SŠPU
STŘEDNÍ ŠKOLA PRŮMYSLOVÁ
A UMĚLECKÁ, OPAVA

ZÁVĚREČNÁ STUDIJNÍ PRÁCE

dokumentace

Systém pro sdílení nabíječek elektroaut s využitím OCPP

Autor: Filip Jedlička

Obor: 18-20-M/01 INFORMAČNÍ TECHNOLOGIE
se zaměřením na počítačové sítě a programování

Třída: IT4

Školní rok: 2025/26

Prohlášení

Prohlašuji, že jsem závěrečnou práci vypracoval samostatně a uvedl veškeré použité informační zdroje.

Souhlasím, aby tato studijní práce byla použita k výukovým a prezentačním účelům na Střední průmyslové a umělecké škole v Opavě, Praskova 399/8.

V Opavě 1. 1. 2026

. . . . . .

Podpis autora

Abstrakt (CZ)

Tato práce se zabývá návrhem a realizací systému pro sdílení domácích nabíječek elektromobilů, který uživatelům umožňuje zpřístupnit svou nabíječku ostatním podobně jako model Airbnb. V Česku sice není infrastruktura nabíjecích stanic kriticky nedostatečná, ale komunitní projekt tohoto typu může zvýšit jejich dostupnost, zlevnit nabíjení a urychlit návratnost domácích fotovoltaických elektráren. Existuje komunita uživatelů, kteří si nabíječky půjčují neformálně, avšak doposud se domlouvají složitě přes chatovací platformy.

Klíčová slova

OCPP 1.6J, nabíjecí stanice, elektromobilita, sdílení nabíječek, RFID autorizace, FastAPI, Node.js, WebSocket server, PostgreSQL, JWT, API Key, RBAC.

Abstract (EN)

This thesis focuses on the design and implementation of a system for sharing home electric vehicle chargers, allowing users to make their chargers available to others in a manner similar to the Airbnb model. Although the charging infrastructure in the Czech Republic is not critically insufficient, a community-driven project of this type can increase charger availability, reduce charging costs, and accelerate the return on investment for home photovoltaic systems. A community of users who share chargers already exists, but they currently rely on informal and inefficient coordination through chat platforms.

Keywords

OCPP 1.6J, charging station, electromobility, charger sharing, RFID authorization, FastAPI, Node.js, WebSocket server, PostgreSQL, JWT, API Key, RBAC.

Obsah

Úvod

V posledních letech zaznamenává Česká republika, i díky dotačním titulům Evropské unie, výrazný nárůst počtu domácích fotovoltaických elektráren. Tento trend, společně s rostoucí dostupností elektromobilů a tlakem na snižování provozních nákladů, vedl k masivnímu rozšíření privátní nabíjecí infrastruktury. Tyto domácí nabíjecí body (wallboxy) však zůstávají po většinu času nevyužité, jelikož slouží primárně pouze pro potřeby majitele nemovitosti.

Sdílení těchto soukromých nabíječek by přineslo benefity všem stranám: majitelům by zkrátilo návratnost investice do fotovoltaiky a wallboxu, a zároveň by došlo k efektivnímu zahuštění sítě nabíjecích stanic, která je v tuzemsku stále nedostatečná.

V současnosti je trh se sdílením privátních nabíječek roztříštěný. Existují neformální komunity na sociálních sítích, kde je domluva zdlouhavá a manuální, nebo komerční projekty (např. evmapa.cz či goplugable.com), které jsou však často administrativně náročné, vyžadují specifický hardware nebo cílí primárně na firemní klientelu.

Cílem této práce je navrhnout a implementovat otevřený systém, který propojí existující domácí nabíječky a zajistí jejich automatizovanou správu. Systém bude zajišťovat komunikaci se stanicí, autorizaci uživatelů pomocí RFID karet, řízení nabíjecího procesu a následné vyúčtování spotřebované energie. Důraz je kladen na to, aby řešení nevyžadovalo od majitelů nákup proprietárního hardware, ale využívalo standardizovaný protokol.

1 VÝBĚR POSTUPŮ A TECHNOLOGIÍ

Při návrhu architektury systému byl kladen důraz na modularitu, škálovatelnost a použití moderních, ale ověřených technologií. Systém je navržen jako sada mikroslužeb běžících v kontejnerizovaném prostředí Docker.

1.1 OCPP PROTOKOL

Jako klíčový prvek komunikace mezi serverem a hardwarem nabíjecí stanice byl zvolen protokol **OCPP (Open Charge Point Protocol)** ve verzi **1.6 JSON (OCPP-J)**. Ačkoliv již existují novější specifikace (2.0.1, 2.1), verze 1.6 je v současnosti průmyslovým standardem s nejširší podporou výrobců domácích wallboxů. Použití formátu JSON namísto staršího SOAP (XML) výrazně zjednodušuje parsování zpráv a snižuje datovou náročnost, což je klíčové při komunikaci přes mobilní sítě. Pokud by měl projekt v budoucnu pokračovat, je architektura připravena na implementaci novějších verzí pro zajištění maximální kompatibility.

1.2 OCPP BACKEND

Pro službu zajišťující přímou komunikaci s nabíječkami (CSMS - Charging Station Management System) bylo zvoleno prostředí **Node.js**. Hlavním důvodem je jeho asynchronní architektura řízená událostmi (event-driven), která je ideální pro zpracování velkého množství WebSocket spojení v reálném čase. Node.js efektivně funguje jako router, který přijímá zprávy od nabíječek, validuje je a předává relevantní data do API backendu.

1.3 API BACKEND

Druhou částí systému je API backend, pro který byl zvolen jazyk **Python** s frameworkem **FastAPI**. Tato služba tvoří „mozek“ celého systému. Umožňuje OCPP backendu (a v budoucnu i dalším klientům, jako jsou webové či mobilní aplikace) komunikovat s databází a nabízí sadu nástrojů pro správu uživatelů a nabíječek. FastAPI bylo zvoleno pro svou rychlost, automatickou generaci dokumentace (Swagger UI) a striktní typovou kontrolu dat (Pydantic).

