízení a to ručně nebo naskenováním QR kódu a ke každému zařízení musí být nahrán tzv. řídící soubor, který obsahuje soubor *index.html* a soubor *eedc-core.js*, představující knihovnu *EEDC.js* vytvořenou v rámci této práce. HTML stránka řídícího souboru je zobrazena v aplikaci a pomocí JavaScript rozhraní je realizována komunikace mezi knihovnou *EEDC.js* a prostředím Androidu, které má na starosti vlastní komunikaci s vestavěným zařízením.

Pro otestování aplikace byly vytvořeny 3 testovací aplikace pro každý typ připojení. Pro připojení přes Wi-Fi (TCP protokol) byl vytvořen jednoduchý MQTT klient, pro připojení přes klasický Bluetooth byl implementován testovací server v Javě a pro připojení přes Bluetooth Low Energy byl vytvořen ovladač cyklistického trenažéru. Tyto testovací aplikace vznikli již nezávisle na Android aplikaci, jen za pomoci vytvoření řídícího souboru (HTML a JavaScript) a tímto způsobem je možné komunikovat téměř s jakýmkoliv zařízením.

K aplikaci byla vytvořena webová dokumentace poskytující návod jak k ovládání vlastní aplikace, tak k vytváření vlastních řídících souborů a aplikace byla publikována ke stažení na Google Play.

Oproti uvedeným existujícím řešením, které se většinou specializují jen na jeden typ připojení, podporuje naše aplikace 3 druhy připojení (Wi-Fi, Bluetooth a Bluetooth Low Energy). Výrazně se liší také v uživatelském rozhraní. Uvedené aplikace buď neměli možnost upravovat vzhled uživatelského rozhraní, nebo mohli použít omezené množství již připravený widgetů. Oproti tomu naše aplikace umožňuje tvorbu vlastního uživatelského rozhraní pomocí webových technologií, která nabízí více možností, ale je časově (a technologicky) náročnější než v případě použití již vytvořených widgetů.

Aplikaci je v budoucnu možné rozšířit o podporu dalších typů připojení, podporovaných mobilními telefony, jako je usb. Dalším možným vylepšením by mohla být např. možnost mít více připojených zařízení najednou, nebo ovládání notifikací (pokud je aplikace na pozadí) přímo z řídícího souboru.

Literatura

[1] Embedded Systems Tutorial: History, Types, Advantages, EXAMPLES. In: *Guru99* [online]. [cit. 2019-11-17]. Dostupné z: https://www.guru99.com/embedded-systems-tutorial.html

[2] A Brief About Embedded System their Classification and Applications. In *Watelectronics* [online]. [cit. 2019-11-17]. Dostupné z: https://www.watelectronics.com/classification-of-embedded-systems/

[3] Communication(Wireless) Protocols in IOT. In: *Medium.com* [online]. [cit. 2019-11-17]. Dostupné z: https://medium.com/@hardy96tech/communication-wireless-protocols-in-iot-7da097ebbe96

[4] Remote Desktop Protocol. In: *Docs.microsoft.com* [online]. 2020 [cit. 2020-04-26]. Dostupné z: https://docs.microsoft.com/en-us/windows/win32/termserv/remote-desktop-protocol

[5] Package Index. *Android developers* [online]. [cit. 2020-01-12]. Dostupné z: https://developer.android.com/reference/packages

[6] Secure communication in IoT. *Avaloninnovation.com* [online]. [cit. 2019-12-18]. Dostupné z: https://avaloninnovation.com/en/secure-communication-in-iot-en

[7] Android version history. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2019 [cit. 2019-12-18]. Dostupné z: https://en.wikipedia.org/wiki/Android_version_history

[8] Distribution dashboard. *Android developers* [online]. [cit. 2020-01-12]. Dostupné z: https://developer.android.com/about/dashboards

[9] WebView for Android. *Chrome* [online]. [cit. 2020-01-13]. Dostupné z: https://developer.chrome.com/multidevice/webview/overview

[10] Bluetooth low energy overview. In: *Android developers* [online]. 2020 [cit. 2020-04-26]. Dostupné z: https://developer.android.com/guide/topics/connectivity/bluetooth-le

[11] Guide to app architecture. *Android developers* [online]. [cit. 2020-04-27]. Dostupné z: https://developer.android.com/jetpack/docs/guide

[12] The Clean Code Blog. *Cleancoder* [online]. [cit. 2020-04-27]. Dostupné z: http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[13] Test apps on Android. *Android developers* [online]. [cit. 2020-05-02]. Dostupné z: https://developer.android.com/training/testing

[14] The mobile app face of embedded devices. In: *Spore lab* [online]. [cit. 2019-12-09]. Dostupné z: https://www.sporelab.io/updates/2015/9/22/the-mobile-app-face-of-embedded-devices

Uživatelská dokumentace

Nejjednodušší je aplikaci nainstalovat z Google Play, na adrese:
https://play.google.com/store/apps/details?id=cz.martinforejt.eedc

Druhou možností je ruční instalace pomocí apk souboru umístěném na přiloženém disku v adresáři *app/release/app-release.apk*. Tento soubor je nutné nejprve uložit do svého zařízení (s minimální verzí Androidu 4.4, API 19) a poté musí být uživatelem povolena instalace aplikací z neznámých zdrojů. Tuto možnost lze najít v nastavení, konkrétní umístění se liší u různých verzí Androidu a výrobců. Po povolení instalace z neznámých zdrojů stačí kliknutím na soubor *app-release.apk* spustit instalaci. Při případné nabídce více aplikací přes kterou tento soubor otevřít zvolíme aplikaci *Package installer*. V případě že během instalace vyskočí dialog Google Play Protect se zprávou, že aplikace může být nebezpečná, je nutné kliknout na tlačítko *Install anyway*. Pokud kliknete na tlačítko *OK*, aplikace již nepůjde znovu nainstalovat a bude nutné vygenerovat nové apk s novým číslem verze.

Vygenerování souboru apk je možné buď z vývojového prostředí Android Studia pomocí Build -> Build APK(s), nebo pomocí gradle tasku assembleRelease nebo assembleDebug, příkazy *gradlew assembleRelease* nebo *gradlew assembleDebug* z kořenové složky projektu. Oba tyto způsoby vygenerují apk do složky *app/build/outputs/apk.*

Po úspěšné instalaci, ať už z Google Play nebo pomocí apk souboru je možné začít aplikaci používat a přidat první zařízení ať už dle popisu v této práci v kapitole 10.2 nebo dle dokumentace na webu http://eedc.martinforejt.cz.

Obsah přiloženého disku

- složka *app* – zdrojové kódy modulu *app*
- složka *data* – zdrojové kódy modulu *data*
- složka *domain* – zdrojové kódy modulu *domain*
- složka *presentation* – zdrojové kódy modulu *presentation*
- složka *lib* – zdrojové kódy knihovny *EEDC.js* a uvedené 3 testovací aplikace
- složka *www* – webové stránky http://eedc.martinforejt.cz/
- složka *grafika* – logo aplikace a grafické návrhy v psd a xd
- složka *doc* – soubor Forejt_BPINI.pdf tato práce ve formátu pdf
- složka *literatura* – uložená literatura v době citace
- soubor *readme.txt* – zkopírovaný obsah této přílohy
- ostatní soubory pro konfiguraci vývojového prostředí a gradlu