



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

**KOMPLETNÍ ŘEŠENÍ CHYTRÉ DOMÁCNOSTI S VYUŽITÍM
MODULŮ ESP A RASPBERRY PI**

THESIS TITLE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN BARTOŠ

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.

BRNO 2020

Abstrakt

V dnešní době se velice rychle rozrůstá svět chytrých zařízení v domácnostech, které mezi sebou komunikují a usnadňují lidem život. Avšak velkým problémem pořízení takových zařízení do běžných domácností je vysoká cena a kompatibilita zařízení pouze od jednoho výrobce. Cílem této bakalářské práce je vytvořit komplexní levné řešení, které dokáže komunikovat se zařízeními od různých výrobců. Toto řešení je implementováno pomocí nejmodernějších technologií a nástrojů.

Abstract

Nowadays, the world of smart devices in households is growing up very fast, where the devices communicate with each other and make people's lives easier. However, a major problem of purchase the devices to common households, is high price and compatibility of the devices from one vendor. The goal of this bachelor thesis is to create a comprehensive low-cost solution that can communicate with devices from various vendors. This solution is implemented by the latest technologies and tools.

Klíčová slova

chytrá domácnost, Internet věcí, Java, React.JS, C++, Quarkus, Javascript

Keywords

Smart home, IoT, Java, React.JS, C++, Quarkus, Javascript

Citace

BARTOŠ, Martin. *Kompletní řešení chytré domácnosti s využitím modulů ESP a Raspberry Pi*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Kompletní řešení chytré domácnosti s využitím modulů ESP a Raspberry Pi

Prohlášení

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

.....
Martin Bartoš
20. května 2020

Poděkování

Rád bych poděkoval za odbornou pomoc od mého vedoucího pana doc. Vladimíra Janouška, který mi pomohl dostat se na správný směr mé bakalářské práce.

Obsah

Kapitola 1

Úvod

V dnešní době, kdy lidé jsou zvyklí na větší pohodlí než kdysi, narůstá i počet chytrých domácností. Chytré domácnosti by měly usnadnit život lidem, okolí, i dokonce celé planetě. Když se řekne chytrá domácnost, může se někomu například vybavit jakýsi robot, který vám uklidí, či dokonce přinese nákup. Chytrá domácnost může být jakkoliv chytrá, dokonce se může podobat i domácnostem z různých sci-fi filmů. Můžeme si uvést příklad z každodenního života většiny z nás. Večer přijdete domů s těžkým nákupem, musíte vyndat z kapsy klíče (se štěstím, že jsouprávě v kapse), pracně otevřít, položit nákup, rozsvítit na chodbě, zhasnout na chodbě, rozsvítit v kuchyni atd. Ted si představte, že se blížíte ke dveřím, kamera rozpozná váš obličej, otevřou se dveře, rozsvítí se světla, vozík odvezete váš nákup k lednici a Vy si nemusíte ničeho všímat.

Chytré domácnost má ale i další výhody a to především při úspoře energie. Úspora energie hraje v dnešní době velice důležitou roli. Při výrobě elektrické energie se vyprodukuje mnoho skleníkových plynů, které zatežují životní prostředí. Chytré domácnost může rapidně snížit produkci těchto plynů. Největším odběratelem elektrické energie v domácnostech jsou především různá topná telésa, ať už v podobě elektrického topení, nebo ohřívaců vody, dále různá osvětlení apod. Tito největší odběratelé pracují většinou celý den a komplexně pro celý dům, avšak určité změny by byly velice přínosné, např. vytápění pouze obývaných prostor, snižování intenzity osvětlení podle okolní intenzity světla, různé akce vykonané podle aktuálního času apod. Podle mého názoru se chytré domácnosti dostanou do popředí, na trhu v oblasti informačních technologiích, velice rychle. Je samozřejmé, že chytré domácnosti a chytré zařízení již existují. Ovšem nastává problém, kdy tato zařízení jsou velice draha a výrobce, většinou, podporuje pouze svá zařízení a je téměř nemožné sestavit chytrou domácnost z různých komponentů od různých výrobců.

Cílem této práce je vytvořit kompletní řešení chytré domácnosti, které bude levné a připojená různá zařízení budou mezi sebou komunikovat bez ohledu na výrobce. Dále hlavním rysem tohoto projektu je poskytnutí přístupnosti k domácnosti. Tento systém je navržen tak, aby mohl být nasazen v cloudu a být přístupný pro tisíce uživatelů zároveň. Vyžaduje se určitá robustnost systému a hlavně bezpečnost. Někteří uživatelé však cloudové služby nepodporují a bezpečnost je u nich na prvním místě. Pro tento případ je systém navržen tak, aby mohl běžet na samostatném mini počítači pouze v rámci domácnosti. Dalším cílem práce je poskytnout uživateli intuitivní rozhraní pro snadné zprovoznění chytré domácnosti a následné správě domácnosti.

V kapitole ?? najdete vše, co je potřeba k porozumění zbytku práce. V další kapitole ?? je popsán návrh daného řešení, v kapitole ?? implementace řešení, v kapitole ?? popsáno testování a v poslední řadě, v kapitole ??, závěr práce.

Kapitola 2

Důležité termíny

V této kapitole jsou popsány důležité termíny které jsou nezbytné k porozumění zbytku práce. Asi nejdůležitější je pochopit koncept *IoT*, který je popsán v podkapitole ???. V dalších podkapitolách najdeme stručné vysvětlení pojmu *MCU - neboli mikrokontrolér*(kapitola ??), informace o různých zařízeních, která jsou obsažena v implementaci této chytré domácnosti. Specifický je jedná hlavně o moduly *ESP* (kapitola ??) a mini počítač *Raspberry Pi* (kapitola ??). Komunikace mezi zařízeními probíhá pomocí protokolu *MQTT*, komunikace s uživatelským rozhraním (dále jako *frontend* nebo *klientská aplikace*) a serverem, kde je obsažena celá business logika (dále *backend*), zajišťuje protokol *HTTP*.

2.1 IoT - Internet věcí (Internet of Things)

V informatice se jedná o označení pro síť fyzických zařízení, vozidel, domácích spotřebičů a dalších zařízení, která jsou vybavena elektronikou, softwarem, senzory, pohyblivými částmi a síťovou konektivitou, která umožňuje těmto zařízením se navzájem propojit a vyměňovat si data. Každé z těchto zařízení je jasné identifikovatelné díky implementovanému výpočetnímu systému, ale přesto je schopno pracovat samostatně v existující infrastruktuře internetu. [?]

Jak vyplývá z definice *IoT*, zařízení v domácnosti jsou propojena (např. přes *switch*, nebo *router*) a navzájem komunikují. Po síti se posílají data z různých senzorů a různá zařízení na tato data reagují jiným způsobem. Data mohou být vyhodnocována přímo daným zařízením, nebo systémem, který je pro zařízení dostupný. Tento systém může být ve formě distribuované sítě, kde data mohou být vyhodnocována a zasílána i na druhý konec světa. V *IoT* velmi často vystupuje *umělá inteligence* (dále jako *AI - artificial intelligence*), která může předávat data různým zařízením a vyhodnocovat pomocí neuronových sítí.

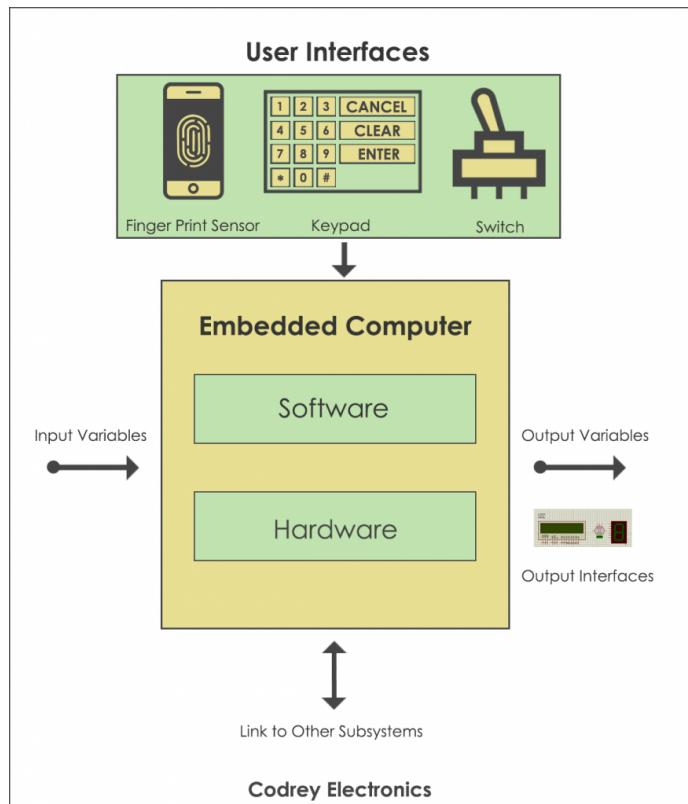
IoT je využíváno i v prostředí různých výrobních procesů, či linek. V továrnách, kde velké stroje plní svoji práci, mohou také mezi sebou komunikovat dané stroje a rozdělení práce mezi dané stroje může být tímto přístupem velice efektivní. Stroje rozloží výrobní proces tak efektivně, že všechny stroje jsou zatížené způsobem, který odpovídá požadavkům zavedeným na daný proces. Amortizace těchto strojů se sníží, a proto i náklady na výrobu jsou nižší. Stroje mohou mezi sebou komunikovat dokonce i mimo lokální továrny a vyměňovat si data např. o počtu materiálu, který je přítomen na skladě a opět efektivně přerozdělovat materiál do více skladů a továren.

2.2 Vestavěný systém (Embedded system)

Vestavěný systém, neboli *Embedded system* je hardwarový systém, který je tzv. *microprocessor-based* (založený na mikroprocesoru). Obsahuje software, který je navržen k vykonávání určité specifické činnosti buď jako nezávislý systém, nebo jako část většího celku. Nejvýznamnější komponenta vestavěného systému je integrovaný obvod, který je navržen pro operace, které jsou závislé na reálném čase (dále jako *real-time* systém). [?]

K vestavěnému systému lze připojit mnoho vstupních/výstupních periferních zařízení, které spolupracují s mikrokontrolérem (více info viz. ??) obsaženým ve vestavěném systému. Na obrázku níže je vyznačeno jednoduché schéma, které pojednává o asociaci mezi vestavěným systémem, vstupních/výstupních zařízení a dalších náležitostí.

Nejdůležitější částí je připojení vstupních zařízení, mezi které může patřit např. klávesnice, tlačítka a různé senzory. Vestavěný systém pomocí mikrokontroléra vyhodnotí získaná data a přivede na výstup patřičnou hodnotu, odpovídající programovému zpracování dat podle obsaženého softwaru (také označovaný jako *firmware*). Mezi výstupní zařízení můžeme zahrnout např. různá relé, tranzistory, audio a video výstupy a mnoho dalších.



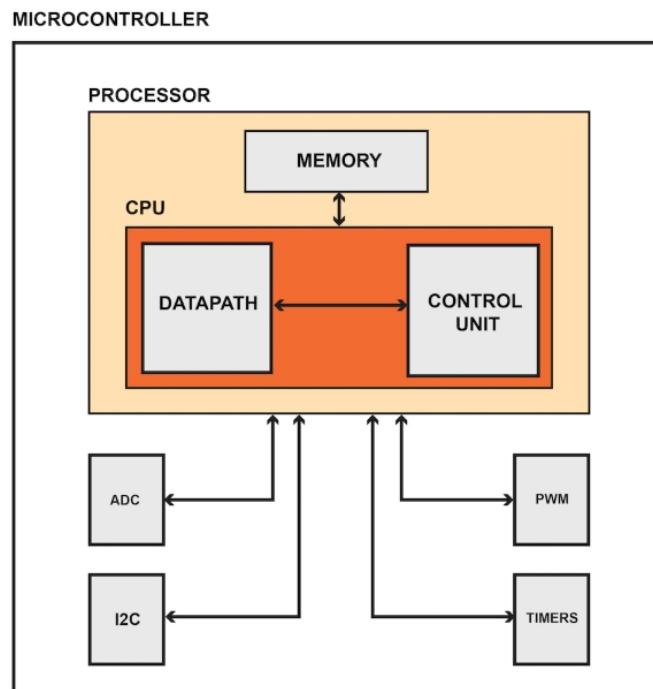
Obrázek 2.1: Koncept vestavěného systému

2.3 MCU - mikrokontrolér (Microcontroller unit)

Mikrokontrolér, nebo také jednočipový počítač, je integrovaný obvod, který obsahuje kompletní *mikropočítáč*. Mikrokontroléry jsou velice spolehlivé, kompaktní a proto hojně využívány k řízení různých elektronických systémů, obvykle pomocí mikroprocesoru, paměti a různých periferních zařízení. Tato malá zařízení jsou optimalizována pro vestavěné systémy popsané výše, které potřebují zpracovat interakci s digitálními, analogovými, nebo elektromechanickými komponentami.

Název pro mikrokontrolér byl velice vhodně zaveden, protože opravdu zdůrazňuje charakteristiku typu daného zařízení. Mikrokontroléry jsou malá zařízení, která dokáží velké věci. Mikrokontroléry hrají důležitou roli v technologické budoucnosti. Získavají si velikou oblibu nejen u různých SW¹/HW² expertů, ale i studentů a spousty hobby nadšenců.[?] Díky mikroprocesorům lze jednoduše propojit SW nástroj, či aplikaci s HW komponentami.

Blížící se budoucnost přinese spoustu nových možností v oblasti mikrokontrolérů a počet zařízení obsahující mikrokontrolér může v blízké době přesáhnout i několika desítek miliard kusů. Výkon u těchto zařízení bude vyšší a velikost menší, což je občas velice důležitý faktor na poli chytrých domácností. V chytré domácnosti jsou zařízení sestavena s pomocí mikrokontroléra, který se stará o danou specifickou funkci. V této bakalářské práci jsou především použity mikrokontroléry s čipem ESP-8266 od firmy *Espressif Systems* a minipočítáč *Raspberry Pi*. Obrázek níže obsahuje odpovídající architekturu skoro každého mikrokontroléra. Mezi hlavní prvky patří analogově-digitální převodník, čítače, časovače, modul řídící PWM³.



Obrázek 2.2: Koncept mikrokontroléru

¹SW - software

²HW - hardware

³PWM - Pulse-Width Modulation (pulsně šířková modulace).

2.4 MQTT protokol

MQTT neboli *Message Queuing Telemetry Transport* protokol je jeden z nejvýznamnějších komunikačních protokolů IoT zařízení a systémů. *MQTT* patří do kategorie aplikačních protokolů, kde jeho hlavní předností je malá velikost datové hlavičky a možnost komunikovat v sítích s omezenou propustností. Patří do skupiny *Publish-Subscribe*(dále *Pub-Sub*) protokolů. Hlavním principem *Pub-Sub* protokolů je výměna zpráv mezi dvěma typy účastníků. První z nich je tzv. odebíratel(dále *subscriber*), který se přihlašuje k odběru zpráv s daným temátem(dále *topic*). *Subscriber* může přijímat několik zpráv s různými tématy a v průběhu se od odběru i odhlašovat. Druhým typem je tzv. vydavatel(dále *publisher*), který posílá zprávy do určitého topicu. [?]

