Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Porovnání open-source integračních frameworků pro oblast IoT

Bc. Maksim Khiuttiulia

Diplomová práce

2021

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Maksim Khiuttiulia**
Osobní číslo:	**I19281**
Studijní program:	**N0613A140007 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Porovnání open-source integracních frameworku pro oblast IoT**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je realizace porovnání vybraných, alespon tří, dostupných open-source integračních frameworků (napr. Node-RED, Flogo, Eclipse Kura apod.) pro oblast IoT, zejména se zameřením na oblast domácí automatizace.
V rámci diplomové práce budou vytvořeny netriviální příklady automatizovaných úloh pomocí vybraných frameworků a bude provedeno jejich porovnání z ruzných úhlů pohledu, např.:
- rychlost vytváření řešení
- možnosti kombinace vizuálních prvku a psaného kódu
- uživatelská přívětivost
- ladění vytvářených programů
- HW nároku frameworku při zpracování rozsáhlejších úloh apod.

Text diplomové práce musí obsahovat metodiku vytváření řešení v každém z posuzovaných frameworků, popis a vysvětlení rešení rozsáhlých příkladů a také zdůvodnění doporučení nejvhodnejšího frameworku dle autora.

Rozsah pracovní zprávy: **50-60 normostran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

*TAMBOLI A. Build Your Own IoT Platform: Develop a Fully Flexible and Scalable Internet of Things Platform in 24 Hours. Apress; 1st ed. edition, 2019, 240 pp. ISBN 978-1484244975.
*GUINARD D., TRIFA V. Building the Web of Things: With examples in Node.js and Raspberry Pi. Manning Publications; 1 edition, 2016, 344 pp. ISBN 978-1617292682.

Vedoucí diplomové práce: **doc. Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **6. listopadu 2020**
Termín odevzdání diplomové práce: **15. května 2021**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2020

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 17. 05. 2022

Maksim Khiuttiulia

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu mé diplomové práce doc. Ing. Michaelu Bažantovi, Ph.D., za podnětné rady a odbornou pomoc, kterou mi poskytoval při zpracovávání mé diplomové práce, a za čas, který mi věnoval.

ANOTACE

Diplomová práce se zabývá porovnáním frameworků pro oblast IoT. V rámci diplomové práce budou porovnány open-source frameworky: Flogo, Node-RED, Zetta. Bude popsána instalace pro každý framework a také budou vytvořeny tři scénáře použití pro každý framework. Na konci bude provedeno porovnání a výběr nejlepšího frameworku z pohledu autora.

KLÍČOVÁ SLOVA

Internet věcí, Flogo, Node-RED, Zetta

TITLE

Comparison of open-source integration frameworks for IoT

ANNOTATION

The diploma thesis deals with a compared framework for the area of IoT. The diploma thesis will cover open-source frameworks: Flogo, Node-RED, Zetta. The installation for each framework will be described, and the 3 scenarios created will also be used for each framework. At the end, a comparison will be made and the best framework will be selected from the author's point of view.

KEYWORDS

Internet of Things, Flogo, NodeRed, Zetta

OBSAH

ÚVOD 16

1 STRUČNÁ HISTORIE INTERNETU VĚCÍ 17

1.1 Historie internetu věcí 17

1.2 Perspektivy rozvoje 18

1.3 Zabezpečení internetu věcí 19

1.4 Protokoly pro výměnu dat mezi IoT zařízeními 20

1.4.1 MQTT 20

1.4.2 REST API 20

1.4.3 CoAP 21

2 PŘEHLED POUŽITÝCH TECHNOLOGIÍ 22

2.1 Zetta 22

2.1.1 Instalace 22

2.2 Node-RED 23

2.2.1 Instalace 24

2.3 Project Flogo 24

2.3.1 Instalace 25

2.4 Přehled zařízení v domácnosti 26

2.4.1 Kávovar 26

2.4.2 Topení 27

2.4.3 Přehrávač hudby 27

2.4.4 Ovladač oken 28

2.4.5 Ovladač světel 28

3 MODELOVÉ SCÉNÁŘE PRO DOMÁCÍ AUTOMATIZACI 29

3.1 Scénář 1: ovládání světel pomocí tlačítka 29

3.1.1 Zetta 29

3.1.2 Node-RED 31

3.1.3 Project Flogo 34

3.2 Scénář 2: návrat domů 36

3.2.1 Zetta 37

3.2.2 Node-RED 40

3.2.3 Project Flogo 44

3.3 Scénář 3: dovolená 48

3.3.1 Zetta 49

3.3.2 Node-RED 51

3.3.3 Flogo 57

4 VÝVOJ MODULŮ AKCÍ PRO PROJECT FLOGO 61
4.1 GPIO Output 64
4.2 Sleep 65
5 POROVNÁNÍ FRAMEWORKŮ 66
5.1 Časová náročnost 66
5.2 Obtížnost vývoje 66
5.3 Náročnost na hardware a software 67
5.4 Možnosti psaní kódu a použití grafických prvků 68
ZÁVĚR 69
POUŽITÁ LITERATURA 70
PŘÍLOHY 72

SEZNAM OBRÁZKŮ

Obrázek 1: Top investic do IoT, zdroj: www.bain.com 19
Obrázek 2: Inicializace projektu, zdroj: vlastní 22
Obrázek 3: Spuštění serveru, zdroj: vlastní 23
Obrázek 4: Webové rozhraní Node-RED, zdroj: vlastní 24
Obrázek 5: Webové rozhraní Flogo, zdroj: vlastní 26
Obrázek 6: Start serveru, zdroj: vlastní 30
Obrázek 7: Zapnutí světelného zdroje, zdroj: vlastní 30
Obrázek 8: Výstupní uzel, zdroj: vlastní 31
Obrázek 9: Výběr kontaktů, zdroj: vlastní 32
Obrázek 10: Přidání uzlu, zdroj: vlastní 32
Obrázek 11: Úpravy vlastností uzlu, zdroj: vlastní 33
Obrázek 12: Výsledný proud, zdroj: vlastní 33
Obrázek 13: Výsledek ve webovém rozhraní, zdroj: vlastní 33
Obrázek 14: Název proudu, zdroj: vlastní 34
Obrázek 15: Nastavení koncového bodu, zdroj: vlastní 35
Obrázek 16: Nastavení triggeru pro scénář 1, Flogo, zdroj: vlastní 35
Obrázek 17: Vyladěný projekt, zdroj: vlastní 36
Obrázek 18: Konfigurace http in modulu, zdroj: vlastní 41
Obrázek 19: Konfigurace modulu switch pro uživatelské jméno, zdroj: vlastní 42
Obrázek 20: Konfigurace modulu switch pro aktuální stav kávovaru, zdroj: vlastní 43
Obrázek 21: Subflow pro ovládání kávovaru, zdroj: vlastní 44
Obrázek 22: Výsledná struktura scénáře 2, zdroj: vlastní 44
Obrázek 23: Přidání větvení do Flogo, zdroj: vlastní 46
Obrázek 24: Nastavení větve pro vypnutý stav kávovaru, zdroj: vlastní 46
Obrázek 25: Nastavení Path params v akci REST Invoke, zdroj: vlastní 46
Obrázek 26: Flow pro kávovar, zdroj: vlastní 47
Obrázek 27: Hlavní flow pro scénář 2, zdroj: vlastní 48
Obrázek 28: Definice uzlu Template v Node-RED, zdroj: vlastní 52
Obrázek 29: Flow pro posílání notifikací v Node-RED, zdroj: vlastní 53
Obrázek 30: Definice podmínky pro HTTP status, zdroj: vlastní 53
Obrázek 31: Definice uzlu plánovače v Node-RED, zdroj: vlastní 55
Obrázek 32: Flow pro spuštění plánovaných úloh v Node-RED, zdroj: vlastní 56

Obrázek 33: Nastavení globální proměnné v Node-RED, zdroj: vlastní.....56
Obrázek 34: Scénář 3 v Node-RED, zdroj: vlastní.....57
Obrázek 35: Akce pro nastavení režimu dovolené v Project Flogo, zdroj: vlastní 58
Obrázek 36: Posílání notifikací v Project Flogo, zdroj: vlastní.....59
Obrázek 37: Funkce pro porovnání času v Project Flogo, zdroj: vlastní.....60
Obrázek 38: Akce plánovače v Project Flogo, zdroj: vlastní 60

SEZNAM TABULEK

Tabulka 1: Časová náročnost vývoje, zdroj: vlastní 66

SEZNAM ZDROJOVÉHO KÓDU

Zdrojový kód 1: Základní inicializace projektu Zetta, zdroj: vlastní 23
Zdrojový kód 2: Ovládání GPIO pinu v Zettě, zdroj: vlastní 30
Zdrojový kód 3: Ovladač kávovaru v Zettě, zdroj: vlastní 38
Zdrojový kód 4: Scout třída pro Zettu, zdroj: vlastní 38
Zdrojový kód 5: Definice zařízení v rámci scénáře Zetty, zdroj: vlastní 39
Zdrojový kód 6: Přidání zařízení do aplikace Zetta, zdroj: vlastní.....39
Zdrojový kód 7: Rozhodování podle uživatelského jména v Zettě, zdroj: vlastní 40
Zdrojový kód 8: Inicializace serveru Zetty pro scénář 2, zdroj: vlastní 40
Zdrojový kód 9: Příklad funkce aktivace senzoru pohybu, zdroj: vlastní 49
Zdrojový kód 10: Funkce pro posílání zpráv policii, zdroj: vlastní.....50
Zdrojový kód 11: Plánovaní úlohy v 7:00, zdroj: vlastní 51
Zdrojový kód 12: Definice vstupních parametrů do aktivity, zdroj: vlastní.....61
Zdrojový kód 13: Definice funkce metadata, zdroj: vlastní 62
Zdrojový kód 14: Definice funkce konstruktoru, zdroj: vlastní 62
Zdrojový kód 15: Příklady definice funkce init, zdroj: vlastní.....62
Zdrojový kód 16: Definice funkcí FromMap a ToMap, zdroj: vlastní.....63
Zdrojový kód 17: Definice funkce Eval, zdroj: vlastní 63
Zdrojový kód 18: Definice souboru descriptor.json, zdroj: vlastní 64

SEZNAM ZKRATEK A ZNAČEK

API – Application Programming Interface

AR – Augmented Reality

GPIO – General Purpose Input Output

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

IoT – Internet of Things

IP – Internet Protocol

JSON – JavaScript Object Notation

LED – Light-Emitting Diode

MQTT – Message Queuing Telemetry Transport

REST – Representational State Transfer

RFID – Radio Frequency Identification

TCP – The Transmission Control Protocol

TLS – Transport Layer Security

VR – Virtual Reality

TERMINOLOGIE

Arduino – jednočipový počítač s mikrokontrolérem od ATmega nebo Atmel

Framework – platforma pro vývoj softwarových aplikací

Raspberry Pi – malý jednodeskový počítač s deskou plošných spojů

Spark Core – jednočipový počítač s architekturou ARM

TensorFlow – otevřená knihovna pro strojové učení

ÚVOD

V posledních desetiletích lidstvo udělalo velký pokrok v technickém směru. Vývoj internetu a chytrých zařízení umožnil udělat život lidí mnohem pohodlnější. Jedna z koncepcí tohoto směru je tzv. internet věcí. Internet věcí je nová technologie, která v sobě zahrnuje několik odvětví, například distribuované výpočty, protokoly komunikace, chytrá zařízení atd. Kvůli internetu věcí už není třeba starat se například o úklid domácnosti, je možné zapnout chytrý vysavač z mobilu. Nebo například často nastává situace, kdy člověk jde do práce a zapomene vypnout světlo v pokoji. Tuto situaci je možné vyřešit pomocí moderních žárovek nebo spínačů, které budou připojeny k internetu a lze je ovládat odkudkoliv. Internet věcí se také využívá v průmyslu. Už není potřeba, aby člověk kontroloval každé výrobní zařízení a sledoval jeho stav. Zařízení v případě poruchy může samo informovat zaměstnance o incidentu. Ve své diplomové práci jsem poskytnul informace o vývoji internetu věcí a jeho aktuálním vývoji a porovnal mezi sebou několik frameworků pro vytvoření chytré domácnosti.

1 STRUČNÁ HISTORIE INTERNETU VĚCÍ

1.1 Historie internetu věcí

Samotný koncept připojených zařízení sahá až do roku 1832, kdy byl vyvinut první elektromagnetický telegraf. Telegraf umožňoval přímou komunikaci mezi dvěma stroji přenášením elektrických signálů. Skutečná historie internetu věcí však začala vynálezem internetu koncem šedesátých let, který se během následujících desetiletí rychle rozvinul.

Prvním připojeným zařízením byl automat Coca-Cola umístěný na Carnegie Mellon University a provozovaný místními programátory. Do automatu zabudovali mikrospínače a pomocí rané formy internetu kontrolovali, zda je chladící jednotka dostatečná pro chlazení nápojů a zda jsou k dispozici plechovky od koly. Tento vynález přispěl k dalšímu výzkumu v této oblasti a vývoji vzájemně propojených strojů po celém světě.

V roce 1990 John Romkey poprvé připojil toustovač k internetu pomocí protokolu TCP/IP. O rok později přišli vědci z University of Cambridge s nápadem použít první prototyp webové kamery ke sledování množství kávy dostupné v konvici na kávu v jejich místní počítačové laboratoři. Naprogramovali webovou kameru tak, aby třikrát za minutu vyfotografovala konvici na kávu a poté snímky odeslala do místních počítačů, což každému umožnilo zjistit, zda je káva k dispozici [8].

Na počátku 21. století se pojem „internet věcí“ rozšířil v médiích a zmiňují jej publikace jako The Guardian, Forbes a Boston Globe. Zájem o této technologie neustále rostl, což vyústilo v 1. mezinárodní konferenci IoT ve Švýcarsku v roce 2008, kde účastníci z 23 zemí diskutovali o RFID, bezdrátové komunikaci krátkého dosahu a senzorových sítích.

K vývoji internetu věcí navíc přispělo několik důležitých událostí. Jednou z nich byla lednička připojená k internetu, kterou společnost LG Electronics představila v roce 2000 a umožnila uživatelům nakupovat online a uskutečňovat videohovory. Dalším důležitým vývojem byl malý robot vytvořený v roce 2005, který byl schopen komunikovat nejnovější zprávy, předpovědi počasí a změny na akciovém trhu [10].

Boom IoT byl podpořen jeho zahrnutím do Gartner Hype pro nové technologie v roce 2011. Ve stejném roce byl veřejně spuštěn IPv6 – protokol síťové vrstvy v srdci IoT.

Od té doby se vzájemně propojená zařízení stala běžnou součástí našeho každodenního života. Globální technologičtí giganti, jako jsou Apple, Samsung, Google, Cisco a General Motors, se zaměřují na senzory a zařízení IoT, od vzájemně propojených termostatů a chytrých brýlí až po bezpilotní vozidla. Internet věcí našel uplatnění téměř ve všech průmyslových odvětvích: zpracovatelský průmysl, zdravotnictví, doprava, energetika, zemědělství,

maloobchod a mnoho dalších. Tento kardinální posun nás přesvědčil, že revoluce internetu věcí už je tady, právě teď [13].

1.2 Perspektivy rozvoje

Vzhledem k rychlému tempu vývoje internet věcí bude ve světě brzy dominovat. V roce 2019 společnost Gartner předpověděla, že trh IoT ke konci roku 2020 v automobilovém průmyslu poroste na 5,8 miliardy, což je nárůst o 21 % oproti roku 2019. Připojeno bude cokoli, co lze připojit, čímž se vytvoří všezahrnující digitální systém, ve kterém všechna zařízení komunikují s lidmi i mezi sebou [8].

Tuto rychlou expanzi internetu věcí řídí několik důležitých faktorů:

- Náklady na snímač pádu
- Snížení nákladů na sběr a ukládání dat díky cloudovým řešením
- Široce se rozšiřující připojení k internetu
- Zvýšený výpočetní výkon
- Zvýšená penetrace chytrých telefonů a tabletů

Rychlý rozvoj internetu věcí nepochybně zásadně změní svět, ve kterém žijeme. Představte si, že připojené auto přistupuje k vašemu pracovnímu rozvrhu a upozorňuje kolegy, že jdete pozdě na schůzku, pokud na cestě do práce dojde k dopravní zácpě nebo k nehodě.

Aktuální nevyhnutelná vzájemně propojená budoucnost jistě přinese lidem mnoho hodnot a vzrušujících příležitostí. I zde však nastanou zcela jistě určité problémy.

V blízké budoucnosti se výrobci IoT zaměří spíše na vývoj řešení pro konkrétní průmyslová odvětví a průmyslové segmenty než na obecné potřeby. Roste poptávka po konkrétních případech použití, které pomáhají plnit průmyslové výzvy. Například řešení IoT pro vzdálené monitorování pacientů zaměřená na nákladů a zlepšování kvality péče o pacienty. Podle společnosti Grand View Research dosáhne globální trh pro vzdálené monitorování pacientů do roku 2026 1,8 miliardy USD [14].

Také vznikají nové oblasti na křižovatce propojených technologií a různých průmyslových odvětví:

- Průmyslový internet věcí
- Automobilový internet věcí
- Inteligentní města a inteligentní budovy
- Inteligentní zemědělství
- Inteligentní marketing

I když je samotný IoT výkonný, poskytuje mnohem více energie v kombinaci s dalšími technologiemi, jako je blockchain, umělá inteligence, strojové učení, velká data, AR/VR a cloud a edge computing. V budoucnu bude existovat mnohem více smíšených řešení.

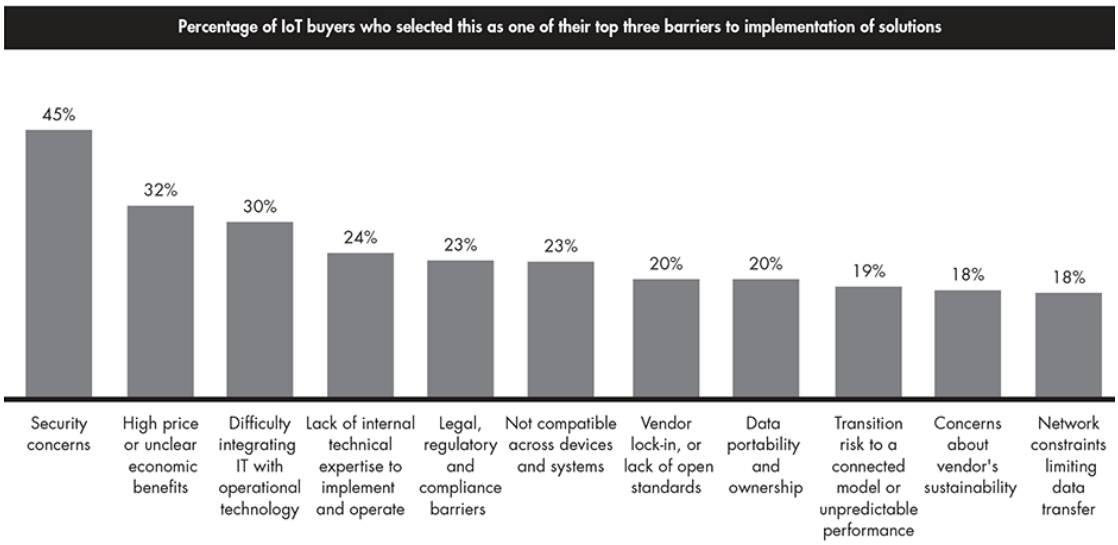
Například použití blockchainu v IoT pomůže decentralizovat sítě a zajistit bezpečnější přenos dat mezi propojenými zařízeními. Blockchain je již předním trendem IoT a kombinace těchto dvou řešení přinese hodně výhod.

Budoucnost internetu věcí také úzce souvisí s umělou inteligencí a strojovým učením. Mezi příklady aplikací patří prediktivní údržba propojených zařízení, automatická optimalizace výrobních procesů a inteligentní domácí zařízení, která se učí vaše preference. V blízké budoucnosti budou zařízení IoT nejen komunikovat, ale také budou činit autonomní rozhodnutí a stanou se chytřejšími díky zavedení metod umělé inteligence [11].

Cloud a edge computing zůstanou nedílnou součástí datového skladu IoT a odborníci předpovídají, že edge computing se brzy stane ještě populárnějším.

1.3 Zabezpečení internetu věcí

Navzdory snahám mnoha vlád o posílení bezpečnostních předpisů IoT a zlepšení ochranných mechanismů pro interoperabilitu se obavy o bezpečnost dat a soukromí nikdy nezmenší. Kyberzločinci používají stále sofistikovanější taktiku k hledání zranitelných míst v připojených zařízeních, čímž získávají přístup k soukromým informacím. Výsledkem je, že spotřebitelé a organizace se stále více zajímají o bezpečnost internetu věcí a považují to za hlavní překážku jeho širokého přijetí.



Obrázek 1: Top investic do IoT, zdroj: www.bain.com

Snad bude nalezení řešení IoT pro zabezpečení a ochranu osobních údajů v budoucnu hlavní průmyslovou prioritou.

