



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

FACULTY OF INFORMATION TECHNOLOGY

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**

DEPARTMENT OF INFORMATION SYSTEMS

**ZPRACOVÁNÍ VIZUÁLNÍHO PROGRAMOVACÍHO  
JAZYKA NA STRANĚ SERVERU**

BACKEND PROCESSING OF VISUAL PROGRAMMING LANGUAGE

**BAKALÁŘSKÁ PRÁCE**

BACHELOR'S THESIS

**AUTOR PRÁCE**

AUTHOR

**VEDOUCÍ PRÁCE**

SUPERVISOR

**MATÚŠ TÁBI**

**Ing. PETR JOHN**

BRNO 2025

## Zadání bakalářské práce



161051

Ústav: Ústav informačních systémů (UIFS)  
Student: **Tábi Matúš**  
Program: Informační technologie  
Název: **Zpracování vizuálního programovacího jazyka na straně serveru**  
Kategorie: Informační systémy  
Akademický rok: 2024/25

### Zadání:

1. Prostudujte oblast internetu věcí (*Internet of Things, IoT*) a chytrých měst (*Smart City*).
2. Prostudujte principy reprezentace vizuálních programovacích jazyků (VPL), zaměřte se na vizuální jazyky použitelné v IoT.
3. Analyzujte požadavky na vyhodnocování těchto programů. Analyzujte existující platformu pro zprávu IoT zařízení.
4. Na základě analýzy navrhněte vhodný způsob spracování VPL v jazyce Go a jeho integraci do aplikace Pocketix.
5. Navržené rozšíření implementujte a integrujte.
6. Proveďte zhodnocení zvoleného přístupu a navrhněte pokračování projektu.

### Literatura:

- Greengard, S. (2015). *The Internet of Things*. MIT Press, ISBN 978-026-2527-736.
- Badii, C., Bellini, P., Difino, A., Nesi, P., Pantaleo, G., & Paolucci, M. (2019). Microservices suite for smart city applications. *Sensors*, 19(21), 4798.
- Hynek, J. a spol. (2023). Služby pro systém řízení a monitoringu vody v retenčních nádržích. Výzkumná zpráva. Vysoké učení technické v Brně.
- Kuhail, M. A., Farooq, S., Hammad, R. & Bahja, M. (2021). Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*, 9, (pp. 14181-14202).
- Ray, P. P. (2017). A survey on visual programming languages in internet of things. *Scientific Programming*, 2017.
- John, P. (2024). Optimising processes in IoT. Pojednání o tématu disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Tomáš Hruška, CSc.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 - 4.

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **John Petr, Ing.**  
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.  
Datum zadání: 1.11.2024  
Termín pro odevzdání: 14.5.2025  
Datum schválení: 22.10.2024

## Abstrakt

Práca sa zaoberá spracovaním vizuálneho programovacieho jazyka (VPL) na strane servera v kontexte Internetu vecí (IoT). Cieľom je umožniť koncovým používateľom – aj bez znalostí tradičných programovacích jazykov – konfigurovať a automatizovať správanie IoT zariadení prostredníctvom jednoduchého vizuálneho rozhrania. Práca analyzuje existujúce prístupy a jazyky vhodné pre vizuálne programovanie v oblasti IoT, pričom sa zameriava na koncept end-user development (EUD). Výsledkom je návrh a implementácia serverového interpreta napísaného v jazyku Go, ktorý zabezpečuje validáciu a vyhodnocovanie vizuálnych programov. Riešenie bolo integrované do existujúceho systému RIoT, čím sa rozšírili možnosti v oblasti riadenia IoT zariadení. V práci sú taktiež uvedené možnosti ďalšieho rozšírenia a zhodnotenie zvoleného prístupu.

## Abstract

This bachelor's thesis focuses on server-side processing of a visual programming language (VPL) in the context of the Internet of Things (IoT). The aim is to enable end users, even without programming knowledge, to configure and automate the behavior of IoT devices using a simple visual interface. The thesis explores existing approaches and visual programming languages suitable for IoT, with emphasis on the concept of end-user development (EUD). The outcome is the design and implementation of a backend interpreter written in Go, which validates and evaluates programs. The solution has been integrated into the existing RIoT system, extending its capabilities in managing IoT devices. The thesis also presents potential future improvements and an evaluation of the chosen approach.

## Klíčové slová

Internet vecí, IoT, vizuálne programovanie, end-user development, Go, RIoT

## Keywords

Internet of Things, IoT, visual programming, end-user development, Go, RIoT

## Citácia

TÁBI, Matúš. *Zpracování vizuálního programovacího jazyka na straně serveru*. Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Petr John

# Zpracování vizuálního programovacího jazyka na straně serveru

## Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Petra Johna. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....  
Matúš Tábi  
13. mája 2025

## Podakovanie

Velké podakovanie patří predovšetkým vedúcemu bakalárskej práce p. Ing. Petrovi Johnovi za jeho ochotu, čas, pomoc, usmernenia a cenné rady pri vypracovaní tejto práce, a svojej rodine a priateľom za podporu.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>3</b>
<b>2</b>	<b>Internet vecí</b>	<b>5</b>
2.1	Architektúra IoT . . . . .	6
2.2	IoT technológie . . . . .	7
2.3	Aplikácie a využitie IoT . . . . .	10
2.4	Smart cities . . . . .	10
<b>3</b>	<b>Vizuálne programovanie</b>	<b>12</b>
3.1	Vývoj softvéru koncovým užívateľom . . . . .	12
3.2	Nástroje vizuálnych programovacích jazykov . . . . .	13
3.3	Trigger action programming a parametrizácia . . . . .	20
<b>4</b>	<b>Analýza</b>	<b>22</b>
4.1	Vizuálny programovací jazyk . . . . .	22
4.2	Pôvodný interpret . . . . .	26
4.3	RIoT . . . . .	29
<b>5</b>	<b>Návrh</b>	<b>31</b>
5.1	Štruktúra interpreta . . . . .	31
5.2	Návrh integrácie do systému . . . . .	33
5.3	Databáza a GraphQL API . . . . .	34
<b>6</b>	<b>Implementácia</b>	<b>36</b>
6.1	Implementácia interpreta . . . . .	36
6.2	Integrácia do systému RIoT . . . . .	42
<b>7</b>	<b>Testovanie</b>	<b>46</b>
7.1	Navrhované rozšírenie . . . . .	47
<b>8</b>	<b>Záver</b>	<b>48</b>
	<b>Literatúra</b>	<b>49</b>
<b>A</b>	<b>Obsah externej prílohy</b>	<b>55</b>

# Zoznam obrázkov

2.1	Základná trojvrstvová architektúra IoT systémov. Zahŕňa snímáciu vrstvu na zber dát, sieťovú vrstvu na prenos dát a zabezpečenie spojenia a aplikačnú vrstvu na prezentáciu a vizualizáciu dát. . . . .	7
2.2	Rozdelenie bezdrôtových technológií na krátke a veľké vzdialenosti . . . . .	7
3.1	Hra vyvinutá v jazyku Scratch . . . . .	14
3.2	Príklad jednoduchého programu v Blockly . . . . .	15
3.3	Príklad programu popisujúci automatické zvonenie budíkov na základe nastaveného času [46] . . . . .	15
3.4	Jednoduchý príklad webového servera na tvorbu grafov s Fitbit údajmi <sup>1</sup> . . . . .	16
3.5	Uzlové nastavenie v Blenderi na tvorbu procedurálneho materiálu . . . . .	17
3.6	Vytvorenie jednoduchej služby v programe EUCalipTool [52] . . . . .	18
3.7	Ikony používané v aplikácii MicroApp a jednoduchý príklad [19] . . . . .	19
3.8	Príklad <i>applet</i> -u v platforme IFTTT <sup>2</sup> . . . . .	20
4.1	Model štruktúr jazyka a ich prepojenie . . . . .	28
5.1	Zjednodušený model interpreta . . . . .	32
5.2	Návrh architektúry riešenia . . . . .	34
5.3	Navrhnutý ER diagram . . . . .	35
6.1	Diagram spracovania hlavičky programu . . . . .	38
6.2	Diagram spracovania tela programu . . . . .	39
6.3	Diagram logiky spracovania podmieneného príkazu . . . . .	40
6.4	Diagram spracovania stromu argumentov a vyhodnotenie . . . . .	41
6.5	ER diagram popisujúci databázové objekty a vzťahy . . . . .	43
6.6	Tok správ RPC mechanizmu pri ukladaní programu . . . . .	44
7.1	Diagram posielania správ pri zmene hodnoty parametra . . . . .	47

# Kapitola 1

## Úvod

Vďaka rýchlemu rozvoju internetu a bezdrôtových technológií môžu dnes elektronické zariadenia komunikovať nielen s používateľmi, ale aj medzi sebou vzájomne. Tento koncept nazývame Internet vecí (anglicky *Internet of Things*, skrátene *IoT*). Zariadenia obsahujú rôzne senzory a riadiace jednotky, pomocou ktorých sú zariadenia schopné si vymieňať a posielat informácie a dáta, čím sa zvyšuje efektívnosť a automatizácia procesov. Jedným z najvýznamnejších príkladov v praxi sú takzvané inteligentné mestá (*Smart Cities*), ktoré využívajú tieto zariadenia na optimalizáciu verejných služieb, dopravy a znižovanie energetickej spotreby.

Zariadenia internetu vecí nachádzajú uplatnenia v mnohých odvetviach. V priemysle sa používajú na monitorovanie výrobných procesov, optimalizáciu logistiky, čím sa prispieva k vyššej efektívnosti a znižovaniu nákladov. V zdravotníctve vedú inteligentné zariadenia pomáhať v meraní cukru v krvi, tlaku alebo teploty. V domácnostiach sú bežné inteligentné termostaty, osvetlenie, zámky na dverách alebo samozavlažovací systém v záhrade. Zariadenia využívajú rôzne technológie na komunikáciu a tak vznikajú systémy, ktoré zaisťujú konfiguráciu, správu a vzájomné prepojenie.

V dnešnej dobe IoT ekosystém trpí veľkým množstvom rôznych aplikácií. Každé zariadenie alebo jeho výrobca ponúka vlastný softvér. Používatelia tak môžu skončiť s viacerými aplikáciami na rôzne zariadenia, čo môže skomplikovať prehľadnosť a aj integráciu. Riešením môže byť jeden softvér alebo platforma, ktorá dokáže spojiť a spravovať zariadenia na jednom mieste bez ohľadu na výrobcu. Tento systém vďaka jednotnému užívateľskému rozhraniu zvyšuje prehľadnosť, jednoduchšiu konfiguráciu a interakciu so zariadeniami.

V súvislosti s týmto sa do popredia dostáva koncept *End-User Development* (EUD), teda vývoj koncovými používateľmi. Tento koncept umožňuje bežným používateľom prispôbiť softvérové riešenia svojim potrebám bez toho, aby mali znalosti v programovaní alebo v informačných technológiách. Tým, že EUD umožňuje vytvárať vlastné aplikácie a riešenia, je tento koncept obzvlášť dôležitý v oblasti IoT. Používatelia si môžu sami vytvoriť automatizačné procesy na správu svojich zariadení v domácnosti podľa potreby.

Jednou z techník, ktoré EUD podporuje, sú vizuálne programovacie jazyky (VPL – *Visual Programming Languages*). Tieto jazyky poskytujú používateľom jednoduché a intuitívne prostredie, kde si môžu vytvárať vlastné programy a procesy prostredníctvom skladaním grafických prvkov namiesto písania zložitejšieho textového kódu. VPL výrazne uľahčuje proces vytvárania aplikácií, keďže koncoví užívatelia sa nemusia učiť syntax a sémantiku textových programovacích jazykov, to im umožňuje sústrediť sa na logiku a funkcionálnosť. Integrácia EUD a VPL do sveta IoT otvára mnoho možností pre bežných používateľov, ktorí si môžu svoje zariadenia konfigurovať podľa svojich predstáv. Cieľom práce je naim-

plementovať vyhodnocovací program, ktorý je schopný validovať a vyhodnotiť daný program vytvorený užívateľom a integrovať toto riešenie do existujúceho systému.

Teoretická časť tejto práce je rozdelená na dve kapitoly, kapitola 2 popisuje definíciu internetu vecí, architektúru a používané technológie. Kapitola 3 popisuje vizuálne programovanie ako celok, vývoj koncovým používateľom a existujúce vizuálne programovacie jazyky. Praktickou časťou práce sa zaoberajú zvyšné kapitoly. Kapitola 4 analyzuje serverové časti systému spolu s analýzou spracovávaného programovacieho jazyka a pôvodného interpreta. Návrh v kapitole 5 sa zaoberá konkrétnymi časťami, ktoré bude treba naimplementovať pre správnu funkcionálnu vyhodnocovania programov.

## Kapitola 2

# Internet vecí

Internet vecí (*Internet of Things, IoT*) je definovaný Európskou úniou [17] ako distribuovaná sieť spájajúca fyzické objekty, ktoré sú schopné snímať alebo pôsobiť na svoje okolie a komunikovať navzájom, s inými strojmi alebo počítačmi. Americká medzinárodná technologická spoločnosť IBM definuje IoT [26] ako sieť fyzických zariadení, dopravných prostriedkov, spotrebičov a iných objektov, ktoré sú vybavené senzormi, softvérom a sieťovým pripojením, čo im umožňuje zhromažďovať a zdieľať údaje. Inteligentné objekty môžu siahať od jednoduchých zariadení „inteligentnej domácnosti“, ako sú inteligentné termostaty, cez nositeľné zariadenia, ako sú inteligentné hodinky a oblečenie s technológiou RFID, až po zložité priemyselné stroje a dopravné systémy. Online portál Statista [48], ktorý sa zaoberá zbieraním a vizualizáciou dát, píše, že počet IoT zariadení sa celosvetovo v roku 2023 pohyboval približne okolo 15.9 miliárd.

Internet vecí je veľmi dôležitá oblasť, ktorá dokáže vysoko zefektívniť, optimalizovať a zautomatizovať rôzne procesy. Modely a rozhodnutia môžu byť uskutočnené vďaka veľkému množstvu zozbieraných dát zo zariadení. Analýzou týchto dát sa dajú získať napríklad prehľady o správaní zákazníkov, prehľady o trendoch, čo umožňuje robiť lepšie rozhodnutia v stratégiách alebo vo vývoji. Keďže IoT zariadenia dokážu zjednodušiť a zredukovať manuálne a opakujúce sa práce, umožňuje to zároveň aj zredukovanie nákladov. Okrem mnohých pozitívnych vplyvov internetu vecí na životy ľudí sa vyskytujú aj situácie, ktoré z bezpečnostného hľadiska vzbudzujú obavy. Je zjavné, že toto odvetvie sa stalo centrom hackerských útokov, škodlivého softvéru, kybernetických útokov, špehovania a mnohých ďalších problémov. Európska únia sa preto rozhodla vytvoriť Akt o kybernetickej bezpečnosti<sup>1</sup> (*CRA – The Cyber Resilience Act*), ktorého cieľom je určiť povinnosti v oblasti kybernetickej bezpečnosti všetkým výrobkom s digitálnymi prvkami, ktoré zahŕňajú dátové pripojenie k zariadeniam alebo internetovej sieti.

Aby zariadenia mohli efektívne plniť svoje úlohy, musia byť navrhnuté pre rôzne funkcie, ktoré vykonávajú, a na základe tohto sa rozdeľujú do niekoľkých typov. Autori v práci [47] popisujú prvým typom senzory, ktoré merajú a spracúvajú zozbierané dáta z prostredia, napríklad teplotu, vlhkosť, pohyb alebo svetlo. Vždy, keď dôjde k zmene nejakého fyzikálneho javu, ktorý daný senzor meria, vytvorí odozvu pre ďalšie procesy v rámci IoT systémov, ktoré následne túto odozvu ďalej spracúvajú. Ďalším typom sú aktory, ktoré majú za úlohu vykonávať fyzické akcie na základe pokynov, ako napríklad zapínanie svetiel alebo otváranie

---

<sup>1</sup>CRA predstavuje návrh legislatívy Európskej únie, ktorý zavádza povinnosti kybernetickej bezpečnosti s cieľom zlepšiť odolnosť voči kybernetickým hrozbám. <https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act>

a zatváranie ventilov. Tretím typom sú riadiace jednotky, ktoré môžeme označovať aj ako centrálné uzly. Tie majú za úlohu spracovávať prijaté dáta a rozhodovať o ďalších krokoch.

Každé inteligentné zariadenie obsahuje jednoznačné identifikačné číslo (*UID*) a IP adresu. Na to, aby tieto zariadenia medzi sebou efektívne komunikovali, je potrebné vytvoriť siete rôznych úrovní. Na pripojenie zariadení na fyzickej linkovej vrstve do týchto sietí sa používajú technológie ako Bluetooth alebo Wi-Fi, na veľmi krátke vzdialenosti technológie identifikácie na rádiovkej frekvencii (RFID – *Radio-Frequency Identification*) alebo NFC (*Near Field Communication*) [23]. Na samostatnú komunikáciu medzi zariadeniami sa používajú špecializované protokoly, napríklad MQTT (*Message Queue Telemetry Transport*), HTTP (*Hypertext Transfer Protocol*) alebo CoAP (*Constrained Application Protocol*), ktoré definujú, akým spôsobom sa prenášajú dáta medzi zariadeniami a ako sú formátované.

## 2.1 Architektúra IoT

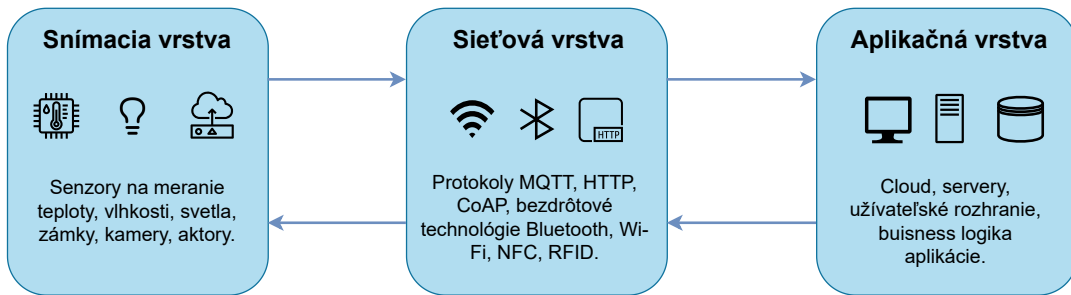
Služby a systémy IoT sa na rozdiel od služieb a systémov internetu líšia tým, že nemajú dostatok štandardizovaných protokolov a technológií [5]. Hlavným dôvodom je, že IoT aplikácie sú veľmi rôznorodé, s rozdielnymi potrebami a charakteristikami. Preto časom vzniklo mnoho architektúr, najzákladnejšia je trojvrstvová architektúra, od ktorej sa odvíja mnoho ďalších architektúr [3]. Na obrázku 2.1 je štruktúra architektúry spolu s príkladmi ku konkrétnej vrstve. Pozostáva z troch hlavných vrstiev:

- **Snímacia vrstva** – v IoT systémoch tiež známa aj ako fyzická vrstva. V tejto vrstve sa nachádzajú senzory, aktory a vstavané systémy, ktoré zbierajú dáta, komunikujú s okolím a reagujú na ostatné okolité zariadenia. Zozbierané dáta a príkazy sú ďalej poslané sieťovej vrstve na ďalšie spracovanie.
- **Sieťová vrstva** – hlavnou úlohou je prepojiť snímáciu a aplikačnú vrstvu. Má na starosti vytváranie spojení medzi ostatnými inteligentnými zariadeniami, prípadne cloudom pomocou rôznych protokolov a bezdrôtových technológií.
- **Aplikačná vrstva** – stará sa o spracovanie, uloženie a zobrazenie dát. Spadá sem business logika aplikácie, spôsob ukladanie dát a podľa potreby posielanie týchto dát späť do snímacej vrstvy.

Najzákladnejší koncept IoT systémov je popísaný trojvrstvou architektúrou, postupným výskumom je ale táto architektúra nedostatočná, preto mnohé literatúry uvádzajú detailnejšiu päťvrstvomú architektúru [3, 5]. Zvyšné dve vrstvy sú popísané nasledovne:

- **Vrstva spracovania** – funguje ako sprostredkovateľ medzi snímacou, sieťovou a aplikačnou vrstvou. Jej úlohou je zozbierať a spracovať dáta, spravovať ukladanie dát do databáz, prípadne vytvárať rozsiahle výpočty nad dátami.
- **Business vrstva** – má na starosti logiku a pravidlá, ktoré ovplyvňujú chod celého IoT systému, generovanie výsledkov a rozhodnutí. Aplikačná vrstva má na starosti potom prezentovanie dát a poskytuje rozhranie užívateľom.

Existuje ešte niekoľko ďalších architektúr, ktoré rozširujú základnú trojvrstvomú architektúru. Ako vo svojej téze uvádza John P. [30], jednou z nich je SoA (*Service-Oriented Architecture*), ktorá je rozšírená o vrstvu služieb. Tá je zodpovedná za spravovanie služieb, ktoré sú vyžadované od užívateľov alebo aplikácií, ako napríklad spracovanie a ukladanie dát, zabezpečovanie pripojenia medzi inými zariadeniami a aplikáciami.



Obr. 2.1: Základná trojvrstvová architektúra IoT systémov. Zahŕňa snímaciu vrstvu na zber dát, sieťovú vrstvu na prenos dát a zabezpečenie spojenia a aplikačnú vrstvu na prezentáciu a vizualizáciu dát.

## 2.2 IoT technológie

Internet vecí používa rozličné technológie na prenos dát, na pripojenie zariadení k internetu a medzi ostatnými zariadeniami. Medzi najhlavnejšie patria bezdrôtové technológie, ktoré zabezpečujú prenos dát na krátke a stredné vzdialenosti (*Bluetooth*, *Wi-Fi*) alebo aj na väčšie vzdialenosti pomocou technológie LPWAN, viz 2.2. Na to, aby zariadenia medzi sebou dokázali komunikovať a vymieňať si dáta, existujú rôzne sieťové a komunikačné protokoly, ktoré definujú pravidlá a štandardy komunikácie. Nižšie je prehľad dôležitých technológií a protokolov použitých v IoT.