MQTT Broker

MQTT Broker je služba (nebo-li software, který běží v cloudu, či lokálním PC), která se stará o rozesílaní a spravování zpráv, které různá zařízení posírají. *MQTT Broker* je prostředník mezi účastníky typu *subscriber* a *publisher*. Každé zařízení, které chce využívat *MQTT* protokol a komunikovat s ostatními zařízeními, musí definovat *URL*⁴ brokeru, přes který budou dané zprávy procházet. Jak již bylo řečeno, *MQTT* je *Pub-Sub* protokol, proto když přijde od účastníka typu *publisher* zpráva ke specifickému topicu, musí se podívat, kdo všechno je přihlášen k odběru daného topicu a rozeslat všem účastníkům, typu *subscriber*, danou zprávu. Každé zařízení může jak přijímat, tak i odesílat zprávy k danému topicu. [?]

Zabezpečení

V IoT je velice důležitá i bezpečnost a pomocí *MQTT Brokeru* lze dosáhnout určitého bezpečnostního cíle. Je opravdu nevítané, aby se někdo cizí dostal bez autentizace k odběru vašich zpráv procházející přes *MQTT Broker* a mohl zasílat zprávy do různých dalších zařízení připojená v domácnosti. Útočník tak má šanci kontrolovat celou vaši domácnost. Každý *MQTT Broker* od různých výrobců disponuje různou sadou zabezpečení a vlastností. [?]

Možné typy zabezpečení:

- Využívání šifrovaného portu **8883** namísto nešifrovaného **1883**
- Autentizace pomocí uživatelského jména a hesla
- **TLS**⁵ připojení
- **OAuth**⁶ správa

⁴**URL** - Uniform Resource Locator (jednotná adresa zdroje)

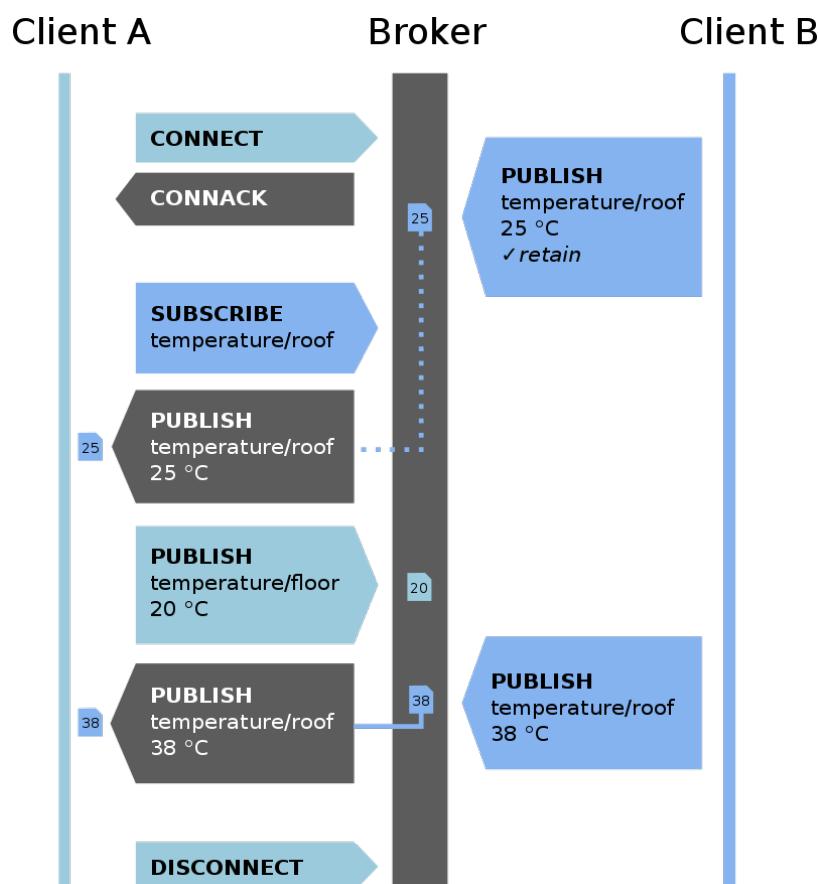
⁵**TLS** - Transport Layer Security

⁶**OAuth** - Poskytuje autentizační a autorizační správu

Typy zpráv

MQTT protokol využívá sadu různých typů zpráv pro vlastní běh. Na obrázku ??lze zahlédnout posloupnost zpráv. [?] Mezi základní typy zpráv u protokolu MQTT patří:

- **CONNECT** - slouží pro ustanovení připojení.
- **CONNACK** - (*Connection acknowledge*) typ zprávy je odezva z brokeru klientu, který požadoval o připojení a vytvoří se spojení mezi uzly.
- **SUBSCRIBE** - neboli odběr; klient pošle tuto zprávu brokeru na určitý topic, u kterého chce přijímat dané zprávy
- **UNSUBSCRIBE** - slouží k odhlášení klienta od odebírání zpráv z určitého topicu
- **PUBLISH** - slouží na publikování zpráv na různý topic (např. zasílání dat ze senzorů)
- **DISCONNECT** - slouží k odhlášení klienta z celého systému



Obrázek 2.3: Posloupnost MQTT zpráv

QoS - Kvalita služeb (Quality of services)

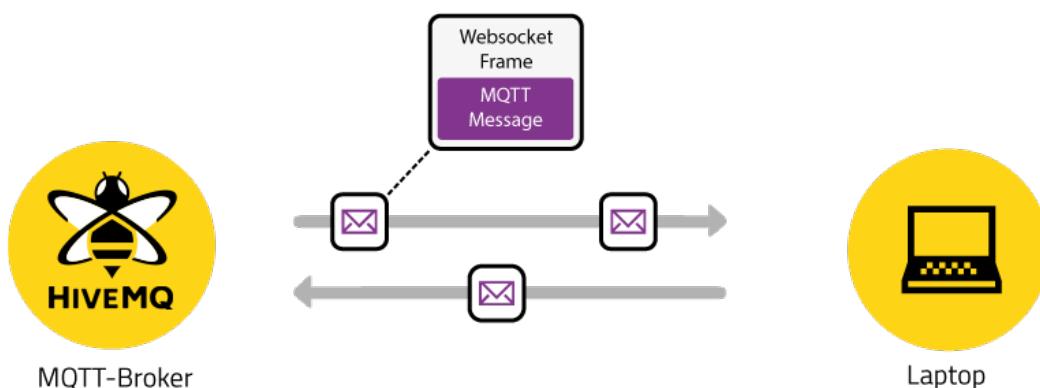
Každá zpráva může specifikovat, jak bude zajištěna kvalita služeb u poslání daných zpráv. [?]

- **QoS 1** - Nejvýše jednou; zpráva je zaslána pouze jednou bez potvrzení.
- **QoS 2** - Alespoň jednou; zpráva stále zasílána dokud nedorazí potvrzení o přijetí.
- **QoS 3** - Přesně jednou; zajištění, že přjemce dostane zprávu pouze jednou.

MQTT over WebSockets (*MQTT pomocí protokolu WebSocket*)

MQTT přes protokol *WebSocket* je technika, kdy se zabalí komunikační protokol *MQTT* do tzv. *WebSocket frame*(*WebSocketový rámec*). Tato technika přenosu je hlavně přínosná pro webové aplikace, tudíž pro vše, co běží v prohlížeči. Udává se, že s touto technikou může být jakýkoli prohlížeč na různých zařízeních plnohodnotný *MQTT klient*.??

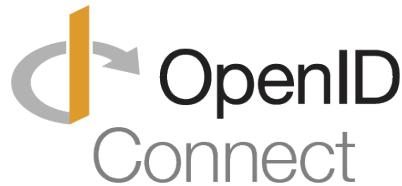
Protokol *WebSocket* je v prostředí webových aplikací velmi dlouhou dobu, poskytuje *full-duplex*⁷ komunikaci a umožňuje hladký chod *real-time* aplikací. Pro inicializaci spojení využívá protokol *HTTP*. Tato technika tak poskytuje optimální způsob komunikace přes *MQTT* protokol u zařízení a prohlížeče.



Obrázek 2.4: MQTT over WebSockets

⁷Full-duplex - obousměrný provoz

2.5 OIDC



Obrázek 2.5: OIDC logo

OIDC nebo-li *OpenID Connect* je autentizační protokol, založený na specifikaci *OAuth 2.0*. Tento protokol využívá služba *KeyCloak*, která je použita v této bakalářské práci. *OIDC* protokol využívá *JSON Web Tokens* (dále jako *JWT*), což je formát využit pro reprezentaci ID tokenu. *OIDC* je hlavně o autentizaci uživatelů. Jeho účelem je poskytnout pouze jeden přihlašovací proces pro více webových stránek. ??

2.6 JSON

JSON nebo-li **JavaScript Object Notation** je formát pro strukturovaná data. Formát dat se sestavuje podle dvojice atribut-hodnota a případně i pole. *JSON* je nezávislý na programovacích jazycích a je odvozen, jak název vypovídá, z programovacího jazyka *JavaScript*. *JSON* je velice rozšířený ve spustě aplikací, kde téměř všude nahrazuje *XML*.

Tento formát zápisu dat je použit v systému, který popisuje tato bakalářská práce, ve dvou odlišných případech. První z nich je při komunikaci s klientskou aplikací, kde se data zprostředkovávají pomocí tohoto formátu *JSON*. V druhém případě je tento formát použit i u *MQTT* zpráv posílané ze zařízení. Jedná se hlavně o zprávy, které se posílají pomocí *MQTT* protokolu na správu zařízení v domácnosti. Na obrázku ?? je vidět ilustrační zápis dat pomocí *JSON* formátu.

```
[  
  {  
    "date": "2013-11-05",  
    "locations": {  
      "United States": 4,  
      "Germany": 8  
    }  
  },  
  {  
    "date": "2013-11-11",  
    "locations": {  
      "South Africa": 9  
    }  
  },  
  {  
    "date": "2013-11-12",  
    "locations": {  
      "Japan": 6  
    }  
  }  
]
```

Obrázek 2.6: Ukázka *JSON*

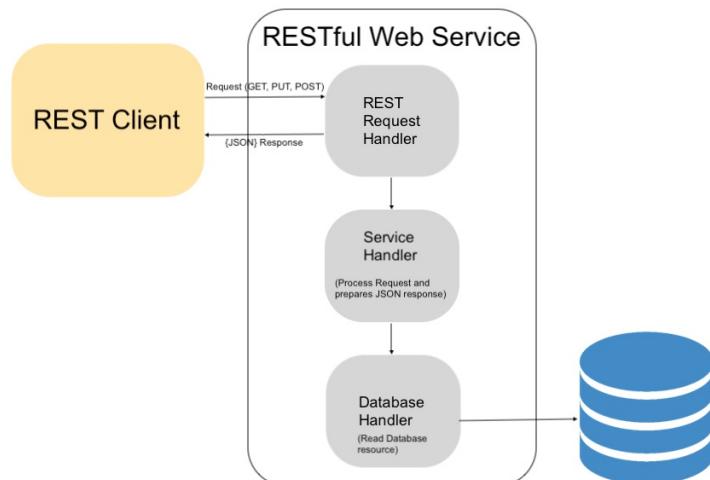
2.7 RESTful aplikace

REST je akronym⁸ pro **R**epresentational **S**tate **T**ransfer, které značí architekturu rozhraní pro distribuované *hypermédia* systémy a byl poprvé prezentován Royem Fieldingem v jeho disertační práci roku 2000.

Jestliže jakékoli rozhraní chce být považováno za *RESTful*, musí splnit určité podmínky:

- **Klient-server** - Oddělení uživatelského rozhraní ze serverové části. Zlepšení přenositelnosti, škálovatelnosti díky zjednodušení serverových komponentů.
- **Bezstavovost** - Každý požadavek od klienta na server musí obsahovat všechny nezbytné informace k porozumění požadavku. *Session state*(stav sezení) se celý ukládá pouze u klienta.
- **Cacheable** (schopný být uložen v mezipaměti) - Požaduje aby data v odpovědi ze serveru byla implicitně, nebo explicitně označena jako *cacheable*, nebo *non-cacheable*. Jestliže je odpověď *cacheable*, mezipaměť u klientské aplikace má právo být znovupoužita později jako odpověď na ekvivalentní požadavek.
- **Jednotné rozhraní** - Použitím principu obecného softwarového inženýrství na rozhraní komponent se celková architektura systému zjednoduší a zlepší se viditelnost interakcí.
- **Vrstvený systém** - Vrstvený systém umožňuje, aby se architektura skládala z hierarchických vrstev způsobem omezení chování komponent tak, že každá komponenta nemůže „vidět“ za nejbližší vrstvu se kterou interagují.

Klíčovou abstrakcí informace v *REST* architektru je tzv.*resource* (zdroj informací). Každá informace, která může být pojmenovaná smí být také považována za zdroj. *REST* využívá identifikátory zdrojů informací aby dokázala identifikovat určitý obsah v interakci mezi komponentami.?? Na obrázku ?? je popsán tok zpracování požadavků.



Obrázek 2.7: MQTT over WebSockets

⁸Akronym - nehláskovaná zkratka

Kapitola 3

Současný stav a návrh řešení

V této kapitole je shrnut současný stav řešení chytré domácnosti od nejvýznamnějších firem, které se zabývají tímto tématem a samotný návrh řešení, který by měl danou problematiku vyřešit. Jak již bylo popsáno v úvodu, největším úskalím řešení chytré domácnosti je celková cena, a kompatibilita se zařízeními od různých výrobců.

V kapitole ?? jsou popsány existující řešení chytré domácnosti od různých firem a v kapitole ?? vlastní řešení problému a schopnost dosažení požadovaných cílů.

Existující řešení

Kompletní řešení chytré domácnosti existuje v dnešní době už opravdu nepřeberné množství. Podle mého názoru se postupem let dostane do popředí ještě více *IoT* zařízení a systémů kompletní chytré domácnosti. Už v dnešní době spousta obrovských IT firem, mezi které patří např. *Samsung*, *LG*, *Apple* a *Google*, vyvíjí takové systémy.

Například firma *Samsung* představila na trh produkty s názvem *SmartThings*, které lze využít pro sestavení chytré domácnosti. Hlavním prvkem, který potřebujete je tzv. *Hub*, který slouží jako mozek celého systému. U svých zařízení většinou využívají pro komunikaci protokol zvaný *Zigbee*, který se řadí do tzv. *ad-hoc*¹ sítí. Signály ze zařízení dosahují vzdálenosti 10 až 20 metrů. *Samsung* se pyšní tím, že jejich zařízení jsou kompatibilní s více než 100 zařízeními od různých výrobců, ale jejich cena je obrovská.

Lze si to představit na jednoduchém příkladu u chytrého tlačítka od firmy *Samsung*. Toto tlačítko se vyznačuje tím, že dokáže po nastavení ovládat několik světel současně, spotřebiče, apod. Tlačítko se v průměru prodává za nějakých 15\$, což je v přepočtu něco okolo 380kč. Takové tlačítko, které dokáže sadu podobných, dokonce i stejných úkonů, lze sestrojit pro můj vlastní návrh chytré domácnosti asi za cenu okolo 30kč.