1.4 DATABÁZOVÉ SYSTÉMY

Architektura využívá kombinaci dvou databázových systémů pro různé typy dat:

- **PostgreSQL:** Slouží jako primární perzistentní úložiště pro trvalá data (uživatelé, transakce, definice nabíječek).
- **Redis:** Slouží jako in-memory úložiště pro dočasná a často měněná data. Ukládají se zde například aktuální stavy konektorů nebo poslední autorizované RFID tagy pro rychlý přístup.

2 Analýza a návrh řešení

Po výběru vhodných technologií následovala fáze detailní analýzy požadavků a návrhu architektury celého systému.

2.1 Metodika vývoje: Přechod na API-First

Původním záměrem projektu bylo vytvoření kompletního „Full-stack“ řešení zahrnujícího jak serverovou část, tak klientskou webovou aplikaci. V průběhu analýzy a prvotních fází vývoje se však ukázalo, že komplexita protokolu OCPP a požadavky na robustnost backendu jsou natolik rozsáhlé, že by vývoj kvalitního frontendu v časovém rámci práce vedl k nutným kompromisům v kvalitě celého systému.

Z tohoto důvodu bylo učiněno strategické rozhodnutí přejít na přístup **API-First**. Tento model klade maximální důraz na návrh a implementaci stabilního, dobře zdokumentovaného a bezpečného rozhraní (API).

Tento přístup přináší několik výhod:

- **Nezávislost na klientovi:** Backend není svázán s konkrétní vizuální podobou a může v budoucnu obsluhovat web, mobilní aplikaci (iOS/Android) nebo systémy třetích stran.
- **Kvalita architektury:** Umožnilo to věnovat více času klíčovým aspektům, jako je správná implementace standardu OCPP, bezpečnost a testování, namísto ladění uživatelského rozhraní.
- **Škálovatelnost:** Systém je od začátku navržen modulárně, což usnadňuje jeho budoucí rozšiřování.

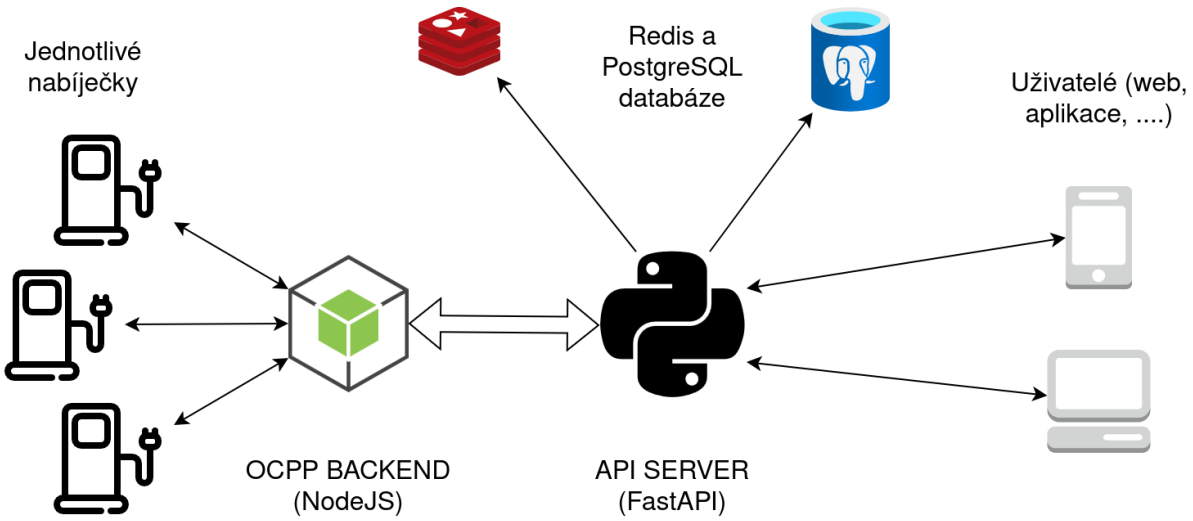
V rámci této práce je tedy výstupem kompletní backendová platforma, jejíž funkčnost je demonstrována a testována prostřednictvím dokumentovaných API endpointů.

2.2 Architektura systému

Systém je navržen jako sada mikroslužeb běžících v oddělených kontejnerech, což zajišťuje modularitu a snadnou údržbu. Komunikace uvnitř systému probíhá ve třech rovinách:

1. **Komunikace s hardwarem (WebSocket):** Fyzické stanice udržují trvalé spojení se službou `ocpp-backend` (Node.js). Zprávy protokolu OCPP jsou přenášeny v reálném čase formátem JSON, což vyžaduje, aby tato služba byla stavová (stateful).
2. **Interní komunikace (REST API):** Logiku požadavků řeší služba `fastapi-backend` (Python). Node.js server se jí dotazuje pomocí synchronních HTTP požadavků (např. při autorizaci karty). Tato komunikace je izolována v privátní síti a chráněna API klíčem.
3. **Datová vrstva:** Aplikační jádro využívá **PostgreSQL** pro trvalé uložení uživatelských dat a historie transakcí. Pro sdílení aktuálního stavu konektorů (pro potřeby frontendové mapy) se využívá rychlá in-memory databáze **Redis**.
4. **Externí přístup (Public API):** Klientské aplikace přistupují k systému skrze veřejné REST rozhraní služby `fastapi-backend`. Tato komunikace je na rozdíl od interního provozu zabezpečena pomocí **JWT tokenů** (OAuth2 standard). Slouží pro autentizaci uživatelů, správu jejich nabíječek a zobrazování historie nabíjení.

Schéma na obrázku 2.1 vizualizuje tyto toky dat a oddělení veřejné části od privátní infrastruktury.



