VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO ALOKACI A KONFIGURACI HARDWAROVÝCH ZDROJŮ

SYSTEM FOR ALLOCATION AND CONFIGURATION OF HARDWARE RESOURCES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAN SLOVÁČEK**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. JITKA KRESLÍKOVÁ, CSc.**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/19510/2017/xslova17

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Slováček Jan, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Systém pro alokaci a konfiguraci hardwarových zdrojů**
System for Allocation and Configuration of Hardware Resources
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s problematikou sdílení hardwarových zdrojů ve vývojové laboratoři Siemens, s.r.o. Nastudujte komunikační protokoly používaných vstupně/výstupních zařízení.
2. Analyzujte požadavky a možné vlastnosti systému pro alokaci a konfiguraci vyvíjených zařízení.
3. Na základě provedené analýzy a po dohodě s konzultantem firmy Siemens, s.r.o., navrhněte architekturu systému.
4. Zvolte vhodné vývojové prostředí a implementujte prototyp navržené aplikace. Lze využít i knihovny třetích stran. Vytvořenou aplikaci otestujte.
5. Použitelnost aplikace demonstrujte na vhodném vzorku dat vybraném po dohodě s vedoucí.
6. Zhodnoťte dosažené výsledky a navrhněte možné rozšíření projektu.

Literatura:

- RICHARDSON, L. *RESTful Web APIs*. Sebastopol, Calif: O'Reilly, 2013. ISBN 978-1449359737.
- PIGAN, R. *Automating with PROFINET industrial communication based on Industrial Ethernet*. Erlangen: Publicis Corporate Publ, 2006. ISBN 3-89578-256-4.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů zadání 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kreslíková Jitka, doc. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a implementací systému pro rezervování a vzdálené řízení sdílených prostředků. Předmětem sdílení jsou vývojové a testovací prototypy. Jejich řízení probíhá pomocí vstupně/výstupních zařízení ET-7042 a SITOP PSU8600. V práci je čtenář obeznámen s cílovým prostředím, REST aplikačními rozhraními a komunikačními standardy Modbus a Profinet. Tyto znalosti jsou využity v návrhu a implementaci systému. Hlavním cílem práce je zefektivnění vývoje a testování v prostředí vývojové laboratoře.

Abstract

This master's thesis is about analysis, design and implementation of a system for shared resources reservation and its remote control. Subjects of sharing are test and development prototypes. For remote control are used I/O devices ET-7042 and SITOP PSU8600. The reader is introduced to target environment, REST application interfaces and communication standards Modbus and Profinet. The knowledge gathered is used in design and implementation of the system. The main goal of thesis is to make development and testing in lab more effective.

Klíčová slova

Modbus, PROFINET, ET-7042, SITOP PSU8600, REST API, Python, sdílené prostředky

Keywords

Modbus, PROFINET, ET-7042, SITOP PSU8600, REST API, Python, shared resources

Citace

SLOVÁČEK, Jan. *Systém pro alokaci a konfiguraci hardwarových zdrojů*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Jitka Kreslíková, CSc.

Systém pro alokaci a konfiguraci hardwarových zdrojů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením paní doc. RNDr. Jitky Kreslíkové, CSc. Další informace mi poskytli Ing. Petr Kramný a Mgr. Petr Raška. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Slováček
21. května 2018

Poděkování

Tímto bych chtěl poděkovat mé vedoucí, doc. RNDr. Jitce Kreslíkové, CSc., za systematické vedení při tvorbě této diplomové práce. Také bych rád poděkoval rodině, kolegům, kamarádům a v neposlední řadě přítelkyni za projevenou podporu a trpělivost.

Obsah

Kapitola 1

Úvod

Sdílení prostředků v počítačové síti je dnes již poměrně běžnou záležitostí. Ať už se jedná o domácnost či kancelář, důvody sdílení jsou obvykle ekonomické. Ne všude však lze uplatnit řešení navržená pro zmíněná prostředí. Stejně je tomu i v případě vývojových laboratoří, kde jsou v podobě sdílených prostředků prototypy nejnovějších zařízení. Z výše uvedeného plyne cíl této práce, a sice vytvořit systém, který řeší sdílení prostředků v netypických podmínkách vývojové a testovací laboratoře společnosti Siemens.

Tuto technickou zprávu tvoří sedm části. Nejprve jsou v kapitole 2 shrnuty požadavky na výsledný systém a přiblíženo prostředí, pro které byla aplikace navržena. Jedním z požadavků popsaných ve zmíněné kapitole je integrace používaných zařízení. Kapitola 3 je proto věnována relevantním komunikačním protokolům. V další kapitole 4 je poté popsána REST architektura jako jedna z možností pro webové aplikační rozhraní. Na základě získaných poznatků byly v kapitole 5 navrženy jednotlivé komponenty systému, entity relační databáze či zmiňované rozhraní. Stěžejní části implementace jsou shrnuty v následující kapitole 6. Na závěr jsou v kapitole 7 uvedeny způsoby zajištění kvality výsledného řešení a popsáno zkušební nasazení v cílovém prostředí.

V průběhu tvorby této práce bylo využito konzultací u externího zadavatele, společnosti Siemens s. r. o., oddělení Corporate Technology Brno, který poskytl zdroje i zařízení potřebné k vývoji a z větší části specifikoval požadavky na výsledný systém.

Tato diplomová práce přímo navazuje na stejnojmenný semestrální projekt, z kterého byla převzata analýza prostředí v kapitole 2, shrnutí komunikačních protokolů v kapitole 3 a návrh systému v kapitole 5. Zmíněné kapitoly byly na základě této práce doplněny o další rozšiřující informace.

Kapitola 2

Analýza požadavků

Jak bylo zmíněno v úvodu této zprávy, požadavek na vývoj vyplynul z potřeb externího zadavatele. Primární využití systému je sdílení prostředků ve vývojové laboratoři nejen mezi samotnými vývojáři, ale také mezi automatizovanými testovacími nástroji. Hlavní motivací je právě zajištění efektivního sdílení zdrojů a zabránění možných konfliktů. Dalším cílem je pak integrace existujících řešení automatizace.

Tato kapitola tedy upřesňuje cílové prostředí (podkapitola 2.1) a popisuje požadované vlastnosti vyvíjeného systému (podkapitola 2.2). Nakonec jsou shrnuty zařízení, pro která je v systému vyžadovaná podpora (podkapitola 2.3).

2.1 Cílové prostředí

Laboratoř, která je cílovým prostředím vyvíjeného systému, slouží pro vývoj a testování zařízení podporujících průmyslový standard Profinet. Jelikož se jedná většinou o prototypy nebo o finančně nákladné produkty, jsou tato zařízení sdílena mezi všechny účastníky laboratoře. Ty lze rozdělit dle chování v prostředí následovně:

- *Vývojář* – využívá prostředky během dne pro účely vývoje, obvykle se jedná o krátkodobé použití a případný konflikt není kritický.
- *Tester* – využívá prostředky během dne pro účely manuálního testování, obvykle se jedná o použití v řádech hodin a případný konflikt může být kritický.
- *Automatizovaný testovací nástroj* – využívá prostředky téměř nepřetržitě, spouští se automaticky na základě plánování a běží dlouhodobě až v řádech dnů, případný konflikt je kritický.

Doposud laboratoř neobsahovala žádný systém pro bezpečné sdílení prostředků a konflikty mezi účastníky nebyly nijak výjimečné. Příčinou byla obvykle nedostatečná komunikace mezi uživateli nebo špatně naplánované běhy jednotlivých testovacích nástrojů. Právě v případě automatizovaných nástrojů se vzhledem k délce běhu jednalo o značnou časovou ztrátu ve vývoji produktu.

Sdílená zařízení jsou ve většině případech vzdáleně ovladatelná pomocí vstupně/výstupních zařízení (dále V/V zařízení). Obvykle se jedná o zapnutí a vypnutí napájení daného zařízení. Vyskytují se však i jiné typy — například změna síťové topologie pomocí zařízení Ethernet Breaker[1]. Ve stávajícím řešení jsou stavy výstupů těchto zařízení nastavovány

[1] *Ethernet Breaker* - zařízení vyvíjené společnosti Siemens pro účely automatizovaného testování, které umožňuje přepínání síťové topologie na fyzické vrstvě.

pomocí jednoduchých skriptů, případně je funkcionalita přímo integrovaná do testovacího nástroje. Použitým V/V zařízením se dále věnuje podkapitola 2.3.

2.2 Požadované vlastnosti

Jak už vyplývá z názvu této práce, hlavní funkcionalita vyvíjeného systému je sdílení prostředků. Bude se jednat o klient/server aplikaci. Základní funkcionalita serverové části je shrnuta v následujících bodech:

- *Správa prostředků* – možnost vytvořit, editovat a odstranit určitý prostředek.
- *Sdílení prostředků* – možnost alokovat a uvolnit požadovaný prostředek.
- *Zabránění konfliktu* – neumožnit alokaci prostředku v případě jeho alokace jiným uživatelem.

Další požadavky na serverovou část systému jsou uvedeny níže. Klientská část pak bude, jak je běžné, sloužit k odesílání požadavků na server a reprezentaci obdržených odpovědí.

Aplikační rozhraní

Jelikož uživatelem systému nebudou pouze lidé, ale také automatizované nástroje, je jedním s požadavků jednotné aplikační rozhraní. Cílem je zajistit snadnou integraci klientské části systému do existujících nástrojů bez ohledu na jejich implementační detaily. Jednou z variant může být implementace serverové části systému jako webovou službu využitím protokolu HTTP (viz kapitola 4).

Integrace V/V zařízení

Další požadovanou vlastností je integrace V/V zařízení použitých v laboratoři. Uživatel, kterému se podaří úspěšně alokovat daný prostředek, bude mít možnost měnit jeho stav. Systém bude na základě změny stavu aktualizovat výstupy V/V zařízení.

Další požadavky

Mezi další požadavky na výsledný systém patří:

- *Zajištění persistence dat* – data systému budou ukládaná perzistentně na disk do databáze nebo souboru.
- *Autentizace* – systém bude umožňovat autentizaci uživatelů a základní rozdělení rolí.

2.3 Podporovaná zařízení

Pro vzdálené ovládání sdílených prostředků jsou aktuálně využity dva typy V/V zařízení. Ve vyvíjeném systémů je vyžadována integrace obou těchto zařízení a proto si je v této podkapitole přiblížíme.

ET-7042

ET-7042 je V/V zařízení vyvinuté společností ICP DAS a jeho primární využití je v automatizaci budov, výroby, strojů a podobně [10]. Použitá varianta disponuje 16 digitálními výstupy. Zařízení je díky Ethernet vstupu možné vzdáleně konfigurovat, monitorovat a diagnostikovat. K tomu lze využít zabudovaný webový server nebo protokol Modbus — zařízení podporuje slave funkcionalitu protokolu Modbus/TCP[2]. Protokolu Modbus je dále věnována podkapitola 3.1.

V současné době jsou ve vývojové laboratoři využívány dvě zařízení tohoto typu. První z nich je integrováno do třífázového napájecího okruhu a umožňuje přerušit napájení vybraného spotřebiče. V tomto případě je přes každý digitální výstup ovládán jiný sdílený prostředek. Druhé ET-7042 je vyčleněno k ovládání zařízení Ethernet Breaker zmíněného výše. Digitální výstupy jsou zde připojeny na jednotlivé Ethernet porty zařízení a umožňují jejich variabilní propojení. Sdílené prostředky představují nejrůznější kombinace takových propojení a jeden prostředek tedy může potencionálně zahrnovat více než jeden digitální výstup. Tento fakt je nutné zohlednit při návrhu systému.

SITOP PSU8600

SITOP PSU8600 je inteligentní napájecí zdroj vyvíjený společností Siemens, který může disponovat až 36 výstupy s výstupními proudy 2.5, 5, 10, 20 nebo 40 ampér [21]. Pro každý z nich lze individuálně nastavit výstupní napětí v rozsahu 5 až 28 voltů. Zařízení je vybaveno dvěma Ethernet porty s podporou standardů Profinet IO (viz podkapitola 3.2) a OPC[3]. Společně s integrovaným webovým serverem umožňují tyto standardy vzdálenou konfiguraci a monitorování. SITOP PSU8600 je modulární zařízení — lze zvolit jeden ze čtyř typů základní jednotky doplněný o kombinaci až šesti rozšiřujících modulů následujících typů:

- *SITOP CNX8600* – rozšíření základní jednotky o 4 nebo 8 výstupů.
- *SITOP BUF8600* – záložní napájení pří výpadcích.

Ve vývojové laboratoři je aktuálně jedno takové zařízení. Použitá kombinace obsahuje základní jednotku se 4 výstupy doplněnou o další 3 rozšiřující jednotky SITOP CNX8600, každá disponující také 4 výstupy, a 1 modul typu SITOP BUF8600. Dohromady je tedy k dispozici 16 selektivně ovládaných výstupů, které jsou využívány k napájení spotřebičů v 24 voltovém okruhu. Podobně jako v případě zařízení ET-7042 integrovaného do třífázového napájecího okruhu je tedy zařízení používáno jako spínač a každý výstup je reprezentován samostatným sdíleným prostředkem.

[2] *TCP* (Transmission Control Protocol) – protokol transportní vrstvy zajišťující spolehlivý přenos dat.
[3] *OPC* (OLE for Process Control) – standard pro bezpečnou a spolehlivou výměnu dat v prostoru průmyslové automatizace a v jiných průmyslových odvětvích.

Kapitola 3

Komunikační protokoly

Jak již bylo řečeno, jedním z požadavků je integrace V/V zařízení ET-7042 a SITOP PSU8600. V této kapitole jsou proto shrnuty protokoly, které lze využít pro komunikaci právě s těmito zařízeními. Jmenovitě se jedná o Modbus (podkapitola 3.1) a Profinet (podkapitola 3.2).

3.1 Modbus

Modbus je otevřený komunikační protokol pracující na aplikační vrstvě modelu ISO/OSI a umožňuje klient-server komunikaci mezi zařízeními nezávisle na použité sběrnici či typu sítě [12, 13]. Vzhledem ke svému širokému rozšíření se stal de facto komunikačním standardem v průmyslovém odvětví. Protokol je aktuálně implementován v následujících variantách (rozdělení dle přenosového média a typu komunikace):

- TCP/IP komunikace přes Ethernet,
- asynchronní sériový přenos přes různá média (RS-232, RS-422, optická vlákna atd.),
- Modbus Plus[1].

Jelikož zařízení ET-7042 podporuje pouze první zmíněnou variantu TCP/IP (viz podkapitola 2.3), jsou další části této podkapitoly věnovány především této implementaci.

Popis protokolu

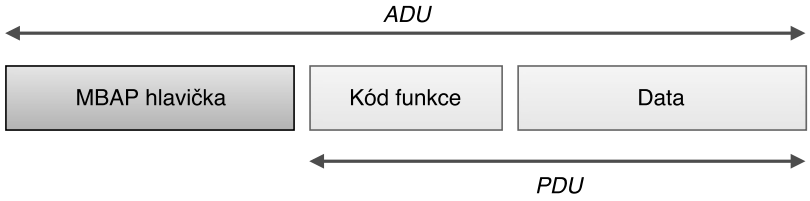
Protokol Modbus definuje zprávu jako protokolovou datovou jednotku, neboli PDU (angl. „Protocol Data Unit“), která se skládá z následujících položek:

- *Kód funkce* – vyjadřuje požadovanou operaci. Jedná se o jeden bajt a validní hodnoty jsou v rozsahu 1 až 255 (kódy 128 až 255 jsou vyhrazeny k vyjádření chyby).
- *Data funkce* – položka je nepovinná a její velikost je proměnlivá dle zvolené operace.

Dle použité varianty protokolu je PDU před odesláním rozšířeno o další položky. Dále je pak označováno jako aplikační datová jednotka, neboli ADU (angl. „Application Data Unit“). V případě varianty TCP/IP je k PDU přidána hlavička MBAP (angl. „Modbus Application

[1] *Modbus Plus* – vysokorychlostní peer-to-peer komunikace.

Protocol“). Popsaná struktura je znázorněna na obrázku 3.1, obsah MBAP hlavičky je poté popsán v tabulce 3.1.



Obrázek 3.1: Struktura zprávy protokolu Modbus [inspirováno z 13]

Název položky	**Velikost**	**Popis**
ID transakce	2 bajty	Určeno pro párování požadavku a odpovědi
ID protokolu	2 bajty	0 = Modbus
Délka	2 bajty	Počet následujících bajtů
ID jednotky	1 bajt	Identifikace jednotky na daném hostu

Tabulka 3.1: Struktura MBAP hlavičky [inspirováno z 13]

Veškerá Modbus/TCP komunikace probíhá na portu 502. Spojení iniciuje klient zasláním požadavku. MBAP hlavička je u požadavku i odpovědi identická (s výjimkou délky). V případě, že je přijaté ADU validní a během provádění požadované operace se nevyskytla žádná chyba, je kód funkce v odpovědi identický s kódem v požadavku. Následovaný je vyžadovanými daty. V případě selhání je kód funkce odpovědi získán z kódu funkce požadavku a nastavením jeho nejvýznamnějšího bitu na logickou 1. Datová část pak obsahuje kód výjimky, které jsou shrnuty v sekci *Zpracování chyb* na straně 10.

Datový model a adresování

Datový model protokolu Modbus je založen na sérii tabulek rozdílného významu. Základní čtyři typy lze vidět v tabulce 3.2. Každá tabulka má přidělený vlastní adresový prostor. Mohou se však navzájem, částečně či úplně, překrývat. Mapování závisí na konkrétní implementaci. Adresa položky má velikost 2 bajty. Protokol tedy umožňuje adresovat až 65536 datových položek pro každou z tabulek. Takto definovaný model je dostatečně obecný pro vytvoření libovolného aplikačního rozhraní.

Název	**Velikost**	**Čtení**	**Zápis**
Discretes Input	1 bit	✓	✗
Coil	1 bit	✓	✓
Input Registers	16 bitů	✓	✗
Holding Registers	16 bitů	✓	✓

Tabulka 3.2: Základní tabulky datového modelu protokolu Modbus [inspirováno z 12]

K jednotlivým tabulkám lze přistupovat pomocí příslušných funkcí. Dle kódu jsou tyto funkce rozděleny do tří kategorií:

a) *Veřejné kódy* – jasně definované funkce, které jsou schválené organizací Modbus. Tyto funkce jsou dobře zdokumentované, otestované a je garantována jejich unikátnost.

b) *Uživatelsky definované kódy* – umožňují uživateli implementovat dodatečné funkce, které nejsou součástí specifikace. Jedná se o kódy v rozsahu 65 až 72 a 100 až 110. U těchto kódů není garantována unikátnost. Je však možné zažádat organizaci Modbus o přesun funkce mezi veřejné kódy.

c) *Rezervované kódy* – použité komerčními společnostmi z důvodu zpětné kompatibility u starších zařízeních. Tyto kódy nejsou veřejně dostupné.