Obr. 2.2: Rozdelenie bezdrôtových technológií na krátke a veľké vzdialenosti

### Bezdrôtové technológie

Podľa definície na stránke Intel [11], **Bluetooth** je definovaný ako „technológia umožňujúca bezdrôtové pripojenie dvoch zariadení bez potreby podpornej sieťovej infraštruktúry, ako napríklad router“. V praxi sa používa napríklad na prepojenie inteligentných hodínok s mobilným telefónom alebo na pripojenie bezdrôtových sluchadiel k počítaču. Bluetooth pracuje na rádiových frekvenciách v rozsahu 2.4 GHz, v súčasnosti existujú 2 druhy štandardov, a to Bluetooth Classic, ktorý podporuje dve rôzne rýchlosti prenosu dát (*Basic Rate* a *Enhanced Data Rate*). Druhým štandardom je Bluetooth Low Energy, ktorý je optimalizovaný na nízku spotrebu energie.

**Wi-Fi** je najpoužívanejšia technológia na pripojenie k internetu. Funguje na štandarde IEEE 802.11b [24]. Ten definuje protokoly, ktoré umožňujú komunikáciu so zariadeniami

vrátane bezdrôtových smerovačov a bezdrôtových prístupných bodov [55]. Funguje na princípe vysielania rádiových vln z routera k zariadenia ako napríklad TV, mobilné telefóny, tablety alebo počítače.

**RFID** alebo **rádiofrekvenčná identifikácia** (*Radio-frequency identification*) je systém automatickej identifikácie pomocou rádiových vln, ktorý spočíva v ukladaní a načítaní dát s malým mikročipom nazývaným RFID tag a čítačkou. [28, 29]. RFID používa aktívne tagy s pripojeným zdrojom napätia (batérie) a pasívne tagy, ktoré nevyžadujú pripojený zdroj energie alebo batérie [23]. Dosah komunikácie závisí od použitého frekvenčného pásma. Podľa [31] majú nízkofrekvenčné tagy (124 – 135 kHz) dosah do pol metra, vysokofrekvenčné tagy (13.56 MHz) niečo vyššie metra a ultravysokofrekvenčné (860 – 960 MHz) až desiatky metrov. Aktívne tagy, pri ktorých sú prenosy napájané batériou, môžu mať dosah až vyššie 100 metrov. RFID je použité napríklad v skladoch na získanie detailov o tovaroch, v zdravotníctve na sledovanie chirurgických pomôcok a pacientov, na identifikáciu osôb pomocou elektronických pasov alebo sledovanie počtu osôb na podujatiach a festivaloch [44].

**NFC** (*Near-field communication*) je bezdrôtová technológia na veľmi krátky dosah (zo pár centimetrov), ktorá umožňuje komunikáciu a prenos dát medzi zariadeniami. Používa podobný princíp ako RFID, nepoužíva sa ale len na identifikáciu, ale aj na obojsmernú komunikáciu [6]. Pracuje na vysokofrekvenčnom pásme a podporuje maximálnu rýchlosť prenosu dát do 424 kb/s. Protokol podporuje komunikáciu nielen medzi aktívnou čítačkou a pasívnym tagom, ale aj medzi dvoma čítačkami (peer-to-peer) [54]. NFC sa najčastejšie používa pri bezkontaktných platbách, overovanie totožnosti alebo kupovanie lístkov vo verejnej doprave.

**LPWAN** (*Low-power wide-area network*) je typ rozsiahlej siete, ktorá umožňuje komunikáciu na veľké vzdialenosti s nízkou prenosovou rýchlosťou. Príkladom rozsiahlej siete je napríklad LoRa technológia. Na prenos správ sa používa tradičná rádiová komunikácia s otvorenou frekvenciou. Všetky správy sú posielané cez *broadcast*, čiže správy z jedného snímača môže prijímať viacero brán [30]. Je to fyzická vrstva, ktorá je štandardizovaná a rozšírená o MAC (*Medium access control*) LoRaWAN vrstvu [16], ktorá definuje komunikáciu medzi zariadeniami a bránami.

## Sieťové protokoly

**MQTT** (*Message Queue Telemetry Transport*) je protokol zasielania správ, ktorý predstavili v roku 1999 Andy Stanford-Clark z IBM a Arlen Nipper z Arcom-u [4]. MQTT protokol sa skladá z troch hlavných komponentov. Ako uvádza [35], klientom môže byť vydavateľ/odberateľ (*publisher/subscriber*), ktorého úlohou je otvorenie a ukončenie sieťového spojenia so serverom alebo vytvorenie a publikovanie správ a ich prijatie. Druhým komponentom je MQTT broker, ktorý je zodpovedný za prijímanie sieťových spojení od klientov a za prijímanie správ publikovaných klientami.

Kvalita služieb (*QoS – Quality of Service*) je definovaná firmou Fortinet<sup>2</sup> [18] ako „používanie mechanizmov alebo technológií, ktoré fungujú v sieti s cieľom riadiť prevádzku a zabezpečiť výkon kritických aplikácií s obmedzenou kapacitou siete. Umožňuje upraviť celkovú sieťovú prevádzku uprednostnením konkrétnych aplikácií s vysokým výkonom.“ MQTT protokol ponúka tri kvality služieb:

- **QoS 0** (at most once) – v tejto kvalite služby sa správy doručujú najviac raz, ponúka mechanizmus najlepšieho doručenia, pri ktorom odosielateľ neočakáva potvr-

<sup>2</sup>Fortinet je americká firma zaoberajúca sa kybernetickou bezpečnosťou: <https://www.fortinet.com/>

denie alebo záruku doručenia správy [36]. Príkladom môže byť posielanie údajov zo snímačov tlaku alebo vlhkosti do aplikácie, kde nevadí, ak sa spojenie s aplikáciou čítajúcou údaje zo snímača na chvíľu stratí [35].

- **QoS 1** (at least once) – správy sú doručované aspoň jedenkrát, s možnosťou duplicity. Pri posielaní správy si odosielateľ uchováva kópiu správy, kým nedostane od príjemcu PUBACK paket, ktorý potvrdzuje prijatie. Ak odosielateľ nedostane PUBACK paket, odošle správu znova, aby bolo zabezpečené doručenie [36].
- **QoS 2** (exactly once) – zaručuje, že správy budú doručené presne raz. Na dosiahnutie tohto sa používa štvorfázová výmena správ medzi odosielateľom a príjemcom [36]. Táto služba je užitočná napríklad na upozornenie, keď dôjde k neobvyklému odpojeniu [35].

**HTTP** (*Hyper Text Transport Protocol*) je prevažne webový protokol na zasielanie správ cez internet, ktorý pôvodne vyvinul Tim Berners-Lee, neskôr bol vyvíjaný organizáciami IETF a W3C a publikovaný bol v roku 1997 [38]. Podobne ako v protokole CoAP, HTTP protokol používa URI, kde server posieľa správy prostredníctvom URI a klient prijíma cez konkrétne URI. HTTP protokol definuje metódy pre rôzne typy operácií:

- **GET** – získanie dát zo servera,
- **POST** – posielanie dát na spracovanie na server,
- **PATCH, PUT** – aktualizácia existujúcich dát, PUT sa používa na úplnú aktualizáciu, PATCH na čiastočnú,
- **DELETE** – odstraňovanie dát.

Je to textový protokol, ktorý používa TCP<sup>3</sup> ako predvolený transportný protokol, na zabezpečenie používa protokol TLS/SSL<sup>4</sup> a explicitne nedefinuje žiadne kvality služieb [38].

**CoAP** (*Constrained Application Protocol*) je protokol na aplikačnej vrstve vytvorený skupinou IETF Constrained RESTful Enviroments (CoRE), ktorý bol založený na REpresentational State Transfer (REST) nad funkciami HTTP [4]. CoAP protokol používa URI (*Universal Resource Identifier*) na identifikáciu zdrojov. Komunikácia funguje tak, že vydavateľ (publisher) zverejňuje dáta na URI a odberateľ (subscriber) sa prihlasuje na odber konkrétneho zdroja označeného URI. Ak vydavateľ zverejní nové dáta, všetci odberatelia sú informovaní [38]. Ako píše N. Naik vo svojej práci [38], namiesto troch úrovní kvality služieb v protokole MQTT, CoAP disponuje dvomi úrovňami a to *confirmable messages* a *non-confirmable messages*. V prvom prípade je vyžadované, aby prijímateľ potvrdil správu ACK paketom, v druhom prípade nie je potrebný.

**AMQP** (*Advanced Message Queuing Protocol*) je štandardný protokol aplikačnej vrstvy používaný pri internete vecí. Zameriava sa primárne na také prostredia, ktoré sú orientované na správy [4]. Hlavným účelom AMQP je umožniť aplikáciám vymieňať si správy a to nezávisle na implementácii, platforme alebo jazyku. Ponúka širokú škálu funkcií súvisiacich so zasielaním správ, ako je napríklad spoľahlivé radenie do front, publikovanie a odber správ na základe tém, flexibilné smerovanie a transakcie [38]. Vďaka tomuto je mimoriadne vhodný pre zložitejšie komunikačné scenáre.

---

<sup>3</sup>RFC 9293: Transmission Control Protocol. Dostupné na: <https://datatracker.ietf.org/doc/html/rfc9293>

<sup>4</sup>RFC 5246: The Transport Layer Security Protocol. Dostupné na: <https://datatracker.ietf.org/doc/html/rfc5246>

## 2.3 Aplikácie a využitie IoT

Počet potencionálnych aplikácií je obrovský. Počet pripojených zariadení v dnešnej dobe dosahuje niekoľko miliárd a toto číslo bude stále rapídne stúpať. Tento koncept sa stáva neoddeliteľnou súčasťou nášho života a transformuje ho do takého spôsobu, akým spoločnosť pristupuje k bežným úlohám v domácnostiach, verejnej správe alebo na pracoviskách. IoT zariadenia spolu s technológiami, ktoré dokážu monitorovať, zbierať a vymieňať si dáta, prinášajú inovácie takmer do každej oblasti.

Jednou z najznámejších aplikácií pre bežných spotrebiteľov sú inteligentné domácnosti. Slovné spojenie inteligentná domácnosť označuje takú domácnosť, kde je možné spotrebiče a zariadenia ovládať diaľkovo cez internet. Zariadenia inteligentnej domácnosti sú vzájomne prepojené a je možné k nim pristupovať z jedného centrálného bodu, zvyčajne mobilného telefónu alebo počítača, poprípade pomocou hlasových asistentov. Telefón teda môže slúžiť ako diaľkové ovládanie zariadení domáceho kina, termostaty alebo inteligentných spotrebičov [23]. Príkladom systému s hlasovým asistentom je *Google Home Assistant*<sup>5</sup>, kde prostredníctvom hlasu dokážu užívatelia vykonávať mnoho úloh (ovládanie osvetlenia, klimatizácie alebo kúrenia, ovládanie multimédií). Bežne používané zariadenia sú inteligentné zabezpečovacie systémy, ktoré zahŕňajú kamery, senzory pohybu alebo inteligentné zámky. Čoraz populárnejšie sa stávajú inteligentné zásuvky a vypínače, ktoré monitorujú spotrebu elektrickej energie.

V oblasti zdravotníctva prináša oblasť IoT revolučné zmeny. Pomocou nositeľných zariadení (wearables), akými sú napríklad hodinky alebo náramky, môžu sledovať vitálne funkcie, ako sú tep, tlak alebo hladina kyslíka v krvi. Ako sa uvádza v [27], IoT zariadenia sa používajú najmä na diaľkové sledovanie pacientov, monitorovanie nálady, monitorovanie Parkinsonovej choroby alebo operácie za pomoci robotov pripojených na internet.

Podľa [56] internet vecí bude zohrávať čoraz dôležitejšiu úlohu v odvetví logistiky a dopravy. Keďže čoraz viac fyzických predmetov je vybavených čiarovými kódmi, RFID tagmi alebo senzormi, spoločnosti môžu monitorovať pohyb fyzických predmetov v reálnom čase. Vozidlá majú čoraz výkonnejšie možnosti snímania, komunikácie a spracovania dát, a teda tieto možnosti môžu byť využité napríklad na monitorovanie počtu miest na parkovisku alebo na ceste, sledovať pohyb vozidiel alebo predikovať jeho budúcu polohu. Zdroj [56] ďalej uvádza aj využitie IoT systémov v baniach na bezpečnejšiu ťažbu s pomocou RFID, Wi-Fi a iných bezdrôtových technológií alebo na využitie internetu vecí pri hasení požiarov.

## 2.4 Smart cities

Označenie „inteligentné mestá“ sa používa v poslednej dobe čoraz častejšie a to predovšetkým na označenie využívania technických zariadení a prístupov v mestskom prostredí s cieľom maximalizovať kvalitu života obyvateľov [49]. Podľa Kantera a Litowa [32] na to, aby sa mesto stalo inteligentným, nestačí, aby sa inteligencia dostala postupne do každého odvetvia, inteligentné mesto by sa malo vnímať ako celok, ako sieť, prepojený systém. Podľa zdroja [7] začlenením senzorov, umelej inteligencie, strojového učenia alebo *Big Data* (dátové súbory príliš veľké na spracovanie bežnými metódami) vznikajú možnosti v navrhovaní a riadení miest, zvýšení efektivity správy mesta alebo znížení environmentálnej záťaže. Tieto technológie navyše umožňujú trénovať počítače na napodobňovanie vzorcov myslenia alebo vykonávanie simulácií ľudského myslenia. Tento koncept rieši problémy dnešnej urbanizá-

<sup>5</sup>Domovská stránka Google Home Assistant: <https://assistant.google.com/>

cie, ako sú napríklad zvyšujúca sa hustota obyvateľstva, znečisťovanie, rastúce energetické nároky alebo problémy v doprave [8].

Dnešné inteligentné mestá sú efektívnejšie a inteligentnejšie vďaka pokroku v oblasti informačných a komunikačných technológií (ICT<sup>6</sup> – *Information and Communications Technology*). Tieto technológie pomáhajú inteligentným mestám poskytovať lepšie služby, znižovať náklady na prevádzku a údržbu [8]. Efektívnym používaním ICT sa mestá stávajú prepojenými, vďaka schopnosti komunikovať a interagovať so širokou skupinou zainteresovaných strán; inteligentnými, vďaka schopnosti analyzovať situácie a rýchlo reagovať na ich riešenie; a nástrojovými, vďaka schopnosti snímať, detegovať, merať a zaznamenávať rôzne dáta [20].

Mnohí výskumníci potvrdzujú, že spracovanie a interpretácia údajov je základným krokom k obohateniu mestskej štruktúry [9, 10, 7]. Spracovávanie údajov zozbieraných z IoT zariadení prostredníctvom umelej inteligencie môže zabezpečiť vhodnejší život prostredníctvom čistoty, zdravia a priaznivého prostredia pre život a prácu bez typických mestských problémov. Okrem toho sa predpokladá, že veľké zozbierané objemy dát môžu prilákať vyššie hospodárske výnosy [7]. Dostupnosť rôznych typov údajov zozbieraných prostredníctvom všadeprítomného mestského internetu vecí sa môže využiť aj na zvýšenie transparentnosti a podporu činnosti miestnej samosprávy voči občanom, zlepšenie informovanosti ľudí o stave ich mesta alebo stimulovanie aktívnej účasti občanov na riadení verejnej správy [58].

Autori v článku [58] píšú o niekoľkých službách, ktoré by mohli priniesť obojstranne výhodnú situáciu, a to zvýšenie kvality služieb ponúkaných občanom a zároveň ekonomickú výhodu pre mestskú správu. Príkladom môže byť správna údržba historických budov mesta, čo vyžaduje neustále monitorovanie skutočného stavu. Riešením by mohla byť databáza, kde by sa pomocou vhodných senzorov umiestnených v budovách (senzory vibrácií, deformácií, teploty alebo vlhkosti) alebo pomocou monitorovania úrovne znečistenia zbierali dáta, čo by malo znížiť potrebu periodického testovania konštrukcie ľuďmi. Veľkým problémom v mnohých mestách je nakladanie s odpadom a to najmä kvôli nákladom na služby, ale aj kvôli problému so skladovaním. Riešením takéhoto problému môžu byť inteligentné kontajnery na odpad, ktoré zisťujú úroveň naplnenia a umožňujú optimalizáciu trasy zberných vozidiel.

Hluk môže byť považovaný za formu akustického znečistenia rovnako ako oxid uhličitý. Zariadenia IoT v mestách môžu ponúkať služby monitorovania hluku v danej hodine na určitých miestach. Takáto služba môže zvýšiť verejnú bezpečnosť, napríklad pomocou algoritmov, ktoré dokážu rozoznať zvuk rozbitého skla pri krádeži alebo šarvátkach. Služby inteligentných miest dokážu sledovať zápchy a dopravnú situáciu v mestách. V dnešnej dobe sú už síce systémy monitorovania dopravy pomocou kamier v mnohých mestách, rozsiahla komunikácia s nízkou spotrebou energie môže poskytnúť hustejší zdroj informácií, napríklad s pomocou GPS v moderných vozidlách. Zároveň tento systém dokáže pomôcť s parkovacími službami v meste a navigovať vodičov po najlepšej ceste na parkovanie v meste.

Zo spomínaných služieb je zjavné, že väčšina služieb inteligentných miest je založená na periférnych zariadeniach rozmiestnených v rozličných mestských oblastiach, ktoré generujú rôzne typy dát. Hlavnou charakteristikou mestskej infraštruktúry internetu vecí je preto jej schopnosť integrovať rôzne technológie s existujúcimi komunikačnými infraštruktúrami s cieľom podporiť postupný vývoj internetu vecí [58]. Prístup k zozbieraným dátam môže byť dosiahnutý hlavne webovými službami, čo sa javí ako sľubný a flexibilný prístup, ale aj mobilnými zariadeniami, tabletmi alebo počítačmi a pomocou technológií na spojenie so zariadeniami a transferom dát [58], popísanými v 2.2.

<sup>6</sup>ICT umožňujú efektívne prepojenie dát v inteligentných mestách. <https://aims.fao.org/information-and-communication-technologies-ict>

## Kapitola 3

# Vizuálne programovanie

Čoraz viac softvérových aplikácií píšú koncoví používatelia bez formálneho vzdelania v oblasti vývoja softvéru [33]. Veľkú časť softvéru nevyvíjajú softvéroví inžinieri, ale skôr pracovníci, technici alebo bežní ľudia s malým alebo žiadnym vzdelaním v oblasti softvérového inžinierstva [53]. Tento posun vo vývoji softvéru sa prejavuje v oblasti zvanej *end-user development* (EUD), ktorý umožňuje bežným používateľom vytvárať a prispôbovať softvér podľa svojich potrieb.

Vizuálny programovací jazyk (VPL – *visual programming language*), jednou z techník v EUD, umožňuje koncovým užívateľom vytvárať program skladaním grafických prvkov namiesto textovej špecifikácie [33]. Tento prístup uľahčuje pochopenie konceptu programovania, keďže poskytuje vyššiu vrstvu abstrakcie, nie je nutné sa sústreďiť na syntax a sémantiku, prípadne zložitejšie konštrukcie, ako pri klasických programovacích jazykoch [41]. Pôvodne teda vizuálne programovacie jazyky mali pomáhať začiatočníkom pri učení sa základov programovania, vizuálne programovacie jazyky sa ale začali používať aj v iných oblastiach, napríklad IoT, vývoj mobilných aplikácií, robotika alebo virtuálna realita [33, 42].

### 3.1 Vývoj softvéru koncovým užívateľom

Narastajúci dopyt po integrovaných aplikáciách často naráža na bariéry, ako je znalosť programovacích jazykov pre webové aplikácie, napríklad PHP alebo JavaScript, alebo porozumenie webovým protokolom, akým je HTTP. Tieto prekážky obmedzujú vytváranie a rozširovanie aplikácií. Preto je potreba používateľom poskytnúť nástroje a možnosti, ktoré im umožnia tieto aplikácie vytvárať. End-User Development (EUD), teda vývoj koncovými používateľmi predstavuje prístup, ktorý umožňuje bežným ľuďom bez vzdelania v oblasti programovania a informačných technológií vytvárať softvérové aplikácie. Okrem vizuálneho programovania k tomuto odvetviu patria aj ďalšie dva koncepty [33], a to End-User Programming (EUP) a End-User Software Engineering (EUSE). EUP sa zaoberá samostatným procesom programovania s konkrétnymi technikami a nástrojmi, EUSE rieši celkový návrh softvéru, udržateľnosť a kvalitu kódu [41].

Hlavným cieľom EUD je znížiť bariéry pri vývoji softvéru tým, že poskytuje intuitívne nástroje a vizuálne rozhrania. Vďaka tomu si môžu koncoví používatelia prispôbiť aplikácie svojim špecifickým potrebám, poprípade vytvárať nové riešenia bez nutnosti znalosti typických programovacích jazykov. Napríklad pracovník administratívy si môže vytvoriť vlastnú databázovú aplikáciu na správu úloh (napríklad v programe *Microsoft Access*).

EUD v podstate prenáša vývojové úsilie na koncového užívateľa. Jedným z prvkov nákladov je potrebný čas na návrh riešenia, ďalšími nákladmi je učenie samotné. Tým, že pre koncových používateľov nie je programovanie a vývoj aplikácií primárnou úlohou, ide o veľmi dôležité náklady. Naučiť sa používať prostredie EUD je počiatočným nákladom, ktorý musí byť motivovaný vnímanou odmenou v podobe zvýšenej efektívnosti. Náklady alebo chyby sú pre používateľov EUD významnou pokutou pri prevádzke aj pri učení. Okrem toho majú chyby demotivačný účinok [50]. Nástroje EUD by teda mali byť intuitívne a jednoduché. To ale prináša aj nevýhody. Tým, že nástroje sú navrhnuté tak, aby sa jednoducho používali, môže to znamenať určité obmedzenia v možnostiach pokročilého prispôsobenia. To znamená, že pokročilé a zložitejšie softvérové riešenia si už vyžadujú profesionálnych a školených vývojárov.

## 3.2 Nástroje vizuálnych programovacích jazykov

Podľa [33] až 56.6% nástrojov bolo vyvinutých pre webové platformy, 23.3% pre mobilné platformy a len 20% pre počítače. M. A. Kuhail a spol. [33] uvádzajú, že počet nástrojov vyvinutých pre počítače je nízky. Dôvodom môže byť ich nepraktické používanie. Tieto nástroje si totiž vyžadujú stiahnutie, inštaláciu a často aj pravidelné aktualizácie, čo môže byť pre užívateľov odradzujúce. Na druhej strane webové nástroje nie je nutné inštalovať ani sa nemusia aktualizovať. Do popredia sa postupne dostávajú nástroje vyvinuté na mobilné platformy vďaka narastajúcemu počtu používateľov mobilných telefónov.

