Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj vstupního uzlu datové platformy pro data z IoT zařízení

Petr Anděl

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Petr Anděl

Informatika

Název práce

Vývoj vstupního uzlu datové platformy pro data z IoT zařízení

Název anglicky

Development of a data platform entry node for IoT devices

Cíle práce

Cílem práce je vytvoření základní struktury a funkcionality vstupního uzlu datové platformy, skrze který by byla stahována, tříděna a transformována data, která jsou čerpána z různých sběrných míst. Prvním dílčím cílem je návrh a implementace back-end části aplikace skrze platformu NodeRED tak, aby bylo možné zpracovávat data alespoň z jednoho vybraného sběrného místa. Druhým dílčím cílem je návrh a implementace front-end části aplikace, skrze kterou má být back-end část aplikace ovládána a monitorována.

Metodika

Diplomová práce sestává ze dvou částí – teoretických východisek a vlastního řešení.

Metodika zpracování teoretické části vychází ze studia odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována východiska pro zpracování praktické části.

Praktická část práce spočívá ve vytvoření back-end a front-end části aplikace, která má sloužit jako vstupní uzel pro data z IoT zařízení, která jsou dále odesílána do datové platformy univerzity. Vypracování vlastního řešení bude uzpůsobeno požadavkům a limitacím, které budou stanoveny před zpracováním praktické části práce. Pro back-end část aplikace je nutno použít platformu NodeRED. Využití platformy NodeRED a dostupných rozšíření je doporučeno i pro realizaci front-end části aplikace.

Doporučený rozsah práce

60-80 stran

Klíčová slova

IoT, vývoj, software, webová aplikace, NodeRED, VueJS

Doporučené zdroje informací

Dokumentace HTML, web: https://www.w3schools.com/html/default.asp
Dokumentace NodeRED, web: https://nodered.org/docs/
Dokumentace skriptovacího jazyka Javascript, web: https://developer.mozilla.org/en-US/docs/Web/JavaScript
MCFEDRIES, Paul. – Web Coding and Development All-In-One for Dummies – John Wiley & Sons, Incorporated, 2018. ISBN 9781119473923
RANJAN, Alok, Abhilasha SINHA a Ranjit BATTEWAD – JavaScript for Modern Web Development – BPB Publications, 2020. ISBN 9789389328721.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 4. 9. 2023

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 30. 11. 2023

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vývoj vstupního uzlu datové platformy pro data z IoT zařízení" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 24.3.2024 _____

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu diplomové práce Ing. Jiřímu Brožkovi, Ph.D. za odborné vedení a ochotnou pomoc v podobě konzultací v průběhu zpracovávání práce. Dále bych rád poděkoval Ing. Vojtěchu Novákovi, Ph.D. za poskytnutí možnosti podílet se na vývoji projektu datové platformy univerzity.

Vývoj vstupního uzlu datové platformy pro data z IoT zařízení

Abstrakt

Tato diplomová práce se zabývá vytvořením základní struktury a funkcionality vstupního uzlu datové platformy, skrze který by byla stahována, tříděna a transformována data, která jsou čerpána z různých sběrných míst.

V teoretické části práce jsou čtenáři přiblíženy základní pojmy či východiska z oblastí vývoje softwaru, do kterých tato práce zasahuje. V praktické části práce je zachycen návrh a realizace konkrétního řešení back-end a front-end části aplikace.

Klíčová slova: IoT, vývoj, software, webová aplikace, NodeRED, VueJS

Development of a data platform entry node for IoT devices

Abstract

This thesis deals with the creation of basic structure and functionality of the input node of the data platform, through which the data from various collection points would be downloaded, sorted and transformed.

In the theoretical part of the work, the reader is approached with basic concepts and starting points from the areas of software development which this work interferes with. The practical part of the work shows the result of design and implementation of a specific solution for the back-end and front-end part of the application.

Keywords: IoT, development, software, web application, NodeRED, VueJS

Obsah

1 Úvod

V době rozvíjející se digitalizace, rozvoje senzorů, kamer a dalších chytrých zařízení, kdy je možné monitorovat prostředí kolem nás a odesílat data na velké vzdálenosti, zde vyvstává s tímto technologickým pokrokem i řada problémů.

Za jeden z daných problémů lze označit i nesourodost a omezenou kompatibilitu napříč jednotlivými druhy či skupinami těchto chytrých zařízení – tedy zkrátka a jednoduše nízkou provázanost aktuálně využívaných řešení. Jednotliví výrobci či distributoři využívají rozdílné prostředky - hardware, software, typy komunikačních protokolů či datové struktury. Tato skutečnost ve výsledku ovlivňuje především práci se samotnými daty, která jsou skrze tato zařízení shromažďována.

Katedra informačních technologií Provozně ekonomické fakulty ČZU v Praze ve spolupráci s dalšími katedrami této univerzity vyvíjí v době psaní této práce projekt datové platformy, která slouží pro ukládání a management heterogenních dat z různých projektů a oblastí.

Tento projekt zahrnuje vše od sběru a zpracování dat až po jejich analýzu a prezentaci. Tím pod tento projekt spadá i nutnost řešení transformace různorodých datových struktur přijímaných z jednotlivých zařízení do sjednocené a univerzální podoby, čímž se tato diplomová práce zabývá.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je vytvoření základní struktury a funkcionality vstupního uzlu datové platformy, skrze který by byla stahována, tříděna a transformována data, která jsou čerpána z různých sběrných míst. Prvním dílčím cílem je návrh a implementace back-end části aplikace skrze platformu NodeRED tak, aby bylo možné zpracovávat data alespoň z jednoho vybraného sběrného místa. Druhým dílčím cílem je návrh a implementace front-end části aplikace, skrze kterou má být back-end část aplikace ovládána a monitorována.

2.2 Metodika

Diplomová práce sestávala ze dvou částí – teoretických východisek a vlastního řešení.

Metodika zpracování teoretické části vycházela ze studia odborných informačních zdrojů. V případě této práce se jednalo především o oficiální dokumentace vybraných technologií, které jsou většinou dostupné výhradně v elektronické podobě, především kvůli jejich častým aktualizacím. Na základě syntézy zjištěných poznatků byla formulována východiska pro zpracování praktické části.

Praktická část práce spočívala ve vytvoření webové služby, respektive back-end a front-end části aplikace, která má sloužit jako vstupní uzel pro data z IoT zařízení, která jsou dále odesílána do datové platformy univerzity. Vypracování vlastního řešení bylo uzpůsobeno požadavkům a limitacím, které byly stanoveny před zpracováním praktické části práce.

Nejprve byla provedena analýza datových vstupů a požadovaných datových transformací, na základě kterých byla definována základní struktura aplikace, která byla rozvržena do tří částí – funkcionální, transakční a prezentační. Každá z těchto částí byla poté postupně realizována skrze dostupné, v teoretické části práce popsané prostředky.

Pro back-end část aplikace bylo nutno použít platformu NodeRED. Platforma NodeRED a její rozšíření byly následně využity i pro realizaci front-end části aplikace, která byla realizována především skrze Javascript framework Vue.js.

V průběhu zpracovávání praktické části práce byly průběžně realizovány konzultace s vybranými osobami podílejícími se na projektu datové platformy a to především z důvodu kontroly naplnění požadavků a realizace konkrétních funkcionalit, kterými měla aplikace disponovat.

3 Teoretická východiska

Tato část práce zachycuje vybraný souhrn teoretických poznatků nabytých před započetím vypracovávání této práce či poznatků nabytých v průběhu jejího vypracovávání. Na následující text lze pohlížet jako na text obsahující informace, které jsou stěžejní pro pochopení dalších částí práce.

3.1 Virtualizace

V kontextu této práce lze virtualizaci definovat jako vytvoření abstraktní vrstvy nad počítačovým hardwarem. Tato vrstva umožňuje využívat hardwarové prvky jednoho zařízení pro realizaci vícero virtuálních zařízení, nebo-li pro chod virtuálních strojů. Každý z těchto virtuálních strojů se pak chová jako nezávislé zařízení, i když běží pouze na části hardwaru původního zařízení, které je k dispozici [1].

Pro virtualizaci lze použít mnoho různých, specificky zaměřených nástrojů. Jedním z virtualizačních nástrojů, který je využíván při vývoji softwaru a který je použit i pro dosažení cílů této práce, je Docker Engine.

3.2 Docker Engine – vývoj kontejnerových aplikací

Docker Engine představuje sadu nástrojů, které, jak je zmíněno výše, využívají virtualizaci (konkrétně virtualizaci na úrovni operačního systému) k běhu a distribuci softwaru jakožto jednotlivých balíků či kontejnerů [2].

Jednotlivé kontejnery jsou ve výchozím stavu izolovány jeden od druhého. Kontejner jako takový pak sdružuje konkrétní software, knihovny či konfigurační soubory, které jakožto celek definují například právě konkrétní aplikaci. Tyto kontejnery lze mezi sebou propojovat či tvořit jejich seskupení. Zároveň je možné pro jednotlivé kontejnery zpřístupnit komunikaci mimo prostředí Docker Engine – například komunikaci s hostitelským prostředím, kde Docker Engine běží, či komunikaci mezi jednotlivými, hardwarově oddělenými zařízeními.

3.3 IoT – Internet věcí

Internet věcí, nebo-li anglicky Internet of Things (IoT), lze definovat jako síť zařízení (spotřebičů, senzorů, vysílačů a mnoha dalších), která lze napříč sítí propojit a docílit tak výměny dat. Do této oblasti ovšem nespadají pouze koncová zařízení, která data zaznamenávají. Pod pojem "Internet věcí" lze zařadit i veškerou infrastrukturu, která je využívána pro zpracování, uložení a interpretaci daných nasbíraných dat – tedy různé servery poskytující přístup k naměřeným datům skrze API (Application Programming Interface) či samotné datové sklady a databáze, kde mohou být data archivována [3].

3.4 NodeRED – vývoj softwaru z oblasti IoT

NodeRED lze popsat jako kompaktní, vysoce optimalizované běhové prostředí (server) postavené na technologii Node.js, které lze využít pro propojení jednotlivých zařízení (čidel, snímačů, kamer či jakýchkoliv jiných alternativ) a API (přístupových bodů úložišť dat či různých online služeb) pomocí standardních komunikačních protokolů (UDP, TCP-IP, http a jiné).

Základní instalace tohoto nástroje obsahuje editor dostupný skrze webový prohlížeč, který je stěžejní pro práci s tímto nástrojem. Logiku aplikace, kterou lze skrze editor sestavit, je možné exportovat či importovat skrze souborový formát JSON, tudíž není žádný problém s distribucí konkrétních implementací povelů napříč vícero zařízeními [4].

Základní stavební jednotkou NodeREDu je uzel, který lze popsat jako objekt reprezentující elementární povel či sadu povelů. Dané uzly lze řadit a shlukovat do takzvaných toků, pomocí kterých lze definovat rámec určité části aplikace a vhodně strukturalizovat projekt [5].

Kromě předdefinovaných uzlů, které plní specifické povely, lze v tomto nástroji vytvářet vlastní funkcionální uzly povelů skrze skriptovací jazyk Javascript.

3.4.1 Rozšíření a balíky NodeRED

Nástroj NodeRED v základu disponuje pouze nezbytným minimem uzlů (funkcí), které stačí na základní povely. Výchozí skupiny těchto uzlů jsou přiblíženy v seznamu níže:

- **Distribuční uzly** – zajišťují základní propojení, distribuci, zachycení či inicializaci zpráv v rámci prostředí NodeRED;
- **Funkcionální uzly** – zastupují základní funkcionální struktury (switch, trigger, delay atp);
- **Síťové uzly** – lze pomocí nich navázat mimo serverové spojení (http, udp, tcp, websocket atp);
- **Překladačské uzly** – umožňují převod dat mezi různými formáty (javascript objekty, JSON řetězce, csv, xml atp);
- **Uzly úložiště** – zprostředkovávají zápisy, čtení či monitoring souborů v úložišti;

Komplexnější aplikace se častokrát neobejdou bez instalace dodatečných rozšíření, která rozšiřují paletu uzlů o další funkce. V době psaní této práce je v oficiálním repozitáři balíků dostupných přes 225 000 rozšíření, které více či méně ovlivňují či rozšiřují funkcionalitu základního webového rozhraní NodeRED.

3.5 Webové aplikace

Za webovou aplikaci lze považovat webovou stránku, která umožňuje interaktivní a dynamické zobrazování či vkládání dat bez nutnosti stahování celého zobrazovaného obsahu při každém vzneseném požadavku. Z pohledu koncového uživatele připomíná více počítačový program než původní (statickou) podobu webových stránek. Jedná se prakticky o synonymum pro výraz „dynamická webová stránka“ [6].

3.5.1 Příprava vývoje webové aplikace

Při vývoji webové aplikace, respektive ještě před zahájením realizace, by mělo být definováno několik základních východisek:

Funkcionalita aplikace

Jaké funkce jsou od dané aplikace očekávány a jaký je tedy účel zamýšlené aplikace. S tím souvisí i definování faktu, zda-li je nutné danou aplikaci skutečně vyvíjet či jestli existují jiné alternativy. Kromě souhrnných funkcí jakýmiž mohou být například “transformace dat

x" či "zobrazení výstupu y" je vhodné definovat i dílčí funkce aplikace, například zda-li bude aplikace vyžadovat přihlášení k účtu, jak bude probíhat ověřování požadavků atp.

Data a práce s nimi

S jakými daty bude aplikace pracovat, odkud budou data čerpána, kam budou ukládána a co se s nimi bude v aplikaci dít. Jaká data budou uživateli poskytnuta a jaká data bude naopak uživatel generovat. Následně by mělo proběhnout rozvržení datových toků nejen z pohledu aplikace vůči vnějšímu prostředí, nýbrž i z pohledu jednotlivých částí aplikace, aby bylo jasné, jak a kam budou data uvnitř aplikace proudit.

Struktura aplikace

Jak bude aplikace rozvržena, jaké technologie a jazyky budou využity k realizaci, či jak bude aplikace prezentována koncovému uživateli. Do této oblasti může spadat vše od strohého slovního popisu aplikace až po komplexní strukturální diagramy či návrhy "Use case" scénářů [7].

3.5.2 Front-end a back-end webové aplikace

Za front-end webové aplikace lze zpravidla považovat tu část aplikace, která slouží k vykreslení požadovaného obsahu webové stránky. V této části je definována struktura zobrazovaného obsahu, který koncový uživatel uvidí ve webovém prohlížeči, když s aplikací interaguje.

Back-end webové aplikace je naopak ta část, ve které probíhá zpracování příchozích požadavků, distribuce či ukládání dat a jiné akce, které jsou při interakci s aplikací koncovému uživateli skryty.

3.6 HTML a kaskádové styly

HTML, nebo-li Hypertext Markup Language, je standardizovaný značkovací jazyk, který slouží pro zobrazování webových stránek a jejich obsahu (textů, obrázků, odkazů a dalších elementů). Jednotlivé značky HTML definují prvky dokumentu a označují tak i jejich účel (nadpisy, odstavce, tabulky a jiné) [8].

Kaskádové styly slouží k definování vzhledu webových stránek – lze pomocí nich definovat, jak bude dokument HTML vykreslen ve webovém prohlížeči. Styly lze definovat jak pro samostatné značky, tak i pro konkrétní identifikátory či naopak vlastní skupinu elementů pomocí tříd [9].

3.7 Skriptovací jazyk Javascript

Javascript je programovací jazyk, který slouží pro realizaci funkcionality webových stránek. Lze pomocí něj definovat instrukce, které se mají provést při určitých událostech - a to jak na front-endu, tak back-endu aplikace.

3.7.1 Osvědčené postupy psaní zdrojového kódu

Při psaní zdrojového kódu by měla být brána v potaz některá doporučení a postupy, které mají za následek lepší čitelnost a znovupoužitelnost daného kódu. V rámci zpracovávání praktické části práce bylo dbáno především na stupeň zanořování instrukcí, pojmenování proměnných či metod a dostatečnou modularitu zdrojového kódu.

Zanořování instrukcí

Zanořováním instrukcí je myšleno psaní kódu do bloků zajišťujících podmíněné spouštění či opakované iterativní provádění instrukcí (bloky “if-else”, “for-each”, “for” a podobné alternativy). Ve chvíli, kdy jsou v rámci jednoho skriptu či metody zanořeny do sebe více než 3 takové bloky kódu, kód se stává obtížně čitelným a složitým na pochopení. Proto je dobré problematické bloky kódu rozdělit do vícero metod tak, aby zanoření kódu nepřesahovalo třetí, v ojedinělých případech čtvrtou úroveň zanoření. Jinými slovy, pokud vývojář dojde do stavu, kdy se v jedné metodě dostane ke třetí či čtvrté úrovni zanoření kódu, měl by to být signál k přepsání dané metody a rozdělení její funkcionality do vícero menších metod [10].

Pojmenování proměnných a metod

Každá deklarovaná metoda či proměnná by měla být vhodně pojmenována tak, aby z daného názvu bylo alespoň částečně zřejmé či odvoditelné, jaký účel daná metoda či proměnná reprezentuje. Dodržování této metodiky opět značně přispívá k lepší čitelnosti a přehlednosti výsledného zdrojového kódu [11].

Modulární kód

Modularita kódu souvisí s výše zmíněným zanořováním instrukcí. Cílem by mělo být navrhnout a sepsat metody či bloky kódu tak, aby byly znovupoužitelné i v jiných částech projektu. Pokud se někde nachází duplicitní sada instrukcí, je vhodné uvažovat o možnosti danou sadu instrukcí vyjmout z aktuálního řešení a logiku implementovat jako samostatnou metodu, kterou lze volat z vícero míst, kde je daná logika vyžadována [12].