Po nastínění minulosti, současnosti a budoucnosti internetu věcí můžeme dojít k závěru, že vzájemně propojené technologie se rychle rozvíjejí navzdory rostoucím obavám o kybernetickou bezpečnost. Je zaručeno, že se řešení IoT budou v nadcházejících letech nadále vyvíjet a na oplátku zlepší způsob, jakým žijeme a pracujeme.

1.4 Protokoly pro výměnu dat mezi IoT zařízeními

1.4.1 MQTT

MQTT (Message Queuing Telemetry Transport) je protokolem, který je vyvinut na základě návrhového vzoru publisher-subscriber. První verzi protokolu vyvinuli programátoři Andy Stanford-Clark z IBM a Arlen Nipper z Cirrus Link. MQTT je určen pro komunikaci v rámci nestabilní sítě s omezenou rychlostí. Pro použití protokolu musí existovat server (tzv. broker), který bude přijímat zprávy od zařízení, třídit zprávy a doručovat zprávy klientům. Každá zpráva se posílá do vybraného kanálu, což zaručuje, že klient dostane pouze tu zprávu, která je pro něho určena.

Vlastnosti protokolu jsou:

- Neutrálnost k obsahu zprávy – protokol nemá omezení na obsah zprávy kvůli tomu, že zprávy musí mít binární formát.
- Podporuje funkci LWT (Last Will and Testament), která je určena pro notifikace, když klient byl odpojen při anomální situaci, např. při výpadku napětí.
- Podporuje šablony doručení zpráv: doručit pouze jednou, doručit minimálně jednou a doručit maximálně jednou.
- Každý klient MQTT může posílat zprávy, přijímat zprávy nebo obojí najednou.
- Založen na transportním protokolu TCP, který zaručuje, aby zpráva byla doručena.

1.4.2 REST API

REST API je architekturním stylem komunikace mezi různými částmi systému, lze říct, že je to seznam pravidel, které musí dodržovat programátor během návrhu REST rozhraní. Komunikace v rámci REST API probíhá pomocí protokolu HTTP nebo jeho zabezpečené verze HTTPS. Formát zpráv a typ dat může byt jakýkoliv, ale nejčastěji se používá JSON nebo XML.

Vlastnosti REST API jsou:

- Komunikace typu klient-server, komunikace probíhá mezi zařízeními, bez použití třetí strany, ale je potřeba, aby zařízení měla adresu, kam posílat zprávu.
- Synchronní komunikace – po zpracování příchozí zprávy bude odeslána odpověď.
- Podpora skoro všech programovacích jazyků.
- Záruka doručení zprávy díky tomu, že HTTP je založen na transportním protokolu TCP.

1.4.3 CoAP

CoAP (Constrained Application Protocol) je protokolem přímo určeným pro IoT zařízení, která mají nízký výkon a malou paměť. Protokol má shodu s protokolem HTTP, ale používá transportní protokol UDP.

Vlastnosti jsou:

- Minimální délka zprávy 4 bajty.
- Podpora odesílání zpráv několika uživatelům najednou.
- Podpora metod pro posílání zpráv typu: GET, POST, PUT, DELETE.
- Nezaručuje přijetí zprávy adresátem.

2 PŘEHLED POUŽITÝCH TECHNOLOGIÍ

2.1 Zetta

Zetta je open source platforma postavená na Node.js pro vytváření serverů IoT, které běží na geograficky rozptýlených počítačích a v cloudu. Zetta kombinuje REST API, WebSockets a reaktivní programování – ideální pro sestavení více zařízení do datově náročných aplikací v reálném čase [11].

Zetta přemění jakékoliv zařízení na API. Servery Zetta komunikují s mikrokontroléry, jako jsou Arduino a Spark Core, a poskytují každému zařízení REST API, a to jak lokálně, tak v cloudu. Reaktivní design Zetty kombinuje reaktivní programování s API Siren, což umožňuje shromažďovat distribuované systémy zařízení, která interagují a reagují prostřednictvím API.

Budování systémů IoT je náročné. Zetta poskytuje užitečné abstrakce ke zlepšení produktivity vývojářů a také poskytuje přímý přístup k základním protokolům a konvencím [13].

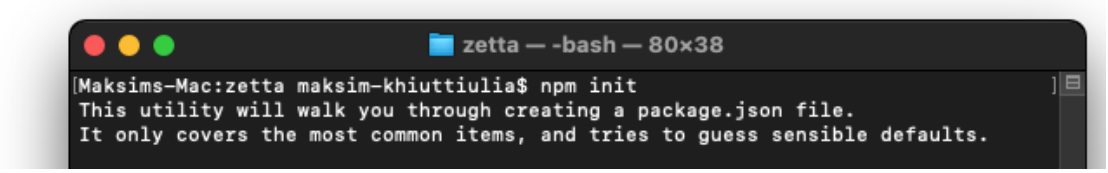
Architektura Zetty je optimalizována pro aplikace náročné na data v reálném čase. Zetta vám umožňuje sledovat a reagovat na chování zařízení a systému v kódu a pomocí nástrojů vizualizace, abyste mohli získat praktické informace a podniknout potřebné kroky. Můžete také nahrávat data na analytické platformy strojů, jako je Splunk.

2.1.1 Instalace

Pro instalaci Zetty je potřeba mít na počítači softwarový systém Node.js a správce balíčků NPM. Pomoci příkazu *npm install zetta* proběhne stahování balíčků, které jsou potřeba pro fungovaní frameworku.

Nejprve je potřeba inicializovat základní projekt v Node.js. Pro inicializaci je nutné vytvořit katalog budoucího projektu, přejít do něj a spustit příkaz *npm init,* pomocí kterého proběhne inicializace projektu.

V dalším kroku je potřeba zadat parametry projektu nebo použít výchozí hodnoty.



Obrázek 2: Inicializace projektu, zdroj: vlastní

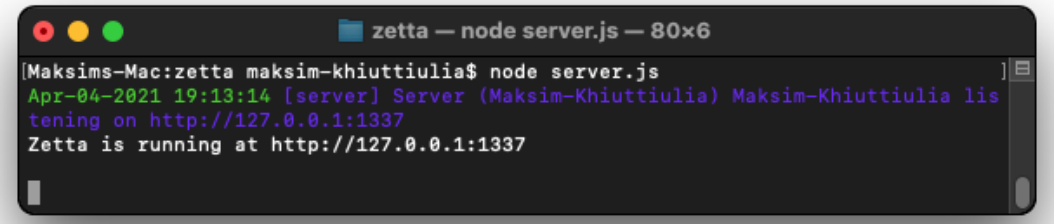
Po inicializaci projektu je nutné vytvořit soubor *server.js* a přidat následující kód:

```
proccess.EventEmitter = require('events').EventEmitter
var zetta = require('zetta')
zetta()
    .name('Maksim-Khiuttiulia')
    .listen(1337, function() {
       console.log("Zetta is running at localhost:1337")
    });
```

Zdrojový kód 1: Základní inicializace projektu Zetta, zdroj: vlastní

Funkce *name* nastavuje název serveru, funkce *listen* definuje, na jakém portu server poběží, a když server bude spuštěn, zavolá funkci, která vypíše zprávu „Zetta is running at localhost:1337“.

Příkaz *node server.js* spouští server, a když server bude spuštěn, do konzole bude vypsána zpráva, která byla zadána, viz obrázek 3.



Obrázek 3: Spuštění serveru, zdroj: vlastní

2.2 Node-RED

Node-RED je vývojový nástroj pro vizuální programování původně vyvinutý společností IBM za účelem spojení hardwarových zařízení, rozhraní API a online služeb v rámci internetu věcí [14].

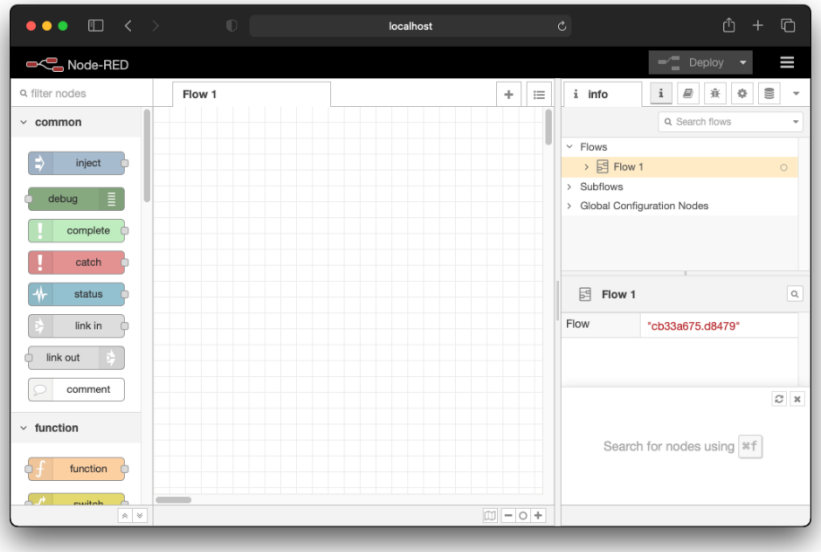
Node-RED poskytuje editor proudů založený na webovém prohlížeči, který můžete použít k vytváření funkcí v programovacím jazyce JavaScript. Položky aplikace můžete uložit nebo sdílet k opětovnému použití. Prostředí je postaveno na Node.js. Proudy vytvořené v Node-RED se ukládají s použitím JSON. Od verze 0.14 mohou uzly MQTT navázat správně nakonfigurovaná připojení TLS [3].

V roce 2016 IBM představila Node-RED jako projekt JS Foundation s otevřeným zdrojovým kódem.

2.2.1 Instalace

Pro instalaci Node-RED, stejné jako pro instalaci Zetty, je potřeba mít nainstalovaný nastroj Node.js a nástroj NPM. Pomocí příkazu *npm install -g --unsafe-perm node-red* proběhne instalace frameworku.

Pro spuštění frameworku je potřeba zadat příkaz *node-red*. V případě spuštění bez chyb v konzoli bude vypsána adresa, na které Node-RED server běží, obvykle se jedná o http://localhost:1880.



Obrázek 4: Webové rozhraní Node-RED, zdroj: vlastní

2.3 Project Flogo

Project Flogo je zdrojově efektivní ekosystém s otevřeným zdrojem založeném na jazyce Go pro vytváření aplikací řízených událostmi.

Aplikace Flogo se skládá ze spouštěčů a akcí. Spouštěče umožňují přijímat data z externích zdrojů, jsou poháněny vlastním modelem vlákna a sdílejí společné rozhraní. Akce zpracovávají události požadovaným způsobem a mají také společné rozhraní [5].

Všechny funkce ekosystému Flogo mají několik společných věcí: všechny zpracovávají události (způsobem vhodným pro konkrétní účel) a všechny implementují akční rozhraní poskytované Flogo Core.

Hlavní výhody Flogo:

- Správa událostí. Výkonný programovací model řízený událostmi a založený na spouštěčích a akcích.
- Společné jádro. Jedno společné jádro umožňuje opakované použití a flexibilitu pro všechny konstrukty událostí.
- Napsáno výhradně v jazyce Go Lang.
- Flexibilita nasazení. Nasazení jako ultralehké funkce bez serveru, kontejnery nebo statické binární soubory na periferiích zařízení IoT.
- Nativní strojové učení. Schopnost používat TensorFlow.

Flogo Core poskytuje tři hlavní rozhraní, která vývojářům umožňují vytvářet aktivity, dělat posloupnosti aktivit a poslouchat zprávy z různých zdrojů. Mezi tato další rozhraní patří:

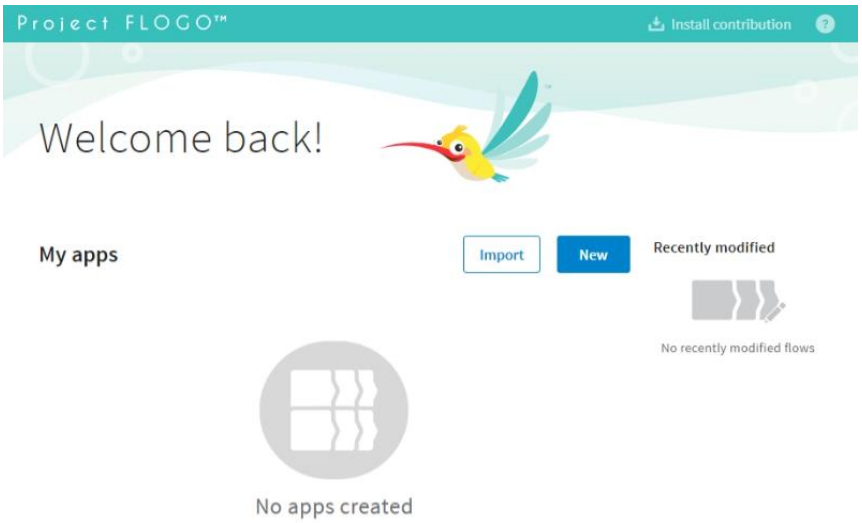
- Rozhraní spouštěče (Trigger Interface) je obecné rozhraní pro vstupní body, které reagují na událost a posílají informace o událostech jedné nebo více akcím.
- Rozhraní aktivity (Activity Interface) je společné rozhraní pro opětovné použití běžné logiky aplikace.
- Akční rozhraní (Action Interface) je rozhraní pro zpracování událostí. Akce obsahují logiku specifickou pro konkrétní logiku možností, jako je integrace, streamování, zpracování pravidel atd. Akce mají velkou flexibilitu v tom, jak jsou navrženy a jak vývojáři používají akce ve svých běžných aplikacích.

2.3.1 Instalace

Flogo má několik možností instalace, v rámci této diplomové prací bude využita instalace pomocí nástroje Docker z toho důvodu, že není potřeba instalovat nástroje pro programovací jazyk Go.

Pro použití Flogo je potřeba mít nastroj Docker, dále pomocí příkazu *docker run -it -p 3303:3303 flogo/flogo-docker eula-accept* proběhne stahování, pokud Flogo ještě nebyl stažen, a spuštění serveru na portu číslo 3303.

Po instalaci můžeme přistupovat k webovému rozhraní na adrese http://localhost:3303.



Obrázek 5: Webové rozhraní Flogo, zdroj: vlastní

2.4 Přehled zařízení v domácnosti

Každé zařízení v rámci chytré domácnosti namodelované v této diplomové práci má následující vlastnosti:

- Pro komunikaci používá REST API, formát zpráv je JSON.
- Každé zařízení má koncový bod pro získávání aktuálního stavu ve formátu: *typ_zarizeni/state*, společným stavem, který může nastat pro každé zařízení, je stav ERROR, je to chybový stav, který vyžaduje kontrolu uživatelem. Do stavu ERROR zařízení může přijít z jakéhokoliv jiného stavu.
- Každé zařízení má seznam příkazů, které je možné spouštět, ne každý příkaz je možné spustit kdykoliv, každé zařízení má seznam stavů a seznam povolených příkazů pro každý stav.

Zařízení jsou implementována v programovacím jazyce Java s použitím frameworku Spring.

2.4.1 Kávovar

Kávovar představuje jednoduchou realizaci zařízení, které má 4 stavy:

- DISABLED – kávovar je vypnutý.
- READY – kávovar je připraven k použití a je zapnutý.
- ON_WORK – kávovar je v procesu přípravy kávy.
- ERROR – vznikla chyba a je potřeba kontrola uživatelem.

Seznam koncových bodů pro kávovar:

- /coffee-machine/state (GET) – vrací aktuální stav.
- /coffee-machine/turn-on (GET) – zapnutí kávovaru, povoleno jen ve stavu DISABLED.
- /coffee-machine/turn-off (GET) – vypnutí kávovaru, povoleno jen ve stavu READY.
- /coffee-machine/prepare/{druhKavy} (GET) – příprava kávy zvoleného druhu, povoleno jen ve stavu READY.

2.4.2 Topení

Kontrolér topení umožňuje nastavovat teplotu topení a také nabízí možnost vypnout topení v domě.

Kontrolér topení má následující stavy:

- DISABLED – topení je vypnuté.
- T18C – topení je nastaveno na 18 stupňů Celsia.
- T20C – topení je nastaveno na 20 stupňů Celsia.
- T22C – topení je nastaveno na 22 stupňů Celsia.
- T24C – topení je nastaveno na 24 stupňů Celsia.
- ERROR – nastal chybový stav.

Seznam koncových bodů pro topení:

- /heating-controller/state (GET) – vrací aktuální stav.
- /heating-controller/turn-off (GET) – vypíná topení.
- /heating-controller/temperature/{teplota} (GET) – nastavuje teplotu na zadanou hodnotu. Možné hodnoty: T18C, T20C, T22C, T24C.

2.4.3 Přehrávač hudby

Přehrávač hudby má možnost zapnutí vybraného seznamu hudebních skladeb, nastavení hlasitosti nebo vypnutí. Hlasitost má interval od 0 do 100, krok změny je 10.

Přehrávač má následující stavy:

- DISABLED – přehrávač je vypnutý.
- PLAYING_RELAX_MUSIC – přehrávání relaxační hudby.
- PLAYING_ROCK – přehrávání rockové hudby.
- PLAYING_POP – přehrávání popové hudby.

- ERROR – přehrávač je v chybovém stavu.

Seznam koncových bodů pro přehrávač hudby:

- /music-player/state (GET) – vrací aktuální stav.
- /music-player/turn-off (GET) – vypíná přehrávač.
- /music-player/set-playlist/{nazevPlaylistu}(GET) – spouští přehrávání zadaného playlistu. Možné hodnoty: ROCK, POP, RELAX.
- /music-player/turn-up-volume (GET) – zvyšuje hlasitost o 10 %.
- /music-player/turn-down-volume (GET) – snižuje hlasitost o 10 %.

2.4.4 Ovladač oken

Kontrolér pro ovládání oken je určen pro otevírání nebo zavírání oken.

Kontrolér má následující stavy:

- OPEN – okna jsou otevřena.
- CLOSED – okna jsou zavřena.
- ERROR – nastala chyba.

Seznam koncových bodů pro ovladač oken:

- /windows-controller/state (GET) – vrací aktuální stav.
- /windows-controller/open (GET) – otevírá okna.
- /windows-controller/close (GET) – zavírá okna.

2.4.5 Ovladač světel

Kontrolér pro ovládání světel je určen pro zapnutí nebo vypnutí světel v domácnosti.

Kontrolér má následující stavy:

- ENABLED – světla jsou zapnutá.
- DISABLED – světla jsou vypnutá.
- ERROR – nastala chyba.

Seznam koncových bodů pro ovladač světel:

- /light-controller/state (GET) – vrací aktuální stav.
- /light-controller /turn-on (GET) – zapíná světla.
- /light-controller /turn-off (GET) – vypíná světla.

3 MODELOVÉ SCÉNÁŘE PRO DOMÁCÍ AUTOMATIZACI

3.1 Scénář 1: ovládání světel pomocí tlačítka

Docela často lidi začínají budovat chytrou domácnost od ovládání světel. Často se nejedná o žádné scénáře, například když nastane nějaký čas, je potřeba vypnout nebo zapnout světlo; chtějí jen mít možnost ovládání světel z mobilu nebo mít několik přepínačů napojených na jeden druh světla v domácnosti. Často se také jedná o lidi, kteří chtějí zkusit, co to je chytrá domácnost a jak může fungovat.

Daný scénář je tzv. „Hello World“ ze světa programování, což znamená jednoduchý program, který programátor často vyvine jako první, když začíná programovat. Tento scénář obsahuje jen jeden krok: když bude stisknuto tlačítko, proběhne zapnutí nebo vypnutí světel. Tento scénář nebude používat zařízení, která jsou popsaná v kapitole 2.4. Tento scénář je implementován pro počítač Raspberry Pi 2 a používá GPIO piny pro ovládání LED diod. Schéma zpracování scénáře krok za krokem je v příloze A.

3.1.1 Zetta

Zetta má různé možnosti napojení různých světel, například *zetta-led-pi-driver* umožňující kontrolovat GPIO piny Raspberry PI nebo *zetta-hue-driver* pro kontrolu žárovek Phillips Hue. Pro testování existuje balíček *zetta-led-mock-driver*, který umožňuje provádět kontroly scénářů bez potřeby mít fyzické zařízení. V rámci diplomové práce bude použit balíček *zetta-led-pi-driver*.

Nejprve je potřeba nainstalovat balíček *zetta-led-pi-driver* pomocí příkazu *npm install zetta-led-pi-driver --save.*

Do souboru, který byl vytvořen v rámci kapitoly 2.1.1, je nutné importovat nainstalovaný balíček a pomocí funkce *use* balíček použít. V daném příkladu je žárovka napojena na GPIO port Raspberry PI číslo 18.