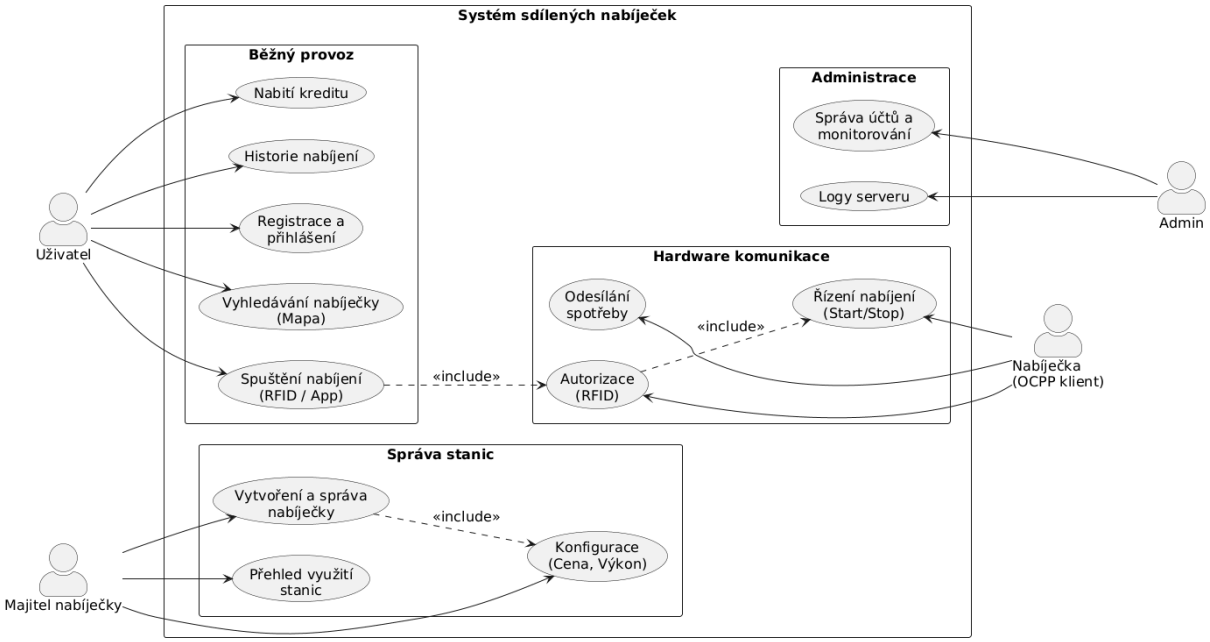
Obrázek 2.1: Architektura systému a komunikace mikroslužeb

2.3 Funkční požadavky (Use Case)

Pro ujasnění interakcí mezi uživateli a systémem byl vytvořen diagram případů užití (Use Case Diagram), který zachycuje **cílový stav celého systému**. Diagram slouží jako plán pro kompletní funkcionalitu platformy.

V rámci této práce a realizovaného prototypu byly implementovány nejdůležitější případy užití (tzv. Core Features) nezbytné pro prokázání funkčnosti konceptu:

- **Nepřihlášený uživatel:** Zobrazení mapy nabíječek, možnost registrace
- **Registrovaný uživatel:** Přihlášení, správa účtu a vlastních nabíječek, autorizace pomocí RFID a prohlížení historie nabíjení,
- **Administrátor:** Monitorování a technická správa systému.



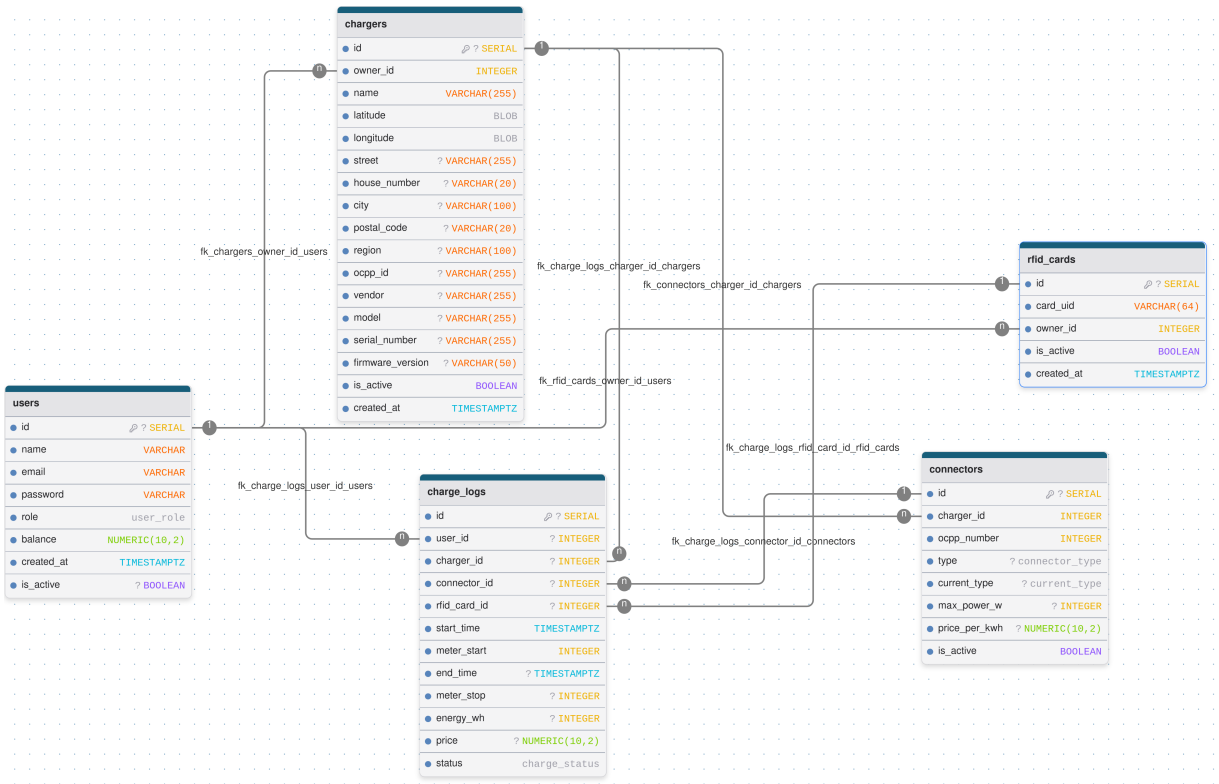
Obrázek 2.2: Diagram případů užití (návrh celého systému)

2.4 Datový model

Návrh databáze vychází z požadavků na evidenci uživatelů, nabíječek a historie nabíjení. Jako relační databázový systém byl zvolen PostgreSQL. Při návrhu schématu byl kladen důraz na zachování datové integrity a historie. U klíčových entit je proto implementován mechanismus tzv. Soft Delete. Místo odstranění záznamů dochází pouze k jejich deaktivaci (nastavení příznaku `is_active` na `false`). Tím je zajištěno, že smazáním uživatele nebo nabíječky nedojde ke ztrátě vazeb na již proběhlé transakce a statistiky.