Existujú dve všeobecne známe kategorizácie vizuálnych programovacích jazykov. Myers [37] klasifikuje vizuálne programovacie jazyky podľa techník špecifikácie. Niektoré kategórie v Myersovej klasifikácii možno zovšeobecniť do jednej kategórie, zatiaľ čo Burnett a Baker [13] uviedli tri široké podkategórie, a to diagramatické jazyky, ikonické jazyky a jazyky založené na statických obrazových sekvenciách. Kombináciou týchto dvoch taxonómií rozdelujeme VPL do štyroch kategórií:

- **Jazyky založené na blokoch** (*Block-based*) – umožňuje užívateľom vytvárať programy skladaním a spájaním blokov dohromady podľa potreby, použité najmä na edukačné potreby (Scratch, Blockly).
- **Jazyky založené na diagramoch** (*Diagram-based*) – charakterizované sú spájaním grafických objektov šípkami alebo čiarami, ktoré vyjadrujú vzťahy medzi objektami (Node-Red, Blender Nodes<sup>1</sup>)
- **Jazyky založené na formulároch** (*Form-based*) – umožňujú používateľom podobne ako pri blokových VPL vyberať z predpripravených objektov a pomocou formulárov popisovať chovanie daného objektu (EUCalipTool).
- **Jazyky založené na ikonách** (*Icon-based*) – používajú ikony a obrázky, ktoré reprezentujú objekty a akcie, ako napríklad odstránenie alebo editovanie súboru (MicroApp).

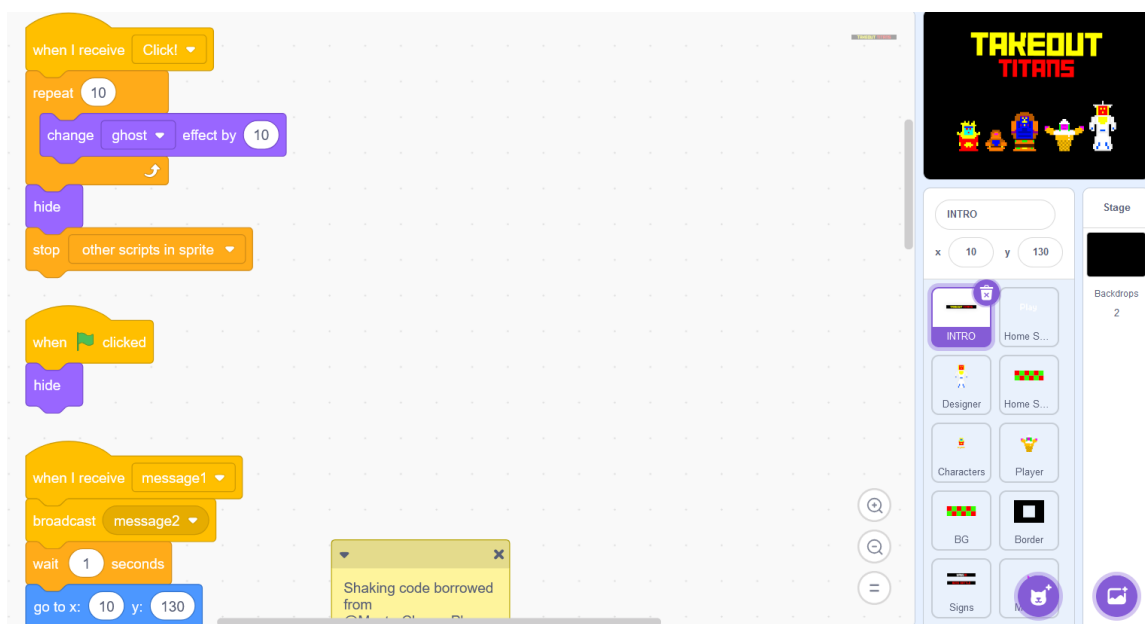
### Jazyky založené na blokoch

Blokové jazyky umožňujú používateľom si vyberať a presúvať bloky z vopred pripraveného zoznamu. Tieto bloky sa spájajú dohromady a vytvárajú výsledný program [33]. Tento

<sup>1</sup>Oficiálna dokumentácia k Blender Nodes: [https://docs.blender.org/manual/nb/2.79/render/blender\\_render/materials/nodes/introduction.html](https://docs.blender.org/manual/nb/2.79/render/blender_render/materials/nodes/introduction.html)

spôsob je obzvlášť vhodný pre začiatočníkov bez predchádzajúcich skúseností s programovaním. Tým, že užívatelia spájajú rôzne komponenty dokopy, eliminuje sa potreba písať syntakticky správny kód, čím sa znižuje riziko chýb.

Jedným z najznámejších príkladov programovacieho jazyka založeného na blokoch je Scratch<sup>2</sup>, ktorý vyvinulo MIT. Scratch je najväčšia svetová kódovacia komunita a kódovací jazyk s jednoduchým vizuálnym rozhraním, ktorý umožňuje mladým ľuďom vytvárať napríklad digitálne príbehy, hry a animácie [1]. Predovšetkým je tento program určený pre deti vo veku od 8 do 16 rokov na edukačné účely. Podľa [43] hlavnými dôvodmi, prečo Scratch bol vyvinutý, boli najmä príliš zložité prvé programovacie jazyky, pri ktorých mladí ľudia nedokázali zvládnuť syntax programovania. Programovanie sa tiež často zavádzalo pomocou aktivít, ktoré nesúviseli so záujmami mladých ľudí, poprípade nebolo poskytnuté poradenstvo v prípade, keď sa naprogramované veci nedarili. Programovacie jazyky by mali byť ľahké na začatie, s možnosťou postupom času vytvárať komplexnejšie projekty. Aby boli tieto body splnené, Scratch bol vyvinutý tak, aby bol viac hravý, zmysluplný a viac spoločenský [43]. Podobne ako pri stavebnici Lego, konektory grafických blokov naznačujú, ako majú byť poskladané a spájané, čím vznikajú výsledné programy. Obrázok 3.1 ukazuje rozhranie programu Scratch. Na ľavej strane sa nachádza skriptovacia oblasť, kde sú vytvorené bloky, ktoré ovládajú správanie programu. Na pravej strane obrazovky je zobrazená scéna hry, konkrétne úvodná obrazovka s názvom hry a postavami. Pod ňou sa nachádza zoznam objektov, ktoré tvoria hru. Ide napríklad o postavy, pozadie, hranice alebo informačné prvky.



Obr. 3.1: Hra vyvinutá v jazyku Scratch<sup>3</sup>

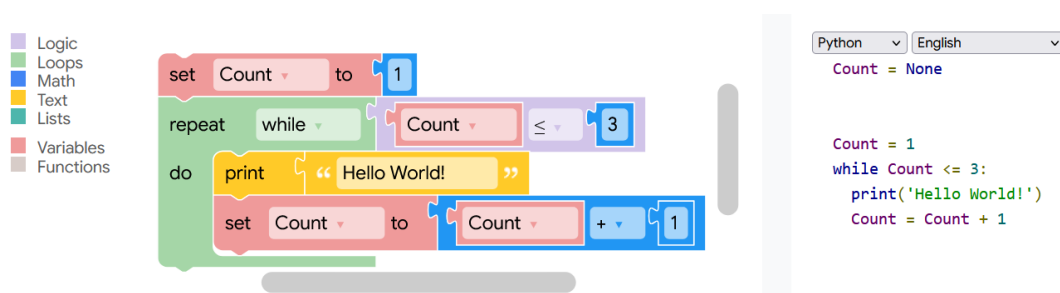
Rozhranie programu Scratch je rozdelené do troch častí: oblasť scény, paleta blokov, kde sa nachádzajú objekty, a oblasť kódovania na umiestnenie objektov, ktoré vytvoria výsledný skript. Oblasť scény je miesto, kde sa vykonávajú naprogramované akcie. Program sa spustí

<sup>2</sup>Domovská stránka vizuálneho programovacieho jazyka Scratch: <https://scratch.mit.edu/>

<sup>3</sup>Adresa projektu: <https://scratch.mit.edu/projects/1036569793/>

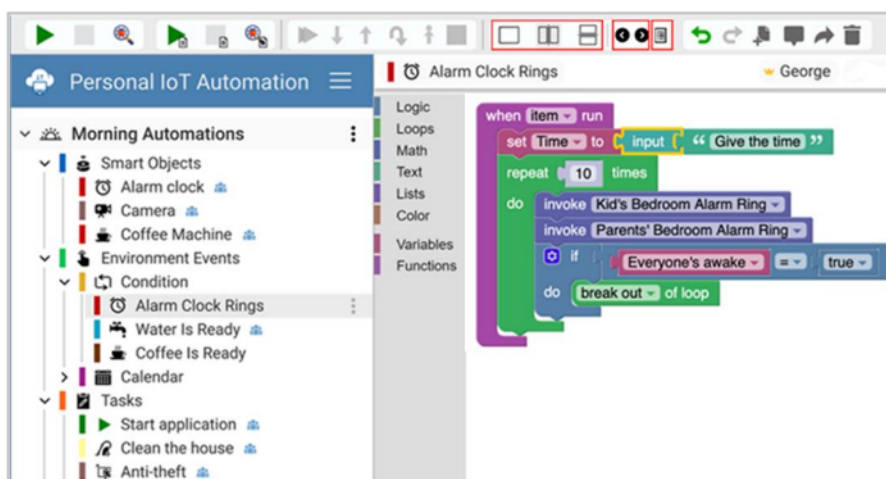
kliknutím na zelenú vľajku (podobne ako `main()` funkcia v tradičných programovacích jazykoch).

Ďalším populárnou platformou je Blockly<sup>4</sup>, ktorá bola vydaná v roku 2012 spoločnosťou Google. Funguje na podobnom princípe ako platforma Scratch, výhodou ale je možnosť výsledný program exportovať do rôznych textových jazykov, ako napríklad JavaScript, Python, Dart alebo PHP [40]. Programovanie v Blockly je jedno z riešení pre programátorov začiatočníkov, pretože toto programovanie je založené na forme blokov, ich spájanie a presúvanie [2]. Obrázok 3.2 popisuje jednoduchý príklad programu (vľavo) vytvorený v Blockly a vygenerovaný výsledný kód v Pythone (vpravo). Program začína nastavením premennej `count` na 1, nasleduje blok `repeat` (ekvivalent cyklu `while`) s podmienkou, v tele ktorého sa nachádza výpis správy `"Hello World!"` a inkrementácia premennej.



Obr. 3.2: Príklad jednoduchého programu v Blockly

Okrem vzdelávania je nástroj Blockly používaný aj v IoT. V roku 2019 bol vykonaný výskum [2], kde na naprogramovanie snímača pohybu na M5Stack doske bola použitá platforma Blockly za pomoci MQTT protokolu. Aby sa podporil vývoj programov pre IoT v Blockly, boli pridané nové objekty, ktoré umožňujú manipuláciu s vlastnosťami zariadení, udalosťami, ktoré umožňujú definovať akcie súvisiace so zmenami stavu zariadenia alebo s plánovanými akciami, ktoré sa môžu uskutočniť raz alebo opakovane [46].



Obr. 3.3: Príklad programu popisujúci automatické zvonenie budíkov na základe nastaveného času [46]

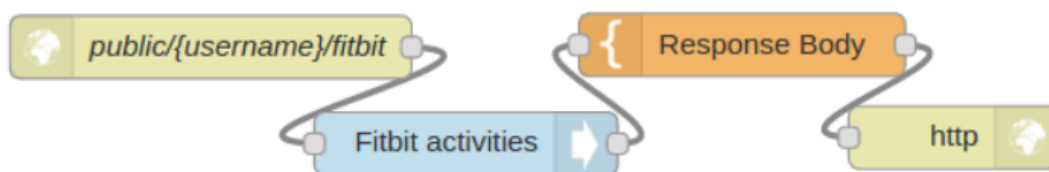
<sup>4</sup>Domovská stránka platformy Blockly: <https://developers.google.com/blockly>

A. Savidis a spol. vo svojej práci [46] popisujú platformu Blockly ako dobrý nástroj na programovanie inteligentných zariadení. Obrázok 3.3 ukazuje platformu Blockly, kde sa na ľavej strane obrázka nachádza zoznam programov, ktoré si užívateľ naprogramuje pre svoje inteligentné zariadenia. Napravo sa nachádza už konkrétny program. Program sa začne vykonávať v hlavnej udalosti označenej ako `when Item run` a následne sa nastaví premenná `Time`, ktorej hodnotu si určí užívateľ na základe bloku `input`. Následne sa vykoná cyklus `repeat` desaťkrát, ktorý v každej svojej iterácii vyvolá zvonenie budíku pre detskú izbu a rodičovskú izbu. Podmienka `if Everyone's awake = true` kontroluje, či sú všetci členovia rodiny zobudení a v prípade pravdy vyskočí z cyklu, inak sa cyklus opakuje ďalej.

## Jazyky založené na diagramoch

Jazyky založené na diagramoch sú po jazykoch založených na blokoch druhé najpoužívanejšie. Charakterizujú sa spájaním grafických objektov pomocou šípiek alebo čiar, ktoré symbolizujú vzťah medzi objektami [33]. Podobne ako pri jazykoch založených na blokoch, tento prístup je výhodný pre používateľov, ktorí nemajú znalosti programovania. Zjednodušuje proces návrhu programu a zvyšuje prehľadnosť. Výhodou diagramovo orientovaných jazykov je intuitívnosť, kde používateľ môže vizuálne sledovať, ako údaje prechádzajú systémom. Ďalej umožňujú ľahkú modifikáciu programu jednoduchým presúvaním vzťahov medzi objektami.

Príkladom je Node-Red<sup>5</sup> vývojový nástroj, ktorý sa používa na integráciu hardvérových IoT zariadení, rozhraní API (*Application Programming Interfaces*) a online služieb, pôvodne vyvinutý tímom IBM Emerging Technology Services [34, 42]. Je to bezplatný nástroj vybudovaný na JavaScripte, postavený na platforme Node.js, ktorý tiež poskytuje webový vizuálny editor. Systém je postavený na uzloch, napríklad uzol *Inject*, ktorý sa používa na spustenie toku, uzol *HTTP Request*, ktorý slúži na odosielanie HTTP požiadaviek alebo uzol *Function*, ktorý umožňuje používateľom napísať vlastný JavaScript kód. Všetky uzly sú reprezentované príslušnými ikonami. Okrem spomínaných funkcií poskytuje uzly, ktoré sú schopné monitorovať tok pomocou uzlu ladenia alebo uzlu Raspberry Pi, ktorý je schopný čítať a zapisovať pomocou pinov GPIO (*General-Purpose Input/Output*) [34].



Obr. 3.4: Jednoduchý príklad webového servera na tvorbu grafov s Fitbit údajmi<sup>6</sup>

Program na obrázku 3.4 sa skladá zo štyroch uzlov, kde prvý a posledný uzol sú vstupné a výstupné HTTP uzly, ktoré spolupracujú pri počúvaní HTTP požiadaviek a odosielaní HTTP odpovedí. Keď príde HTTP požiadavka na vstupný uzol (`public/{username}/fitbit`), vytvorí správu, ktorá spustí Fitbit uzol (`Fitbit activities`). Ten získa aktuálne štatistiky

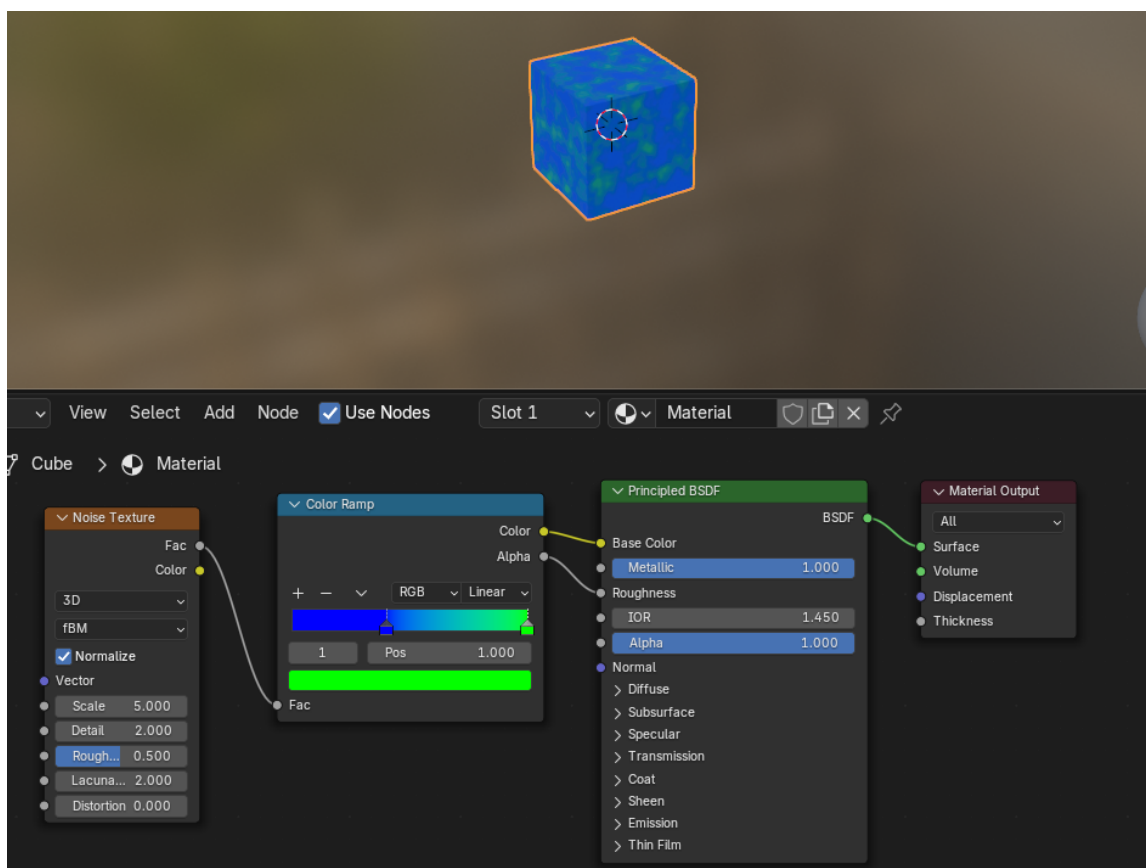
<sup>5</sup>Domovská stránka Node-Red: <https://nodered.org/>

<sup>6</sup>Príklady projektov v nástroji Node-Red: <https://noderedguide.com/nr-lecture-1/>

používateľa a potom ich odovzdá uzlu šablóny (*Response Body*). Uzol šablóny je predpripravený uzol v Node-Red systéme, ktorý dokáže vytvárať ľubovoľný kód, podobne ako uzol funkcie, v tomto prípade v jazyku HTML. Uzol šablóny vygeneruje jednoduchý HTML kód na zobrazenie získaných dát, ktorý následne odovzdá ako správu konečnému uzlu. Výstupný uzol (*http*) zabalí HTML ako HTTP odpoveď a tú následne odošle späť prehliadaču, čím sa používateľovi zobrazí výsledok.

Ďalším príkladom môžu byť Blender Geometry/Shader Nodes, ktoré sú súčasťou softvéru Blender<sup>7</sup>, ktorý umožňuje tvorbu 3D grafiky, animácií a vizuálnych efektov. Ide o podobný systém ako Node-Red, ktorý využíva uzly. Tie sú prepájané čiarami, ktoré znázorňujú tok údajov medzi jednotlivými krokmi. Každý uzol predstavuje konkrétnu operáciu alebo funkciu (napríklad výpočet farieb, textúrovanie, spracovávanie obrazových dát alebo modifikácia geometrie).

Obrázok 3.5 ukazuje použitie Shader Nodes v Blenderi, ktorý vytvára procedurálny materiál na objekte. Uzol *Noise Texture* generuje šumovú textúru, kde výstup *Fac* (faktor) je pripojený na vstup *Fac* uzla *Color Ramp*. Tento uzol umožňuje mapovať hodnoty šumu na farebný gradient, pričom je upravený tak, aby obsahoval modrú a zelenú farbu. Výstup *Color* z uzla *Color Ramp* je pripojený k vstupu *Base Color* a vstupu *Roughness* uzla *Principled BSDF*, ktorý definguje vlastnosti materiálu a výstup tohto uzla je pripojený k *Material Output*, ktorý aplikuje materiál na objekt, v tomto prípade na kocku.



Obr. 3.5: Uzlové nastavenie v Blenderi na tvorbu procedurálneho materiálu

<sup>7</sup>Domovská stránka softvéru Blender: <https://www.blender.org/>

## Jazyky založené na formulároch

Jazyky založené na formulároch sú používané iba v 13.3% podľa [33]. Umožňujú vytvárať formuláre, do ktorého sa spúšťače a akcie pridávajú pomocou textových ponúk alebo pomocou *drag-and-drop* [33]. Do tejto skupiny vizuálnych programovacích jazykov patrí EUCalip-Tool. Jedná sa o mobilná aplikácia, primárne určená používateľom bez programátorských skúseností, ktorá im umožňuje jednoducho vytvárať aplikácie a služby [51]. Kompozícia aplikácie je tvorený pridávaním prvku do rámca. Táto kompozícia slúži ako hlavný rámec a môže obsahovať aktivity a fragmenty. Zatiaľ čo aktivity predstavujú vysokoúrovňové reprezentácie služieb, fragmenty sú konkrétne štruktúry (cykly alebo podmienené štruktúry). Tie môžu znova obsahovať aktivity alebo fragmenty [51, 52]. Všetky programy sú exportované do BPMN (Business Process Modeling Notation). Na generovanie sa používajú kusy (angl. *pieces*). Každý tento kus predstavuje časť programu, ktorú môže jedna mikroslužba vykonať, čo umožňuje používateľovi vytvoriť program, ktorý spolupracuje s viacerými mikroslužbami [30].