Základní funkce

Základní veřejné funkce relevantní pro Modbus/TCP jsou následující (hexadecimální číslo v závorce reprezentuje kód funkce):

- *Read Coils* (0x01) – vyčtení aktuálního stavu až 2000 navazujících položek typu *Coil*. Data požadavku obsahují adresu první položky a počet požadovaných položek. Data pozitivní odpovědi jsou ve formátu bitového pole a obsahují stavy jednotlivých položek. Nejméně významný bit prvního bajtu obsahuje stav položky adresované v požadavku.

- *Read Discrete Inputs* (0x02) – funkce je identická s funkcí *Read Coils*. Tentokrát se však jedná o čtení položek typu *Discrete Input*.

- *Read Holding Registers* (0x03) – vyčtení aktuálního stavu až 125 navazujících položek typu *Holding Register*. Data požadavku obsahují adresu první položky a počet požadovaných položek. Každá položka je v odpovědi reprezentována dvěma bajty ve formátu big-endian.

- *Read Input Registers* (0x04) – funkce je identická s funkcí *Read Holding Registers*. Tentokrát se však jedná o čtení položek typu *Input Register*.

- *Write Single Coil* (0x05) – zápis hodnoty do jedné položky typu *Coil*. Data požadavku obsahují adresu položky a novou hodnotu. Jelikož velikost pole nové hodnoty je 2 bajty, logická 1 je reprezentována jako hexadecimální číslo 0xFF00 a logická 0 jako 0x0000. Jiné hodnoty nejsou přípustné. V případě úspěchu je jako odpověď vráceno PDU požadavku.

- *Write Single Register* (0x06) – funkce je identická s funkcí *Write Single Coil*. Jedná se však o položku typu *Holding Register* a nová požadovaná hodnota velikosti 2 bajty je tedy validní v celém svém rozsahu.

- *Write Multiple Coils* (0x0F) – zápis hodnot až 1968 navazujících položek typu *Coil*. Data požadavku obsahují adresu první položky a nové hodnoty reprezentovány bitovým polem. V případě úspěchu obsahuje odpověď adresu první položky a počet nastavených položek.

- *Write Multiple Registers* (0x10) – zápis hodnot až 123 navazujících položek typu *Holding Register*. Data požadavku obsahují adresu první položky a nové požadované hodnoty, které jsou obdobně jako u funkce *Read Holding Registers* reprezentovány sérií dvou bajtů. V případě úspěchu obsahuje odpověď adresu první položky a počet nastavených položek.

Protokol Modbus definuje i další veřejné funkce. Vesměs se však jedná o kombinace funkcí výše popsaných.

Zpracování chyb

V případě selhání během zpracování požadavku je serverem vrácena negativní odpověď, která je indikována nastavením nejvýznamnějšího bitu v kódu funkce. V takovém případě obsahuje datová část jeden z následujících kódů výjimek:

- *Nevalidní funkce* (0x01) - požadovaný kód funkce není na daném serveru dostupný. Tato výjimka se vyskytuje v případě starší implementace bez podpory nejnovějších funkcí. Může také znamenat chybnou konfiguraci.
- *Nevalidní datová adresa* (0x02) - na požadované adrese nejsou mapovány žádná data. Často se vyskytuje při operacích s více položkami, kdy kombinace nevhodné počáteční adresy a počtu položek vede k přístupu mimo definovaný rozsah.
- *Nevalidní data* (0x03) - některá hodnota datové části je pro server nepřípustná. Neznamená to však, že nějaká hodnota je nevalidní pro aplikační program — protokol Modbus je abstrahován od implementace daného programu a nezná význam konkrétních hodnot pro konkrétní položky.
- *Selhání zařízení* (0x04) - během vykonávání požadované operace došlo k selhání, ze kterého se nelze zotavit.
- *Potvrzení* (0x05) - prevence proti vypršení časového limitu na straně klienta. Používá se v případech, kdy byl požadavek úspěšně zpracován, ale jeho provádění je časově náročné.
- *Zařízení je zaneprázdněno* (0x06) - server je zaneprázdněn prováděním dlouhotrvajícího požadavku a klient by měl zopakovat požadavek později.
- *Chyba parity paměti* (0x08) - server detekoval chybu parity v paměti. Klient může požadavek zopakovat, ale pravděpodobně bude nutná údržba zařízení.
- *Nedostupná cesta* (0x0A) - speciální výjimka při použití Modbus brány[2]. Zasílá se v případech, kdy brána nedokáže nalézt vnitřní komunikační cestu ze vstupního na výstupní port. Obvykle znamená chybnou konfiguraci brány nebo její přetížení.
- *Zařízení neodpovídá* (0x0B) - speciální výjimka při použití Modbus brány. Zasílá se v případech, kdy brána neobdrží odpověď od cílového zařízení. Obvykle to znamená, že zařízení se v dané síti vůbec nevyskytuje.

[2] *Modbus brána* – je úzce spjatá s Modbus/TCP. Umožňuje propojit zařízení, které podporují Modbus sériový přenos, s Ethernet sítí.

Modbus v ET-7042

Jak již bylo několikrát zmíněno, V/V zařízení ET-7042 podporuje protokol Modbus/TCP [10]. Dostupné funkce zahrnují například obnovu systému do výchozího nastavení, aktualizaci firmwaru, zapnutí/vypnutí webového rozhraní a podobně. Z pohledu této práce je však nejzajímavější funkcionalitou možnost nastavení stavů všech 16 digitálních výstupů.

Adresový rozsah pro jednotlivé tabulky protokolu určuje v zařízení první číslice adresy (viz tabulka 3.3). Zmíněná funkcionalita pro konfiguraci digitálních výstupů je mapována na položky typu *Coil* na adresách 0 - 15, kdy jednotlivé adresy jsou shodné s označením příslušného digitálního výstupu. Zařízení podporuje všechny popsané funkce protokolu Modbus. Pro konfiguraci výstupů tedy lze použít funkce *Read Coils*, *Write Single Coil* a *Write Multiple Coils*. Poslední podstatnou informací je, že výchozí identifikace jednotky, která je potřebná v MBAP hlavičce, je 1.

Typ položky	**Adresa**	**Rozsah**
Coil	0xxxx	0 - 9999
Discretes Input	1xxxx	10000 - 19999
Input Registers	3xxxx	30000 - 39999
Holding Registers	4xxxx	40000 - 49999

Tabulka 3.3: Adresy základních tabulek protokolu Modbus v ET-7042 [inspirováno z 10]

3.2 Profinet

Profinet (Process Field Net) je otevřený komunikační standard zaměřený na automatizaci v průmyslovém odvětví [15, 17, 22]. Koncept Profinetu byl standardizován normami IEC 61158 a IEC 61784. Jeho základ tvoří průmyslový Ethernet a je tedy plně kompatibilní vzhledem ke standardům IEEE 802. Nabízeny jsou dvě navzájem nezávislé varianty:

a) *Profinet IO* – propojení distribuovaných V/V zařízení. Data jsou periodicky vyměňována s PLC[3] skrz tři různé komunikační kanály.

b) *Profinet CBA* – vychází z technologie DCOM[4]. Komunikace probíhá skrz uniformní rozhraní a data jsou zasílána nahodile.

Další části podkapitoly jsou věnovány výhradně prvnímu zmíněnému typu Profinet IO, jelikož zařízení SITOP PSU8600 spadá právě do této kategorie.

Profinet komunikace

Komunikace je v Profinetu odstupňovaná do následujících úrovní:

- *NRT* (Non Real-Time) – standardní TCP/IP nebo UDP[5]/IP datový přenos. Používá se při výměně časově nekritických dat.

[3] *PLC* (Programmable Logic Controller) – počítač, který byl svou konstrukcí a použitými technologiemi přizpůsoben k řízení průmyslových procesů.

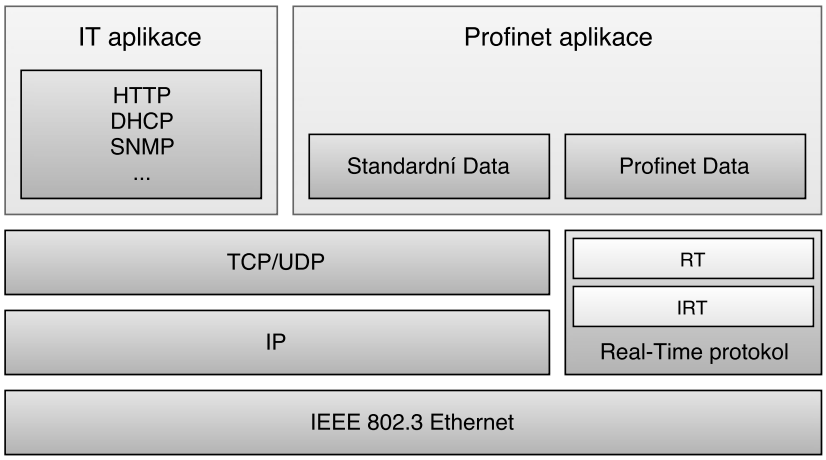
[4] *DCOM* (Distributed Component Object Model) – jeden ze standardů pro vzdálené volání procedur, který s využitím objektově orientovaného návrhu definuje způsob komunikace mezi klientem a serverem.

[5] *UDP* (User Datagram Protocol) – protokol transportní vrstvy, který zajišťuje nespolehlivý přenos dat.

- *RT* (Real-Time) – přenos dat v reálném čase[6]. Používá se pro výměnu časově kritických procesních dat.
- *IRT* (Isochronous Real-Time) – izochronní datový přenos v reálném čase. Je určen pro specifické aplikace vyžadující rychlou odezvu s minimální odchylkou zpoždění na síti (jitter).

Všechny typy komunikace probíhají na jednom přenosovém médiu. To je umožněno díky rozdělení přenosového cyklu na deterministickou část pro IRT a otevřenou část pro NRT a RT komunikaci. Pro rozlišení jednotlivých úrovní se taktéž využívá standard IEEE 802.1Q.

Pro komunikaci v reálném čase disponuje Profinet optimalizovanou druhou vrstvou modelu ISO/OSI nazvanou *Real-Time* protokol. Ta nahrazuje protokoly vyšších vrstev a přímo zapouzdřuje aplikační data. Oproti protokolům UDP/IP a TCP/IP tak výrazně snižuje celkovou velikost rámce, čímž je taktéž snížena doba přenosu a zpracování. Vzhledem k tomu, že protokol odpovídá standardu IEEE 802.3, je navíc s těmito protokoly umožněn souběžný provoz. Z důvodu chybějící IP vrstvy však není možné směrovat tyto pakety mezi sítěmi — pro adresaci se používá MAC adresa. Protokoly používané v Profinetu jsou znázorněny na obrázku 3.2.



Obrázek 3.2: Komunikační protokoly v Profinetu [převzato z 22]

Při komunikaci PLC a V/V zařízení je pro každý typ dat určen speciální přenosový kanál nazvaný komunikační vztah, neboli CR (angl. „Communication Relationship“). Jednotlivé komunikační vztahy se liší nejen typem přenášených dat, ale taktéž použitým typem komunikace. Profinet definuje následující typy:

a) *Record Data CR* – acyklický přenos datových struktur pomocí UDP/IP.

b) *IO Data CR* – cyklický přenos V/V dat pomocí RT protokolu.

c) *Alarm CR* – acyklický přenos událostí pomocí RT protokolu.

Před samotnou výměnou aplikačních dat je mezi PLC a V/V zařízením navázán aplikační vztah, neboli AR (angl. „Application Relationship“), jehož součástí jsou zmíněné komunikační vztahy. Vazby mezi komunikačními a aplikačními vztahy lze vidět na obrázku 3.3. Jednotlivé typy aplikačních vztahů jsou následující:

[6] *Komunikace v reálném čase* – systém je schopen zpracovat požadavek v definovaném čase a jeho odpověď je deterministická.

Obrázek 3.3: Komunikační a aplikační vztahy [převzato z 22]

a) *IOC-AR* – běžný aplikační vztah mezi PLC a V/V zařízením, který zahrnuje všechny tři typy komunikačních vztahů. Po navázání spojení je tak PLC zařízení umožněna výměna cyklických vstupních a výstupních dat, acyklické čtení a zápis datových struktur a taktéž je mu zasílána diagnostika v případě problémů.

b) *S-AR* – aplikační vztah se supervizorem je prakticky identický s typem IOC-AR. Navíc však umožňuje převzetí kontroly nad zařízením namísto PLC komunikujícího pomocí zmíněného typu.

c) *Implicitní AR* – zahrnuje pouze komunikační vztah Record Data CR a to pouze v režimu čtení. Na rozdíl od předchozích typů však není nutné navazovat spojení — jedná se pouze o jednorázovou operaci.

Popis protokolu

Navazování aplikačního vztahu probíhá na protokolu UDP/IP a skládá se z následujících kroků:

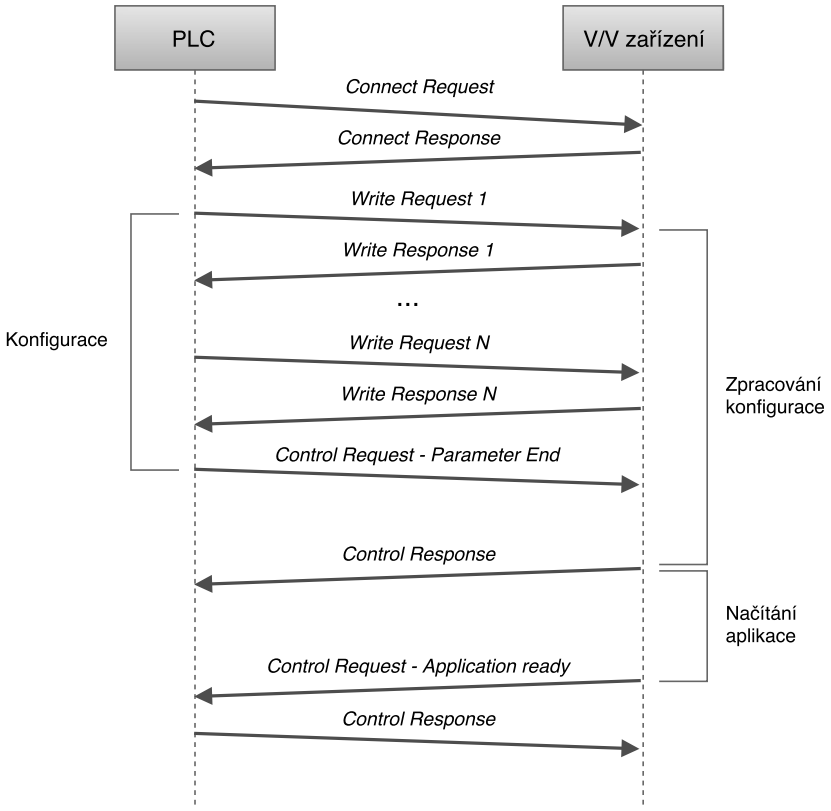
1. *Navázání spojení* – PLC iniciuje komunikaci zasláním požadavku o připojení (angl. „Connect Request“) vybranému V/V zařízení. Požadavek zahrnuje model V/V zařízení doplněn o data jednotlivých komunikačních vztahů. V/V zařízení po zpracování zasílá odpověď o úspěšném připojení (angl. „Connect Response“), která obsahuje potvrzení požadovaných kanálů a případná diagnostická data.

2. *Konfigurace* – předchozím krokem se úspěšně navázal komunikační vztah typu Record Data CR a právě tento kanál je využíván pro konfiguraci V/V zařízení. Ta je provedena zasíláním požadavků na zápis konfiguračních hodnot (angl. „Write Request“). V/V zařízení potvrzuje každý úspěšný i neúspěšný zápis zasláním odpovědi (angl. „Write Response“). V případě, že je některý z požadavků odmítnut, je navazování aplikačního vztahu přerušeno. Zápis konfigurace je ukončen řídícím požadavkem (angl. „Control Request - Parameter End“). Jakmile jsou všechna zapsaná data zpracována, je požadavek potvrzen zasláním řídící odpovědi (angl. „Control Response“).

3. *Spuštění aplikace* – po zpracování konfiguračních dat je V/V zařízením zahájeno načítání uživatelské aplikace. Její úspěšné spuštění je oznámeno zasláním dalšího řídícího požadavku (angl. „Control Request - Application Ready“). Ve chvíli, kdy je tento požadavek potvrzen ze strany PLC zasláním řídící odpovědi, je aplikační vztah úspěšně navázán a k dispozici jsou všechny požadované komunikační kanály.

Při zahájení popsaného procesu naslouchá V/V zařízení na portu 34964. Od druhého kroku již však komunikace pokračuje na portu 49153 až 49156 (konkrétní port se liší v závislosti na typu V/V zařízení a verzi standardu Profinet). V případě úspěšného navázání aplikačního vztahu probíhá na stejném portu také acyklický přenos dat. Dále je potřeba zmínit, že zápis konfiguračních hodnot v druhém kroku je nepovinný — ihned po obdržení odpovědi

o úspěšném připojení může následovat odeslání řídícího požadavku oznamující konec konfigurace. V případě absence konfigurace však není garantovaný správný stav V/V zařízení a proto je tato část prováděna většinou PLC podporujících Profinet. Popsaná komunikace je také znázorněna na obrázku 3.4.



Obrázek 3.4: Navázání aplikačního vztahu [inspirováno z 15]

Při následné výměně dat mezi PLC a V/V zařízením jsou dodržovány dva modely komunikace:

- *klient-server* – server v podobě V/V zařízení pasivně naslouchá, klient v podobě PLC iniciuje spojení. Model se používá pro acyklický přenos datových struktur přes komunikační kanál Record Data CR.
- *konzument-poskytovatel* – poskytovatel zasílá data konzumentovi v pevných intervalech, který je zpracovává. Model se používá pro cyklický přenos uživatelských dat a acyklický přenos diagnostických dat v reálném čase. Konzument i poskytovatel mohou být současně oba účastníci komunikace. Jestliže PLC zasílá data V/V zařízení, jedná se o výstupní data. V opačném směru se jedná o data vstupní.

Datový model a adresování

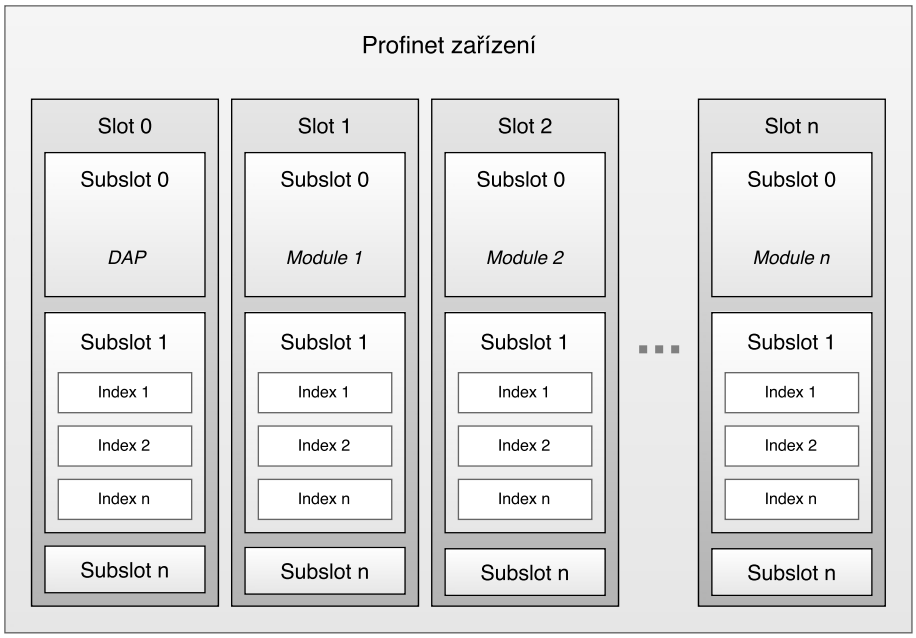
Datový model Profinet zařízení je členěn do následujících úrovni:

- *Slot* – představuje strukturu komponent nebo funkcí a obvykle se jedná o fyzické moduly nebo logické celky uvnitř V/V zařízení.