Klíčové entity systému jsou:

- **Users:** Ukládá přihlašovací údaje, role a zůstatky uživatelů.
- **Chargers & Connectors:** Definují fyzické nabíjecí stanice a jejich konektory. Každá nabíječka má unikátní `ocpp_id`, který slouží k identifikaci v síti.
- **RFID Cards:** Tabulka čipů pro autorizaci, vazba na uživatele (Owner).
- **Charge Logs:** Záznamy o proběhlých transakcích (kdo, kde, kolik kWh, čas startu a konce).



Obrázek 2.3: Entitně-relační diagram (ERD) databáze

3 VÝVOJOVÉ A PROVOZNÍ PROSTŘEDÍ

3.1 ADRESÁŘOVÁ STRUKTURA PROJEKTU

Celý projekt je organizován jako tzv. **monorepo** (monolitický repozitář), což znamená, že zdrojové kódy všech mikroslužeb, dokumentace i konfigurační soubory pro nasazení jsou udržovány na jednom místě. Toto uspořádání zjednodušuje správu verzí a sdílení znalostí napříč projektem.

Níže je uveden zjednodušený výpis adresářové struktury s popisem klíčových částí:

```
|-- docker-compose.yaml        # Orchestrace pro produkci
|-- docker-compose.dev.yaml    # Orchestrace pro vývoj
|-- README.md                  # Obsahuje návod ke spuštění aplikace
|-- docs/                      # Dokumentace a diagramy
+-- fastapi-backend/           # API Server (Python)
|   |-- Dockerfile             # Image pro produkci
|   |-- Dockerfile.dev         # Image pro vývoj (hot-reload)
|   |-- alembic/               # Migrace databáze
|   +-- app/
|       |-- main.py            # Vstupní bod, init DB, CORS
|       |-- api/v1/            # Routery a endpointy
|       |-- core/              # Konfigurace, security
|       |-- db/                # Schéma DB
|       |-- models/            # SQLAlchemy modely
|       +-- services/          # Logika endpointů
+-- ocpp-backend/              # OCPP Server (Node.js)
|   |-- Dockerfile             # Image pro produkci
|   |-- Dockerfile.dev         # Image pro vývoj (hot-reload)
|   +-- src/
|       |-- index.js           # WebSocket server, handshake
|       |-- router.js          # Směrování OCPP zpráv
|       |-- handlers/          # Logika handlerů (Boot, Heartbeat...)
|       |-- schemas/           # Schémata pro validaci JSON payloadů
```

Kód 3.1: Adresářová struktura projektu (vybrané soubory)

Struktura je rozdělena do tří hlavních kořenových adresářů:

- `fastapi-backend`**:** Obsahuje veškerou logiku API serveru. Vnitřní členění odpovídá vrstvené architektuře (API → Services → Models).
- `ocpp-backend`**:** Obsahuje implementaci WebSocket serveru v Node.js. Složka `schemas` obsahuje oficiální JSON definice protokolu OCPP, které slouží k validaci příchozích a odchozích zpráv.
- `docs`**:** Slouží jako centrální úložiště pro technickou dokumentaci, specifikace protokolu a jednotlivých diagramů.

3.2 KONTEJNERIZACE A ORCHESTRACE SLUŽEB

Aby bylo zajištěno konzistentní prostředí pro běh aplikace na lokálním počítači vývojáře i na produkčním serveru, je celý systém kontejnerizován pomocí technologie Docker. Architektura je navržena tak, aby striktně oddělovala vývojový a produkční režim.

3.2.1 Strategie sestavování obrazů (Dockerfiles)

Pro každou mikroslužbu (FastAPI i Node.js) existují dva typy definic pro sestavení obrazu.

Produkční prostředí (Multi-stage build)

Pro nasazení aplikace je kladen důraz na bezpečnost a minimalizaci velikosti výsledného obrazu. Využívá se technika *Multi-stage builds*. V první fázi (Builder) se nainstalují kompilátory a sestaví se závislosti. Do druhé, finální fáze (Runner), se zkopírují pouze výsledné artefakty a nutné knihovny. Tím se zmenší velikost obrazu z stovek megabajtů na desítky a v kontejneru nezůstávají zbytečné nástroje, které by mohl útočník zneužít.

Vývojové prostředí (Hot-reloading)

Vývojová verze (Dockerfile.dev) je naopak optimalizována pro pohodlí programátora. Neobsahuje optimalizace velikosti, ale instaluje nástroje pro ladění. Klíčovým prvkem je tzv. hot-reloading. To znamená, že proces sleduje změny v souborech a při uložení kódu automaticky restartuje server, což výrazně zrychluje vývoj.

3.2.2 Orchestrace pomocí Docker Compose

Jelikož se systém skládá ze čtyř služeb (API, OCPP Server, PostgreSQL, Redis), je pro jejich správu použit nástroj Docker Compose. Konfigurace je rozdělena do vrstev:

1. `docker-compose.yaml` **(Base/Prod):** Definuje základní služby, sítě a produkční nastavení. Služby komunikují výhradně po interní Docker síti a databáze nevystavuje porty do internetu.
2. `docker-compose.dev.yaml` **(Override):** Tento soubor se používá pouze při lokálním vývoji a přepisuje nastavení produkce.