Dalším příkladem může být chytrá zásuvka, kterou *Samsung* prodává za cenu blízkou 35\$, což je v přepočtu asi 880kč a ve své vlastní domácnosti lze sestrojit podobný kus asi za 60kč. Samozřejmě lze brát v úvahu i to, že *Samsung* zaručuje kvalitu a dlouhodobou spolehlivost svých zařízení, využívá odlišný protokol, ale cena, podle mého názoru, je až přehnaně vysoká.

¹**ad-hoc** - decentralizovaná bezdrátová síť

Obecný návrh řešení

Návrh vlastního řešení systému chytré domácnosti je poněkud komplikovanější záležitost, díky své rozmanitosti. Vývojář tak musí být obeznámen s technologiemi a principy z různých odvětví aplikačního vývoje. Systém chytré domácnosti by měl disponovat velikou škálou možností a být jednoduše škálovatelný. Pro návrh takového systému je potřeba v první řadě vytvořit analýzu požadavků.

Hlavní aspekty, které by měly být splněny:

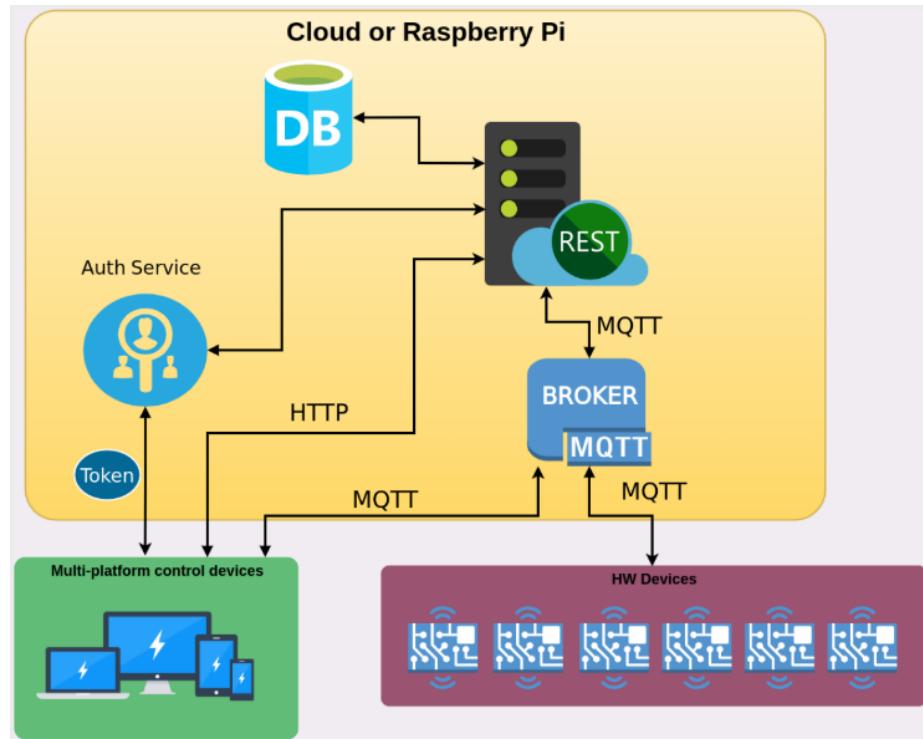
- **Cena pořízení** - Kompletní systém i jednotlivé komponenty.
- **Zabezpečený systém** - Na všech dostupných úrovních (server, databáze, klientská aplikace, zařízení, přenos dat).
- **Intuitivnost a přístupnost** ovládacích prvků
- **Jednoduchost** - Vytvoření domácnosti a inicializování/připojení zařízení i příliš netechnicky zdatným uživatelem.
- **Sdílení domácnosti** - Schopnost sdílet domácnost s více uživateli.
- **Autorizace uživatelů v domácnosti** - Každý uživatel v různých domácnostech může mít různá přístupová práva.
- **Dostupnost** - Dostupnost systému jak v případě pouze lokální sítě, tak i s pomocí cloudových služeb.
- **Použitelnost** - Schopnost systému obstarávat několik tisíc uživatelů zároveň.
- **Customizace² zařízení** - Přidávání/odebírání schopností zařízení jednoduchým způsobem.

Jak již bylo řečeno, když na návrh systému chytré domácnosti je pouze jeden vývojář, musí se orientovat v rozmanité sféře technologií, návrhových vzorů apod. V dnešní době jsou tyto typy vývojářů označovány jako *Full Stack Developers*, kteří vyvíjí *backend*, tak i *frontend* část aplikace.

V kapitole ?? bude detailněji popsán **návrh serverové (*backend*) části** systému, která je nejspíše nejkomplikovanější a nedůležitější ze všech odvětví této bakalářské práce. Dále v kapitole ?? lze nalézt popis **návrhu relační databáze** a související věci. V kapitole ?? je **návrh klientské aplikace (*frontend*)**, kde hlavním jádrem věci je rozložení a vzhled stránky. Samozřejmě je zde i popis struktury uložení stavu aplikace a operací provádených na pozadí. V poslední kapitole ?? je **návrh HW zařízení**, struktura programu.

²**Customizace** - úprava dle požadavků uživatele

Na obrázku ?? je graficky znázorněna architektura navrhovaného systému. Navrhovaná architektura systému obsahuje server, *MQTT* broker, autentizační/autorizační službu, databázi, klientské aplikace a HW zařízení. Služby, které budou nasazeny v cloudu, či v lokální síti na minipočítači jsou vyznačeny ve žluté části obrázku. V zelené části obrázku jsou vyznačeny klientské aplikaci a ve fialové části samotná HW zařízení. Server komunikuje s *MQTT brokerem* pomocí komunikačního protokolu *MQTT*, kde server je další *MQTT klient*(detailně popsáno v ??). Server dále komunikuje s klientskou aplikací(v zelené části obrázku) pomocí protokolu *HTTP*, která posílá požadavky na server. Klientská aplikace se serverem dále komunikují s příslušnou autentizační/autorizační službou.



Obrázek 3.1: Základní architektura systému

3.1 Návrh serverové části

Z hlediska vývoje aplikací je z mého pohledu tato část práce nejkomplikovanější a zároveň nejobtížnejší. Vývojář musí mít dobrý přehled o technologiích, návrhových vzorech, které lze vhodně aplikovat na danou aplikaci. Struktura programu této části by měla být napsána přehledně a stylem takovým, aby bylo možno jednoduše rozšiřovat. Požadavky na aplikaci se mohou velice rychle měnit a vývojář musí dokázat bez většího úsilí implementovat tyto změny.

Serverová část(dále jako *Server*) aplikace se stará o požadavky, které klientská aplikace, potažmo zařízení, posílá na daný server. Dále se také zabývá samotnou logikou celého systému, vykonáváním požadavků na databázi, poskytnutím výpočetních sil pro složitější operace a požadavky od zařízení v domácnosti. Vytváří rozhraní, ke kterému může přistupovat spousta klientských aplikací a spravovat tak celý management systému chytré domácnosti.

Jedná se tak o jádro celého systému. Hlavním úkolem této backend aplikace je zprostředkovat služby, pro zařízení chytré domácnosti, pro připojení do systému. Ukládá důležité hodnoty ze zařízení, co se týče stavů výstupních zařízení a poslední hodnoty vstupních zařízení, aby byly stále dostupné při připojení klientské aplikace. Klientské aplikace vytváří požadavky na tuto backend aplikaci a získávají tázaná data. Tato aplikace je vytvořena robustnějším způsobem, aby byla schopna obstarávat požadavky několika tisíců uživatelů. K dané aplikaci se váže i otázka bezpečnosti.

Uživatel se musí autentizovat v daném systému, aby bylo možno identifikovat přihlášeného uživatele a s tím související autorizační práva na různé zdroje informací.

Mezi velikou škálu programovacích jazyků, které jsou vhodné na vývoj serverových aplikací, jsem si vybral programovací jazyk *Java*, který se v popularitě těchto technologií drží na předních místech. Programovací jazyk *Java* je objektově orientovaný jazyk, který se velkým způsobem zasadil do vývoje enterprise aplikací. Jazyk *Java* disponuje rozsáhlým ekosystémem a existuje spoustu výborných frameworků.

Rozhodl jsem se, že serverová část bude sestrojena jako *RESTful*(podrobněji ??) webová služba komunikující nad protokolem *HTTP*, kde u serveru nebude žádné uživatelské rozhraní, ale pouze programové, známe jako *API*³. Server poskytne *API*, které využívají klientské aplikace, nebo další služby pro komunikaci se serverem. Server je bezstavový, což znamená, že si server neukládá stav o požadavcích a každý požadavek je přijímán stejně, bez jakýkoliv priorit, či upřednostnění díky danému stavu uživatele v systému.

Zabezpečení

Každý uživatel, který pošle požadavek na server, musí být autentizován. Server je asociován s autentizační službou a při každém požadavku na server přepošle tzv.*token*⁴ autentizační službě a ta vyhodnotí, zda je uživatel autentizován, potažmo autorizován získat informace z daného endpointu⁵. Pokud uživatel není autentizován a nevyplňuje žádaný *token*, musí poslat požadavek na autorizační službu s danými *credentials*⁶, kde získá daný token pro přístup ke zdrojům informací serveru. Token je zasílán v HTTP hlavičce v atributu *Authorization*.

O autorizaci se stará přímo server, který ověří u dané autentizační služby, zda je uživatel autentizován a zda má dostatečná práva na zisk informací z daného zdroje. Pokud uživatel není autentizován a není přesměrován na stránku autentizační služby, vrací se *HTTP response*⁷ kód s číslem 401, který se označuje jako *Unauthorized*(neoprávněný). Uživatel který není autorizován k získání informací dostane odpověď ze serveru s kódem 403, který značí *Forbidden*(zamítnuto).

Server vyhledá uživatele v databázi a nejčastěji podle role, kterou má uživatel přidělenou pro danou domácnost, se ověří, zda má přístupová práva. Server musí být tudíž schopný ověřit vše už před samotným vykonáváním operací daných u příslušných *REST Endpointů*. Dále je zavedeno autorizační pravidlo v systému, které zkoumá, zda se požadované zdroje informací týkají žádajícího uživatele. To platí např. u spravování místností, kde uživatel není schopen spravovat místnosti, ale v případě, že je uživatel vlastník místnosti, tyto práva mu naleží.

³API - Application Programming Interface (rozhraní pro programování aplikací)

⁴Token - žeton pro přístup ke zdrojům informací

⁵Endpoint - HTTP path (cesta požadavku definována pomocí API)

⁶Credentials - poveření v různých formách (heslo, PIN, OTP, biometrika,...)

⁷Response - odpověď na požadavek ze serveru

Přístupová práva

Přístupová práva pro uživatele jsou různá. Každý uživatel může mít odlišné role v různých domácnostech, tudíž nemá specifikovanou jednu roli jako to občas bývá zvykem u serverových aplikací. Každá role má různé vlastnosti a práva přístupu k informacím, či práva k vykonání určitých operací.

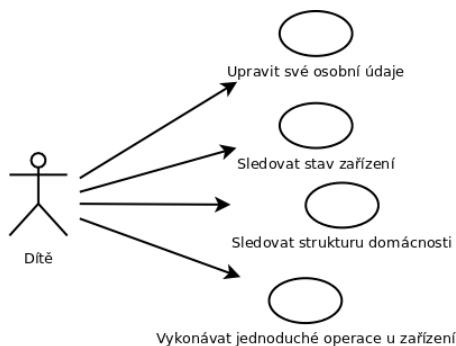
Mezi tři základní skupiny se řadí:

- **Administrátor**
- **Klasický uživatel**
- **Dítě**

Tyto typy uživatelů jsou hierarchicky rozpoložené, kde uživatel ve skupině *Dítě* je na nejnižším místě s nejmenším počtem přístupových práv, poté *Klasický uživatel* a nejvýše položený je *Administrátor*, který má největší počet přístupových práv. Tyto základní skupiny jsou dále modifikovatelné. Administrátor domácnosti může vytvořit další podřazené skupiny u kterých definuje přístupová práva a může přidat uživatele do skupin. Uživatelé ve skupině, které administrátor domácnosti přidělí právo spravovat uživatele ve skupinách, jsou schopni spravovat uživatele ve skupinách pouze v podřazených skupinách.

Uživatel, který si vytvoří vlastní domácnost se automaticky stává administrátorem dané domácnosti a může ji plně spravovat, dokonce může přidávat a odebírat uživatele z domácnosti. Uživatel, který má minimální roli *Klasický uživatel* v domácnosti je schopen si vytvořit vlastní pokoj, který může plně spravovat. Uživatel s rolí *Dítě* takovou možnost nemá. Administrátor má také právo poslat pozvánku uživateli do domácnosti s určenou rolí v domácnosti. Uživatel může buď přijmout pozvánku a být tak zařazen do domácnosti s určitou rolí, nebo v opačném případě odmítnout a smazat pozvánku.

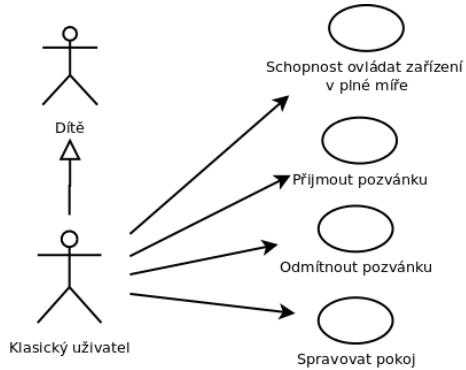
Na obrázcích níže lze nalézt grafické znázornění operací, které lze vykonávat s určitou základní rolí. Grafické znázornění je vytvořeno pomocí *Use Case*⁸ diagramu, který spadá do kategorie diagramu chování definovaný v *UML*⁹. *Use Case* diagram zachycuje pouze vnější pohled na modelovaný systém a nepoukazuje na způsob implementace daných operací. ?? Na obrázku ?? je zachyceno rozhraní pro uživatele s rolí *Dítě*, na obrázku ?? pro uživatele s rolí *Klasický uživatel* a na posledním obrázku ?? pro uživatele s rolí *Administrátor*.



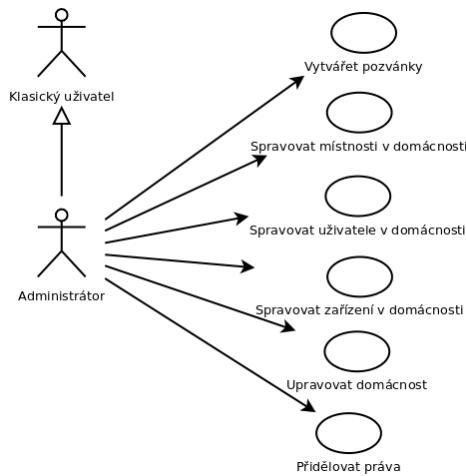
Obrázek 3.2: Diagram užití pro roli **Dítě**

⁸Use Case diagram - diagram případů užití

⁹UML - Unified Modeling Language (grafický jazyk pro vizualizaci)



Obrázek 3.3: Diagram užití pro roli **Klasický uživatel**



Obrázek 3.4: Diagram užití pro roli **Administrátor**

Správa požadavků ze zařízení

Důležitou součástí serverové části systému je správa požadavků ze zařízení asociovaných s domácností. Komunikace mezi zařízením a serverem probíhá pomocí protokolu *MQTT*. Pro každou domácnost existuje na serveru jedna instance *MQTT* klienta. U každé domácnosti lze definovat pouze jeden *MQTT Broker* a náležitá zařízení se připojují přímo k němu. Daná instance *MQTT* klienta je přihlášená k odběru celého provozu dané domácnosti. Lze tak jednoduše odchytávat komunikaci mezi zařízeními v domácnosti za účelem správy perzistentního obsahu zařízení. Analýza provozu a komunikace u zařízení slouží k vytváření statistik posledních hodnot, které mohou být použity k efektivnímu chodu domácnosti a její správě. Lze tak efektivně řídit provoz elektricky náročných zařízení. Dále je možnost analyzovat data, která jsou dále zpracována pomocí většího výpočetního výkonu serveru, než u daných zařízení a generovat tak příslušný výstup.