3.7.2 Formát JSON

JSON, nebo-li Javascript Object Notation je textový formát, skrze který lze prezentovat strukturovaná data, která jsou založena na syntaxi objektů programovacího jazyka Javascript. Pro zjednodušení lze JSON formát popsat v praxi jako textový řetězec s jasně definovanou strukturou, díky čemuž může být jeho obsah opět převeden do podoby Javascript objektu. V souvislosti s vývojem webových aplikací se používá především pro přenos dat – například při výměně dat mezi jednotlivými aplikačními rozhraními (API) či mezi jednotlivými vrstvami konkrétního softwaru (back-end a front-end) [13].

3.7.3 Javascript frameworky

V průběhu existence jazyka Javascript byly postupně vytvořeny různorodé nadstavby, nebo-li frameworky, které se liší jak ve funkčnosti, tak i mnohdy v syntaktických pravidlech a v celém procesu vývoje aplikací. Tyto nadstavby vychází z open-source, multiplatformního a asynchronního běhového prostředí nazývaného Node.js, které je schopné spouštět kód napsaný v jazyce Javascript [14].

Tyto frameworky značně zrychlují a ulehčují vývoj moderních webových aplikací, především díky řadě předdefinovaných funkcí, které lze dále rozšiřovat pomocí konkrétních knihoven. Ve výsledku tak vývojář webové aplikace nemusí řešit na elementární úrovni například vykreslovací a obnovovací cyklus aplikace, udržování stavu jednotlivých elementů či ukládání a distribuci dat.

Mezi nejrozšířenější, aktuálně používané frameworky se řadí například React.js, Next.js či Vue.js, nicméně k aktuálnímu dni, kdy je tato práce sepisována, existují desítky, ne-li stovky různých frameworků. Pro realizaci praktické části této práce byl vybrán framework Vue.js

a to sice na základě osobní preference autora této práce a na základě poznatků nabytých v průběhu analýzy a příprav vývoje aplikace.

3.8 Framework Vue.js

V rámci zpracování praktické části práce byl využit framework Vue.js ve verzi 2. Vue.js je rychlý a kompaktní framework, který je vhodný nejen pro malé a jednoduché aplikace, ale i pro komplexní projekty. Základním stavebním kamenem jsou takzvané „single-file" komponenty, nebo-li komponenty, které jsou reprezentovány jedním souborem specifického formátu [15].

Tyto komponenty umožňují díky své modularitě izolovaný vývoj různých částí aplikace – tedy jednotlivé části či funkce aplikace lze dělit do jednotlivých komponent, které jsou na sobě navzájem nezávislé a lze je tak použít kdekoliv v dané aplikaci dle potřeby, pokud to jejich struktura umožňuje.

V základní podobě jsou tyto komponenty reaktivní, tedy při běhu aplikace si udržují svůj vlastní stav (data). Vue.js ovšem podporuje i principy funkcionálního programování a lze tak vytvářet i funkcionální komponenty, které slouží čistě k vykreslení obsahu dle poskytnutých dat a žádný vlastní stav (data) si neudržují [16].

Vue.js komponenty v základu obsahují zpravidla 3 hlavní sekce – skriptovací sekci pro kód v jazyce Javascript rozšířený o vlastní syntaxi Node.js a Vue.js, sekci pro definování obsahu skrze obohacené HTML a sekci pro zápis kaskádových stylů, které budou v rámci daného komponentu použity.

3.8.1 Skriptovací sekce Vue.js komponenty

Skriptovací část Vue.js komponenty se dále dělí na další části, v nichž je nutné dodržovat konkrétní syntaxi zdrojového kódu. Níže jsou stručně popsané základní součásti skriptovací sekce, kterými reaktivní Vue.js komponenta standardně disponuje.

Sekce „properties" či „props"

Slouží pro definování jednotlivých vlastností, které může komponenta přijmout z hierarchicky výše postavené komponenty a následně je využít.

Sekce „data“

V této sekci je nutné definovat veškeré datové proměnné, které mají být napříč komponentou využity a které nejsou definované ve výše zmíněné sekci „properties“.

Sekce „methods“

Jedná se o prostor, kde lze definovat jednotlivé metody, které má komponenta využívat, případně které má komponenta poskytovat hierarchicky podřízeným komponentám.

Kromě výše zmíněných podsekcí lze ve skriptovací sekci definovat řadu sekundárních bloků, jmenovitě například „mounted“, „watch“ či „computed“. Tyto sekce opět slouží pro definování určité logiky (metod) komponenty, která se má ovšem provádět za specifických podmínek či v určité části životního cyklu komponenty [17].

3.8.2 Sdílení vlastností (Properties)

Sdílení vlastností mezi jednotlivými komponentami lze popsat na následujícím příkladu. V první komponentě je v sekci data definována proměnná obsahující seznam jmen.

```
<script>
export default {
   data: function() { return {
      names : ['Petr', 'Adam', 'Eva', 'Jana'],
   }},
}
</script>
```

Zdrojový kód 1- příklad definování datové proměnné ve Vue.js komponentě; Zdroj: vlastní

Tento seznam jmen má být sdílen do hierarchicky podřazené komponenty. Jak bylo zmíněno výše, každá z komponent, která má přijímat vnější vlastnosti, musí mít v sekci „properties“ danou skutečnost deklarovanou, viz zdrojový kód níže.

```
<script>
export default {
  props: {
    inputNames: {type: Array, default: []},
  },
}
</script>
```

Zdrojový kód 2 - příklad deklarace možnosti přijetí datových proměnných ve Vue.js komponentě; Zdroj: vlastní

Následně lze v hierarchicky nadřazené komponentě definovat odkaz v bloku kódu, skrze který se v sekci pro vykreslení HTML obsahu definuje vykreslení hierarchicky podřazené komponenty. Tím bude obsah proměnné v nadřazené komponentě propagován jakožto vlastnost do podřazené proměnné [18].

```
<hierarchicky-podrazena-komponenta
   :input-names="names"
></hierarchicky-podrazena-komponenta>
```

Zdrojový kód 3 - definování odkazu na proměnnou mezi Vue.js komponentami, Zdroj: vlastní

3.8.3 Odkazování na metody

Za předpokladu, že pro správu stavu front-endu aplikace není využita externí knihovna, může být změna hodnoty datové proměnné v rámci Vue.js zdrojového kódu prováděna pouze v té komponentě, ve které byla daná datová proměnná deklarována. Pokud jsou tedy data určité datové proměnné sdílena skrze postup, který byl popsán v předešlé kapitole, není pak správné (a možné) provádět změny hodnot původní datové proměnné skrze hierarchicky podřazenou komponentu, ve které máme na původní proměnnou k dispozici odkaz. Dané odkazy slouží především k propagaci dat směrem „dolů“, nikoliv k umožnění provádění akcí směrem „nahoru“. Tento problém lze efektivně řešit odkazováním na metody.

Správným postupem je v hierarchicky podřazené komponentě zavolat odkaz na metodu, která se nachází v hierarchicky nadřazené komponentě. V rámci vstupních parametrů dané metody lze vhodně propagovat dodatečná data zpět z podřazené komponenty do nadřazené komponenty, kde je následně provedena požadovaná instrukce.

Nejprve je nutné mít v hierarchicky nadřazené komponentě definovanou metodu.

```
methods: {
   writeName(index) {
      console.log("Name is:" + this.names[index]);
    },
},
```

Zdrojový kód 4 - příklad deklarace metody ve Vue.js komponentě; Zdroj: vlastní

Aby bylo možné se na danou metodu odkázat v hierarchicky podřazené komponentě, musí se opět definovat odkaz v bloku kódu, kde se definuje vykreslení podřazené komponenty - velmi podobně jako u definování odkazu pro vlastnost. Jediným rozdílem v tomto kroku je syntaxe, kdy se odkaz na metodu uvozuje znakem „@“ a nikoliv „:“, viz zdrojový kód níže.

```
<hierarchicky-podrazena-komponenta
   @write-name="writeName"
   :input-names="names"
></hierarchicky-podrazena-komponenta >
```

Zdrojový kód 5 - deklarace podřazené komponenty rozšířena o odkaz na metodu; Zdroj: vlastní

Následně lze danou metodu nadřazeného komponentu zavolat z hierarchicky podřazeného komponentu pomocí vestavěné funkce „emit()“ [19].

```
callWriteName: function(nameIndex) {
      this.$emit("write-name", nameIndex);
},
```

Zdrojový kód 6 - příklad odkázání na metodu v hierarchicky nadřazené komponentě; Zdroj: vlastní

3.8.4 Kondicionální a iterativní vykreslování obsahu

Vue.js v základu poskytuje možnost kondicionálně a iterativně vykreslovat obsah přímo skrze sekci komponenty určenou pro zápis obohaceného HTML kódu. Do HTML kódu stačí přidat specifickou Vue.js direktivu. V případě kondicionálního vykreslování se jedná například o direktivy „v-if“ či „v-else“ viz ukázka kódu níže [20].

```
<div v-if="timestampList.length < 10" class="shortList">
      <p> Tento text se vykreslí, pokud délka pole je menší než 10. </p>
</div>
<div v-else class="longList">
      <p> Tento text se vykreslí, pokud není splněna podmínka výše. </p>
</div>
```

Zdrojový kód 7 - ukázka kondicionálního vykreslování obsahu pomocí Javascript frameworku Vue.js

V rámci iterativního generování obsahu lze použít direktivu „v-for“, skrze kterou lze definovat iterativní podmínku. Iterativní podmínka může obsahovat jednoduché opakování na základě poskytnutého čísla nebo pokročilé opakování na základě výčtu prvků

v předdefinovaném poli, což umožňuje pro konkrétní prvek v iteraci generovat obsah závislý na konkrétních vstupech [21].

```
<div v-for="n in 10">
      <p> Aktuální číslo prvku je: {{ n }} </p>
</div>

<div v-for="user in users">
      <p> Jméno: {{ user.name }} </p>
      <p> E-mail: {{ user.mail }} </p>
</div>
```

Zdrojový kód 8 - ukázka možností iterativního vykreslování obsahu ve Vue.js komponentě; Zdroj: vlastní

3.9 Mikroslužby v oblasti vývoje softwaru

Pojem „mikroslužby“ lze definovat jako určitý způsob strukturalizace projektu či aplikace – tedy jako určitý přístup k softwarové architektuře. Aplikace je tvořena kolekcí malých autonomních služeb, přičemž každá ze služeb je samostatná a implementující pouze jednu dílčí činnost či vlastnost [22].

Tento styl architektury lze v prostředí NodeRED efektivně aplikovat na několika různých úrovních agregace, zejména pomocí shlukování vybraných NodeRED uzlů či naopak rozdělování logiky aplikace do vícero datových toků.

3.10 Časové značky

V rámci praktické části práce jsou zmiňovány takzvané “časové značky”. V souvislosti s tématem práce se jedná o data zachycující konkrétní časový okamžik v podobě deseticiferného či třínácticiferného čísla, které bývá standardně označováno jako UNIXová časová značka. Pakliže se jedná o deseticifernou variantu, dané číslo reprezentuje počet sekund, které uplynuly od 1. ledna roku 1970. V případě třínácticiferné varianty čísla je reprezentována obdobná informace, tentokrát ale měřená v milisekundách [23].

UNIXová časová značka je nedílnou součástí jakéhokoliv softwaru, který pracuje s časovými údaji a potřebuje být odolný například vůči faktoru časových pásem, různorodým formátům zápisu data či provádí nad časovými záznamy další výpočty. Z jakékoliv časové značky lze zpětně sestavit datum a čas ve specifickém, člověkem lépe čitelném formátu.

3.11 BPMN

BPMN nebo-li Business Process Model and Notation je standardizovaný soubor pravidel a grafických značek, pomocí kterých lze modelovat systémové, podnikové či jakékoliv jiné procesy za použití procesních diagramů [24].

V praktické části této práce jsou k nalezení diagramy, které se touto notací řídí a které jsou v práci zahrnuty jakožto doplňkový prostředek prezentace řešení dílčích cílů.

4 Vlastní práce

Praktická část práce se skládala z definování a analýzy požadavků, výběru vhodných technologií a realizace řešení. Realizace řešení zahrnovala dílčí analýzy a opakované ověřování dosažení funkčních kritérií. Tyto činnosti byly na sobě vzájemně závislé. Při sepisování této části práce byl vzhledem k jejímu zaměření kladen větší důraz na technické hledisko a výslednou realizaci řešení.

4.1 Vývojová prostředí

K dosažení cílů bylo využito především dvou vývojových prostředí – webového rozhraní NodeRED skrze virtualizační nástroj Docker Engine a pomocného vývojového prostředí Visual Studio Code s dodatečnými doplňky nezbytnými pro výkon činnosti.

Webové rozhraní NodeRED bylo využito pro sestrojení veškeré logiky back-end části webové aplikace (stahování, třídění, transformaci a ukládání dat). Vývojové prostředí Visual Studio Code bylo využito pro realizaci front-end části webové aplikace, tedy pro vývoj plnohodnotné prezentace stavu a vybraných dat uživateli.

4.2 Požadavky a limitace

Vzhledem k tomu, že tato práce byla realizována v rámci projektu univerzitní datové platformy, byla realizace cílů práce ovlivněna níže uvedenými požadavky a limitacemi, které bylo nutné respektovat.

4.2.1 Zvolené běhové prostředí

Logika back-end části aplikace musela být realizována v nástroji NodeRED, který se nachází v podobě kontejneru virtualizačního nástroje Docker na jednom z produkčních serverů ČZU.

Pro vývoj proto bylo vytvořeno podobné běhové prostředí na vývojářském serveru – v tomto případě lokálním zařízení autora této práce. Vzhledem k tomu, že některé funkce (například příjem dat skrze User Datagram Protocol) nebyly na vývojářském serveru dostupné ve stejné míře jako na produkčním serveru, byl zde vývoj realizován na základě poskytnutých testovacích vzorků dat.

4.2.2 Skriptovací jazyk Javascript

Při realizaci back-end části aplikace bylo nutné použít jazyk Javascript, jelikož pokročilé funkcionální uzly v NodeRED prostředí podporují výhradně tento programovací jazyk.

4.2.3 Front-end aplikace

Front-end část aplikace měla být pokud možno realizována jakožto nadstavba na běhovém prostředí NodeRED, nikoliv jako samostatná aplikace.

Uživateli by skrze tento front-end měla být dostupná základní data ohledně synchronizace jednotlivých vstupních bodů. V případě, že vstupní bod podporuje ruční stahování dat, mělo by být možné vytvořit požadavek pro ruční synchronizaci dat v konkrétním časovém úseku, který si uživatel definuje.

Taktéž by mělo být možné skrze front-end aplikace vyvolat některé z řídících akcí aplikace, jakýmiž mohou být například výpis či reset dat mezipaměti aplikace (například výpis nedávno zpracovaných časových značek zařízení).

Koncovým uživatelem tohoto rozhraní nemá být veřejný činitel, nýbrž pověřený člověk spolupracující na tomto projektu. Z daného důvodu nejsou kladeny žádné konkrétní požadavky na vzhled aplikace či rozložení jejího obsahu – tato problematika zůstává čistě v režii autora práce.

4.2.4 Podoba výstupních dat

Stěžejním požadavkem, kvůli kterému celá aplikace vznikla, bylo transformovat podobu příchozích dat do sjednocené výstupní podoby, ve které by data byla odeslána (a následně uložena) do datové platformy univerzity.

Výstupní data měla být distribuována ve formě objektu s následujícími atributy:

- **Source** – zdroj dat, nebo-li systémové označení původu záznamu. Měla by být dodržována konvence pojmenování ve formátu “vstupní_bod.zařízení” (příklad: envirosystem.3030 – data pochází ze vstupního bodu nesoucí název “envirosystem”, jedná se konkrétně o zařízení “3030”).

- **Headers** – kopie hlavičky záznamu v nepozměněné podobě. Může obsahovat energetický stav zařízení, časovou značku, označení zařízení a jiné řídící atributy.
- **Timestamp** – časová značka záznamu ve formátu UNIXové časové značky. Udává informaci o tom, k jakému časovému okamžiku jsou přidružená data vázána (kdy byla data pořízena).
- **Values** – data záznamu. Každý dílčí záznam je zde reprezentován ve formě objektu, který má tvar {key: x , value: y}, kde "x" vyjadřuje systémové označení zasílané hodnoty a "y" označuje samotnou hodnotu.
- **RawData** – data záznamu v nepozměněné podobě (v podobě, ve které byla ze vstupního bodu přijata).

4.2.5 Modularita a znovupoužitelnost aplikace

Základní struktura aplikace měla být vytvořena tak, aby do ní bylo možné v budoucnu přidávat nové vstupní body, ze kterých budou čerpána data z nových zařízení. Byl stanoven předpoklad, že se aplikace bude nadále udržovat a rozšiřovat o podporu nových vstupních bodů.

S tím souvisí i struktura a celkové provedení front-end části aplikace, která by měla být schopna patřičná data nových vstupních bodů či jednotlivých zařízení zobrazovat bez nutnosti složitých úprav – zdrojový kód front-end aplikace by měl data načítat dynamicky na základě poskytnuté konfigurace.

4.3 Výsledek analýzy požadavků a struktura aplikace

Na základě teoretických východisek, konzultací s odpovědnými osobami podílejícími se na vývoji datové platformy univerzity a výše zmíněných požadavků a limitací byla definována základní distribuce dat. Od distribuce dat se odvíjelo navržení struktury aplikace, která byla rozdělena na tři stěžejní části – funkcionální, transakční a prezentační.

4.3.1 Data aplikace

Data aplikace byla rozdělena do čtyř skupin:

- Systémové zprávy
- Vstupní data uživatele
- Vstupní data zařízení
- Výstupní data systému/ zařízení

Systémové zprávy

Systémové zprávy jsou ve své podstatě Javascript objekty, které jsou distribuovány skrze jednotlivé NodeRED uzly. Mohou reprezentovat strukturalizované uživatelské požadavky či odpovědi na ně. Stejně tak mohou obsahovat data zařízení v jakékoliv fázi jejich transformace. Systémová zpráva se vyznačuje především tím, že vždy obsahuje identifikátor a zdroj původu.