Hlavní rozdíly v konfiguraci ukazuje následující tabulka:

Aspekt konfigurace	Vývojové prostředí (Dev)	Produkční prostředí (Prod)
Zdrojový kód	Dynamické připojení (Bind Mount)	Statický obraz (Immutable Build)
Síťová expozice	Porty databází mapovány na hosta (localhost)	Databázové porty izolované v interní síti (Internal Only)
Strategie běhu	Hot-reloading (sledování změn)	Restart Policy: Always
Úroveň logování	Debug (vysoká granularita)	Info / Warning (minimalizace I/O)
Správa závislostí	Flexibilní z definičních souborů (`pyproject.toml`, `package.json`)	Deterministická ze zámků (`uv.lock`, `package-lock.json`)

Tabulka 3.1: Srovnání konfigurace vývojového a produkčního prostředí

3.3 Logování a diagnostika

Vzhledem k tomu, že je systém rozdělen do více nezávislých kontejnerů, nebylo možné spoléhat na jednoduché ladění a výpisy chyb v reálném čase. Z toho důvodu byl robustní logovací mechanismus implementován jako jedna z prvních komponent celého systému, ještě před samotnou logikou protokolu OCPP.

V kontejnerizovaném prostředí je logování řešeno standardizovaným výstupem do proudů `stdout` a `stderr`. Docker tyto proudy zachytává a umožňuje jejich centralizované zobrazení příkazem `docker logs`, případně jejich agregaci externími nástroji.

Implementace konzistentního logování přinesla během vývoje několik klíčových výhod:

- **Sledovatelnost (Traceability):** Každý logovací záznam obsahuje časové razítko a identifikátor služby. Díky tomu lze snadno rekonstruovat tok požadavku – od přijetí zprávy na Node.js serveru, přes její zpracování, až po zavolání Python API a uložení do databáze.
- **Úrovně závažnosti (Log Levels):** Systém rozlišuje úrovně `DEBUG`, `INFO`, `WARNING` a `ERROR`. Ve vývojovém prostředí je zapnuta úroveň `DEBUG`, která vypisuje detailní obsah všech přenášených JSON zpráv. To bylo neocenitelné při ladění komunikace s hardwarem, kdy i malá odchylka v syntaxi OCPP zprávy může způsobit odmítnutí spojení.

Tento přístup umožnil rychle odhalit chyby v integraci mezi mikroslužbami, které by jinak zůstaly skryté nebo by se projevovaly pouze nestandardním chováním připojených nabíječek.

4 OCPP BACKEND

Tato kapitola detailně rozebírá logiku jednotlivých procesů, které probíhají na pozadí komunikace mezi nabíjecí stanicí a serverem. Implementace pokrývá profil *Core*, který je nezbytný pro základní fungování nabíjení, autorizaci a sběr telemetrických dat.

Backend je navržen tak, aby reagoval na události (Event-driven). Každá příchozí zpráva spustí specifickou rutinu (handler), která provede validaci dat, komunikaci s databází a následně odešle standardizovanou odpověď zpět do nabíjecí stanice.

4.1 AUTOMATICKÉ NAČÍTÁNÍ HANDLERŮ A SCHÉMAT

Pro zajištění snadné rozšiřitelnosti serveru byl implementován mechanismus automatického načítání modulů (Auto-loading).

Při startu aplikace server proskenuje adresáře `src/handlers` a `schemas`. Pokud nalezne soubor, jehož název odpovídá názvu OCPP akce (např. `BootNotification.js`), automaticky jej importuje a zaregistruje do směrovací logiky.

Tento přístup výrazně zefektivňuje vývoj. Pokud chce programátor přidat podporu pro novou zprávu, stačí mu vytvořit jediný soubor se správným názvem. Odpadá nutnost manuální registrace importů v hlavním souboru aplikace, což eliminuje riziko chyb a udržuje kód routeru čistý a obecný.

```
...
export async function loadHandlers() {
        const handlersDir = path.join(process.cwd(), "src/handlers");
        const files = fs.readdirSync(handlersDir);
        const handlers = {};
        for (const file of files) {
                if (!file.endsWith(".js")) continue;
                const name = path.basename(file, ".js");
                const modulePath = path.join(handlersDir, file);
                // Dynamicky import modulu
                const handlerModule = await import(modulePath);
                // Ulozeni handleru do mapy. Ocekavame 'export default'
                handlers[name] = handlerModule.default;
        }
... }
```

Kód 4.1: Část implementace dynamického načítání handlerů (src/utils/handlerLoader.js)

4.2 HANDSHAKE A BOOTNOTIFICATION

- **Trigger:** Start zařízení.
- **Obsah zprávy:** Identifikace výrobce (`chargePointVendor`), model (`chargePointModel`) a verze firmwaru.
- **Logika serveru:** Server nejprve extrahuje unikátní identifikátor nabíječky z URL spojení. Následně ověří v databázi, zda je toto zařízení v systému registrováno a zda má příznak `is_active`. Pokud je zařízení neznámé nebo blokované, server vrátí status `Rejected`, čímž nabíječce zakáže další komunikaci.
- **Odpověď':** V případě úspěšného ověření server vrací status `Accepted`, aktuální serverový čas (pro synchronizaci hodin nabíječky) a interval pro zasílání zpráv Heartbeat.

4.3 STATUSNOTIFICATION (STAV A DETEKCE KONEKTORŮ)

Pomocí této zprávy nabíječka informuje centrální systém o změně svého stavu nebo o stavu konkrétního konektoru. To umožňuje systému zobrazovat uživatelům v reálném čase, zda je nabíječka volná.

- **Trigger:** Připojení kabelu, porucha, uvolnění konektoru.
- **Klíčové stavy:** `Available` (volno), `Preparing` (kabel připojen, čeká na autorizaci), `Charging` (nabíjí), `Faulted` (chyba).
- **Funkce Autodiscovery:** Systém využívá tuto zprávu k automatické detekci konfigurace stanice. Pokud server obdrží notifikaci od konektoru, který v databázi dosud neexistuje (např. konektor č. 2 na nové stanici), automaticky ho v systému vytvoří. To výrazně zjednodušuje proces nasazování nových stanic, protože správce nemusí ručně definovat počet konektorů.

4.4 AUTHORIZE (OVĚŘENÍ UŽIVATELE)

Před zahájením nabíjení musí uživatel prokázat svou totožnost, obvykle přiložením RFID karty nebo čipu.