Obr. 3.6: Vytvorenie jednoduchšej služby v programe EUCalipTool [52]

V práci, ktorej autori sú Valderas, Torres, Mansanet a Pelechano [52], sa zameriavali podporou bežných užívateľov pri tvorbe služieb prostredníctvom mobilných zariadení. Práca predstavuje DSVL (*Domain-Specific Visual Language*), teda vizuálny jazyk špecifický pre danú oblasť, v tomto prípade EuCalipTool. Úlohou používateľa bolo vytvoriť službu, ktorá mimikuje deň študenta a jeho každodenné povinnosti (rezervácia miest v knižnici, kontrola predpovede počasia alebo výber dopravného prostriedku).

Na obrázku 3.6 je popísaný proces vytvorenia jednoduchkej služby, ktorá bola zadaná v práci [52]. Na začiatku je vytvorená prázdna kompozícia (časť č. 1). Po kliknutí na tlačidlo '+' sa zobrazí zoznam dostupných služieb, ako je zobrazené v časti 2. Z dostupných služieb si podľa práce používateľ vybral rezerváciu miesta v knižnici, ako je vidieť v časti 3. Následne po stlačení tlačidla '+' si používateľ na obrázku č. 4 vybral *Weather Item* z preddefinovaných možností. Možnosť počasia je rozvetvená na ďalšie možnosti (č. 5), kde si používateľ pre každý druh počasia môže ďalej nastaviť ďalšiu funkcionality; v tomto prípade pre snečné počasia si používateľ nastavil službu rezervácie parkovacieho miesta na bicykel a zobrazenie informácií o aktuálnej dopravnej situácii (časť č. 6).

### Jazyky založené na ikonách

Jazyky založené na ikonách sú najmenej používané, iba v 6.6% [33]. Hlavným prvkom týchto jazykov je používanie ikon alebo grafických objektov na reprezentáciu určitého objektu alebo akcie. Základné ikony reprezentujú objekty (súbor) alebo akcie (mazanie, editovanie), zatiaľ čo zložené ikony pozostávajú z viacerých základných ikon a predstavujú vety [33]. Takáto reprezentácia môže byť účinná, keďže využíva výkonný a vysoko paralelný ľudský vizuálny systém, ktorý dokáže efektívne spracovávať obrázky. Vizuálne reprezentované informácie sa môžu aj ľahšie zapamätať vďaka efektu nadradenosti obrazu [19].



Obr. 3.7: Ikony používané v aplikácii MicroApp a jednoduchý príklad [19]

Autori [19] popisujú aplikáciu MicroApp nasledovne: je založená na princípe kompozície služieb. To znamená, že prvky sú ikony reprezentujúce služby. Ikonické symboly sú znázornené zaoblenými obdĺžnikmi a každá ikona služby vo vnútri zobrazuje obrázok reprezentujúci kategóriu objektu, s ktorým služba pracuje (napr. číselný údaj), alebo vykonávanú činnosť (napr. fotoaparát na označenie činnosti vyhotovenia snímky). Služby sú reprezentované rôznymi typmi (obrázok 3.7 vľavo), napríklad služby zariadení (*Phone.Call*, *Ca-*

*mera.Take* alebo *SMS.Send*), webové služby (prehľad počasia, rezervácia leteniek) a služby automatizácie domova (zapnutie klimatizácie).

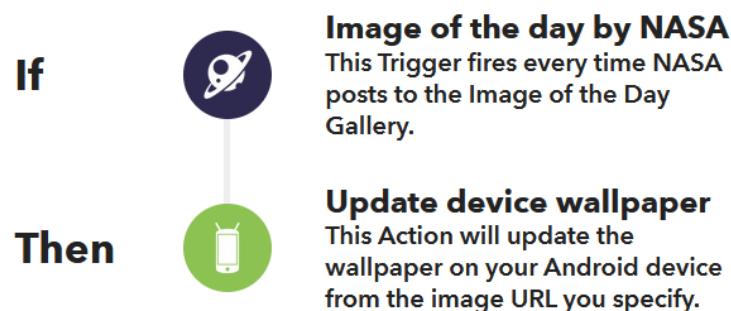
Príklad na obrázku 3.7 vpravo popisuje situáciu, kde používateľ chce odoslať každú vytvorenú fotografiu cieľovému príjemcovi spolu s adresou, kde bola fotografia vytvorená. Na začiatku sa pridá ikona *Camera.Take* a uloží sa na zariadenie *Camera.Save*. Ikona *Contact.Static* vyberie kontakt, ktorému má byť fotografia poslaná. Služba *GPS.Location* získa zo zariadenia GPS súradnice a výstup tejto služby ide do *Maps.GeoLocation*, ktorá získa adresu z internetu na základe súradníc, následne ikona *Send.StaticMail* sa spojí s výstupmi ostatných ikon. Týmto spôsobom sa vykoná vizuálna veta pomocou súradníc v aplikácii MicroApp.

### 3.3 Trigger action programming a parametrizácia

**Trigger action programming** (TAP) je model programovania, ktorý umožňuje užívateľom bez technických znalostí automatizovať služby v inteligentných zariadeniach [59]. Využíva na to pravidlá ak-toto-potom-toto (angl. *if-this-then-that*), pomocou ktorých si užívatelia definujú svoje vlastné scenáre. V. Zhao a spol. vo svojej práci [59] uvádzajú napríklad takúto situáciu: "**AK** Alice zaspí **POKÝM** je noc **A** predné dvere sú odomknuté **POTOM** zamkni predné dvere". Pomocou takýchto konštrukcií sú užívatelia schopní si vymýšľať vlastné scenáre na rôzne situácie v rámci svojich inteligentných domovov.

Jednou z takýchto platforiem, ktorá ponúka tento prístup programovania, sa nazýva obdobne ako pravidlá v TAP. Jedná sa o platformu IFTTT<sup>8</sup>, pomocou ktorej si užívatelia môžu prepájať rôzne zariadenia a umožňuje rôzne kombinovať pravidlá medzi nimi. Jedná sa o webovú službu, kde si užívatelia môžu pomocou reťazcov vytvárať podmienené príkazy, zvané anglicky *applets* [57]. Platforma funguje na princípe "Ak je nejaký spúšťač aktivovaný, vykonaj príslušnú akciu".

Your device can now contain the  
amazing vastness & beauty of space!  
Basically.



Obr. 3.8: Príklad *applet*-u v platforme IFTTT<sup>9</sup>

<sup>8</sup>Domovská stránka platformy IFTTT: <https://ifttt.com/>

<sup>9</sup><https://ifttt.com/applets/yNvHX9VQ-update-your-android-wallpaper-with-nasa-s-image-of-the-day>

Platforma IFTTT integruje spolu aj s ostatnými populárnymi službami, menovite Gmail, Instagram alebo Facebook [57]. Konkrétny príklad, ktorý si užívatelia môžu sami nastaviť, môže byť situácia, keď užívatel' vojde do miestnosti so senzorom pohybu, rozsvieti sa mu Philips Hue lampa alebo si môžu nastaviť *applet* na obrázku 3.8, ktorý v momente, keď NASA vydá obrázok dňa, ho užívatel'ovi nastaví ako pozadie na telefóne. Ďalšou aplikáciou s podobným princípom vytvorila spoločnosť Samsung. Nazýva sa SmartThings<sup>10</sup>. Jej cieľom je ovládať rôzne inteligentné zariadenia v rámci jednej aplikácie, poprípade s využitím hlasového klienta.

**Parametrizácia** v kontexte EUD znamená proces, kde si užívatelia môžu prispôbiť systémy alebo aplikácie pomocou nastavovania vstupov, hodnôt alebo pravidiel. Pri TAP si užívatelia nastavujú pravidlá v schéme "**ak** niečo, **potom** niečo", parametrizácia sa líši tým, že si užívatelia prispôbujú alebo konfigurujú systém pomocou parametrov. Parametrizácia môže priamo ovplyvňovať správanie TAP, napríklad si užívatel' nastaví prahovú teplotu na 20°. Pri TAP spúšťači si potom vie užívatel' vytvoriť pravidlo "Ak teplota klesne pod {*prahova\_teplota*}, zapni kúrenie".

---

<sup>10</sup>Domovská stránka aplikácie SmartThings od spoločnosti Samsung: <https://www.samsung.com/us/smartthings/>

# Kapitola 4

## Analýza

Pri vyhodnocovaní programov napísaných pomocou VPL je potrebné zohľadniť viaceré požiadavky a pravidlá. Pri vyhodnocovaní programov je potrebné zabezpečiť validáciu syntaxe a štruktúru programu. To zahŕňa kontrolu správneho prepojenia blokov, typovú kompatibilitu alebo či použité bloky a ich kombinácie zodpovedajú pravidlám VPL. Okrem toho je potrebné vykonať sémantickú analýzu, t. j. či sú deklarované použité premenné, či má program správne definované poradie použitých príkazov (napríklad či vetva `else` nie je skôr ako vetva `if`) alebo overiť logické chyby, napríklad delenie nulou alebo nesprávne logické výrazy.

Počas vykonávania programu sa môžu vyskytnúť aj chyby, ktoré priamo ovplyvňujú správnosť výsledkov, tieto chyby ale nie je možné detekovať pomocou syntaktickej a sémantickej analýzy. Preto je nutné vykonávať správnu kontrolu aj počas behu programu. Pri výskyte chyby je dôležité, aby interpret poskytol zrozumiteľný výstup pre používateľa. Každá chyba by teda mala byť podrobne popísaná, o akú chybu sa jedná a kde nastala (napríklad názov bloku, popríklad číslo riadku). Okrem tohto vstup do interpreta by mal byť reprezentovaný v štruktúrovanej forme, ktorú vyhodnocovací program vie jednoducho spracovať (napríklad JSON alebo XML). Teda pre interpret je potrebné, aby bolo jasné, o aké konštrukcie jazyka sa jedná.

### 4.1 Vizualný programovací jazyk

Práca P. Johna [30] a výskumná správa [25] v spolupráci s firmou Logimic<sup>1</sup> rieši problematiku, kde sa mnohé vizuálne programovacie jazyky síce zameriavajú na inteligentné zariadenia internetu vecí, obsahujú ale určité nevýhody a preto nenapĺňajú svoj kompletný potenciál. Navrhované riešenie v práci spomína vytvorenie nového vizuálneho programovacieho jazyka spolu s grafickým editorom, ktorý by sa zameriaval primárne na mobilné telefóny a zariadenia s malými dotykovými obrazovkami [30].

Prototypové riešenie vizuálneho programovacieho jazyka je tvorené serverovou časťou (vyhodnocovanie programov) a klientskou časťou (editor), obe napísané v jazyku TypeScript. Ako sa píše v téze [30], samotný jazyk používa metajazyk, ktorý je zodpovedný za definovanie povolených jazykových konštrukcií a modelu jazyku ako takého. To znamená, že editor môže fungovať so skupinou jazykov, kde každý jazyk má vstavané bloky programu (podmienené alebo dynamicky pridané). Metajazyk popisuje, čo je povolené a čo nie. Vďaka tomuto je možné jazyk nadefinovať tak, aby napríklad nebolo možné vytvoriť

<sup>1</sup>Firma špecializujúca sa na internet vecí: <https://www.logimic.com/cs/>

program, ktorý by obsahoval zavorené bloky v príkaze zariadenia. Zjednodušenú podobu jazyka ukazuje výpis 4.1.

```
type Language = {
  variables: Variable[],
  stmt: {
    [name: string]: LanguageStatement
  }
}

type LanguageStatement = {
  positions?: PreferredPosition[], // list of preferred positions
  avoidPositions?: PreferredPosition[], // list of prohibited positions
  parents?: string[], // list of preferred parents
  avoidParents?: string[], // list of prohibited parents
  levels?: number[], // list of preferred levels
  avoidLevels?: number[], // list of prohibited levels
}

type Variable = string

type PreferredPosition = (number | "first" | "middle" | "last" )
```

Výpis 4.1: Zjednodušený model metajazyku, prevzatý z [30].

Okrem toho metajazyk podporuje dva typy príkazov, ktoré rozširujú základný príkaz `LanguageStatement` 4.2. Prvý typ jednoduchého príkazu obsahuje zoznam parametrov (číslo, reťazec alebo logickú hodnotu), okrem toho je ale možné nadefinovať rôzne ďalšie rozšírenia tohto príkazu. Druhý typ podporovaných príkazov sú zložené príkazy, ktoré umožňujú povoliť alebo zakázať podmienku. To umožňuje vytvoriť príkazy typu `if` alebo `while`.

```
type LanguageCmdStatement = LanguageStatement & {
  component: "cmd",
  extensions: {
    params?: {
      type: "array"
      defs: Type<Number | String | Boolean>
    }
  }
}

type LanguageCompoundStatement = LanguageStatement & {
  component: "compound",
  extensions: {
    enableCondition?: boolean
  }
}
```

Výpis 4.2: Jednoduchý metamodel typov príkazov, prevzaté z [30].

Novšiu verziu vizuálneho programovacieho jazyka, ktorý vychádza z metajazyka, a grafického editoru realizoval vo svojej bakalárskej práci [41] L. Podvojský. Vizualný programovací jazyk bol implementovaný ako knižnica, ktorú je možné integrovať do webových projektov. Na implementáciu bol použitý jazyk TypeScript a celá knižnica je rozdelená do dvoch hlavných častí. V prvej časti sa implementujú sady grafických komponentov, druhá časť obsahuje moduly pre prácu s VPL. Grafické komponenty sú vytvorené pomocou knižnice Lit<sup>2</sup>, na vyhodnocovanie programov ale nemajú tieto komponenty vplyv, preto nie je potrebná ďalšia analýza. Vytvorený program pomocou vizuálneho programovacieho jazyka je transformovaný a vygenerovaný do JSON formátu. Syntax je následne kontrolovaná statickou analýzou pri akejkoľvek zmene v reprezentácii programu [41]. Týmto je splnená jedna z dôležitých podmienok pri interpretácii programu. JSON sa skladá z hlavičky a tela programu.

## Hlavička

Hlavička v JSON-e je identifikovaná pomocou atribútu `"header"` a obsahuje užívateľsky definované premenné a procedúry. Výpis 4.3 ukazuje príklad definície premennej. Premenné sú definované v atribúte `"userVariables"` a sú identifikovateľné názvom, typom (string, číslo, boolean alebo výraz) a samotnou hodnotou.

```
"userVariables": {  
  "foo": {"type": "string", "value": "abc"}  
},
```

Výpis 4.3: Príklad definície premennej `foo`

Procedúry sú definované v atribúte `"userProcedures"`. Procedúra je identifikovateľná názvom, vo výpise 4.4 to je `myProc`. Slúžia na definíciu zapuzdrenia znovupoužiteľných častí alebo na zapuzdrenie a abstrakciu zložitejšej logiky [41]. Telo procedúry je definované rovnakým spôsobom ako hlavný program. V hlavnom programe sú potom procedúry volané pomocou atribútu `"id"` a názvu procedúry (`"id": "myProc"`).

```
"userProcedures": {  
  "myProc": [...]  
}
```

Výpis 4.4: Príklad definície procedúry `myProc`

## Riadiace príkazy

Hlavný program je definovaný v hlavnom atribúte `"block"` na rovnakej úrovni ako hlavička. Blok programu obsahuje zoznam príkazov, kde atribút `"id"` určuje, o aký typ príkazu ide. Riadiace príkazy (`if`, `else`, `while`, `repeat`, `switch`) majú v sebe atribúty `"block"` a `"arguments"`. Blok riadiaceho programu predstavuje jeho telo, argumenty pri podmienených príkazoch a cykle `while` predstavujú podmienku spolu s operandami a operátormi, ako je ukázané vo výpise 4.5. V tomto prípade sa jedná o podmienený príkaz, ktorý by v pseudokóde vyzeral nasledovne: `if (foo === "active")`. Cyklus `repeat` má vo svojich argumentoch počet opakovaní, poprípade premennú s typom `number`, ktorá značí počet

<sup>2</sup>Stránka knižnice Lit: <https://lit.dev/>

opakovaní. Príkaz `switch` má v argumente výraz, ktorý sa porovnáva s hodnotami v `case`, ten má ako argument porovnávanú hodnotu spolu s blokom kódu.

Postupom času program prešiel miernou revíziou, kde sa premenovali niektoré názvy kľúčov a spresnili/premenovali do jednoznačnejšej a výstižnejšej formy, napríklad `args` na `arguments`, dátový typ `num` na `number` alebo v podmienkach `opds` (operandy) a `opr` (operátor) na `value` a `type`.

```
{
  "id": "if",
  "block": [],
  "arguments": [
    {
      "type": "boolean_expression",
      "value": [
        {
          "value": [
            {"type": "variable", "value": "foo"},
            {"type": "string", "value": "active"}
          ],
          "type": "==="
        }
      ]
    }
  ]
}
```

Výpis 4.5: Príklad podmieneného príkazu

## Akčné príkazy

Jazyk ďalej podporuje dva akčné príkazy. Príkaz poslania notifikácie je identifikovateľný ako `"alert"`. Obsahuje tri argumenty, prvý z nich určuje, či sa má notifikácia poslať na telefónne číslo alebo mail. Ďalšie dva argumenty určujú konkrétne telefónne číslo alebo mail a text správy. Druhý akčný príkaz je jednoduché nastavovanie premenných (4.6). Príkaz je identifikovateľný ako `"setvar"` a obsahuje dva argumenty: názov premennej a hodnotu.

```
{
  "id": "setvar",
  "arguments": [
    {"type": "variable", "value": "foo"},
    {"type": "number", "value": 10}
  ]
}
```

Výpis 4.6: Príklad príkazu nastavenia premennej var

Okrem typických operandov vo výrazoch si môže užívateľ zvoliť parametre zariadení alebo použiť priamo príkaz zariadenia (viz 4.7). Jednotlivé argumenty je možné identifikovať na základe rozdelenia reťazca cez znak `".`", kde prvá časť je identifikátor zariadenia a druhá časť popisuje konkrétny parameter zariadenia, poprípade jeho príkaz.

```
{"type": "variable", "value": "DistanceSensor-1.waterLevel"}
{"id": "Doorbell-1.takePicture"}
```

Výpis 4.7: Príklad použitia parametra zariadenia a príkazu zariadenia

## 4.2 Pôvodný interpret

Pôvodný interpret vizuálneho programovacieho jazyka je implementovaný v jazyku TypeScript. Implementácia interpreta je založená na **objektovo-orientovanom dizajne** (angl. *Object-Oriented Design, OOD*). Je to programovací prístup, ktorý sa sústreďí primárne na škálovateľnosť (schopnosť systému rásť), modularitu (rozdelenie programu na menšie samostatné časti) a opakovanú použiteľnosť (časti kódu môžu byť použité na viacerých miestach). Táto technika rieši problémy vytváraním vzájomne prepojených objektov, využíva rôzne koncepty, napríklad triedy a objekty a ich mechanizmy: zapuzdrenie, dedičnosť, polymorfizmus a abstrakcia [39].

Interpret ďalej využíva **továrenský vzor** (angl. *Factory Pattern*), ktorý poskytuje rozhranie na vytváranie objektov. Je užitočný najmä v situáciách, keď je treba vytvoriť viacero objektov rovnakého typu, ktoré sa ale môžu líšiť v určitých parametroch alebo sa musia určiť za behu programu. Okrem tohto vzoru využíva základné koncepty objektovo-orientovaného programovania: rozhrania a abstraktné triedy. Rozhranie špecifikuje, aké metódy musí trieda, ktorá ho používa, implementovať. Rozhrania obsahujú len čisté deklarácie, neobsahujú implementáciu metód, len ich názov, návratový typ a parametre. Abstraktná trieda je trieda, ktorá môže obsahovať implementované aj neimplementované (abstraktné) metódy.

### Štruktúra a funkcionálnosť interpreta

Trieda `ProgramRunner` je základným komponentom interpreta, ktorá slúži ako základný mechanizmus pri interpretácii programu. Slúži na spúšťanie, spracovanie a vyhodnocovanie programov definovaných vo formáte štruktúrovaných dát. Kód 4.8 ukazuje spôsob, akým je možné spustiť vyhodnotenie programu. Výsledkom sú premenné `commands` a `toUpdate`, ktoré obsahujú príkazy, ktoré sa majú spustiť a poslať do inteligentných zariadení spolu so zmenenými parametrami/stavmi týchto zariadení.

```
const runner = new ProgramRunner()
  .setCommander(commander)
  .setReferenceManager(referenceManager)
  .parseProgram(program);

const {commands, toUpdate} = await runner.run();
```

Výpis 4.8: Kód na vyhodnotenie programu

Trieda `ProgramRunner` obsahuje komponenty `Program`, ktoré reprezentujú spracovaný program, pomocou triedy `Commander` po vyhodnotení programu odosiela vykonateľné príkazy a `ReferenceManager`, ktorý umožňuje načítanie a aktualizáciu referencovaných hodnôt. Trieda `ProgramRunner` implementuje rozhranie `IRepresentable` s metódou `represent()`, pomocou ktorej je možné vytvoriť serializáciu údajov uložených v objektových štruktúrach.