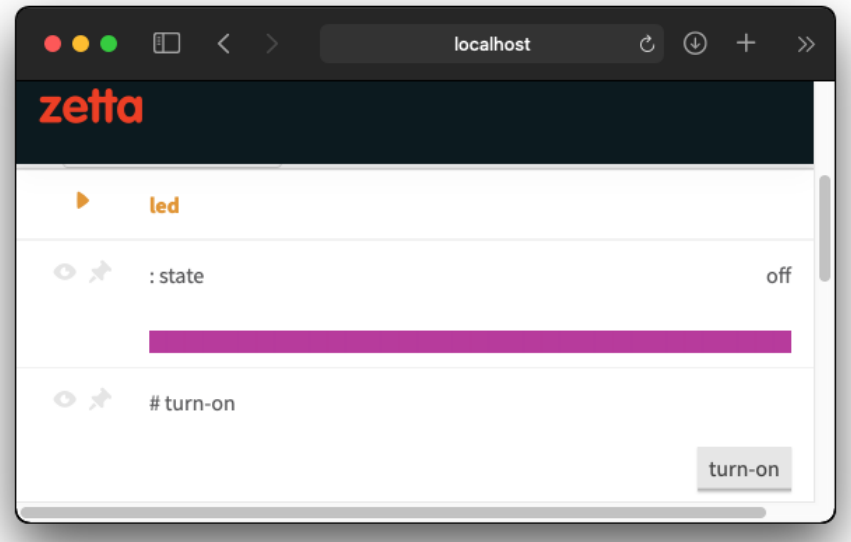
```
proccess.EventEmitter = require('events').EventEmitter
var zetta = require('zetta') // import balíčku zetta
var led = require('zetta-led-pi-driver'); // import balíčku
zetta()
    .name('Maksim-Khiuttiulia')
    .use(led, 18) // napojení modulu na port 18
    .listen(1337, function() {
       console.log("Zetta is running at localhost:1337")
    });
```

Zdrojový kód 2: Ovládání GPIO pinu v Zettě, zdroj: vlastní

Po provedení změn je potřeba uložit soubor a spustit server pomocí příkazu *node server.js*.

Obrázek 6: Start serveru, zdroj: vlastní

Když server bude spuštěn, ve webovém rozhraní vznikne nová položka a tlačítko pro vypnutí a zapnutí žárovky.



Obrázek 7: Zapnutí světelného zdroje, zdroj: vlastní

3.1.2 Node-RED

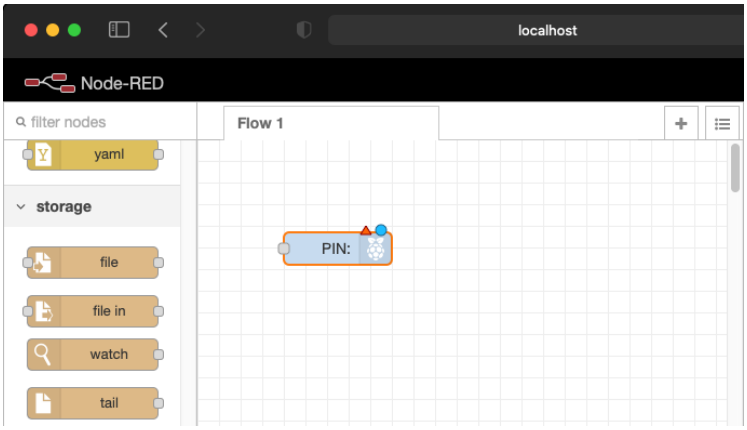
Node-RED pro komunikaci mezi zařízeními nebo pomocí webového rozhraní používá vstupní a výstupní uzly.

Vstupní uzly umožňují zadávat data do aplikace Node-RED. Vstupní uzly se používají k připojení dat z jiných služeb nebo k ručnímu zadávání dat do proudu pomocí vstupního uzlu. Existují možnosti přijetí vstupních signálů pomoci fronty zpráv, která je implementací protokolu MQTT, pomocí rozhraní REST nebo pomocí GPIO pinu v případě použití Raspberry PI nebo podobných zařízení.

Výstupní uzly umožňují odesílat data mimo proud Node-RED. Mají jediný vstupní koncový bod na levé straně. Výstupní uzly se používají k odesílání dat do jiných služeb, například přes TCP, sériových uzlů nebo e-mailových uzlů, nebo k použití ladícího uzlu k výstupu do ladícího panelu [2].

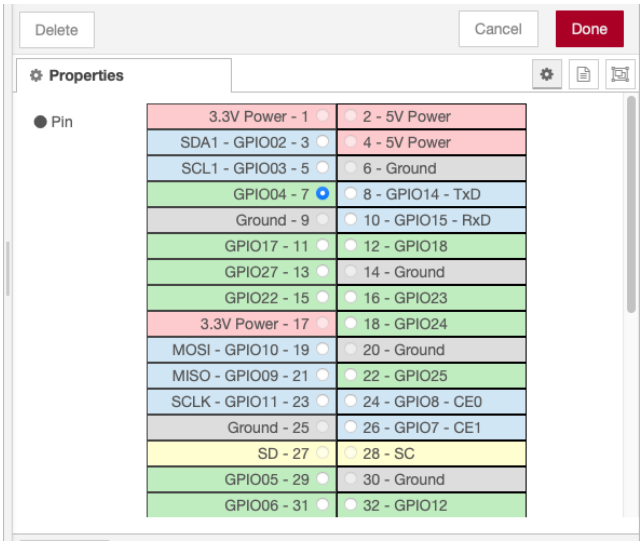
Jako výstupní bod bude použit jeden z kontaktů GPIO. Pro použití GPIO je potřeba použít uzel *rpi-gpio*, protože v dané implementaci bude použita kontrola žárovky, která je napojena na GPIO pin Raspberry PI. Také existuje možnost jako výstupní uzel použít odesílání zprávy do fronty zpráv, například MQTT, nebo volání rozhraní REST API.

Nejprve je potřeba přetáhnout uzel *rpi-gpio out* do pracovní oblasti. Po přetažení uzel bude označen červeným trojúhelníkem, který znamená, že vlastnosti uzlu nebyly nastaveny, a modrým kruhem, který znamená, že uzel nebyl nasazen na server.



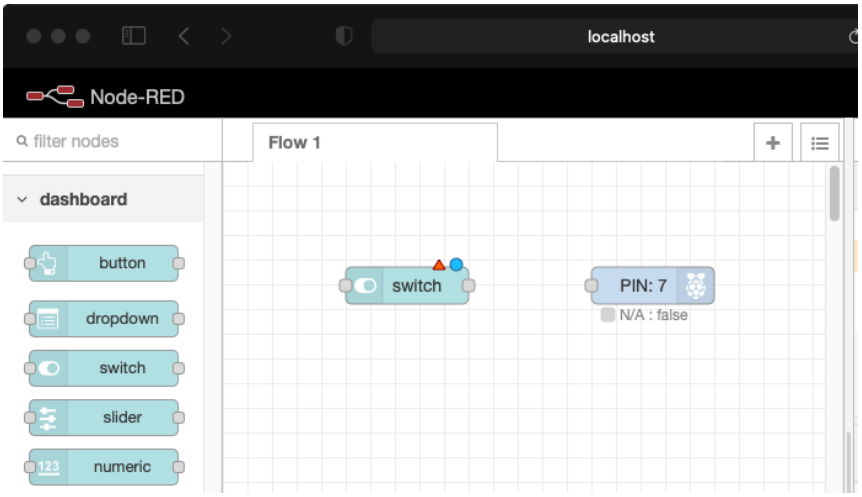
Obrázek 8: Výstupní uzel, zdroj: vlastní

Po kliknutí na uzel bude otevřeno okno nastavení vlastností, ve kterém je potřeba nastavit GPIO port, na který je napojena žárovka. V daném příkladu byl vybrán port číslo 7.



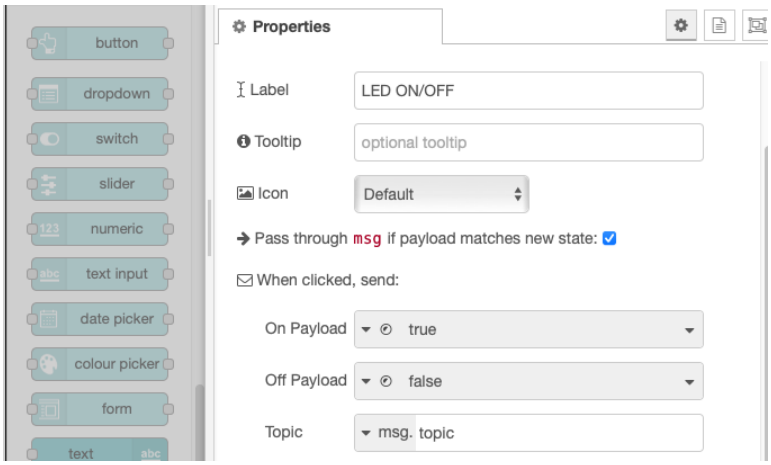
Obrázek 9: Výběr kontaktů, zdroj: vlastní

Jako vstupní uzel je možné využít zprávu z nějakého rozhraní, například fronty zpráv, aktivaci GPIO pinu nebo tlačítka ve webovém rozhraní. Pro použití webového rozhraní je potřeba přidat uzel *switch* do pracovní oblasti. Switch funguje jako přepínač a má dva stavy: vypnuto a zapnuto.



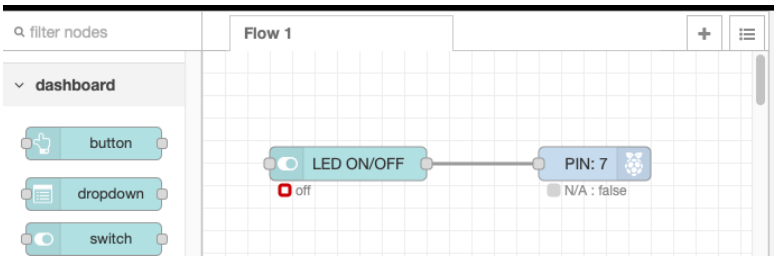
Obrázek 10: Přidání uzlu, zdroj: vlastní

Dalším krokem je změna vlastností uzlu. Pomocí vlastnosti *Label* lze nastavit název uzlu ve webovém rozhraní, v daném případě Label byl nastaven na LED ON/OFF. Dalšími vlastnostmi jsou *On Payload* a *Off Payload*. On Payload nastavuje hodnotu, která bude použita, když přepínač byl zapnut, naopak Off Payload nastaví hodnotu pro stav, kdy je přepínač vypnut. V daném případě On Payload má hodnotu *true* a Off Payload má hodnotu *false*.



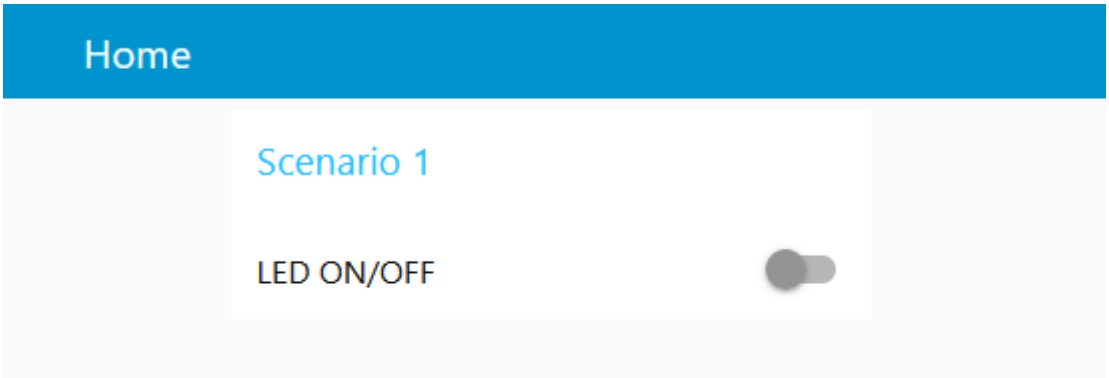
Obrázek 11: Úpravy vlastností uzlu, zdroj: vlastní

Posledním krokem je propojení dvou uzlů mezi sebou. To je minimální sada prvků pro realizaci daného scénáře.



Obrázek 12: Výsledný proud, zdroj: vlastní

Na adrese http://localhost:1880/ui bude dostupné webové rozhraní pro kontrolu žárovky.

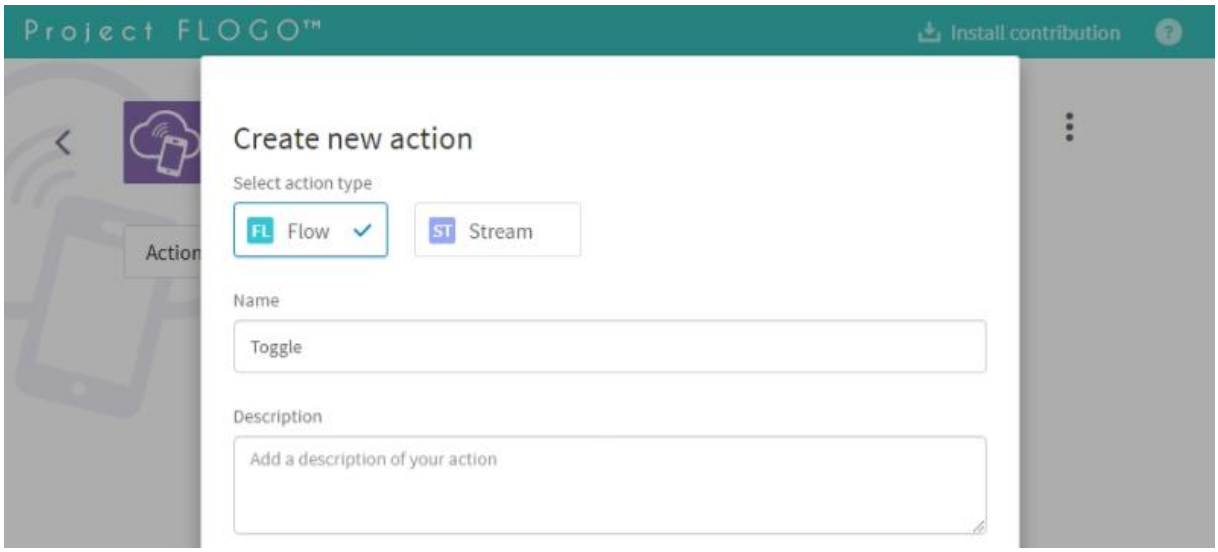


Obrázek 13: Výsledek ve webovém rozhraní, zdroj: vlastní

3.1.3 Project Flogo

Aplikace poskytuje koncový bod REST, na který přijímá zprávy, místo REST lze použít MQTT nebo časovač. Pro založení nového projektu je nutné kliknout na tlačítko New, po vytvoření projektu je možné změnit název projektu a popis.

Dalším krokem je založení nové aktivity. Aktivita je algoritmus, podle kterého aplikace bude pracovat se vstupními a výstupními daty. Dále je potřeba zadat název aktivity, popřípadě popis, a vybrat typ Flow.



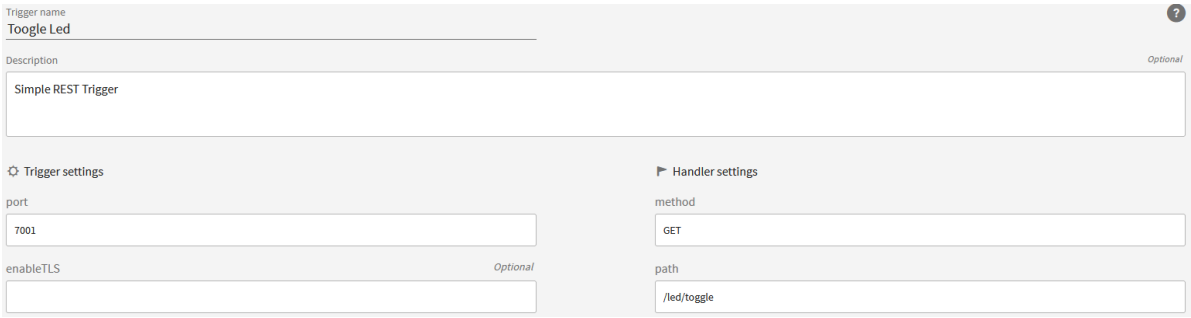
Obrázek 14: Název proudu, zdroj: vlastní

Po vytvoření nové akce bude zobrazena pracovní oblast pro práci s akcí, kde můžeme zadat vstupní body, postup, jak bude spravována vstupní událost a popřípadě výstupní data.

Trigger (spouštěč) má následující parametry:

- Trigger name – jméno triggeru v rámci flow, musí byt unikátní.
- Description – popis triggeru, nepovinný parametr.
- Port – číslo portu, na kterém trigger bude poslouchat příchozí zprávy, povinný parametr.
- Method – HTTP metoda, pomocí které trigger bude přijímat zprávy, povinný parametr.
- Path – cesta, na kterou trigger bude přijímat zprávy, povinný parametr.
- EnableTLS – parametr pro zapnutí podpory protokolu HTTPS, nepovinný parametr.
- CertFile – cesta pro soubor s certifikátem pro HTTPS protokol.
- KeyFile – cesta pro soubor s klíčem pro HTTPS protokol.

Pro daný scénář byl vytvořen spouštěč s názvem Toogle Led, který bude přijímat zprávy na portu 7001, které byly poslány metodou GET, cesta pro trigger je */led/toggle*.



Obrázek 15: Nastavení koncového bodu, zdroj: vlastní

Spouštěč sám o sobě nedělá nic jiného, než poslouchá příchozí zprávy nebo požadavky a posílá data do akce. Pro fungování aplikace je potřeba přidat aktivitu, která bude používat data, která spouštěč přijal.

Pro ovládání GPIO pinu existuje modul *GPIO Control*, který byl součástí Project Flogo do verze 0.5.8, v aktuální verzi 1.0.0 tento modul není a neexistuje možnost instalace modulu z důvodu nekompatibilnosti. Pro řešení tohoto problému byl vyvinut modul *GPIO Output*, jehož zdrojový kód je uložen na webových stránkách https://github.com/maksim-khiuttiulia/contrib/tree/master/activity/gpio-output. Nový modul je určen pro ovládání GPIO pinu v režimu výstupu. Popis a postup pro vytvoření modulu je popsán v kapitole 4.1.

Pro instalaci modulu do frameworku existují dvě možnosti: instalace pomocí příkazového řádku příkazem *flogo install url_modulu* anebo přes grafické rozhraní, do kterého je potřeba zadat URL adresu. Adresa modulu GPIO Output pro instalaci přes grafické rozhraní je: https://github.com/maksim-khiuttiulia/contrib/activity/gpio-output.

V rámci scénáře byl použít pin č. 4 a jako parametr *action* byla použita varianta *Toggle*.

Obrázek 16: Nastavení triggeru pro scénář 1, Flogo, zdroj: vlastní

Výsledkem bude aplikace, která bude na každý požadavek měnit stav žárovky ze stavu zapnuto na stav vypnuto a naopak. Tato aplikace obsahuje minimální počet prvků pro ovládání

žárovky, ale lze přidat i takové funkce, jako je přidání záznamu do databáze nebo logování se do souboru.

Obrázek 17: Vyladěný projekt, zdroj: vlastní

3.2 Scénář 2: návrat domů

Prvním častým scénářem v rámci chytré domácnosti je provádění nějaké posloupnosti akcí, když proběhne určitá aktivita. Aktivitou může být stisk tlačítka, aktivace nějakého senzoru. Uživatelé v rámci chytré domácnosti mají možnost nastavit akce, které budou prováděny například po návratu domů po práci. Pro takový případ vstupní aktivitou může být RFID čip, pomocí kterého mohou být otevřeny dveře.

Tento scénář je určen pro situace, kdy se uživatel vrací domů a chce mít zapnuto to, co je potřeba. Scénář bude navržen pro dva uživatele a každý z nich bude mít vlastní nastavení. Jako vstupní bod pro rozlišení, kdo přišel domů, se předpokládá, že u vstupu do bytu je RFID nebo NFC modul, který je zapojen do lokální sítě a přes HTTP protokol posílá zprávy s informací, kdo přišel.

Uživatel 1 má uživatelské jméno user1. Když se vrací domů, proběhne uzavření oken, zapnutí topení na 22 stupňů Celsia, zapnutí kávovaru a příprava kávy druhu cappuccino. Pro druhého uživatele je použito uživatelské jméno user2 a proběhne otevření oken, zapnutí kávovaru a příprava kávy typu espresso a zapnutí přehrávače pro přehrávání playlistu ROCK. Diagram scénáře je v příloze B.

Tento scénář bude používat následující zařízení:

- Kávovar
- Topení
- Přehrávač hudby
- Ovladač oken

3.2.1 Zetta

Nejprve je potřeba vytvořit nový projekt viz kapitola 2.1.1, dalším krokem je implementace ovladačů pro každé zařízení z toho důvodu, že v chytré domácnosti, která byla implementována pro danou diplomovou práci, neexistují žádná zařízení, pro která existují hotové moduly. Pro každé zařízení bude realizován vlastní ovladač, protože každé zařízení má vlastní funkce. Každý ovladač musí být potomkem třídy *Device* pro to, aby framework mohl sledovat stav zařízení a ovládat ho pomocí interních funkcí.

Zetta je realizována ve verzi jazyka JavaScript ES5, je potřeba používat funkci *inherits*, která je součástí balíčku *util* pro dědění. Každý ovladač bude komunikovat se zařízením pomocí REST API, pro práci s daným protokolem lze využít funkci *fetch*, ale v rámci diplomové práce byla použita knihovna *axios*.