- **Trigger:** Přiložení RFID tagu ke čtečce.
- **Obsah zprávy:** Unikátní identifikátor tagu (`idTag`).
- **Logika serveru:** Server provádí dotaz do databáze uživatelů. Kontrolují se tři podmínky:
  1. Existuje daný RFID tag v systému?
  2. Je svázán s aktivním uživatelským účtem?
  3. Je tag aktivní (pole `is_active`)?
- **Odpověď:** Výsledkem je status autorizace. Pokud je `Accepted`, nabíječka odblokuje konektor a připraví se k dodávce energie. V opačném případě (`Invalid`, `Blocked`) zůstane konektor uzamčen.

4.5 STARTTRANSACTION (ZAHÁJENÍ NABÍJENÍ)

Toto je kritická zpráva, která iniciuje tok energie a začátek účtovacího období.

- **Trigger:** Úspěšná autorizace a následné připojení kabelu (nebo naopak).
- **Obsah zprávy:** Číslo konektoru, použitý RFID tag a aktuální stav elektroměru (`meterStart`) ve Wh.
- **Logika serveru:** Server vytvoří nový záznam v tabulce transakcí (`ChargeLogs`). Záznamu je přiděleno unikátní `transactionId`, které server vrátí nabíječce. Toto ID je klíčové, protože v budoucnu spáruje ukončení nabíjení se správným začátkem. Pokud by se zápis do databáze nezdařil, server transakci odmítne a nabíjení nezačne.

4.6 METERVALUES (PRŮBĚH A TELEMETRIE)

Během aktivní transakce nabíječka periodicky odesílá data o průběhu nabíjení.

- **Trigger:** Uplynutí nastaveného intervalu nebo změna odebíraného výkonu o definovanou hodnotu.
- **Obsah zprávy:** Telemetrická data, typicky napětí (V), proud (A), aktuální výkon (W) a teplota.
- **Využití dat:** Poslední zaslanou hodnotu lze využít jako zálohu v případě výpadku komunikace, kdy stanice neodešle zprávu `StopTransaction`. Systém v takové situaci po uplynutí ochranné lhůty transakci automaticky ukončí a pro finální vyúčtování použije poslední známý stav elektroměru z `MeterValues`.

4.7 STOPTRANSACTION (UKONČENÍ A VYÚČTOVÁNÍ)

Zpráva ukončující nabíjecí relaci. Na jejím základě dochází k finančnímu vyrovnání.

- **Trigger:** Odpojení kabelu, opětovné přiložení karty nebo dálkový příkaz.
- **Obsah zprávy:** Konečný stav elektroměru (`meterStop`), ID transakce a důvod ukončení.
- **Logika vyúčtování:** Server vyhledá otevřenou transakci podle ID. Následně vypočítá celkovou spotřebovanou energii jako rozdíl mezi koncovým a počátečním stavem elektroměru:

$$E_{total} = \text{meterStop} - \text{meterStart}$$

  Výsledná hodnota je uložena do historie a transakce je označena jako ukončená. Na základě tarifu nabíječky je následně vypočítána cena, která je stržena z kreditu uživatele.

5 API BACKEND (FASTAPI)

Zatímco předchozí kapitola popisovala komunikaci s hardwarem, tato část se věnuje centrálnímu bodu celé architektury – REST API serveru. Ten byl implementován v jazyce Python s využitím moderního asynchronního frameworku **FastAPI**.

Tato služba plní roli aplikačního jádra. Zajišťuje perzistenci dat do databáze PostgreSQL, komunikaci s cache systémem Redis a poskytuje rozhraní jak pro interní potřeby OCPP serveru, tak případně pro klientské aplikace (frontend).

5.1 ARCHITEKTURA A ORGANIZACE KÓDU

Pro zajištění udržitelnosti a testovatelnosti kódu byl využit návrhový vzor **Dependency Injection** (vstřikování závislostí).

5.1.1 Sdílené závislosti (Dependencies)

Veškeré opakující se logické celky jsou vyčleněny do modulu `app/api/v1/deps.py`. FastAPI tyto závislosti automaticky „vstřikuje“ do jednotlivých endpointů, které je vyžadují. Klíčové závislosti zahrnují:

- `get_db`: Zajišťuje, že každý požadavek dostane vlastní izolovanou relaci s databází, která je po dokončení požadavku automaticky uzavřena.
- `get_current_user`: Zpracovává bezpečnostní token z hlavičky požadavku, ověřuje jeho podpis a načítá data uživatele.
- `verify_api_key`: Slouží k ověření interních požadavků od jiných mikroslužeb.

5.1.2 Datová integrita (Enums a Pydantic)

Aby se předešlo chybám v datech, systém striktně využívá typovou kontrolu.

- **Enums:** Hodnoty, které mohou nabývat jen omezené množiny stavů (např. role uživatele `admin/user` nebo typ konektoru `Type2/CCS`), jsou definovány jako výčtové typy v souboru `enums.py`.
- **Pydantic modely:** Všechna vstupní i výstupní data jsou validována knihovnou Pydantic. Pokud klient pošle data ve špatném formátu (např. chybějící e-mail při registraci), API automaticky vrátí detailní chybovou hlášku `422 Unprocessable Entity`, aniž by se spustila logika aplikace.

5.2 Bezpečnost a autentizace

Bezpečnostní model aplikace je navržen s ohledem na dva odlišné typy klientů API.

5.2.1 OAuth2 a JWT (Pro uživatele)

Pro autentizaci koncových uživatelů (lidí) je využit standardní protokol **OAuth2** s Bearer tokeny. Po úspěšném přihlášení (verifikaci hesla pomocí `bcrypt`) server vygeneruje **JWT (JSON Web Token)**. Tento token je podepsán tajným klíčem serveru a obsahuje ID uživatele a dobu platnosti. Díky tomu je API *bezstavové* (stateless) – server si nemusí pamatovat přihlášené uživatele v paměti RAM, což usnadňuje škálování.

5.2.2 API Key (Pro OCPP Server)

Komunikace mezi Node.js serverem (OCPP) a Python API (FastAPI) probíhá na pozadí (Machine-to-Machine). Použití expirovaných JWT tokenů by zde bylo nepraktické a náchylné k chybám při obnovování. Místo toho je pro interní komunikaci zavedena autentizace pomocí statického **API klíče**, který se přenáší v hlavičce `X-API-Key`. Tento klíč je sdílen pouze mezi kontejnery uvnitř zabezpečené Docker sítě.