- *Subslot* – představuje fyzické podmoduly nebo logické celky uvnitř slotu.
- *Index* – představuje data uvnitř subslotu.

V Profinetu je každé zařízení reprezentováno hierarchickou strukturou slotů, subslotů a indexů. Zařízení jako celek je pak dostupné skrz takzvaný DAP (Device Access Point), který mimo jiné zahrnuje data komunikačního rozhraní. Na základě popsaného modelu, který je znázorněn taktéž na obrázku 3.5, umožňuje Profinet následující adresování:

- zařízení jako celek skrz DAP,
- jednotlivé V/V moduly na úrovni slotů,
- jednotlivé V/V kanály ve V/V modulech na úrovni subslotů,
- jednotlivé programové sekce V/V modulů na úrovni indexů.



Obrázek 3.5: Obecný datový model Profinet zařízení [převzato z 22]

K datům jednotlivých struktur lze přistupovat pomocí acyklického datového přenosu kanálem Record Data CR selektivně během navázaného aplikačního vztahu. Jinak je tomu u cyklických dat kanálu IO Data CR, kdy každý přenos obsahuje všechny dostupné informace. Data jednotlivých modulů jsou ve zprávě uspořádány do sekcí, jejichž pozice a velikost byly definovány během navázání aplikačního vztahu. Je potřeba zmínit, že množina dat dostupných pro acyklický přenos nemusí být (a typicky ani není) shodná s množinou dat přenášených cyklicky.

Profinet v SITOP PSU8600

SITOP PSU8600 je Profinet V/V zařízení a jeho datový model je tak navržen dle obecného modelu popsaného v předchozí sekci. Základní jednotka je vždy dostupná na nultém slotu. Rozšiřující moduly pak postupně na slotech následujících dle fyzického zapojení. Subslot 1

každého slotu reprezentuje modul jako celek. V případě základní jednotky navíc obsahuje data relevantní pro celé zařízení. Následující subsloty jsou určené pro komponenty specifické pro daný modul. V případě základní jednotky a rozšiřujících modulů SITOP CNX8600 podmoduly reprezentují jednotlivé výstupy. Chceme-li tak přistoupit například k druhému výstupu prvního rozšiřujícího modulu, slot je roven jedné a subslot třem.

Zařízení nabízí širokou škálu možností pro konfiguraci a diagnostiku. Nejzajímavější položka z pohledu této práce je selektivní vypínání výstupů. Toho lze docílit dvěma způsoby:

a) *Acyklická konfigurace vybraného výstupu* – datová strukturu na indexu `0x1` každého výstupu obsahuje pravdivostní hodnotu určující právě jeho stav. Zasíláním acyklických požadavků o zápis těchto struktur tedy lze kontrolovat stav napájených produktů. Aktuální stav výstupu lze získat zasláním acyklického požadavku na čtení stejné struktury.

b) *Cyklická konfigurace všech výstupů* – výstupní cyklická data, která jsou přenášená mezi PLC a zařízením SITOP PSU8600 obsahují mimo jiné požadované stavy všech výstupů. Vstupní cyklická data poté obsahují jejich aktuální stavy. Modifikací výstupních dat lze tedy taktéž řídit stav napájených produktů. Nevýhodou však je nutnost konfigurace všech dostupných výstupů současně. Výhoda naopak může být okamžitá odezva v podobě vstupních dat.

Kapitola 4

Webová aplikační rozhraní

Webová aplikační rozhraní jsou v dnešní době součástí velkého množství aplikací. A ať už se jedná o globální sociální sítě či malé interní projekty, vždy je cílem poskytnout rozhraní, které je strojově přívětivé a čitelné. Rozhraní, které umožní snadnou implementaci klientské aplikace v jakémkoliv prostředí a využitím libovolných technologií. Zprávy jsou obvykle v XML[1] či JSON[2] formátu a komunikace probíhá modelem požadavek-odpověď využitím protokolu HTTP, který je popsán v podkapitole 4.1. Při návrhu je velmi populární REST architektura, jejíž principy jsou shrnuty v podkapitole 4.2.

4.1 HTTP

HTTP (Hypertext Transfer Protocol) je protokol aplikační vrstvy modelu ISO/OSI určený pro výměnu hypertextových[3] dokumentů a společně se značkovacím jazykem HTML[4] se jedná se o základní stavební kameny technologie World Wide Web [7]. Data jsou přenášená využitím transportního protokolu TCP a je tedy zaručeno spolehlivé doručení zprávy. Služba obvykle naslouchá na portu 80 pro nezabezpečený a na portu 443 pro zabezpečený přenos.

Popis protokolu

Způsob komunikace je v případě HTTP založen na modelu požadavek-odpověď. Protokol je bezstavový, každý požadavek je tedy možné obsloužit nezávisle a bez dalších znalostí o odesílateli. Struktura zprávy požadavku i odpovědi je velmi podobná. Rozdělena je do tří částí:

- úvodní řádek specifikující zprávu,
- hlavičky s atributy,
- volitelné tělo obsahující data.

Úvodní řádek a hlavičky jsou text v ASCII kódování oddělené v řádcích pomocí CRLF. U těla je typ dat uveden v hlavičce `Content-Type`. V případě World Wide Web je to

[1] *XML* (Extensible Markup Language) – značkovací jazyk navržený pro uchovávání a přenos dat v podobě čitelné lidmi i stroji.
[2] *JSON* (JavaScript Object Notation) – způsob zápisu dat určený pro jejich uchovávání či přenos.
[3] *Hypertext* – strukturovaný text, který používá odkazy k propojení dokumentů.
[4] *HTML* (HyperText Markup Language) – standardní značkovací jazyk pro vytváření webového obsahu.

nejčastěji obsah v HTML. Tělo je od hlaviček odděleno dalším ukončením řádku CRLF. Níže lze vidět základní formát (výpis 4.1) a příklad (výpis 4.2) požadavku a taktéž formát (výpis 4.3) a příklad (výpis 4.4) odpovědi.

```
<metoda> <URL> <verze HTTP>
<atribut 1> <hodnota>
...
<atribut N> <hodnota>

<data>
```

Výpis 4.1: Formát HTTP požadavku

```
GET /index HTTP/1.1
User-Agent: Mozilla/4.0
Accept: text/*
Host: www.example.com
```

Výpis 4.2: Příklad HTTP požadavku

```
<verze HTTP> <status> <popis>
<atribut 1> <hodnota>
...
<atribut N> <hodnota>

<data>
```

Výpis 4.3: Formát HTTP odpovědi

```
HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 12

Hello World!
```

Výpis 4.4: Příklad HTTP odpovědi

Metoda, která je první položka HTTP požadavku, oznamuje serveru požadovanou operaci. HTTP specifikace definuje vícero metod. Například metoda `GET` slouží ke stažení obsahu, metoda `POST` pak k odeslání dat serveru ke zpracování. Základní přehled je v tabulce 4.1, kde je mimo jiné uvedeno zda je metoda bezpečná z pohledu změny dat. Bezpečné metody by neměly měnit stav dat serveru a opakované zaslání požadavku by mělo mít stejný následek jako zaslání jednoho. Oproti tomu metody, které bezpečné nejsou, mohou měnit data serveru a potencionálně tak vyvolat i nežádoucí chování. Nutno podotknout, že informace uvedené v tabulce 4.1 nemusí být striktně dodržovány a vždy záleží na implementaci konkrétního serveru. Ten taktéž nemusí podporovat všechny uvedené metody a naopak může být rozšířen o vlastní rozšiřující jako doplněk k těm obsaženým v HTTP specifikaci.

Metoda	**Účel**	**Tělo**	**Bezpečná**
`GET`	Získání dokumentu ze serveru	✗	✓
`POST`	Zaslání dat serveru ke zpracování	✓	✗
`PUT`	Uložení těla požadavku na serveru	✓	✗
`DELETE`	Odstranění dokumentu ze serveru	✗	✗
`HEAD`	Získání pouze hlavičky dokumentu	✗	✓
`TRACE`	Sledování zprávy skrz Proxy servery	✗	✓
`OPTIONS`	Výpis serverem podporovaných metod	✗	✓

Tabulka 4.1: Přehled základních HTTP metod [inspirováno z 7]

Výsledek požadované operace je popsán v odpovědi statusem reprezentovaným tříciferným číslem následovaný člověkem čitelným popisem. První číslice statusu popisuje jeho zařazení do jedné z pěti kategorií:

- *Informace* (100-199) – kódy informativního charakteru, jejichž využití ovšem není příliš rozšířené.
- *Úspěch* (200-299) – kódy oznamující úspěšné provedení požadavku. Nejčastěji se lze setkat s kódy 200 (OK), 201 (úspěšně vytvořeno) a 204 (žádný obsah).
- *Přesměrování* (300-399) – kódy sloužící k oznámení alternativní lokace pro získání dokumentu v případě přesunutí obsahu. Odpověď je obvykle doplněna o hlavičku `Location` upřesňující novou lokaci.
- *Selhání klienta* (400-499) – kódy určené pro chyby na straně klienta. Jedná se například o případy, kdy server nedokáže zpracovat požadavek z důvodu špatného formátu (kód 400) nebo kdy server nemůže nalézt obsah pro požadovanou URL (kód 404).
- *Selhání serveru* (500-599) – kódy této kategorie indikují validní požadavek klienta, který však vyvolal chybu na straně serveru.

Dalšími elementy ve zprávě jsou již několikrát zmíněné hlavičky, které upřesňují operace prováděné mezi klientem a serverem. HTTP definuje hlavičky, které jsou specifické pro typ zasílané zprávy, ale také hlavičky s širokým využitím. Shrnout je lze do pěti hlavních tříd:

- *Obecné hlavičky* – poskytují základní informace o zprávě, které jsou užitečné pro klienta i server. Příkladem může byt hlavička `Date` obsahující datum a čas vytvoření zprávy.
- *Hlavičky požadavku* – poskytují doplňující informace pro server jako například jaký typ dat klient akceptuje (`Accept`) či jakou aplikací byl požadavek odeslán (`User-Agent`).
- *Hlavičky odpovědi* – poskytují klientovi další informace o serveru či další instrukce spojené s odpovědí. Jako příklad lze uvést položku `Server`, která obsahuje jméno a verzi aplikace použité na straně serveru.
- *Hlavičky těla* – popisují obsah těla zprávy. Množina hlaviček se může lišit v závislosti na typu přenášených dat. Mezi nejpoužívanější patří `Content-Type` obsahující typ dat a `Content-Length` obsahující jejich délku.
- *Rozšiřující hlavičky* – nestandardní hlavičky vytvořené vývojáři, které nejsou součástí HTTP specifikace a umožňují rozšířit funkcionalitu na straně serveru či klienta.

Autentizace

HTTP zahrnuje několik autentizačních technik. Server iniciuje autentizační výzvu zasláním odpovědi se statusem 401 a hlavičkou `WWW-Authenticate` oznamující typ autentizace. Klient poté požadavek opakuje s hlavičkou `Authorization` obsahující požadované údaje. Základní autentizační metody jsou:

a) *Základní autentizace* (angl. „Basic Authentication“) – nejjednodušším typem HTTP autentizace, ve které může server odmítnout transakci a vyzvat uživatele k zaslání přihlašovacích údajů. Uživatelské jméno a heslo je odděleno dvojtečkou a zakódováno do Base64. Formát hlavičky `Authorization` je následující:

```
Authorization: Basic Base64(<login>:<heslo>)
```

b) *Digest autentizace* (angl. „Digest Authentication“) – podobná metoda jako základní autentizace, přihlašovací údaje jsou však namísto Base64 kódování zahašovány společně s požadovanou HTTP metodou, URL a libovolným jednorázovým číslem poskytnutým serverem. Použit je algoritmus MD5.

c) *Autentizace na základě přístupového tokenu* (angl. „Token Based Authentication“) – poměrně rozšířený způsob autentizace, kdy klient obdrží přístupový token po zaslání validních přihlašovacích údajů (obvykle jedním z výše popsaných způsobů). Získaný token je poté zasílán v požadavku jako součást hlavičky `Authorization` a opravňuje klienta provádět chráněné operace po určitou dobu bez použití uživatelského jména a hesla. Formát hlavičky je následující:

```
Authorization: Bearer <token>
```

Cílem této metody je snížení počtu zasílání přihlašovacích údajů a tedy i rizika jejich odcizení. Autentizace na základě přístupového tokenu bývá často nazývána také jako Bearer autentizace (angl. „Bearer Authentication“), kdy je odkazováno právě na použitý typ hlavičky `Authorization`.

Ani jeden ze způsobů výše však nelze považovat za bezpečný — kódování Base64 není šifrování a algoritmus MD5 je již prolomený. Proto by se tyto metody měly používat výhradně v kombinaci s šifrovaným spojením, které lze zajistit pomocí kryptografického protokolu TLS[5].

4.2 REST

REST (Representational state transfer) definuje principy pro návrh webových služeb [20, 4]. Z pohledu počtu webových služeb založených na REST se v posledních letech jedná o dominantní návrhový model. REST architektura vychází z šesti základních principů:

- *Uniformní rozhraní* – definuje rozhraní mezi klientem a serverem a zjednodušuje architekturu tak, aby každá část byla vyvíjená samostatně.
- *Bezstavovost* – informace o kontextu potřebné k vyřízení požadavku jsou obsaženy v požadavku samotném. Libovolný požadavek tak lze zpracovávat nezávisle a bez nutnosti udržovat si vnitřní stav o aktivních klientech. Bezstavovost, vzhledem k oddělenýmodpovědnostem, umožňuje jednodušší návrh, lepší údržbu i škálovatelnost serveru.
- *Využití dočasné paměti* – odpovědi serveru musí obsahovat informaci, zda je možné je uložit do dočasné paměti (angl. „cache“). Dobře navrženým ukládáním odpovědi lze výrazně omezit interakci se serverem a tedy zlepšit výkonnost.
- *Klient-server model* – uniformní rozhraní rozděluje systém na klienta a server. Klient tak nemusí řešit operace spojené s ukládáním dat, což zlepšuje přenositelnost. Server se naopak nezatěžuje uživatelským rozhraním a stavem klienta, což zjednodušuje

[5] *TLS* (Transport Layer Security) – kryptografický protokol, který zajišťuje bezpečnost při síťové komunikaci.

finální implementaci a umožňuje lepší škálovatelnost. Vývoj klienta a serveru může taktéž probíhat oddělené za předpokladu, že není změněno uniformní rozhraní.

- *Vrstvený systém* – systém je rozdělen na nezávislé komponenty. Klient tak není schopen určit, zda je připojen přímo ke koncovému serveru či k prostředníku. Z pohledu serveru je tak opět zvýšená škálovatelnost a umožněno využití techniky vyvažování zátěže či bezpečnostních politik.
- *Kód na požádání* – server je schopen dočasně rozšířit funkcionalitu klienta zasláním určité logiky, kterou klient provede. Může se jednat například o skript v jazyce JavaScript.

V případě, že navržené rozhraní splňuje všechny uvedené principy (kód na požádání je volitelný), jedná se o takzvané *RESTful* rozhraní.

Základní charakteristiky

Jednou z klíčových charakteristik webové služby založené na REST modelu je používání HTTP metod výhradně způsobem definovaným v RFC 2616. Stručný přehled HTTP metod a jejich určení je součástí předchozí podkapitoly 4.1, konkrétně tabulky 4.1. REST tímto základním principem jednoznačně určuje vztahy mezi HTTP metodami a operacemi typu čtení, vytvoření, editace a odstranění:

- `GET` – k získání zdroje,
- `POST` – pro vytvoření zdroje,
- `PUT` – pro editování nebo změny stavu zdroje,
- `DELETE` – pro odstranění zdroje.

Důraz je taktéž kladen na to, aby metoda `GET` zůstala idempotentní a nevyvolávala žádné postranní efekty.

U adresování zdrojů je pak cílem maximální použitelnost a intuitivnost. Jeden ze způsobu jak toho lze dosáhnout jsou URI ve formě adresářové struktury. Jedná se o hierarchickou strukturu s jedním kořenovým adresářem, odkud se větví na podadresáře věnující se konkrétním oblastem služby. URI tedy není jen řetězec identifikující určitý dokument, ale spíše strom s nadřazenými a podřazenými uzly. Příklad URI může být následující:

```
http://www.example.com/cars/skoda/octavia
```

Kořen `/cars` má pod sebou uzel `skoda`, který obsahuje konkrétní zdroje. V případě `GET` požadavku na URI `/cars/skoda/` pak získáme kolekci všech zdrojů, které jsou hierarchicky pod daným uzlem. Kromě právě popsané struktury by server neměl dále definovat žádné další procedury jako například `/add-user` — operace jsou již definovány HTTP metodami. Další z doporučených postupů zahrnuje skrytí koncovek souborů, nepoužívání předávání argumentů jako součást URL a podobně.

Data požadavků i odpovědí jsou nejčastěji ve formátu JSON či XML. Kromě samotných atributů požadovaného zdroje jsou obvykle v datech obsaženy také hypertextové odkazy. Klientské aplikaci je tak umožněna navigace mezi souvisejícími zdroji. V dnešní době již navíc existuje velké množství nadstaveb pro zmíněné formáty, které již zahrnují použití hypertextu či jsou přizpůsobeny pro specifické oblasti využití.

Důležitým avšak často opomíjeným prvkem REST jsou profily. Zjednodušeně řečeno se jedná o specifikaci rozhraní, která je dostupná na serveru. Obsahem je obvykle popis dostupných zdrojů, podporované HTTP metody, možné stavové kódy odpovědí, validní formáty dat a tak dále. Oproti běžné dokumentaci jsou však profily strojově čitelné. Běžná praxe je tak importování profilu do klientské aplikace, kde je automaticky vygenerována funkcionalita potřebná ke komunikaci se serverem. To může výrazně zrychlit nejen vývoj klienta, ale také zjednodušit jeho následnou údržbu.

4.3 Alternativní technologie

REST aplikační rozhraní samozřejmě nejsou jedinou možností pro vytvoření webového aplikačního rozhraní. Za zmínku stojí minimálně dvě další používané technologie:

- *SOAP* (Simple Object Access Protocol) – na rozdíl od REST, který je jen souborem pravidel a doporučení, je SOAP plnohodnotný protokol. Zprávy jsou ve formátu XML a přenášené jsou pomocí protokolů aplikační vrstvy HTTP nebo SMTP[6].
- *GraphQL* – je dotazovací jazyk pro aplikační rozhraní vyvinutý společností Facebook. Hlavním rozdílem oproti REST je, že GraphQL je vázaný pouze na jednu URL a umožňuje klientům definovat v rámci požadavku kompletní strukturu vrácených dat.