Vstupní data uživatele

Jedná se o data vygenerovaná lidským faktorem skrze webové rozhraní této aplikace. Tato data je před zpracováním nutno převést do podoby NodeRED zprávy, nebo-li "systémové zprávy", kde je uživatelský vstup vhodně reprezentován jednotlivými atributy.

Vstupní data zařízení

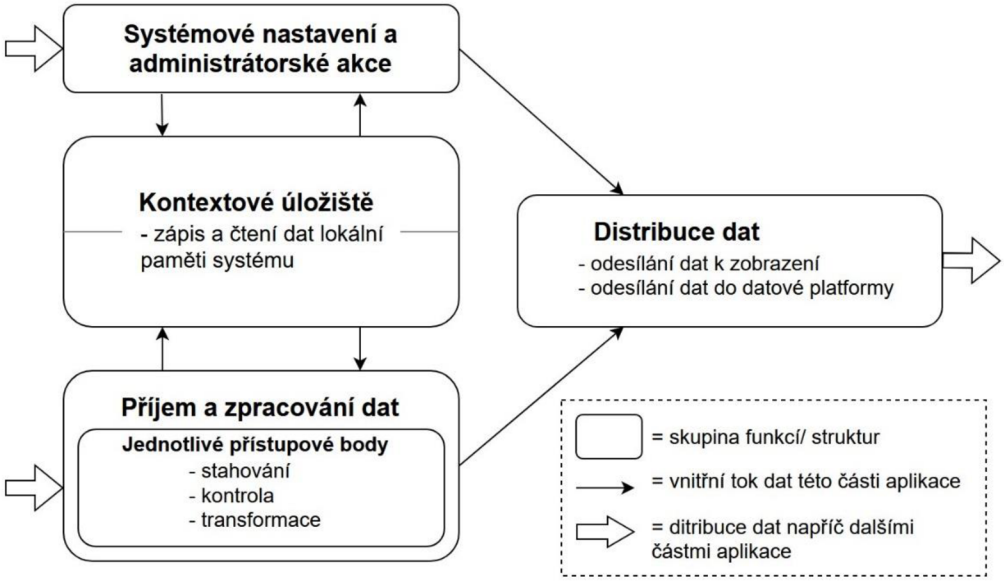
Data, která byla distribuována (stažena) do této aplikace a která reprezentují pořízený záznam zařízení. Tato data mají napříč jednotlivými vstupními body různorodou podobu. V některých případech se jedná o data ve formátu JSON, jindy nabývají podoby hexadecimálních textových řetězců či objektů, které mají různorodé struktury. Aby bylo možné s daty vhodně pracovat, je nutné zajistit jejich převod do takové podoby, ve které bude možné je jednoznačně identifikovat a dále transformovat.

Výstupní data systému/ zařízení

Výstupní data jsou součástí systémové zprávy, která je dle řídících atributů odeslána na výstup do datové platformy. Jedná se o objekty mající specifickou strukturu definovanou v kapitole *4.2.4 Podoba výstupních dat*.

4.3.2 Funkcionální část aplikace (back-end)

První a zároveň nejdůležitější částí je back-end struktura obsahující logiku práce s daty – stahování, třídění a transformaci dat. Na dále přiloženém obrázku lze vidět konkrétní rozvržení této části aplikace.



Obrázek 1 - znázornění struktury back-end části aplikace (1); Zdroj: vlastní

Skupina funkcí „Systémové nastavení a administrátorské akce“ (na obrázku výše) reprezentuje veškeré struktury, které nestandardně zasahují do běhu této části aplikace či modifikují základní data kontextového úložiště. Jedná se o funkce úvodního nastavení systému, funkce automaticky volané v průběhu procesu synchronizace či funkce představující ruční zásahy vyvolané lidským faktorem.

Kontextové úložiště představuje místní úložiště platformy NodeRED, které v případě této části aplikace slouží k perzistentnímu uchování nastavení systému a nedávno zpracovaných časových značek zařízení.

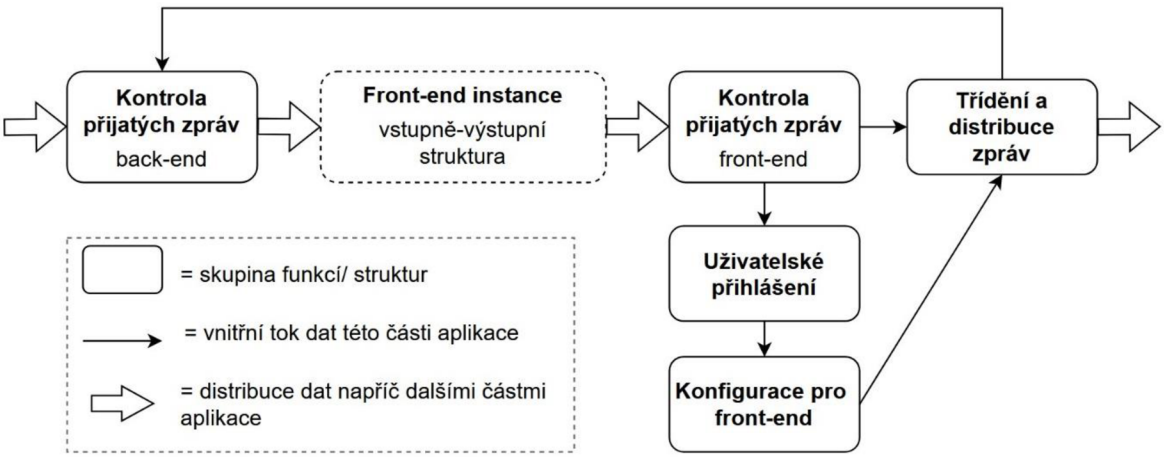
Funkcionální část aplikace je strukturována tak, aby součásti obsahující nastavení, distribuci požadavků a ukládání dat byly odděleny od implementace logiky jednotlivých vstupních bodů. Každý vstupní bod je zde reprezentován oddělenou skupinou funkcí, které realizují příjem, kontrolu a transformaci dat ve formě upravené na míru konkrétnímu vstupnímu bodu. V případě, že se zde nachází vícero vstupních bodů využívajících stejnou metodu příjmu dat, mohou být tyto vstupní body shlukovány do skupin.

Výsledkem je zachování základní funkcionality této části aplikace bez ohledu na to, kolik vstupních bodů je připojených. Každý nově přidaný vstupní bod ovšem musí splňovat určitá kritéria a způsob komunikace, aby mohl být do stávající implementace zapojen.

4.3.3 Transakční část aplikace (back-end)

Back-end struktura aplikace, kterou lze označit pojmem „middleware“. Jedná se o transakční vrstvu projektu, která umožňuje komunikaci mezi prezentační částí (front-endem) a funkcionální částí (zbytkem back-endu) aplikace. Taktéž se zde nachází konfigurační funkce, na základě kterých je (či naopak není) na front-endu aplikace obsah vykreslován.

Požadavky přicházející z front-endu aplikace jsou zde zachyceny a zpracovány. Pokud příchozí požadavek splňuje určitá kritéria, je distribuován zpět na front-end aplikace či do jednoho ze vstupů funkcionální části aplikace, aby na jeho základě byla navrácena specifická data či provedena konkrétní sada povelů.



Obrázek 2 - znázornění struktury back-end části aplikace (2); Zdroj: vlastní

4.3.4 Prezentační část aplikace (front-end)

Třetí část představuje webovou stránku, která vykresluje dostupná data ohledně aktuálního stavu aplikace. Zobrazovaný obsah je rozdělen do několika sekcí – vývojářské, obsahu úložiště a sekce synchronizace.

Vývojářská sekce obsahuje výpis přijatých a odeslaných zpráv, které byly mezi back-end a front-end částí aplikace vyměněny. Slouží především k odhalování chyb, pomoci při jejich opravě a kontrole stavu komunikace mezi prezentační a transakční částí aplikace.

Sekce s obsahem úložiště poskytuje výpis jednotlivých časových značek konkrétních synchronizovaných záznamů uložených v kontextovém úložišti aplikace, které jsou setříděné dle konkrétních zařízení či jejich skupin.

Sekce synchronizace obsahuje uživatelské rozhraní, skrze které lze zadat příkaz ruční synchronizace pro vstupní body, které takový typ synchronizace umožňují. Zároveň se zde nachází zjednodušený přehled aktuálního stavu automatické synchronizace, který je setříděn dle jednotlivých vstupních bodů či skupin zařízení.

4.3.5 Shrnutí výsledků analýzy

V rámci efektivního naložení s identifikovanými datovými zdroji se aplikace skládá celkem ze tří hlavních částí – funkcionální, transakční a prezentační. Funkcionální část může být v provozu nezávisle na ostatních částech aplikace, čímž lze zajistit základní automatizovanou funkčnost distribuce dat do datové platformy i v případě výskytu problému či odstávky některé ze zbylých dvou částí aplikace.

Ačkoliv jsou jednotlivé části aplikace vzájemně propojené, všechny tři části mohou v případě potřeby fungovat odděleně na různých hardwarových či softwarových prostředcích (mohou být rozděleny do vícero NodeRED instancí na různých výpočetních jednotkách).

4.4 Příprava vývojového prostředí

Před implementací jednotlivých částí aplikace bylo nutné vytvořit a nastavit vývojové prostředí tak, aby věrně napodobovalo prostředí produkčního serveru.

4.4.1 Instalace prostředí Docker engine

První krok spočíval v instalaci prostředí Docker na systému Windows. Prostředí Docker není v základu na systémech Windows podporováno, tudíž musel být operační systém rozšířen o doplněk WSL 2 (Windows Subsystem for Linux). Kromě běhového prostředí byl

nainstalován i program Docker Desktop, který dané prostředí rozšiřuje o grafickou reprezentaci jeho obsahu, díky čemuž není uživatel odkázán pouze na příkazový řádek.

4.4.2 Instalace prostředí NodeRED

V dalším kroku bylo nutné vytvořit Docker kontejner obsahující instalaci NodeRED. Toho lze docílit skrze příkazový řádek pomocí příkazu majícího následující strukturu:

```
docker run -it -p 1880:1880 -v node_red_storage:/data --name
nodered-1 nodered/node-red
```

Zdrojový kód 9 - ukázka zavedení NodeRED kontejneru v prostředí Docker Engine; zdroj: vlastní

Výše uvedený příkaz specifikuje:

- "**docker run -it**" – spuštění kontejneru a připojení okna terminálu (příkazové řádky)
- "**-p 1880:1880**" – vnitřní port 1880 Docker prostředí má být propojen s lokálním portem 1880 nadřazeného systému (obsah tohoto kontejneru bude dostupný na systémovém portu 1880)
- "**-v node_red_storage:/data**" – definování úložiště dat (pro adresář "/data" nového kontejneru se má připojit (vytvořit) Docker úložiště s názvem "node_red_storage")
- "**—name nodered-1**" – kontejner ponese název „nodered-1"
- "**nodered/node-red**" – pro vytvoření kontejneru bude použita čistá instance (snímek) prostředí NodeRED (pakliže se taková instance v lokálním Docker prostředí nenachází, bude stažena skrze online repozitář poslední stabilní verze)

4.4.3 Nastavení NodeRED

V čisté instanci NodeRED prostředí bylo potřeba provést dodatečné úpravy.

Stěžejní úprava se týkala kontextového úložiště. Kontextové úložiště v základu slouží pro uchovávání dat během doby, kdy je NodeRED server spuštěn. Toto nastavení se dá změnit skrze konfigurační soubor „settings.js" tak, aby byla data persistentní a nedocházelo k jejich ztrátě při vypnutí či restartu NodeRED serveru.

Toho bylo docíleno editací kategorie "contextStorage" konfiguračního souboru. Byla přidána nová instance úložiště dat s názvem "sync_api_flow", přičemž bylo specifikováno, aby se data ukládala do souborového systému lokálního NodeRED prostředí viz ukázka zdrojového kódu na další straně.

```
contextStorage: {
      default: { module: "memory" },
      sync_api_flow: { module: "localfilesystem" },
},
```

Zdrojový kód 10 - úprava konfiguračního souboru settings.js platformy NodeRED; Zdroj: vlastní

4.5 Implementace: Funkcionální část (back-end)

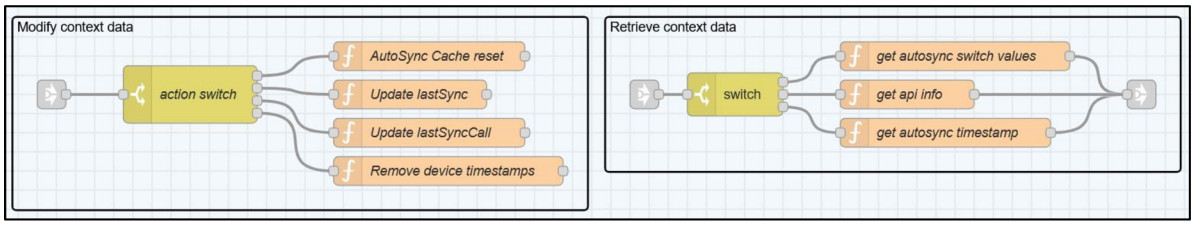
Jak již bylo zmíněno výše, funkcionální část aplikace je back-end struktura obsahující logiku práce s daty – stahování, třídění a transformaci dat. Zároveň se zde nachází struktury, které přistupují ke konfiguračním datům týkajících se procesu synchronizace.

Dle definovaných výsledků analýzy v kapitole *4.3 Výsledek analýzy požadavků a struktura aplikace* byla tato část aplikace koncipována tak, aby bylo možné přidávat či odebírat libovolné množství přístupových bodů bez nutnosti větších zásahů do aktuální implementace. Z daného důvodu byly řídící uzly (uzly zajišťující odesílání zpracovaných dat či uzly přistupující k datům v kontextovém úložišti) navrženy univerzálně – je jedno, odkud požadavek do takového uzlu přijde, pokud bude obsahovat potřebné vstupní hodnoty.

Níže jsou popsané hlavní řídící struktury této části aplikace. Zároveň s tím jsou dále zahrnuty i příklady zpracování konkrétních vstupních bodů a jejich napojení na řídící strukturu.

4.5.1 Řídící struktura: Kontextové úložiště

Tato sada NodeRED uzlů obsahuje funkcionalitu pro zapisování a čtení dat kontextového úložiště. Struktura je rozdělena na dvě části – zapisovací a čtecí. Obě části struktury obsahují jednotlivé, paralelně zapojené funkcionální uzly, přičemž každý z těchto uzlů je určen pro jinou sadu instrukcí, které lze v tomto případě označit za elementární prvky operující nezávisle na zbytku implementace – mají pevně definovaný požadovaný vstup i možnosti výstupu. V případě zapisovací části struktury se jedná například o instrukce pro vymazání konkrétních dat (vymazání časových značek konkrétního zařízení či časových značek celé sady zařízení).



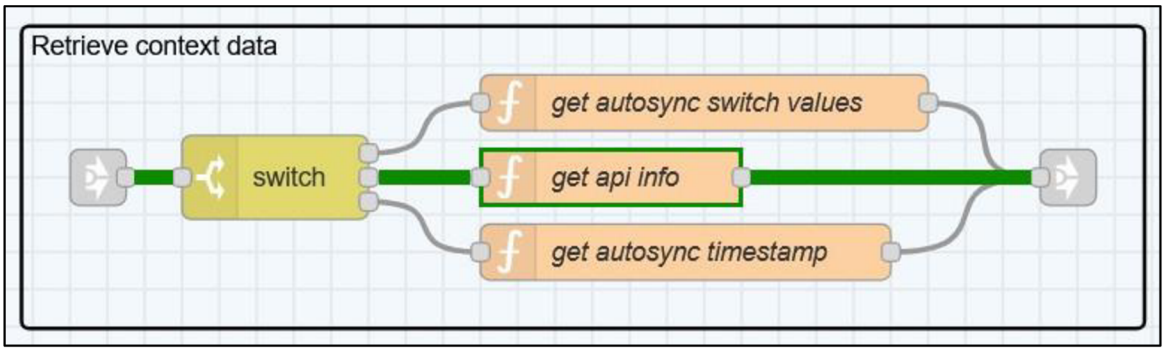
Obrázek 3 - náhled na řídící strukturu kontextového úložiště (NodeRED) ; Zdroj: vlastní

Příchozí zprávu může přijmout zapisovací i čtecí část struktury. Po příjmu zprávy je v obou případech pomocí rozhodovacího uzlu určeno, která ze sady instrukcí bude provedena. Rozhodování je realizováno na základě řídícího atributu zprávy, viz demonstrativní obrázek níže.

Obrázek 4 - příklad rozhodovacího uzlu v NodeRED (action switch); Zdroj: vlastní

Příklad užití:

Z front-endu aplikace byl odeslán automatický požadavek pro načtení dat konkrétního vstupního bodu (konkrétní sady zařízení). Daná zpráva je v transakční části aplikace přesměrována na vstup této struktury, konkrétně na vstup, který se nachází ve čtecí části. Na základě řídícího atributu je zpráva přesměrována do funkcionálního uzlu “get api info”, viz následující obrázek, kde je šíření zprávy vyznačeno zvýrazněnou čarou.