Každý ovladač v Zettě má parametry jako druh ovladače (type), aktuální stav (state) a jméno (name). Konfigurace ovladače probíhá ve funkci *init*, kde je možné nastavit povinné hodnoty, definovat, které příkazy ovladače lze použít v případě každého stavu zařízení, a udělat mapování příkazů na konkrétní funkci, která bude zavolána pro zpracování příkazu.

Ovladač pro kávovar má následující stavy, které jsou pro snadné použití definovány pomocí číselníku:

- DISABLED – kávovar je vypnutý.
- READY – kávovar je připraven k použití.
- ON_WORK – kávovar připravuje kávu.
- ERROR – nastala chyba.

Pro ovládání kávovaru jsou příkazy:

- espresso – příprava kávy typu espresso, je možné volat ve stavu READY.
- cappuccino – příprava kávy typu cappuccino, je možné volat ve stavu READY.
- turnOn – zapnutí kávovaru, je možné volat ve stavu DISABLED.
- turnOff – vypnutí kávovaru, je možné volat ve stavu READY.

Ve zdrojovém kódu 3 probíhá nastavení typu ovladače na hodnotu *coffee-machine* (kávovar), výchozí stav je nastaven na DISABLED, je přiděleno jméno zařízení, které bude nastaveno pro konkrétní zařízení. Pomoci funkce *when* probíhá definice možností volání příkazů na základě aktuálního stavu. Ve funkci *map* je definice, jaká funkce bude vyvolána pro jaký příkaz. Po definici konfigurace probíhá volání REST API pro zjištění aktuálního stavu zařízení. Stejným způsobem byly implementovány ostatní ovladače pro daný scénář.

```
CoffeMachine.prototype.init = function (config) {
  config
    .type('coffee-machine') // Nastavení typu ovladače
    .state(State.ERROR) // Nastavení výchozího stavu
    .name(this.assignedName); // Nastavení jména zařízení
  config
    .when(State.READY, { allow: [Command.espresso,
Command.cappuccino, Command.turnOff] }) // Definice možných příkazů
ve stavu READY
...
    .map(Command.turnOn, this.turnOn)
  const self = this
  axios.get(COFFEE_MACHINE_ADDRESS + "/state").then(resp => {
    self.state = resp.data
  }).catch(error => {
    self.state = State.ERROR;
  })
}
```

Zdrojový kód 3: Ovladač kávovaru v Zettě, zdroj: vlastní

Dalším krokem je definice komponenty pro nalezení zařízení. V Zettě pro to je určena třída *Scout*, prostřednictvím které lze definovat funkce, pomocí kterých proběhne inicializace zařízení a případně periodická kontrola stavu. Stejné jako v předchozím kroku je potřeba vytvořit třídu, která bude potomkem třídy *Scout*, a definovat funkci *init*, která je určena pro definice ovladačů a pomocných rozhraní (GPIO, Bluetooth atd.) a jako vstupní parametr má funkci *next*, kterou je potřeba zavolat po inicializaci všech zařízení. Pro nalezení zařízení v síti je funkce *discover*, která prvním argumentem přijímá třídu, která je potomkem třídy *Device*, další parametry jsou určeny pro konstruktor třídy, která je prvním parametrem. Ve zdrojovém kódu 4 je uveden příklad inicializace kávovaru, kde prvním parametrem je třída *CoffeMachine*, která byla vytvořena v minulém kroku, a jméno zařízení.

```
ZettaScout.prototype.init = function(next) {
  const self = this;
  setTimeout(function() {
    self.discover(CoffeMachine, 'Coffee machine'),
...
  }, 1000)
  next();
}
```

Zdrojový kód 4: Třída Scout pro Zettu, zdroj: vlastní

Předposledním krokem je implementace aplikačního modulu, který je určen pro samotnou implementaci scénáře a obsahuje logiku, podle které bude scénář fungovat. Nejprve je potřeba získat seznam ovladačů, které byly inicializovány v modulu *Scout* a uloženy do registru Zetty. Pro tyto účely v objektu *server* je funkce *where*, která jako parametr přijímá seznam parametrů ve formátu klíč-hodnota, podle kterých je potřeba najít ovladač. Tato funkce vrací ovladač, se kterým je pak možné pracovat. Ve zdrojovém kódu 5 je uveden příklad pro získání ovladače podle typu (type) a názvu (name).

```
const coffeeMachine = server.where({type : 'coffee-machine', name :
"Coffe machine"});
```

Zdrojový kód 5: Definice zařízení v rámci scénáře Zetty, zdroj: vlastní

Pro to, aby aplikační modul mohl pracovat s ovladači, je potřeba zavolat funkci *observe*, která má 2 vstupní parametry: pole ovladačů, ve kterém je potřeba pracovat, a funkci, ve které bude popis logiky scénáře. Tato funkce musí mít jako vstupní parametry seznam ovladačů ve stejném pořadí jako pole ovladačů, které je prvním parametrem funkce *observe*.

```
server.observe([coffeeMachine, heatingController, musicPlayer,
windowsController, rfidSensor],
         function(coffeeMachine, heatingController, musicPlayer,
windowsController, rfidSensor) {
```

Zdrojový kód 6: Přidání zařízení do aplikace Zetta, zdroj: vlastní

Pro realizaci akce způsobené nějakým ovladačem, například stiskem tlačítka, které jsou definovány pro každý ovladač ve funkci *map*, je v ovladači funkce *on*, kde je prvním parametrem název příkazu, který byl definován pro ovladač, a druhým parametrem je funkce, která bude zavolána, když proběhne akce na ovladači. Pro použití určitého příkazu, který byl definován pro ovladač, je u ovladače funkce *call*. Prvním parametrem funkce je název příkazu, který je potřeba zavolat v ovladači, druhým parametrem je pole parametrů, které budou použity jako vstupní parametry pro funkci, která je mapována na příkaz, třetím parametrem je tzv. *callback* funkce, která bude vyvolána, když příkaz proběhne. Druhý a třetí parametr jsou nepovinné.

Ve zdrojovém kódu 7 vidíme, že když bude spuštěn příkaz *USERCAME*, který znamená, že uživatel přišel domů, z ovladače *username* bude získáno uživatelské jméno a na základě hodnoty proběhne spuštění části scénáře pro konkrétního uživatele.

```
rfidSensor.on('USERCAME', function() {
      console.error(rfidSensor.username)
      if (rfidSensor.username == "user1") {
         user1(windowsController, heatingController, coffeeMachine)
      } else if (rfidSensor.username == "user2") {
                  user2(windowsController, heatingController,
coffeeMachine, musicPlayer)
      }
            });
```

Zdrojový kód 7: Rozhodování podle uživatelského jména v Zettě, zdroj: vlastní

Posledním krokem je import modulu, v souboru *index.js* je pomocí funkce *use* potřeba importovat moduly do serveru Zetty, viz zdrojový kód 8.

```
zetta()
  .name('SmartHome')
  .use(ZettaScout)
  .use(ZettaApp)
```

Zdrojový kód 8: Inicializace serveru Zetty pro scénář 2, zdroj: vlastní

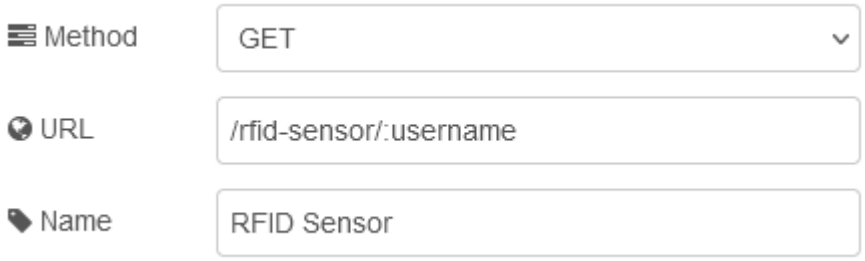
3.2.2 Node-RED

Jako vstupní bod do scénáře byl použit modul *http in*, který slouží pro přijetí zpráv pomocí protokolu HTTP. Každý modul *http in* musí být navázán na alespoň jeden modul *http out*, aby odesílatel zprávy dostal odpověď.

Konfigurace *http in* má následující parametry:

- Method – metoda, pomocí které bude přijata zpráva, na výběr jsou možnosti: GET, POST, PUT, DELETE, PATCH.
- URL – cesta pro koncový bod, do cesty je možné také přidat parametry pomocí dvojtečky, např. *:username*.
- Name – název modulu v rámci aplikace.

Konfigurace modulu pro RFID senzor je na obrázku 18. V cestě je ukázán parametr *username* pro další rozhodování, jaký uživatel použil RFID čip a jaké operace je potřeba provést.

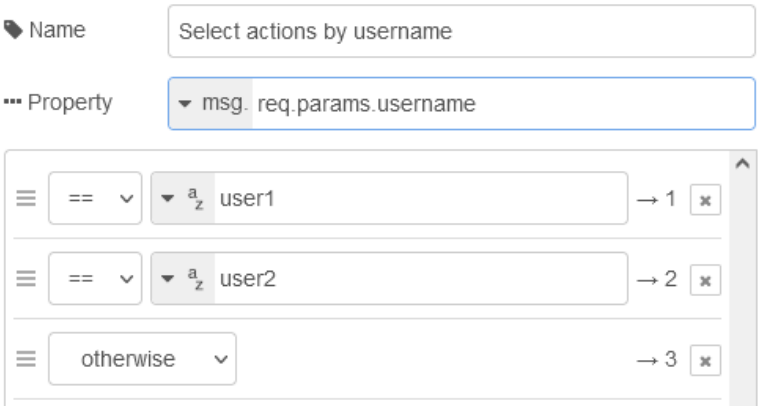


Obrázek 18: Konfigurace http in modulu, zdroj: vlastní

Dalším krokem je rozhodování, kdo přišel, na základě uživatelského jména, k čemuž slouží modul *switch*. Modul *switch* dostává na vstup hodnotu a tu porovnává s vybranou hodnotou a podmínkou. Modul má následující parametry:

- Name – název modulu v rámci aplikace.
- Property – vstupní hodnota, na jejímž základě proběhne rozhodování. Může být použita hodnota, která je výstupem z předchozího uzlu (msg), hodnota, která je nastavena pro celý scénář (flow), globální hodnota (global), výraz (expression) nebo proměnná prostředí (env. variable).
- Seznam podmínek – podmínky, které musí splňovat hodnota, aby proběhlo další zpracování scénáře. Podmínky mohou být typu rovná se, porovnání pomocí regulárního výrazu, hodnota je prázdná atd.
- Parametr zpracování podmínek – může být nastaven na hodnotu *stopping after first match*, což znamená, že rozhodování bude ukončeno po nalezení první splněné podmínky, anebo *checking all rules* pro případ, když zpracování může jít víc než jednou větví.

Konfigurace rozhodování na základě uživatelského jména je definována na obrázku 19. Podmínka *otherwise* (jinak) je určena pro případ, kdy bude posláno uživatelské jméno odlišné od user1 nebo user2, a pomoci modulu *http out* bude vrácena zpráva se statusem 404 – nebyl nalezen záznam. Ostatní větve budou napojeny na moduly pro zpracování scénáře a každá větev bude také napojena na modul *http out*, který bude vracet zprávu odesílateli se statusem 200 (úspěch).



Obrázek 19: Konfigurace modulu switch pro uživatelské jméno, zdroj: vlastní

Pro ovládání zařízení je potřeba implementovat jednotlivé kroky, to lze udělat v rámci scénáře nebo v podobě tzv. subflow. Subflow je část scénáře, která je v samostatném modulu pro opakované použití v jiných scénářích. Každý subflow může mít vstup, na který je možné předávat seznam parametrů, a neomezený počet výstupů pro návratové hodnoty.

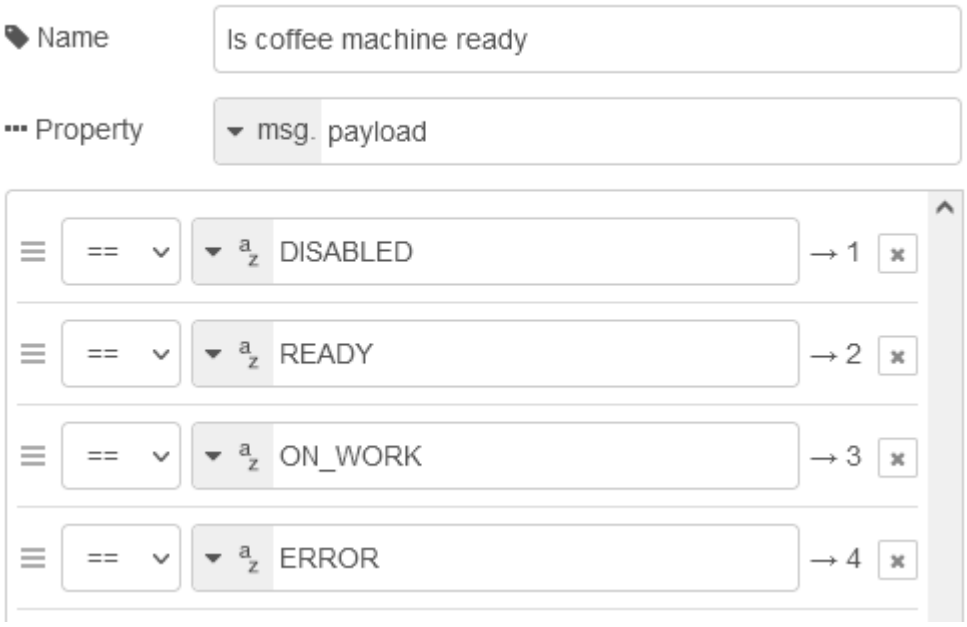
Pro ovládání kávovaru byl vytvořen subflow, který jako vstupní parametr přijímá druh kávy, kterou je potřeba připravit, a URL adresu kávovaru. Prvním krokem je zjištění aktuálního stavu kávovaru. Pro komunikaci pomocí REST API v Node-RED je modul *http request*.

Modul *http request* je určen pro komunikaci pomocí protokolu HTTP. Má následující možnosti konfigurace:

- Method – metoda, pomocí které bude probíhat komunikace, lze vybrat z GET, POST, UPDATE, DELETE a HEAD. Dále je možné předat typ metody ve vstupní zprávě v pole *msg.method*.
- URL – adresa, na kterou bude poslán požadavek.
- Payload – zpráva, která bude poslána. Zpráva může být poslána jako parametr v adrese (query string) nebo v těle zprávy (request body), dále je také možné zprávu neposílat (ignore).
- Enable secure (SSL/TLS) connection – zapnutí komunikace pomocí protokolu HTTPS, který je bezpečnější než HTTP.
- Use authentication – zapnutí autentifikace pro případy, kdy to vzdálený server vyžaduje.
- Enable connection keep-alive – zapnutí režimu, kdy připojení k serveru nebude zrušeno po přijetí odpovědi, primárně určeno pro zrychlení komunikace.
- Use proxy – umožňuje posílání požadavků přes zvolené proxy pro případy, kdy server, na kterém běží Node-RED, nemá přímý přístup na zvolenou adresu.

Pro posílání požadavku o stavu je potřeba poslat GET požadavek na adresu */coffee-machine/state*. Tento koncový bod vrací aktuální status. Po získání odpovědi následuje rozhodování na základě aktuálního stavu kávovaru, pro to byl použit modul *switch*, který určuje, jakým způsobem proběhne zpracování scénáře, viz obrázek 20. Rozhodování má čtyři větve:

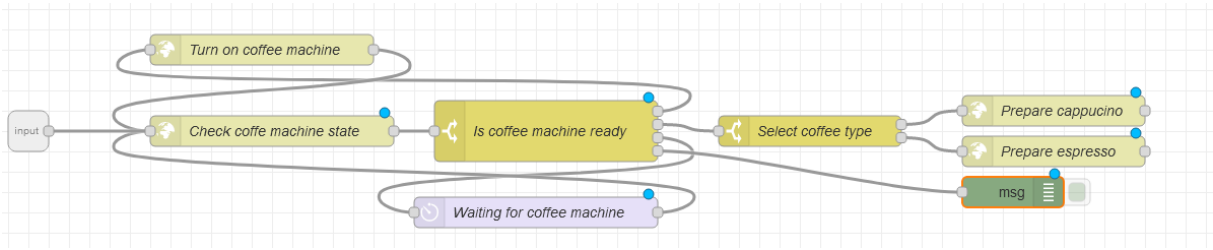
- Stav je DISABLED – proběhne volání koncového bodu pro zapnutí kávovaru, poté znovu proběhne kontrola stavu.
- Stav je READY – proběhne rozhodování, jaký druh kávy je potřeba připravit, a bude zavolán jeden ze dvou koncových bodů pro přípravu kávy.
- Stav je ON_WORK – proběhne čekání 5 sekund pomocí modulu *delay* a pak opakovaná kontrola stavu.
- Stav je ERROR – proběhne výpis do konzole pomocí modulu *debug*, že kávovar hlásí chybu.



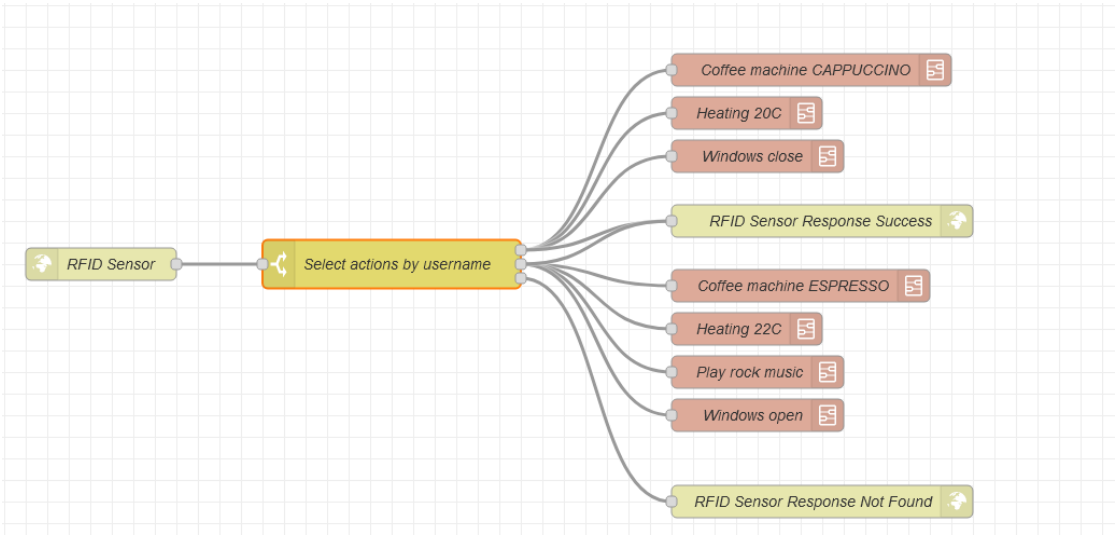
Obrázek 20: Konfigurace modulu switch pro aktuální stav kávovaru, zdroj: vlastní

Modul *debug* je určen pro výpis do konzole různých dat, které jsou potřeba pro vývoj nebo sledování stavu. Modul umí vypsat celou zprávu, kterou dostal na vstup, nebo její určitou část v JSON formátu.

Finální subflow pro přípravu kávy je na obrázku 21. Stejným způsobem byly vytvořeny subflow pro ostatní zařízení.



Obrázek 21: Subflow pro ovládání kávovaru, zdroj: vlastní



Obrázek 22: Výsledná struktura scénáře 2, zdroj: vlastní

3.2.3 Project Flogo

Stejně jako u implementace scénáře pro Node-RED i Flogo podporuje tzv. subflow, ale na rozdíl od Node-RED ve Flogo každý flow může být použít jako subflow a může běžet samostatně. Pro každé zařízení bude vytvořen samostatný flow, ve kterém bude zpracována veškerá logika pro ovládání zařízení, a flow bude mít jako vstupní parametr jen to, co je potřeba, v případě kávovaru to bude název kávy, kterou je potřeba připravit.

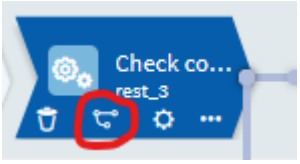
Prvním krokem je vytvoření nového proudu kliknutím tlačítka New Action. Je potřeba zadat unikátní název proudu v rámci aplikace a popřípadě popis proudu. Pro kávovar byl zvolen název Coffee Machine (angl. kávovar). Po vytvoření proudu je potřeba definovat vstupní parametr. Pro definice vstupních a výstupních parametrů je potřeba v okně pro editaci proudu kliknout na tlačítko Input/Output (vstup/výstup). Každý proud může mít vstupní parametry, které bude dostávat na vstup z triggeru nebo z jiného proudu, ve kterém byl volán, a také výstupní parametry, které je možné použít pro další zpracování. Proud pro kávovar bude mít jen jeden vstupní parametr: druh kávy, který bude datovým typem *string* (řetězec).