Pomocí protokolu *MQTT* jsou spravovány i požadavky, týkající se *CRUD* operací daného zařízení. Tyto operace pomocí *MQTT* jsou pouze přístupné zařízením a ne klientským aplikacím. Zařízení je tak schopno zaslat požadavek na server, který spojí dané zařízení s domácností, či připojení již existujícího zařízení a následně aktivovat dané zařízení pro přístup z klientských aplikací.

3.2 Návrh databáze

Návrh databáze úzce souvisí se serverovou částí systému. Server vytváří rozhraní, pomocí kterého lze vytvářet operace nad danou databází. Server disponuje určitým *ORM* frameworkm, který mapuje objektově orientový model do relační databáze a vývojař tak není nucen pracovat s databází na nižší úrovni např. pomocí SQL jazyka.

Návrh databáze u daného řešení chytré domácnosti je modelován pomocí *ER diagramu*¹⁰, kde jsou znázorněné entity¹¹ modelující tabulky databáze a jejich propojení. Databáze je tak modelována pomocí modelovacího jazyka s názvem *UML*. Diagram obsahuje název entity, primární a cizí klíče, atributy entity a asociace tabulek s určitou kardinalitou.

Mezi základní entity řešení chytré domácnosti patří:

- **User** - základní informace o uživateli
- **Home** - domácnost
- **Room** - místnost/pokoj
- **Device** - zařízení
- **Capability** - schopnost zařízení

Entita **User** - uživatel

Entita **User**(uživatel) obsahuje dva identifikátory. První z nich je primární klíč celé entity a druhý identifikátor typu *UUID*¹² slouží k identifikaci uživatele z *JWT* tokenu. Tato tabulka je pouze pomocnou entitou v udržování informací o uživatelích. Primárním uložištěm informací je autentizační služba, která dále obsahuje i přístupová data. Daná entita se využívá hlavně v případě, kdy uživatel pro každou domácnost disponuje různými rolemi. Tato entita je dále využívána v identifikaci uživatele v pozvánkách do domácnosti.

Entita **Home** - domácnost

Nejdůležitějším celkem celé databáze je entita **Home**, která obsahuje základní potřebné informace o domácnosti. Jako každá tabulka databáze musí mít přiřazený primární klíč s jednoznačným identifikátorem dané entity. Dále obsahuje prvek *name*, která je typu *String*(řetězec znaků) a je zde uložen název domácnosti. Důležitým atributem entity je *brokerURL*, což je URL *MQTT* brokeru. Domácnost může mít pouze jeden *MQTT Broker*, tudíž stačí uložit jeden prvek s typem *String*. Entita **Home** dále obsahuje cizí klíč pro entitu *MQTTClient*, kde kardinalita daného vztahu je 1:1 díky tomu, jak již bylo zmíněno, může být pouze jeden *MQTTBroker* a *MQTTClient* pro domácnost. U klienta se rozumí spíše klient, který spravuje a analyzuje zprávy z komunikačního protokolu *MQTT*.

Entita **Room** - místnost

Entita **Home** je provázána s entitou **Room**. Domácnost může obsahovat více místností, ale daná místnost může být zahrnuta pouze v jedné domácnosti. Dále je provázána s entitou **Device**(zařízení), kde představuje seznam zařízení, které nejsou zatím přiřazeny do určitého

¹⁰**ER Diagram** Entity-Relationship diagram(entitně vztahový diagram)

¹¹**Entita** - věc schopná samostatné existence

¹²**UUID** - Universally Unique Identifier(Univerzální unikátní identifikátor)

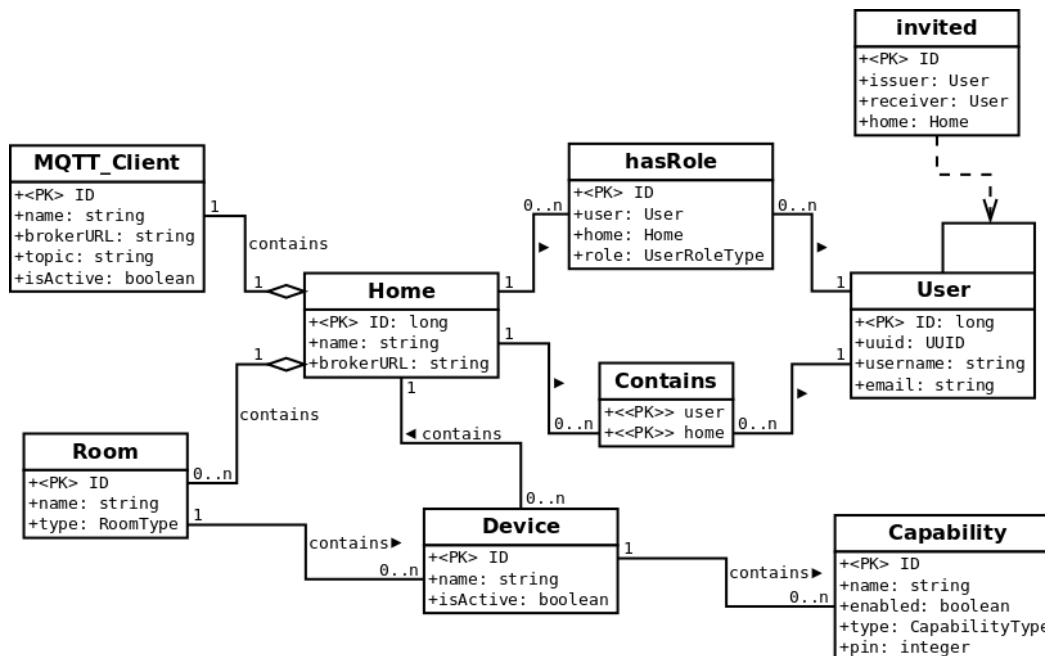
pokoje a jsou proto tzv. *unassigned*(nepřiřazené). Dále se k entitě *Home* vztahuje i entita *HomeInvitation*(pozvánky do domácnosti), kde jsou ve vztahu 1:N kvůli tomu, že domácnost může obsahovat několik pozvánek, ale v pozvánce může být zahrnut pouze jedna domácnost. Jako další provázána s *Home* entitou je entita *User*, která je ve vztahu M:N. Tento vztah představuje asociaci, kde domácnost může obsahovat více uživatelů a uživatelé mohou být obsaženi ve více domácnostech. Vytváří se tak jedinečná asociace v celé databázi. V poslední řadě si uchovává množinu uživatelů, kteří tuto místo vlastní a mají určitá vyhrazená práva k místo.

Entita *Device* - zařízení

Podstatná entita systému je *Device*, která obsahuje opět jednoznačný identifikátor, název zařízení a status, zda je zařízení aktivní. Zařízení může být provázáno s místo, nebo i s domácností a proto tudíž obsahuje dva cizí klíče referující na *Home* a *Room*. Pokud zařízení nemá přiřazeno žádnou místo, automaticky je vložen do množiny, kterou spravuje entita *Home* a obsahuje nepřiřazená zařízení. Každé zařízení je lehce modifikovatelné a uživatel si může vytvořit své vlastní zařízení disponující určitými tzv. *capabilities*(schopnostmi), jako je např. měření teploty, vlhkosti, ovládání světelných zařízení atd.

Entita *Capability* - schopnost zařízení

Entita podřazená zařízení nese název *Capability*. Tato entita obsahuje prvky, které jsou nejblíže samotnému HW zařízení. Obsahuje identifikátor, název, příznak *enabled*(zda je schopnost povolená), typ dané schopnosti a pin ke kterému je připojena komponenta zprostředkovávající schopnost připojena. Kardinalita vztahu mezi entitou *Device* a příslušnou entitou je v poměru 1:N, kde zařízení může mít několik schopností a daná jednoznačná schopnost může být přiřazena jednomu zařízení.



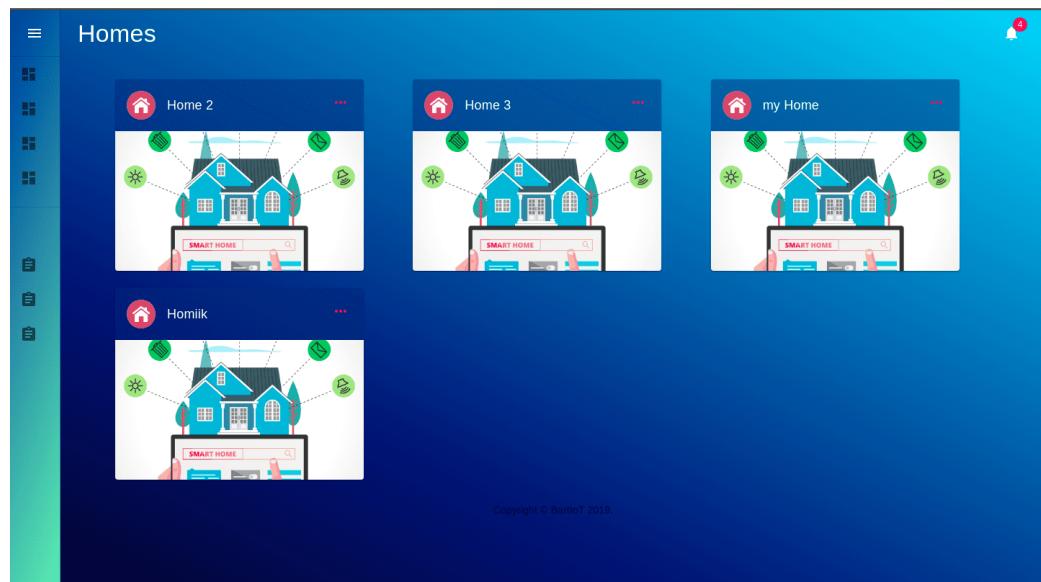
Obrázek 3.5: ER Diagram databaze

3.3 Návrh klientské aplikace

Klientská aplikace slouží k přímé interakci s uživatelem a navrhovaným systémem. Tato aplikace slouží ke správě domácností, místností, zařízení a všech dodatečných vlastností systému. Pro přístup do aplikace je nutné být autentizován. Zda uživatel autentizován není, je přesměrován na stránku autentizační služby, kde musí poskytnout své přihlašovací údaje, či jiným definovaným způsobem prokázat svoji identitu. Po úspěšném přihlášení je uživatel automaticky přesměrován zpět do klientské aplikace, kde vidí obecný přehled. Na této úvodní stránce vidí uživatel sebou definované oblíbené domácnosti, místnosti, zařízení, nebo dokonce jenom schopnosti zařízení(např. teplota určité místnosti, vlhkost atd.). Dále je zde možnost vidět různé statistiky domácnosti, kde vedle historie změn prostředí různých místností, může být i celková spotřeba domácnosti.

Klientská aplikace je ve stylu tzv. *Dashboardu*, který se velice hodí pro různé informační systémy a systémy starající se o správu. Klientských aplikací může být větší množství a nemusí být přímo svázány s aktuální. Aktuální aplikace je v tomto kontextu myšlená aplikace, která je vytvořena jako defaultní pro tuto bakalářskou práci(dále budeme uvažovat pouze tato konkrétní). Daná aplikace je vytvořena v podobě webové aplikace, která je přístupná přes internetový prohlížeč. Aplikace disponuje responzivním designem, tudíž je schopno v upravené formě přistupovat k aplikaci přes mobilní zařízení.

Jak již bylo řečeno, klientských aplikací může být několik, protože využívají API, které definuje server a posílají požadavky na daný server. Žádná hlubší aplikační svázанost s touto určitou aplikací nemá vliv na chod systému. Klientská aplikace také komunikuje se zařízeními pomocí protokolu *MQTT*, ale tento protokol je ještě zabalený v protokolu *WebSocket*(více info ??).



Obrázek 3.6: Vizuální návrh klientské aplikace

Vzhled

Vizuální vzhled webové aplikace bude využívat tzv. *tiles*(dlaždice), které představují různé komponenty aplikace(viz obrázek ??). Po levé straně aplikace(v desktop responzivní verzi) je postranní menu, které obsahuje navigační tlačítka, které uživatele přesměrují do jiných částí obsahu. Důležitý prvek je položka s názvem *Homes*, kde je po rozkliknutí uživatel přesměrován na stránku, která obsahuje všechny domácnosti, ve kterých je obsažen. Domácnosti jsou také ve stylu dlaždice, kde ve spodní části obsahuje informaci o roli uživatele, kterou má daný uživatel přiřazenou k domácnosti. V právě spodní části je, pro většinu dlaždic obsažených v aplikaci, tlačítko, které po rozkliknutí poskytne možnosti úpravy a správy dané dlaždice.

Po rozkliknutí dlaždice je uživatel přesměrován do dané domácnosti a opět ve stylu dlaždic jsou dostupné místnosti obsažené v domácnosti. Místnosti mají zase odlišnou nabídku možností, než jak to bylo u domácností. Uživatel v daném prostředí má schopnost vytvořit si svou místo(zda má dostatečná práva). Po dalším rozkliknutí dlaždice místo je uživatel přesměrován do dané místo, kde již vidí připojená zařízení a jejich *capabilities*(schopnosti). Schopnosti, mezi které se může řadit např. teplota místo, vlhkost, ovládání světel atd., jsou barevně odlišeny podle zařízení, do kterého patří např. schopnosti zařízení A budou mít zelenou barvu a schopnosti zařízení B žlutou, atd.

U mobilních zařízení je postranní menu skryto a dlaždice jsou rozprostřeny souměrně na obrazovce podle velikosti displeje daného zařízení. U mobilních telefonů standardních rozměrů jsou tyto dlaždice vertikálně rozpoloženy v jednom sloupci. V pravé horní části obrazovky je tzv. *Hamburger menu*(tlačíko), které po rozkliknutí zobrazí postranní menu definované v desktop responzivní verzi.

Programová část

Programová část v kontextu návrhu klientské aplikace se zabývá kompletní strukturou programu, či služeb, které probíhají na pozadí aplikace. Aplikace se sestává z jedné základní šablony, kde ostatní prvky, neboli komponenty, jsou dynamicky měněny. Tím při přechodu na jinou záložku v menu se nemusí vyrenderovat celá stránka, ale pouze změněné komponenty. Klientská aplikace si udržuje stav, kterým právě uživatel disponuje. Díky ukládání stavu do cache prohlížeče uživatele je zamezeno několikanásobné posílání dotazů na server a plynulejší průchod aplikací. Při každém požadavku na server a následném zpracování dat se změní i stav požadovaných informací uložených v cache prohlížeče.