Zatímco SOAP je v prostředí webových služeb považováno za předchůdce REST aplikačních rozhraní, kterými byl postupně vytlačován, GraphQL je relativně novou technologií s velkým příslibem do budoucnosti. Vzhledem k jednoduchosti a aktuálnímu rozšíření však bylo před výše uvedenými možnostmi upřednostněno REST aplikační rozhraní.

[6] *SMTP* (Simple Mail Transfer Protocol) – internetový standard určený pro přenos zpráv elektronické pošty.

Kapitola 5

Návrh systému

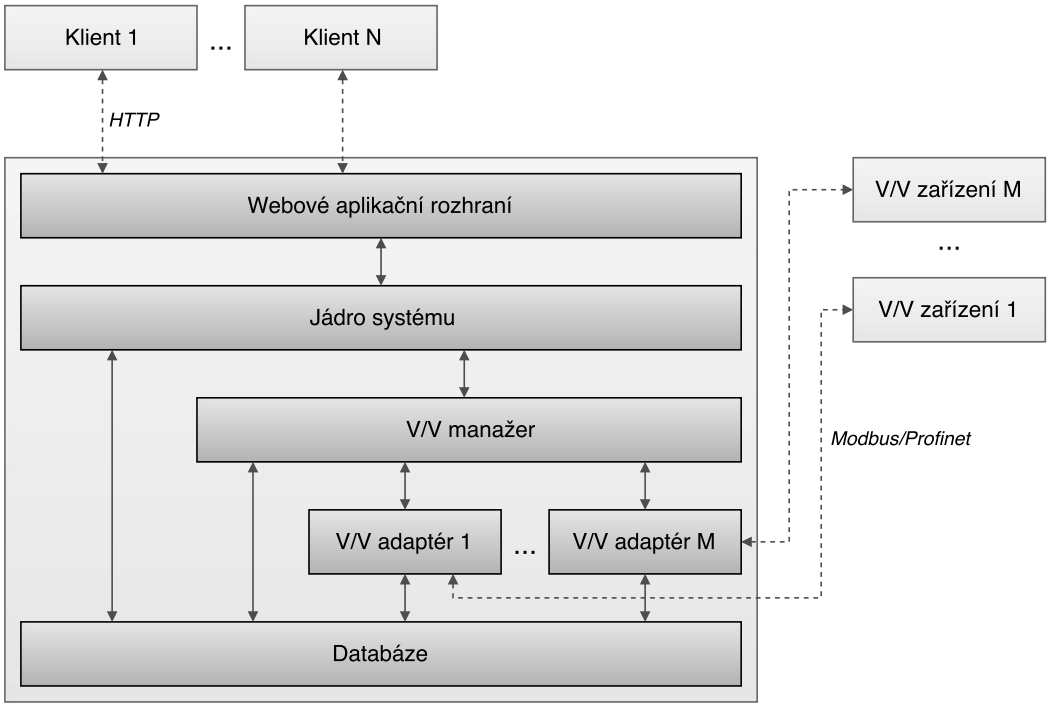
Na základě analýzy požadavků a teoretických poznatků popsaných v předchozích kapitolách je v této kapitole navržen systém pro sdílení a konfiguraci prostředků. Na úvod je v podkapitole 5.1 představena struktura systému a jeho jednotlivé komponenty včetně vzájemných vztahů. Následně je v podkapitole 5.2 popsán návrh databáze. Podkapitola 5.3 se poté věnuje integraci požadovaných V/V zařízení do vyvíjeného systému. Na závěr jsou v podkapitole 5.4 shrnuty případy užití a navrženo aplikační rozhraní.

5.1 Struktura systému

Systém je navržen jako klient-server aplikace. Pro komunikaci budou využity metodiky aplikačních rozhraní REST a komunikační protokol HTTP. Jako klient je označována jakákoliv aplikace, která bude zasílat na server požadavky skrz zmíněné rozhraní a vizualizovat obdržené odpovědi. Serverová část systému se pak skládá z následujících komponent:

- *Databáze* – představuje nejnižší vrstvu systému jejíž účelem je zajištěni persistence dat. Návrhu databáze včetně ER diagramu se dále věnuje podkapitola 5.2.
- *Jádro systému* – obsahuje hlavní funkcionalitu systému a provádí veškeré požadavky předané z webového aplikačního rozhraní. Výsledky jsou perzistentně ukládány využitím databázové vrstvy. V případě operací s V/V zařízením komunikuje s V/V manažerem.
- *Webové aplikační rozhraní* – představuje nejvyšší vrstvu a vstupní bod systému. Hlavním účelem je zpracování požadavku klienta a vyvolání požadované operace v jádru systému. Na této úrovni je taktéž zajištěna autentizace uživatele a validace zasílaných dat včetně reakcí v případě selhání. Navržené aplikační rozhraní je dále popsáno v podkapitole 5.4.
- *V/V manažer* – zastřešuje veškeré operace s V/V zařízeními. Na základě dat získaných z databáze inicializuje příslušné V/V adaptéry a následně mezi ně distribuuje požadavky vyšších vrstev.
- *V/V adaptér* – slouží pro komunikaci s konkrétním V/V zařízením. Pro každé zařízení je vytvořen samostatný adaptér. Požadavky jsou mu předávány přes FIFO frontu.

Popsanou strukturu lze vidět taktéž na obrázku 5.1.



Obrázek 5.1: Návrh systému [zdroj vlastní]

5.2 Databáze

Pro zajištění persistence dat byla zvolena relační databáze. Model navržený dle požadavků obsahuje pět následujících entit:

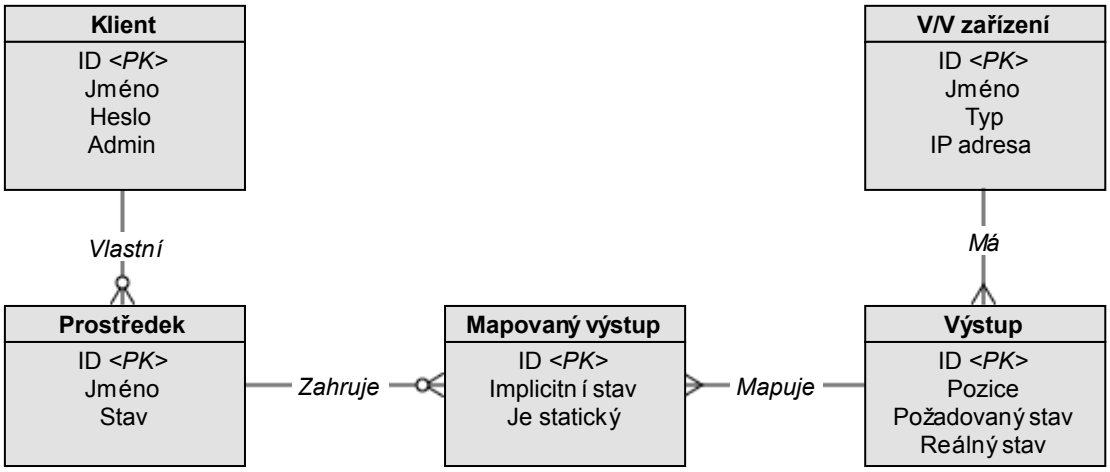
- klient,
- prostředek,
- mapovaný výstup,
- výstup,
- V/V zařízení.

Každá entita obsahuje atribut *ID* sloužící jako primární klíč. Pro všechny další atributy s výjimkou cizích klíčů platí integritní omezení neprázdnosti. Účel a atributy konkrétních entit jsou blíže popsány v příslušných sekcích této podkapitoly. Všechny entity a vztahy mezi nimi jsou taktéž znázorněny ER diagramem na obrázku 5.2.

Klient

Entita reprezentuje klienta systému, kterým může být libovolný uživatel či automatizovaný nástroj. Účel entity je především autentizace a následně udržování informace o aktuálně vlastněných prostředcích. Entita obsahuje následující atributy:

- *Jméno* – řetězec sloužící společně s heslem k autentizaci klienta. Pro tento atribut platí integritní omezení unikátnosti.



Obrázek 5.2: ER diagram [zdroj vlastní]

- *Heslo* – řetězec sloužící společně se jménem k autentizaci klienta.
- *Admin* – pravdivostní hodnota určující zda má klient administrátorská práva.

Prostředek

Entita reprezentuje sdílený prostředek, který může být alokován libovolným klientem. Rezervace prostředků je vyjádřena vztahem s entitou klienta. Absence tohoto vztahu indikuje, že prostředek je volný. Jeden klient může v danou chvíli vlastnit vícero prostředků, jeden prostředek však může být vlastněn pouze jedním nebo žádným klientem. Entita zahrnuje 0 až N mapovaných výstupů, které slouží k navázání jednotlivých výstupů V/V zařízení k danému prostředku. Atributy entity jsou následující:

- *Jméno* – řetězec sloužící k identifikaci a textové reprezentaci daného prostředku. Pro tento atribut platí integritní omezení unikátnosti.
- *Stav* – hodnota reprezentující aktuální stav prostředku získaný na základě stavů jednotlivých mapovaných výstupů. Stav může nabývat čtyř různých hodnot — všechny výstupy jsou v kladném stavu, všechny výstupy jsou v záporném stavu, výstupy jsou částečné v kladném a částečné záporném stavu a nelze tak jednoznačně určit stav prostředku, anebo není momentálně možné získat stav daných výstupů a proto je stav neznámý.

Mapovaný výstup

Jedná se o pomocnou entitu mezi entitou *Prostředek* a entitou *Výstup* a slouží k mapování výstupů V/V zařízení v daném prostředku. Umožňuje tak použití jednoho výstupu ve vícero prostředcích a naopak vícero výstupů v jednom prostředku. Entita obsahuje následující atributy:

- *Implicitní stav* – pravdivostní hodnota určující stav výstupu V/V zařízení v implicitním stavu prostředku.

- *Je statický* – pravdivostní hodnota určující, zda je stav daného výstupu V/V zařízení měněn při operacích s daným prostředkem nebo zda je ponechán v implicitním stavu.

Výstup

Entita reprezentuje jeden konkrétní výstup V/V zařízení a obsahuje následující atributy:

- *Pozice* – číselná hodnota reflektující pozici výstupu na konkrétním V/V zařízení.
- *Požadovaný stav* – pravdivostní hodnota určující požadovaný stav výstupu.
- *Reálný stav* – hodnota reflektující reálný stav výstupu, která může nabývat tří různých hodnot — výstup je v kladném stavu, výstup je v záporném stavu, anebo není momentálně možné získat stav výstupu a proto je neznámý.

V/V zařízení

Entita reprezentuje reálné V/V zařízení s alespoň jedním výstupem a obsahuje následující atributy:

- *Jméno* – řetězec sloužící k identifikaci a textové reprezentaci daného prostředku. Pro tento atribut platí integritní omezení unikátnosti.
- *Typ* – řetězec označující typ V/V zařízení. Na základě této hodnoty je zvolen vhodný V/V adaptér k zajištění komunikace.
- *IP adresa* – řetězec ve formátu IPv4. IP adresa je použita ve V/V adaptéru k navázání komunikace s V/V zařízením.

5.3 Integrace V/V zařízení

Při integraci V/V zařízení je hlavním cílem umožnit jednoduchou interakci jádra systému s aktivními zařízeními a to nezávisle na jejich typu či použitém komunikačním protokolu. To je zajištěno pomocí dvou typů komponent — V/V manažer a V/V adaptér. Obě tyto komponenty jsou blíže popsány právě v této podkapitole.

V/V manažer

Všechny operace, které vyžadují interakci s V/V zařízeními, komunikují primárně s V/V manažerem. Komponenta slouží jako prostředník, který poskytuje uniformní rozhraní pro práci s V/V adaptéry. Zahrnuta je následující funkcionalita:

- získání výčtu podporovaných V/V zařízení,
- inicializace adaptéru pro libovolné podporované V/V zařízení,
- zrušení libovolného existujícího adaptéru,
- změna stavu výstupu pomocí libovolného existujícího adaptéru,
- získání aktuálního stavu adaptéru (například stavu připojení).

Jelikož komponenta spravuje globálně všechny V/V adaptéry, je navržena jako *Singleton*. V systému se tedy vyskytuje pouze jedna instance tohoto typu.

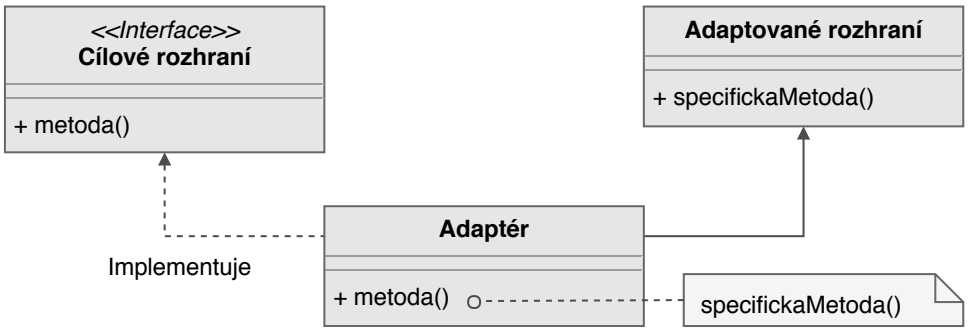
V/V adaptér

Účel komponenty je obsluha jednoho V/V zařízení. Počet V/V adaptéru v systému je tedy ekvivalentní k počtu připojených zařízení. Komponenta je zodpovědná za operace s obsluhovaným zařízením jako:

- navázání spojení,
- změna stavu libovolného výstupu,
- čtení aktuálního stavu libovolného výstupu,
- získání diagnostických informací.

Kromě V/V manažera, který iniciuje vykonání operací uvedených výše, komunikuje adaptér taktéž s databázovou vrstvou. Té jsou předávány relevantní data získaná ze zařízení.

Aby bylo zachováno jednotné rozhraní pro přístup k libovolnému V/V zařízení, vychází komponenta z návrhového vzoru Adaptér [5]. Ten je určen k převodu rozhraní určité třídy na jiné rozhraní, které je očekáváno. Adaptér tak umožňuje interoperabilitu tříd, která by jinak nebyla z důvodu nekompatibilních rozhraní možná. Jedná se tak o vhodné řešení pro integraci libovolného V/V zařízení. Jednoduchá demonstrace zmíněného návrhové vzoru je na obrázku 5.3.



Obrázek 5.3: Návrhový vzor Adaptér [inspirováno z 5]

Operace uvedené na začátku této sekce jsou vyvolávány na základě požadavků skrz webové aplikační rozhraní. Lze tedy předpokládat paralelní přístup více vláken. V případě síťové komunikace s V/V zařízením však není zaručena bezpečná konkurence. Požadavky jsou z toho důvodu vkládány do FIFO fronty, přičemž jsou V/V adaptérem vykonávány sekvenčně. Veškeré operace s FIFO frontou zároveň zahrnují výlučný přístup.

Při sekvenčním vykonávání však může vzniknout problém při vyšším počtu požadavků. V závislosti na rychlosti komunikace s V/V zařízením a aktuálním počtu požadavků ve frontě může být doba mezi obdržením nového požadavku na úrovni webového rozhraní a odesláním odpovědi příliš velká. To může vyvolat i vypršení časového limitu na straně klientské aplikace. Z toho důvodu se v návrhu předpokládá, že každý V/V adaptér bude implementován jako samostatné vlákno. Díky tomu není nutné čekat na vykonání požadavku. Odpověď lze odeslat prakticky ihned po vložení požadavku do fronty, zatímco operace se bude vykonávat asynchronně.

5.4 Aplikační rozhraní

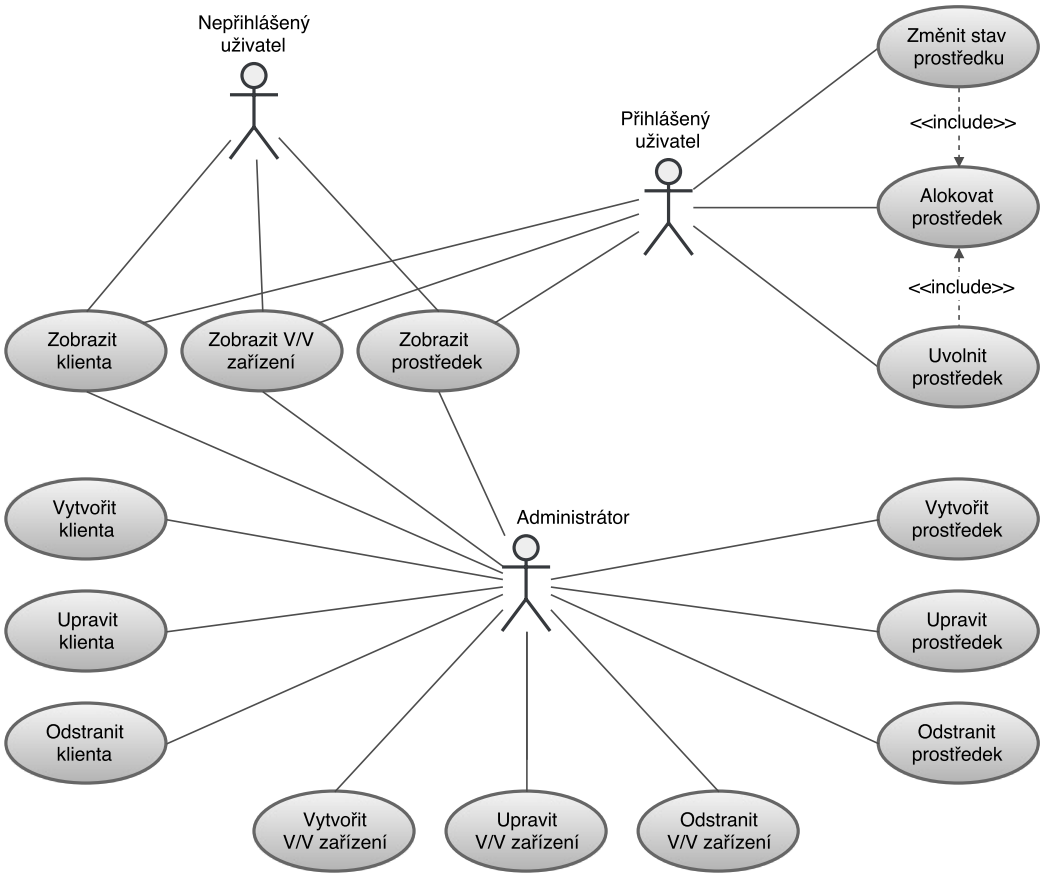
Aplikační rozhraní představuje vstupní bod do systému a při návrhu proto bylo vycházeno zejména z požadavků cílových uživatelů. Ti budou v systému rozděleni do tří uživatelských rolí s rozdílným oprávněním:

- nepřihlášený uživatel,
- přihlášený běžný uživatel,
- administrátor.

Operace dostupné v systému lze taktéž rozdělit do tří kategorií:

- prohlížení obsahu,
- operace na úrovni informačního systému (vytváření, odstraňování, editace),
- operace na úrovni sdílení prostředků (alokace, uvolňování, změna stavu).