Obrázek 5 - příklad šíření zprávy; vyřízení požadavku pro načtení dat vstupního bodu; Zdroj: vlastní

V uzlu „get api info“ je po validaci vstupu zavolána funkce pro načtení dat definovaného vstupního bodu z kontextového úložiště a zpráva je poslána na výstup.

```
function apiDataCall() {
    let apiData = {
        dates:
            flow.get("dates_" + msg.targetAPI, "sync_api_flow"),
        lastSyncCall:
            flow.get("lastSyncCall", "sync_api_flow")[msg.targetAPI],
        lastSync:
            flow.get("lastSync", "sync_api_flow")[msg.targetAPI],
        syncTime: new Date().getTime()
    }
    return apiData;
}
```

Zdrojový kód 11 - funkce pro načtení dat přístupového bodu z kontextového úložiště NodeRED; Zdroj: vlastní

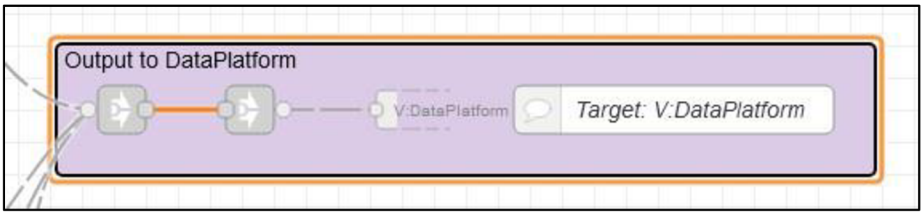
Jak lze vidět v ukázce zdrojového kódu výše, zpráva pro úspěšné vyřízení požadavku touto strukturou potřebuje mít kromě rozhodovací (řídící) proměnné definovanou i proměnnou, ve které je uložen název vstupního bodu, jehož data mají být navrácena v odpovědi.

4.5.2 Řídící struktura: Odesílání zpracovaných dat

Za zpracovaná, nebo-li výstupní data jsou v rámci této části aplikace považovány systémové zprávy, které prošly skrze určité uzly do některého z „koncových“ uzlů toku (uzlů nacházejících se na konci jednotlivých toků či koncích definovaných sekcí toku).

Výstupní data mohou být odeslána buď do datové platformy či do transakční části aplikace, kde jsou přesměrována na front-end aplikace. Pokud výstupní data nesplňují definovaná kritéria, mohou být nahrazena či zcela zahozena.

Výstup do datové platformy je realizován skrze jednoduchou strukturu, která plní funkci sjednocení výstupních bodů jednotlivých sběrných míst do jednoho výstupního uzlu, viz obrázek níže.



Obrázek 6 - sjednocení datových toků na 1 výstup do datové platformy; Zdroj: vlastní

Skrze výše zachycenou strukturu jsou zprávy přeposílány do jiné funkcionální struktury, jejíž podoba a obsah zde nejsou z bezpečnostních důvodů zachyceny. V dané struktuře je z příchozí zprávy vybrán konkrétní atribut obsahující data určená pro datovou platformu. Tato data jsou převedena do souborového formátu JSON a odeslána vybraným (zde blíže nespecifikovaným) komunikačním protokolem do datové platformy.

4.5.3 Obecný popis implementace vstupního bodu

Každý ze vstupních bodů je reprezentován třemi hlavními strukturami – přijímačem dat, procesorem dat a zachytávačem problémů.

Přijímač dat

Jedná se o skupinu uzlů obsahující funkcionalitu získávání dat z přístupového bodu. V rámci této práce lze rozlišovat dva různé druhy přijímačů dat – přijímač na bázi pasivního odposlechu (například UDP přijímač) a iniciátor – tedy přijímač, který komunikaci iniciuje (například na bázi http GET).

Procesor dat

V procesoru dat se nachází veškerá logika, skrze kterou jsou příchozí data transformována do podoby, ve které jsou odesílána do datové platformy. Standardně se jedná o funkcionální uzly zajišťující správné mapování hodnot do předem stanoveného formátu.

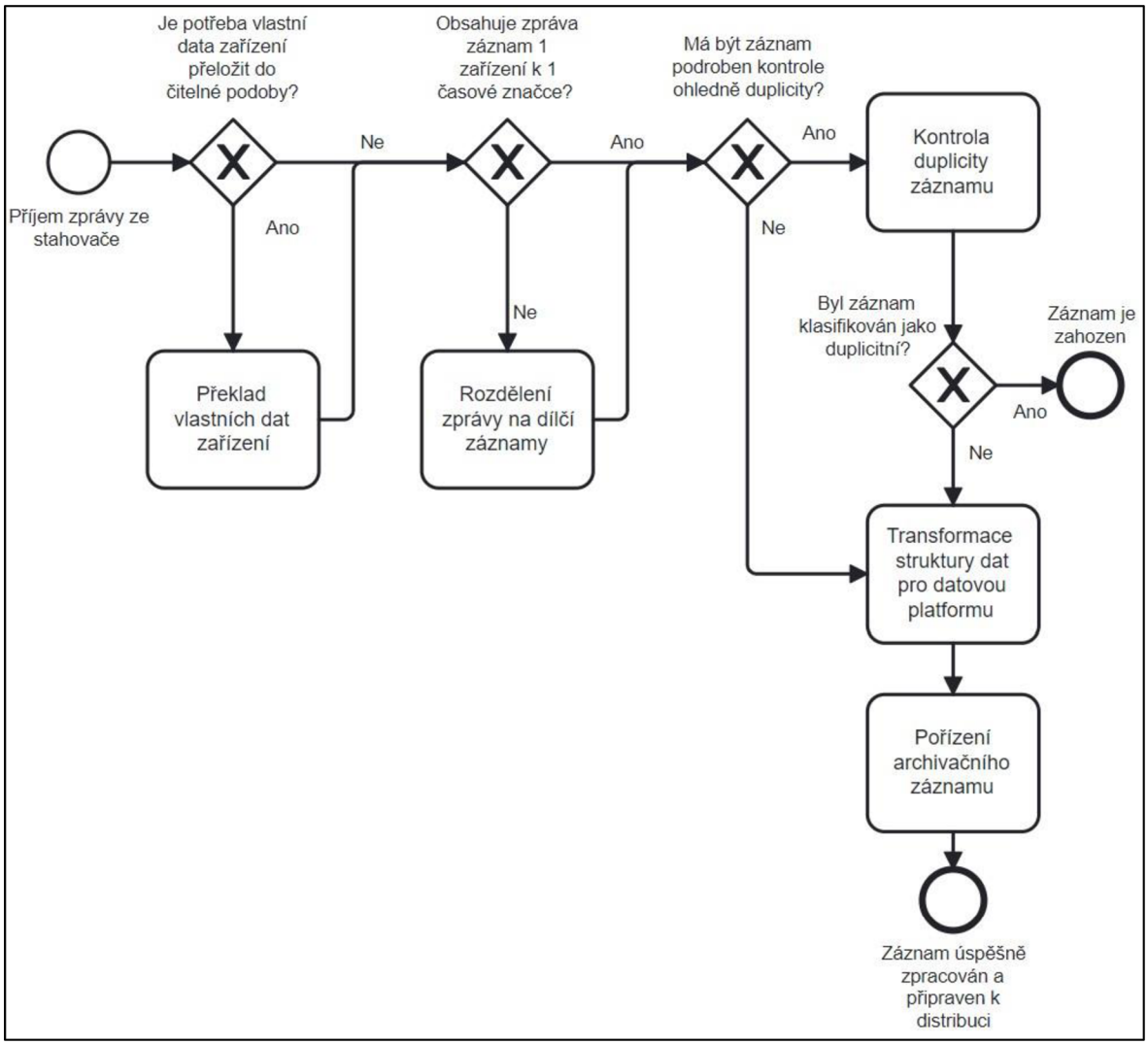
Ne vždy jsou data zařízení přijata ve vhodné podobě (formátu JSON). Ve většině případů jsou data přijata jako proud bitového pole (binárně) či v různých, pro člověka obtížně čitelných formátech (hexadecimální řetězec, řetězec Base64 a další varianty). Procesor dat proto ve většině případů obsahuje překladač, pomocí kterého jsou přijatá data převedena do vhodného formátu (Javascript objektu).

V případě, že skrze přijímač proudí data, která obsahují více než jeden záznam z jednoho zařízení na jednu příchozí zprávu (tedy data, která obsahují měření z několika zařízení či měření vztahující se k několika různým časovým značkám v jedné zprávě), je nutné v této struktuře definovat také uzel pro třídění a rozdělování příchozích dat.

Po převodu dat do vhodného formátu může také proběhnout kontrola příchozího záznamu. Pokud byl v nedávné minulosti zpracován záznam, který obsahoval stejný identifikátor zařízení a stejnou časovou značku jako záznam, který struktura dostane ke zpracování, je duplicitní záznam vyřazen a do datové platformy již znovu poslán není. Tato funkcionalita je standardně aplikována při automatické synchronizaci, kdy může být definováno zpětné překrytí intervalu z důvodu zajištění kontinuity dat. Pokud přijímač dat vstupního bodu spadá do skupiny, která komunikaci iniciuje a lze tedy kromě automatické synchronizace provést také manuální požadavek pro stažení dat, je tato kontrola při zpracovávání manuálního požadavku standardně neaktivní.

Po zpracování dat je skrze vhodný uzel pořízen archivační záznam. Ve výchozí konfiguraci je zapisováno datum zpracování záznamu, identifikátor zařízení, časová značka zpracovaného záznamu a informace o tom, zda byl záznam pořízen skrze manuální či automatický způsob synchronizace.

Na následujícím diagramu notace BPMN je zachycena obecná implementace výše nastíněného procesu.



Obrázek 7 - diagram BPMN zachycující proces zpracování stažených dat

Zachytávač problémů

Tato struktura obsahuje uzly, pomocí kterých lze zachytit a ošetřit problémové situace, které při stahování či zpracování dat mohou nastat. Standardně obsahuje uzel, který odposlouchává stav kritických funkcionálních či jiných uzlů, které se nachází ve strukturách stahování a zpracování dat. V případě, že v některém z odposlouchávaných uzlů dojde ke kritické chybě, je zpráva propagována do naslouchajícího uzlu, kde může být následně definován tok, který na danou chybu reaguje či alespoň chybový stav zaznamená do archivačního souboru.

4.5.4 Realizace řešení: Vstupní bod typu http GET

V této podkapitole je zahrnut příklad implementace struktury přístupového bodu, konkrétně struktura přijímající data skrze http požadavky.

Přijímač dat vstupního bodu typu http GET, jak bylo zmíněno výše, spadá do kategorie přijímačů, které komunikaci s vybraným aplikačním rozhraním (API) iniciují. V takovém případě je v rámci automatické synchronizace jednou za určitý čas automaticky vyvolán požadavek pro synchronizaci dat.

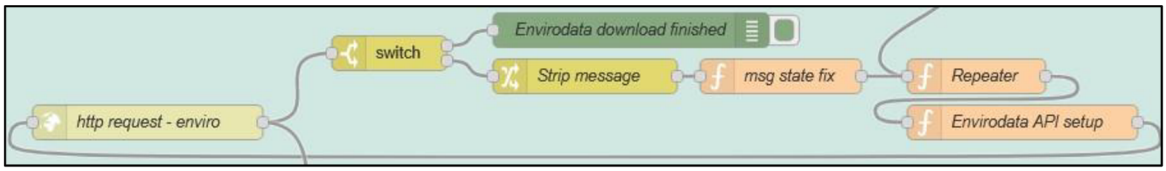
K tomu bylo využito NodeRED rozšíření "node-red-contrib-cron-plus", které do rozhraní NodeRED přispívá novým uzlem časovače, ve kterém lze zadat vlastní opakovací cykly dle konkrétních potřeb nejen skrze definici Cron příkazu, ale například i skrze solární události či sekvenci kalendářních dat [25]. Na přiloženém obrázku lze vidět konkrétní nastavení uzlu, které definuje, že se má uzel opakovaně spouštět každé dvě hodiny.

Obrázek 8 - nastavení uzlu iniciátoru automatické synchronizace (cron); Zdroj: vlastní

Zároveň s tím může uživatel aplikace provádět ruční synchronizaci, při které zvolí cílený časový interval, ze kterého chce data synchronizovat. V obou případech (automatické i ruční synchronizace) lze (teoreticky) zadat libovolně dlouhé časové intervaly, pro které se mají data synchronizovat.

Z daného důvodu jsou vstupní body tohoto typu osazeny strukturou opakovače, který má zamezit přetížení cíleného aplikačního rozhraní, odkud se mají data stahovat. Zároveň s tím je omezena velikost přijímaných zpráv, čímž je následně nepřímo optimalizováno zpracování jejich obsahu.

Výše zmíněná struktura opakovače je realizována skrze smyčku uzlů, viz obrázek na další straně.



Obrázek 9 - grafická reprezentace struktury opakovače v prostředí NodeRED; Zdroj: vlastní

Když do opakovače dorazí zpráva reprezentující požadavek synchronizace, je upraven zacílený časový interval tak, aby pokrýval nejvýše 1 hodinu zatím nezpracované části celkového zacíleného intervalu, viz následující ukázka zdrojového kódu.

```
const timerange_seconds_interval = 3600;
let targetedDate;

if(msg.repeated){
    targetedDate = new Date(msg.dateTo).getTime();
    targetedDate = targetedDate + 1000;
    msg.dateFrom = targetedDate;

    targetedDate = new Date(msg.dateTo).getTime();
    targetedDate = targetedDate + (timerange_seconds_interval * 1000);
    msg.dateTo = targetedDate;
} else {
    targetedDate = new Date(msg.dateFrom).getTime();
    targetedDate = targetedDate + (timerange_seconds_interval * 1000);
    msg.dateTo = targetedDate;
}
if (targetedDate >= new Date(msg.dateEnd).getTime()){
    msg.dateTo = msg.dateEnd;
    msg.finish = true;
    msg.lastFromGroup = true;
}
return msg;
```

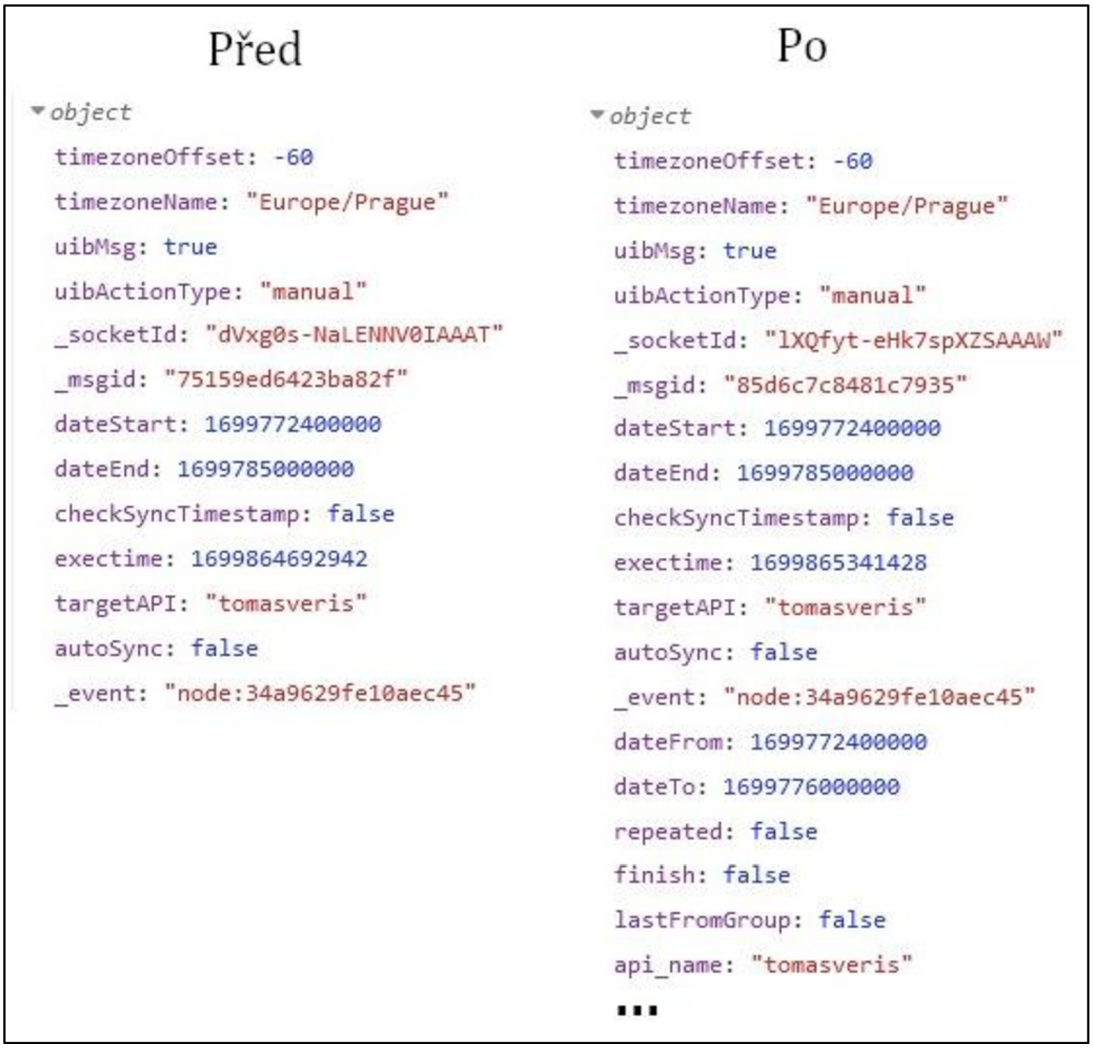
Zdrojový kód 12 - opakovač přijímače dat vstupního bodu typu http GET; Zdroj: vlastní

Příklad:

- Celkový interval 10:00 – 14:30, zpráva zatím nebyla opakována
  - Vstupní hodnoty: dateStart = 10:00; dateEnd = 14:30;
  - Výstupní hodnoty: dateStart = 10:00; dateEnd = 14:30; dateFrom = 10:00; dateTo = 11:00

Po každém průchodu nastavení dílčího intervalu synchronizace je zpráva zaslána do uzlu pro provedení http požadavku. Pokud je přijata odpověď, je příchozí zpráva odeslána ke zpracování a zároveň je opětovně poslána smyčkou zpět k opakovači, čímž dojde k její duplikaci. Pokud se jedná o zprávu, která navrátila data pro poslední z dílčích intervalů, je proces opakování volání http požadavku ukončen a duplicitní zpráva zahozena. Pokud zatím nebyl pokryt celý původně zacílený interval, je proces popsaný výše opakován s tím, že duplicitní zprávě jsou vymazána dříve stažená data.