Celá interpretácia sa deje v metóde 4.9, ktorá definuje postup pri vyhodnocovaní programu. Dôležitou časťou celej interpretácie je rozhranie `IReferenceManager` s metódami `store` a `load`. Úloha metódy `store` je počas ukladania programu nájsť a uložiť všetky parametre zariadení, s ktorými daný program pracuje. Metóda `load` naopak pri vykonaní programu si získa z neho referencie, ktoré sú použité v programe a načíta hodnoty, ktoré sa následne v tabuľke referencií v hlavnom programe nastaví pomocou funkcie `setReferencesTargets`. Následne sa program vykoná a vráti hodnoty, ktoré boli zmenené (`toUpdate`) a príkazy zariadení (`commands`), ktoré je treba vykonať.

```
async run(dry: boolean = false): Promise<{ toUpdate: ReferencedValue[];
                                           commands: Command[] }> {
    const references = await this._referenceManager
      .load(this.program.getReferencesToLoad().map(
        item => ReferencedValue.fromReference(item)
      )
    );

    this.program.setReferencesTargets(references);

    const commands = this.program.evaluate();

    await this._commander.sendCommands(dry, commands);

    const toUpdate = this.program.getReferencesToUpdate();
    await this._referenceManager.store(toUpdate);

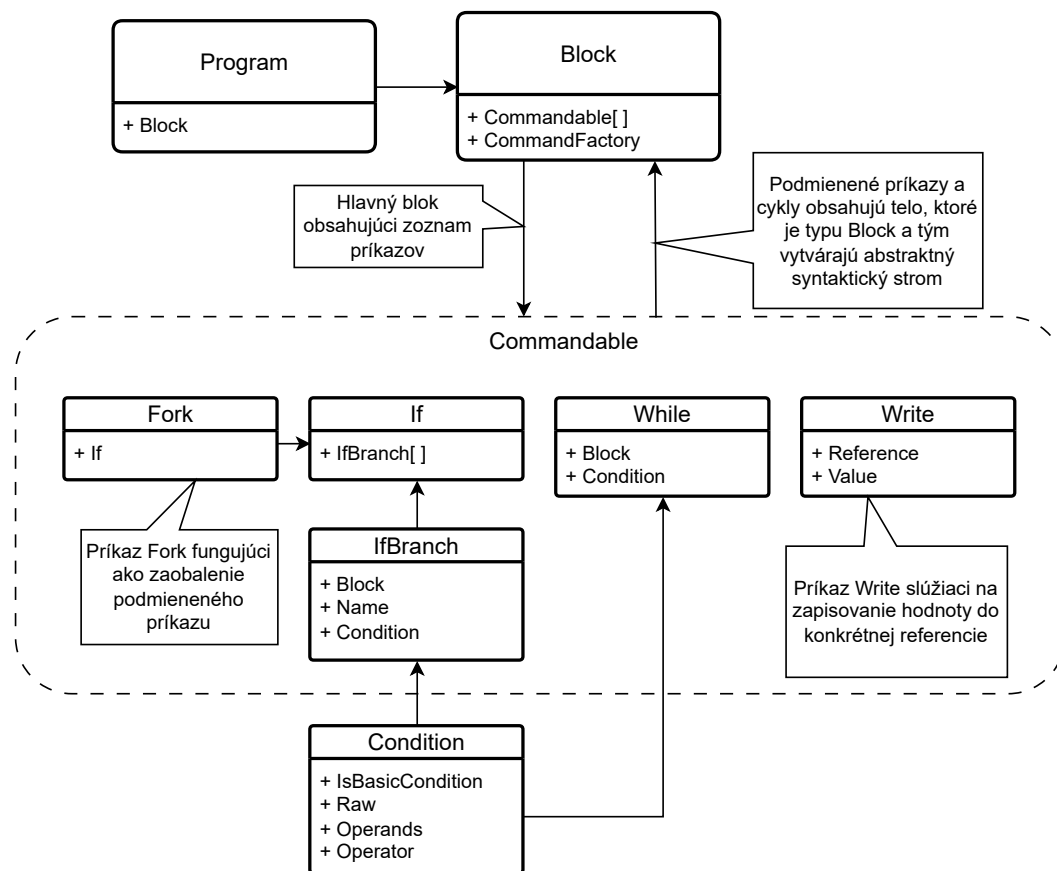
    return {commands, toUpdate};
}
```

Výpis 4.9: Metóda interpretujúca program

Samotná trieda `Program` implementuje rozhranie `IEvaluable` s metódou `evaluate()`. Táto metóda slúži na vyhodnotenie jednotlivých konštrukcií jazyka, ako sú bloky, podmienené príkazy alebo cykly. Okrem toho implementuje aj rozhranie `IRepresentable`, ktoré umožňuje reprezentovať stav programu v čitateľnej forme. Program obsahuje komponentu `Block`, ktorá reprezentuje hlavný program. Tá má v sebe zoznam príkazov typu `Commandable`, viz 4.10. Tento typ obsahuje konštrukcie jazyka, ktoré vytvára na základe kontextu trieda `CommandFactory`. Podmienené príkazy a cykly obsahujú svoje telo, ktoré je typu `Block`, čím sa zavrňajú a vytvárajú abstraktný syntaktický strom príkazov.

```
type Commandable = (If | IfBranch | Fork | While | Command | Write);
```

Výpis 4.10: Typ `Commandable` predstavujúci kolekciu možných príkazov



Obr. 4.1: Model štruktúr jazyka a ich prepojenie

Obrázok 4.1 popisuje vzťahy medzi jednotlivými štruktúrami a ich skladanie. Skladaním jednotlivých blokov spolu s možnými zavorenými blokmi pôvodný interpret vytvára abstraktný syntaktický strom príkazov programu. Jednotlivé príkazy typu `Commandable` definujú vlastné metódy a definujú správanie daného príkazu. Podmienený príkaz `If` je zabalený príkazom `Fork`, ktorý značí rozdelenie programu do viacerých vrstiev a samotný podmienený príkaz obsahuje zoznam vetiev `IfBranch`, ktoré majú v sebe svoje telo, meno (`if`, `elseif` alebo `else`) a podmienku. V prípade, že podmienka v triede nie je definovaná, program túto triedu vyhodnocuje ako `else` príkaz. Trieda `If` vo svojom konštruktore zároveň aj kontroluje správnosť vetiev, teda či obsahuje vetvy, popri prípade či nemá viac ako jeden príkaz `else`. Obdobne funguje aj príkaz `while`, ktorý ale vo svojej triede má iba svoju podmienku a telo príkazu. Podmienené príkazy a cykly obsahujú podmienku, ktorá sa skladá z viacerých parametrov. Obsahuje parameter `isBasicCondition`, ktorý kontroluje, či sa jedná o jednoduchú podmienku a podľa toho ju ďalej vyhodnotí z parametra `Raw`, ktorý v sebe uchováva pôvodnú reprezentáciu podmienky. V prípade komplexnejšej podmienky trieda obsahuje operandy a operátor. Príkaz `Write` slúži na reprezentáciu a vykonanie príkazu, ktorý zapisuje hodnoty do konkrétnej referencie.

Každá konštrukcia jazyka (4.10) spolu s `Block` implementuje rozhranie `IEvaluable` s metódou `evaluate()`, to znamená, že pri prvom zavolaní vyhodnotenia na prvom základnom bloku sa prejde a vyhodnotí celý abstraktný syntaktický strom príkazov programu,

ktorého výsledok je zoznam príkazov, ktoré je potrebné zaslať zariadeniam spolu so zoznamom referencií, ktoré boli zmenené.

### 4.3 RIoT

RIoT (**R**eal-time **I**o**T**) je informačný systém na spracovávanie dát a monitorovanie stavov zariadení pomocou kritérií definovaných užívateľom (*KPIs* – *Key performance indicators*). Systém bol navrhnutý M. Burešom v rámci jeho bakalárskej práce [12], ktorý je ale ďalej vyvíjaný. Systém funguje na trojvrstvovej architektúre (frontend, backend, relačná databáza), štandardná interpretácia tejto architektúry smeruje k tzv. *monolitickému backend-u*<sup>3</sup>. Preto sa autor rozhodol pre architektúru mikroslužieb, kde backend je rozdelený do "malých" častí aplikácie, ktoré sú zodpovedné za určitú časť funkcionality. Podľa slov M. Bureša [12] "je možné systém tak do určitej miery škálovať horizontálne a vďaka rozdeleniu zdrojového kódu medzi služby je možné ho lepšie udržiavať a rozvíjať." Autor vo svojej práci popisuje tri mikroslužby:

- **MQTT Preprocessor** prijíma a dekoduje MQTT správy a na základe ich obsahu generuje správy, ktoré sú ďalej zaslané do systému,
- **Message processing unit** vykonáva samotné spracovanie správ,
- **Backend core** je zodpovedný za ostatnú funkcionality systému, t. j. príjem výsledkov spracovaných dát, obsluha relačnej databázy a CRUD operácie (Create – vytvoriť, Read – čítanie, Update – aktualizácia, Delete – mazanie)

Informačný systém je implementovaný jazykom Go<sup>4</sup> a využíva GraphQL<sup>5</sup> na manipuláciu s dátami. Jedná sa o flexibilný nástroj, pomocou ktorého vie klientská strana špecifikovať, aké dáta potrebuje. Systém využíva relačnú databázu PostgreSQL<sup>6</sup> a na komunikáciu s ňou je použitá knižnica GORM<sup>7</sup>. Táto knižnica zjednodušuje prácu s relačnými databázami a umožňuje efektívne mapovanie medzi objektmi a tabuľkami. Mikroslužby, ktoré tvoria backend, komunikujú medzi sebou pomocou technológie RabbitMQ<sup>8</sup>. RabbitMQ je open-source message broker, ktorý umožňuje asynchrónnu komunikáciu medzi rôznymi aplikáciami alebo komponentami s použitím protokolu AMQP (viz popis protokolov v sekcii 2.2).

Mikroslužba **Backend-core** definuje niekoľko vrstiev deliace logiku mikroslužby na jednotlivé časti. Vrstva **api/graphql** poskytuje rozhranie pre klientov na komunikáciu s backendom prostredníctvom GraphQL API (**schema.graphqls**) spolu s typmi, operáciami čítania (*Queries*) a operáciami modifikácie (*Mutations*). Typy v GraphQL podľa oficiálnej dokumentácie [22] "popisujú, aké údaje možno z rozhrania API získať." *Queries* slúžia na čítanie dát zo servera, teda majú podobný princíp ako "GET" pri REST API s tým rozdielom, že pri *Query* si klient špecifikuje, aké dáta potrebuje. *Mutations* sa používajú pri modifikácii dát na serveri. Sú podobné ako "POST", "PUT" a "DELETE" pri REST API, opäť si ale klient vie špecifikovať, aké dáta chce modifikovať a zároveň požiadať o návrat

<sup>3</sup>Monolitický backend je architektúra, kde je všetka logika aplikácie spojená v jednom celku.

<sup>4</sup>Domovská stránka jazyka Go: <https://go.dev/>

<sup>5</sup>Domovská stránka GraphQL: <https://graphql.org/>

<sup>6</sup>Stránka relačnej databázy PostgreSQL: <https://www.postgresql.org/>

<sup>7</sup>Stránka knižnice GORM: <https://gorm.io/index.html>

<sup>8</sup>Domovská stránka technológie RabbitMQ: <https://www.rabbitmq.com/>

dát, ktoré boli zmenené. Súbor `schema.resolvers.go` obsahuje implementácie obslužných funkcií (angl. *resolvers*), ktoré sú zodpovedné, že z GraphQL požiadaviek sa načítajú alebo upraví požadované dáta.

Vrstva **Domain Logic Layer** (DLL) obsahuje aplikačnú logiku, ktorá spracováva dáta a implementuje správu jednotlivých častí. Funkcie v rámci tejto vrstvy volajú databázové operácie a vykonávajú potrebné transformácie dát. Databázová vrstva (db) spravuje prístup k databáze, definuje databázové entity a operácie nad databázou. Vrstva **modelMapping** mapuje dáta medzi rôznymi vrstvami (napríklad databázové modely na GraphQL modely).

# Kapitola 5

## Návrh

Na základe predchádzajúcich analýz v kapitole 4, ktorých cieľom bolo preskúmať a zanalyzovať podstatné časti systému a existujúcich riešení, je v tejto kapitole popísaný návrh interpreta vizuálneho programovacieho jazyka. V rámci architektúry, ktorá vychádza z pôvodného metajazyka, je možné vytvárať aj vlastné bloky programu a to buď kompozíciou z existujúcich blokov alebo úplne nových. Dôležité je ale poznamenať, že vyhodnocovací program v prvej verzii bude schopný spracovať iba tie bloky programu, ktoré boli explicitne definované v čase jeho nasadenia. Vyhodnotenie vlastných blokov programu si vyžaduje rozšírenie aktuálneho riešenia, čo však už ale presahuje rozsah tejto práce. Pri návrhu vyhodnocovacieho programu bolo potrebné premyslieť, akým štýlom budú výsledné programy ukladané, akým spôsobom budú získavané a následne ako budú vyhodnocované. Analýzou systému RIoT 4.3 bolo zistené, že využíva technológie GraphQL spolu s relačnou databázou PostgreSQL a na komunikáciu medzi mikroslužbami využíva RabbitMQ.

Pred tým, ako sa program uloží do databázy, je potrebné ho validovať. Na základe analýzy vizuálneho programovacieho jazyka v sekcii 4.1 je nutná kontrola syntaxe, ale aj sémantiky. Kontrola syntaxe je riešená počas generovania výsledného JSON-u, program je ale možné poslať cez GraphQL API, teda spoliehanie sa na validný program z pohľadu syntaxe nie je vhodné. Sémantická analýza sa stará o sémantickú správnosť programu. Táto analýza bude musieť spolupracovať s hlavičkou programu v časti "header", keďže jej úlohou je aj kontrola správneho použitia premenných, poprípade procedúr alebo kontrola chýb pri nekompatibilitate typov.

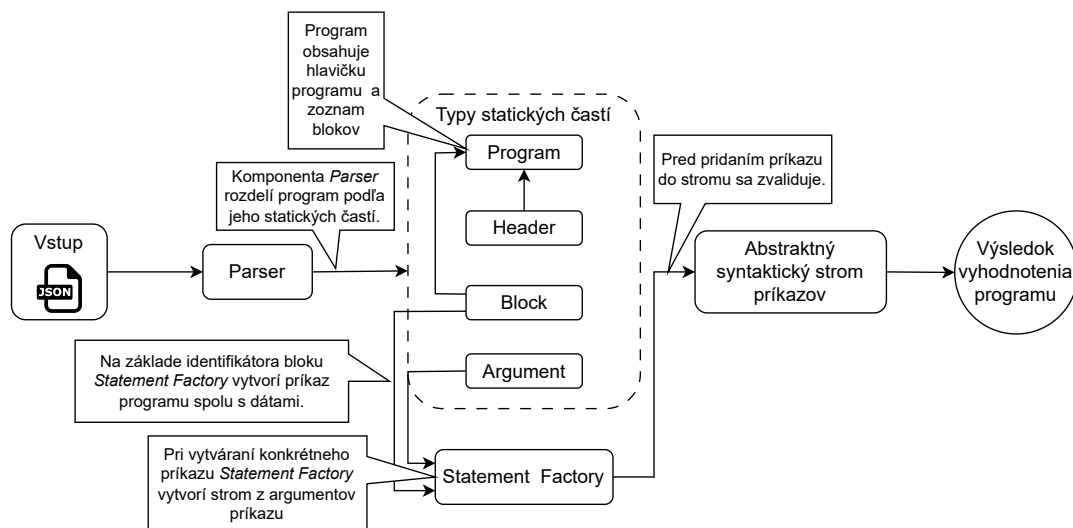
### 5.1 Štruktúra interpreta

Návrh interpreta bude spočívať v modulárnom návrhu a bude pozostávať z niekoľkých častí, ktoré budú vzájomne spolupracovať na rozdeľovaní programu na menšie časti, validácii a vyhodnotení. Obrázok 5.1 popisuje model interpreta a jeho častí. Interpret je navrhnutý tak, aby na vstup vedel prijať program vo formáte JSON, ktorý bude vstupovať do komponenty **Parser**. Táto komponenta rozdelí program na jednotlivé pevne dané časti z programu. Analýzou programu v sekcii 4.1 bolo zistené, že program sa na začiatku skladá z pevne danej hlavičky a zoznamu hlavných blokov na najvyššej vrstve, zároveň aj parameter **arguments** je pevne daný vo väčšine príkazov programu. Rozdelenie týchto štyroch typov bude vytvorené v balíku **types**. Programovací jazyk Go poskytuje podporu na rozdelenie formátu JSON do

štruktúr cez balík `encoding/json`<sup>1</sup>. Komponenta `Parser` teda rozčlení vstupný text tak, aby s ním mohli ďalšie komponenty systému efektívne pracovať.

Nasleduje fáza, kde sa na základe identifikátora každého bloku aktivuje komponenta `Statement Factory`. Táto komponenta bude mať na starosti vytvorenie konkrétneho príkazu, ktorý je reprezentovaný aj so všetkými potrebnými dátami. `Statement Factory` teda zohráva úlohu tvorcu inšancií jednotlivých typov príkazov. Štruktúry príkazov budú definované v balíku `statements` spolu s komponentou `Statement Factory`. Každý príkaz obsahuje rozdielne dáta, väčšina z nich ale obsahuje parameter `arguments`. Tieto argumenty príkazov budú transformované do stromu operandov a operátorov, prípadne iných argumentov. Pri výrazoch bude strom vytvorený tak, aby uzol reprezentoval operátor a jeho listy reprezentovali operandy. Keďže v programe je možné vytvoriť výraz pozostávajúci z  $N$  operátorov, (napríklad `if (a === b === c)`), musí sa jednať o  $n$ -árny strom argumentov.

Podmienené príkazy spolu s cyklami obsahujú telo blokov, teda po spracovaní všetkých blokov programu (vonkajších aj zavorených) sa vytvorí abstraktný syntaktický strom (angl. *AST* – *Abstract Syntax Tree*) príkazov. Strom predstavuje štruktúrovanú reprezentáciu programu, kde každý uzol zodpovedá jednému príkazu. Predtým, ako sa príkaz vloží do stromu, prebehne jeho validácia, kde sa overí jeho správnosť z hľadiska štruktúry aj obsahu. Tento krok je dôležitý pre zabezpečenie integrity programu a pre predchádzanie chybám počas jeho vyhodnocovania.



Obr. 5.1: Zjednodušený model interpretácie

Po vytvorení úplného AST nasleduje fáza vyhodnotenia programu. Interpreter prechádza syntaktickým stromom a vykonáva príkazy podľa ich poradia a významu. Výsledkom tohto procesu je finálny výstup programu, ktorý odráža správanie definované používateľom vo vstupnom texte. Výsledkom vyhodnotenia programu teda bude zoznam parametrov zariadení, ktoré daný program modifikuje, a zároveň zoznam príkazov zariadení, ktoré je nutné vykonať. Celý model je navrhnutý s dôrazom na modularitu, pričom každá komponenta má presne definovanú úlohu. `Parser` sa sústreďuje na syntaktické členenie, `Statement Factory` na vytváranie a validáciu príkazov, a AST na reprezentáciu štruktúry programu a

<sup>1</sup>Dokumentácia balíku `encoding/json`: <https://pkg.go.dev/encoding/json>

na reprezentáciu argumentov príkazov. Tento prístup umožňuje jednoduchú rozšíriteľnosť systému, napríklad o nové typy príkazov, alebo rozšírenie validácie bez potreby zásadných zásahov do ostatných častí modelu.

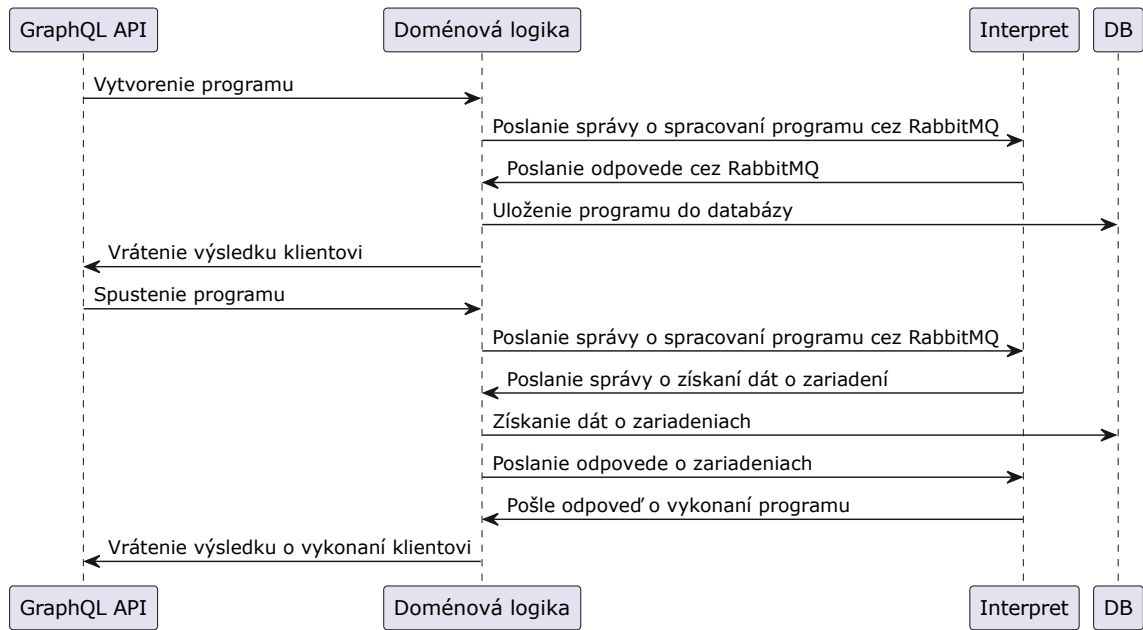
## 5.2 Návrh integrácie do systému

Po vytvorení programu pomocou vizuálneho programovacieho jazyka je výsledný program vygenerovaný vo formáte JSON. Jedná sa o program, ktorý je vytvorený užívateľom za účelom opakovaného vykonávania nad zariadeniami. Preto je nutné dané programy vhodne ukladať do databázy. Okrem tohto bolo potrebné dôsledne premyslieť, ako budú definované typy a operácie nad jazykom v GraphQL schéme. Analýzou bolo zistených niekoľko potrebných operácií nad jazykom. Jedná sa o typické CRUD operácie, teda vytvorenie programu, načítanie jedného alebo všetkých programov v databáze, úprava programu alebo jeho mazanie. Okrem toho je možné spúšťať program a to priamo príkazom alebo na základe udalosti (angl. *event-driven*). Táto architektúra zabezpečuje to, že systém bude môcť okamžite reagovať na externé zmeny. V momente, že nastane nejaká zmena na zariadení, systém bude môcť reagovať spustením príslušných programov.

Obrázok 5.2 popisuje návrh riešenia architektúry vyhodnocovacieho programu. Najjednoduchšie operácie nad jazykom sú operácie čítania a mazania programu. V tomto prípade do toku priamo nezasahuje mikroslužba interpreta ako taká, keďže sa vykonajú iba operácie priamo nad databázou. Cez GraphQL rozhranie sa spracuje konkrétna operácia a s pomocou doménovej logickej vrstvy serverovej časti systému RIoT, ktorá slúži ako prostredník medzi klientskou vrstvou a databázou, sa vykoná konkrétna akcia. Pri operácii čítania je nutné definovať dve funkcie, jedna, ktorá načíta jeden konkrétny program skrz identifikátor a druhá, ktorá načíta všetky programy z databázy, čo je použiteľné potom na klientskej časti systému na zobrazenie všetkých užívateľom vytvorených programov.