Role a operace popsané výše jsou pomocí diagramu případů užití znázorněny na obrázku 5.4. Pro zjednodušení diagramu je předpokládáno, že se uživatel úspěšně autentizoval do systému.



Obrázek 5.4: Diagram případů užití [zdroj vlastní]

Jak již bylo zmíněno, vrstva aplikačního rozhraní je navržena jako REST webová služba. Hlavní entity systému — tedy klienti, sdílené prostředky a V/V zařízení — jsou reprezentovány jako kolekce, do kterých lze přidávat nové prvky. Existující prvky pak lze odstranit či upravit. K dispozici jsou také operace pro autentizaci uživatele a operace zaměřené na hlavní funkcionalitu systému, tedy sdílení prostředků. V tabulce 5.1 jsou přehledně zobrazeny všechny uvažované zdroje včetně použité metody HTTP požadavku a návratového stavového kódu HTTP odpovědi v případě úspěšného provedení. V návrhu se taktéž předpokládá, že v případě požadavků typu `POST` a `PUT` budou přiloženy data ve formátu JSON, která blíže specifikují požadovanou operaci. Příkladem může být vytvoření nového klienta, kdy přiložená data budou obsahovat jeho jméno, heslo a roli.

URI	**Metoda**	**Kód**	**Popis**
/login	`POST`	200	Přihlášení klienta
/logout	`POST`	200	Odhlášení klienta
/clients	`GET`	200	Získat seznam všech klientů
/clients	`POST`	201	Vytvořit nového klienta
/clients/{jméno}	`GET`	200	Získat vybraného klienta
/clients/{jméno}	`DELETE`	204	Odstranit vybraného klienta
/clients/{jméno}	`PUT`	200	Upravit vybraného klienta
/devices	`GET`	200	Získat seznam všech V/V zařízení
/devices	`POST`	201	Vytvořit nové V/V zařízení
/devices/{jméno}	`GET`	200	Získat vybrané V/V zařízení
/devices/{jméno}	`DELETE`	204	Odstranit vybrané V/V zařízení
/devices/{jméno}	`PUT`	200	Upravit vybrané V/V zařízení
/resources	`GET`	200	Získat seznam všech prostředků
/resources	`POST`	201	Vytvořit nový prostředek
/resources/{jméno}	`GET`	200	Získat vybraný prostředek
/resources/{jméno}	`DELETE`	204	Odstranit vybraný prostředek
/resources/{jméno}	`PUT`	200	Upravit vybraný prostředek
/resources/{jméno}/owner	`POST`	200	Alokovat vybraný prostředek
/resources/{jméno}/owner	`DELETE`	200	Uvolnit vybraný prostředek
/resources/{jméno}/state	`POST`	200	Změnit stav vybraného prostředku

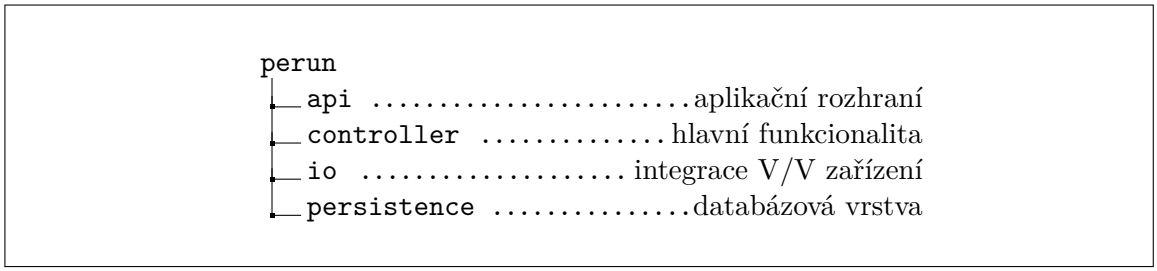
Tabulka 5.1: Uvažované operace webového aplikačního rozhraní [zdroj vlastní]

Kapitola 6

Implementace

Systém byl implementován dle návrhu popsaném v kapitole 5. Pro implementaci byl zvolen jazyk Python [19, 16]. V případě externího zadavatele se jedná o hlavní programovací jazyk pro vývoj interních projektů, což byl jeden z hlavních rozhodovacích faktorů. Během vývoje jsem se navíc snažil řešení co nejvíce zobecnit a umožnit použití vícero technologií. A právě jazyk Python se ukázal být pro tento záměr vhodný, ať už z důvodu, že se jedná o jazyk multiplatformní nebo z důvodu existence velkého množství knihoven pro nejrůznější využití.

Serverová část systému je dle návrhu rozdělena do několika částí. Základní struktura je znázorněna na obrázku 6.1. Všechny klíčové komponenty implementace jsou popsány právě v této kapitole a to ve směru odspoda nahoru. Nejprve je v podkapitole 6.1 popsána databázová vrstva. Následuje integrace požadovaných V/V zařízení v podkapitole 6.2 a samotné jádro systému, kterému je věnována podkapitola 6.3. Implementace REST aplikačního rozhraní je následně shrnuta v podkapitole 6.4. S touto podkapitolou je úzce spjatá autentizace, které je věnována hned další podkapitola 6.5. Na závěr této části je v podkapitole 6.6 popsána implementace klientské části systému. Kompletní strukturu klientské i serverové části systému lze vidět v příloze na obrázku A. Seznam všech použitých knihoven pak v příloze B.



Obrázek 6.1: Základní struktura serverové části systému [zdroj vlastní]

6.1 Databázová vrstva

Pro implementaci databázové vrstvy systému byla použita knihovna *SQLAlchemy*, která umožňuje abstrahovat kód aplikace od připojené databáze [14, 23]. To byl také hlavní důvod jejího použití — nezávislost implementace databázové vrstvy na typu použité databáze. Při vývoji byla využívána databáze PostgreSQL, základní testování pak bylo prováděno i na databázích SQLite a MySQL. Knihovna SQLAlchemy podporuje i další databáze, mimo jiné

Oracle a Microsoft SQL Server. Knihovna i její konkrétní aplikace v systému je popsána v následujících sekcích.

SQLAlchemy

SQLAlchemy je knihovna pro interakci s velkým množstvím databází. Umožňuje vytvořit datový model a následně se dotazovat způsobem jako by se jednalo o běžné objekty jazyka Python. Knihovna využívá obecné konstrukce a datové typy a zajišťuje tak, že jsou její SQL dotazy podporovány většinou databází, aniž by to muselo být při vývoji explicitně řešeno. Zároveň ošetřuje všechny vstupy před provedením v databázi, čímž zabraňuje běžným problémům jako například útoku vložení SQL (angl. „SQL injection“). K dispozici jsou dva hlavní způsoby použití:

a) *SQLAlchemy Core* – dívá se na data jako na schéma. Podobně jako běžné SQL je tedy zaměřené na struktury jako jsou tabulky, klíče, indexy a podobně. Jeho hlavní využití je v datových skladech, analýzách a dalších scénářích, kde je užitečné pracovat s konkrétními dotazy nebo s nemodelovanými daty. Silná vazba na databázi je vhodná pro zpracování velkého množství dat a to i z více databází současně. Pro dotazování se zde používá *SQL Expression Language*, který je pouze mírnou abstrakcí běžného SQL jazyku. Hlavní účel je zajištění jednotného dotazovacího jazyku pro velké množství databází běžících v pozadí.

b) *SQLAlchemy ORM* – poskytuje vysokoúrovňovou abstrakci pro SQL Expression Language způsobem jako objektově relační mapování v jiných jazycích. Je tedy zaměřeno na reprezentaci struktur relačních databází v podobě elementů typických pro objektové orientované programovací jazyky – obvykle tedy jako instance určitých tříd. Zapouzdření této logiky tak dává při interakci s databází pocit běžných operací v jazyce Python. Díky tomu se lze při vývoji více soustředit na business logiku aplikace. Tento přístup se taktéž, oproti čistým SQL dotazům, více hodí při vkládání podpory databáze do existující aplikace. I přesto, že ORM přístup je velmi užitečný, jsou mezi běžnými třídami a tím, jak SQLAlchemy mapuje relační databáze, určité rozdíly, které mohou být při nevhodném použití kontraproduktivní.

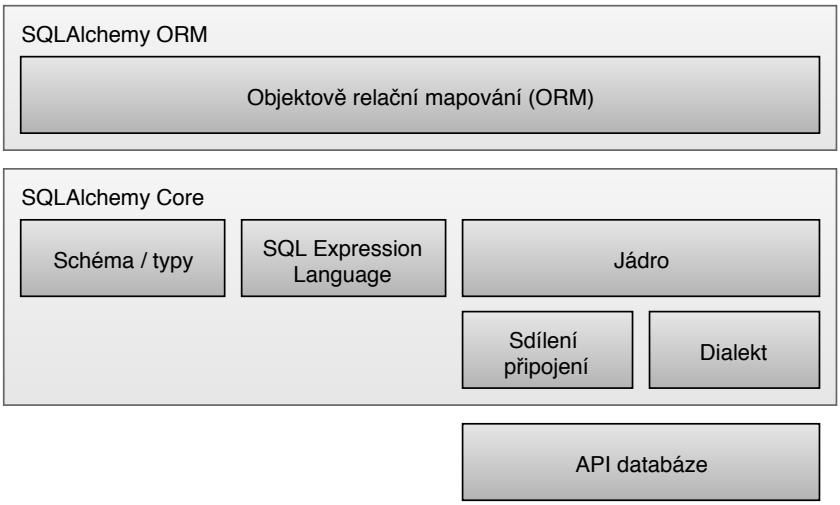
Jak lze vidět na obrázku 6.2, kde jsou znázorněny jednotlivé komponenty knihovny, SQLAlchemy ORM je založeno na SQLAlchemy Core. To umožňuje oba zmíněné přístupy kombinovat v závislosti na potřebách vyvíjené aplikace.

Popis implementace

Jak již bylo řečeno, při implementaci databázové vrstvy byla použita knihovna SQLAlchemy. Z výše uvedených způsobů použití byl zvolen druhý zmíněný – SQLAlchemy ORM. Databáze patří z pohledu množství dat spíše mezi menší a zároveň není nutné explicitně kontrolovat databázové dotazy za účely optimalizace. Hlavní výhody SQLAlchemy Core tak nejsou pro systém žádným výrazným přínosem. Naopak vyšší abstrakce u SQLAlchemy ORM umožní jednodušší konstrukce a tedy i přehlednější a efektivnější implementaci, což je výhodné nejen při vývoji, ale taktéž při následné údržbě či rozšiřování systému.

Implementace vrstvy interagující s databází je obsažena v balíčku `perun.persistence` a skládá se ze tří modulů:

- `model` – obsahuje model systému dle návrhu popsaném v podkapitole 5.2.



Obrázek 6.2: Komponenty knihovny SQLAlchemy [inspirováno z 23]

- `db` – obsahuje třídu `Database`, která iniciuje knihovnu SQLAlchemy a poskytuje základní operace.
- `event_hooks` – obsahuje funkce, které jsou vázané na určité události v databázi (například změna vybraného atributu).

Vedle očekávané funkcionality umožňuje balíček také vytvořit kompletní databázové schéma či přepsat to stávající na základě modelu definovaném v modulu `model`. Samotná interakce s databází probíhá skrz *SQLAlchemy session*. To zapouzdřuje spojení s databází, načtené objekty a transakci, která je otevřená dokud není potvrzena operací `COMMIT` či zrušena operací `ROLLBACK`.

V případě ukládání uživatelských hesel je využita knihovna *Passlib*, která mimo jiné poskytuje algoritmy pro hašování a následné ověřování hesel. Použitý je algoritmus SHA-256 pro 32 bitové systémy a SHA-512 pro 64 bitové systémy.

6.2 Integrace V/V zařízení

Integrace V/V zařízení je implementována v balíčku `perun.io` a vychází z návrhu popsaného v podkapitole 5.3. Obsahuje tedy dva hlavní typy komponent — V/V manažera a V/V adaptéry.

Prvně zmíněná komponenta V/V manažer je implementována v modulu `io.dispatcher` a jedná se o výhradní rozhraní pro komunikaci s připojenými zařízeními. Poskytovány jsou všechny dříve uvedené funkce jako inicializace a zrušení adaptéru, změna stavu libovolného výstupu a tak dále. Při zaslání požadavku je V/V manažeru ve většině případech předáván objekt knihovny *SQLAlchemy* reprezentující V/V zařízení, případně jeho výstup. Manažer následně distribuuje požadavek odpovídajícímu adaptéru na základně svého interního stavu. Komponenta si uchovává veškeré aktivní adaptéry v datové struktuře typu slovník, kdy klíčem je databázové ID V/V zařízení a hodnotou pak právě příslušný adaptér. Díky GIL[1]

[1] *GIL* (Global Interpreter Lock) – mechanismus používaný CPython interpretem, který zajišťuje, že operace s datovým modelem jazyka (včetně seznamů a slovníků) jsou implicitně bezpečné při souběžném přístupu.

mechanismu jazyka Python není navíc potřeba explicitně řešit synchronizaci při souběžném přístupu — všechny operace se zmíněným slovníkem jsou prováděny atomicky.

Adaptéry pro všechna podporovaná V/V zařízení jsou obsaženy v balíčku `io.adapters`. Zde je využita silná stránka jazyka Python, kdy je již za běhu systému provedena inspekce zmíněného balíčku a adaptéry jsou importovány dynamicky. Díky tomu je velmi jednoduché přidat podporu pro další V/V zařízení bez nutnosti zasahovat do dalších komponent. Pro rozšíření o další adaptér je potřeba pouze přidat třídu v daném balíčku, která implementuje uniformní rozhraní zděděné z třídy abstraktního adaptéru. Adaptér se po restartování serveru načte nejen do komponenty V/V manažera, ale také do REST aplikačního rozhraní pro účely validace požadavků. Načítání adaptérů je implementováno v inicializačním modulu `adapters.__init__`. Abstraktní adaptér a všechny aktuálně implementované adaptéry jsou popsány v následujících sekcích.

Abstraktní adaptér

Abstraktní adaptér je implementovaný v modulu `adapters.abs_adapters` a definuje uniformní rozhraní adaptérů pro podporovaná V/V zařízení. Zahrnuje také veškerou funkcionalitu potřebnou pro zpracování požadavků jiných komponent systému i pro udržování aktuálních informací o stavu zařízení a usnadňuje tak implementaci konkrétního adaptéru. Komponenta se skládá z následujících částí:

- *FIFO fronta* – slouží k sekvenčnímu provádění požadavků pro V/V zařízení, díky čemuž nedochází ke konfliktům při vícenásobném přístupu. Požadavky jsou do fronty vkládány ve formě funkce. Pro tuto část byla využita standardní knihovna `queue`, která implementuje fronty určené pro vícero konzumentů a producentů a užitečná je především při potřebě bezpečné výměny informací mezi vlákny.

- *Obsluha požadavků* – je implementována ve formě samostatného vlákna, které sekvenčně vykonává požadavky umístěné do FIFO fronty. Získání dalšího požadavku z fronty je blokující operace. Vlákno je ukončeno na základě příznaku umístěného do fronty.

- *Generování cyklického požadavku* – je taktéž implementováno ve formě samostatného vlákna a jeho účel je opakované vkládání požadavku do FIFO fronty. Obvykle se jedná o požadavek na čtení stavů jednotlivých výstupů, čímž dochází k aktualizaci jejich stavů v systému. V případě vyskytnutí problémů se pak do fronty vkládá požadavek na testování připojení k zařízení. Generování požadavku se provádí v pevném intervalu, který je možné definovat individuálně pro jednotlivé V/V adaptéry. Vlákno je ukončeno na základě nastavení příznaku.

- *Obsluha databáze* – je využívána v případě potřeby perzistentního uložení získaných informací. Většinou se jedná o aktualizaci stavů jednotlivých výstupů zařízení. Pro tyto účely je využito jádro systému, které je popsáno v následující podkapitole 6.3.

V/V manažer tedy na základě požadavku vyšší vrstvy systému zavolá relevantní funkci příslušného adaptéru. Tato funkce vloží požadavek do FIFO fronty, přičemž se kontext vrací zpět přes V/V manažera do iniciující vrstvy. Jakmile je požadavek na čele fronty, je proveden obslužným vláknem. Až v tomto kroku je využita funkcionalita specifická pro jednotlivé V/V adaptéry, která je implementována v podtřídách. Pro přidání podpory V/V zařízení je konkrétně nutné následující:

1. V balíčku `perun.io.adapters` implementovat podtřídu abstraktního adaptéru (třída `AbsAdapter`).
2. Implementovat všechny abstraktní metody. Konkrétně se jedná o metodu pro čtení stavu výstupu (`_read_output_state`), změny stavu výstupu (`_write_output_state`) a testování dostupnosti zařízení (`_is_available`).
3. Definovat požadované třídní atributy. Konkretně se jedná o řetězec reprezentující typ zařízení (`TYPE`), počet výstupů daného zařízení (`OUTPUT_COUNT`) a volitelně interval generování cyklického požadavku (`CYCLIC_INTERVAL`).
4. V případě chyby či jiného problému v komunikaci s V/V zařízením vyvolávat výhradně výjimku typu `AdapterError`, která je zachycována a následně zpracovávána funkcionalitou abstraktního adaptéru.
5. V případě potřeby implementace chování pro ukončení spojení tak vykonat v rámci přetížené funkce určené pro úklid adaptéru (`_clean`).

Jak již bylo dříve řečeno, nově implementovaný V/V adaptér není nutné registrovat v dalších částech systému a výše uvedený seznam je konečný.

Testovací adaptér

Tento adaptér je implementován v modulu `sandbox_adapter` a v systému je dostupný pod názvem *SANDBOX*. Jedná se o velmi jednoduchý adaptér, který je určen pro testovací účely a při reálném nasazení systému nemá větší význam. Při jeho použití není k systému připojeno žádné fyzické zařízení a výstupy jsou simulovány jednoduchým seznamem o čtyřech prvcích. Testování systému, kde byl využit mimo jiné i tento adaptér, je popsáno v podkapitole 7.1.

ET-7042 adaptér

Adaptér pro V/V zařízení ET-7042 byl implementován v modulu `et7042_adapter` dle poznatků popsaných v podkapitole 3.1. Využita byla knihovna *Pymodbus3*, která implementuje protokol Modbus pro jazyk Python. Knihovna poskytuje funkcionality klienta i serveru pro všechny varianty protokolu. V případě tohoto adaptéru byla využita funkcionalita implementující Modbus/TCP klienta a funkce *Read Coils* a *Write Single Coil*.

SITOP PSU8600 adaptér

Při implementaci adaptéru pro V/V zařízení SITOP PSU8600 bylo původně uvažováno o komunikaci pomocí protokolu OPC. Za tímto účelem bylo provedeno několik experimentů s knihovnami *OpenOPC* a *python-opcua*. Bohužel se však ukázalo, že ani jedna z uvedených knihoven není kompatibilní se zařízením SITOP PSU8600. Z toho důvodu byla zvolena implementace pomocí protokolu Profinet. Využita byla knihovna *Sitop-VCPU*, která simuluje Profinet řídící systém, umožňuje navázat aplikační vztahy typu IOC-AR, S-AR i implicitní AR a následně zasílat požadavky pomocí komunikačního vztahu Record Data CR. Knihovna byla dodána zadavatelem a jedná se o nástupce knihovny implementované v rámci bakalářské práce Bc. Tomášem Mazurkem [11].