Klientská aplikace je pomocí adaptéra na daný programovací jazyk připojena k autentizační/autorizační službě se kterou inicializuje spojení a dále komunikuje. Po autentizaci uživatele aplikace získá informace o uživateli spravovaném autentizační službou pro další zpracování. Zde se uloží atributy uživatele do objektu, který se následně uloží do autentizačního stavu. Stav, který se stará o autentizovaného uživatele obsahuje také *access token*¹³ a *refresh token*¹⁴. Autentizační služby nabízejí klientské adaptéry s možností autonomního revalidování, nebo-li získání nového tokenu. Programátor se tak nemusí starat o vypršení limitu tokenu.

Další vlastnost klientské aplikace je pracovat přímo se zařízeními i při výpadku serveru. V tomto případě však musí být zařízení už inicializovány na serveru a přidány do místo. Poté lze přistupovat k místo i bez aktivity serveru, kde se po rozkliknutí dané místo

¹³ **Access Token** - token pro přímý přístup ke zdroji informací; krátká životnost

¹⁴ **Refresh Token** - token pro získání nového *access* tokenu; dlouhá životnost

vytvoří instance *MQTT klienta*, který dále přímo komunikuje přes *MQTT broker* s danými zařízeními. Pro každou místnost se inicializuje jeden klient a pokud v klientské aplikaci není rozkliknuta místnost spravující daná zařízení, klient inicializován není. To ušetří mnoho instancí klientů, což je výhodou u některých *MQTT brokerů*, které povolují pouze dané množství klientů a nezahlcuje se tak ve velkém množství sítí.

3.4 Návrh HW zařízení

Je to zařízení, které je schopno reagovat na vstupy, nebo akce a schopno generovat příslušný výstup. Zařízení v konceptu řešení této chytré domácnosti je soubor součástek spojených v elektrickém obvodě lokalizovaných na *DPS*¹⁵. Hlavní komponentou celého zařízení je mikrokontrolér, který se stará o všechny akce spojené s daným zařízením. Mikrokontrolér se dá považovat jako mozek daného zařízení(více v kapitole ??).

Mikrokontrolér obsažený v zařízení se stará i o připojené vnější periférie, v kontextu této práce o tzv. *capabilities*(schopnosti zařízení). *Schopnosti zařízení* jsou připojené I/O periférie k *MCU*(např. teploměr, vlhkometr, tranzistory ovládající světla,...). HW zařízení jsou koncipovaná tak, aby byla jednoduše *customizovatelná* pro zákazníkovy potřeby. To znamená, že *schopnosti zařízení* se dají lehce připojit, odpojit, či přidat jiný modul.

V této bakalářské práci jsou braná v potaz dva typy připojených zařízení:

- **Dedikovaná energeticky úsporná**
- **Centrální pro místnost**

Dedikovaná energeticky úsporná

První typ zařízení jsou dedikovaná energeticky úsporná zařízení, kde napájení pochází většinou z nějakého akumulátoru a musí být maximálně energeticky nenáročná. Tato zařízení by měla vydržet dlouhou dobu bez jakéhokoliv zásahu a nabíjení akumulátoru. K tomuto typu zařízení lze pouze připojit malé množství I/O periferií, neboli *capabilities*. Tato zařízení mají malou velikost a slouží pouze k vykonávání dedikovaných operací bez značné míry *customizace* za účelem efektivního přístupu ke zdrojům. Mezi tyto schopnosti zařízení se řadí např. senzory, ovládání chytrých zásuvek(lze napájení získat jiným způsobem).

Centrální pro místnost

Další typ zařízení jsou tzv. *Centrální pro místnost*. Tento typ zařízení je *customizovatelný* ve velké míře a vyžaduje stabilní příjem energie, protože energetická náročnost těchto zařízení je vyšší oproti prvnímu typu a zařízení by vydržela pouze pár dní s pomocí akumulátoru. K tomuto typu zařízení lze připojit větší množství I/O periferií, neboli *capabilities*. Zařízení jsou dostupná v místnostech a disponují základními schopnostmi, jako je měření teploty, vlhkosti, správa světelých aparátů atd.

¹⁵DPS - Deska Plošných Sponů

Komunikace a správa zařízení

Komunikace zařízení s ostatními zařízeními, serverem, nebo s brokerem probíhá pomocí protokolu *MQTT*(více info viz ??). Před prvním použitím zařízení se musí zařízení inicializovat a získat potřebná data ze serveru. Existují dva přístupy, jak se připojit k serveru a aktivovat tak dané zařízení:

- **Inicializace zařízení** - První použití zařízení
- **Připojení zařízení** - Opětovné použití zařízení

Inicializace zařízení

Před prvním použitím zařízení vytvoří *WiFi AP*¹⁶. Uživatel se přihlásí k danému AP pomocí hesla přiloženém k zařízení. Zařízení má jednoznačný identifikátor, tudíž by neměla nastat kolize více zařízení. Když je uživatel připojen, je vyžádán, aby otevřel určitou webovou stránku, která je zprostředkována zařízením - jedná se o konfigurační stránku zařízení. Po otevření webové stránky se uživateli zobrazí dostupné WiFi AP, uživatel vybere jeho domácí WiFi AP(připojení k routeru, který je v dané domácnosti). Dále uživatel zadá webovou stránku, nebo IP adresu *MQTT brokeru*, která je také přístupná s dodaným softwarem, nebo zprostředkována administrátorem služeb. Další základní parametr, který uživatel musí zadat, je identifikační číslo domácnosti.

Dále je možnost na dané stránce objevit výběr požadovaných schopností, které jsou dostupné namapované na daný port určitého mikrokontroléru. Každý mikrokontrolér má většinou odlišné možnosti ve schopnostech poskytnout určitou funkcionalitu na port. Konfigurace zařízení je tak dokončena a uložena do paměti flash daného mikrokontroléru.

Zařízení poté pošle *MQTT* zprávu na určené téma, kterou odchytí server. Zpráva obsahuje název zařízení a dále pole *capabilities*(schopnosti). U těchto schopností zařízení je přítomný typ schopnosti a pin, na kterém daná schopnost pracuje. Zpráva v poslední řadě obsahuje tzv. *Message ID*(identifikátor zprávy), který slouží k identifikování odpovědi ze serveru.

Server zpracuje zprávu od zařízení a uloží zařízení a schopnosti do databáze. Server zpět zařízení pošle zprávu typu *CREATE* s příslušnými identifikátory zařízení a schopností. Obsahuje také stejný identifikátor zprávy. Zařízení uloží do flash paměti dané identifikátory, zařízení je aktivováno a odebírá zprávy z určitého *topicu*.

Připojení zařízení

Zařízení by v tuto chvíli mělo být již inicializované. Po zapnutí si zařízení přečte uložená data z flash paměti. Připojí se k danému WiFi AP a pošle zprávu typu *CONNECT* s příslušnými identifikátory. Server tak ověří, že zařízení má správně definovány příslušné identifikátory schopností a pošle zpět aktualizovaný seznam. Zařízení se tímto úkonem aktivuje.

¹⁶**WiFi AP** - WiFi Access Point (přístupový bod - samostatný přijímač/vysílač)

Správa zařízení v místnosti

Další procedurou, kterou lze vykonat u zařízení je přidání zařízení do domácnosti. Zařízení musí být vždy obsaženo v místnosti. Když je zařízení přidáno do místnosti pomocí klientské aplikace, server po aktualizování záznamu o zařízení v databázi pošle zprávu zařízení o přidání do místnosti. Zařízení tak začne posílat hodnoty ze svých schopností na dané *topicy*. Také začne přijímat zprávy z daných topiců. V případě odebrání zařízení z místnosti, je znova zařízení poslaná zpráva ze serveru a zařízení přestává posílat svá data ze schopností a čeká do té doby, než je zase přiřazeno do místnosti.

Smazání zařízení z domácnosti

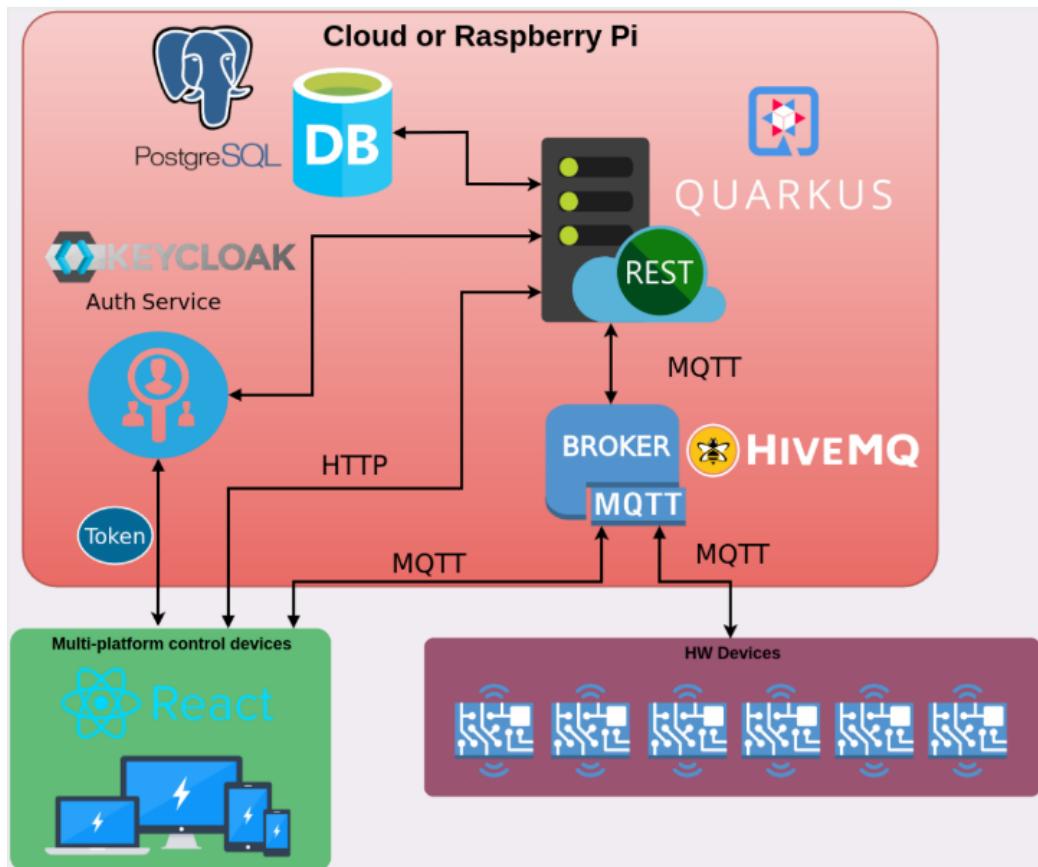
Když je zařízení smazáno pomocí klientské aplikace, odebráno z domácnosti, nebo smazaná domácnost, zaříze potom vymaže vše svá uložená data a přepne se do režimu, kdy musí být znova inicializováno. Uživatel tak musí znova nastavit vše potřebné.

Kapitola 4

Použité technologie a komponenty

V této kapitole jsou popsány technologie, služby a komponenty, které jsou využity pro implementaci návrhu řešení chytré domácnosti. Použití všech těchto prvků bylo pečlivě zváženo a snaha využít nejlepší dostupné technologie a techniky, podle mého názoru, byla úspěšná. Na obrázku ?? jsou znázorněny technologie, či služby, zakomponované v architektuře systémů.

V podkapitole ?? jsou technologie, které byly využity na implementaci **serverové části**, v podkapitole ?? týkající se **databáze**, v podkapitole ?? vše použité k implementaci **klientské webové aplikace** a v poslední podkapitole ?? technologie a komponenty použité pro implementaci **hardwareové části** projektu.



Obrázek 4.1: Použité služby v architektuře

4.1 Serverová část

Jak již bylo řečeno v návrhu serverové části řešení chytré domácnosti, implementace serverové části je asi nejkomplikovanější z celého vývoje řešení (pro více info viz ??). Pro účely vývoje serverové aplikace byl využit Java framework zvaný *Quarkus* (detailní popis ??). Pro síťové služby spojené se serverovou částí byla využita sada nástrojů s označením *Vert.x*.

Quarkus



Obrázek 4.2: Quarkus logo

V dnešní době, kdy žijeme v době cloudových služeb, IoT a open-source projektů přichází do popředí kontejnery (*containers*), mikroslužby (*microservices*), reaktivní programování, cloud-native aplikace a spousty dalšího. Díky těmto novým architekturám, nástrojům, přináší vývoj aplikací jiný rozměr a to speciálně větší produktivitu a výkonnost aplikací. Programovací jazyk Java už je na trhu přes 20 let a stálé zůstává mezi nejpopulárnějšími programovacími jazyky na světě. V informačních technologiích se však technologie a architektury mění velice rychle a je téměř nemožné využívat technologie přes 20 let bez větších změn. Aplikace v kontejnerech by měly mít co nejmenší velikost, rychlý start při restartu a být škálovatelné. To však v případě dosavadních Java aplikací nebylo až tak možné. Framework *Quarkus* by však měl vše změnit.

Quarkus je Kubernetes¹ Native Java framework, který nese označení *Supersonic Subatomic Java* (nadzvuková subatomární Java). Tento framework je přímo ušitý pro GraalVM² a HotSpot (klasická JVM³). Je složen z dostupných Java knihoven a standardů, které patří mezi ty nejlepší svého typu. Vyznačuje se termínem zvaným *Container First*, kde celý framework je založený na nasazení aplikací v kontejnerech a dále v cloudových službách. ??

Na obrázku ?? je porovnání frameworku *Quarkus* s tradičním *Cloud-Native* prvkem a dále porovnání vytvoření spustitelného souboru do nativního kódu pomocí *GraalVM* a klasické využití JVM pomocí *OpenJDK*⁴. Hodnoty jsou určeny pro klasickou *REST*⁵ architekturu rozhraní a dále s přidanými operacemi *CRUD*⁶. První horizontální polovina obrázku se zabývá pamětí celého programu a závislostí. Oproti tradičnímu *Cloud-Native* prvku je program vygenerovaný do nativního kódu až 10x menší. V *Cloud-Native* aplikací je to opravdu znatelný rozdíl. V druhé polovině je k zahlednutí nastartování celé aplikace a první odpověď ze serveru při požadavku. Rapidní snížení rychlosti oproti klasickému tradičnímu *Cloud-Native* prvku a v tom dokáže být *Quarkus* rychlejší více než 250x. ??