Operácie vytvárania a úpravy programu sú už o niečo komplexnejšie, je potrebné zapojiť ďalšie technológie. Pri vytvorení nového programu sa cez mutáciu musí vyvolať funkcia z doménovej logickej vrstvy, ktorá musí umožniť poslanie správy do mikroslužby vyhodnocovacieho programu tak, aby bola tá mikroslužba schopná korektne spracovať daný program. Na tento účel slúži technológia RabbitMQ, ktorej úloha je posilať správy medzi jednotlivými mikroslužbami. Táto technológia využíva komunikačný mechanizmus RPC, čo je „protokol, ktorý poskytuje vysokoúrovňovú komunikačnú paradigmu používanú v operačnom systéme. RPC predpokladá existenciu TCP alebo UDP na prenos dát správ medzi komunikujúcimi programami“ [45]. Medzi mikroslužbami teda nastane komunikácia, kde RabbitMQ klient z jednej mikroslužby pošle správu s programom na vyhodnotenie a zároveň bude čakať na odpoveď. RabbitMQ klient v druhej mikroslužbe bude podobne čakať na prijatie požiadavky na vyhodnotenie programu a pošle správu s výsledkom spracovania programu.

Vykonanie programu prebieha podobne ako jeho ukladanie, zahŕňa ale viacero krokov. Pri vyhodnotení si systém najskôr vyžiada konkrétny program z databázy a následne prostredníctvom RabbitMQ klienta odošle správu na jeho vykonanie. Hlavný rozdiel medzi ukladaním a vykonaním programu je ten, že spolu s programom sa do správy pridáva aj zoznam parametrov zariadení, ktoré daný program využíva. Celý proces potom vyžaduje implementáciu dvoch RPC mechanizmov – jeden, kde sa pošle program a súvisiace parametre do interpreta na vyhodnotenie a druhý, kde počas vyhodnotenia interpret odošle spätnú RPC požiadavku na získanie informácií o zariadeniach.



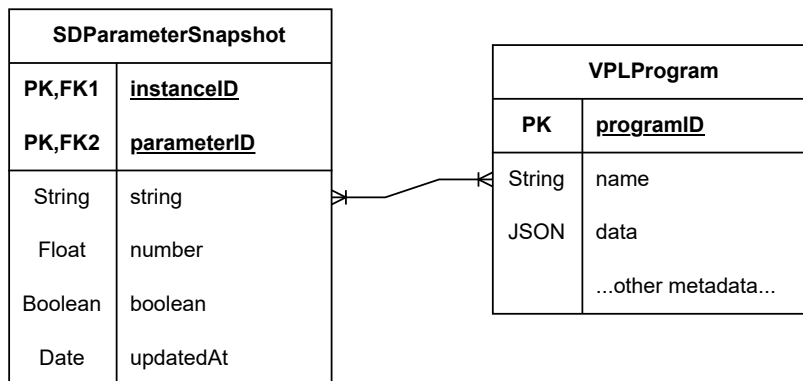
Obr. 5.2: Návrh architektúry riešenia

### 5.3 Databáza a GraphQL API

Prvotným nápadom bolo vytvoriť databázovú schému tak, aby každá časť v programe bola v databáze reprezentovaná jednou tabuľkou. Hlavička programu by sa rozdelila na dve tabuľky v databáze, kde jedna by si ukladala dáta o premenných a druhá o užívateľom definovaných procedúrach. Hlavný program by bol rozdelený do tabuľky `Block`, ktorá by obsahovala príslušné údaje o bloku program (referenciu na tabuľku argumentov alebo príkazov). Toto riešenie sa javí ako najflexibilnejšie a umožnilo by to jednoduchšiu manipuláciu nad jednotlivými časťami programu, zároveň by to zaručilo atomicitu.

Napriek spomenutým výhodám sa pri tomto riešení vyskytla aj nevýhoda, a to implementačná náročnosť. Bola by potrebná vysoká a dôkladná kontrola nad databázou pri referenciách na jednotlivé riadky v tabuľkách. Navyše možnosť sa dostať k jednotlivým častiam programu (k bloku alebo argumentu) by nebola v tejto práci využitá. Preto bola zvolená najjednoduchšia možnosť ukladania do databázy, ktorá spočíva v priamom ukladaní programu vo formáte JSON. Služba ukladajúca program by musela vedieť, aká je jeho štruktúra alebo by si to musela spravovať sama. V tomto prípade by však absentovala priama väzba medzi parametrami zariadení a ich reprezentáciou v relačnej databáze. Toto riešenie je najjednoduchšie z hľadiska implementácie, kde bude potrebné uchovávať identifikátor programu, samotný program, prípadne ďalšie atribúty, ako napríklad názov daného programu. Okrem tohto je potrebné uchovávať v databáze referencie na parametre zariadení, ktoré boli použité v programe. Počas toho, ako komponenta `Parser` bude prechádzať cez program, bude si uchovávať zoznam parametrov zariadení, ktoré sa spolu s programom následne uložia do databázy. Dôvod, prečo si uchovávať tieto parametre, je ten, že pred vyhodnotením programu je potrebné si hodnoty týchto parametrov načítať od zariadení, aby bolo možné korektné vyhodnotiť daný program. Na základe tohto budú navrhnuté GraphQL typy a API tak, aby odrážalo navrhnuté riešenie.

ER diagram 5.3 navrhuje dátový model s dvomi hlavnými entitami. Entita `VPLProgram` reprezentuje program v systéme. Má primárny kľúč, ktorý slúži ako jednoznačný identifikátor každého programu. Okrem toho obsahuje atribút `name` typu `String`, ktorý reprezentuje názov programu, atribút `data` typu `JSON`, ktorý slúži na ukladanie programu ako takého, prípadne ešte ostatné metadáta. Druhá entita, `SDParameterSnapshot`, ktorá už existuje v systéme RIoT, predstavuje snímku (angl. *snapshot*) parametrov spojených s konkrétnou inštanciou programu a konkrétnym parametrom. Má zložený primárny kľúč, okrem toho entita obsahuje atribúty rôznych typov, ktoré reprezentujú hodnotu parametra: `string`, `number`, `boolean`, a `updatedAt` (dátum poslednej aktualizácie).



Obr. 5.3: Navrhnutý ER diagram

# Kapitola 6

## Implementácia

Implementácia prebiehala v dvoch hlavných fázach. V prvej fáze bol vytvorený samotný interpret a je dostupný v GitHub repozitári<sup>1</sup> pod organizáciou Pocketix. Druhá fáza pozostávala z integrovania interpreta do systému RIOT. Na implementáciu interpreta bol použitý programovací jazyk Go, využívajúci vstavný balík `encoding/json` na spracovanie programu vstupujúceho do interpreta vo formáte JSON. Samotný návrh a logické rozdelenie balíkov bolo inšpirované existujúcim riešením pôvodného interpreta pre prvú verziu vizuálneho programovacieho jazyka. Interpret program spracováva v niekoľkých krokoch, v prvom rozdelí program do statických štruktúr. V druhom kroku zvaliduje a zostaví abstraktný syntaktický strom príkazov a v poslednom kroku daný strom vyhodnotí.

### 6.1 Implementácia interpreta

Na základe návrhu v sekcii 5 boli jednotlivé časti aplikácie implementované ako samostatné balíky. Každý balík predstavuje ucelenú funkcionality a spoločne tvoria logicky štruktúrovaný celok v repozitári projektu:

- `models` – balík definuje dátové štruktúry a modely používané v aplikácii, menovite implementuje úložisko premenných alebo procedúr alebo strom argumentov pre príkazy. Obsahuje logiku a manipuláciu nad týmito modelmi, napríklad pridávanie referencovaných hodnôt alebo získavanie informácií o zariadeniach.
- `statements` – obsahuje definície a implementácie rôznych typov príkazov a ich spracovanie. Okrem konkrétnych príkazov vyskytujúcich sa v programe obsahuje továreň `statementFactory`, ktorá vytvára objekty príkazov a rozhranie `statementInterface` reprezentujúce všeobecne príkaz programu.
- `parser` – stará sa o spracovanie JSON vstupu do aplikácie. Rozdeľuje vstupný program do interných statických štruktúr, definovaných v balíku `types`. Stará sa aj o spracovanie hlavičky programu, kde spracováva premenné a procedúry.
- `types` – definuje základné typy programu, ako sú napríklad `Program` alebo `Block`, ktoré reprezentujú základnú štruktúru programu. Používajú sa na spracovanie vstupného programu.
- `services`, `utils` – obsahuje rôzne pomocné služby, ako napríklad načítanie programu zo súboru.

---

<sup>1</sup><https://github.com/pocketix/pocketix-go>

## Spracovanie programu

Predtým, ako sa program spracuje, bolo potrebné nadefinovať štruktúry, na ktoré sa bude mapovať program pomocou balíka `encoding/json`. Definované boli tie štruktúry, od ktorých sa presne dá odvodiť kľúč a hodnoty, teda tie, ktoré sú statické. Kľúče (napríklad premenné alebo procedúry), pri ktorých nie je možné presne určiť, ako budú vyzeráť, sú riešené dynamicky počas behu spracovania. Výpis 6.1 popisuje vytvorené štruktúry `Program` a `Block`. Štruktúra `Program` rozdeľuje hlavný program na dve časti, jedným je hlavička a druhým je zoznam blokov na najvyššej vrstve. `Block` štruktúra obsahuje svoj identifikátor, svoje telo, argumenty a voliteľný zoznam zariadení. Zoznam zariadení je pridaný ako rozšírenie jazyka, kde je možné si nadefinovať všeobecnú procedúru. Toto rozšírenie bolo implementované v rámci bakalárskej práce Ivana Chodáka [15], ktorá bude vykonávať akciu nad zariadeniami. Na rozdiel od pôvodných procedúr, tieto nové procedúry fungujú ako funkcie v bežných programovacích jazykoch s parametrami.

```
type Program struct {
    Header Header `json:"header"`
    Blocks []Block `json:"block"`
}

type Block struct {
    Id      string      `json:"id"`
    Body   []Block    `json:"block"`
    Arguments []Argument `json:"arguments"`
    Devices []Device   `json:"devices"`
}
```

Výpis 6.1: Definícia štruktúr `Program` a `Block`

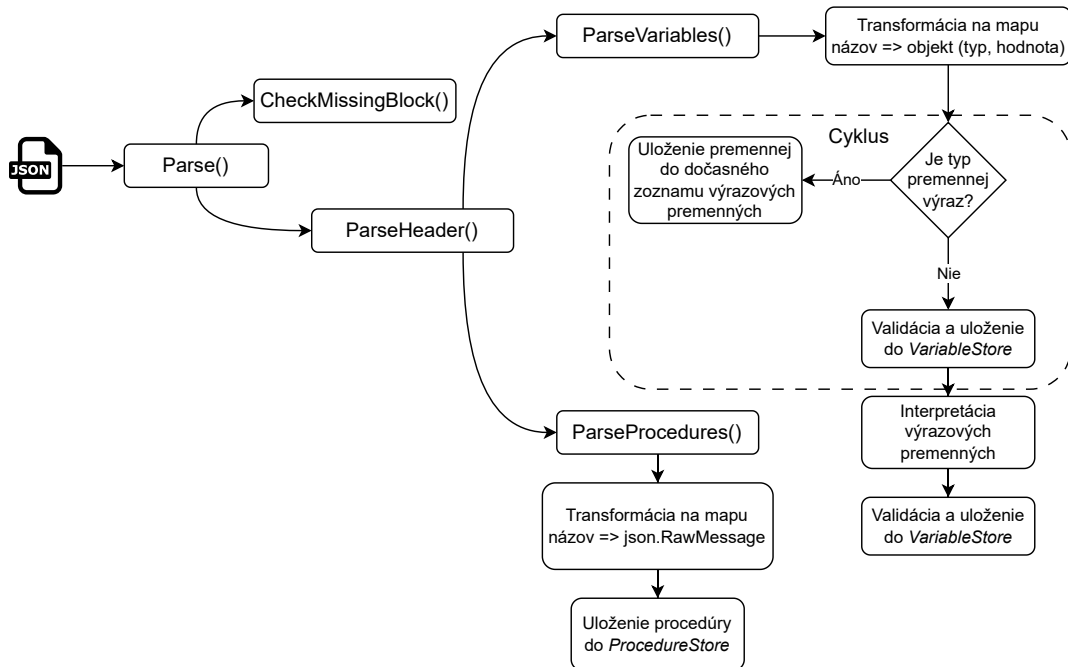
Prvým hlavným bodom pri spracovaní programu je funkcia `Parse` 6.2. Vstupom do funkcie sú dáta programu, rôzne úložiská použité pri spracovávaní a rozhranie `Collector`, ktoré vytvára abstraktný syntaktický strom z príkazov. Funkcia `Parse` teda slúži na prevod JSON formátu na internú reprezentáciu programu do AST štruktúry. Celý proces tvorby prebieha v niekoľkých logických krokoch, ktoré zabezpečujú správne načítanie, validáciu a štruktúrovanie programu.

```
func Parse(
    data []byte,
    variableStore *models.VariableStore,
    procedureStore *models.ProcedureStore,
    referencedValueStore *models.ReferencedValueStore,
    collector statements.Collector,
) error {}
```

Výpis 6.2: Deklarácia funkcie `Parse`

Diagram 6.1 popisuje proces spracovania hlavičky programu. Hneď v úvode sa skontroluje program pomocou funkcie `CheckMissingBlock`, ktorá kontroluje, či program obsahuje povinné časti, konkrétne kľúče `header` a `block`. V prípade, že niektorý z týchto základných prvkov programu chýba, funkcia vráti chybu. Týmto krokom je zabezpečené, že vstupný program má minimálnu požadovanú štruktúru, aby s ním bolo možné ďalej pracovať. Na-

sleduje spracovanie hlavičky programu pomocou funkcie `ParseHeader`. Táto funkcia deserializuje celý JSON vstup do štruktúry `Program`, ktorá reprezentuje program ako celok. V rámci tejto funkcie sa spracúvajú premenné a procedúry vo funkciách `ParseVariables` a `ParseProcedures`. Uloženie premenných je jednou z kľúčových častí spracovania programu.



Obr. 6.1: Diagram spracovania hlavičky programu

```
func ParseVariables(
    data json.RawMessage,
    variableStore *models.VariableStore,
    referencedValueStore *models.ReferencedValueStore
) error {}
```

Výpis 6.3: Deklarácia funkcie `ParseVariables`

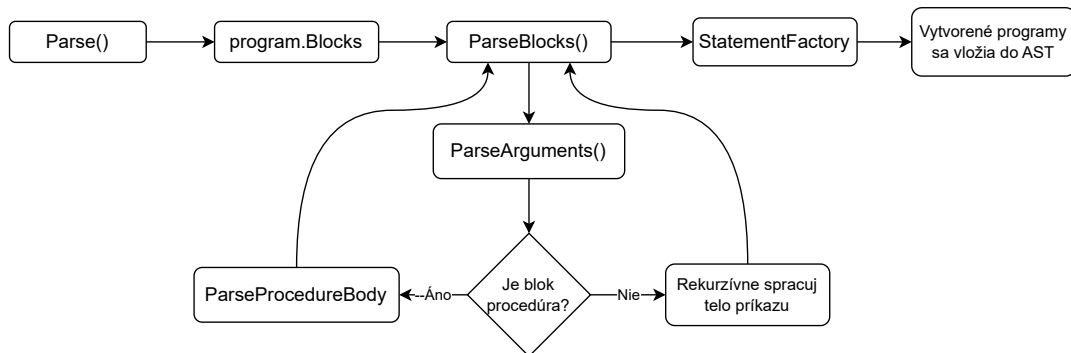
Funkcia `ParseVariables` 6.3 je zodpovedná za načítanie, validáciu a uloženie premenných do centrálného úložiska `VariableStore`. Hlavným cieľom je teda zabezpečiť, aby všetky premenné boli správne interpretované ešte predtým, ako bude spracovaný hlavný program. Na začiatku sa všetky premenné zo vstupných dát transformujú na mapu, kde kľúč predstavuje názov premennej a hodnota predstavuje štruktúrovaný objekt, ktorý obsahuje typ a hodnotu. Dôležité je rozoznať, či sa jedná o obyčajnú premennú alebo o výraz. Premenné typu výraz môžu obsahovať referencie na iné jednoduché premenné (napríklad tak, aby bolo možné vytvoriť príkaz z typických programovacích jazykov `i += 1`). Tým, že premenné sú uložené v mape, cyklenie cez túto mapu sa správa nedeterministicky a môže nastať situácia, kde sa premenné nespracujú v poradí, v akom boli definované. Preto výrazové premenné nie sú počas spracovávania interpretované hneď, ale sú uložené do dočasného zoznamu a interpretujú sa až po tom, čo sa spracujú všetky jednoduché premenné. Funkcia `ParseProcedures` funguje obdobne s tým rozdielom, že sa do úložiska uloží ako kľúč názov

premennej a telo s typom `json.RawMessage`. Telo procedúry sa spracuje až v momente, čo sa daná procedúra v programe má vyhodnotiť. Predíde sa teda spracovaniu tela procedúry aj v prípade, že nie je v programe použitá.

Po úspešnom spracovaní hlavičky nasleduje spracovanie hlavného programu, ktoré je popísané v diagrame 6.2. Cyklus vo funkcii `Parse` prechádza všetky bloky definované v `programs.Blocks` a pre každý jeden z nich zavolá funkciu `ParseBlocks` 6.4, ktorá spracuje daný blok programu. Funkcia vráti v prípade korektného spracovania bloku zoznam konkrétnych príkazov, ktoré sa vložia do stromu, prípadne chybu. Spracovanie konkrétneho bloku na začiatku prebieha spracovaním argumentov pomocou funkcie `ParseArguments`, ktorá vytvára stromové štruktúry, ktoré reprezentujú argumenty príkazov. Následne funkcia prechádza vnútorné bloky príkazu daného bloku, ak nejaké existujú. Rekurzívne sa nad týmito blokmi zavolá funkcia `ParseBlocks` a tým vytvára AST príkazov.

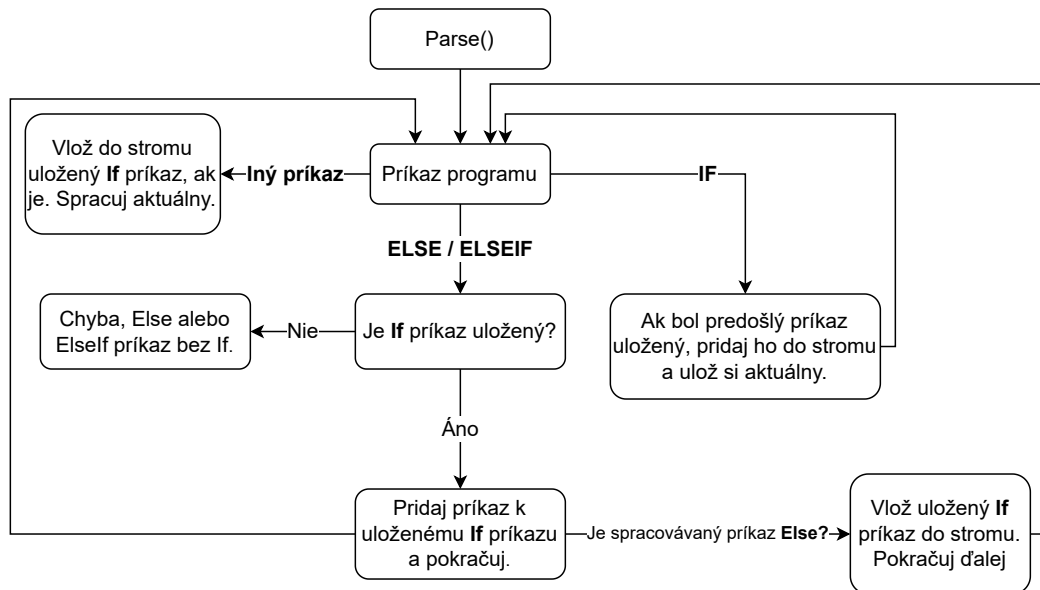
```
func ParseBlocks(
    block types.Block,
    variableStore *models.VariableStore,
    procedureStore *models.ProcedureStore,
    referencedValueStore *models.ReferencedValueStore,
    collector statements.Collector,
) ([]statements.Statement, error) {}
```

Výpis 6.4: Deklarácia funkcie `ParseBlocks` spracujúca blok programu



Obr. 6.2: Diagram spracovania tela programu

Špeciálnym prípadom pri spracovávaní príkazov je spracovanie vetiev príkazov, ako sú `If`, `Elseif` a `Else`. Diagram 6.3 popisuje tok spracovania podmieneného príkazu. Pri spracovaní príkazu sa skontroluje jeho identifikátor. Ak sa jedná o podmienený príkaz, uloží sa do dočasnej premennej, aby sa do nej mohli pridávať prípadné ostatné vetvy. Ak ďalší spracovávaný príkaz je vetva `Elseif`, pridá sa k uchovanému podmienenému príkazu a pokračuje ďalej. Ak ďalší príkaz je vetva `Else`, pridá sa k uloženému podmienenému príkazu, uloží sa do stromu a dočasná premenná pre podmienený príkaz sa vynuluje. V prípade, že v dočasnej premennej nie je uložený podmienený príkaz a spracováva sa príkaz `Elseif` alebo `Else`, program sa vyhodnotí ako neplatný. Ak sa spracováva blok, ktorý nepatrí k podmienenému príkazu a jeho vetvám, dočasne uložený podmienený príkaz sa uloží do stromu, ak bol nastavený, a spracuje sa aktuálny príkaz.