Adaptér byl implementován v modulu `psu8600_adapter` dle poznatků popsaných v podkapitole 3.2. Během inicializace adaptéru je navázán aplikační vztah typu IOC-AR, který

je udržován po celou dobu jeho běhu. Vstupem knihovny *Sitop-VCPU* je konfigurace, která mimo jiné obsahuje cestu k GSD[2] souboru a kombinaci použitých modulů fyzického zařízení. Pro získání aktuálních stavů výstupů i pro jejich změnu je použit acyklický požadavek čtení (angl. „Read Request“), respektive požadavek zápisu (angl. „Write Request“). Přistupováno je k datovým strukturám na indexu `0x1` každého výstupu. Obsah této struktury je popsán v tabulce 6.1. Manipulováno je s hodnotou položky `OutputEnabled`, ostatní položky jsou ponechány ve výchozím stavu.

Název	Dat. typ	Vých. hodnota	Účel
`UTarget`	`UInt16`	2400	Požadované napětí (1 = 10 mV)
`ILimit`	`UInt16`	1000	Proudové omezení (1 = 10 mA)
`IThreshold`	`UInt8`	90	Práh pro proudové omezení (1 = 1%)
`OutputEnabled`	`UInt8`	0	Stav výstupu (0/255)
`Reset`	`UInt8`	0	Resetování výstupu (0/255)
`OperatingMode`	`UInt8`	0	Operační mód (0 = běžný)
`StartupDelay`	`UInt16`	0	Zpožděné zapnutí (1 = 1 ms)

Tabulka 6.1: Obsah datové struktury na indexu `0x1` [zdroj vlastní]

Jak bylo uvedeno dříve, zařízení je v Profinetu adresováno pomocí slotu určující modul a následně subslotu určující výstup, případně modul jako celek. Adresování výstupů ve vyvíjeném systému je však jednoúrovňové dle fyzického či logického pořadí. V implementovaném adaptéru je proto tato informace transformována mezi oběma zmíněnými reprezentacemi.

Adaptér byl implementován pro využití s konkrétní hardwarovou konfigurací, která je ve vývojové laboratoři používána. Vzhledem ke způsobu implementace, jak tohoto adaptéru, tak nadřazeného abstraktního adaptéru, by však neměl být velký problém rozšířit či modifikovat implementaci o jiné konfigurace zařízení SITOP PSU8600.

NET-PwrCtrl adaptér

Tento adaptér přidává podporu pro zařízení NET-PwrCtrl Pro společnosti ANEL-Elektronik AG [1]. Zařízení disponuje 8 standardními zásuvkami o napětí 230 voltů. Ty jsou selektivně vypínatelné skrz vestavěný webový server či jednoduchý textový protokol přenášený pomocí protokolu transportní vrstvy UDP. Přestože podpora zařízení nebyla explicitně vyžadována, během vývoje projevilo vícero týmů ze strany zadavatele zájem o využití systému a podmíněno to bylo právě podporou uvedeného zařízení. To je navíc využíváno při certifikačních testech Profinet zařízení, což ještě rozšiřuje potencionální oblast využití systému.

Adaptér je implementován v modulu `net_pwr_adapter` a komunikace probíhá pomocí zmíněného textového protokolu. Využity jsou následující typy požadavků:

- `Sw_on` – požadavek na zapnutí. Řetězec je následován pozicí vybraného výstupu. Indexování je od 1.
- `Sw_off` – požadavek na vypnutí. Stejně jako v předchozím případě je řetězec následován pozicí vybraného výstupu.

[2] *GSD* (General Station Description) – technická charakteristika zařízení popsaná pomocí jazyku XML.

- `wer da?` – dotázání se na aktuální stav zařízení.

V případě všech uvedených typů požadavků je formát odpovědi shodný a obsahuje informační bloky oddělené znakem dvojtečky. Stavy jednotlivých výstupů jsou umístěny v blocích 7 až 14 v pořadí dle pozice výstupu.

6.3 Jádro systému

Jádro systému implementuje hlavní funkcionality systému a zároveň zastřešuje dříve popsané vrstvy interagující s databází a V/V zařízeními. Hlavní účel komponenty je abstrahování veškeré logiky systému. Poskytovány jsou mimo jiné všechny operace dle diagramu případů užití 5.4 uvedeného při návrhu aplikačního rozhraní na straně 28. Komponenta tím zároveň zabraňuje větším vazbám na typ vstupního rozhraní. V případě této práce bylo zvoleno REST aplikační rozhraní (viz podkapitola 6.4), vzhledem k implementaci této vrstvy je však umožněno jednoduché rozšíření systému o libovolné další vstupní rozhraní (například SOAP jako alternativa pro zmíněné REST API). Jádro systému je využíváno nejen v aplikačních rozhraních, ale také v jednotlivých V/V adaptérech při potřebě interakce s databází.

Popis implementace

Komponenta je implementována v balíčku `perun.controller` a skládá se z následujících modulů:

- `__init__` – inicializace jádra systému.
- `abs_controller` – obsahuje abstraktní třídu, která definuje základní metody pro moduly uvedené níže a implementuje jejich společnou funkcionalitu.
- `client_controller` – implementuje operace spojené s klienty, které lze následně využít v aplikačním rozhraní. Jedná se například o správu či autentizaci klienta.
- `device_controller` – podobně jako předchozí modul jsou zde implementovány operace, které lze využít v aplikačním rozhraní. Modul je zaměřen na V/V zařízení a poskytuje operace pro jejich správu, připojení či základní diagnostiku. Zatímco předešlé moduly komunikují výhradně s databázovou vrstvou, tento modul zapojuje taktéž komponentu V/V manažera.
- `resource_controller` – implementuje operace spojené se sdílenými prostředky. Právě v tomto modulu je tedy mimo jiné implementována funkcionalita pro alokování a uvolnění prostředku a případnou změnu jeho stavu. Podobně jako předchozí modul komunikuje s databázovou vrstvou i V/V manažerem.

Klíčovou částí této komponenty je třída `Controller` obsažená v inicializačním souboru balíčku. Třída je hlavním rozhraním jádra systému, které zapouzdřuje moduly popsané výše a s nimi i veškeré operace spojené s entitami systému. Zároveň obsluhuje vytvořenou *SQLAlchemy session*, která je vytvořena v rámci inicializace objektu. Pro efektivnější a bezpečnější použití je v třídě implementován správce kontextu. Ten umožňuje využít `with` konstrukci jazyka Python, kdy je automaticky prováděna operace `COMMIT`, případně `ROLLBACK` při vyskytnutí problému.

Při selhání jakékoliv operace jádra systému jsou vyvolávány vlastní výjimky. Ty jsou definovány v modulu `perun.misc.expcetions`. Každá výjimka obsahuje také odpovídající stavový kód protokolu HTTP, což usnadňuje případné zpracování těchto událostí na úrovní aplikačního rozhraní.

6.4 REST aplikační rozhraní

REST aplikační rozhraní zastupuje v serverové části systému roli jediného koncového bodu. Pro jazyk Python existují desítky knihoven určených k tvorbě webových aplikací, které lze použít pro jeho implementaci [8]. Mezi ty nejrozšířenější patří:

- *Django* – je dnes nejpopulárnější webový framework pro Python, který je navržen tak, aby pokryl nejběžnější potřeby webových aplikací. Django patří do skupiny takzvaných Full-Stack knihoven a kromě URL směrování a šablonování zahrnuje taktéž vlastní ORM, dynamické administrativní rozhraní či správu uživatelů. Knihovna cílí především na běžné informační systémy a další aplikace s databází v pozadí, které mohou být díky mnoha integrovaným nástrojům vyvinuty velmi rychle. Pro menší aplikace však může být příliš komplexní a omezující.

- *Flask* – jedná se o mikro framework — obsahuje tedy jen základní funkcionalitu v podobě URL směrování a šablonování. Jeho hlavním cílem je být flexibilní a umožnit uživateli vybrat si nástroje, které nejlépe vyhovují potřebám jeho projektu. Poskytuje zároveň mnoho možností customizace a rozšíření. Flask se nejlépe hodí pro menší až střední projekty. Zvlášť vhodný je pro webové API a služby.

- *Tornado* – je webový framework kombinovaný s knihovnou pro asynchronní síťovou komunikaci a je určený pro aplikace, které vyžadují dlouhodobě navázaná spojení s klienty. Jeho součástí je HTTP server založený na vlastní asynchronní knihovně. Přestože Tornado umožňuje použití s jiným WSGI[3] serverem, ztrácí tím svou hlavní přednost v asynchronním chování. Podobně jako u jiných knihoven je zahrnuto šablonování. Poskytována je také autentizace pomocí služeb třetích stran jako Google, Facebook nebo Twitter. Tornado je ideální v případě použití technologie *WebSocket*[4] nebo jiných dlouho trvajících spojení. Pro běžné aplikace je však lepší volbou spíše některá z dalších zmíněných knihoven.

- *Bottle* – je mikro framework a nemá jedinou závislost mimo standardní knihovny jazyka Python. Knihovna poskytuje velmi jednoduché URL směrování a šablonování. I přesto, že umožňuje rozšíření k uspokojení velkého množství komplexních požadavků, její hlavní využití je v případě malých aplikací a prototypování.

- *Pyramid* – je obecný framework pro webové aplikace, jehož hlavním cílem je jednoduchost a rychlost vývoje. Jeden z jeho hlavních cílů je umožnit vývoj s minimální znalostí dané knihovny s následným přirozeným růstem aplikace tak, jak se rozvíjí uživatelovo povědomí o další funkcionalitě. Pyramid se soustředí na základní potřeby webových aplikací — URL směrování, šablonování a bezpečnost. Navíc nabízí široké

[3] *WSGI* (Web Server Gateway Interface) – rozhraní pro směrování požadavků mezi webovým serverem a webovou aplikací v jazyce Python.

[4] *WebSockets* – komunikační protokol poskytující obousměrný kanál přes TCP připojení.

možnosti konfigurace a rozšíření. Oproti ostatním zmíněným knihovnám nehraje u Pyramid roli velikost uvažované aplikace a je stejně dobře použitelný pro menší i rozsáhlé projekty.

- *CherryPy* – je jedna z nejstarších webových knihoven pro jazyk Python a zaměřuje se pouze na URL směrování. Vše ostatní je ponecháno na vývojáři. Zabudovaný je vlastní webový server a nástroje pro autorizaci, cache, uživatelské session a podobně. Jedná se o velmi flexibilní framework, který je jednoduchý pro začátečníky. Vývojář však musí zvládnout úkony, které modernější knihovny obstarávají automaticky a použití CherryPy tak vyžaduje více práce.

Při výběru knihovny z výše uvedených bylo rozhodováno na základě vícero faktorů. V první řadě to byla dostatečná flexibilita umožňující výběr vhodných technologií. Další pak jednoduché a efektivní směrování URL, které vyhovuje potřebám REST aplikačních rozhraní. Výběr tak byl omezen na dvě knihovny — Flask a Pyramid. Pro implementaci byl nakonec zvolen webový framework Flask, jelikož má oproti knihovně Pyramid větší aktivní komunitu a z pohledu vyvíjeného systému také zajímavější rozšíření.

Flask-RESTPlus

Flask-RESTPlus je rozšíření webové knihovny Flask, které přidává podporu pro rychlé vytváření REST aplikačních rozhraní [3]. Poskytovány jsou promyšlené dekorátory a nástroje, díky kterým lze popsat dané aplikační rozhraní dle nejlepší praxe a s minimálním úsilím. Rozšíření zahrnuje následující funkcionalitu:

- *Optimalizované směrování* – URL a HTTP metody jsou zde oproti původního směrování, kde jsou vázány na jednotlivé funkce, definovány pro konkrétní prostředky reprezentované třídami. Tím je docíleno ucelenější struktury kódu, který tak více odpovídá aplikačnímu rozhraní než běžné webové aplikaci. Optimalizované směrování je taktéž důležité vzhledem k další funkcionalitě rozšíření.

- *Parsování požadavků* – položky jednotlivých požadavků lze zpracovat pomocí analyzátoru argumentů, který je inspirován standardní knihovnou `argparse`. Zdrojem analýzy mohou být hlavičky, cookies či argumenty v URL. V případě chyby je v těle odpovědi vrácena obdobná nápověda jako v případě zmíněné knihovny.

- *Validace a formátování dat* – v případě serializace ve formátu JSON poskytuje rozšíření jednoduchý způsob jak určit, která data jsou formátována do těla odpovědi a zároveň jaká data jsou očekávána v těle požadavku. Lze toho docílit definováním modelu, na základě kterého je posléze prováděna zmíněná validace nebo formátování. Model je založen na médiu JSON Schema[5]. K dispozici je celá řada datových typů, pro které lze definovat nejrůznější vlastnosti k co nejpřesnějšímu pokrytí požadovaného formátu. Umožněno je také vnořování modelů či jejich dědění a klonování. Model je zároveň nezávislý na typu vstupu, ať už se jedná o položku ORM, vlastní třídu či slovník.

- *Zpracování chyb* – v případě nevalidních dat, neexistujícího URL směrování či jiné chyby požadavku je automaticky zaslána odpověď s relevantním stavovým kódem

[5] *JSON Schema* – definuje typ média založený na JSON formátu pro popis struktury JSON dat.

a náležitě serializovanými upřesňujícími informacemi. Flask-RESTPlus navíc disponuje mechanismem, pomocí kterého lze různým výjimkám vyvolaných nižšími vrstvami aplikace přiřadit odpovídající reakci na úrovni aplikačního rozhraní. Díky tomu lze efektivně převádět vnitřní stavy aplikace na jejich serializovanou podobu, která je následně zaslána klientovi v podobě HTTP odpovědi.

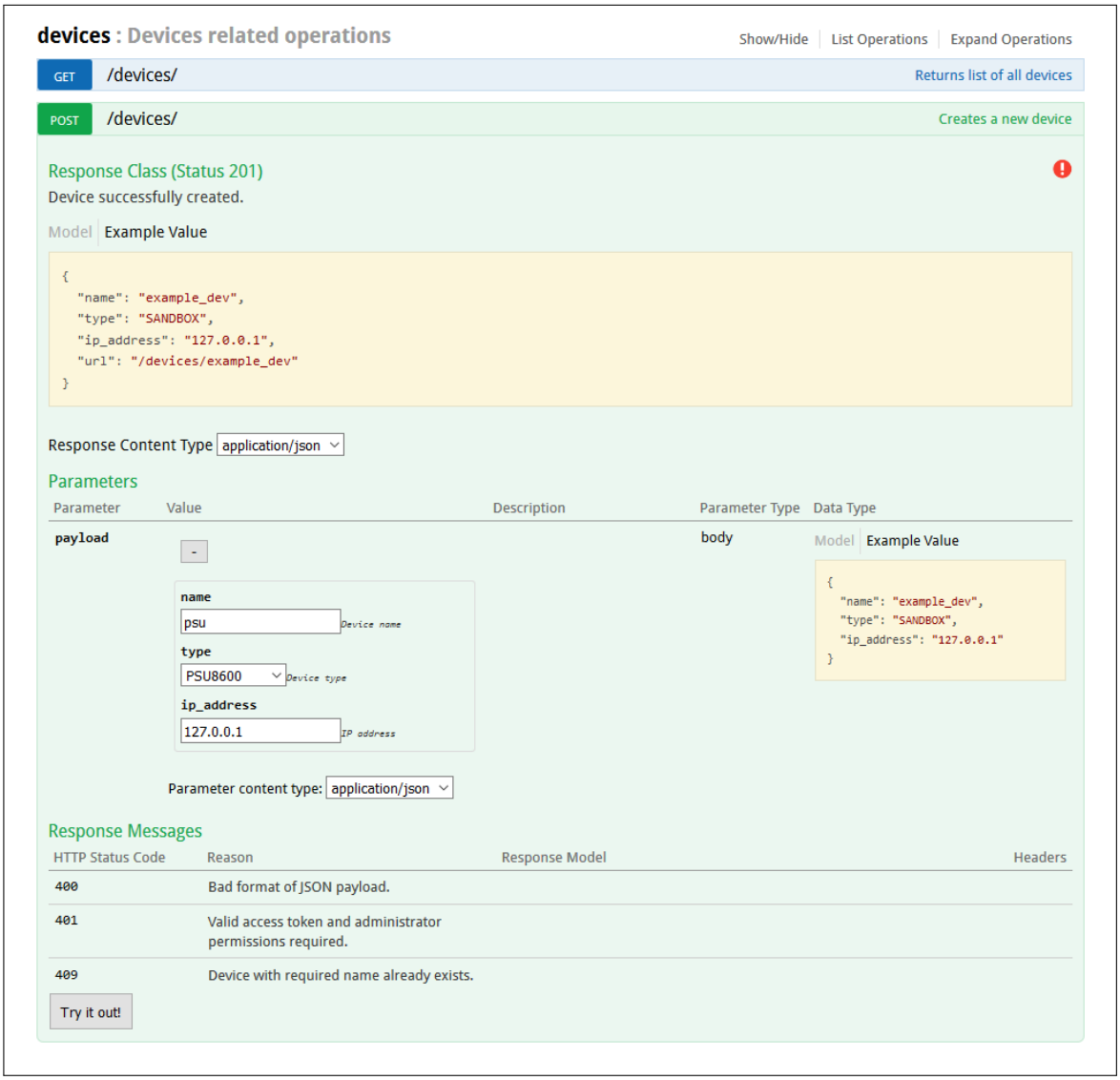
- *Generování profilu* – velmi zajímavou funkcionalitou je generování OpenAPI profilu. Jedná se o popis aplikačního rozhraní ve formátu JSON, které umožňuje lidem i strojům prozkoumat a porozumět všem možnostem služby bez nutnosti přístupu ke zdrojovému kódu, dodatečné dokumentaci či analýze síťového provozu. Profil je generován na základě kódu, což dává výše popsaným funkcionalitám nový rozměr. Součástí profilu je taktéž nadstavba v podobě webového uživatelského rozhraní Swagger UI. To vizualizuje REST aplikační rozhraní a umožňuje interakci s poskytovanými zdroji pomocí zabudovaného HTTP klienta cURL. Výsledný profil a tedy i uživatelské rozhraní zahrnuje mimo jiného jednotlivé zdroje, jejich URL, podporované metody, očekávaný obsah požadavku či formát a možné stavové kódy odpovědi. OpenAPI profil i Swagger UI má taktéž podporu pro různé autentizační techniky.