Níže na obrázku je zachycena transformace zprávy. Vlevo se nachází zpráva před přijetím do opakovače, vpravo se nachází část zprávy, která prošla opakovačem a přijala data zařízení skrze http požadavek.



Obrázek 10 - porovnání obsahu NodeRED zprávy před a po iteraci stahování dat; Zdroj: vlastní

Příchozí zpráva v tomto případě obsahuje data zařízení ve formátu JSON, tudíž se v tomto konkrétním případě nemusí provádět překlad přijatých dat, který je v ostatních případech jednou ze stěžejních částí procesu zpracování dat.


```
payload: "[{"srcImsi":"901288001028672","name":"MP KL3","data":
{"version":1,"timestampUtc":"2023-11-12T07:00:03+00:00","batteryCpu":3606,"battery
Nb":3270,"humidity":87.02525,"temperature":0.47837067,"waterLevelMillimeters":167.
08052,"waterTemperature":3.77029,"resetCounter":20,"signal":0,"waterLevelWithOffse
t":92.08052,"waterFlow":0.003602031}},{"srcImsi":"901288910042969","name":"KL
III/5","data":
{"version":1,"timestampUtc":"2023-11-12T07:00:03+00:00","batteryCpu":3590,"battery
Nb":3584,"humidity":88.62287,"temperature":1.8509178,"waterLevelMillimeters":296.2
94,"waterTemperature":8.44975,"resetCounter":20,"signal":24,"waterLevelWithOffset"
:-158.706,"waterFlow":null}},{"srcImsi":"901288001028673","name":"MP KL1","data":
{"version":1,"timestampUtc":"2023-11-12T07:00:03+00:00","batteryCpu":3585,"battery
Nb":3541,"humidity":91.352715,"temperature":1.5197983,"waterLevelMillimeters":68.9
77325,"waterTemperature":1.4258239,"resetCounter":20,"signal":21,"waterLevelWithOf
fset":22.977325,"waterFlow":0.000112..."
```


Obrázek 11 - náhled na přijatá data zařízení ve formátu JSON; Zdroj: vlastní

Data zařízení tak mohou být hned po přijetí převedena do formátu Javascript objektu. Na obrázku níže lze vidět, že zpráva obsahuje data k mnoha různým zařízením, což znamená, že bude potřeba zprávu dále dělit.

Obrázek 12 - náhled na příchozí záznamy zařízení v rámci jedné NodeRED zprávy; Zdroj: vlastní

Dělení zpráv je prováděno skrze funkcionální uzel, kde je pro každý jednotlivý záznam zařízení z uzlu odeslána dílčí systémová zpráva. Pokud tento uzel přijme zprávu, která obsahuje data k poslednímu dílčímu intervalu aktuální synchronizace, označí poslední dílčí zprávu, kterou odesílá dále, jako konečnou. Tato dodatečná informace je pak využita například při navrácení odpovědi na prezentační část aplikace.

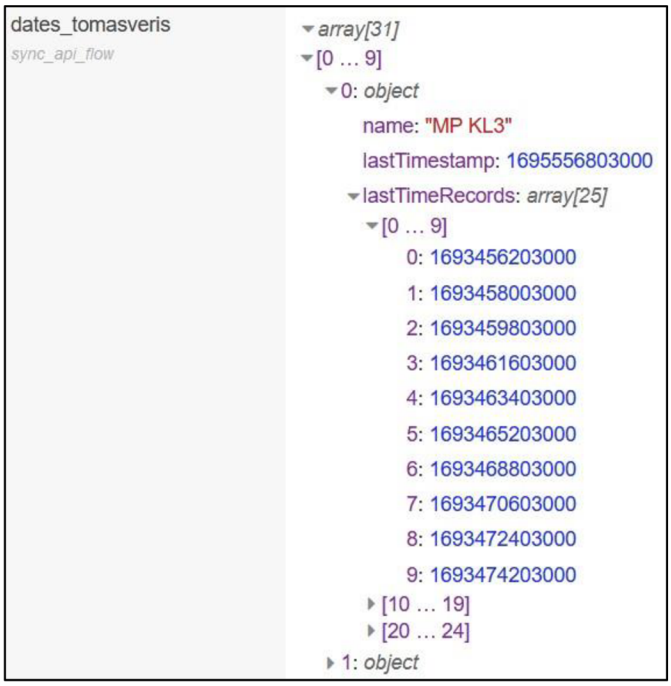
```
const numberOfRecords = msg.payload.length;
const payloadData = msg.payload;
msg.lastCutMsgFromGroup = false;

if(msg.lastFromGroup == true){
    for (let index = 0; index < numberOfRecords; index++) {
        msg.payload = payloadData[index]
        if (index == numberOfRecords - 1) {
            msg.lastCutMsgFromGroup = true;
        }
        node.send(msg);
    }
}
else {
    for (let index = 0; index < numberOfRecords; index++) {
        msg.payload = payloadData[index]
        node.send(msg);
    }
}
```

Zdrojový kód 13 - příklad dělení příchozí NodeRED zprávy na elementární záznamy zařízení; Zdroj: vlastní

Po výše uvedeném rozdělení zpráv na elementární záznamy (záznamy vztahující se k jednomu zařízení a jedné časové značce) může nastat kontrola duplicity záznamu, viz kapitola *4.5.3 Obecný popis implementace vstupního bodu*.

Pakliže mají být v rámci synchronizace záznamy prověřeny z hlediska duplicity, je provedena kontrola na základě seznamu záznamů, které jsou již dočasně uloženy v kontextovém úložišti. Záznamy v kontextovém úložišti jsou ukládány do polí v podobě časových značek. Značky jsou tříděny na základě příslušnosti ke konkrétnímu vstupnímu bodu a konkrétnímu zařízení, viz následující obrázek.



Obrázek 13 - náhled na uložené časové značky v kontextovém úložišti NodeRED; Zdroj: vlastní

Pokud je nalezena shoda mezi aktuálně zpracovávanou časovou značkou a některou časovou značkou, která je uložena v kontextovém úložišti pro dané zařízení, zpráva bude zahozena. V opačném případě je do kontextového úložiště přidána aktuálně zpracovávaná časová značka a zpráva je odeslána dál datovým tokem.

V kontextovém úložišti je standardně ke každému zařízení udržováno 25 záznamů. Pokud by mělo dojít k překročení limitu, je z pole odebrána nejstarší časová značka, která je nahrazena časovou značkou aktuálně zpracovávanou.

```
...
if (datesArr[deviceIndex].lastTimeRecords.find(element => element ===
timestamp)) {
    msg.alreadySaved = true;
    return msg;
} else {
    datesArr[deviceIndex].lastTimestamp = timestamp;
    datesArr[deviceIndex].lastTimeRecords.push(timestamp);
}
...
if (datesArr[deviceIndex].lastTimeRecords.length > 25){
    datesArr[deviceIndex].lastTimeRecords.shift();
}
```

Zdrojový kód 14 - příklad hledání duplicintích záznamů a uložení aktuální časové značky záznamu; Zdroj: vlastní

Po vykonání uvedených kroků jsou data připravena k transformaci do výstupní podoby.

Jednotlivé vybrané parametry zprávy jsou transformovány do požadované podoby, kdy jsou jednotlivé atributy měření převedeny na objekty ve tvaru { key: x, value: y}, viz ukázka zdrojového kódu níže. Zároveň s tím je proveden záznam do archivačního souboru ohledně zpracování záznamu.

```
...
msg.forJSON = {
    "source" : msg.payload.name,
    "headers" : headers(),
    "timestamp": timestamp,
    "values" : formatData(),
    "rawData" : msg.payload
}

function formatData() {
    let valuesArr = []
    for (const [key, value] of Object.entries(msg.payload.data)) {
        valuesArr.push({ "key": `${key}`, "value": `${value}` });
    }
    return valuesArr;
}
...
```

Zdrojový kód 15 - příklad mapování jedné z položek výstupních dat; Zdroj: vlastní

Po tomto kroku nabývají zpracovaná data požadované struktury a mohou být odeslána do datové platformy k uložení.

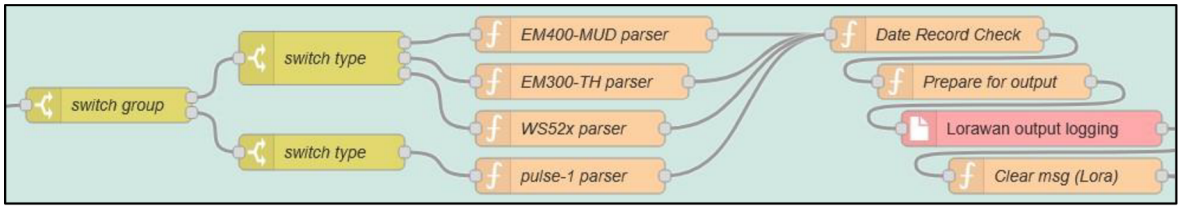
4.5.5 Realizace řešení: Vstupní bod LoRaWAN

V rámci implementace vícero vstupních sběrných míst byla implementována i struktura přijímající data ze školní instance technologie LoRaWAN. Tato struktura se od výše rozebrané struktury liší především tím, že se jedná o přijímač na bázi pasivního odposlechu a nelze tedy realizovat například ruční synchronizaci.

Dalším rozdílem je, že každý přijatý záznam se vždy vztahuje k jednomu konkrétnímu zařízení a jedné časové značce, čímž odpadá nutnost dělení zprávy dle jednotlivých záznamů. Na druhou stranu jsou data v rámci tohoto vstupního bodu přijímána ve formátu Base64 či případně ve formátu hexadecimálního zásobníku, které je nutné před transformací vlastních dat zařízení nejprve přeložit do vhodnějšího formátu (Javascript objektu).

Taktéž je nutné zmínit, že v rámci tohoto vstupního bodu mohou být přijímána data z různorodých typů zařízení. U každého typu se může lišit struktura zasílaných dat a tedy i způsob realizace překladu dat do vhodnějšího formátu. Pro každý typ zařízení, který měl být v rámci této struktury podporován, bylo tedy nutné doložit na míru upravený skript překladače.

Na základě výše zmíněných informací byla tato struktura osazena skupinou rozhodovacích uzlů, které příchozí zprávu přesměrují na základě unikátní identifikace zařízení na vhodný skript překladače, viz následující obrázek.



Obrázek 14 - náhled na část struktury realizující překlad zprávy ze vstupního bodu technologie LoRaWAN; zdroj: vlastní

Obsah skriptu překladače může vypadat například jako na přiložené ukázce kódu.

```
function DecodeMsg(bytes) {
   var decoded = {};
   for (let i = 0; i < bytes.length;) {
      let channel_id = bytes[i++];
      let channel_type = bytes[i++];
      // BATTERY
      if (channel_id === 0x01 && channel_type === 0x75) {
         decoded.battery = bytes[i];
         i += 1;
      }
      // TEMPERATURE
      else if (channel_id === 0x03 && channel_type === 0x67) {
         decoded.temperature = readInt16LE(bytes.slice(i, i + 2)) / 10;
         i += 2;
      }
      ...
```

Zdrojový kód 16 - ukázka části skriptu překladače; Zdroj: vlastní

Po průchodu zprávy skriptem překladače mohou být přeložená data transformována do požadované výstupní podoby a odeslána do datové platformy. Níže je uveden konkrétní příklad překladu příchozích dat – zachycena je vstupní a výstupní podoba.

```
data: "AXVkA2fvAASC/f8FAAE="

values: array[4]
        0: object
                key: "battery"
                value: "100"
        1: object
                key: "temperature"
                value: "23.9"
        2: object
                key: "distance"
                value: "65533"
        3: object
                key: "position"
                value: "tilt"
```

Zdrojový kód 17 - porovnání přijatých a zpracovaných dat zařízení (LoRaWAN); Zdroj: vlastní

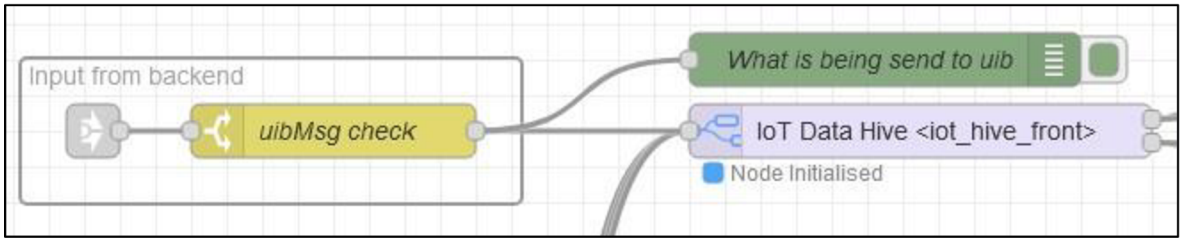
4.6 Implementace: Transakční část (back-end)

Jedná se o část aplikace, která umožňuje komunikaci mezi prezentační částí (front-endem) a funkcionální částí (zbytkem back-endu) aplikace. Dále je zde řešena problematika autentizace uživatele či například definování konfiguračních souborů pro front-end aplikace. Jednotlivé funkce či skupiny funkcí jsou popsány v podkapitolách níže.

Pro realizaci této části aplikace bylo nutné nainstalovat NodeRED rozšíření „uibuilder“, které mimo jiné umožňuje zobrazení prezentační části aplikace a které pro tento účel disponuje několika různorodými NodeRED uzly, viz obrázek 15.

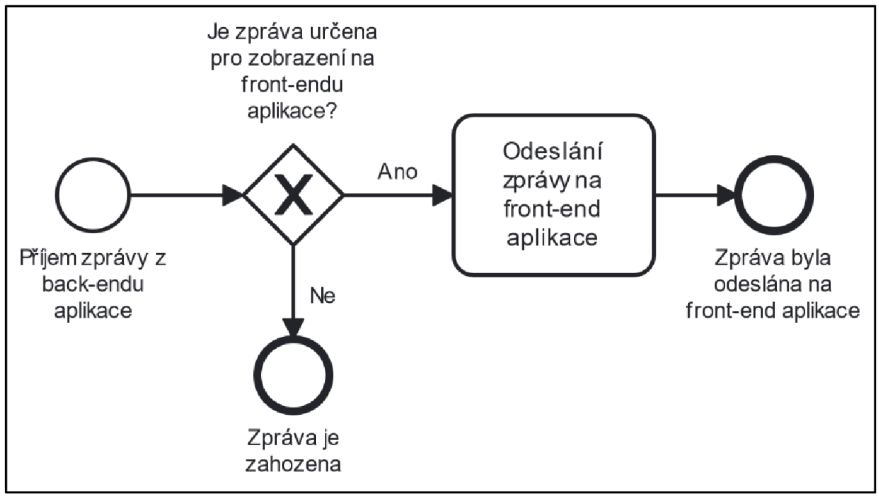
4.6.1 Příjem zpráv z funkcionální části aplikace

Příjem zpráv z funkcionální části aplikace je napojen na vstup prezentační části aplikace (na uzel nainstalovaného rozšíření „uibuilder“).



Obrázek 15 - NodeRED implementace - příjem zprávy z funkc. části aplikace; Zdroj: vlastní

Posloupnost přesměrování zprávy obsahuje jednoduchý kontrolní bod realizovaný skrze uzel rozhodování, jehož rozhodovací proces je zachycen na přiloženém diagramu notace BPMN.



Obrázek 16 - znázornění rozhodovacího procesu - příjem zprávy z funkc. části aplikace; Zdroj: vlastní

Rozhodování je prováděno na základě hodnoty konkrétního atributu zprávy, který může nabývat pouze hodnot “true” a “false” (1 a 0). Zpráva je na front-end aplikace přesměrována pouze v případě, že je daný atribut zprávy definován a zároveň nastaven na hodnotu “true”.