Obr. 6.3: Diagram logiky spracovania podmieneného príkazu

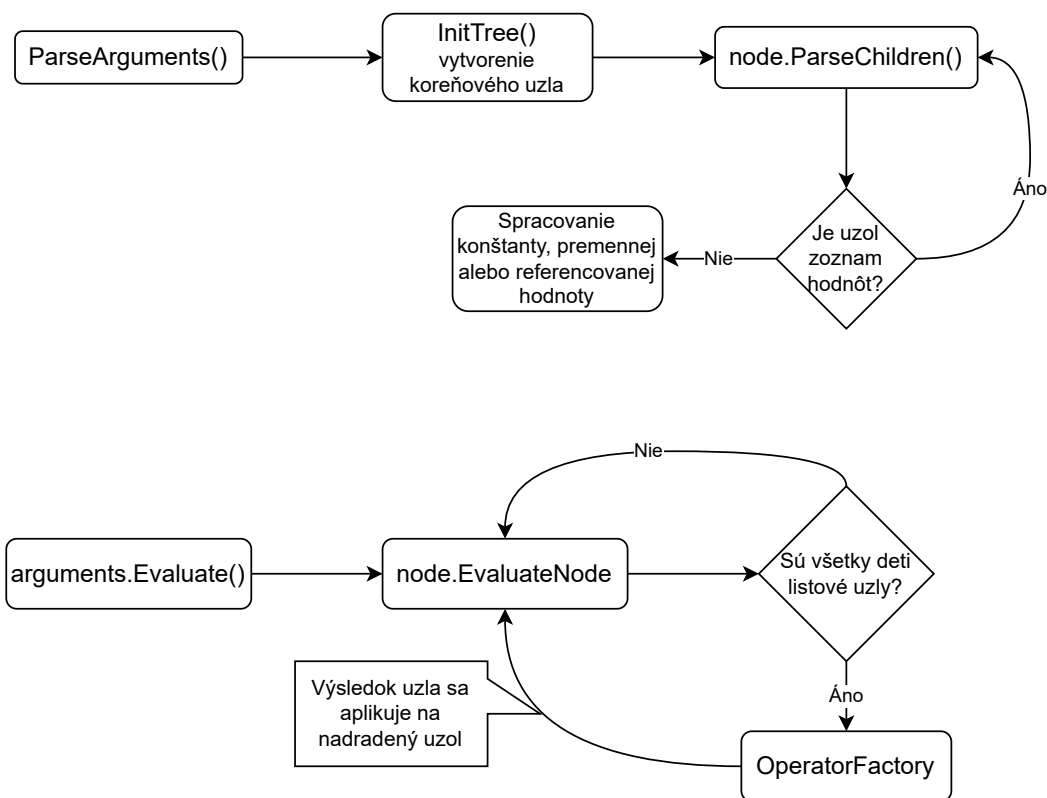
Ak funkcia `ParseBlocks` natrafi na procedúru, získa si ju z globálneho úložiska a spracuje jej telo rovnako pomocou funkcie `ParseProcedureBody`, ktorá funguje na rovnakom princípe ako spracovávanie hlavného programu. Následne, na základe spracovaného tela programu a argumentov, sa pomocou `StatementFactory` vytvorí konkrétna inštancia príkazu. Inštancie príkazov sú definované v balíku `statements` (výpis 6.5 definuje štruktúru inštancie podmieneného príkazu). Každá inštancia implementuje rozhranie `Statement`, ktoré deklaruje základné metódy `Execute`, `GetId` a `Validate` použité pri spracovávaní programu.

```

type If struct {
    Id          string
    Block       []Statement
    Arguments   *models.TreeNode
    IfElseBlocks []ElseIf
    ElseBlock   Else
}
  
```

Výpis 6.5: Definícia štruktúry inštancie podmieneného príkazu

Spracovanie argumentov je riešené podobným spôsobom, popísaným v diagrame 6.4. Funkcia `ParseArguments` z argumentov zostaví strom z argumentov príkazu. `InitTree` funkcia má na starosti inicializáciu daného stromu a vytvorenie stromovej štruktúry. Na začiatku vytvorí koreňový uzol a následne spracuje ostatné uzly rekurzívne pomocou funkcie `ParseChildren`. Ak argumenty nie sú zoznamom, overí sa ich typ a hodnota sa uloží priamo do uzla ako list stromu. V prípade zoznamu sa iteruje cez všetky prvky a pre každý sa vytvára nový poduzol, pričom sa rozlišuje, či ide o jednoduchý typ alebo zložitejší výraz. Ak sa v uzle nachádza premenná, program sa pokúsi vytvoriť spočiatku `ReferencedValue` a ak toto zlyhá, pokúsi sa získať premennú v úložisku premenných. V aktuálnej verzii jazyka nie je možné rozlíšiť, či sa v prípade premennej jedná o parameter zariadenia alebo premennú, keďže oba majú v programe typ `variable`. Implementované riešenie teda uprednostňuje



Obr. 6.4: Diagram spracovania stromu argumentov a vyhodnotenie

referencované hodnoty pred obyčajnými premennými a momentálne je možné ich rozlíšiť tým, že referencované hodnoty obsahujú v identifikátore znak bodky (.).

Vyhodnocovanie výrazu sa začína metódou `Evaluate`, ktorá zavolá rekurzívnu metódu `EvaluateNode`. Táto funkcia vyhodnocuje uzly od spodku stromu smerom nahor. Ak má uzol poduzly, najprv sa vyhodnotia tie. Potom sa výsledok aplikuje v rámci aktuálneho uzla, najmä v prípade operátorov a výrazov. Ak má uzol iba jeden poduzol, výpočet sa presúva do neho, čo umožňuje spracovanie jednočlenných výrazov. Všetky výpočty prebiehajú v spolupráci s objektom `OperatorFactory`, ktorý je zodpovedný za overenie operátorov a vykonanie príslušných operácií podľa hodnoty a typu uzla.

Spomínané rozhranie `Collector` vo výpise 6.6 definuje dva typy. Prvý typ tohto rozhrania je `NoOpCollector`, ktorý má v tele zoznam príkazov. Táto štruktúra sa využíva na spracovanie programu vtedy, keď nie je nutné zostaviť abstraktný syntaktický strom, ale je potrebné spraviť iba validáciu daného programu. Zoznam príkazov v tele je použitý pri kontrole `switch` príkazu, kde je nutné si uchovávať rodičovský príkaz `switch`, aby bolo možné kontrolovať `case` príkazy, či majú správnu hodnotu na základe hodnoty v argumente `switch` príkazu. Druhým typom je `ASTCollector`, ktorý obsahuje strom príkazov, ktorý postupne pomocou funkcie `Collect` vytvára. Vždy, keď sa spracuje príkaz, sa zavolá táto funkcia a pridá sa do stromu. Následne po spracovaní všetkých príkazov je možné abstraktný syntaktický strom vyhodnotiť. Vyhodnotenie prebieha v cykle, kde sa postupne prejde celý strom príkazov a nad každým príkazom sa zavolá funkcia `Execute`.

```

type Collector interface {
    NewCollectorBasedOnType(
        collectorType Collector,
        target *[]Statement
    ) Collector
    Collect(statement Statement)
    Type() Collector
    GetTarget() *[]Statement
}

```

Výpis 6.6: Definícia rozhrania Collector

## 6.2 Integrácia do systému RIoT

Aby bolo možné integrovať interpret do systému RIoT, bolo nutné naimplementovať novú mikroslužbu v serverovej časti systému. Služba `interpret-unit` má na starosti spracovanie správ, ktoré posiela mikroslužba systému `backend-core` a využívať aplikačné rozhranie naimplementovaného interpreta.

### GraphQL API a databázová vrstva

Na základe návrhu boli vytvorené *queries* a *mutácie* v GraphQL schéme. Vytvorené boli typy nad programom a najzákladnejšie CRUD operácie spolu s mutáciou, ktorá vykonáva konkrétny program, ako je ukázané vo výpise 6.7. Typ `VPLProgram` definuje konkrétny program, ktorý obsahuje svoj identifikátor, meno, dáta vo formáte JSON a ostatné metadáta. Obsahuje tiež zoznam snímkov parametrov zariadení, využiteľné pri *event-driven* prístupe. Návratový typ `VPLProgramExecutionResult` definuje hodnotu, ktorá sa vráti z vykonávacej mutácie. Okrem programu, snímkov parametrov, ktoré boli zmenené a zoznamu príkazov zariadení, ktoré sa majú vykonať, sa vráti z mutácie aj ostatné dôležité metadáta.

```

type VPLProgram {
    id: ID!
    name: String!
    data: JSON!
    lastRun: Date
    enabled: Boolean!
    sdParameterSnapshots: [SDParameterSnapshot!]!
}

type VPLProgramExecutionResult {
    program: VPLProgram!
    sdParameterSnapshotsToUpdate: [SDParameterSnapshot!]!
    SdCommandInvocations: [SDCommandInvocation!]!
    executionTime: Date!
    enabled: Boolean!
    success: Boolean!
    error: String
    executionReason: String
}

```

```

}

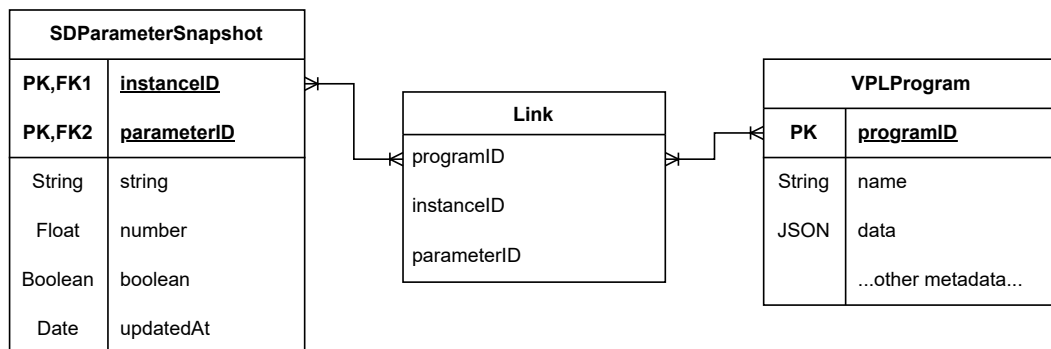
type Query {
  vplPrograms: [VPLProgram!]!
  vplProgram(id: ID!): VPLProgram!
}

type Mutation {
  createVPLProgram(name: String!, data: JSON!): VPLProgram!
  updateVPLProgram(id: ID!, name: String!, data: JSON!): VPLProgram!
  deleteVPLProgram(id: ID!): Boolean!
  executeVPLProgram(id: ID!): VPLProgramExecutionResult!
}

```

Výpis 6.7: Vytvorená typy a operácie nad programom v GraphQL schéme

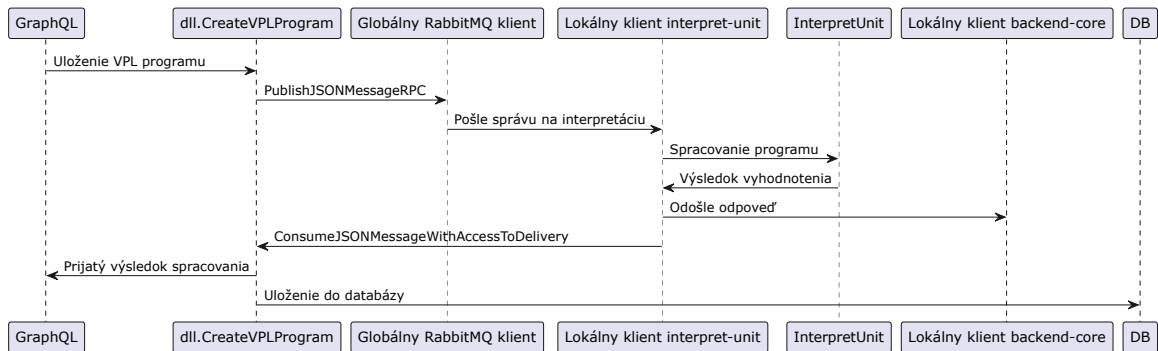
Na databázovej vrstve bolo potrebné definovať štruktúry, ktoré budú mapovať pomocou balíku `gorm` štruktúry na relačné objekty. ER diagram na obrázku 6.5 popisuje vytvorené štruktúry. Diagram popisuje M:N vzťah medzi snímkami parametrov zariadení a programami. Program môže používať viacero parametrov zariadení a zároveň sa jeden parameter môže vyskytovať vo viacerých programoch. Okrem toho bolo potrebné vytvoriť objekty na doménovej logickej vrstve a zaručiť mapovanie medzi rôznymi vrstvami (napríklad z databázovej vrstvy na vrstvu doménovej logickej vrstvy).



Obr. 6.5: ER diagram popisujúci databázové objekty a vzťahy

## Komunikácia medzi službami

Komunikácia medzi jednotlivými mikroslužbami využíva technológiu RabbitMQ a viacfázovú komunikáciu medzi službami `backend-core` a `interpret-unit`. Tento proces využíva architektúru RPC (Remote Procedure Call), čo umožňuje oddelenie aplikačnej logiky od logiky spracovania programov. Aby posielanie správ fungovalo, bolo potrebné definovať fronty, na ktorých budú mikroslužby počúvať. Tie sa vytvoria v zdieľaných konštantách v mikroslužbe `backend-core`, napríklad na ukladanie programu sa vytvoria dve fronty, jedna na požiadanie, aby sa uložil program a druhá, ktorá bude očakávať výsledky od interpreta. Okrem tohto bolo potrebné dané fronty registrovať pri nastavovaní RabbitMQ klienta.



Obr. 6.6: Tok správ RPC mechanizmu pri ukladaní programu

Tok ukladania programu, popísaný v obrázku 6.6, začína v službe `backend-core`, kde prostredníctvom GraphQL mutácie na uloženie programu sa zavolá príslušná funkcia, ktorá spracuje požiadavok (angl. *resolver*). Táto funkcia si zavolá z doménovej logickej vrstvy funkciu, ktorá má na starosti RPC architektúru. Funkcia `CreateVPLProgram` si získa jedného globálneho RabbitMQ klienta, ktorého úloha je odosielať správy do iných služieb a druhého lokálneho klienta, ktorého úloha je počúvať na prichádzajúcu odpoveď. Vstup je spracovaný do JSON formátu a poslaný do druhej mikroslužby pomocou publikovacej funkcie `PublishJSONMessageRPC`. Po publikovaní správy začne služba prostredníctvom svojej čakacej služby počúvať na odpoveď z interpretovacej služby. Posielanie správy aj čakanie sú napojené na prislúchajúce fronty.

Služba `interpret-unit` funguje na podobnom princípe. Služba si už ale vytvára iba jedného RabbitMQ klienta. Ten má na starosti čakanie na správu, kde telo je program, nad ktorým sa má vykonať akcia. Aby sa program spracoval, služba si vytvorí potrebné objekty a začne spracovávať program, ako je ukázané na výpise 6.8. Spracovávacia funkcia pri ukladaní alebo editovaní programu využíva `NoOpCollector`, ktorý zbiera iba niektoré určité príkazy z programom, aby bola zabezpečená validita programu, ale nevytvára abstraktný syntaktický strom príkazov, ktorý nie je vyslovene nutný pri ukladaní. Výsledkom tohto spracovania je buď chyba, ktorá nastala počas vykonávania programu, alebo korektne spracovaný program spolu s referencovanými hodnotami parametrov zariadení. Tieto dáta sa pošlú späť cez RabbitMQ klienta ako odpoveď na vyhodnotenie do `backend-core`, kde táto služba spracuje odpoveď.

```

variableStore := models.NewVariableStore()
procedureStore := models.NewProcedureStore()
referencedValueStore := models.NewReferencedValueStore()
err := parser.Parse(
    []byte(messagePayload.Data),
    variableStore, procedureStore,
    referencedValueStore,
    &statements.NoOpCollector{}
)

```

Výpis 6.8: Definovanie objektov potrebných na spracovanie programov

```

outputChannel := make(chan sharedUtils.Result[sharedModel.VPLProgram])
go func() {
// Consuming JSON message from interpret-unit
    if error {
        outputChannel <- sharedUtils.NewFailureResult [] (error)
    } else {
        outputChannel <- sharedUtils.NewSuccessResult [] (program)
    }
    close(outputChannel)
}
result := <-outputChannel
return result

```

Výpis 6.9: Pseudokód s Go rutinou a kanálom

Prijatie správy v `backend-core` je vytvorené ako Go rutina, teda funkcia, ktorá sa spúšťa súčasne s hlavným vláknom programu. Podľa Go dokumentácie rutina je ľahké vlákno, ktoré je spravované Go *runtime*-om [21]. Na to, aby bolo možné dostať dáta z rutiny do hlavného vlákna, sa používajú kanály. Kanály sú typizované štruktúry, cez ktoré je možné posielat a prijímať hodnoty. To umožňuje synchronizáciu rutinám bez explicitných zámkov alebo podmienkových premenných [14]. Výpis 6.9 popisuje spracovanie správy a zaslanie dát z rutiny do hlavného vlákna. Na začiatku pred rutinou sa inicializuje kanál príslušného typu a v rutine po získaní výsledku od `interpret-unit` sa na základe vráteného výsledku uloží do kanála príslušná hodnota. Po spracovaní sa daný kanál zatvorí a na konci sa hodnoty z kanála vrátia správnym mapovaním ku klientovi.

Po spracovaní a prijatí správ sa prípadný správne spracovaný program uloží do databázy cez pripravené rozhranie serverovej časti systému. Nutné bolo vytvoriť funkciu v databázovom klientovi `PersistVPLProgram`, kde bolo potrebné si namapovať model z doménovej logickej vrstvy do modelu databázovej vrstvy a s pomocou databázového rozhrania ho správne uložiť. Mapovací proces sa zopakoval tentokrát opačným smerom a po úspešnom uložení do databázy sa vytvoril GraphQL model a ten sa vrátil naspäť ku klientovi.

Samotné vyhodnotenie programu prebieha v podobnom toku správ. Jeden z podstatných rozdielov je ten, že vykonanie programu využíva dva RPC mechanizmy. Prvý je rovnaký ako pri uložení programu, teda zaslanie správy, aby sa vykonal daný program a čakanie na odpoveď o úspešnom alebo neúspešnom vykonaní. Druhý RPC mechanizmus spočíva v načítaní dát o parametroch a príkazoch zariadení. Pri spracovávaní programu v službe `interpret-unit` v prípade, že program narazí na parameter zariadenia alebo príkaz, tak si zašle správu, že potrebuje získať dáta o tomto zariadení (angl. *lazy loading*). Riešením by mohlo byť aj získanie dát o zariadeniach predtým, ako sa zašle správa o vykonaní programu. Nevýhodou ale tohto riešenia je tá, že by sa z jednej služby do druhej posielala potenciálne príliš veľká správa s programom a informáciami o zariadeniach, kde by nebolo zaručené, že reálne všetky parametre a príkazy zariadení by boli pri spracovaní programu použité. Výsledkom vyhodnotenia mali byť príkazy, ktoré sa majú vykonať, a zoznam parametrov, ktoré sa zmenia. Pôvodný jazyk mal na toto pripravený príkaz `Write`, ktorý zapíše hodnotu do konkrétnej referencie. Aktuálna verzia jazyka ale nemá takýto príkaz. Z príkazu zariadenia nie je možné získať parameter, ktorý daný príkaz bude upravovať. Preto výsledok vyhodnotenia bude obsahovať iba program samotný a zoznam príkazov, ktoré sa vykonajú.

## Kapitola 7

# Testovanie

Testovanie interpreta prebiehalo pomocou jednotkových testov. Tieto testy pokrývajú rôzne časti správania aplikácie, od validácie vstupov až po vykonávanie jednotlivých blokov programu. Testy sú implementované taktiež v jazyku Go a využívajú knižnicu `testify`<sup>1</sup> na asercie. Testy sú rozdelené do viacerých súborov na základe testovacích funkcií alebo objektov. Každý príkaz programu obsahuje svoj vlastný súbor, kde sa testuje jeho očakávané správanie a chybné štruktúry. Príklad súborov, ktoré testujú určité funkcie alebo objekty v programe:

- **alert\_test.go** – Testuje správanie príkazu `Alert`, vrátane rôznych metód odosielania správ.
- **operator\_test.go** – Testuje rôzne operátory (napríklad `===`, `!==`, `<`, `>`, `+`, `-`), ich správne vyhodnocovanie a zároveň aj nesprávne typy operandov.
- **validProgram\_test.go** – Testuje validitu rôznych blokov programu (`if`, `while`, `switch`) a ich správne spracovanie do interných štruktúr.
- **complexPrograms\_test.go** – Testuje komplexné programy, ktoré kombinujú viacero blokov a spolu s referencovanými hodnotami parametrov.

Na zabezpečenie konzistencie a automatického overovania funkcionality po každej zmene v kóde bol projekt integrovaný s GitHub Actions<sup>2</sup>. V rámci CI (angl. *Continuous Integration*) procesu sú po každom *push*-nutí do hlavnej vetvy automaticky spúšťané všetky jednotkové testy. Tento mechanizmus pomáha okamžite identifikovať chyby spôsobené novými zmenami a výrazne prispieva k udržiavaniu spoľahlivosti celého systému. V projekte je definovaných vyše 70 testov, ktoré pokrývajú väčšinu základnej aj hraničnej logiky interpreta. Pokrytie kódu jednotkovými testami sa pravidelne meria pomocou nástroja `go test -cover`. Priemerné pokrytie kódu sa pohybuje okolo 80%. Výsledky pokrytia sú taktiež generované pri každom spustení *CI pipeline* cez GitHub Actions.

Okrem jednotkových testov počas testovania správnosti interpreta bol vytvorený v koreňovom adresári priečinkov `programs/`, ktorý obsahuje sadu najzákladnejších programov, ktoré využívajú jednotlivé príkazy programu. Interpret bol pre testovacie účely spúšťaný nad týmito programami za účelom korektného spracovania programov mimo testov. Za týmto účelom bol doprogramovaný argument v príkazovom riadku k interpretovi, ktorý určuje

<sup>1</sup><https://github.com/stretchr/testify>

<sup>2</sup><https://github.com/features/actions>

cestu k \*.json súboru s programom. Toto spúšťanie sa priamo viazalo na krokovanie programu (angl. *debugging*) priamo v programovacom IDE, ktoré veľmi uľahčilo programovanie spracovávacej aplikácie a odchyťovanie chýb počas vývoja.

## 7.1 Navrhované rozšírenie

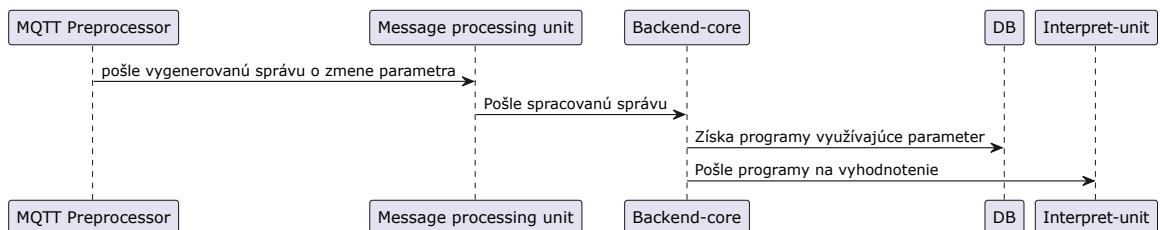
Okrem štandardného spúšťania programov ručne, na vyžiadanie, cez GraphQL API žiada aj automatickejší spôsob vyhodnotenia. Tento spôsob zahŕňa spustenie programu na základe nejakého podnetu (angl. *event-driven*). Teda napríklad pri zmene parametra nejakého zariadenia, ktoré je zahrnuté v programoch. Tento prístup je z užívateľského hľadiska oveľa viac priateľský, kde automatické spúšťanie programu na základe nejakej zmeny sa javí ako lepšia alternatíva oproti manuálnemu spúšťaniu.