¹Kubernetes - orchestrace kontejnerů na úrovni OS

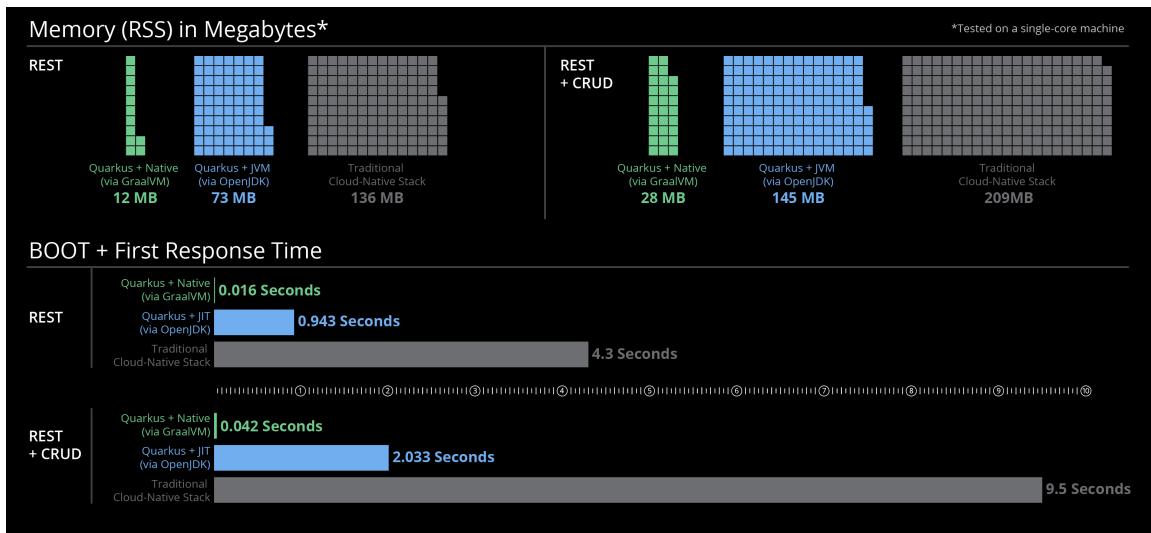
²GraalVM - generuje nativní kód

³JVM - Java Virtual Machine

⁴OpenJDK - Open Java Development Kit

⁵REST - REpresentational State Transfer (architektura rozhraní)

⁶CRUD - Create, Read, Update, Delete (Vytvořit, Číst, Aktualizovat, Smazat)



Obrázek 4.3: Porovnání frameworku Quarkus s alternativami

Vert.x



Obrázek 4.4: Vert.x logo

Vert.x je sada nástrojů pro vytvoření reaktivních aplikací, kde hlavní předností je malá velikost, nízké nároky na výpočetní prostředky a rozsáhlý ekosystém. Samotný *Quarkus* je založený na *Vert.x* technologii a skoro všechny síťové featury v technologii *Quarkus* závisí na technologii *Vert.x*. Tato technologie přináší mnoho zajímavých vylepšení, jako jsou neblokující požadavky na server, jednoduchá souběžnost procesů (*Concurrency*), nebo například podpora několika programovacích jazyků. ??

4.2 Databázová část

Hibernate ORM



Obrázek 4.5: Hibernate logo

Hibernate ORM (dále pouze jako *Hibernate*) je ORM⁷ framework vyvíjen společností *Red Hat*. Poskytuje možnost mapování objektově orientovaného modelu do relační databáze. Vývojář tak není zatížen vytvářením mezivrstvy, které konvertuje objekty a vytváří SQL dotazy na databázi. Vývojář tak může definovat tabulkou databáze dvěma způsoby:

1. Pomocí XML⁸ mapování, kdy vývojář vytvoří XML soubor s definovanými atributy tabulkou a jejich omezeními.
2. Vývojář pouze vytvoří POJO⁹, kde definuje tabulkou v relační databázi. Pomocí anotací se označí atributy třídy, které se namapují do tabulky databáze. Tímto způsobem lze vytvořit několik tabulek v databázi a lze i pomocí anotací definovat asociaci mezi tabulkami a kardinalitu asociací.

Hibernate zprostředkuje rozhraní, které lze využít pro dotazy na databázi. Vývojář může vytvořit vlastní metody, u kterých lze definovat dotaz na databázi pomocí HQL¹⁰ jazyka, který je jednodušší a bližší samotným třídám (označovaným jako *entity*).

PostgreSQL



Obrázek 4.6: PostgreSQL logo

PostgreSQL je objektově-relační, nejpokročilejší open-source databázový systém. PostgreSQL byl vyvinut na Kalifornské univerzitě v Berkeley. Systém je vydáván pod MIT licencí, tudíž ho lze svobodně distribuovat a měnit. Byl hlavně navržen pro UNIXové systémy, ale později byl navržen tak, aby byl přenositelný na více platform (např. Max OS X, Solaris a Windows). Vyžaduje pouze minimální úsilí na jeho údržbu díky jeho stabilitě.??

Má za sebou více než dvacet let aktivního vývoje a má vynikající pověst pro svou spolehlivost a bezpečnost. Výkonnostně nezaostává za srovnatelnými komerčními systémy a v častokrát je i přední.?? PostgreSQL byl vybrán pro účely implementace řešení této chytré domácnosti pro zmiňovanou stabilitu, spolehlivost a vysoký výkon.

⁷ORM- Objektově Relační Mapování

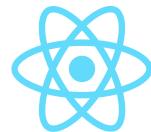
⁸XML - Extension Markup Language (značkovací jazyk)

⁹POJO - Plain Old Java Object (obyčejný Java objekt)

¹⁰HQL - Hibernate Query Language (dotazovací jazyk podobný SQL)

4.3 Klientská aplikace

React



Obrázek 4.7: React logo

React je JavaScript knihovna pro vytváření uživatelských rozhraní. Tato knihovna je udržována společností Facebook a komunitou developerů a společností. React může být využit jako základ pro tvorbu single-page nebo mobilních aplikací, protože je optimální pro práci s rychle se měnícími daty. React se pouze zabývá renderováním dat do *DOM*¹¹. V aplikaci je dále použita další klíčová komponenta a to je *React Router*, který se stará o routování(směrování) v aplikaci. Dále je s aplikací propojena i knihovna pro *state management*¹² a to konkrétně *Mob.X.??*

MobX



Obrázek 4.8: MobX logo

MobX je knihovna, která tvoří *state management* jednoduchý a škálovatelný pomocí *TFRP*¹³. Knihovny *React* a *MobX* jsou velmi mocná kombinace, protože:

- React renderuje stav aplikace takovým způsobem, že poskytuje mechanismus na překlad do stromů *renderable*(schopných být renderovány) komponentů.
- MobX poskytuje mechanismus uložení a aktualizování stavu aplikace, který poté *React* použije.

Jak *React* tak i *MobX* poskytují optimální a jedinečná řešení běžných problémů ve vývoji aplikací. *React* poskytuje mechanismus, který optimálně renderuje uživatelské rozhraní použitím virtuálního *DOM*, který redukuje počet nákladných *DOM* mutací. *MobX* poskytuje mechanismus, který optimálně synchronizuje stav aplikace s *React* komponenty pomocí *RVDSG*¹⁴, který je pouze aktualizovan v případě, kdy je striktně potřeba.??

¹¹DOM - Document Object Model (objektový model dokumentu)

¹²State management - (management stavu)

¹³TFRP - Transparent Functional Reactive Programming (transparentní použití funkcionálního reaktivního programování)

¹⁴RVDSG - Reactive Virtual Dependency State Graph (Reaktivní virtuální závislostní stavový graf)

4.4 Hardwarová část

V této podkapitole jsou obsaženy moduly, které jsou využity v daném řešení chytré domácnosti. Konkrétní zařízení, která jsou v kontextu této práce jsou sestaveny pomocí různých modulů obsahující mikročip *ESP8266*(více v podkapitole *Moduly obsahující čip ESP8266*). Pro nasazení služeb mimo cloud v lokální síti je použit výkonný mini počítač s označením *Raspberry Pi 4B*(pro obecné info viz podkapitolu *Minipočítač Raspberry Pi*).

Moduly obsahující čip ESP8266

ESP8266 je levný WiFi mikročip, který s přidáním několika komponentů je schopen plnohodnotně plnit úlohu mikrokontroléra. Tento mikročip je velice oblíbený v oblasti chytrých zařízení díky své stabilitě, výkonosti, velikosti, ceně a mnoho dalšího. [?] Na trhu existuje spousty modulů, které obsahují právě zmíněný mikročip a plní tak úlohu mikrokontroléra.

Mezi nejoblíbenější moduly, které obsahují tento mikročip jsou *ESP-01*, který obsahuje pouze 2 GPIO¹⁵ piny, poté větší *Wemos D1 mini* a *NodeMCU*. Tento projekt chytré domácnost je sestaven hlavně z těchto modulů díky svým přednostem a využití. Pro maximálně 2 jednotlivé I/O¹⁶ periferie připojené k modulu je využit modul *ESP-01*. Pro komplexní využití a připojení více periferií je použit modul *Wemos D1 mini*.



Obrázek 4.9: Samostatný ESP8266 čip

Minipočítač Raspberry Pi

Raspberry Pi je v informatice označení pro jednodeskový počítač, který je se svým výkonem srovnatelný se slabším stolním počítačem. Obsahuje video a audio výstupy, ethernetový port, USB porty a výstup na dedikovaný monitor určený přímo pro *Raspberry Pi*. Tento malý počítač, rozměry podobný kreditní kartě, je využit v této bakalářské práci na nasazení služeb, kdy uživatel nechce využívat cloudové služby.

Modelů *Raspberry Pi* je na trhu více, kde se liší svým výkonem, velikostí, nebo kompatibilitou s různými rozhraními. Za účelem nasazení služeb v chytré domácnosti lze využít i menšího sourozence z rodiny počítačů *Raspberry Pi* a to přesně *Raspberry Pi W*, kde vytvořené, nebo poskytnuté služby jsou optimalizované pro běh na zařízeních s menším výpočetním výkonem a operační pamětí.

¹⁵**GPIO** - univerzální vstupní/výstupní pin (General-purpose input/output)

¹⁶**I/O** - vstup/výstup (Input/Output)



Obrázek 4.10: Raspberry Pi 4

4.5 Autentizační/autorizační služba

Pro účely autentizace uživatelů v klientské aplikaci a u serveru je využita open-source služba s názvem *KeyCloak*. Tento produkt využívám ve své práci pro autentizační služby a správu uživatelů. Vývojář se tak nemusí starat o persistenci uživatelů a jejich správu. O vše se stará služba *KeyCloak*. *KeyCloak* poměrně dobře znám, protože jsem jedním z aktivních přispěvatele do produktu.

KeyCloak



Obrázek 4.11: Keycloak logo

KeyCloak je open-source služba, která se stará o autentizaci a autorizaci uživatelů a další předností je *Identity and Access Management*(Management přístupu a identit), kde administrátor může upravovat práva uživatelů, přidávat do skupin s určitými právy apod. Tato služba je podporovaná firmou *Red Hat* a několik let už je v popředí popularity služeb, které se starají o bezpečnost webových aplikací. *KeyCloak* disponuje rozsáhlou komunitou vývojářů, kteří jsou velice aktivní v příspívání do daného komunitního produktu. *KeyCloak* obsahuje spousty nových vylepšení a je opravdu velice dobré škálovatelný(*scalable*) a upravovatelný(*customizable*).

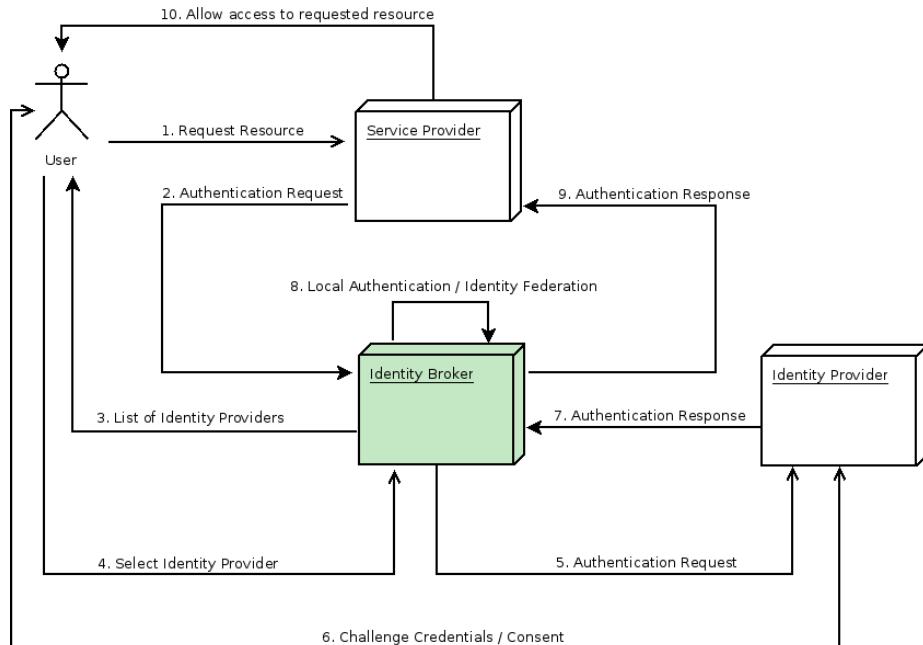
Obsahuje spousty adaptérů pro klientské aplikace, tudíž nezáleží v takové míře na programovacím jazyce. Využívá standardní zabezpečovací protokoly, mezi které patří *OIDC*, *OAuth 2.0*¹⁷ a *SAML 2.0*¹⁸. Lze dokonce využít i asociované autentizační a autorizační služby třetích stran pro udělení přístupu uživateli, registraci a mnoho dalšího. Lze získat přístup i od tzv. *Social providers*(poskytovatelů sociálních sítí) jako jsou např. *Facebook*, *Google*, *Twitter* apod.

¹⁷**OAuth 2.0** - autorizační protokol

¹⁸**SAML 2.0** - Security Assertion Markup Language(autentizační/autorizační protokol)

Na obrázku ?? je popsán autentizační proces uživatele. V prvním kroku uživatel žádá o zdroj informací u aplikace, která je asociována s *KeyCloak* aplikací (v mém případě s mojí *backend* aplikací). Aplikace pošle na službu *KeyCloak* autentizační požadavek, služba se podívá, zda už je uživatel autentizován a když ano, pošle aplikaci odpověď o úspěšném přihlášení a aplikace povolí uživateli přístup k datům.

V opačném případě je uživateli poskytnut seznam poskytovatelů služeb starající se o identitu uživatelů. Uživatel si může podle definovaného autentizačního toku (*Authentication flow*) vybrat jakým způsobem se autentizuje. Když vše proběhne v pořádku, *KeyCloak* pošle odpověď aplikaci o úspěšném přihlášení a uživatel dostane přístup k datům.



Obrázek 4.12: Autentizační tok *Keycloak*

4.6 Komunikace pomocí *MQTT* protokolu

V této sekci lze nalézt knihovny a technologie určené pro bezproblémovou komunikaci pomocí *MQTT* protokolu. Nejdůležitější komponentou komunikace přes *MQTT* protokol je *MQTT Broker*(více info viz. ??). Instance pro poskytnutí brokeru byla využita služba *HiveMQ*.