5.3 Rozdělení API (Endpoints)

API je logicky rozděleno do dvou sekcí, které odpovídají výše zmíněným bezpečnostním modelům - interní (OCPP server) a veřejné.

5.3.1 Veřejné API (User-facing)

Tyto endpointy jsou určeny pro frontendovou aplikaci a uživatele. Implementují **RBAC (Role-Based Access Control)** – řízení přístupu na základě rolí. Typickým příkladem je správa uživatelů (`user.py`):

- **Nepřihlášený uživatel:** Může volat pouze endpoint pro registraci a přihlášení.
- **Běžný uživatel:** Může číst a upravovat pouze svůj vlastní profil.
- **Administrátor:** Má plná práva pro výpis, úpravu i mazání všech uživatelů v systému.

5.3.2 Interní API (Service-to-Service)

Modul `internal.py` obsahuje endpointy, které slouží výhradně pro potřeby OCPP serveru. Tyto cesty nejsou veřejně dokumentovány ve Swagger UI a jsou chráněny API klíčem. Právě tyto endpointy vykonávají databázové operace, které byly teoreticky popsány v předchozí kapitole o OCPP logice:

- `POST /internal/boot-notification`: Ověřuje existenci nabíječky při startu.
- `POST /internal/authorize`: Kontroluje validitu RFID karty.
- `POST /internal/transaction/start`: Zakládá záznam o nabíjení a vrací ID transakce.
- `POST /internal/connector-status`: Aktualizuje stav v Redisu pro zobrazení na mapě.

Tímto rozdělením je zajištěno, že kritické operace systému jsou odděleny od běžné uživatelské interakce, což zvyšuje bezpečnost celého řešení.

6 Životní cyklus aplikace v praxi

Tato kapitola stručně shrnuje integraci všech vytvořených komponent (OCPP server, API, Databáze) na příkladu typického scénáře použití systému.

6.1 Příprava (Onboarding)

Proces začíná registrací. Majitel nabíječky si vytvoří účet přes API, zaregistruje své zařízení a server mu vygeneruje `ocpp_id`. Tyto údaje zadá do nastavení nabíječky.

6.2 Inicializace (Booting)

Uživatel toto ID zadá do konfigurace nabíječky spolu s adresou serveru.

1. Jakmile nabíječka nastartuje, pošle žádost o Websocket spojení.
2. Node.js server se dotáže Python API: „Existuje toto zařízení?“. API odpoví ano a OCPP server povolí spojení.
3. Nabíječka pošle `BootNotification`. OCPP server předá obsah API a to doplní metadata o nabíječce do databáze.
4. Nabíječka pošle sérii `StatusNotification` pro své konektory. Systém díky funkci *Autodiscovery* automaticky vytvoří entity konektorů v databázi.
5. Uživatel si ke konektorům doplní jejich maximální výkon a cenu.

Nyní je nabíječka viditelná v API jako „Dostupná“.

6.3 Průběh nabíjení

1. **Připojení kabelu**: Po připojení kabelu pošle nabíječka `StatusNotification`. Server zapíše změnu stavu do Redis databáze. Nabíječka vyčkává na ověření karty.
2. **Autorizace:** Řidič přiloží RFID kartu. Nabíječka pošle `Authorize`. Server ověří kartu. Pokud je vše v pořádku, odpoví `Accepted`. Pokud karta v systému neexistuje pošle `Invalid`, nebo `Blocked` pokud má `is_active` hodnotu `False`.
3. **Průběh:** Každou minutu chodí `MeterValues` (napětí, proud). Tato data se ukládají pro analytiku, nebo pro případ výpadku komunikace.
4. **Konec:** Řidič odpojí kabel. Nabíječka pošle `StopTransaction`.

6.4 VYÚČTOVÁNÍ

Okamžitě po přijetí `StopTransaction` API server vypočítá celkovou dodanou energii. Na základě nastavené ceny za kWh se vypočte finální částka, která je stržena z virtuální peněženky řidiče a připsána majiteli stanice. Konektor se přepne zpět do stavu „Dostupný“.

7 TESTOVÁNÍ A VALIDACE SYSTÉMU

Vzhledem k distribuované povaze systému a závislosti na externím hardwaru probíhalo ověřování funkčnosti ve dvou fázích. Prvním krokem bylo testování proti softwarovým simulátorům a druhou fází byla integrace s reálnou domácí nabíjecí stanicí.

7.1 VALIDACE POMOCÍ SIMULÁTORŮ

Pro ověření správnosti implementace serverové logiky (OCPP Backend i API) byly využity nástroje vyvinuté komunitou okolo protokolu OCPP. Použití více různých simulátorů zajistilo, že implementace není „ušita na míru“ jednomu klientovi, ale je univerzální.

K testování byly využity následující nástroje:

1. **OCPPSimulator** (autor *ozgurbayram*)[1]: Velmi povedený a přehledný webový nástroj k simulaci nabíjení. Umožňuje přidání i více nabíječek.
2. **OCPP Virtual Charge Point** (autor *solidstudiosh*)[2]: Komplexnější terminálový simulátor, který umožňuje větší kontrolu na posílanými zprávami.

V prostředí těchto simulátorů systém fungoval bezchybně. Byly úspěšně ověřeny scénáře tzv. *Happy Path*:

- Připojení stanice a handshake (BootNotification).
- Ověření RFID tagu v databázi (Authorize).
- Správné založení transakce při startu nabíjení (StartTransaction).
- Příjem a ukládání průběžných měření (MeterValues).
- Korektní ukončení a vyúčtování transakce (StopTransaction).

Tyto testy potvrdily, že aplikační logika backendu je implementována v souladu se specifikací OCPP 1.6J.

7.2 TESTOVÁNÍ S REÁLNÝM HARDWAREM

Pro ověření v reálném provozu byla využita domácí nabíjecí stanice. Cílem bylo potvrdit, že systém dokáže komunikovat i s fyzickým zařízením a řídit tok energie.