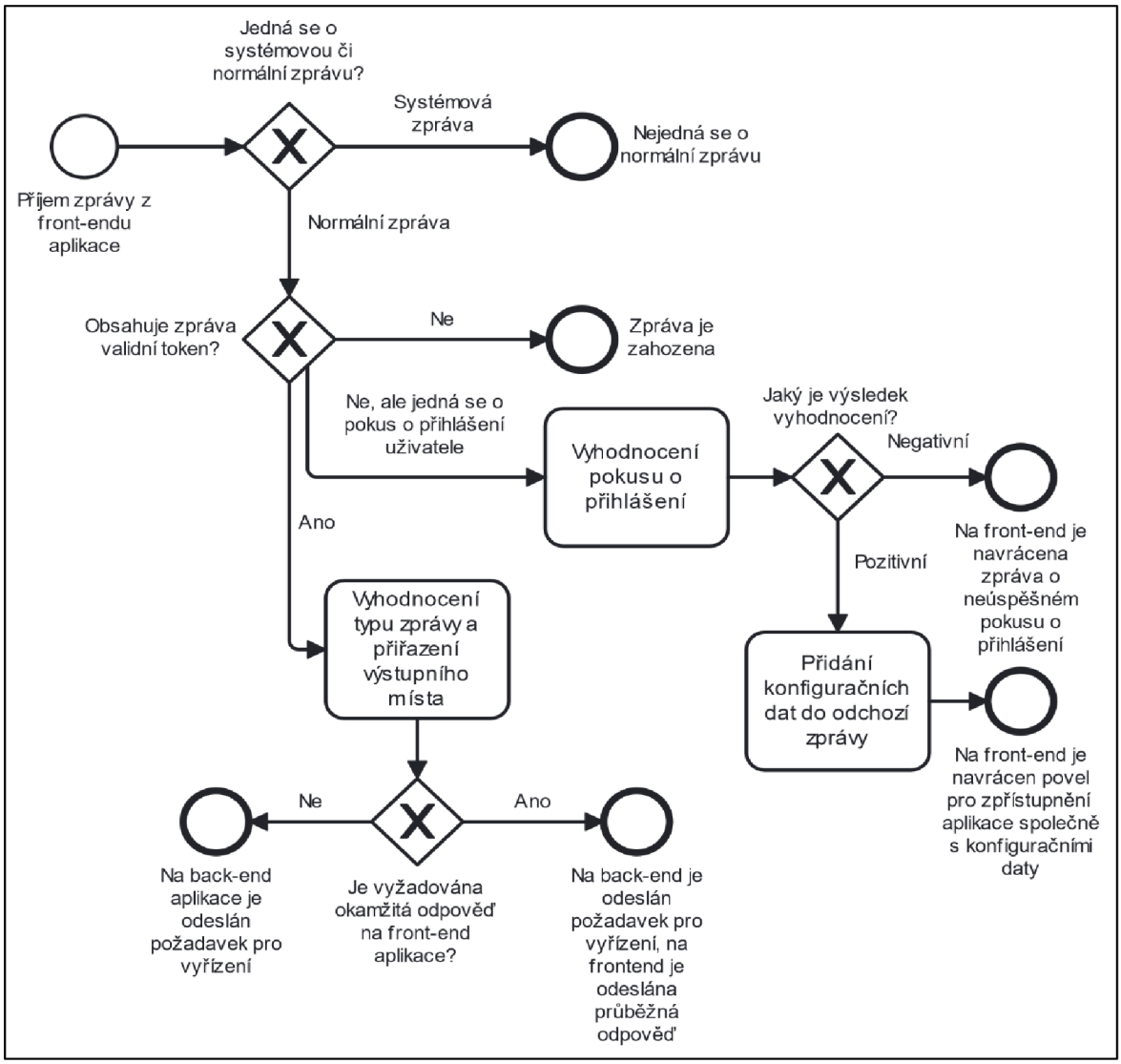
Tím je zamezeno odesílání takových zpráv, které nebyly k odeslání na front-end aplikace přímo určeny a které mohou obsahovat citlivé údaje, které nemají být uživateli zobrazeny.

4.6.2 Příjem zpráv z prezentační části aplikace

Z front-endu aplikace jsou skrze „uibuilder“ uzel přijímány dva různé typy zpráv – „kontrolní“ a „normální“ zprávy.

Kontrolními zprávami jsou zprávy, které generuje rozšíření „uibuilder“ na základě své vnitřní funkcionality nezávisle na vnějším zásahu vývojáře či koncového uživatele. Tyto zprávy mají v NodeRED struktuře vyhrazen svůj oddělený kanál pro příjem a jsou tak odděleny od zbytku „normálních“ zpráv. Z bezpečnostních důvodů projektu zde není konkrétní využití či metoda zpracování kontrolních zpráv blíže rozebírána. Zpravidla se jedná o zprávy informující o navázání spojení s novým klientem či zprávy, které toto spojení nastavují a případně udržují aktivní.

Normální zprávy jsou pak standardní zprávy, které obsahují příchozí požadavky generované uživatelem či systémovou akcí, kterou uživatel nepřímo vyvolal. Tyto zprávy při zpracování prochází několika různými fázemi. Celý proces zpracování je ve zjednodušené podobě zachycen dále na diagramu notace BPMN.



Obrázek 17 - znázornění rozhodovacího procesu - příjem zprávy z prezentační části aplikace; Zdroj: vlastní

První fází je ověření tokenu zprávy, kdy je token příchozí zprávy porovnán s tokenem aktivního připojení, který je generován při úspěšném dokončení autentizace. Pokud není nalezena shoda a nejedná se o zprávu obsahující požadavek pro provedení autentizace, je zpráva terminována a požadavek, který je ve zprávě obsažen, není zpracován. Zároveň s tím je na front-end aplikace konkrétnímu klientu odeslána zpráva vynucující zobrazení přihlašovacího okna.

Pokud zpráva projde skrze fázi ověření, je na základě jejích řídících parametrů určeno, kam bude zpráva dále odeslána. Dle typu zprávy je zároveň určeno, zda-li se má na front-end aplikace odeslat průběžná odpověď, na základě které je upraven stav zobrazení front-endu, dokud není přijata požadovaná odpověď na původní požadavek.

Příklad: Uživatel skrze front-end rozhraní zadá požadavek na ruční synchronizaci konkrétního vstupního bodu v konkrétním časovém intervalu. Nejprve je mu na front-end aplikace navrácena průběžná odpověď, na základě které je uživateli zobrazena informace, že je jeho požadavek vyřizován. Ve chvíli, kdy je na front-endu aplikace přijata odpověď na původně vyvolaný požadavek, zobrazí se uživateli výsledek jeho požadavku.

4.6.3 Autentizace uživatele

Autentizace uživatele je iniciována na základě příjmu zprávy se specifickými parametry. Pro zpracování tohoto typu zprávy není vyžadováno ověření tokenu, jelikož pro dané spojení ještě žádný token vygenerován nebyl. Konkrétní proces autentizace není z bezpečnostních důvodů v této práci zachycen, nicméně lze zmínit, že tato funkcionalita byla vystavěna právě na prostředí NodeRED a nevyužívá žádné ověřování třetí stranou. To umožnilo proces přihlašování a jeho dílčí kroky upravit na míru pro tuto konkrétní realizaci.

Pokud je výsledkem procesu autentizace úspěšné ověření uživatele, transakční část aplikace vygeneruje a uloží dočasný token relace a zaregistruje jeho spojení jako ověřené. Následně je do zprávy, která má být na front-end aplikace odeslána, kromě vygenerovaného tokenu zahrnuta i konfigurace, na základě které se má uživateli přizpůsobit obsah stránky.

Každý ze zapojených vstupních bodů je v této konfiguraci reprezentován objektem, který obsahuje základní parametry pro vykreslení dat uživateli, viz obrázek níže. Konkrétní příklad užití lze najít v kapitole "*4.7.4 Generování obsahu aplikace*".

Obrázek 18 - ukázka části konfiguračních dat pro front-end aplikace

4.7 Implementace: Prezentační část (front-end)

K implementaci prezentační části aplikace bylo využito NodeRED rozšíření „uibuilder“, které je zmíněno v kapitole *4.6 „Implementace: Transakční část (back-end)“*. Toto rozšíření disponuje vlastním běhovým serverem, který může být spuštěn v instanci NodeRED serveru, díky čemuž lze zpřístupnit prezentační část aplikace bez nutnosti zajišťování dalšího běhového prostředí. Kromě běhového serveru rozšíření „uibuilder“ zajišťuje i sjednocený postup posílání NodeRED zpráv mezi back-endem a front-endem aplikace (mezi transakční a prezentační částí) [26].

4.7.1 Příjem a odesílání zpráv

Příjem i odesílání zpráv skrze rozšíření „uibuilder“ lze na front-endu aplikace realizovat v jakékoliv části aplikace. Pro zpřístupnění této funkcionality je ovšem nutné se v jednotlivých částech aplikace odkázat na deklarovanou instanci daného rozšíření.

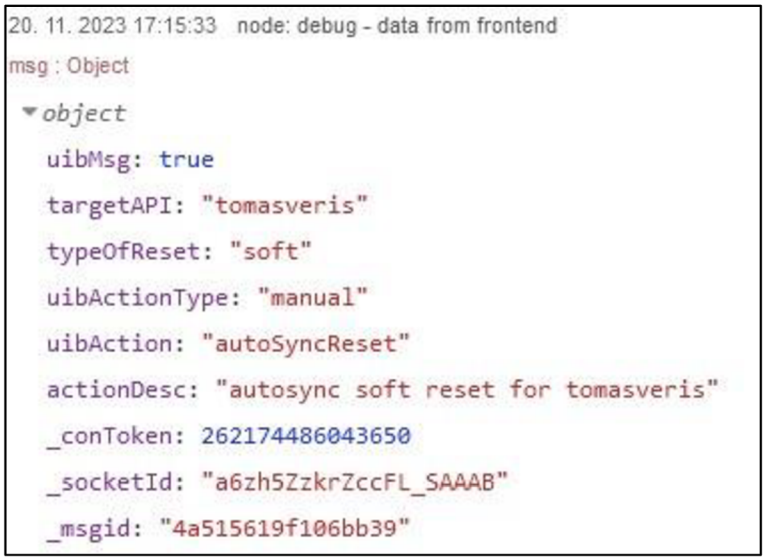
Za tělo zpráv, které se mezi transakční a prezentační částí vyměňují, lze považovat standardní Javascript objekt, který nabývá podoby NodeRED zprávy. Na front-endu aplikace tak mohou být přijímána a zpracovávána data v přesně takové podobě, v jaké se s nimi zachází přímo v rozhraní NodeRED.

Níže je zachycen příklad odeslání zprávy z prezentační části do transakční části aplikace. Jak bylo zmíněno výše, je definován standardní Javascript objekt, ke kterému následně při zpracování přidá rozšíření „uibuilder“ řídící parametry jako vygenerovaný identifikátor a předmět zprávy. Zpráva je odeslána zavoláním „uibuilder“ funkce „uibuilder.send()“.

```
resetAutoSync: function(resetTarget, resetType){
      uibuilder.send( {
            'uibMsg' : true,
            'targetAPI' : resetTarget,
            'typeOfReset' : resetType,
            'uibActionType' : "manual",
            'uibAction' : "autoSyncReset",
            '_conToken' : this.clientSecret,
      } );
}
```

Zdrojový kód 18 - příklad odeslání zprávy z front-endu na back-end aplikace skrze rozšíření "uibuilder"; Zdroj: vlastní

Pokud je skrze funkci, která se nachází v předešlé ukázce zdrojového kódu, odeslána zpráva, přijde do transakční části aplikace zpráva v podobě zachycené na obrázku níže.



Obrázek 19 - ukázka zprávy přijaté z front-endu aplikace skrze rozšíření uibuilder; Zdroj: vlastní

Pokud má být na front-endu aplikace zpráva naopak zachycena a zpracována, musí být deklarována instance funkce „uibuilder.onChange()". V daném bloku kódu lze následně pracovat s příchozími zprávami – například je třídit pomocí struktury "if-else", viz zdrojový kód níže.

```
uibuilder.onChange('msg', function(msg) {
      if (msg.uibRtrnAction == "processing") {
            view_download.showBottomToast_error = false;
            view_download.showBottomToast_info = true;
            view_download.bottomToast_infoMsg = "Zpracovávání požadavku";
      }
      ...
```

Zdrojový kód 19 - ukázka zachycení příchozí zprávy na front-endu aplikace

4.7.2 Knihovny a Node.js balíky použité pro realizaci prezentační části

V rámci začlenění rozšíření „uibuilder" do front-endu aplikace byla zpřístupněna možnost instalovat a využívat jakékoliv knihovny z repozitáře NPM, kde se nachází Node.js kompatibilní balíky funkcí.

Pro realizaci prezentační části byly vybrány balíky funkcí, které jsou stručně popsané níže. Výběr balíků byl realizován na základě subjektivního uvážení autora této práce, které se odvíjelo od jeho dosavadní praxe v oboru a od limitací definovaných pro tuto práci.

Framework Vue.js – komplexní balík funkcí, pomocí kterých lze realizovat vývoj dynamických responzivních stránek (webových aplikací) . Popsán v teoretické části práce.

Knihovna VueRouter – kompaktní knihovna zajišťující načítání různých pohledů (stránek) aplikace [27].

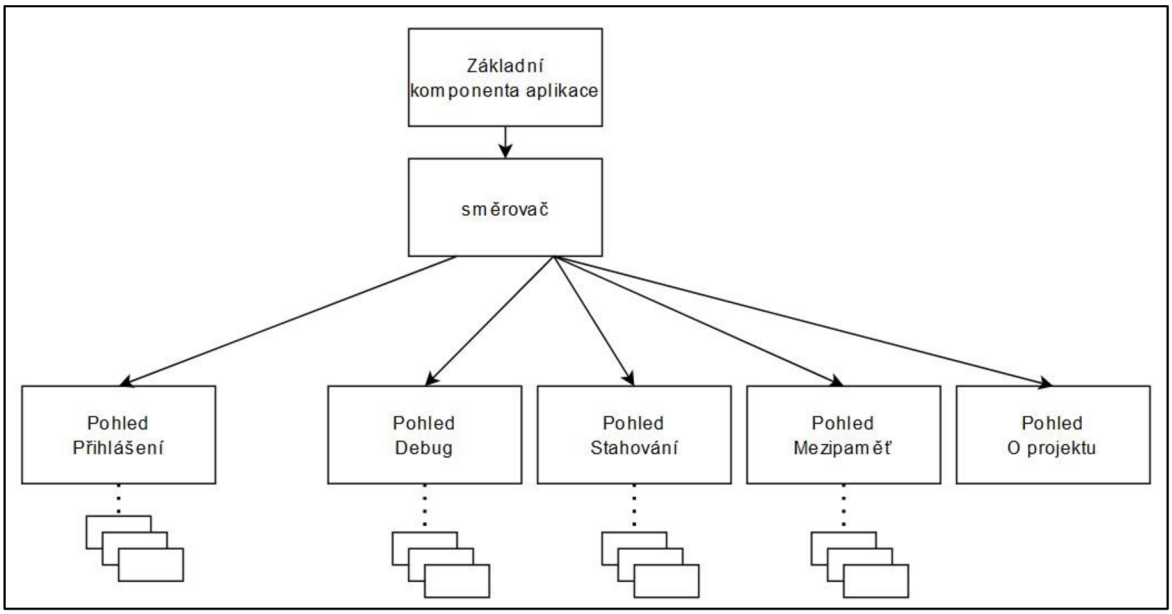
Knihovna http-vue-loader – kompaktní knihovna, pomocí které lze zpřístupnit Vue aplikaci a veškerou funkcionalitu Vue komponentů bez nutnosti transformace aplikace skrze proces exportování do nativního javascriptového prostředí [28].

Knihovna Bootstrap a Bootstrap-vue – knihovny obsahující řadu předdefinovaných stylů a komponent, které lze ve Vue aplikaci využít a urychlit či sjednotit tím její vývoj [29].

4.7.3 Informační architektura

Aplikace je z pohledu uživatele rozdělena do čtyř sekcí – „Debug“, „Mezipaměť“, „Stahování“ a dodatečné sekce „O projektu“, která neobsahuje žádnou důležitou funkcionalitu a tudíž nebude dále rozebírána. Z hlediska zdrojového kódu se jedná o čtyři hlavní Vue.js komponenty nebo-li pohledy aplikace, skrze které se načítá příslušná část obsahu aplikace. Jednotlivé pohledy pak obsahují konkrétní, hierarchicky podřazené komponenty.

Pokud není uživatel přihlášen, je vynuceno zobrazení pohledu pro přihlášení nezávisle na tom, který z pohledů aplikace se uživatel pokusil zobrazit. Pokud je uživatel přihlášen, může libovolně přepínat mezi výše zmíněnými pohledy až do ukončení aktuální relace. Tuto funkcionalitu zajišťuje výše zmíněná knihovna „VueRouter“, která umožňuje skrze javascriptový soubor definici jednotlivých pohledů, respektive jednotlivých cest aplikace (demonstrováno na přiloženém obrázku a ukázce zdrojového kódu na další straně).



Obrázek 20 - náhled na základní možnosti zobrazení obsahu aplikace uživateli; zdroj: vlastní

```
const MainPage = httpVueLoader('./views/mainoverview.vue');
const CachePage = httpVueLoader('./views/contextstorage.vue');
...
export default {
    routes: [
        {
            path: '/',
            name: 'home',
            component: MainPage
        },
        {
            path: '/cache',
            name: 'cache',
            components: {
                default: CachePage,
            },
        },
        ...
    ],
};
```

Zdrojový kód 20 - příklad části skriptu směrovače front-endu aplikace (VueRouter); zdroj: vlastní

Jednotlivé pohledy mohou obsahovat další dílčí komponenty, které jsou použity pro vykreslování obsahu aplikace. Využití těchto dílčích komponent má pozitivní přínos

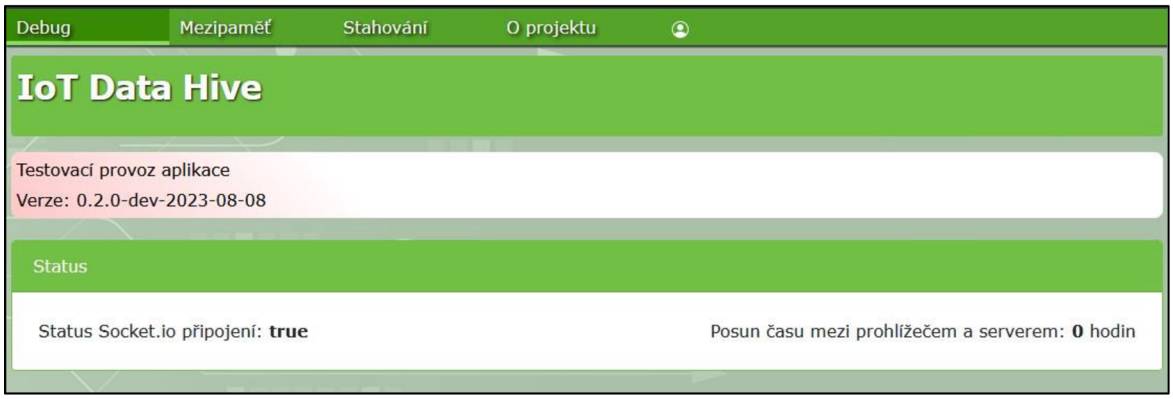
především pro znovupoužitelnost zdrojového kódu, přičemž je zároveň zlepšena jeho čitelnost a přehlednost. Komunikace a princip sdílení dat mezi těmito komponentami jsou blíže specifikovány v teoretické části práce (kapitola *3.8 Framework Vue.js*), přičemž v následujících kapitolách věnovaných konkrétním pohledům front-endu aplikace je k náhledu dostupné konkrétní použití daných teoretických podkladů.