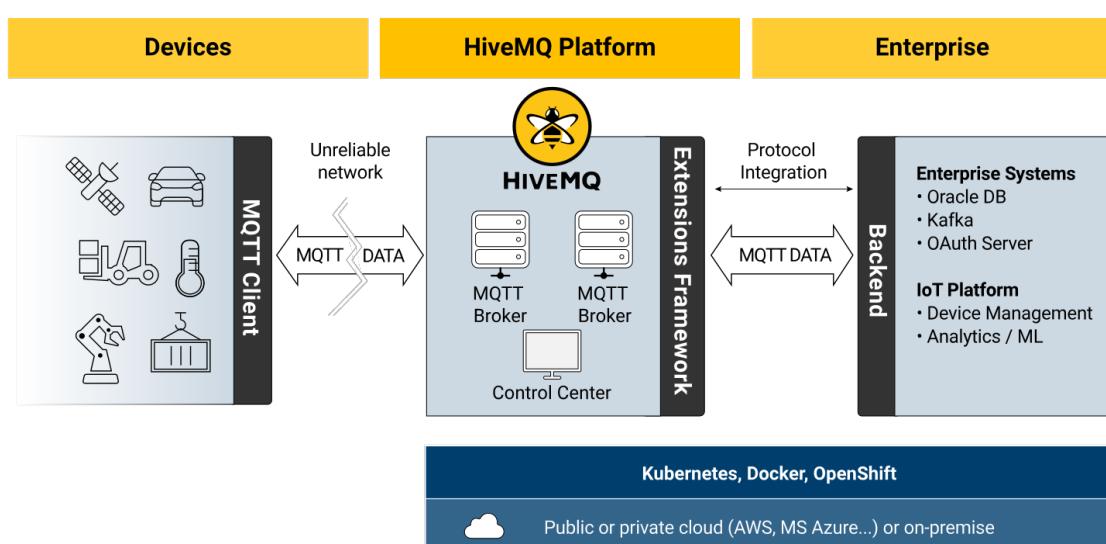
Každá část této aplikace(serverová část, klientská část, zařízení,...) komunikující přes daný protokol, musí disponovat knihovnou, která poskytuje rozhraní a implementaci standardních operací nad protokolem(*MQTT klient*). Pro serverovou část a pro klientskou část to byla zvolena knihovna *Eclipse Paho*. Zvolená knihovna poskytuje knihovnu pro několik programových jazyků a je velice rozšířená a oblíbená. Pro HW zařízení byla zvolena knihovna *PubSub* od autora se jménem *Nick O'Leary* díky své jednoduchosti, spolehlivosti a výkonnosti.

HiveMQ

HiveMQ je *MQTT broker* komunikační platforma navržená pro rychlý, efektivní, spolehlivý pohyb dat DO a Z připojených *IoT* zařízení. Používá protokol *MQTT* pro okamžité obousměrné toky dat mezi zařízením a podnikovými systémy. *HiveMQ* je stvořen tak, aby dokázal čelit technickým výzvám, které organizace vyžadují u vytváření nové *IoT* aplikace.

Prvky, které jsou nejdůležitější a *HiveMQ* je řeší:

- Vytvoření spolehlivé a škálovatelné *IoT* aplikace.
- Rychlé dodání dat pro splnění očekávání koncových uživatelů.
- Nižší provozní náklady díky efektivnímu využití hardwarových, sítových a clouдовých zdrojů.??



Obrázek 4.13: Enterprise architektura MQTT

Paho knihovna



Obrázek 4.14: Eclipse Paho logo

Paho projekt byl vytvořen pro poskytnutí škálovatelných open-source implementací otevřených a standardních komunikačních protokolů zaměřených na nové a existující aplikace pro *M2M*¹⁹ a *IoT* síť. *Paho* odráží přirozené fyzické a náklady omezující možnosti připojení zařízení. Cíle obsahují efektivní úroveň oddělení mezi zařízeními a aplikacemi, navrženou pro podporu rychlého růstu škálovatelného webového a podnikového middleware a aplikací.??

¹⁹M2M - Machine to Machine

Kapitola 5

Implementace

V této kapitole je zahrnuto, jakým způsobem se dané řešení chytré domácnosti implementovalo. Návrh řešení a použité technologie ji byly popsány dříve (návrh v kapitole ?? a použité technologie v kapitole ??). Návrh daného řešení byl směrodatný a obsahuje vše potřebné ke schopnosti implementovat dané řešení. V kapitole použitých technologií byly zavedeny také důvody vběru daných technologií a modulů.

Řešení chytré domácnosti je téma, které obsahuje různorodé prostředí obsažené v komplexním systému. Jedná se o spojení serverové části, klientských aplikací, databáze a samotných hardwarových zařízení. Návrh daného řešení je klíčový. Postupem času implementace daných prvků systému se požadavky, struktura a dokonce i architektura, měnily. Hlavním důvodem bylo to, že při implementaci přicházely různé problémy, které se musely řešit. Na způsob řešení implementačních problémů se vždy nějakým způsobem došlo a různé prvky aplikace jsou sestaveny po inspiraci z různých větší produktů. Tudíž, dané prvky systému by měly být architektonicky sestrojeny správným způsobem.

Implementace systému zahrnuje několik různých aspektů. V podkapitole ?? je popsána implementace serverové části aplikace, v podkapitole ?? implementace klientské aplikace. Dále velice důležitá část implementace je perzistence dat v systému a tudíž databáze. Zde je popsáno mapování, komunikace mezi serverem a databází (viz podkapitola ??). Poslední podkapitola ?? se zabývá implementací daných HW zařízení komunikujících v systému.

5.1 Serverová část

V této podkapitole je zahrnuta implementace serverové části aplikace, kde hlavním jádrem implementace je komunikace s klientskými aplikacemi a HW zařízeními. V části komunikace s danými prvky je nezbytné oddělit *business logiku* od komunikačních prostředků. V aplikaci jsou zřízeny tzv. *services*(služby), které udržují danou *business logiku* ke které přistupují komunikační prostředky(*REST controller*, *komunikace přes MQTT*). Jak již bylo popsáno dříve v návrhu, architektura serverové části aplikace dodržuje předpisy pro *RESTful* aplikaci. Tudíž se zde počítá s bezstavostí požadavku na server. V podkapitolách serverové části budou zahrnutы nejdůležitější, nebo nejzajímavější implementační detaily implementace dané části.

Architektura

Architektura serverové části systému(dále pouze jako *aplikace*) je klíčová. Aplikace je strukturována jako *multi-module project*(projekt obsahující více modulů).

Tato struktura aplikace má spousty výhod:

- **Separace logiky** - Oddělení *business logiky* od dalších částí projektu. Moduly do sebe nezasahují.
- **Vysoká škálovatelnost** - Projekt lze jednoduše rozšířit. Přidání dalšího modulu je jednoduché a není potřeba předělávat celou aplikaci. Platí i v opačném případě při odebrání modulu z projektu.
- **Udržovatelnost** - Vysoká míra udržovatelnosti projektů a modulů. Projekt je snadno upravitelný.
- **Zachování API** - *API* je v rámci celé aplikace většinou ve velké míře neměnné a mění se pouze implementace daného rozhraní s rozdílnou logikou.

Jak je vidět z výhod této struktury aplikace, je velice užitečná ve vývoji rozsáhlější aplikace. Dané aspekty jsou velice přínosné pro udržovatelnost a hlavně škálovatelnost aplikace, která je klíčová. Moduly jsou vytvořeny pomocí buildovacího nástroje *Maven*¹. U většiny daných modulů se generuje samostní *JAR*² spustitelný soubor, který je poté zkombinován s dalšími do jednoho spustitelného souboru, který představuje celou serverovou aplikaci.

Základní moduly této aplikace jsou:

- **API** - Definuje rozhraní pro přiložené moduly.
- **Authz** (*Authorization* - Autorizace) - Stará se o vlastní autorizaci celé aplikace.
- **Controller** ("Ovladač") - Řízení požadavků na server
- **Dist** (*Distribution* - Distribuce systému) - Stará se o kompozici modulů a vytváří tak komplexní aplikaci.
- **Persistence** (Peristence dat) - Rodičovský modul modulů starající se o perzistenci dat. Aktuálně pouze submodul *JPA*.
- **Protocols** (Protokoly) - Komunikace aplikace pomocí dalších protokolů (*MQTT*, *WebSockets*).
- **Services** (Služby) - Udržuje *business logiku* celé aplikace.

¹Maven - nástroj na management projektu

²JAR - Java ARchive

Přijímání požadavků

O přijímání požadavku se stará implementace rozhraní *JAX-RS*³ a to přesněji *RESTeasy*. Tento framework používá anotace na usnadnění vývoje aplikací s daným *API*. Přijímá a odesílá základní data pouze ve formátu *JSON*. Klient pošle požadavek na daný *REST endpoint*, ke kterému je svázána nějaká operace(metoda), která se vykoná. Každá třída implementující rozhraní pro přístup k *resources*(zdrojům) je vlastně *Java Bean*⁴ se životností pouze po dobu vykonávání určitého požadavku (*RequestScoped*), poté je destruována.

V kódu níže je ukázka rozhraní, které vyobrazuje nastavení pro *RESTeasy* framework. Dané rozhraní se nachází v modulu *API* a implementace daného rozhraní v modulu *Controller*. Jako první anotace použita na dané rozhraní je *@Path*, v které se definuje cesta k danému zdroji. Dále následuje anotace *@Consumes* a *@Produces*, u kterých je definovaná hodnota *MediaType.APPLICATION_JSON*. Znamená to, že implementace daného rozhraní komzumuje a produkuje pouze informace ve formátu *JSON*.

Anotace *@Transactional* značí, že požadavek na daný zdroj je brán transakčně, tudíž splňovat požadavky *ACID*⁵. Poslední *class annotation*(anotace třídy) s názvem *@Authenticated* značí, že uživatel musí být autentizován před vykonáním specifické operace.

Anotace *@GET* a *@POST* definují typy požadavku. Lze použít i anotaci *@Path* na metody rozhraní, např. kdyby u metody byla přidaná anotace *@Path* s obsahem *"/info"*, cesta k danému zdroji by musela mít podobu konkrétně *"/homes/info"*. Anotace *@Path* pracuje i parametry. Uvažujme například *URI* *"/homes/3"*, které je možné namapovat pomocí anotace *@Path* s parametry *"/homes/{id}"*.

```
1  @Path("/homes")
2  @Consumes(MediaType.APPLICATION_JSON)
3  @Produces(MediaType.APPLICATION_JSON)
4  @Transactional
5  @Authenticated
6  public interface HomesResource {
7
8      @GET
9      Response getAll();
10
11     @POST
12     HomeModel createHome(String JSON);
13
14     ...
15 }
```

Výpis 5.1: Ukázka deklarování rozhraní pro správu domácností

³**JAX-RS** - Java API for RESTful Web Services(rozhraní pro *RESTful* služby)

⁴**Java Bean** - Java EE komponenta

⁵**ACID** - Atomicity, Consistency, Isolation, Durability (Atomicita, Konzistence, Izolace a Trvanlivost)

Předávání stavu při požadavku na server

Jak již bylo v předchozí kapitole určeno, framework na přijímání požadavku musí určit, jaké třídě, nebo metodě, směrovat požadavek. To lze přidáním anotace `@Path`, v které se definuje *URI*⁶, který se musí shodovat s cílem požadavku. Každá třída reagující na požadavky od klientů má většinou definovanou cestu zdroje, o který se stará. V úryvku kódu výše to bylo např. `"/homes"`. V tomto úskalí však přichází problém.

Problém

Systém řešení chytré domácnosti disponuje strukturou hierarchického rozpoložení zdrojů (např. `homes -> rooms, rooms -> devices, devices -> capabilities,...`). *URI* každého zdroje obsahuje identifikátor každého prvku (např. `"/homes/42"` díky kterému lze mapovat určitý jedinečný prvek. Co když ale přijde situace, kdy potřebujeme dostat informace uložené ve schopnosti zařízení v domácnosti s identifikátorem 11, v místnosti s identifikátorem 22, zařízení s identifikátorem 33 a schopnost s identifikátorem 44.

Daná cesta ke zdroji vypadá konkrétně:

- `"/homes/11/rooms/22/devices/33/caps/44"`

Namapování takového zdroje pomocí anotace `@Path` by muselo vypadat takto:

- `@Path("/homes/{homeID}/rooms/{roomID}/devices/{deviceID}/caps/{capID}")`

Způsob zápisu je takřka nepoužitelný a nedovoluje škálovatelnost dané správy požadavků. Může přijít situace, kdy budeme nuteni připojit k cestě zdroje do zařízení další jiný zdroj. Poté by jsme museli vytvořit další třídu, která tento ohromný zápis zpracuje. Proto přichází řešení ve formě předávání stavu do různých komponent.

Řešení

Vytvořil jsem *Java Bean* komponentu s názvem *BartSession*, která má životnost také omezenou na vykonávání určitého požadavku. Tato komponenta poskytuje uložení aktuálního stavu požadavku, distribuování *services*(služeb) a informace o autentizovaném uživateli. Každá třída poskytující správu cílového zdroje z požadavku (dále jako *resource* třída) ve svém konstruktoru zahrnuje právě tuto komponentu. Je tak zaručeno, že se stav dokáže předat do každé dané *resource* třídy.

Při daném hierarchickém rozpoložení se lze z jedné *resource* třídy přesměrovat do další, z dané třídy do další a tak dále, než se dorazí do požadované třídy obstarávající zdroj informací. Každá *resource* třída se stará pouze o své vlastní zájmy a nepotřebuje vědět o prostředí mimo ni. Každá *resource* třída disponuje různými metodami pro správu zdrojů. V úryvku kódu níže je vyobrazeno přesměrování požadavku do jiné *resource* třídy s poskytnutím změněného stavu. Kód je pouze ilustrační; obsahuje pouze základní prvky potřebné pro ilustraci řešení.

⁶**URI** - Uniform Resource Identifier (jednotný identifikátor zdroje)

```

1 // Rodicovska trida 'HomesResourceProvider'
2 // implementujici rozhrani 'HomesResource'
3
4 @Path("/homes")
5 public class HomesResourceProvider implements HomesResource {
6     BartSession session;
7     ...
8
9     @Path("/{id}")
10    public HomeResource forwardToSpecificHome(@PathParam("id") Long id){
11        return new HomeResourceProvider(session.setActualHome(id));
12    }
13}

```

Výpis 5.2: Ukázka přesměrování požadavku

```

1 // Podrazena trida 'HomeResourceProvider'
2 // implementujici rozhrani 'HomeResource'
3
4 public class HomeResourceProvider implements HomeResource {
5     BartSession session;
6
7     public HomeResourceProvider(BartSession session){
8         this.session = session;
9     }
10
11    @GET
12    public HomeModel getHome() {
13        return session.getActualHome();
14    }
15    ...
16}

```

Výpis 5.3: Ukázka zpracování požadavku z přesměrované třídy

5.2 Databázová část

Tato sekce pojednává o propojení databázové části se serverem včetně způsobu mapování tabulek pomocí *ORM* frameworku. Uživatel posílající požadavky na server, po náležitých autentizační/autorizačních operacích, dostane data přávě ze zmiňované databáze. Díky databázi jsou perzistentně uložená data, která slouží pro správu celé domácnosti. Pro implementaci databázové části systému byla využita databáze *PostgreSQL*(více info viz. ??). Hlavní entitou databáze je samotná domácnost, kde je vytvořena asociace mezi dalšími tabulkami.

Jak již bylo řečeno v návrhu, mezi databázovou vrstvou a serverovou vrstvou jsou objekty mapovány do relační databáze. Ke schopnosti mapování tímto způsobem je zapotřebí *ORM* framework, který jsem podle mého nejlepšího uvážení vybral konkrétně *Hibernate ORM* (více info viz. ??). U *Hibernate ORM* lze definovat entity pomocí anotací u *POJO*, nebo pomocí *XML*. Pro přehlednější, lépe škálovatelný přístup jsem vybral definování entit pomocí anotací.