První věcí, kterou je potřeba provést, je kontrola aktuálního stavu kávovaru. Pro volání koncového bodu zařízení je potřeba přidat akci typu *REST Invoke,* která je určena k volání REST API. Akce má následující parametry:

- Settings (nastavení):
  - Headers – parametr definuje hlavičky pro požadavek, seznam hlaviček se definuje v JSON formátu, nepovinný parametr.
  - Method – HTTP metoda, definuje metodu, kterou bude odeslána zpráva, povinný parametr.
  - Proxy – definice adresy pro proxy server, pro případ, kdy server, na kterém běží Flogo, nemá přímý přístup na zvolenou adresu, nepovinný parametr.
  - SSLConfig – definice SSL parametru pro protokol HTTPS.
  - Timeout – definuje, jak dlouho akce bude čekat na odpověď, než ukončí spojení, nepovinný parametr.
  - URI – adresa, na kterou bude poslán požadavek, povinný parametr, podpora parametru v cestě ve formátu *:parametr*.
- Map Inputs (mapování vstupu):
  - Content – obsah požadavku, který bude odeslán ve formátu JSON.
  - Headers – definuje hlavičky požadavku, ale na rozdíl od hlaviček v sekci Settings je možné používat hodnoty, které akce dostala jako vstupní.
  - Path params – definuje parametry cesty, které byly definovány přes dvojtečku v parametru URI, definice parametru musí mít JSON formát.
  - Query params – definuje parametry, které budou přidaný na konec cesty za otazníkem, např. */cesta?param=1*. V tomto případě je název parametru param, hodnota 1. Definice parametru musí mít JSON formát.
- Iterator:
  - Iterate – definuje, kolikrát akce bude volána.

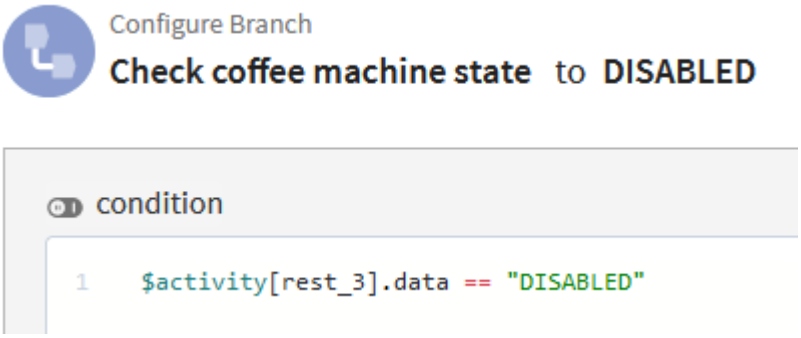
Pro získání stavu kávovaru byla vytvořena akce, která volá koncový bod */coffee-machine/state* pomocí metody GET.

Po získání odpovědi je potřeba provést rozhodování, jakým směrem půjde zpracování aktivity. Pro rozhodování u každé akce je jako výstup možné přidat větvení. Pro přidání větvení je potřeba kliknout na tlačítko u vybrané akce viz obrázek 23.



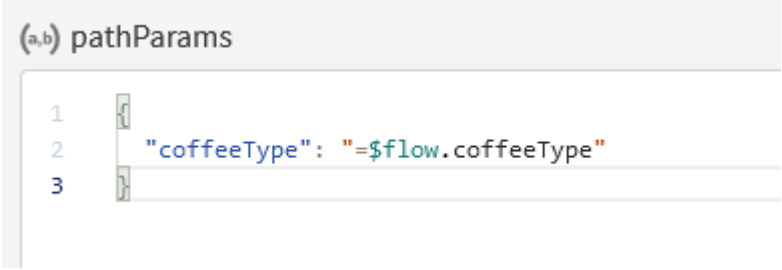
Obrázek 23: Přidání větvení do Flogo, zdroj: vlastní

Každá větev má vlastní podmínku, která definuje, zda zpracování tímto směrem půjde, či nikoliv. Větev jako vstupní hodnotu může používat hodnotu, která je výstupem jakékoliv akce, která byla před tím v rámci proudu, nebo vstupní parametr proudu. Na obrázku 24 je uvedena konfigurace větve pro případ, kdy je kávovar ve vypnutém stavu. Ve větvi probíhá porovnání odpovědi z akce pro zjištění stavu kávovaru se stavem DISABLED (vypnuto).



Obrázek 24: Nastavení větve pro vypnutý stav kávovaru, zdroj: vlastní

Pro případ, kdy je kávovar vypnutý, je potřeba provést volání koncového bodu pro zapnutí kávovaru, a začít zpracování akce od začátku. Framework nepodporuje cykly v rámci akcí, pro řešení tohoto problému byl vytvořen trigger typu *Receive HTTP Message*, který přijímá požadavek na portu číslo 7000 a cestou */coffee-machine/:coffeeType*, kde *coffeeType* je parametr, který definuje druh kávy. Pro volání koncového bodu z flow byla použita akce typu *REST Invoke*, úplná cesta do triggeru je *http://localhost:7000/coffee-machine/:coffeeType*, metoda je typu GET a do Path params byl přidán parametr, viz obrázek 25.



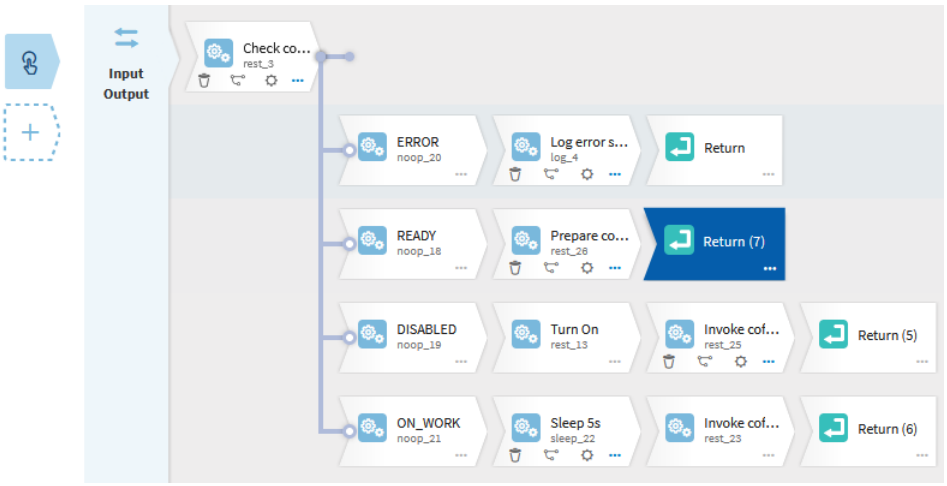
Obrázek 25: Nastavení Path params v akci REST Invoke, zdroj: vlastní

Pro případ, kdy kávovar vrací chybový stav (ERROR), byla použita aktivita typu *Log*, která je určena pro zápis do log souboru. Akce má následující parametry:

- Add Details – přidá do záznamu informace o aktuálním kontextu, vstupních parametrech atd.
- Message – zpráva, která bude vypsána do logu.
- Use Print – místo zápisu do log souboru bude použit výpis do konzole.

Pro případ, kdy kávovar již připravuje kávu (ON_WORK), je potřeba počkat 5 vteřin a opakovat celý flow od začátku. Akce pro čekání není součástí Project Flogo, a proto byla vyvinuta vlastní akce *Sleep*, která je kompatibilní s verzí Framework 1.0.0. Akci je možné nainstalovat z adresy https://github.com/maksim-khiuttiulia/contrib/activity/gpio-output. Postup vývoje akce je popsán v kapitole 4.2. Akce má pouze jeden parametr *SleepTime*, který definuje, kolik sekund je potřeba počkat. Po vykonání akce *Sleep* se volá koncový bod triggeru, který byl vytvořen.

Na obrázku 26 je uvedena struktura flow pro kávovar, stejným způsobem byly vytvořeny flow pro ostatní zařízení pro daný scénář.



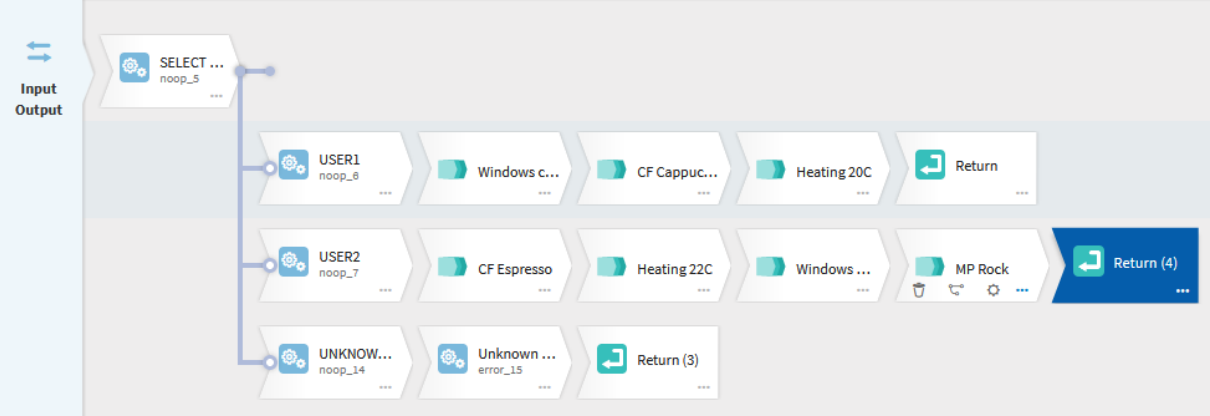
Obrázek 26: Flow pro kávovar, zdroj: vlastní

Hlavní flow v rámci scénáře má název Scenario 2 a vstupním bodem do něj je trigger typu *Receive HTTP Message*, který poslouchá port číslo 7001 a přijímá zprávy, které jsou poslány metodou GET, na adresu */rfid-sensor/:username*. Vstupní parametr do flow je username (uživatelské jméno) typu *string*.

Prvním krokem ve flow je větvení na základě jména uživatele. Na začátku flow není možné přidat větvení, řešením daného problému je akce typu *No-Op*, která nedělá nic a slouží jako označení pro sadu aktivit nebo jako první aktivita ve flow pro větvení. Z aktivity No-Op je větvení pro tři případy:

1. Uživatelské jméno se rovná user1
2. Uživatelské jméno se rovná user2
3. Uživatelské jméno se nerovná user1 ani user2

Pro první a druhý případ budou spuštěny subflow s vybranými parametry, pro poslední případ vyskočí výjimka, že uživatel neexistuje. Na obrázku 27 je uveden finální stav hlavního flow pro daný scénář.



Obrázek 27: Hlavní flow pro scénář 2, zdroj: vlastní

3.3 Scénář 3: dovolená

Lidé, když jedou na dovolenou a budou určitou dobu mimo místo svého bydliště, chtějí mít svou domácnost chráněnou proti vloupání. Jedním ze způsobů prevence vloupání je imitace přítomnosti člověka v domácnosti, protože pachatel nebude chtít vloupat se do bytu, když nemá jistotu, že v onom bytě nikdo není, a vybere si pro vloupání jiný cíl.

Pro případ, kdy se pachatel do domácnosti vloupe, je potřeba mít možnost to detekovat a informovat uživatele, ochrannou službu nebo policii. Způsoby detekce mohou být různé, například kamerový systém s detekcí pohybu, senzor rozbití oken, senzor otevření dveří, senzor pohybu atd. Pro informování uživatele nebo policie mohou být využity různé způsoby od poslání e-mailu po volání robotem.

V daném scénáři budou implementovány možnosti imitace přítomnosti uživatele, zapnutí různých zařízení dle rozvrhu. Při spuštěném scénáři bude podle rozvrhu probíhat automatické zapnutí světla, spuštění přehrávače hudby, otevíraní oken pro imitaci přítomnosti uživatele. Také budou sledovány zprávy ze senzoru pohybu, v případě aktivace senzoru bude poslána notifikace uživateli a policii pomocí REST API.

Rozvrh pro aktivaci zařízení je:

- 7:00 – otevření oken, zapnutí playlistu POP, zapnutí světel.
- 8:00 – vypnutí přehrávače hudby a světel.
- 18:00 – zapnutí světel.
- 21:00 – zapnutí playlistu RELAX a zavření oken.
- 22:00 – vypnutí světel v celé domácnosti a vypnutí přehrávače.

Seznam zařízení pro daný scénář:

- Ovladač světel
- Přehrávač hudby
- Ovladač oken
- Senzor pohybu

Diagramy chování scénáře jsou v příloze C.

3.3.1 Zetta

Stejně jako pro předchozí scénář je i zde potřeba založit nový projekt, příklad zakládání projektu v Zettě je popsán v kapitole 3.1.1. V rámci scénáře, stejně jako ve scénáři 2, má každé zařízení vlastní ovladač, servis pro posílání zpráv a servis pro ukládání a změnu stavu režimu dovolené, dále také jsou nastaveny typy zařízení pro dodržení struktury a možnosti opakovaného použití.

Ovladač pro senzor pohybu má jednoduchou strukturu, má pouze jeden stav a jednu funkci, která dělá jen to, že vypíše do konzole zprávu o tom, že byl zaznamenán pohyb. Tato funkce je potřeba pro správné chovaní a možnost detekovat volání funkce v rámci frameworku. Ve zdrojovém kódu 9 je uveden příklad této funkce.

```
MotionSensor.prototype.motion = function (cb) {
  console.log("Motion detected"); // Výpis do konzole
  cb(); // Volání callback funkce
}
```

Zdrojový kód 9: Příklad funkce aktivace senzoru pohybu, zdroj: vlastní

Ovladače pro přehrávač hudby a oken byly použity ze scénáře 2. Stejným způsobem byl implementován ovladač pro ovládání světel, který akceptuje dva příkazy: *turnOn* pro zapnutí světel a *turnOff* pro vypnutí.

Ovladač pro posílání zpráv policii a uživateli má pouze jeden stav READY (připraven) a přijímá dva příkazy: *notifyPolice* pro posílání zpráv policii a *notifyUser* pro posílání zpráv

uživateli. Pro případ, že nastane chyba a zpráva policii nebude doručena, uživateli bude poslána další zpráva, že je potřeba kontaktovat policii jiným způsobem.

Ve zdrojovém kódu 10 je uveden příklad funkce pro posílání zpráv policii pomocí knihovny *axios*. Proměnná *POLICE_URL* definuje adresu pro přijetí zpráv, proměnná *MOTION_DETECTED_POLICE_MESSAGE_BODY* má obsah zprávy se jménem uživatele, adresou atd. Pomocí funkce *catch* bude zachycena výjimka pro případ, že zpráva policii nebude doručena a bude poslána zpráva uživateli. *SMS_GATE_URL* je adresou servisu třetí strany pro posílání SMS zpráv, *FAILED_SENT_TO_POLICE_MESSAGE_BODY* definuje zprávu, která má v sobě telefonní číslo uživatele a text, že zpráva policii nebyla doručena.

```
NotificationController.prototype.notifyPolice = function (cb) {
    console.error("Notify police");
    axios.post(POLICE_URL, MOTION_DETECTED_POLICE_MESSAGE_BODY)
    .catch(error => {
        console.error("Failed to notify police")
        axios.post(SMS_GATE_URL,
FAILED_SENT_TO_POLICE_SMS_MESSAGE_BODY);
    })
    cb();
}
```

Zdrojový kód 10: Funkce pro posílání zpráv policii, zdroj: vlastní

Modul *Scout* je implementován stejně jako v kapitole 3.2.1, jen používá zařízení, které je definované v seznamu pro daný scénář.

Pro implementaci plánování úloh je použita knihovna *node-cron*, kterou je možné nainstalovat do projektu pomocí příkazu *npm install node-cron*. Knihovna je určena pro naplánování volání funkce v JavaScriptu a podporuje formát rozvrhu typu *cron*. Formát typu *cron* podporuje definice rozvrhu pro vteřinu (0-59), minutu (0-59), hodinu (0-23), den v měsíci (1-31), číslo měsíce (1-12) a den v týdnu (0-7), kde 0 a 7 je neděle.

Ve zdrojovém kódu 11 je uveden příklad pro spuštění úloh, které mají proběhnout v 7:00, každý den. Funkce *schedule* z knihovny má dva parametry: *cron* výraz a funkci, kterou plánovač spouští. V daném případě jako první krok probíhá kontrola stavu režimu dovolené. Pokud je režim ve vypnutém stavu, bude vypsána zpráva do konzole a běh funkce. Dále proběhne volání příkazu pro vykonání akce, v daném případě proběhne otevření oken, spuštění přehrávání hudby žánru POP a zapnutí světel.

```
cron.schedule("0 0 7 * * *", date => {
   if (vacationController.state == "DISABLED") {
      console.log("Vacation is disabled");
      return;
   }
   windowsController.call("open");
   musicPlayer.call("playPopMusic");
   lightController.call("turnOn");
});
```

Zdrojový kód 11: Plánování úlohy v 7:00, zdroj: vlastní

Pro případ aktivace senzoru pohybu (proběhlo spuštění příkazu *motion* u ovladače senzoru pohybu), stejně jako v příkladu před tím, proběhne kontrola, zda režim dovolené je zapnutý, pokud ne, vypíše se zpráva do konzole a běh funkce bude ukončen. Pokud režim dovolené je zapnutý, proběhne spuštění příkazu *notifyPolice* a *notifyUser* v ovladači pro posílání zpráv.

Posledním krokem je import modulu, v souboru *index.js* pomocí funkce *use* je potřeba importovat moduly do serveru Zetty, viz zdrojový kód 8.

3.3.2 Node-RED

Implementace daného scénáře v rámci frameworku Node-RED se bude skládat z následujících částí:

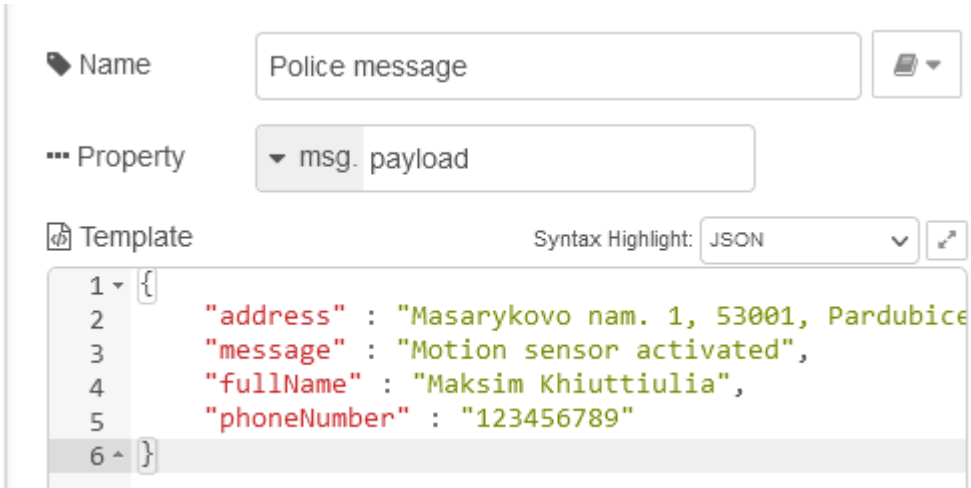
- Hlavní flow – obsahuje veškerou logiku a vstupní body do scénáře.
- Subflow pro notifikace – obsahuje logiku spojenou s posíláním zpráv policii a uživateli.
- Subflow pro ovládání zařízení – obsahuje logiku pro práci se zařízeními, ovladače přehrávače hudby a oken byly použity z předchozího scénáře.

Flow pro notifikace má jako vstupní parametry URL adresy policie a servisu pro posílání SMS zpráv. Pro posílání REST požadavku byl použit modul *HTTP Request*, detailní popis modulu je v kapitole 3.2.2. Tento modul nenabízí možnost zadat tělo požadavku přímo v modulu, existuje jenom možnost nastavit tělo požadavku jako vstupní parametr typu *msg.payload*. Pro řešení tohoto problému existuje několik možností, například načtení dat ze souboru pomocí modulu *Read File* (načíst soubor) nebo přímé nastavení pomocí modulu *Template* (šablona). V rámci této diplomové práce byl použit modul *Template*, a to z toho důvodu, že data budou statická a k jejich změně nedojde.

Modul *Template* má následující možnosti konfigurace:

- Name – název modulu.
- Property – vlastnost, do které bude nastavena hodnota, může být typu *msg*, *flow* a *global*, pro další použití s modulem *HTTP Request* je potřeba zvolit typ *msg* a název vlastnosti *Payload*.
- Template – obsah zprávy, může byt v jakémkoliv formátu, například obyčejný text nebo formát JSON.
- Format – pomocí parametru je možné zadat formát, ve kterém bude možné přidávat hodnoty proměnných. V aktuální verzi Node-RED jsou možnosti:
  - Plain text – obyčejný text, nepodporuje žádné proměnné, což znamená, že výstup bude stejný jako text, který je zadán do parametru *Template*.
  - Mustache template – podporuje nastavení proměnných v textu ve formátu *Mustache*. Tento formát má syntaxi, která se skládá ze třech párů složených závorek a názvu proměnné, například *{{{msg.text}}}*, kde *msg.text* je proměnnou, která je vstupem do modulu *Template*.
- Output as – definuje výstupní formát, může to být *Plain Text* (obyčejný text), *Parsed JSON* (parsování do JSON formátu) nebo *Parsed YAML* (parsování do YAML formátu).