Popis implementace

REST aplikační rozhraní systému je založeno na webové knihovně Flask v kombinaci s rozšířením Flask-RESTPlus a je implementováno v balíčku `perun.api` dle návrhu v podkapitole 5.4. Řešení se skládá z následujících modulů:

- `__init__` – inicializace Flask aplikace.
- `auth` – obsahuje mechanismy potřebné pro autentizaci v systému. Tato funkcionalita je blíže popsána v podkapitole 6.5.
- `fields` – obsahuje třídy reprezentující datové typy, které jsou využitelné při vytváření modelů požadavků či odpovědí a rozšiřuje tím množinu datových typů poskytovaných knihovnou Flask-RESTPlus. Konkrétně je implementována třída pro validaci IP adres verze 4 a třída rozšiřující formátování pro položky typu URL.
- `rest.__init__` – inicializace REST aplikačního rozhraní. Zahrnuto je mimo jiné přiřazení směrovacích pravidel pro dílčí rozhraní z balíčku `rest` či registrování obslužné funkce pro vlastní výjimky vyvolávané nižšími vrstvami systému.
- `rest.common` – obsahuje obecné zdroje určené například pro účely autentizace.
- `rest.clients` – obsahuje rozhraní pro zdroje reprezentující klienty, se kterými umožňuje základní CRUD[6] operace.
- `rest.devices` – rozhraní pro zdroje reprezentující V/V zařízení obsahuje kromě CRUD také operace pro navázání či přerušení spojení mezí systémem a zařízením či získání základní diagnostiky.
- `rest.resources` – rozhraní pro zdroje reprezentující sdílené prostředky. Poskytováno je opět CRUD a dále operace pro alokaci, uvolnění či změnu stavu konkrétního prostředku.

[6] *CRUD* (Create, Read, Update, Delete) – čtyři základní operace v podobě vytvoření, čtení, editace a odstranění prostředku.



Obrázek 6.3: Ukázka Swagger UI rozhraní [zdroj vlastní]

- `rest.dev` – obsahuje informace, které jsou užitečné v případě vývoje či testování (například získání seznamu všech aktivních vláken). Operace jsou dostupné pouze při běhu v ladícím režimu.

V případě všech zdrojů aplikačního rozhraní jsou data ve formátu JSON a specifikovány jsou modely pro validaci vstupů i formátování výstupů. Explicitně jsou uvedeny taktéž všechny možné stavové kódy. Tyto informace jsou využity nejen při zpracování požadavků, ale také pro generování OpenAPI profilu. Ten je dostupný na URI */swagger.json*. Interaktivní rozhraní Swagger UI je pak dostupné z kořenové cesty serveru. Ukázku rozhraní lze vidět na obrázku 6.3. Systém nabízí také alternativní strojově čitelnou reprezentaci systému. Konkrétně se jedná o kolekci pro vývojové prostředí Postman[7]. Kolekce je dostupná na URI */postman*.

[7] *Postman* – prostředí pro vývoj a testování REST aplikačních rozhraní, které zahrnuje nástroje pro návrh, automatizované testování, dokumentování či monitorování.

6.5 Autentizace

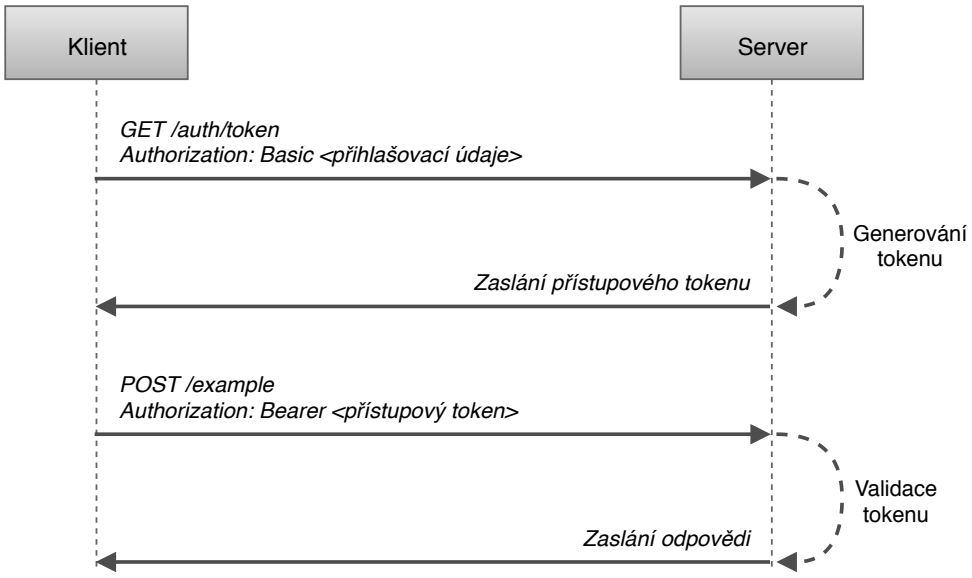
Systém podporuje autentizaci pomocí přístupového tokenu. Ten je vygenerován na základě zaslání validních přihlašovacích údajů metodou *Basic Authentication*. Obě metody byly dříve popsány v podkapitole 4.1. Proces autentizace je následující:

1. Klient zašle požadavek na libovolný chráněný prostředek aplikačního rozhraní bez předchozí autentizace. Jelikož klient není autentizován, server odpovídá zprávou se stavovým kódem 401 vyjadřující neautorizovaný přístup a s `WWW-Authenticate` hlavičkou s upřesňujícími informacemi.

2. Klient zašle `GET` požadavek na zdroj */auth/token*. Požadavek je doplněn o hlavičku `Authorization` obsahující přihlašovací údaje. Jestliže jsou zaslané údaje validní, je v těle odpovědi zaslána struktura ve formátu JSON obsahující mimo jiné vygenerovaný token.

3. Klient zašle opět požadavek na chráněný prostředek, který byl dříve zamítnut. Doplněný je o hlavičku `Authorization` obsahující obdržený přístupový token. V případě, že je přístupový token validní, je požadavek zpracován a je vrácena relevantní odpověď. V opačném případě se opakuje stejná reakce jako v kroku 1.

Proces autentizace je znázorněný taktéž na obrázku 6.4.



Obrázek 6.4: Proces autentizace v systému [zdroj vlastní]

JSON Web Token

Pro generování a validaci přístupového tokenu byl využit otevřený standard *JWT* (JSON Web Token), který byl definován v RFC 7519 a je určen pro zabezpečení JSON objektu pomocí digitálního podpisu [9, 2]. JWT může být podepsán použitím tajemství (hašovací algoritmus HMAC) nebo použitím veřejného/soukromého klíče (asymetrický algoritmus RSA). Nejběžněji je standard využíván právě při autentizaci. Struktura JWT se skládá ze tří částí oddělených tečkou:

- *Hlavička* – se typicky skládá ze dvou částí. První je typ tokenu, tedy JWT. Druhou je pak použitý algoritmus. Hlavička JWT tokenu může vypadat následovně:

  ```
  {
    "alg": "HS256",
    "typ": "JWT"
  }
  ```

  JSON objekt je poté zakódován pomocí `Base64Url` jako první část JWT tokenu.

- *Data* – obsahují samotné atributy (typicky o uživateli) a další metadata. JWT definuje několik rezervovaných atributů, které lze v tokenu použít. Jedná se například o atribut `iss` obsahující vydavatele JWT nebo `exp` s časem expirace daného tokenu. Data JWT tokenu mohou být následující:

  ```
  {
    "iss": "api.example.com",
    "name": "John Doe",
    "admin": true
  }
  ```

  Stejně jako v případě hlavičky je JSON objekt zakódován pomocí `Base64Url`. Výstup poté představuje druhou část JWT tokenu.

- *Podpis* – slouží k zajištěni integrity dat a ověření identity odesílatele. Vstupem algoritmu je zakódovaná hlavička a data oddělené tečkou. V případě použití algoritmu HMAC SHA256 může být podpis vytvořen následovně:

  ```
  HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret)
  ```

  Výstup hašovacího algoritmu představuje třetí část JWT tokenu.

Popis implementace

Jak bylo zmíněno v podkapitole 6.4, autentizace je implementována v modulu `api.auth`. Pro generování a validaci JSON Web Tokenu byla použita knihovna *PyJWT*, která tento standard implementuje v jazyce Python. Podpis tokenu je vytvořen hašovacím algoritmem HMAC SHA256. Jako tajemství je použit 32 bajtů dlouhý řetězec, který je při startu serveru vygenerován pomocí standardní knihovny *secrets*. Ta je určena pro generování kryptograficky silných náhodných číslech, která jsou vhodná pro použití jako hesla, bezpečnostní tokeny, tajemství a podobně. Datová část JWT obsahuje následující atributy:

- `id` – identifikace klienta, pro kterého byl daný token vygenerován.
- `exp` – čas expirace daného tokenu.

Při obdržení požadavku na chráněný zdroj je v první řadě zkontrolováno, zda požadavek obsahuje hlavičku `Authorization` typu `Bearer`. Následně je provedena validace tokenu, což zahrnuje ověření jeho podpisu, kontrolu času expirace a získání instance klienta na základě atributu `id`. Na závěr je pro administrátorské operace ověřeno klientovo oprávnění.

V případě, že neselže provedení žádného ze zmíněných bodů, je klient úspěšně autentizován. Popsaná funkcionalita je implementována pomocí dekorátoru, který lze aplikovat na libovolný zdroj REST aplikačního rozhraní. Pro dekorovanou funkci je taktéž zpřístupněna daná instance klienta.

Kromě generování a validace umožňuje systém taktéž zrušit platnost všech tokenů určitého klienta. Jelikož vygenerované tokeny nejsou na serveru interně uchovávány, je zneplatnění docíleno změnou atributu `id`. Z důvodu možných změn však není vhodné jako hodnotu této položky použít atribut *ID*, primární klíč databázové entity *Klient*. Proto byl oproti návrhu (viz podkapitola 5.2) přidán do této entity atribut *Veřejné ID*, který slouží právě pro identifikaci klienta v přístupových tokenech. Veřejné ID je reprezentováno pomocí *UUID*[8]. Pro generování byla použita stejnojmenná standardní knihovna *uuid*.

Z důvodů uvedených v podkapitole 4.1 není bezpečné používat implementovaný způsob autentizace v kombinaci s nešifrovaným přenosem. Hrozí tak odcizení nejen přístupového tokenu, ale také samotných přihlašovacích údajů během jeho generování. Proto je v systému podporován šifrovaný přenos pomocí protokolu TLS. Tato funkcionalita je volitelná, nicméně její použití je silně doporučováno. Šifrování zajišťuje knihovna *OpenSSL*. Podporovány jsou klíče v PEM[9] formátu.

6.6 Klient

Klientská část této práce slouží k demonstraci možné komunikace se systémem. Stejně jako serverová část byla implementována v jazyku Python. Použita byla knihovna *requests*, která umožňuje zasílání HTTP/1.1 požadavků a následné zpracování odpovědí. Podporováno je mimo jiné přidávání obsahu jako hlavičky a formulářová data, udržování spojení, SSL verifikace či základní autentizační techniky. Hlavní komponenty klientské aplikace jsou následující:

- *Autentizační modul* – implementuje rozhraní pro autentizaci v systému. To zahrnuje zasílání přihlašovacích údajů metodou základní autentizace za účelem vygenerování nového přístupového tokenu, zneplatnění existujících tokenů či změnu uživatelského hesla. Aktivní přístupový token potřebný pro další požadavky je dále dostupný z tohoto modulu.

- *Klientské API* – nabízí veškeré operace podporované serverovou částí systému ve formě metod jazyka Python a skrývá před uživatelem HTTP komunikaci běžící v pozadí aplikace. Tuto komponentu lze využít pro integraci systému do automatizovaných nástrojů vyvinutých ve stejném jazyku.

- *Konzolové rozhraní* – jednoduché konzolové rozhraní, které poskytuje operace povolené pro roli běžného uživatele. Kromě zobrazování obsahu tedy dále rezervaci, uvolnění a změnu stavu vybraného prostředku. Účelem této komponenty je integrace klientské části do systému Jenkins[10], který je v laboratoři využíván jako grafické uživatelské rozhraní pro ovládání různých řešení automatizace.

[8] *UUID* (Universally Unique IDentifier) – pseudonáhodný unikátní řetězec definovaný v RFC 4122.

[9] *PEM* (Privacy-enhanced Electronic Mail) – formát pro přenos a uchovávání kryptografických klíčů, certifikátů a dalších dat vycházející ze standardu X.509.

[10] *Jenkins* – serverová služba, která poskytuje prostředky pro automatizaci procesů ve vývoji softwaru.

Kapitola 7

Zajištění kvality

Vzhledem k oblasti využití, kde každý výpadek může způsobit nemalé problémy, a povaze systému, kdy je předpokládán nepřetržitý chod, je velký důraz kladen na kvalitu výsledného řešení. Jejímu zajištění je věnována právě tato kapitola. Na úvod je v podkapitole 7.1 shrnuto testování systému. V podkapitole 7.2 následuje popis dalších využitých prostředků. Poslední podkapitola 7.3 se poté věnuje zkušebnímu nasazení systému v cílovém prostředí.

7.1 Testování

Při práci na řešení jsem se snažil o vývoj řízený testy neboli TDD (angl. „Test Driven Development“). Jedná se o přístup k vývoji softwaru, kdy vytvoření testovacích případů pro určitou funkcionalitu předbíhá implementaci samotného chování. Přestože ne vždy se mi podařilo přesně definovat požadované chování a testy musely být zpětně modifikovány, celkově měl tento přístup pozitivní efekt na výslednou kvalitu. Obzvlášť přínosný byl v případech většího zásahu do implementace v pozdější fázi vývoje, kdy bylo možné okamžitě ověřit základní funkcionality systému a alespoň částečně tak zabránit vložení chyby.

Samotné ověřování probíhalo na úrovni REST aplikačního rozhraní dle strategie testování černé skříňky (angl. „Black-Box Testing“). Při použití této metody je k testované aplikaci přistupováno bez zájmu o vnitřní implementaci a strukturu programu. Všechna testovací data jsou odvozena na základě specifikace vstupů a výstupů, nikoliv ze znalosti vnitřní struktury. Zaměřením se pak jednalo o takzvané systémové testování (angl. „System Testing“). To se orientuje na dokázání, že výsledný program pokrývá specifikaci požadavků. Testuje se správnost výstupů aplikace, ošetření nestandardních situací a zejména pokrytí všech požadovaných vlastností.

Použité technologie

Pro implementaci testovacích případů byl zvolen testovací framework *pytest* [18]. Velkou předností této knihovny je flexibilita a škálovatelnost, kdy je možné pro testovací případy vytvořit hierarchickou strukturu závislostí. Tyto závislosti lze navíc mezi jednotlivými testy sdílet. Na základě definovaného rozsahu působnosti je pak určeno jejich opětovné vykonání. Framework disponuje taktéž mechanismem pro automatické vyhledávání testovacích případů v adresářové struktuře, čímž je usnadněno jejich definování i následné spuštění. Samozřejmostí je podrobný report selhání, který urychluje nalezení chyby.

Při testování byla využita funkcionalita knihovny Flask, která umožňuje přepnutí obsluhované aplikace do testovacího módu. K dispozici jsou poté operace simulující běžné

HTTP metody, přičemž veškeré testy probíhají lokálně. To usnadnilo inicializaci testovacích případů. Jednoznačnou výhodou tohoto řešení však bylo zahrnutí zásobníku volání (angl. „Stack Trace“) při selhání na straně serveru do výsledného reportu. Vzhledem k tomu, že při běžné komunikaci pomocí HTTP klienta obdrží testovací framework pouze stavový kód 500 oznamující interní chybu serveru, se jedná o výrazné zjednodušení při hledání chyby.

Pro perzistentní uložení dat byla pro testovací účely zvolena databáze SQLite. Hlavním důvodem byla její nenáročnost, kdy je reprezentována jediným binárním souborem. Ten lze navíc vygenerovat automaticky pomocí knihovny SQLAlchemy.

Testování probíhalo ve většině případech na operačních systémech Microsoft Windows 7 a Microsoft Windows 10, na kterých bylo řešení taktéž vyvíjeno.

Testovací případy

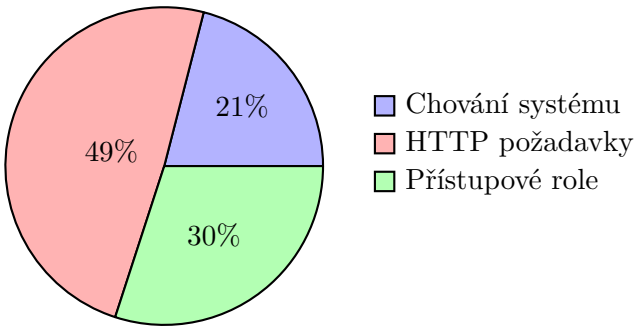
Využitím technologií popsaných výše bylo implementováno celkem 245 testů. Ty lze rozdělit do následujících kategorií:

a) *Testování chování systému* – ověřuje pokrytí všech požadavků na systém ať už z pohledu základních operací s entitami systému či operací pro sdílení prostředků. Všechny testy jsou prováděny s administrátorským oprávněním. Pro operace vyžadující V/V zařízení je použit testovací adaptér popsaný v podkapitole 6.2 a to z důvodu omezeného přístupu k fyzickým zařízením.

b) *Testování HTTP požadavků* – testuje ošetření neplatných HTTP požadavků. To zahrnuje zasílání požadavků s nevalidními JSON daty či požadavků na nepodporované HTTP metody.

c) *Testování přístupových rolí* – ověřuje přístupová oprávnění jednotlivých rolí k nabízeným operacím.

Poměr uvedených kategorií v implementovaných testech je znázorněn na obrázku 7.1. Pro každý jednotlivý test byl vždy obnoven výchozí stav systému včetně databáze, čímž bylo zabráněno jejich vzájemné závislosti. Testování bylo prováděno v HTTPS režimu s certifikátem podepsaným sám sebou (angl. „Self-signed certificate“).



Obrázek 7.1: Poměr testů dle základních kategorií [zdroj vlastní]

Testování V/V adaptérů

Vzhledem k omezenému přístupu k zařízením NET-PowerControl Pro, SITOP PSU8600 a ET-7042 nebyly pro jejich adaptéry implementovány automatizované testy popsané výše.

Řešení bylo testováno manuálně v prostředí vývojové laboratoře. Vzhledem k povaze implementace jednotlivých adaptérů a faktu, že základní operace byly ověřeny pomocí testovacího adaptéru, se jednalo o testování čistě komunikační vrstvy. Pro tyto účely bylo využito interaktivní rozhraní systému Swagger UI, nástroj *Wireshark* pro sledování a analýzu síťové komunikace, webová rozhraní jednotlivých zařízení a v případě potřeby fyzická manipulace.

7.2 Další prostředky zajištění kvality

Cílovou skupinou systému jsou vývojové týmy. Předpokládá se tedy nejen uživatelské využití, ale taktéž možný zásah do implementace za účelem rozšíření. Z toho důvodu jsem se při vývoji soustředil také na kvalitu samotného kódu. Implementace se snaží dodržovat pravidla určené společností Google v dokumentu *Google Python Style Guide* [6], který definuje základní doporučení při vývoji v jazyce Python. Ta mimo jiné zahrnují statickou analýzu kódu a dokumentaci, kterým jsou věnovány následující sekce.

Statická analýza kódu

Statická analýza odhaluje na základě zdrojových kódů nejrůznější nedostatky [16]. Rozdělit je lze do následujících kategorií:

- *Detekce chyb* – odhaluje základní chyby jako například nedefinovanou proměnnou, neimportovaný modul či zda je deklarované rozhraní skutečně implementováno.
- *Nedostatky v návrhu* – zaměřuje se na nedostatky řešení například v podobně duplicitního kódu, příliš komplexních tříd či nevhodně aplikované dědičnosti.
- *Dodržování konvencí* – klade důraz na dodržování stylu kódu dle obecně přijatých konvencí daného jazyka.