Toto rozšírenie by bolo potrebné integrovať do serverovej časti systému. Tok správ medzi jednotlivými službami je popísaný v diagrame 7.1. V aktuálnej verzii v databáze existuje vzťah M:N medzi snímkami parametrov zariadení a programami. To teda umožňuje, ktoré programy môžu byť ovplyvnené zmenou hodnoty niektorého z parametrov zariadenia. Zmeny hodnôt parametrov sa do systému dostávajú cez MQTT správy, ktoré sú spracované službou MQTT Preprocessor. Táto služba generuje správy o zmene parametra, ktoré sú ďalej zasielané do Message Processing Unit. Toto posielanie správ a zisťovanie, ktoré parametre sa zmenili, by sa dalo využiť na zistenie, ktoré programy využívajú daný zmenený parameter. Vďaka vzťahu M:N sa dajú jednoducho získať programy z databázy a vykonať by sa rovnaký princíp posielania správ, ako pri manuálnom spúšťaní programu.

Pôvodný jazyk obsahoval príkaz `write`, ktorého úloha bola zapisovať hodnotu do konkrétneho parametra zariadenia. Tento príkaz v aktuálnej verzii jazyka a v aktuálnej verzii editoru ale chýba, preto nie je možné pri vykonaní programu vrátiť spolu so zoznamom zariadení aj hodnoty parametrov, ktoré sa menia. Aby bolo možné tento príkaz spracovať, je nutné pridať do balíka `statements` nový príkaz `write` s referenciou a hodnotou. Prípadné riešenie vo výpise 7.1 by mohlo byť prídanie nového voliteľného kľúča k príkazu zariadenia `reference`, ktorý by značil, ktorý parameter zariadenia daný príkaz nastavuje. Štruktúru `Block` v balíku `types` by bolo potrebné rozšíriť o novú položku, ktorá by si uchovávala práve tieto referencie. Tento koncept by mohol byť aplikovaný tak, že referencia bude voliteľná na základe schopností digitálneho dvojčata systému, ktorý tento príkaz integruje. Týmto by sa umožnilo prispôbenie spracovania príkazov konkrétnemu systému a jeho implementácii.

```
"id": "TemperatureDevice-1.setTemperature",
"reference": "TemperatureDevice-1.temperature"
"arguments": [{"type": "str_opt", "value": "ideal"}]
```

Výpis 7.1: Navrhované rozšírenie príkazu zariadenia v JSON



Obr. 7.1: Diagram posielania správ pri zmene hodnoty parametra

# Kapitola 8

## Záver

Cieľom tejto bakalárskej práce bolo vytvoriť vyhodnocovací program, ktorý dokáže validovať a vyhodnotiť vizuálny program vytvorený používateľom, a zároveň toto riešenie integrovať do existujúceho systému RIoT. V práci boli podrobne analyzované existujúce technológie systému RIoT a vizuálny programovací jazyk, na základe čoho bol následne navrhnutý a implementovaný samotný interpret. Ten pozostáva z viacerých modulov, ktoré zabezpečujú validáciu programu, vyhodnocovanie inštrukcií a správne narábanie so štruktúrovanými dátami. Práca prináša nový prístup k spracovaniu vizuálnych programov v IoT prostredí, čím zvyšuje možnosti automatizácie a používateľskej konfigurovateľnosti systémov bez nutnosti textového programovania. Tým prispieva k zjednodušeniu práce s IoT zariadeniami najmä pre neškolených programátorov a podporuje princípy vytvárania programom koncovým užívateľom (angl. *end-user development*).

K dosiahnutiu cieľu tejto práce bolo potrebné podrobne preskúmať oblasť internetu vecí a princípy vizuálneho programovania so zameraním na end-user development (EUD). Ďalej nasledovala analýza požiadaviek na spracovanie vizuálnych programov a existujúcej architektúry systému RIoT, aktuálnej verzie vizuálneho programovacieho jazyka, ako aj samotného pôvodného interpreta, ktorý bolo potrebné prepracovať a nahradiť robustnejším riešením. Na základe tejto analýzy bol navrhnutý nový serverový interpret v jazyku Go, ktorý pozostáva z viacerých modulov zodpovedných za kontrolu štruktúry, validáciu logiky a vyhodnocovanie programu. V systéme RIoT bolo potrebné vytvoriť GraphQL API a objekty na databázovej vrstve a zabezpečiť správnu komunikáciu medzi službami pomocou RPC mechanizmov.

Pri ďalšom vývoji interpreta a jeho integrácie do systému by bolo vhodné pridať rozšírenie, ktoré umožní spúšťať programy nie len na základe manuálneho spustenia, ale na základe zmeny parametrov zariadení. Toto spúšťanie sa javí ako využiteľnejší spôsob v reálnom svete, kde by sa tento proces vedel zautomatizovať. Ďalej aj samotný interpret je možné rozšíriť, aby dokázal spracovať viac príkazov, menovite hlavne príkaz `Write` alebo jeho obdobu, ktorý zabezpečí, že bude možné zistiť, ktoré parametre boli zmenené. Vizuálny programovací jazyk je navrhnutý tak, že je schopný vytvárať vlastné bloky programu z existujúcich alebo nových. To ale aktuálna verzia nie je schopná spracovať a v budúcom rozširovaní interpreta je toto dôležitá časť, aby bol schopný spracovávať príkazy programu dynamicky.

# Literatúra

- [1] *About Scratch* online. Dostupné z: <https://scratch.mit.edu/about>. [cit. 2024-12-02].
- [2] ADI, P. D. P. a KITAGAWA, A. A Review of the Blockly Programming on M5Stack Board and MQTT Based for Programming Education. In: *2019 IEEE 11th International Conference on Engineering Education (ICEED)* online. IEEE, 2019, s. 102–107. ISBN 978-1-7281-3461-1. Dostupné z: <https://doi.org/10.1109/ICEED47294.2019.8994922>. [cit. 2024-12-04].
- [3] AL AWAMI, S. H.; MAHFUD AL ATY, M. a AL NAJAR, M. F. Comparison of IoT Architectures Based on the Seven Essential Characteristics. In: IEEE, ed. *2023 IEEE 3rd International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA)* online. 2023, s. 305–310. ISBN 979-8-3503-1989-7. Dostupné z: <https://doi.org/10.1109/MI-STA57575.2023.10169289>. [cit. 2024-11-05].
- [4] AL FUQAHA, A.; GUIZANI, M.; MOHAMMADI, M.; ALEDHARI, M. a AYYASH, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys & Tutorials* online, 2015, zv. 17, č. 4, s. 2347–2376. ISSN 1553-877X. Dostupné z: <https://doi.org/10.1109/COMST.2015.2444095>. [cit. 2024-11-20].
- [5] AL QASEEMI, S. A.; ALMULHIM, H. A.; ALMULHIM, M. F. a CHAUDHRY, S. R. IoT architecture challenges and issues: Lack of standardization. In: IEEE, ed. *2016 Future Technologies Conference (FTC)* online. 2016, s. 731–738. ISBN 978-1-5090-4171-8. Dostupné z: <https://doi.org/10.1109/FTC.2016.7821686>. [cit. 2024-11-04].
- [6] AL SARAWI, S.; ANBAR, M.; ALIEYAN, K. a ALZUBAIDI, M. Internet of Things (IoT) communication protocols: Review. In: IEEE, ed. *2017 8th International Conference on Information Technology (ICIT)* online. 2017, s. 685–690. ISBN 978-1-5090-6332-1. Dostupné z: <https://doi.org/10.1109/ICITECH.2017.8079928>. [cit. 2024-11-13].
- [7] ALLAM, Z. a DHUNNY, Z. A. On big data, artificial intelligence and smart cities. *Cities*, 2019, zv. 89, s. 80–91. ISSN 0264-2751. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S0264275118315968>.
- [8] BADI, C.; BELLINI, P.; DIFINO, A.; NESI, P.; PANTALEO, G. et al. MicroServices Suite for Smart City Applications. *Sensors*, November 2019, zv. 19, s. 4798.
- [9] BERNABÉ, A.; BERNARD, J.; MUSY, M.; ANDRIEU, H.; BOCHER, E. et al. Radiative and heat storage properties of the urban fabric derived from analysis of surface forms. *Urban Climate*, 2015, zv. 12, s. 205–218. ISSN 2212-0955. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S2212095515000115>.

- [10] BIBRI, S. E. The IoT for smart sustainable cities of the future: An analytical framework for sensor-based big data applications for environmental sustainability. *Sustainable Cities and Society*, 2018, zv. 38, s. 230–253. ISSN 2210-6707. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S2210670717313677>.
- [11] *What Is Bluetooth® Technology?* online. Dostupné z: <https://www.intel.com/content/www/us/en/products/docs/wireless/what-is-bluetooth.html>. [cit. 2024-11-08].
- [12] BUREŠ, M. *Systém pro zpracování dat z chytrých zařízení*. Brno, 2024. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. [cit. 2024-11-21].
- [13] BURNETT, M. M. a BAKER, M. J. A Classification System for Visual Programming Languages. *Journal of Visual Languages & Computing*, 1994, zv. 5, č. 3, s. 287–300. ISSN 1045-926X. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S1045926X84710159>.
- [14] *Channels* online. Dostupné z: <https://go.dev/tour/concurrency/2>. [cit. 2025-05-08].
- [15] CHODÁK, I. *Podpora uživatelsky definovaných funkcí ve Visuálním Programovacím Jazyce* online. Brno, 2025. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: <https://www.vut.cz/studenti/zav-prace/detail/161063>. [cit. 2025-05-06].
- [16] DEVALAL, S. a KARTHIKEYAN, A. LoRa Technology - An Overview. In: IEEE, ed. *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)* online. 2018, s. 284–290. ISBN 978-1-5386-0965-1. Dostupné z: <https://doi.org/10.1109/ICECA.2018.8474715>. [cit. 2024-11-20].
- [17] *The Internet of Things - European Parliament* online. Máj 2015. Dostupné z: [https://www.europarl.europa.eu/RegData/etudes/BRIE/2015/557012/EPRS\\_BRI\(2015\)557012\\_EN.pdf](https://www.europarl.europa.eu/RegData/etudes/BRIE/2015/557012/EPRS_BRI(2015)557012_EN.pdf). [cit. 13.10.2024].
- [18] *What Is QoS In Networking?* online. Dostupné z: <https://www.fortinet.com/resources/cyberglossary/qos-quality-of-service>. [cit. 2025-01-20].
- [19] FRANCESE, R.; RISI, M. a TORTORA, G. Iconic languages: Towards end-user programming of mobile applications. *Journal of Visual Languages & Computing*, 2017, zv. 38, s. 1–8. ISSN 1045-926X. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S1045926X1530063X>. SI:In honor of Prof SK Chang.
- [20] GADE, D. Summary of ICT and Digital Technologies used in Smart Cities. online, Február 2023. ISSN 2581-7795. Dostupné z: [https://www.researchgate.net/publication/368654318\\_Summary\\_of ICT\\_and\\_Digital\\_Technologies\\_used\\_in\\_Smart\\_Cities](https://www.researchgate.net/publication/368654318_Summary_of ICT_and_Digital_Technologies_used_in_Smart_Cities).
- [21] *Goroutines* online. Dostupné z: <https://go.dev/tour/concurrency/1>. [cit. 2025-05-08].
- [22] *Schemas and Types* online. Dostupné z: <https://graphql.org/learn/schema/>. [cit. 2025-01-19].

- [23] GREENGARD, S. *The Internet of Things*. The MIT Press, 2015. ISBN 0262527731.
- [24] HENRY, P. a LUO, H. WiFi: what's next? *IEEE Communications Magazine* online, 2002, zv. 40, č. 12, s. 66–72. Dostupné z: <https://doi.org/10.1109/MCOM.2002.1106162>. [cit. 2024-11-10].
- [25] HYNEK, J.; JOHN, P.; FORMÁNKOVÁ, K. a VALNÝ, M. *Služby pro systém řízení a monitoringu vody v retenčních nádržích* online. Apríl 2023. Dostupné z: <https://www.fit.vut.cz/research/project/c36644/>.
- [26] *What is the Internet of Things (IoT)?* online. Dostupné z: <https://www.ibm.com/topics/internet-of-things>. [cit. 2024-10-29].
- [27] *10 Internet of Things (IoT) Healthcare Examples* online. Dostupné z: <https://ordr.net/article/iot-healthcare-examples>. [cit. 2024-11-21].
- [28] JIA, X.; FENG, Q.; FAN, T. a LEI, Q. Analysis of anti-collision protocols for RFID tag identification. In: IEEE, ed. *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)* online. 2012, s. 877–880. ISBN 978-1-4577-1415-3. Dostupné z: <https://doi.org/10.1109/CECNet.2012.6202109>. [cit. 2024-11-10].
- [29] JIA, X.; FENG, Q.; FAN, T. a LEI, Q. RFID technology and its applications in Internet of Things (IoT). In: IEEE, ed. *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)* online. 2012, s. 1282–1285. ISBN 978-1-4577-1415-3. Dostupné z: <https://doi.org/10.1109/CECNet.2012.6201508>. [cit. 2024-11-11].
- [30] JOHN, P. *Optimising processes in IoT*. Brno, 2024. Dizertačná práca. Vysoké učení technické v Brně, Fakulta informačních technologií.
- [31] JUELS, A. RFID security and privacy: a research survey. *IEEE Journal on Selected Areas in Communications* online, 2006, zv. 24, č. 2, s. 381–394. ISSN 1558-0008. Dostupné z: <https://doi.org/10.1109/JSAC.2005.861395>. [cit. 2024-11-12].
- [32] KANTER, R. a LITOW, S. Informed and Interconnected: A Manifesto for Smarter Cities. *SSRN Electronic Journal* online, Jún 2009. Dostupné z: <https://doi.org/10.2139/ssrn.1420236>. [cit. 2024-11-22].
- [33] KUHAL, M. A.; FAROOQ, S.; HAMMAD, R. a BAHJA, M. Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access* online, 2021, zv. 9, s. 14181–14202. ISSN 2169-3536. Dostupné z: <https://doi.org/10.1109/ACCESS.2021.3051043>. [cit. 2024-12-01].
- [34] LEKIĆ, M. a GARDAŠEVIĆ, G. IoT sensor integration to Node-RED platform. In: *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)* online. IEEE, 2018, s. 1–5. ISBN 978-1-5386-4907-7. Dostupné z: <https://doi.org/10.1109/INFOTEH.2018.8345544>. [cit. 2024-12-05].
- [35] MISHRA, B. a KERTESZ, A. The Use of MQTT in M2M and IoT Systems: A Survey. *IEEE Access* online, 2020, zv. 8, s. 201071–201086. ISSN 2169-3536. Dostupné z: <https://doi.org/10.1109/ACCESS.2020.3035849>. [cit. 2024-11-20].

- [36] *What is MQTT Quality of Service (QoS) 0,1, & 2? – MQTT Essentials: Part 6* online. Dostupné z: [WhatisMQTTQualityofService\(QoS\)0,1,&2?ãMQTTEssentials:Part6](https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call). [cit. 2024-11-20].
- [37] MYERS, B. A. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1990, zv. 1, č. 1, s. 97–123. ISSN 1045-926X. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S1045926X05800369>.
- [38] NAIK, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: IEEE, ed. *2017 IEEE International Systems Engineering Symposium (ISSE)* online. 2017, s. 1–7. ISBN 978-1-5386-3403-5. Dostupné z: <https://doi.org/10.1109/SysEng.2017.8088251>. [cit. 2024-11-21].
- [39] *Object-Oriented Design (OOD) – System Design* online. Dostupné z: <https://www.geeksforgeeks.org/oops-object-oriented-design/>. [cit. 2025-01-14].
- [40] PASTERNAK, E.; FENICHEL, R. a MARSHALL, A. N. Tips for creating a block language with blockly. In: *2017 IEEE Blocks and Beyond Workshop (B&B)* online. IEEE, 2017, s. 21–24. ISBN 978-1-5386-2480-7. Dostupné z: <https://doi.org/10.1109/BLOCKS.2017.8120404>. [cit. 2024-12-04].
- [41] PODVOJSKÝ, L. *Vizuální programování IoT zařízení*. Brno, 2024. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií.
- [42] RAY, P. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, Marec 2017, zv. 2017.
- [43] RESNICK, M.; MALONEY, J.; MONROY HERNÁNDEZ, A.; RUSK, N.; EASTMOND, E. et al. Scratch: programming for all. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery, november 2009, zv. 52, č. 11, s. 60–67. ISSN 0001-0782. Dostupné z: <https://doi.org/10.1145/1592761.1592779>.
- [44] *Where do you find RFID technology in everyday life?* online. Dostupné z: <https://www.trace-id.com/en/where-do-you-find-rfid-technology-in-everyday-life/>. [cit. 2024-11-13].
- [45] *Remote Procedure Call* online. Dostupné z: <https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call>. [cit. 2025-04-29].
- [46] SAVIDIS, A.; VALSAMAKIS, Y. a LINARITIS, D. Blockly Toolbox for Visual Programming of Smart IoT Automations. In: online. ResearchGate, September 2022, s. 93–103. ISBN 978-3-031-06893-5. Dostupné z: [https://doi.org/10.1007/978-3-031-06894-2\\_9](https://doi.org/10.1007/978-3-031-06894-2_9). [cit. 2024-12-04].
- [47] SEHRAWAT, D. a GILL, N. S. Smart Sensors: Analysis of Different Types of IoT Sensors. In: IEEE, ed. *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)* online. 2019, s. 523–528. ISBN 978-1-5386-9439-8. Dostupné z: <https://doi.org/10.1109/ICOEI.2019.8862778>. [cit. 2024-11-04].

- [48] *Statista* online. 2024. Dostupné z: <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>. [cit. 10.10.2024].
- [49] STUDLEY, M.; PRESUTTI, V.; NARDI, D. a CARTER, S. Benchmarking robots in smart cities. *XRDS*. New York, NY, USA: Association for Computing Machinery, apríl 2022, zv. 28, č. 3, s. 20–25. ISSN 1528-4972. Dostupné z: <https://doi.org/10.1145/3522686>.
- [50] SUTCLIFFE, A. Evaluating the costs and benefits of end-user development. *ACM SIGSOFT Software Engineering Notes* online, Júl 2005, zv. 30, s. 1–4. Dostupné z: <https://doi.org/10.1145/1082983.1083241>. [cit. 2024-12-17].
- [51] VALDERAS, P.; TORRES, V.; MANSANET, I. a PELECHANO, V. A mobile-based solution for supporting end-users in the composition of services. *Multimedia Tools and Applications* online. Springer, August 2017, zv. 76, s. 16315 – 16345. Dostupné z: <https://doi.org/10.1007/s11042-016-3910-4>. [cit. 2024-12-06].
- [52] VALDERAS, P.; TORRES, V. a PELOCHANO, V. Supporting a Hybrid Composition of Microservices. The EUCalipTool Platform. *Journal of Software Engineering Research and Development*, Feb. 2020, zv. 8, s. 1:1 – 1:14. Dostupné z: <https://journals-sol.sbc.org.br/index.php/jserd/article/view/457>.
- [53] VIGDER, M. End-user software development in a scientific organization. In: *2009 ICSE Workshop on Software Engineering Foundations for End User Programming* online. 2009, s. 15–19. ISBN 978-1-4244-3738-2. Dostupné z: <https://doi.org/10.1109/SEEUP.2009.5071698>. [cit. 2024-12-01].
- [54] WANT, R. Near field communication. *IEEE Pervasive Computing* online, 2011, zv. 10, č. 3, s. 4–7. ISSN 1558-2590. Dostupné z: <https://doi.org/10.1109/MPRV.2011.55>. [cit. 2024-11-17].
- [55] *What Is Wi-Fi?* online. Dostupné z: <https://www.cisco.com/c/en/us/products/wireless/what-is-wifi.html>. [cit. 2024-11-10].
- [56] XU, L. D.; HE, W. a LI, S. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics* online, 2014, zv. 10, č. 4, s. 2233–2243. ISSN 1941-0050. Dostupné z: <https://doi.org/10.1109/TII.2014.2300753>. [cit. 2024-11-21].
- [57] XU, R.; ZENG, Q.; ZHU, L.; CHI, H.; DU, X. et al. Privacy Leakage in Smart Homes and Its Mitigation: IFTTT as a Case Study. *IEEE Access* online, 2019, zv. 7, s. 63457–63471. ISSN 2169-3536. Dostupné z: <https://doi.org/10.1109/ACCESS.2019.2911202>. [cit. 2025-01-25].
- [58] ZANELLA, A.; BUI, N.; CASTELLANI, A.; VANGELISTA, L. a ZORZI, M. Internet of Things for Smart Cities. *IEEE Internet of Things Journal* online, 2014, zv. 1, č. 1, s. 22–32. ISSN 2327-4662. Dostupné z: <https://doi.org/10.1109/JIOT.2014.2306328>. [cit. 2024-11-23].

- [59] ZHAO, V.; ZHANG, L.; WANG, B.; LITTMAN, M. L.; LU, S. et al. Understanding Trigger-Action Programs Through Novel Visualizations of Program Differences. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* online. New York, NY, USA: Association for Computing Machinery, 2021. CHI '21. ISBN 9781450380966. Dostupné z: <https://doi.org/10.1145/3411764.3445567>. [cit. 2025-01-25].

# Príloha A

## Obsah externej prílohy

- Tento dokument v PDF formáte.
- README.md – bližší popis repozitára.
- src/ – adresár so zdrojovými kódmi.
  - models/ – dátové štruktúry a modely použité v aplikácii.
  - statements/ – definície a implementácie príkazov v programe.
  - parser/ – spracovanie vstupu.
  - types/ – definície základných typov programu.
  - services/, utils/ – pomocné funkcie.
- tests/ – adresár s testami.
- programs/ – adresár s príkladmi programov.
- latex/ – adresár s latexovými súbormi.