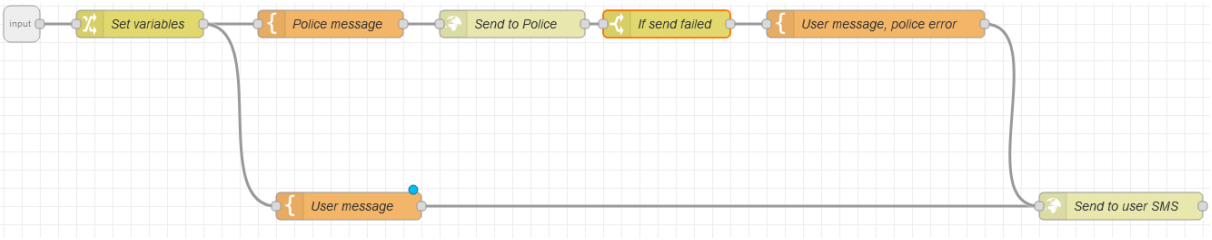
Na obrázku 28 je uveden příklad definice modulu pro zprávu, která bude poslána policii.



Obrázek 28: Definice uzlu Template v Node-RED, zdroj: vlastní

Celý subflow pro posílání zpráv je zobrazen na obrázku 29. Subflow dostává na vstupu URL adresy, kam je potřeba poslat požadavky, a proběhne paralelní posílání požadavků. Pro případ, kdy se požadavek policii nepovedlo poslat kvůli chybě, výstupem z modulu *Send to*

Police (angl. Poslat policii) bude status odpovědi odlišný od 200, což v HTTP/HTTPS protokolu znamená úspěch, a proběhne poslání další zprávy uživateli o tom, že zpráva policii nebyla doručena a je potřeba kontaktovat policii jiným způsobem, například zavolat. Definice modulu *Switch* pro zjištění statusu odpovědi je zobrazena na obrázku 30. Proměnná *msg.res.statusCode* obsahuje status odpovědi. Popis modulu *Switch* je v kapitole 3.2.2.



Obrázek 29: Flow pro posílání notifikací v Node-RED, zdroj: vlastní

Obrázek 30: Definice podmínky pro HTTP status, zdroj: vlastní

Pro plánování provádění akcí v určitém termínu je potřeba použít plánovač. V Node-RED plánovač není, pro řešení tohoto problému byl nainstalován plánovač *node-red-contrib-cron-plus*. Pro framework existují i jiné plánovače, ale z pohledu autora vybraný plánovač je nejlepší, má víc funkcí a jednodušeji se používá.

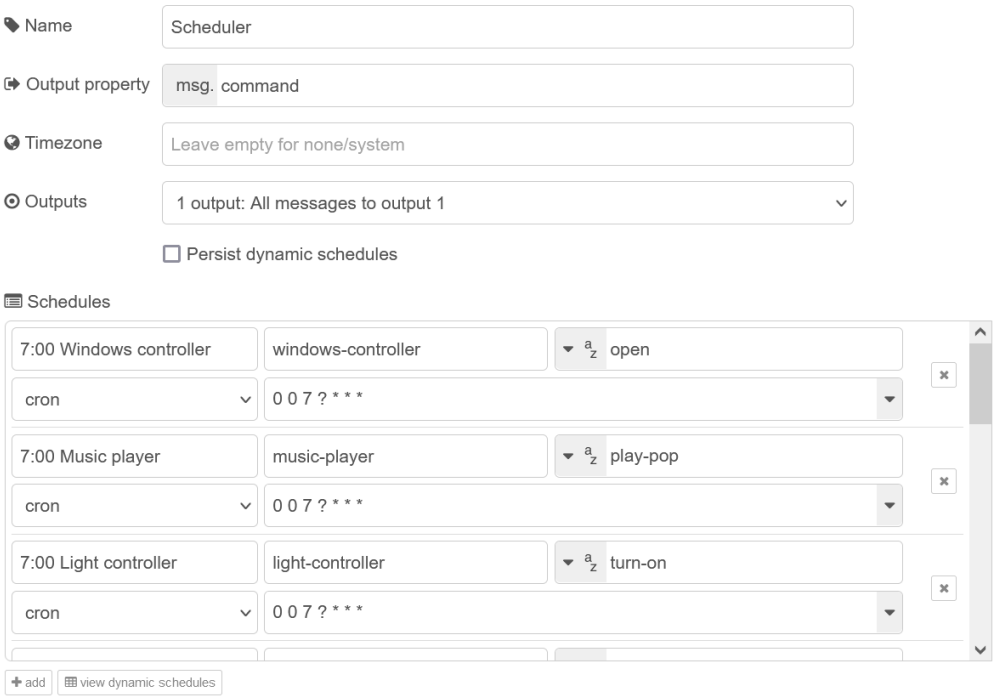
Aktuální verze 1.5.6 má následující možnosti plánování:

- Statické plánovaní ve formátu *cron*, přičemž plánovač má grafické rozhraní pro tvorbu *cron* výrazů, které nevyžaduje znalost formátu *cron*.
- Plánování nových úloh v době běhu programu.
- Plánování úloh podle polohy slunce a spuštění může proběhnout v době svítání nebo západu slunce v určité poloze ve světě.
- Plánování spuštění v přesně definované dny.

Plánovač má následující parametry pro konfiguraci:

- Name – název uzlu plánovače.
- Output Property – do jaké proměnné bude zapsána hodnota při spuštění úlohy.
- Timezone – definuje časové pásmo, pro které je plánovač vytvořen, pokud parametr je prázdný, bude použito pásmo definované v operačním systému.
- Outputs – definuje počet a typy výstupních hodnot z plánovače. Jsou následující možnosti:
  - 1 output – všechny zprávy budou na jednom výstupu.
  - 2 outputs – plánovač bude mít 2 výstupy, první je určen pro zprávy, které plánovač posílá při spuštění úlohy, druhý výstup je určen pro příkazy.
  - Fan out – každá statická úloha bude mít vlastní výstup, jeden výstup bude pro dynamickou úlohu, která je vytvořena v době běhu, a jeden výstup bude pro příkazy.
- Schedules – seznam statických plánovaných úloh. Každá úloha má:
  - Název – unikátní název v rámci jednoho plánovače.
  - Topic – název proměnné, do které bude nastavena hodnota.
  - Payload – hodnota, která bude nastavena do vybrané proměnné.
  - Typ plánování – definuje typ plánování, jsou možnosti:
    - Cron – výraz ve formátu *cron*, jehož popis je v kapitole 3.3.1.
    - Data sequence – seznam dat, kdy je potřeba spustit úlohu.
    - Solar events – poloha slunce nebo měsíce na určitých souřadnicích, při které je potřeba spustit úlohu, například západ slunce v Praze.
    - Timetable – definice, kdy je potřeba spustit úlohu, formát závisí na výběru typu plánování.

Příklad plánovače pro daný scénář je na obrázku 31, pro plánování byl vybrán typ plánování *cron* výrazem kvůli opakujícímu se rozvrhu. Pro každé zařízení byl definován vlastní *topic* a pro každou akci zařízení byl definován vlastní název. Na základě těchto hodnot bude dále probíhat rozhodování.



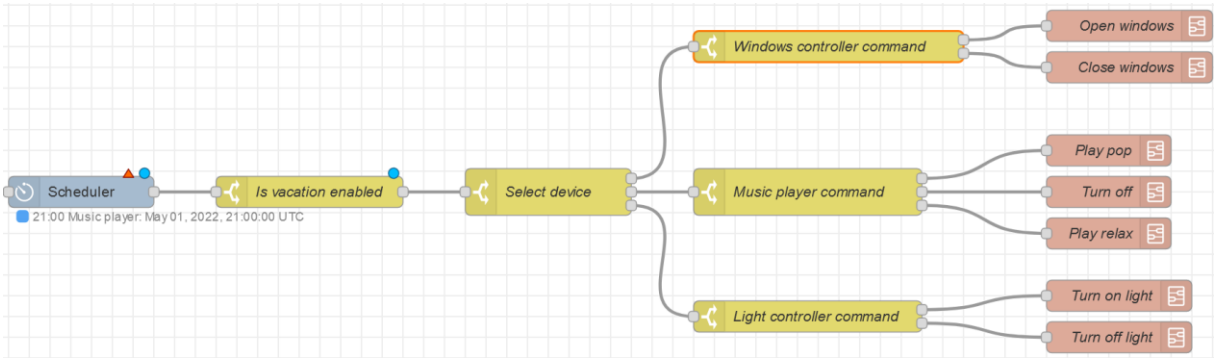
Obrázek 31: Definice uzlu plánovače v Node-RED, zdroj: vlastní

Pro spuštění úloh, jen když režim dovolené je zapnutý, byla použita globální proměnná v rámci flow s datovým typem *boolean* a názvem *vacation-enabled*. Pro rozhodování byl použit modul *Switch* s podmínkou *flow.vacation-enabled* rovná se *true* (pravda). Pokud hodnota je odlišná od hodnoty *true*, zpracování dál neproběhne. Místo globální proměnné v rámci flow (prefix flow.) je možné použít globální proměnnou (prefix global.), která bude přístupna v rámci celé aplikace, nejen v rámci flow.

Dalším rozhodováním je rozhodování na základě typu zařízení pomocí modulu *Switch*. V rámci modulu jsou tři podmínky, pokud *msg.topic* (proměnná *topic* je výstupem z plánovače) rovná se *windows-controller* (ovladač oken), *music-player* (přehrávač hudby) nebo *light-controller* (ovladač světel).

Posledním rozhodováním je akce, kterou je potřeba provést: každé zařízení má vlastní seznam akcí, které je možné provést. Rozhodování probíhá na základě proměnné *command*, která je definována v plánovači pomocí parametru *Output Property*.

Na obrázku 32 je uvedena větev pro ovládání zařízení podle rozvrhu.

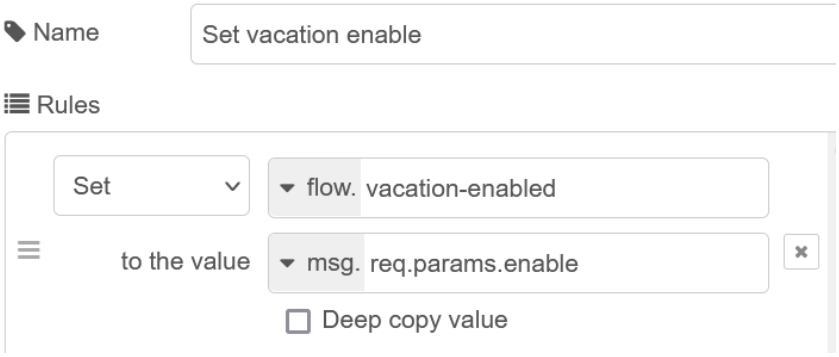


Obrázek 32: Flow pro spuštění plánovaných úloh v Node-RED, zdroj: vlastní

Další věcí, kterou je potřeba implementovat, je zapnutí nebo vypnutí režimu dovolené. Pro ovládání režimu dovolené v rámci celého scénáře pro každý framework byl zvolen způsob ovládání pomoci REST požadavku typu GET, ale Node-RED nabízí možnost vytvářet grafické rozhraní pro uživatele. Příklad tvorby grafického rozhraní je popsán v kapitole 3.1.2.

Pro přijímání REST požadavku typu GET byl použit modul *http in* a *http response*, jejichž popis je uveden v kapitole 3.2.2. Daný modul je nastaven pro přijetí zpráv na adrese */vacation/:enable*, kde *:enable* je proměnnou, hodnota které bude nastavena do proměnné *flow*.

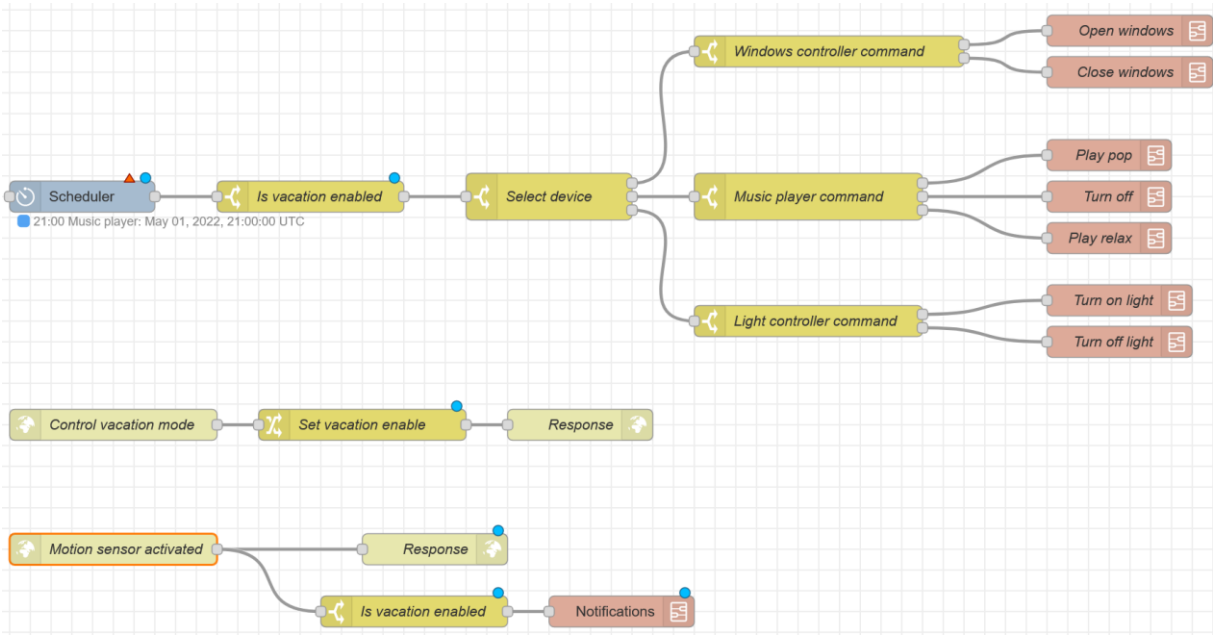
Pomocí modulu *Change* se hodnota, která je výstupem z modulu *http in*, ukládá do proměnné *vacation-enable* v rámci flow viz obrázek 33.



Obrázek 33: Nastavení globální proměnné v Node-RED, zdroj: vlastní

Poslední částí realizace scénáře je posílání zpráv policii a uživateli pomocí subflow *Notifications*, který byl vytvořen předtím. Pro přijetí zpráv byl využit modul *http in* a *http response*. Přijetí zpráv probíhá na adrese */motion-sensor/activated* pomocí metody GET. Pro posílání zpráv jen v případě, kdy režim dovolené je zapnut, je potřeba přidat kontrolu na stav proměnné *flow.vacation-enabled* pomocí modulu *Switch*, stejně jako ve větvi pro plánování úloh.

Na obrázku 34 je uvedena implementace hlavního flow pro daný scénář.



Obrázek 34: Scénář 3 v Node-RED, zdroj: vlastní

3.3.3 Flogo

Implementace scénáře pro framework Project Flogo se skládá z akcí pro ovládání zařízení, část kterých byla převzata z předchozího scénáře, akcí pro ovládání režimu dovolené, akcí pro posílání notifikací a akcí pro plánování úloh.

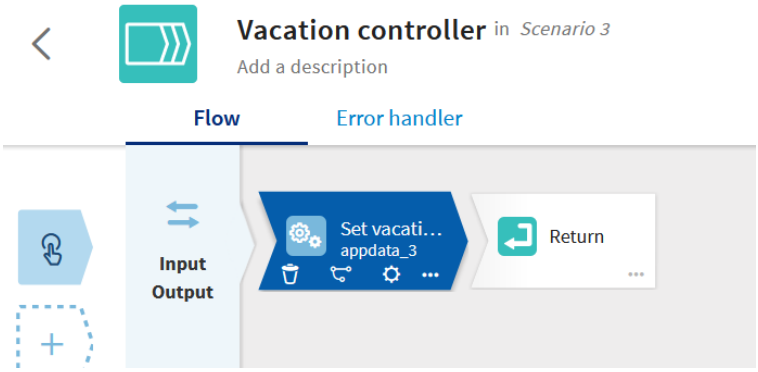
Akce pro ovládání režimu dovolené jako vstupní bod má spouštěč typu *Receive HTTP Message*, detailní popis kterého je v kapitole 3.2.3. Daný spouštěč poslouchá zprávy, které jsou poslány pomocí metody GET, na portu číslo 7001, a má cestu */vacation/:enable*, kde *:enable* je booleovskou proměnnou, hodnota které bude nastavena do globální proměnné.

Pro práci s globálními proměnnými je ve frameworku akce typu *Use global App attribute*. Tato aktivita umožňuje nastavovat proměnné, které jsou přístupné v rámci celé aplikace, nebo načítat hodnotu proměnných. Akce má následující parametry:

- Settings :
  - name – název proměnné definovaný v závorkách.
  - op – operace, kterou je potřeba provést nad proměnnou: *set* (nastavit hodnotu) nebo *get* (načíst hodnotu).
  - type – datový typ hodnoty, může být *string* (řetězec), *int* (cele číslo), *boolean* (booleovská proměnná), *array* (pole hodnot) atd.

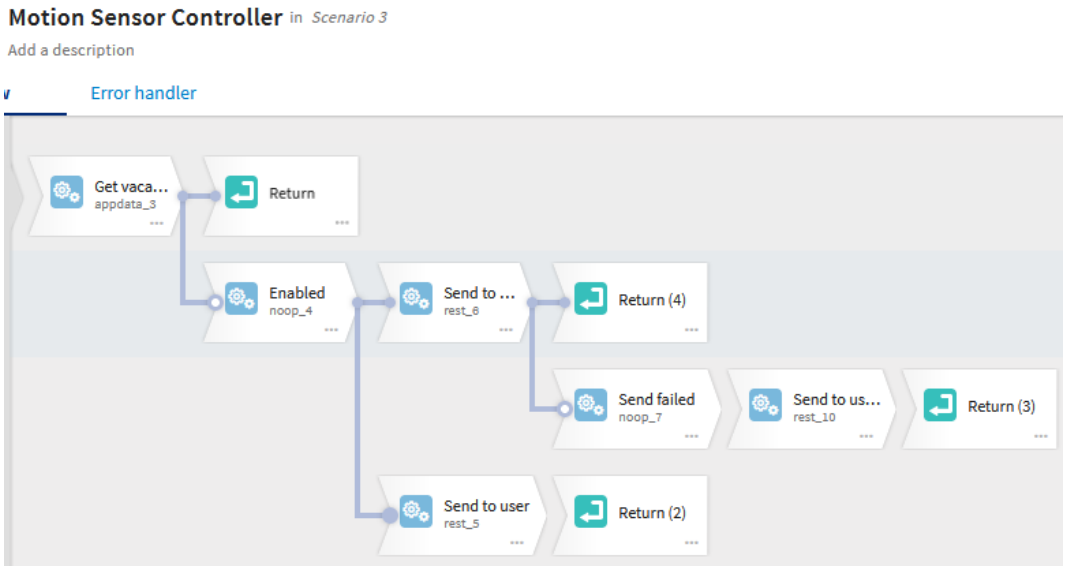
- Map Inputs:
  - value – hodnota, kterou je potřeba nastavit do proměnné, může být statická nebo může být vstupní hodnotou do aktivity.

Pro nastavování stavu režimu dovolené byla použita proměnná s názvem *VACATION_ENABLED* datového typu *boolean*, hodnota pro nastavení proměnné se získává z proměnné *flow.enable*, která je namapována jako vstupní hodnota do akce ze spouštěče. Na obrázku 35 je uvedena akce pro ovládání režimu dovolené.



Obrázek 35: Akce pro nastavení režimu dovolené v Project Flogo, zdroj: vlastní

Další akce je určena pro posílání notifikací, když byl aktivován senzor pohybu. Vstupním bodem do akce je spouštěč typu *Receive HTTP Message*, který poslouchá zprávy na portu 8000, které jsou poslané metodou GET na adresu */motion-sensor/activated*. Prvním krokem pro danou akci je ověření stavu režimu dovolené. Pro načtení hodnoty proměnné *VACATION_ENABLED* je použita aktivita typu *Use global App attribute*, ale oproti předchozímu použití je zvolen typ operace *get*. Aktivita má pouze jednu větev, kterou půjde zpracování pro případ, kdy hodnota proměnné rovná se *true*. Dál probíhá paralelní odesílání zpráv policii a uživateli pomocí akce *REST Invoke*. Po odeslání zprávy policii probíhá kontrola statusu odpovědi, pokud se status odpovědi nerovná 200, proběhne poslání další zprávy uživateli. Na obrázku 36 je uveden případ aktivity pro posílání zpráv.



Obrázek 36: Posílání notifikací v Project Flogo, zdroj: vlastní

Poslední akcí v rámci implementace scénáře pro Project Flogo je implementace plánovače úloh. Project Flogo neobsahuje žádný plánovač a neexistují žádné hotové moduly pro plánování úloh. Pro implementaci plánovače na základě prvků, které jsou ve frameworku, byla použita následující možnost. Byt vytvořen spouštěč typu *Timer*, který je součástí frameworku a je určen pro spuštění akcí přes definovaný časový interval.