[1] Dostupné z: `https://github.com/ozgurbayram/OCPPSimulator`

[2] Dostupné z: `https://github.com/solidstudiosh/ocpp-virtual-charge-point`

7.2.1 Průběh integrace

Stanice byla nakonfigurována pro komunikaci s lokálně běžícím serverem. Následné testy prokázaly funkčnost komunikačního kanálu a správnou implementaci stavového automatu:

- **Konektivita:** Stanice úspěšně navázala WebSocket spojení.
- **Registrace:** Server správně identifikoval zařízení a přijal `BootNotification`.
- **Autorizace:** Po přiložení RFID čipu stanice odeslala požadavek `Authorize`, který server ověřil v databázi a vrátil kladnou odpověď (`Accepted`).
- **Start transakce:** Stanice úspěšně zareagovala na autorizaci a odeslala zprávu `StartTransaction`. Server tuto zprávu zpracoval, založil záznam v databázi a vrátil unikátní `transactionId`.

7.2.2 Zjištěná specifika hardwaru

I přes korektní průběh komunikační sekvence a potvrzení `StartTransaction` nedošlo u testované stanice k sepnutí silových prvků. Zařízení po 15 sekundách ukončilo relaci zprávou `StopTransaction`.

Toto chování naznačuje, že ačkoliv je serverová logika validní (ověřeno simulátory), firmware stanice vyžaduje pro fyzické sepnutí specifické podmínky (např. stav konektoru nebo uzamčení kabelu), případně očekává nestandardní parametry v odpovědi serveru.

7.3 Závěr testování

Testy na dvou nezávislých simulátorech potvrdily, že backend je plně funkční a dodržuje standard OCPP 1.6J. Specifické chování testovaného fyzického zařízení poukazuje na známou fragmentaci implementací protokolu mezi výrobci hardwaru.

Pro případné produkční nasazení by proto bylo nutné:

- **Rozšířit testování** na portfolio stanic různých výrobců (např. Alfen, Keba).
- **Zavést konfigurační profily**, které by dynamicky upravovaly parametry komunikace podle typu připojeného zařízení.

ZÁVĚR

Cílem této práce byl návrh a implementace backendového systému pro sdílení soukromých nabíjecích stanic elektromobilů. Výsledkem je funkční prototyp postavený na architektuře mikroslužeb, který využívá Node.js pro komunikaci přes protokol OCPP 1.6J a Python (FastAPI) pro správu dat. Práce ověřila, že otevřený standard OCPP umožňuje centrální řízení nabíječek nezávisle na jejich výrobci.

Během realizace došlo k odchýlení od původního návrhu v oblasti orchestrace síťového provozu. Původně bylo plánováno nasazení moderní reverse proxy Traefik pro automatické směrování kontejnerů. Protože však byl systém nasazován na server, kde již běžel webový server Apache obsluhující jiné služby, ukázalo se jako efektivnější využít stávající infrastrukturu. Namísto nasazení další vrstvy v podobě Traefiku bylo směrování na veřejné porty serveru. Toto rozhodnutí zjednodušilo správu serveru a eliminovalo konflikty na portech 80 a 443.

Vytvořené řešení v současné podobě pokrývá serverovou část systému – registraci, autorizaci, nabíjení i transakce. Pro komerční nasazení by bylo nutné systém rozšířit o uživatelské rozhraní (webovou a mobilní aplikaci) a provést širší testování kompatibility s různými typy fyzických nabíječek.

Potenciál pro další rozvoj vidím v integraci s domácí energetikou. Backend je připraven na implementaci dynamického řízení výkonu (Dynamic Load Management) na základě dat z fotovoltaické elektrárny, což by umožnilo čistě solární nabíjení.

Kompletní implementace aplikace je publikována v repozitáři na službě GitHub[3].

[3]Dostupné z: `https://github.com/JedlaTypek/shared-ev-chargers`

SEZNAM ZDROJŮ

[1] DOKULIL, Jakub. *Šablona pro psaní SOČ v programu LaTeX* [online]. Brno, 2020 [cit. 2020-08-24]. Dostupné z: https://github.com/Kubiczek36/SOC_sablona

[2] OPEN CHARGE ALLIANCE. *Open Charge Point Protocol 1.6* [online]. Edition 2. Arnhem: Open Charge Alliance, 2017 [cit. 2025-12-29]. Dostupné z: https://www.openchargealliance.org/downloads/

[3] OPEN CHARGE ALLIANCE. *Open Charge Point Protocol JSON 1.6: OCPP-J 1.6 Specification* [online]. Arnhem: Open Charge Alliance, 2017 [cit. 2025-12-29]. Dostupné z: https://www.openchargealliance.org/downloads/

[4] OPEN CHARGE ALLIANCE. *OCPP 1.6 Errata sheet* [online]. Verze 2025-04. Arnhem: Open Charge Alliance, 2025 [cit. 2025-12-29]. Dostupné z: https://www.openchargealliance.org/downloads/

[5] OPEN CHARGE ALLIANCE. *OCPP 1.6-J Errata sheet* [online]. Verze 2025-04. Arnhem: Open Charge Alliance, 2025 [cit. 2025-12-29]. Dostupné z: https://www.openchargealliance.org/downloads/

[6] OPENJS FOUNDATION. *Node.js Documentation* [online]. 2025 [cit. 2025-12-29]. Dostupné z: https://nodejs.org/en/docs/

[7] RAMÍREZ, Sebastián. *FastAPI: High performance, easy to learn, fast to code, ready for production* [online]. 2025 [cit. 2025-12-29]. Dostupné z: https://fastapi.tiangolo.com/

[8] DOCKER INC. *Docker Documentation* [online]. 2025 [cit. 2025-12-29]. Dostupné z: https://docs.docker.com/

[9] THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL 16 Documentation* [online]. 2025 [cit. 2025-12-29]. Dostupné z: https://www.postgresql.org/docs/

[10] REDIS LTD. *Redis Documentation* [online]. 2025 [cit. 2025-12-29]. Dostupné z: https://redis.io/docs/

Seznam obrázků

Seznam tabulek