4.7.4 Generování obsahu aplikace

Zobrazení obsahu je podmíněno přihlášením. Při úspěšném přihlášení uživatele je z back-endu načtena úvodní konfigurace na základě které mohou být později vzneseny automatizované požadavky pro příjem konkrétních dat ohledně jednotlivých vstupních bodů (skupin zařízení), viz kapitola *4.6 Implementace: Transakční část (back-end)*. Na základě příjmu těchto dat je následně vygenerován hlavní obsah aplikace. Data pro jednotlivé pohledy aplikace jsou tedy standardně načítána až ve chvíli, kdy se uživatel rozhodne konkrétní pohled zobrazit. Tím je docíleno snížení síťové zátěže, která by jinak byla soustředěna na moment přihlášení, respektive zpřístupnění front-endu aplikace po autentizaci uživatele.

4.7.5 Pohled: Ladění (Debug)

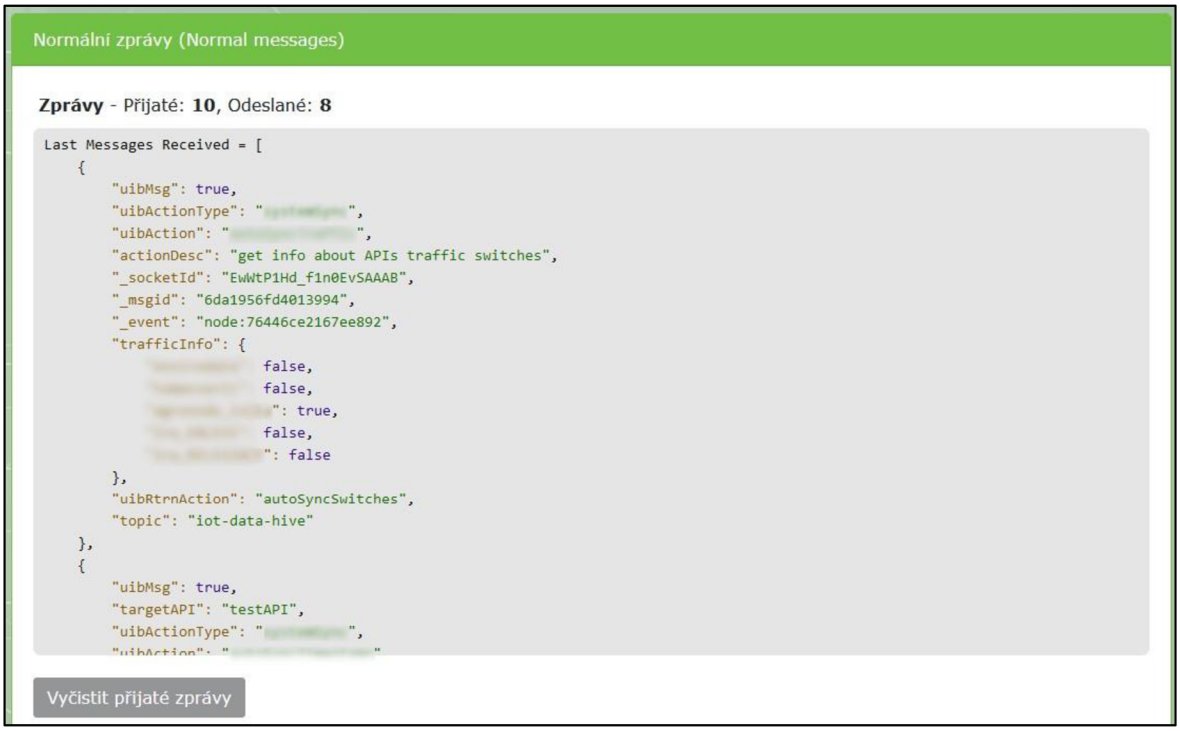
Základním pohledem, který se standardně po přihlášení uživateli zobrazí, je stránka poskytující surové informace o aktuálním stavu front-endu aplikace. V horní části stránky lze nalázt aktuální verzi front-endu aplikace a status připojení klienta k serveru.



Obrázek 21 - horní část stránky ladění na front-endu aplikace; zdroj: vlastní

Dále se na stránce nachází pohled na téměř surovou podobu příchozích a odchozích dat aplikace, která jsou rozdělena dle typu zpráv (normální a řídící), přičemž jsou dále děleny na příchozí a odchozí. Zprávy jsou tedy tříděny do 4 různých kategorií – normální odeslané,

normální přijaté, řídící odeslané a řídící přijaté. Každá z těchto kategorií disponuje na této stránce svým vlastním oknem, kde se zprávy zobrazují. Každá z kategorií má nastavený maximální počet zpráv, který je na straně klienta udržován v paměti (aktuální nastavení v době psaní této práce činilo 10 zpráv na kategorii). U kategorií normálních zpráv je přidána možnost zprávy na straně klienta promazat skrze přidružené tlačítko. Zprávy jsou chronologicky řazeny od nejnovější po nejstarší. Zároveň jsou dodatečně formátovány tak, aby byly lépe čitelné, viz následující obrázek.



Obrázek 22 - stránka ladění, pohled na zobrazení toku zpráv; zdroj: vlastní

Tato stránka tak ve výsledku dokáže poskytnout data, která jsou v ostatních pohledech front-endu aplikace buď nedostupná nebo záměrně skrytá, což lze efektivně použít například při dalším vývoji či ladění front-endu aplikace.

4.7.6 Pohled: Mezipaměť

Druhý pohled front-endu aplikace je zaměřen na podrobný výpis časových značek, které jsou uložené v kontextovém úložišti NodeRED na back-endu aplikace jakožto dočasná data, viz kapitoly *4.5.1 Řídící struktura: Kontextové úložiště* a *4.5.4 Realizace řešení: Vstupní bod typu http GET*.

Veškerý podstatný obsah tohoto pohledu je vykreslován na základě části konfiguračních dat, která byla klientem přijata při procesu přihlášení. Front-end aplikace na základě daných konfiguračních dat disponuje seznamem dostupných vstupních bodů včetně jejich systémových označení, což umožňuje provedení automatizovaných požadavků na transakční část aplikace, kde jsou dané požadavky dále přesměrovány na back-end, přičemž dojde k načtení seznamu zařízení a jejich časových značek konkrétního vstupního bodu z NodeRED kontextového úložiště.

Nejprve je z připojeného klienta do transakční části aplikace odeslána zpráva ve tvaru definovaném níže.

```
retrieveApiData: function(targetAPI){
   this.clearDownloadedCacheData(targetAPI);

   uibuilder.send( {
      'uibMsg' : true,
      'targetAPI' : targetAPI,
      'uibActionType' : "systemSync",
      'uibAction' : "apiInfo",
      'actionDesc' : "retrieve api cache",
      '_conToken' : this.clientSecret,
      }
   );
},
```

Zdrojový kód 21 - metoda pro vyvolání požadavku synchronizace dat vstupního bodu; zdroj: vlastní

Vzhledem k tomu, že daný požadavek může být později vyvolán ručně, je v rámci metody, která obsahuje volání požadavku, zahrnuto volání funkce pro vymazání aktuálně načtených dat zacíleného vstupního bodu, aby nedošlo k nežádoucí duplicitě poskytovaných dat. V rámci dalších optimalizací by mohlo být vymazání aktuálně načtených dat vstupního bodu přesunuto do metody, ve které se zpracovávají nově příchozí data, nicméně za standardních podmínek koncový uživatel nepozná žádný rozdíl.

Po přijetí požadavku v transakční části aplikace je zpráva přesměrována skrze proud uzlů do konkrétní části funkcionální části aplikace, kde je z kontextového úložiště NodeRED načtena požadovaná sada dat. V tomto konkrétním případě je sada dat složena ze seznamu

všech časových značek tříděných dle zařízení zacíleného vstupního bodu a časové značky načtení těchto dat.

Poznámka: Náhled na část tohoto procesu, včetně těla metody, která tento požadavek vyřizuje ve funkcionální části aplikace, byl vyobrazen v kapitole 4.5.1 Řídící struktura: Kontextové úložiště.

Poté, co je výše zmíněná sada dat vstupního bodu odeslána do prezentační části aplikace, jsou daná data převedena do struktury, která je uzpůsobena pro následné vykreslování konkrétních prvků front-endu aplikace.

Nejprve je vytvořena kopie přijatých dat. Vzhledem k principu funkcionality jazyka Javascript je potřeba si dát pozor na to, aby nebyl vytvořen pouhý odkaz na původní pole, nýbrž jeho nezávislá kopie. K tomu je využito funkcí JSON.parse() a JSON.stringify(), pomocí kterých je obsah původního pole nejprve převeden na JSON řetězec a následně z daného řetězce rekonstruován zpět do podoby Javascript objektu, tentokrát již coby nezávislé instance. Po vytvoření kopie datového pole je zavolána funkce pro úpravu přijatých časových značek, viz zdrojový kód 22.

```
...
if(this.apiData[targetAPI].rawData.dates != undefined) {
   rawDataCopy =
      JSON.parse(JSON.stringify(this.apiData[targetAPI].rawData));
   this.apiData[targetAPI].timestamps =
      this.getApiTimestamps(targetAPI, rawDataCopy);
}
...
```

Zdrojový kód 22 - náhled na vytvoření nezávislé kopie pole dat; zdroj: vlastní

Skrze zavolanou funkci je provedena změna pořadí přijatých časových značek. V kontextovém úložišti NodeRED jsou nově zpracované časové značky zapisovány na konec datového pole. V rámci zobrazení dat uživatele však chceme časové značky zobrazit od nejnovější po nejstarší, tudíž musí být pořadí časových značek obráceno (převedeno do reverzní podoby).

Po tomto procesu je v rámci každého zařízení zpracovávaného vstupního bodu definována nová proměnná, která udává binární informaci o aktivitě zařízení (aktivní/ neaktivní). Každý ze vstupních bodů má ve své úvodní konfiguraci pro front-end aplikace definován parametr udávající počet sekund od posledního synchronizovaného záznamu, po kterém má být jakékoliv ze zařízení daného vstupního bodu označeno pro uživatele jako neaktivní.

Tato hodnota je použita při porovnání časové značky posledního synchronizovaného záznamu zařízení s časovou značkou momentu porovnávání (aktuálního času). Pokud je rozdíl těchto dvou časových značek vyšší než hodnota výše zmíněného parametru, je zařízení označeno jako neaktivní, viz zdrojový kód níže. Tato proměnná je následně použita při vykreslení obsahu stránky tak, aby byla neaktivní zařízení graficky i symbolově odlišitelná, viz následující obrázky s čísly 24 a 25.

```
. . .
procDate.responding =
   this.checkLastDeviceResponseTime(procDate.dates[j], inactiveTimer);
. . .

checkLastDeviceResponseTime: function(lastDate, inactiveTimer) {
   let lastTimestamp = parseInt(new Date(lastDate).getTime());
   let currentTimestamp = parseInt(new Date().getTime());
   inactiveTimer = parseInt(inactiveTimer);

  if(lastTimestamp < (currentTimestamp - inactiveTimer)){
     return false;
  } else {
     return true;
  }
},
```

Zdrojový kód 23 - náhled na volání a obsah metody pro určení aktivity zařízení; zdroj: vlastní

Pakliže jsou takto přijata a zpracována data jednotlivých vstupních bodů, aplikace překreslí zobrazovaný obsah aplikace skrze kondicionální a iterativní techniky, které knihovna Vue.js nabízí a které byly popsány v teoretické části práce, viz ukázka zdrojového kódu dále.

Pro každý z definovaných vstupních bodů je vytvořena instance komponenty, skrze kterou se budou data vykreslovat. Na přiloženém zdrojovém kódu lze vidět, že komponenta přijímá jak některá konfigurační data přijatá při autentizaci, tak dodatečně zpracovaná data z kontextového úložiště jakožto vlastnosti komponenty, společně se dvěma odkazy na řídící metody.

```
<template v-for="api in apiConfig" :key="api">
   <contextstorage-tab
      @retrieve-data="retrieveApiData"
      @set-device-wipe-values="setDeviceWipeValues"

      :api-code="api.sysName"
      :api-name="api.desName"
      :api-context-data="apiData[api.sysName].rawData"
      :api-timestamps="apiData[api.sysName].timestamps"
      :render-device-nav="api.multiDevices"
   >
   </contextstorage-tab>
</template>
```

Zdrojový kód 24 - iterativní vykreslování komponenty reprezentující vstupní body; zdroj: vlastní

Z pohledu uživatele je pro každý z definovaných vstupních bodů vytvořena nová záložka v navigační části tohoto pohledu, viz následující obrázek.

Obrázek 23 - náhled na vygenerovanou navigaci; zdroj: vlastní

Pokud se jedná o vstupní bod, kterému náleží vícero různých zařízení, je vykreslena i dodatečná navigační sekce, kde jsou zobrazeny názvy všech příslušných zařízení, které pod vstupní bod spadají, včetně odkazů na konkrétní seznam časových značek daného zařízení.

Pokud definovaný vstupní bod přijímá data pouze z jednoho zařízení, tato dodatečná navigační sekce, která je zachycena na obrázku níže, se nezobrazí.

Obrázek 24 - náhled na vygenerovanou sekundární navigaci pro konkrétní záložku; zdroj: vlastní

Pro každé zařízení se nakonec vykreslí karta, ve které se nachází obecný název zařízení, časové značky synchronizovaných záznamů a tlačítko, které umožňuje tato dočasná data vymazat z kontextového úložiště – tato funkce se zde nachází pro jednoduché odstranění neaktivních či vyřazených zařízení, u kterých již postrádá smysl dočasná data uchovávat. Konkrétní podoba těchto karet je zachycena na následujícím obrázku.

Obrázek 25 - náhled na vygenerované karty reprezentující jednotlivá zařízení; zdroj: vlastní

Odebrání zařízení z mezipaměti (kontextového úložiště NodeRED) je, stejně jako jiné kritické zásahy do systému, nutné potvrdit skrze dialogové okno, které se po kliknutí na příslušné tlačítko zobrazí (viz obrázek 26). I toto dialogové okno je realizováno jako samostatná komponenta, která při potvrzení požadavku volá skrze vestavěnou funkci „$emit()“ příslušnou metodu v hierarchicky nadřazené komponentě.

Obrázek 26 - náhled na dialogové okno pro potvrzení akce; zdroj: vlastní

Na závěr se pod výše popsaným zobrazením obsahu mezipaměti nachází dodatečná sekce, kterou lze zviditelnit po kliknutí na tlačítko „Zobrazit surová data“ (na obrázku níže). V této sekci se nachází výpis neupravených dat mezipaměti (surová data, jejichž kopie byla před zpracováním dat zachována pro tyto účely).

Obrázek 27 - náhled na sekci se zobrazením surových dat mezipaměti; zdroj: vlastní

Shrnutí pohledu Mezipaměť

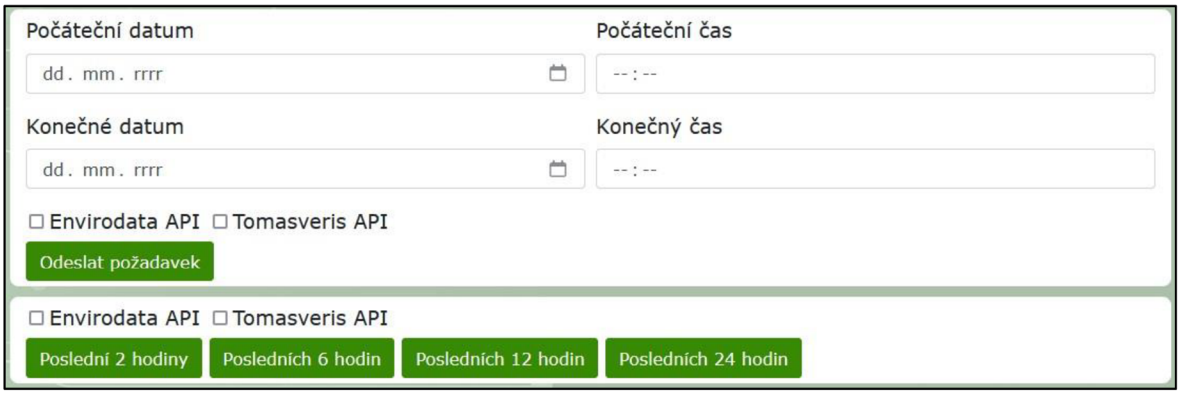
Obsah tohoto pohledu je, dle popisu výše, koncipován následovně. V horní části se nachází dodatečná navigace pro přepínání mezi jednotlivými vstupními body a zařízeními v nich. Následuje výpis časových značek uspořádaný do karet dle jednotlivých zařízení. Jednotlivá zařízení lze z mezipaměti odebírat, aktuálnost zobrazovaných dat lze ručně vynutit příslušným tlačítkem. V dolní části zobrazení lze nahlédnout na data v takové podobě, v jaké byla přijata z back-endu aplikace.

Načítání a zobrazování dat pro tento pohled front-endu aplikace je plně dynamické. Vykreslování obsahu se zcela odvíjí od poskytnutých konfiguračních dat a dodatečně načtených dat při návštěvě tohoto pohledu.

4.7.7 Pohled: Stahování

Jak název napovídá, obsah tohoto pohledu je zaměřen přímo na stahování, respektive synchronizaci dat. Stejně jako v pohledu „Mezipaměť“ se i zde při načtení této sekce front-endu aplikace provede sada automatizovaných požadavků, které jsou principem velmi podobné těm z předešlé kapitoly. Na základě dodatečně přijatých dat a úvodních konfiguračních dat jsou vykresleny dvě hlavní části pohledu – část pro ruční synchronizaci a část pro přehled automatické synchronizace.