Spouštěč má následující parametry:

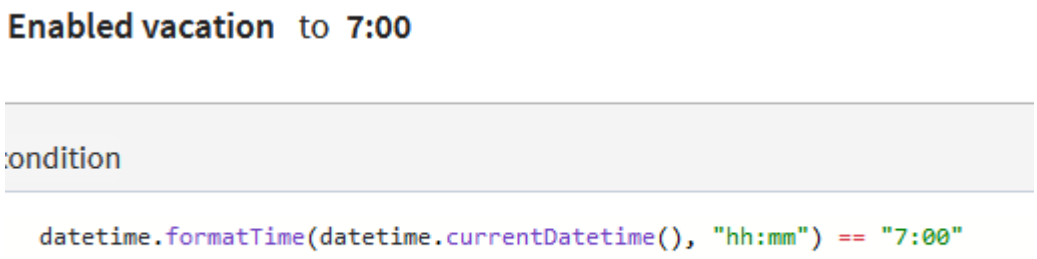
- startDelay – doba zpoždění před prvním spuštěním po spuštění celé aplikace, hodnota se skládá z délky intervalu a časové jednotky, například pro spuštění každých 5 vteřin parametr má být nastaven na hodnotu 5 s.
- repeatInterval – definuje časový interval, po uplynutí kterého proběhne aktivace spouštěče, hodnota má mít stejný formát jako parametr startDelay.

Pro plánovač úloh je spouštěč nastaven na spuštění každou minutu. Pro daný interval odchylka spuštění úloh bude maximálně 60 vteřin, je možné ji ignorovat, a to z důvodu, že scénář nevyžaduje přesnost na vteřiny.

První aktivitou v rámci této akce je zjištění stavu režimu dovolené pomocí akce typu *Use global App attribute*. Pokud režim dovolené je vypnutý, aktivita bude ukončena. Pokud režim dovolené je zapnutý, proběhne rozhodování do šesti větví.

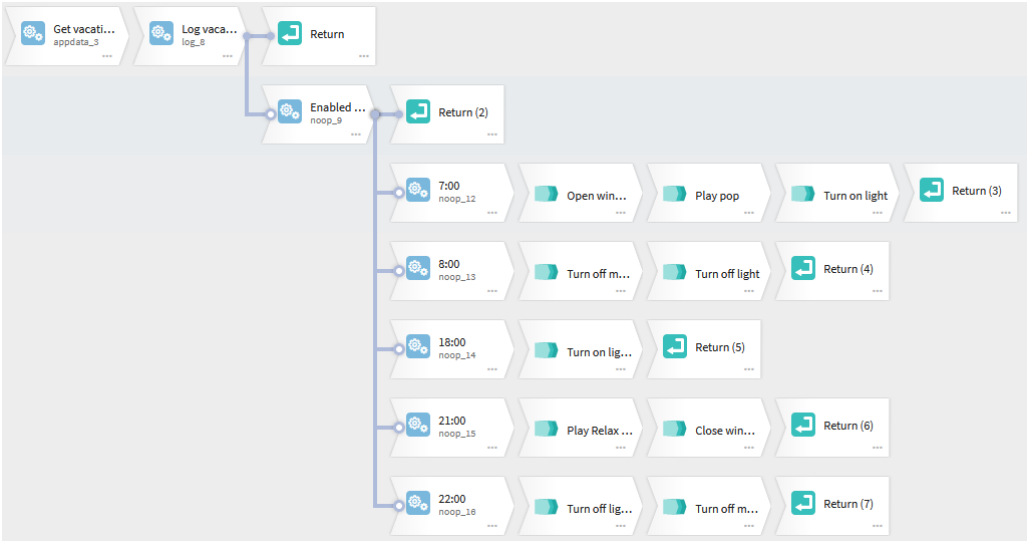
První větev je určena pro případ, kdy ani jeden čas nevyhovuje rozvrhu. Každá další větev má podmínku pro akce, které je potřeba provést v určitý čas. Pro to je potřeba porovnat aktuální čas v systému s časem pro spuštění sady akcí.

Pro získání aktuálního času systému je funkce *currentDatetime* z balíku *datetime*, která vrací aktuální datum a čas. Dalším krokem je formátování času do formátu *hh:mm*, kde *hh* je aktuální hodina a *mm* aktuální minuta. Formátování lze provést pomocí funkce *formatTime*, která má dva parametry: aktuální čas a formát výstupu. Posledním krokem je porovnání s časem, kdy je potřeba spustit sadu úloh. Na obrázku 37 je uveden příklad podmínky pro spuštění akce v 7:00.



Obrázek 37: Funkce pro porovnání času v Project Flogo, zdroj: vlastní

Dále probíhá zpracování větve, ve které bude provedena akce nad zařízením pomocí již existujících akcí. Na obrázku 38 je uveden příklad celého flow pro plánování spuštění úloh.



Obrázek 38: Akce plánovače v Project Flogo, zdroj: vlastní

4 VÝVOJ MODULŮ AKCÍ PRO PROJECT FLOGO

Project Flogo ve verzi 0.5.8 má vetší počet různých modulů, které nejsou kompatibilní s aktuální verzí 1.0.0. Z toho důvodu pro scénáře 1 a 2 byly vyvinuty moduly akcí.

Každý modul, který je určen pro použití v rámci frameworku Project Flogo, musí být vyvinut v programovacím jazyce Go Lang a mít následující soubory:

- activity.go – soubor obsahuje logiku pro aktivitu, v případě vývoje triggeru soubor musí mít název trigger.go.
- activity_test.go – soubor obsahuje testy pro aktivitu, v případě vývoje triggeru soubor musí mít název trigger_test.go, testy jsou nepovinné.
- description.json – soubor obsahuje základní údaje o modulu, např. název, popis, informace o autorovi atd. V souboru je také definice parametrů pro nastavení, vstupní a výstupní parametry.

Pro práci s parametry v rámci logiky je potřeba definovat následující datové typy:

- Settings – nastavení pro modul, hodnoty jsou statické a nemůžou se měnit v době běhu aplikace.
- Input – vstupní parametry pro modul, hodnoty jsou dynamické a můžou se měnit v době běhu aplikace. Hodnoty je možné získat z předchozí akce v rámci aplikace nebo nastavit ručně.
- Output – výstupní hodnoty z modulu.

Struktura definice jednotlivých parametrů v rámci každého datového typu je: *nazev_parametru datovy_typ 'md:nazev_parametru_v_description.json'*. Ve zdrojovém kódu 12 je uveden příklad definice parametru. Parameter1 má datový typ *int* (celé číslo) a je definován v souboru *description.json* jako parametr Parameter1, pomocí slova *required* je definováno, že parametr je povinný.

```
type Input struct {
    Parameter1 int 'md:"Parameter1,required"'
}
```

Zdrojový kód 12: Definice vstupních parametrů do aktivity, zdroj: vlastní

Další věcí, kterou má obsahovat aktivita, jsou tzv. metadata. Metadata jsou sadou, která se skládá ze *Settings*, *Input* a *Output*, ale může obsahovat libovolnou kombinaci prvků. Ve zdrojovém kódu 13 je uveden příklad definice metadat, která se skládají z vstupních a výstupních parametrů.

```
func (a *Activity) Metadata() *activity.Metadata {
  return activity.ToMetadata(&Input{}, &Output{})
}
```

Zdrojový kód 13: Definice funkce metadata, zdroj: vlastní

Pro vytvoření nové instance je možné použití konstruktoru pomocí definice funkce *New*. Konstruktor je určen pro vytváření nové instance aktivity, ale není povinný. Konstruktor je potřeba používat jen v případech, kdy každá aktivita má mít vlastní data pro sekci *Settings*. Ve zdrojovém kódu 14 je uveden příklad, jak má funkce *New* vypadat.

```
func New(ctx activity.InitContext) (activity.Activity, error) {
...
}
```

Zdrojový kód 14: Definice funkce konstruktoru, zdroj: vlastní

Pro inicializaci aktivity je potřeba definovat funkci *init*, která provede registraci aktivity při načítání aplikace. Do této funkce je potřeba dávat to, co je potřeba provést v době inicializace, například inicializaci zařízení. Ve zdrojovém kódu 15 je uveden příklad této funkce, ale s voláním konstruktoru aktivity, pro případ, kdy aktivita má sekci *Settings*.

```
func init() {
  _ = activity.Register(&Activity{})
}


func init() {
  _ = activity.Register(&Activity{}, New)
}
```

Zdrojový kód 15: Příklady definice funkce init, zdroj: vlastní

Pro načítání konfigurace ze souboru během importu a ukládání konfigurace pro export aktivita má mít implementované funkce *ToMap* pro ukládání konfigurace a *FromMap* pro načtení konfigurace. Ve zdrojovém kódu 16 je uveden příklad těchto funkcí, kde probíhá ukládání hodnoty proměnné Parameter1 a pak načítaní hodnoty. Parsování dat probíhá pomocí funkce z knihovny *coerce*, která je součástí Project Flogo. Funkci je potřeba definovat pro každou sekci konfigurace aktivity, které se používají v aktivitě.

```
func (i *Input) ToMap() map[string]interface{} {
  return map[string]interface{}{
    "Parameter1": i.Parameter1,
  }
}

func (i *Input) FromMap(values map[string]interface{}) error {
  var err error
  i.Parameter1, err = coerce.ToInt(values["Parameter1"])
  if err != nil {
    return err
  }
  return nil
}
```

Zdrojový kód 16: Definice funkcí FromMap a ToMap, zdroj: vlastní

Hlavní funkcí v rámci aktivity je *Eval*. V této funkci probíhá zpracování celé logiky aktivity. Funkce má dvě návratové hodnoty, první je výsledek, že se povedlo správné provést zpracování, druhá hodnota obsahuje chybu, kterou je možné zachytit v akci pomocí sekce *Error handler*. Ve zdrojovém kódu 17 je uveden příklad definice funkce *Eval*. Tato definice nedělá nic jiného, než že vypíše do logu řetězec „Hello World!“.

```
  func (a *Activity) Eval(ctx activity.Context) (done bool, err
error) {
    log.RootLogger().Debug("Hello world!")
    return true, nil
  }
```

Zdrojový kód 17: Definice funkce Eval, zdroj: vlastní

Soubor *descriptor.json* obsahuje popis aktivity, název aktivity, verzi, jméno autora, typ v rámci frameworku, pro aktivitu má být hodnota *flogo:activity* atd. Soubor také definuje vstupní a výstupní parametry, jejich výchozí hodnoty a seznam povolených hodnot. Pro vstupní parametry JSON objekt má mít název *input*, pro výstupní parametry *output*, pro nastavení je název *settings*. Ve zdrojovém kódu 18 je uveden příklad definice vstupního parametru s názvem Parameter1 datového typu *string*, který má povolené hodnoty jen Ano a Ne.

```
{
  "name": "test-aktivity",
  "type": "flogo:activity",
  "version": "0.0.1",
  "title": "Test aktivity",
  "description": "Test aktivity",
  "input": [
    {
      "name": "Parameter1",
      "type": "string",
      "value": "Ano",
      "allowed" : ["Ano", "Ne"],
      "description": "Action"
    }
  ]}
```

Zdrojový kód 18: Definice souboru descriptor.json, zdroj: vlastní

4.1 GPIO Output

Aktivita je určena pro ovládání GPIO pinu režimu výstupu mikropočítače Raspberry Pi. Aktivita nemá nastavení a nemá výstupní parametry, používá jen vstupní parametry. Je to uděláno z toho důvodu, aby bylo možné měnit číslo pinu nebo typ operace na pinu v době běhu aplikace. Pro ovládání GPIO pinu byla použita knihovna *go-rpio* verze 4. Tato aktivita byla implementována kvůli tomu, že aktuální verze frameworku 1.0.0 není kompatibilní s aktivitou, která má podobnou funkcionalitu.

Vstupní parametry pro aktivitu jsou:

- action – parametr definuje, jakou operaci modul provede.
  - TurnOn – zapne vybraný pin, což znamená, že na pinu bude napětí.
  - TurnOff – vypne vybraný pin.
  - Toggle – změní status vybraného pinu na opačný.
- gpioPin – číslo pinu, který modul bude ovládat.

Ve funkci *Eval* probíhá definice prázdné proměnné typu *Input*, která obsahuje vstupní parametry. Pomocí funkce *GetInputObject* probíhá nastavení hodnot do proměnné. Po načtení hodnot probíhá načtení objektu typu *Pin*, což je reprezentace GPIO pinu, pomocí funkce *Pin* z knihovny *go-rpio*. Dále probíhá zápis do logu, jaká operace bude provedena na vybraném pinu, a zápis čísla pinu. Pro práci s pinem je potřeba pin inicializovat pomocí funkce *Open*. Funkce má jako návratovou hodnotu chybu, která může vzniknout během inicializace pinu. Pokud chyba je, proběhne zápis do logu, že se nepovedlo inicializovat pin, a budou vráceny hodnoty *false*, což znamená, že aktivita neproběhla korektně a vznikla chyba. Pak probíhá

nastavení pinu do režimu výstupu pomoci funkce *Output*, což znamená, že pin bude mít na výstupu napětí 5 V, když je zapnut, nebo 0 V, když je vypnut.

Na konci funkce proběhne rozhodování, jakou operaci na pinu je potřeba provést:

- Pokud operace je *TurnOn*, proběhne zapnutí pinu pomocí funkce *Write* s parametrem *High*.
- Pokud operace je *TurnOff*, proběhne vypnutí pinu pomocí funkce *Write* s parametrem *Low*.
- Pokud operace je *Toggle*, proběhne změna stavu pinu na opačný pomocí funkce *Toggle*.
- Pro ostatní hodnoty proběhne zápis do logu, že operace je neznámá, a návratové hodnoty budou *false* a prázdná chyba.

Listing zdrojového kódu aktivity je v příloze D.

4.2 Sleep

Aktivita Sleep je určena pro zastavení vlákna na určitou definovanou dobu, často se používá pro případ, kdy je po provedení určité akce potřeba počkat před zpracováním dalšího kroku, jako to bylo v případě kávovaru u přípravy kávy, pokud kávovar již připravuje kávu pro jiného uživatele. Ve frameworku Project Flogo neexistovala a dosud neexistuje žádná podobná aktivita, a proto odpovídající aktivita byla vyvinuta v rámci této diplomové práce.

Aktivita má pouze jeden vstupní parametr *SleepTime*, který definuje, kolik vteřin má vlákno počkat.

Ve funkci *Eval*, stejně jako v předchozí aktivitě, probíhá inicializace nové proměnné *input* typu *Input*, co jsou vstupní parametry. Pomocí funkce *GetInputObject* probíhá načtení hodnot do proměnné. Pro zastavení vlákna byla použita knihovna *time*, která je součástí jazyka Go Lang.

Zastavit vlákno na definovanou dobu je možné pomocí funkce *Sleep*, která má jeden parametr datového typu *Duration*. Pro převod vteřin do *Duration* je funkce *Duration*, která má jeden vstupní parametr, počet milisekund. Z důvodu, že výstupní hodnotou z funkce *Duration* jsou milisekundy a aktivita má používat sekundy, je potřeba násobit hodnotu konstantou *Second* z knihový *time*.

Zdrojový kód aktivity je v příloze E.

5 POROVNÁNÍ FRAMEWORKŮ

5.1 Časová náročnost

Dle časové náročnosti nejméně času zabrala implementace scénářů pro Node-RED. Node-RED má jednoduchou instalaci, pro začátek implementace není potřeba mít nainstalovaný žádný software, stačí jen použít nástroj Docker, pomocí jednoho příkazu spustit kontejner, který v sobě bude mít celý framework. Node-RED má podrobnou oficiální dokumentaci a také několik knih s podrobným postupem pro použití jednotlivých modulů a vývoj scénářů.

Framework Zetta dle časové náročnosti zaujímá druhé místo. Pro implementaci jakékoliv funkcionality je potřeba používat programovací jazyk JavaScript. Pro spuštění Zetty je potřeba mít systém pro správu závislostí NPM a Node.js. Složitost časové náročnosti vývoje pro Zettu je ovlivněna také tím, že Zetta má jen krátkou dokumentaci, ve které jsou popsány pouze základní moduly.

Třetí místo dle časové náročnosti zaujímá Flogo. Instalace je stejná jako u Node-RED, pro začátek vývoje stačí jen použít Docker a spustit oficiální kontejner z Docker Hub. Jedním z důvodu velké časové náročnosti je zpomalené grafické rozhraní, během zakládání jednotlivých scénářů často vzniká situace, že se nezobrazí nový scénář nebo jednotlivý modul a je potřeba nahrát stránku znova. Druhým důvodem jsou neinformativní chyby. Často nastává situace, kdy se během testování scénářů objeví chyba, ale popis chyby není zobrazen.

V tabulce 1 je uveden počet hodin, kolik autor strávil při implementaci jednotlivých scénářů včetně implementace modulů.

Tabulka 1: Časová náročnost vývoje, zdroj: vlastní

	Node-RED	Flogo	Zetta
Scénář 1	0,5 hod	2 hod	0,5 hod
Scénář 2	2 hod	8 hod	4 hod
Scénář 3	3 hod	9 hod	4 hod

5.2 Obtížnost vývoje

Node-RED má nejmenší složitost vývoje a dalšího spuštění scénářů. Každý scénář pro Node-RED byl vytvořen jen pomocí grafického rozhraní, bez psaní kódu v jakémkoliv programovacím jazyce. Framework má velké množství modulů, které jsou součástí balíčku,

a také velký počet modulů, které je možné stáhnout pomocí NPM. Během testování a nasazení jednotlivých scénářů se nevyskytly kritické chyby, které by bylo potřeba řešit.

Flogo má střední obtížnost. Flogo, stejné jako Node-RED, má grafické rozhraní pro vývoj, ale nemá uživatelské rozhraní pro ovládání konečným uživatelem. Framework má malé množství modulů, které jsou v balíčku. Existuje podpora vlastních modulů, které je možné nainstalovat například z GitHub pomocí grafického rozhraní nebo přidat ručně přes příkazový řádek. Verze frameworku 0.5.8 má vetší počet modulů, ale jádro systému je nestabilní, také existuje velký počet modulů, které jsou vyvinuty pro verzi 0.5.8 a nemají podporu aktuální verze 1.0.0. Sestavení hotového scénáře je možné provést pomocí grafického rozhraní, ale výstupní soubor nebude funkční kvůli tomu, že do souboru nebude exportován modul flow, který je základem frameworku. Pro řešení toho problému je potřeba nainstalovat Flogo do operačního systému, exportovat scénář do JSON souboru a provést sestavení na lokální verzi frameworku pomocí příkazového řádku. Výstupem bude spustitelný soubor, který je určen pro systém, na kterém proběhlo sestavení.

Zetta má vysokou obtížnost vývoje. Framework nemá žádné grafické rozhraní a celý vývoj probíhá v programovacím jazyce JavaScript. Zetta má nejmenší počet připravených modulů, a když se jedná o zařízení, které nemá připravený modul, bude potřeba celý modul naprogramovat. Podrobná dokumentace pro Zettu není, existují jen popisy jednotlivých částí frameworku v oficiálním repositáři GitHub.

5.3 Náročnost na hardware a software

Frameworky nemají přesně definované požadavky na hardware. Jen Zetta má minimální doporučené vlastnosti hardwaru:

- CPU: 500 MHz
- Operační paměť: 500 MB
- Úložiště: 500 MB

Každý z frameworků podporuje různé operační systémy typu Windows, Linux, MacOS, protože každý z frameworků má realizace v programovacím jazyce JavaScript nebo Go Lang. Pro vývoj scénářů v Zettě a Node-RED je potřeba mít NPM a Node.js, pro Flogo je potřeba mít nainstalované SDK pro Go Lang. Pro spuštění frameworku Zetta není potřeba mít jen Node.js, protože Zetta má pouze jeden soubor typu JavaScript. Sestavení scénářů pro Flogo probíhá do spustitelného souboru určeného pro typ operačního systému, na kterém proběhlo sestavení. Pro spuštění scénáře Node-RED je potřeba mít nainstalovaný celý Node-RED, ale výhodou je možnost měnit scénář za provozu, bez restartu nebo nahrání nového souboru.

5.4 Možnosti psaní kódu a použití grafických prvků

Každý z vybraných frameworků podporuje psaní programového kódu v různých variantách, ale ne každý framework podporuje použití hotových prvků pro vývoj bez kódu.

Zetta nabízí pouze možnost psaní kódu, bez využití grafických prvků. Na jednu stranu je to mínus daného frameworku pro uživatele, kteří nejsou programátory, nebo pro uživatele, kteří neznají programovací jazyk JavaScript. Na druhou stranu to nabízí více možností pro ovládání hotového scénáře, protože pro JavaScript existuje velké množství knihoven, které je možné použít.