Základní entita

Na úryvku kódu níže je ilustrační příklad definování entity databáze u třídy mapující domácnost. Anotace `@Entity` slouží k určení mapování-schopné třídy do relační databáze (povinné). Další anotace `@Table` je nepovinná a slouží k definování parametrů spojených s tabulkou, např. definování jména tabulky. Tato třída implementuje rozhraní *HomeModel*, která definuje rozhraní pro operace nad domácností. Pomocí rozhraní *HomeModel* jsou entity dále, díky polymorfismu, v aplikaci mapovány pouze jako dané rozhraní kvůli udržovatelnosti *API*.

Každý atribut třídy, je označen anotací `@Column`, která představuje sloupec tabulky. U této anotace lze např. definovat název sloupce, zda může sloupec prázdný, atd. Každá tabulka musí mít jednoznačný identifikátor a atribut plnící roli identifikátoru je označen anotací `@Id`. Identifikátor může být také označen anotací `@GeneratedValue`, kdy se *Hibernate* framework stará o přiřazování jednoznačných identifikátorů sám.

```
1  @Entity
2  @Table(name = "Homes")
3  public class HomeEntity implements HomeModel {
4
5      @Id
6      @GeneratedValue
7      @Column(name = "HOME_ID")
8      Long id;
9
10     @Column(nullable = false)
11     String name;
12
13     ...
14 }
```

Výpis 5.4: Ukázka definování entity

Asociace mezi entitami

Jak to bývá zvykem u relačních databází, tak v tabulkách mohou existovat tzv. *foreign keys* (cizí klíče), které ukazují na identifikátor jiné tabulky. Tím je vytvořena jakási asociace mezi tabulkami. Samožejmě zde existuje jakási *kardinalita*, která určuje četnost provázanosti tabulek.

Lze si to představit na příkladu z daného řešení chytré domácnosti. Tabulka představující domácnost je svázána s tabulkou místností ve vztahu 1:N, což znamená, že domácnost může obsahovat několik místností, ale určitá místnost může být obsažena pouze v jedné domácnosti. *ORM* framework *Hibernate* nám umožňuje dokonce i namapovat dané asociace mezi entitami. V úryvku kódu níže je vyobrazen ilustrační kód pro definování asociace 1:N mezi domácností a místnostmi.

Entita vyznačující domácnost obsahuje množinu místností, které jsou připojeny k domácnosti. Anotace `@OneToMany` určuje asociaci ve vztahu 1:N (domácnost:místnosti), kde v parametrech anotace je určena *targetEntity*, v které je definovaná cílová entita a parametr *mappedBy*, který představuje název atributu v místnosti, pomocí kterého je domácnost mapována v dané místnosti.

```
1 // Entita domacnost
2
3 @Entity
4 @Table(name = "Homes")
5 public class HomeEntity implements HomeModel {
6     ...
7
8     @OneToMany(targetEntity = RoomEntity.class, mappedBy = "home")
9     Set<RoomModel> roomsSet = new HashSet<>();
10 }
```

Výpis 5.5: Ukázka asociace 1:N (domácnost:místnosti)

Entita vyznačující místnost obsahuje, jak již dříve bylo napsáno, pouze referenci na jednu domácnost, ve které je daná místnost umístěna. Anotace `@ManyToOne` určuje asociaci, inverzní vůči domácnosti, která definuje kardinalitu vztahů mezi entitami. Anotace `@JoinColumn` slouží na propojení sloupců tabulek.

```
1 // Entita mistnost
2
3 @Entity
4 @Table(name = "Rooms")
5 public class RoomEntity implements RoomModel {
6     ...
7
8     @ManyToOne
9     @JoinColumn
10    HomeEntity home;
11 }
```

Výpis 5.6: Ukázka asociace 1:N (domácnost:místnosti)

5.3 Klientská část

Klientská aplikace se zabývá interakcí mezi uživatelem a systémem. Tudíž je velmi důležitá v kontextu řešení chytré domácnosti. V návrhu klientské aplikace je více informací o struktuře klientské části (více info viz. ??). Klientská aplikace byla implementována v jazyce *JavaScript* s využitím knihovny *React*(více info viz. ??). Další důležitou součástí aplikace je udržovat stav aplikace o který se stará nástroj *MobX*(více info viz. ??).

Základem klientské aplikace je předpřiravená šablona ve stylu *Dashboardu*, která je vydávána pod *MIT* licencí, tudíž je volně šířitelná. Tato šablona je vytvořena pomocí knihovny *Material UI*⁷. Autorem šablony je *Creative Tim*⁸. Ze šablony byla použita pouze kostra programu a některé jednoduché, dobře vypadající komponenty.

Aplikace je vytvořena z komponent, které v sobě obsahují další komponenty a obsažené komponenty obsahují další komponenty atd. Jedná se o kompozici komponent. V úryvku kódu níže je příklad vytvoření komponenty pomocí knihovny *React*.

```
1 export default function TestComponent(){
2     const [state,setState] = React.useState(null);
3
4     React.useEffect(()=>{
5         setState(42);
6     }, []);
7
8     return(
9         <div>
10         <p>This is test component</p>
11         {state && <p>Number from state is: {state}</p>}
12     </div>
13 );
14 }
```

Výpis 5.7: Ukázka asociace 1:N (domácnost:místnosti)

Jak již bylo zmíněno, důležitou součástí klientské aplikace je i stav aplikace o který se stará nástroj *MobX*. Pro aplikaci jsou vytvořeny tzv. *stores*(úložiště), které slouží pro udržování stavu napříč aplikací. Pro tento systém byly zatím vytvořeny základní úložiště a to: *AuthStore*, *HomeStore*, *RoomStore*, *DeviceStore*, *UserStore*.

⁷<https://material-ui.com/>

⁸<https://www.creative-tim.com>

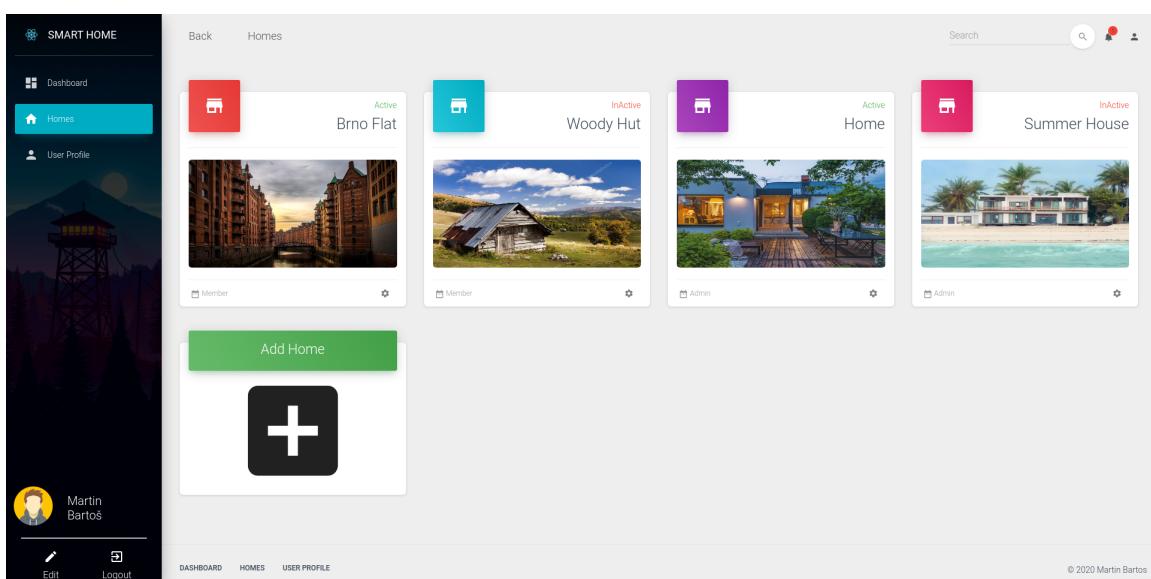
V úryvku kódu níže je příklad definování úložiště pomocí nástroje *MobX*. Nejzajímavější část celého *store* je funkce *decorate*. Tato funkce definuje ve třídě atributy, které budou tzv. *observable*(pozorovatelné) poté definuje *action*(akce) metody, které modifikují stav a *computed*(vypočteno) getter, který vrací daný objekt/objekty.

```

1  export class HomeStore{
2    _homes = new Map();
3
4    setHomes = (homes) =>{
5      this._homes = homes;
6    }
7
8    get homes() {
9      return this._homes;
10   }
11 }
12
13 decorate(HomeStore, {
14   _homes: observable,
15   setHomes: action,
16   homes: computed
17 });

```

Výpis 5.8: Ukázka asociace 1:N (domácnost:místnosti)



Obrázek 5.1: Aktuální vzhled aplikace

5.4 HW část

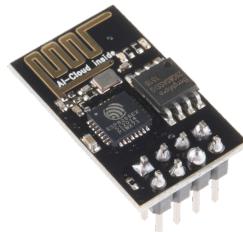
Hardware část systému chytré domácnosti je uzpůsobena pro komunikaci se serverovou částí a klientskou aplikací. Jak již bylo popsáno v návrhu, základem je mikrokontrolér, který slouží jako mozek celého zařízení(více info viz. ??). Moduly, které jsou využity pro sestavení zařízení obsahují mikročip *ESP8266*(více info viz. ??). V návrhů řešení byly zařízení rozděleny do dvou kategorií.

Do první kategorie patří úsporná zařízení, která využívají modul *ESP-01*, který disponuje pouze dvěma GPIO piny a hodí se pro různé samostatné senzory, či ovládání výstupních zařízeních. Příkon tohoto modulu je opravdu nízký a podle mého názoru je tento modul velice vhodný pro dané úlohy.

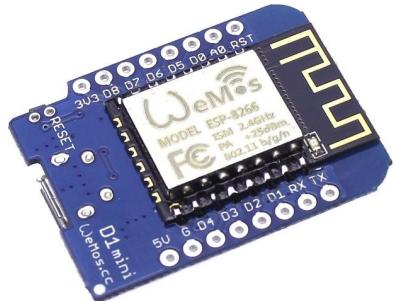
Do druhé kategorie zařízení, které jsou centrální pro celou místnost, byly zvoleny moduly *Wemos D1 mini*, které také využívají čip *ESP8266*. Tento modul využívá 11 GPIO pinů, tudíž lze připojit až 11 I/O jednotek.

Program, který využívají tyto zařízení je naprogramován pomocí programovacího jazyka C++. Kód je předpřipraven pracovat s velkou řadou I/O jednotek. Je také velice dobře škálovatelný pro přidávání nových kompatibilních zařízení(pro více info viz. ??). Jak již bylo zmíněno v návrhu, tak pro zařízení byla zvolena knihovna, která umožňuje zařízení komunikovat přes *MQTT* protokol s názvem *PubSub*. Tuto knihovnu představil autor se jménem *Nick O'Leary* a díky své jednoduchosti, spolehlivosti a výkonnosti jsem ji zařadil jako jádro komunikační části zařízení.

Díky knihovně *WiFiManager*⁹ od uživatele *tzapu* bylo možné využít konfigurační webovou stránku při inicializaci zařízení. Zařízení vytvoří svůj vlastní WiFi AP, na který se uživatel připojí a poté je přesměrován na konfigurační stránku zprostředkovanou pomocí dané knihovny.

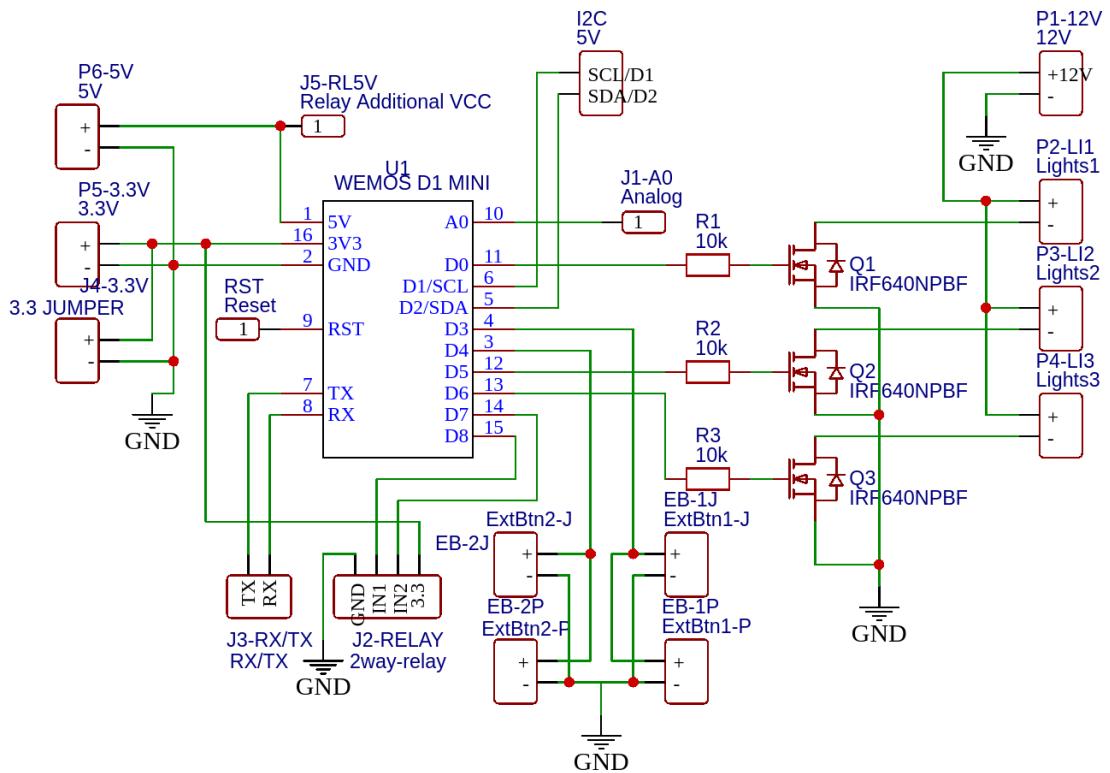


Obrázek 5.2: Modul *ESP-01*

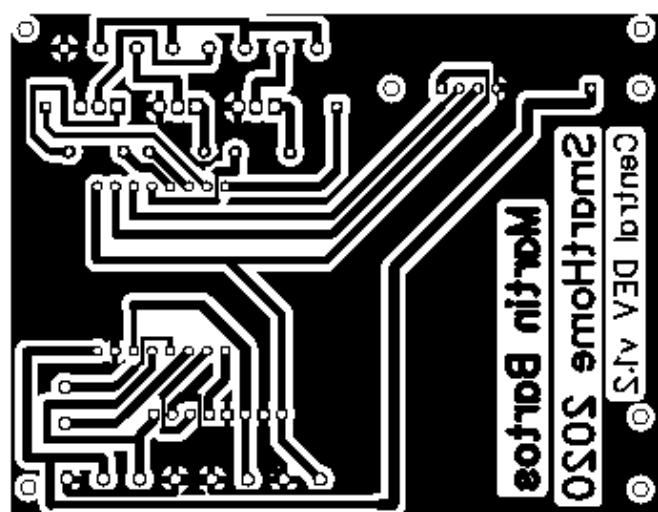


Obrázek 5.3: Modul Wemos D1 mini

⁹<https://github.com/tzapu/WiFiManager>



Obrázek 5.4: Návrh schématu centrálního zařízení



Obrázek 5.5: Návrh DPS pro centrální zařízení

Kapitola 6

Testování

Kapitola 7

Závěr