Analýza kódu je důležitá především při použití interpretovaných jazyků, mezi které patří i jazyk Python. Na rozdíl od kompilovaných jazyků zde totiž nejsou základní chyby odhaleny během překladu.

Pro analýzu kódu implementovaného systému jsem zvolil tyto nástroje:

- *Pylint* – analýza kódu, která detekuje širokou škálu chyb a nedostatků v návrhu. Ověřován je taktéž styl kódu, kdy je volně vycházeno z konvencí definovaných v dokumentu PEP[1] 8. Nástroj vyžaduje vykonání kódů před jeho analýzou.
- *Flake8* – zastřešuje vícero nástrojů pro analýzu kódu jazyka Python. Jedná se o novější nástroj než Pylint a na rozdíl od něj nevyžaduje před analýzou vykonání kódu. Při kontrole dodržování konvencí je pak vycházeno striktně z dokumentu PEP 8.

Přestože jsou oba nástroje ve své podstatě velmi podobné a svou funkcionalitou se mnohdy překrývají, právě jejich odlišnosti jsou přidanou hodnotou jejich souběžného použití.

Pro dodatečnou analýzu byl využit nástroj *mccabe*, který je podporován výše uvedenými analyzátory ve formě rozšíření. Ten slouží k měření cyklomatické složitosti, která vyjadřuje počet možných cest při vykonávání analyzované funkce. Díky této metrice byly značně redukovány komplexní funkce implementace a zvýšena tak přehlednost řešení.

[1] *PEP* (Python Enhancement Proposal) – typ dokumentu, pomocí kterého jsou publikovány informace Python komunitě, popisovány nové funkce jazyka, jeho další procesy a doporučení.

Dokumentace kódu

Za účelem zdokumentování kódu systému obsahují všechny moduly, třídy, metody i funkce takzvaný dokumentační řetězec. Jedná se o první výraz implementace daného objektu, který je poté programově dostupný skrz atribut `__doc__`. Dokumentační řetězec je v případě implementovaného řešení v Google formátu a kromě základního popisu objektu obsahuje informace jako význam vstupních argumentů, význam výstupu či možné vyvolané výjimky. Při implementaci bylo navíc využito možnosti anotace datových typů definované v dokumentu PEP 484. Vzhledem k tomu, že Python je dynamicky typovaný jazyk je však tato informace využita právě jen pro účely dokumentace či jako nápověda pro integrovaná vývojová prostředí.

Pro generování dokumentace z kódu byla využita knihovna *Sphinx*. Ta umoňuje automatické vytvoření dokumentace v nejrůznějších formátech včetně HTML. Tato knihovna je mimo jiné využívána i pro generování oficiální dokumentace jazyka Python.

7.3 Zkušební nasazení

Vyvinuté řešení bylo zkušebně nasazeno v prostředí vývojové a testovací laboratoře společnosti Siemens. Klientské rozhraní bylo instalováno na zařízeních s operačním systémem Microsoft Windows 7. Serverová část pak byla nasazena na zařízení Raspberry Pi 3 s operačním systémem Raspbian. Důvody volby tohoto zařízení jsou blíže popsány v podkapitole 8.1 věnující se dalším rozšířením systému. Jako databáze byla při nasazení zvolena PostgreSQL, která běžela lokálně na stejném zařízení. HTTPS funkcionalita byla povolená, použit byl certifikát podepsaný sám sebou. Zvolený vzorek dat byl vybrán na základě reálné situace v laboratoři a zahrnoval následující:

- *11 klientů*, přičemž 5 z nich byli členové vývojového týmu a zbývajících 6 klientů reprezentovalo automatizované testovací nástroje.
- *3 V/V zařízení*, z nichž 2 zařízení byla typu ET-7042 a 1 zařízení typu SITOP PSU8600,
- *25 sdílených prostředků*, které zahrnovali různé mapování výstupů všech V/V zařízení.

Při demonstraci základní funkcionality nebyla objevena žádná chyba způsobující pád serveru či neoprávněného odmítnutí požadavku. Objevovaly se však problémy se stabilitou spojení se zařízením SITOP PSU8600, které byly způsobeny chybou v knihovně *Sitop-VCPU* při běhu na operačních systémech Linux. Vzhledem k tomu, že se však jedná o interní knihovnu společnosti Siemens, bude chyba v nejbližší době odstraněna. Při testování stejné konfigurace na operačním systému Microsoft Windows 7 se již problém nevyskytl.

Kapitola 8

Závěr

V rámci této diplomové práce jsem se seznámil s problematikou sdílení prostředků ve vývojové laboratoři společnosti Siemens a dále se zařízeními, které jsou v laboratoři používány pro účely automatizace. To zahrnuje taktéž komunikační protokoly Modbus a Profinet, které jsou těmito zařízeními podporovány. Dále jsem se obeznámil s možnostmi webových aplikačních rozhraní, především pak s REST architekturou.

Na základě těchto znalostí a požadavků externího zadavatele jsem navrhl a implementoval systém, jehož účelem je právě rezervování a konfigurace sdílených prostředků. Pro konfiguraci lze využít V/V zařízení ET-7042, SITOP PSU8600 a NET-PowerControl Pro. Vzhledem ke způsobu implementace však není problém přidat podporu dalších zařízení. Systém je dostupný přes zmíněné REST webové rozhraní a umožňuje tak integraci existujících testovacích nástrojů nehledě na použité technologie při jejich implementaci. Zajištěna je také persistence dat pomocí databáze PostgreSQL, MySQL nebo SQLite a podpora zabezpečené HTTPS komunikace.

Výsledné řešení jsem ověřil pomocí automatizovaných i manuálních testů a zkušebně nasadil v cílovém prostředí. Vyvinutý systém bude dále používán a rozšiřován vývojovým týmem společnosti Siemens s. r. o.

8.1 Možnosti rozšíření

Možnosti rozšíření systému jsou poměrně široké. Zajímavou funkcionalitou by mohl být jeden z následujících bodů:

- *Integrace prostředí* – externím zadavatelem byl dodán senzor pro zařízení Raspberry Pi 3, který umožňuje detekovat, zda je testovací laboratoř aktuálně v provozu. Díky tomu by bylo možné v systému lépe reagovat při selhání připojení k V/V zařízení. Tento bod byl také důvodem zkušebního nasazení na zařízení Raspberry Pi 3.

- *Podpora plánování* – klient bude mít možnost rezervovat sdílený prostředek okamžitě nebo v budoucnosti. Zároveň bude nucen uvést, po jakou dobu si prostředek rezervuje. Rezervace nebude umožněna v případě, že se kříží s již existujícím požadavkem. Systém bude zároveň disponovat automatickým uvolněním prostředků po vypršení doby rezervace. Mechanismus může přinést další zefektivněni při sdílení prostředků. Před samotnou implementací rozšíření však bude nutné provést detailnější případovou studii.

- *Vizualizace aktuálního stavu* – jednoduchá aplikace, která se bude v intervalech dotazovat systému na aktuální stav sdílených prostředků. Získaná data budou poté přehledně vizualizovány. Aplikace může být nasazena na jedné z obrazovek, které jsou v laboratoři k dispozici.

Rozšíření budou vycházet výhradně z dalších potřeb externího zadavatele. Kromě položek uvedených výše tedy lze předpokládat integraci klienta do nejrůznějších automatizovaných nástrojů, implementaci dalších klientských aplikací využitím jiných technologií, rozšíření systému o podporu dalších V/V zařízení či přidání alternativního webového rozhraní.

Literatura

[1] *ANEL-Elektronik NET-PwrCtrl* [online]. ANEL-Elektronik AG, 2018 [cit. 2018-04-30]. Dostupné z: <https://anel-elektronik.de/index.htm?src=SITE/produkte/pro/pro.htm>.

[2] *JSON Web Tokens (JWT) in Auth0* [online]. Auth0 Inc., 2018 [cit. 2018-04-10]. Dostupné z: <https://auth0.com/docs/jwt>.

[3] *Flask-RESTPlus* [online]. Axel Haustant, 2014 [cit. 2018-04-21]. Dostupné z: <http://flask-restplus.readthedocs.io/>.

[4] Fredrich, T. : RESTful Service Best Practices: Recommendations for Creating Web Services. 2013 [cit. 2018-03-18]. Dostupné z: <https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf>

[5] Gamma, E.; Helm, R.; Johnson, R.; aj. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994. ISBN 9780321700698.

[6] *Google Python Style Guide* [online]. Google Inc., 2018 [cit. 2018-05-12]. Dostupné z: <https://google.github.io/styleguide/pyguide.html>.

[7] Gourley, D.; Totty, B. *HTTP: The Definitive Guide*. O'Reilly Media, 2002. ISBN 978-1-56592-509-0.

[8] de la Guardia, C. *Python Web Frameworks* [online]. O'Reilly Media, 2016 [cit. 2018-05-05]. Dostupné z: <http://www.oreilly.com/web-platform/free/python-web-frameworks.csp>.

[9] Jones, M.; Bradley, J.; Sakimura, N. : JSON Web Token (JWT). RFC 7519, 2015 [cit. 2018-05-07]. Dostupné z: <http://www.rfc-editor.org/rfc/rfc7519.txt>

[10] Lin, L. : *ET-7000/PET-7000 DIO Series User Manual*. ICP DAS CO., LTD., 2009 [cit. 2017-12-30]. Dostupné z: <https://www.icpdas-usa.com/documents/pet7k_dio_user_manual_v1.01beta2.pdf>

[11] Mazurek, T. *Aplikace pro simulaci řídícího systému PROFINET* Brno: Vysoké učení technické v Brně. Fakulta informačních technologií, 2017.

[12] Modbus Organization, Inc. : Modbus application protocol - specification. 2012 [cit. 2017-12-31]. Dostupné z: <http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf>

[13] Modbus Organization, Inc. : Modbus messaging on TCP/IP - implementation guide. 2006 [cit. 2017-12-31]. Dostupné z: <http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf>

[14] Myers, J.; Copeland, R. *Essential SQLAlchemy: Mapping Python to Databases.* O'Reilly Media, 2015. ISBN 978-1-491-91646-9.

[15] Pigan, R.; Metter, M. *Automating with PROFINET.* Publicis Corporate Publishing, 2006. ISBN 3-89578-256-4.

[16] Pillai, A. B. *Software Architecture with Python.* Packt Publishing, 2017. ISBN 9781786467225.

[17] Popp, M. *Industrial Communication with PROFINET.* PROFIBUS&PROFINET International, interní materiály, 2014 [cit. 2017-12-31]. Dostupné z: <https://www.profibus.com/download/book-industrial-communication-with-profinet/>.

[18] *pytest: helps you write better programs* [online]. pytest-dev team, 2018 [cit. 2018-05-08]. Dostupné z: <https://docs.pytest.org/>.

[19] *Python 3.6.6 Documentation* [online]. Python Software Foundation, 2018 [cit. 2018-04-28]. Dostupné z: <https://docs.python.org/>.

[20] Richardson, L.; Amundsen, M. *RESTful Web APIs: Services for a Changing World.* O'Reilly Media, 2013. ISBN 978-1-449-35806-8.

[21] *SITOP PSU8600 - the unique power supply system with PROFINET* [online]. Siemens AG, 2018 [cit. 2018-01-14]. Dostupné z: <http://w3.siemens.com/mcms/power-supply-sitop/en/psu8600/>.

[22] Slováček, J. *Nástroj pro automatické testování produktů SITOP PSU8600 a SITOP UPS1600* Brno: Vysoké učení technické v Brně. Fakulta informačních technologií, 2016.

[23] *SQLAlchemy - The Database Toolkit for Python* [online]. SQLAlchemy authors and contributors, 2018 [cit. 2018-04-21]. Dostupné z: <http://www.sqlalchemy.org/>.

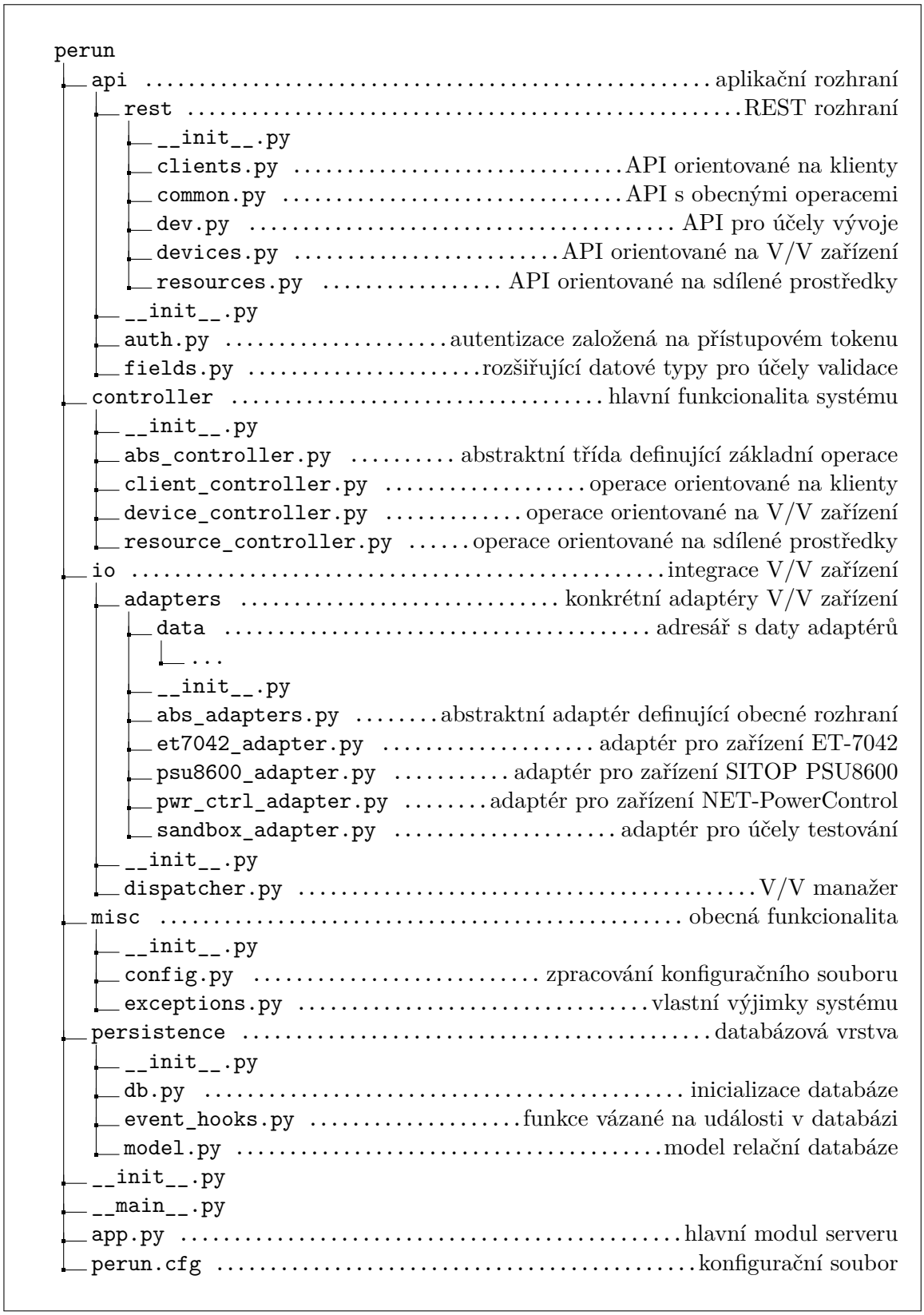
Příloha A

Kompletní struktura systému

V této části přílohy jsou uvedeny kompletní struktury implementovaného řešení. Nejprve je to struktura klientské aplikace (obrázek A.1). Následována je strukturou serverové části (obrázek A.2).

```
perun_client
├── __init__.py
├── __main__.py
├── api.py ..........................................klientské API
├── auth.py .........................................autentizace
├── cli.py ..........................................konzolové rozhraní
└── exceptions.py ..................................vlastní výjimky aplikace
```

Obrázek A.1: Kompletní struktura klientské části systému [zdroj vlastní]



Obrázek A.2: Kompletní struktura serverové části systému [zdroj vlastní]

Příloha B

Použité knihovny

V této části přílohy jsou uvedené všechny použité knihovny třetích stran včetně jejich licence a verze. Rozděleny jsou do několika části — knihovny nutné pro běh serveru (tabulka B.1), knihovny nutné pro běh klienta (tabulka B.2) a knihovny, které byly využity během vývoje za účelem zvýšení kvality výsledného řešení (tabulka B.3).

Název	**Verze**	**Licence**	**Účel**
SQLAlchemy	1.2.7	MIT	Interakce s databází
Psycopg2	2.7.4	LGPL	Adaptér pro PostgreSQL
mysql-connector-python	8.0.11	GPL	Adaptér pro MySQL
Flask	1.0.2	BSD	Webový framework
Flask-RESTPlus	0.10.1	MIT	REST aplikačí rozhraní
PyJWT	1.6.1	MIT	Operace s přístupovým tokenem
Passlib	1.7.1	BSD	Hashování uživatelských hesel
Pymodbus3	1.0.0	BSD	Integrace zařízení ET-7042
Sitop-VCPU	1.0.0	-	Integrace zařízení SITOP PSU8600

Tabulka B.1: Knihovny nutné k běhu serverové části systému [zdroj vlastní]

Název	**Verze**	**Licence**	**Účel**
Requests	2.18.1	Apache 2.0	HTTP klient

Tabulka B.2: Knihovny nutné k běhu klientské části systému [zdroj vlastní]

Název	**Verze**	**Licence**	**Účel**
pytest	3.5.1	MIT	Testovací framework
Pylint	1.8.3	GPL	Analýza kódu
Flake8	3.5.0	MIT	Analýza kódu
mccabe	0.6.1	MIT	Měření cyklomatické složitosti
Sphinx	1.7.2	BSD	Generování dokumentace z kódu
sphinx-rtd-theme	0.2.4	MIT	Šablona pro generovanou dokumentaci

Tabulka B.3: Knihovny využívané při vývoji [zdroj vlastní]

Příloha C

Obsah přiloženého paměťového média

Přiložené CD obsahuje:

- adresář se zdrojovými kódy této technické zprávy,
- adresář se zdrojovými kódy obrázků,
- tuto technickou zprávu ve formátu PDF,
- manuál k instalaci systému,
- adresář se serverovou částí systému obsahující:
  - balíček `perun` znázorněný na obrázku A.2,
  - adresář `tests` se zdrojovými soubory testů,
  - adresář `docs` obsahující zdrojové soubory generované dokumentace,
  - adresář `dist` s instalačním balíčkem,
- adresář s klientskou části systému obsahující:
  - balíček `perun_client` znázorněný na obrázku A.1,
  - adresář `dist` s instalačním balíčkem,
- adresář se soubory relevantními k nasazení:
  - konfigurační soubor použitý na zařízení Raspberry Pi 3,
  - skript pro vytvoření požadovaných objektů v systému,
- adresář obsahující fotodokumentaci cílového prostředí,
- adresář se závislostmi.