Node-RED má z pohledu autora nejlepší grafické prostředí. Framework umožňuje využití velké sady hotových grafických prvků a také má prvky, např. *function*, které umožňují psát krátký kód v jazyce JavaScript, ale nepodporuje import vlastních knihoven. Pro psaní kódu je možné použít také uzel *exec*, který je určen pro volání příkazů operačního systému. Framework umožňuje implementaci vlastních uzlů v jazyce JavaScript.

Flogo, stejné jako Node-RED, má grafické rozhraní pro vytváření scénářů pomocí grafických prvků a nabízí možnost psaní vlastního kódu. Pomocí akce *JSExec Activity* je možné psát jednoduché skripty v jazyce JavaScript. Framework také nabízí podporu vlastních aktivit a triggerů, které je možné implementovat v jazyce Go Lang.

ZÁVĚR

Vývoj internetu a technologií vedl k tomu, že dnes můžete logicky připojit různá zařízení a fakticky naprogramovat svůj život. Dřívější programy mohly běžet pouze na počítačích nebo jiných složitých zařízeních, dnes může téměř jakékoli zařízení vykonávat funkce mikropočítače. A vyvinuté standardy umožňují, aby všechna taková zařízení byla kombinována a podřízena určité logice.

Získáváme tedy potenciál pro programování nikoli v abstraktu, ale ve skutečném reálném světě. Můžeme otevřít a zavřít dům, rozsvítit světla, ohřát jídlo, zapnout nebo vypnout topení, video atd.

Za účelem usnadnění možností programování byly po dlouhou dobu vytvářeny jazyky a frameworky, které nám umožňují abstrahovat od principů fungování zařízení a zaměřit se na programování. Frameworky využité v této práci byly vytvořeny právě pro tento účel: aby se zjednodušilo programování široké škály zařízení pomocí jednoduchých a srozumitelných konstrukcí.

V rámci diplomové práce byly vytvořeny scénáře pro frameworky Zetta, Project Flogo a Node-RED a také byly implementovány vlastní moduly pro framework Project Flogo.

Nejlepším frameworkem z pohledu autora je Node-RED. Tento framework má nejkvalitnější oficiální dokumentaci a také několik knih zaměřených na Node-RED. Dalším důvodem je aktuální verze, pokračující vývoj frameworku, vznik nových funkcí, nových modulů. Framework je stabilní, během implementace scénářů autor nenarazil na chyby spojené se samotným frameworkem.

POUŽITÁ LITERATURA

[1] CASADO-VARA, R. (2019) *Distributed continuous-time fault estimation control for multiple devices in IoT networks*. T. 7. s.11972-11984 IEEE Access.

[2] GRINGARD, S. (2017) *Интернет вещей: Будущее уже здесь.* Alpina Publisher. 186 s. ISBN: 978-5961464726

[3] GUINARD, D., TRIFA V. (2016) *Building the Web of Things: With examples in Node.js and Raspberry Pi. Manning Publications. 1 edition.* 344 s. ISBN 978-1617292682.

[4] KRANTS, M. (2018) *Интернет вещей. Новая технологическая революция*. Eksmo. 336 s. ISBN 978-5040906277

[5] LI, P. (2017) *Архитектура интернета вещей*. DMK-Press. 454 s. ISBN 978-5970606728

[6] PFISTER, C. (2011) *Getting Started with the Internet of Things 1st edition.* O'Reilly. 194 s. ISBN: 978-1449393571.

[7] TAMBOLI, A. (2019) *Build Your Own IoT Platform: Develop a Fully Flexible and Scalable Internet of Things Platform in 24 Hours 1st ed. edition.* Apress. 240 s. ISBN 978-1484244975.

[8] ГУЛИН, К. А., УСКОВ, В. С. (2017) *О роли интернета вещей в условиях перехода к четвертой промышленной революции* //Проблемы развития территории. s. 4 (90).

[9] ДОВГАЛЬ, В. А., ДОВГАЛЬ, Д. В. (2018) *Интернет Вещей: концепция, приложения и задачи* //Вестник Адыгейского государственного университета. Серия 4: Естественно-математические и технические науки. №. 1 (212).

[10] КУПРИЯНОВСКИЙ, В. П. и др. (2017) *Веб Вещей и Интернет Вещей в цифровой экономике* //International Journal of Open Information Technologies. č. 5.

[11] ЛИ, П. (2019) *Архитектура интернета вещей*. Litres. 456s. ISBN 978-5-97060-672-8

[12] ЛЬВОВИЧ, И. Я. и др. (2019) *Проблемы использования технологий интернет вещей* //Вестник Воронежского института высоких технологий. č. 1. s.73-75.

[13] ПРЕОБРАЖЕНСКИЙ, Ю. П., а МЯСНИКОВ, О. А. (2020) *Анализ перспектив информационных технологий в сфере интернет вещей* //Вестник Воронежского института высоких технологий. č. 1. s. 43-45.

[14] ПУСТЫЛЬНИК, И. Е., а ПРЕОБРАЖЕНСКИЙ, Ю. П. (2019) *Защита сообщений между сервером и приборами интернета вещей* //Вестник Воронежского института высоких технологий. č. 2 s. 40-45.

PŘÍLOHY

Příloha A – Diagram zpracování scénáře 1 73

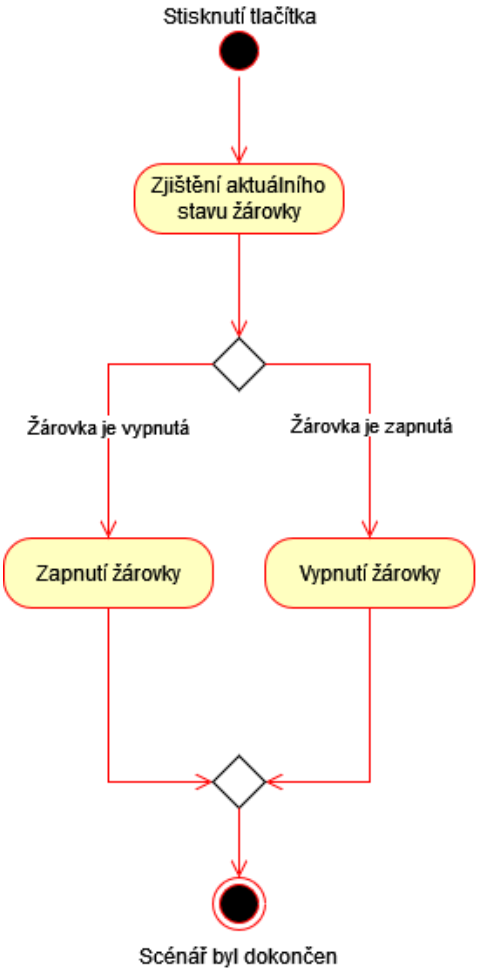
Příloha B – Diagram zpracování scénáře 2 74

Příloha C – Diagram zpracování scénáře 3 75

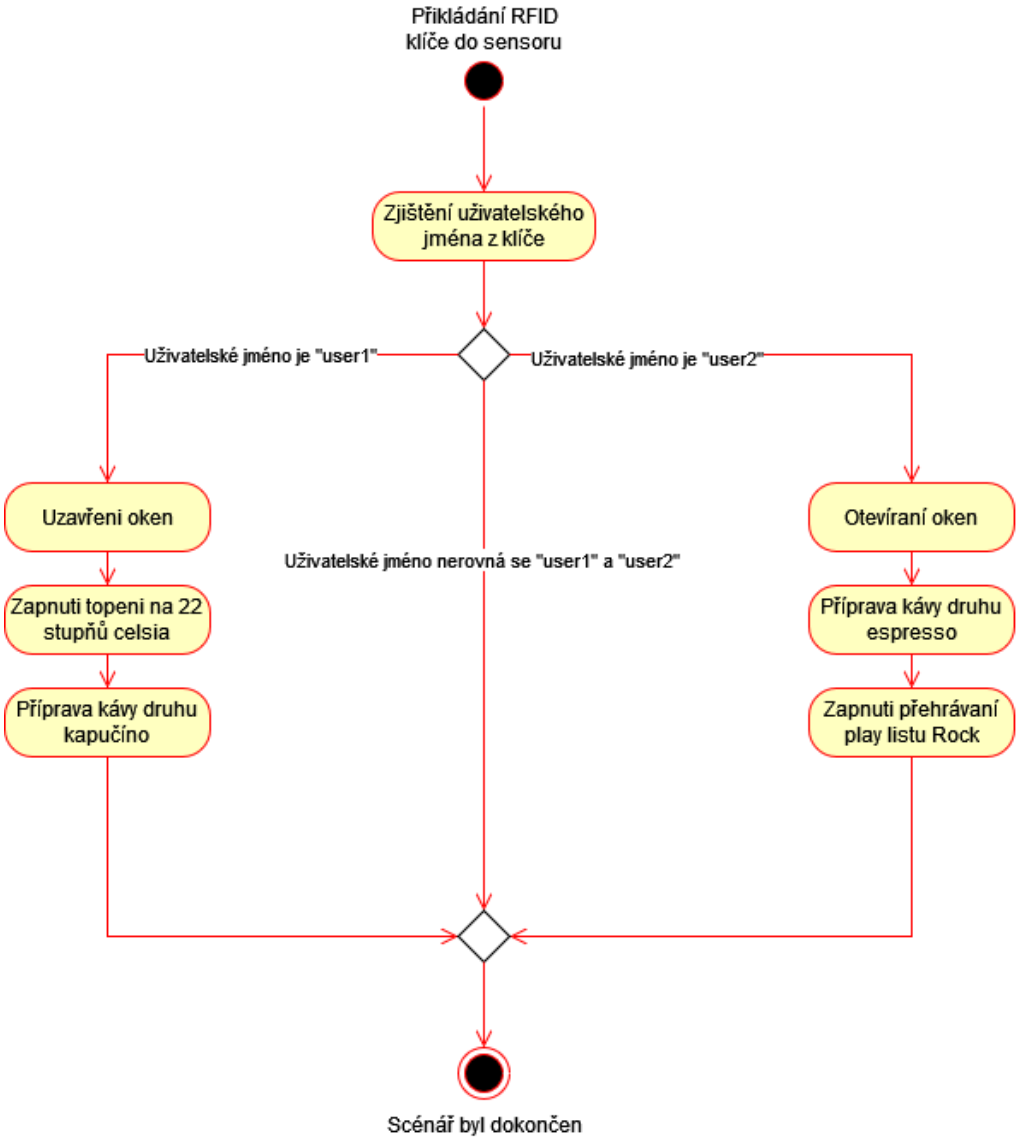
Příloha D – Zdrojový kód pro aktivitu GPIO Output 77

Příloha E – Zdrojový kód pro aktivitu Sleep 79

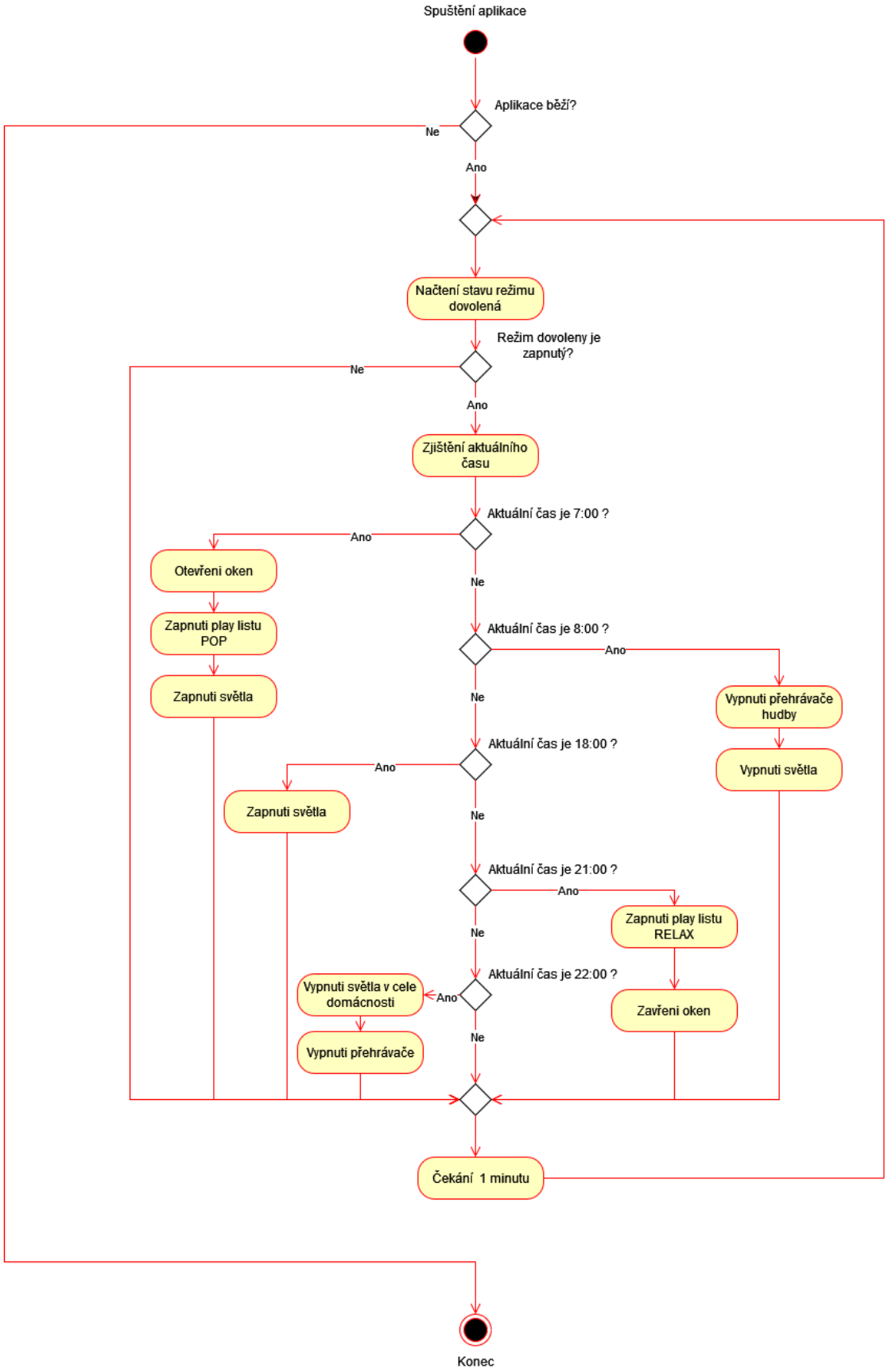
PŘÍLOHA A – DIAGRAM ZPRACOVÁNÍ SCÉNÁŘE 1

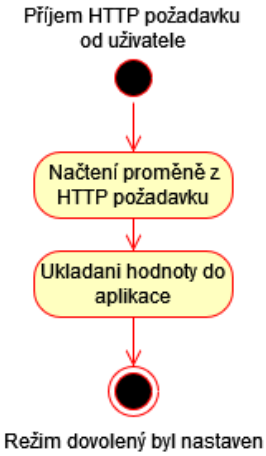


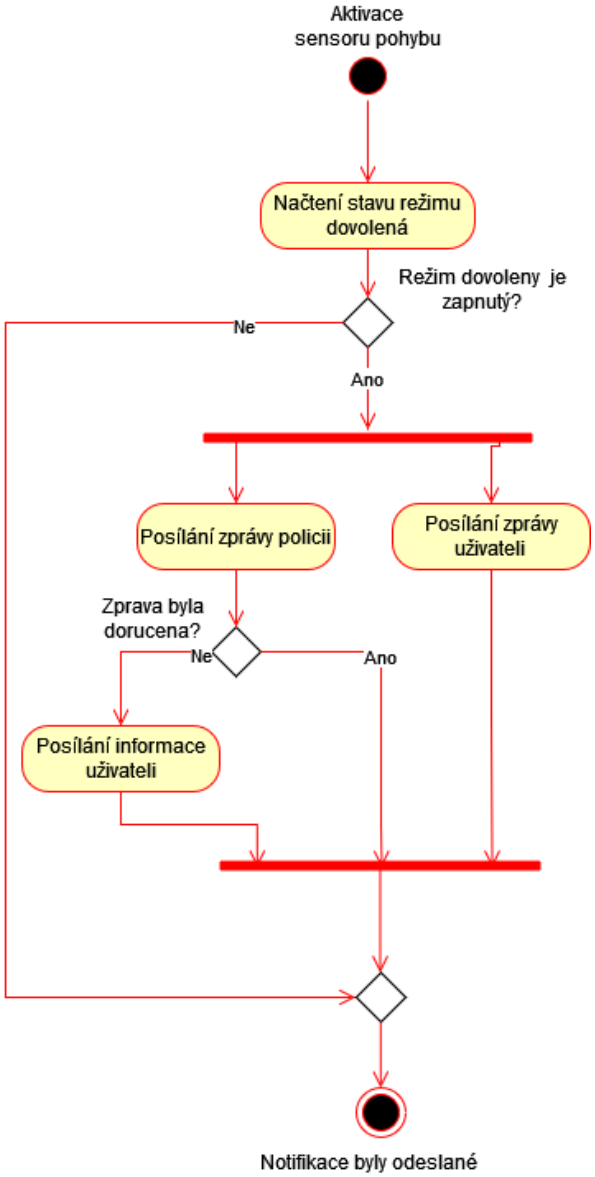
PŘÍLOHA B – DIAGRAM ZPRACOVÁNÍ SCÉNÁŘE 2



PŘÍLOHA C – DIAGRAM ZPRACOVÁNÍ SCÉNÁŘE 3




Příjem HTTP požadavku od uživatele
Načtení proměně z HTTP požadavku
Ukladani hodnoty do aplikace
Režim dovolený byl nastaven


Aktivace sensoru pohybu
Načtení stavu režimu dovolená
Režim dovoleny je zapnutý?
Ne
Ano
Posílání zprávy policii
Posílání zprávy uživateli
Zprava byla dorucena?
Ne
Ano
Posílání informace uživateli
Notifikace byly odeslané

PŘÍLOHA D – ZDROJOVÝ KÓD PRO AKTIVITU GPIO OUTPUT

```
package gpio_output

import (
  "github.com/project-flogo/core/activity"
  "github.com/project-flogo/core/data/coerce"
  "github.com/project-flogo/core/support/log"
  "github.com/stianeikeland/go-rpio/v4"
)

const (
  actionTurnOn  = "TurnOn"
  actionTurnOff = "TurnOff"
  actionToggle  = "Toggle"
)

const (
  jsonGpioPin = "gpioPin"
  jsonAction  = "action"
)

type Input struct {
  GpioPin int    'md:"GPIOPin, required"'
  Action  string 'md:"Action, required, allowed(TurnOn, TurnOff, Toggle)"'
}

type Activity struct {
}

func init() {
  _ = activity.Register(&Activity{})
}

func (i *Input) ToMap() map[string]interface{} {
  return map[string]interface{}{
    jsonGpioPin: i.GpioPin,
    jsonAction:  i.Action,
  }
}
```

```
func (i *Input) FromMap(values map[string]interface{}) error {
  var err error
  i.GpioPin, err = coerce.ToInt(values[jsonGpioPin])
  if err != nil {
    return err
  }

  i.Action, err = coerce.ToString(values[jsonAction])

  return err
}

func (a *Activity) Metadata() *activity.Metadata {
  return activity.ToMetadata(&Input{})
}

func (a *Activity) Eval(ctx activity.Context) (done bool, err error) {

  inputParams := &Input{}
  ctx.GetInputObject(inputParams)

  pinNumber := inputParams.GpioPin
  pin := rpio.Pin(pinNumber)
  action := inputParams.Action

  log.RootLogger().Debug("Action %s, pin %d", action, pinNumber)

  openError := rpio.Open()
  if openError != nil {
    log.RootLogger().Error("Failed to open pin")
    return false, openError
  }
  pin.Output()

  if action == actionTurnOn {
    pin.Write(rpio.High)
  } else if action == actionTurnOff {
    pin.Write(rpio.Low)
  } else if action == actionToggle {
    pin.Toggle()
  } else {
    log.RootLogger().Error("Unknown action %s", action)
    return false, nil
  }

  return true, nil
}
```

PŘÍLOHA E – ZDROJOVÝ KÓD PRO AKTIVITU SLEEP

```
package sleep

import (
  "github.com/project-flogo/core/activity"
  "github.com/project-flogo/core/data/coerce"

  "time"
)

type Input struct {
  SleepTime int 'md:"SleepTime,required"'
}

type Activity struct {
}

func init() {
  _ = activity.Register(&Activity{})
}

func (i *Input) ToMap() map[string]interface{} {
  return map[string]interface{}{
    "sleepTime": i.SleepTime,
  }
}
func (i *Input) FromMap(values map[string]interface{}) error {
  var err error
  i.SleepTime, err = coerce.ToInt(values["sleepTime"])
  if err != nil {
    return err
  }
  return nil
}

func (a *Activity) Metadata() *activity.Metadata {
  return activity.ToMetadata(&Input{})
}

func (a *Activity) Eval(ctx activity.Context) (done bool, err error) {
  input := &Input{}
  ctx.GetInputObject(input)

  interval := input.SleepTime
  time.Sleep(time.Duration(interval) * time.Second)
  return true, nil
}
```