Pokud má přístupový bod v konfiguračních datech uvedeno, že podporuje funkcionalitu ruční synchronizace, je daný vstupní bod zahrnut do seznamu bodů, které se v sekci určené pro ruční synchronizaci zobrazí, viz následující obrázek.



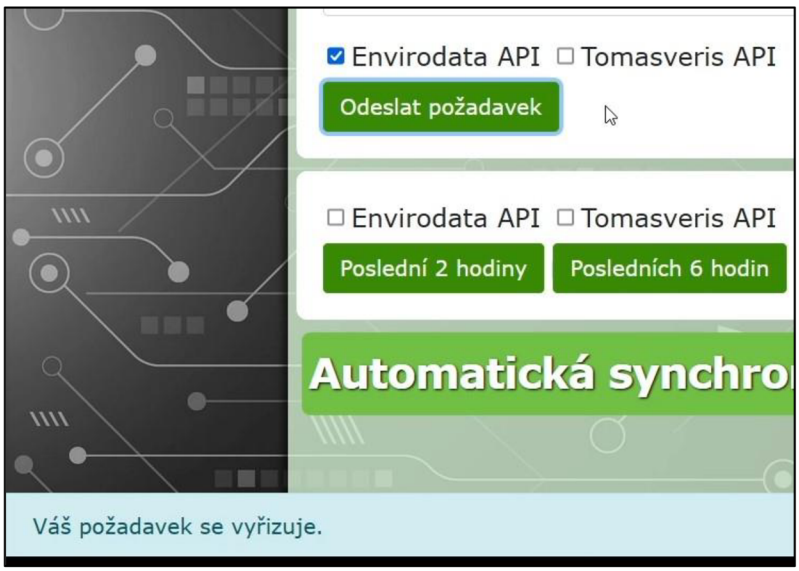
Obrázek 28 - náhled na komponentu ruční synchronizace; zdroj: vlastní

Jak vyplývá z předešlého obrázku, uživatel může pro vstupní body, které ruční synchronizaci podporují, zadat časový interval, ze kterého chce dodatečně stáhnout data zařízení, případně může zvolit jednu z předdefinovaných možností (např. poslední 2 hodiny). Uživatel může také zadat synchronizaci pro vícero vstupních bodů najednou – požadavky budou vyřizovány paralelně. V případě tohoto typu požadavku je na back-end aplikace odesílána zpráva ve tvaru zobrazeném v následující ukázce zdrojového kódu. Seznam zacílených vstupních bodů je odeslán ve formě seznamu (pole).

```
sendInput_manDown_custom:
function(dateFrom, dateTo, timeFrom, timeTo, checkboxes) {
   uibuilder.send( {
      'dateFrom': dateFrom,
      'dateTo': dateTo,
      'timeFrom': timeFrom,
      'timeTo': timeTo,
      'timezoneOffset' : new Date().getTimezoneOffset(),
      'timezoneName' : Intl.DateTimeFormat().resolvedOptions().timeZone,
      'apiList' : checkboxes,
      'uibMsg' : true,
      'customTime' : true,
      'uibActionType' : "manual",
      'uibAction' : "manDown",
      'actionDesc' : "manual download from web interface",
      '_conToken' : this.clientSecret,
      } );
},
```

Zdrojový kód 25 - ukázka struktury zprávy pro vyvolání požadavku ruční synchronizace; zdroj: vlastní

Ve chvíli, kdy uživatel odešle požadavek pro uskutečnění ruční synchronizace, je na základě průběžné odpovědi z back-endu aplikace zobrazena průběžná zpráva, že je požadavek vyřizován (v rámci větších časových intervalů může vyřízení požadavku trvat i vyšší desítky sekund – záleží na počtu zařízení a intervalu pořizování dat v rámci zvolených vstupních bodů). Tato zpráva, zachycena na následujícím obrázku, zůstává zobrazena až do přijetí výsledku požadavku.



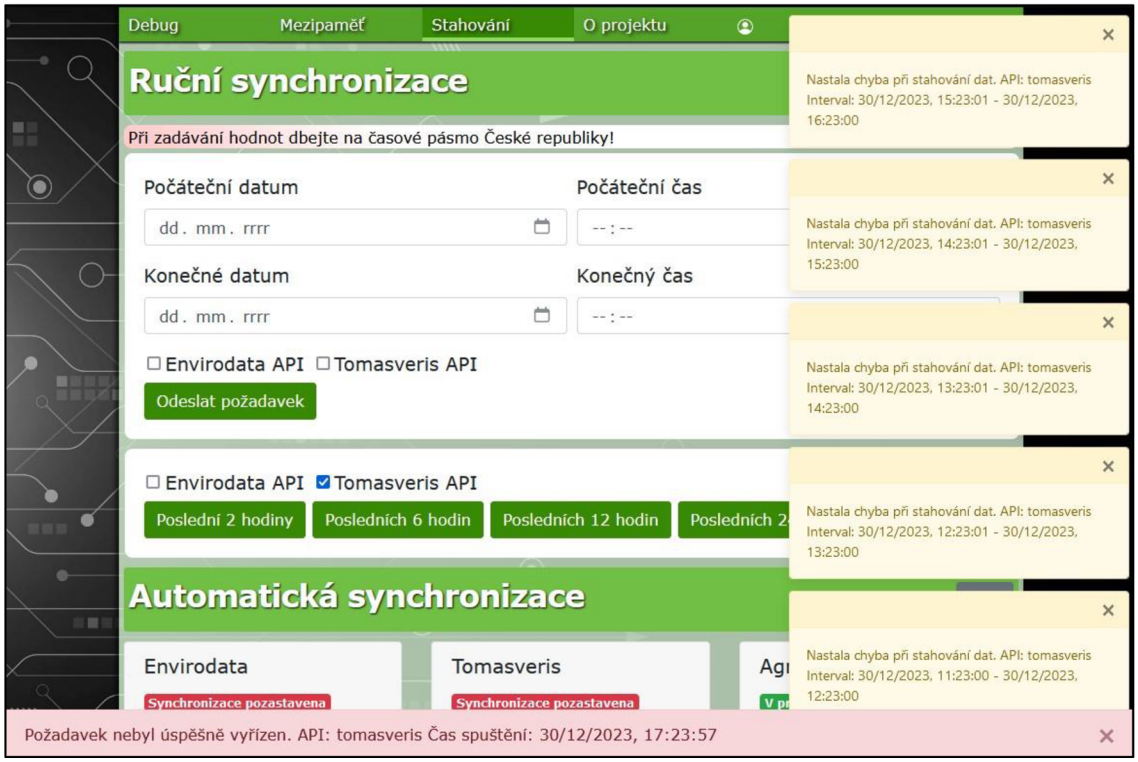
Obrázek 29 - náhled na průběžnou odpověď při zpracovávání požadavku ruční synchronizace; zdroj: vlastní

Mezitím je v transakční části back-endu aplikace přijatá zpráva zpracována. Pro každý zacílený vstupní bod je vyvolán samostatný požadavek synchronizace, který je odeslán do funkcionální části back-endu ke zpracování, viz ukázka zdrojového kódu níže.

```
msg.apiList.forEach(function(api) {
    if (api.checkbox == true){
        msg.targetAPI = api.name
        msg.autoSync = false;
        node.send(msg);
    }
});
```

Zdrojový kód 26 - ukázka dělení přijaté zprávy na vícero zpráv pro funkcionální část aplikace; zdroj: vlastní

Funkcionální část aplikace se následně pokusí požadavek vyřídit. Pokud kdykoliv v průběhu procesu nastane chyba a požadavek nemůže být úspěšně vyřízen, je na prezentační část aplikace navrácena zpráva, na základě které se uživateli zobrazí chybová hláška. Zároveň s tím jsou na prezentační části aplikace vypsány i jednotlivé dílčí intervaly synchronizace, ve kterých se chyba vyskytla. Příklad zobrazení je zachycen na následujícím obrázku.



Obrázek 30 - ukázka odezvy v rámci výskytu chyby při ruční synchronizaci; zdroj: vlastní

V opačném případě, tedy v případě, kdy byl požadavek úspěšně vyřízen, je uživateli zobrazena zpráva informující o úspěšném vyřízení požadavku.

Obrázek 31 - ukázka výstupní zprávy v rámci úspěšně vyřízeného požadavku; zdroj: vlastní

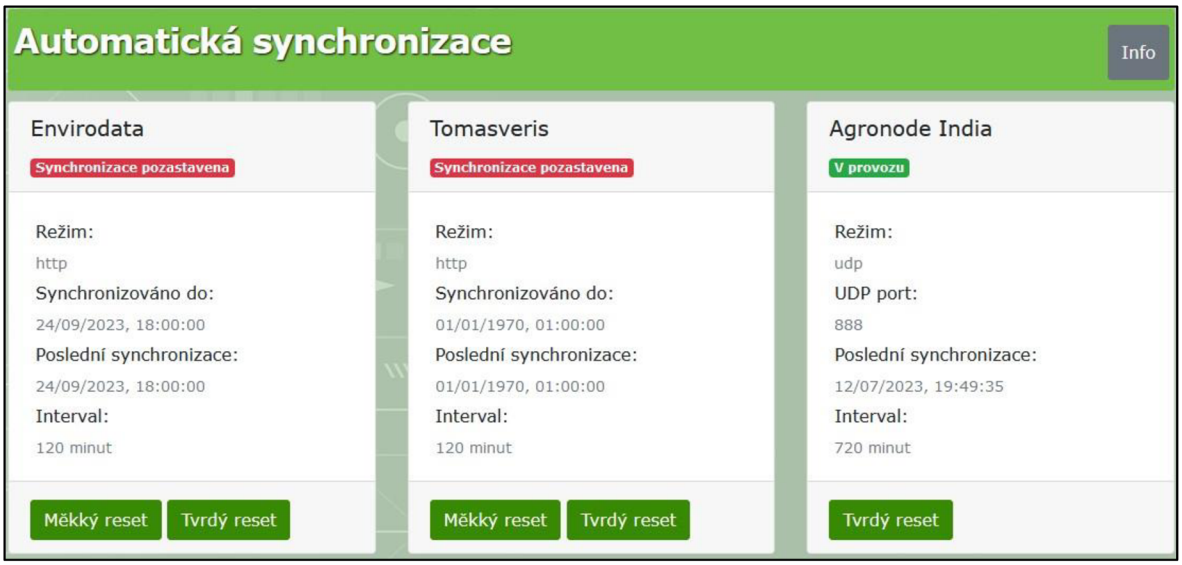
Kromě komponenty pro ruční synchronizaci zařízení se v tomto pohledu aplikace nachází také komponenta pro zobrazení stavu automatické synchronizace. Stav je monitorován v rámci celých vstupních bodů, nikoliv jednotlivých zařízení.

Pro každý ze vstupních bodů je vygenerována karta, která obsahuje název vstupního bodu, informaci o tom, zda-li je zapnuta funkcionalita automatické synchronizace, režim synchronizace (zkratkovité označení pro způsob příjmu dat), časové značky poslední synchronizace a interval synchronizace, viz následující ukázka zdrojového kódu.

```
<template v-for="api in apiConfig" :key="api">
   <div v-if="api.autoSyncType == 'http'" class="card">
      <div class="card-header">
         <h5 class="card-title">{{api.desName}} </h5>
      </div>
      . . .
      <div class="card-body">
         <p>Režim:</p>
         <p>{{ api.autoSyncType }</p>
         <p>Synchronizováno do:</p>
         <p>{{ apiSyncData[api.sysName].autoSyncTime }}</p>
         <p>Poslední synchronizace:</p>
         <p>{{ apiSyncData[api.sysName].autoSyncCallTime }}</p>
         <p>Interval:</p>
         <p>{{ api.syncInterval }} minut</p>
     </div>
     . . .
   </div>
. . .
```

Zdrojový kód 27 – zjednodušená ukázka vykreslení karty automatické synchronizace vstupního bodu; Zdroj: vlastní

Uživateli je zobrazen obsah zachycený na následujícím obrázku.



Obrázek 32 - ukázka vygenerovaných karet v sekci automatické synchronizace; zdroj: vlastní

Na obrázku výše lze vidět různé možnosti stavů karet. Obsah karty je uzpůsoben definovanému režimu synchronizace. Pokud se jedná o vstupní bod, který přijímá data skrze metodu http, obsah karty se bude lišit od obsahu karty, která zastupuje vstupní bod fungující na přenosové technologii UDP.

U prvních dvou vstupních bodů, které jsou na obrázku zachycené, je dostupná informace o tom, že je automatická synchronizace cíleně pozastavena. U vstupního bodu s označením "Tomasveris" lze zároveň vidět, že byl proveden požadavek jednoho ze dvou dostupných typů resetu automatické synchronizace, což pozměnilo řídící data v kontextovém úložišti, viz hodnoty v parametrech "synchronizováno do" a "poslední synchronizace".

Proces resetování automatické synchronizace může uživatel iniciovat skrze příslušné tlačítko, které obsahuje každá karta vstupního bodu, který danou instrukci podporuje. Pod pojmem resetování si v tomto případě lze představit vymazání či nahrazení konkrétních dat kontextového úložiště tak, aby byl upraven průběh následujícího cyklu automatické synchronizace.

Konkrétní dopad provedení určitého typu resetování vstupního bodu je popsán v přidruženém informačním okně. Stručný popis důsledku provedení konkrétního typu resetování je zahrnut i při potvrzování provedení dané akce, jak lze vidět na obrázku níže.

Obrázek 33 - ukázka dialogového okna pro potvrzení požadavku resetování vstupního bodu; Zdroj: vlastní

Resetování vstupního bodu může být užitečné například v případě znovuzapojení vstupního bodu po realizaci dlouhodobé odstávky, případně ke hromadnému vymazání dat spjatých s automatickou synchronizací v rámci celého vstupního bodu.

Shrnutí pohledu Stahování

Pohled stahování je rozdělen do dvou částí. První část slouží pro vyvolání požadavku ruční synchronizace pro vstupní body, které danou možnost podporují. Ve druhé části se nachází přehled jednotlivých vstupních bodů, ve kterém jsou k nalezení jejich základní informace v souvislosti s automatickou synchronizací dat. V rámci této části pohledu lze vyvolat požadavek na resetování konkrétního vstupního bodu, čímž lze ovlivnit parametry dalšího cyklu automatické synchronizace.

5 Výsledky a diskuse

Výsledkem výše popsaných postupů a implementací, a tedy výsledkem této práce, je část webové služby, respektive aplikace sestávající ze tří částí.

První částí je část funkcionální, která realizuje stěžejní zpracování dat z dostupných IoT zařízení. Každý ze vstupních bodů, ze kterých jsou data IoT zařízení přijímána, je zde reprezentován dílčí, nezávislou strukturou, která je na míru upravena specifickým odlišnostem a charakteristikám daného vstupního bodu. Tím je v rámci této části aplikace docílena požadovaná modularita řešení, kdy je možné jednotlivé vstupní body odebírat či přidávat bez zásahu do implementace zbytku aplikace. V průběhu psaní této práce byly postupně úspěšně zapojeny 4 nezávislé vstupní body, které dohromady obsahovaly přibližně 50 různých IoT zařízení.

Další částí aplikace je část prezentační. Tato část aplikace umožňuje realizovat dynamické vykreslování vybraných dat uživateli a zároveň poskytuje možnost vyvolání specifických požadavků pro funkcionální část aplikace. Veškerý obsah aplikace se vykresluje na základě poskytnutého konfiguračního souboru, ve kterém jsou zachyceny dílčí řídící parametry. Prezentační část aplikace tak může vykreslovat data (teoreticky) libovolného množství vstupních bodů bez nutnosti zásahů do aktuálního zdrojového kódu. Případné úpravy zdrojového kódu by se týkaly pouze přizpůsobení vykreslování prezentačních prvků v rámci nové kategorie vstupních bodů (vstupních bodů, které by využívaly zatím nepoužitou metodu přenosu dat).

Tyto dvě výše zmíněné části aplikace propojuje poslední, transakční část aplikace, která zajišťuje distribuci a třídění systémových zpráv napříč zbytkem aplikace. Zároveň je zde implementována základní vrstva zabezpečení, která zamezuje neoprávněnému přístupu k back-endu aplikace skrze prezentační část.

Ačkoliv lze na tyto tři části aplikace nahlížet jako na jeden celek, každou z daných částí lze upravovat separátně bez nutnosti zásahu do jiných částí aplikace. Funkcionální část aplikace navíc může být v provozu nezávisle na stavu prezentační a transakční části. Ve výsledku tedy lze konstatovat, že stěžejní částí pro projekt datové platformy je v rámci této práce

funkcionální část aplikace. Na zbylé dvě části lze pohlížet jako na doplňující software, který plní spíše sekundární cíle (především monitoring a poskytnutí ručních zásahů do automatizovaného procesu).

5.1 Dílčí požadavky a limitace

Před vypracováním praktické části práce byly vzneseny požadavky a definovány určité limitace, kterým bylo potřeba se přizpůsobit.

Požadavky a zároveň limitace v podobě nutnosti zhotovení aplikace v prostředí NodeRED a využití programovacího jazyka Javascript byly naplněny v plné míře. Ke spuštění a běhu aplikace v aktuální chvíli postačuje jedna instance NodeRED prostředí.

Požadavek na funkcionalitu front-end části aplikace byl taktéž naplněn v plné míře. Jak bylo zmíněno výše, prezentační část aplikace je schopna poskytnout veškerá potřebná data související se synchronizací záznamů zařízení a zároveň podporuje vyvolání ručních akcí v podobě zadání vlastního intervalu pro ruční synchronizaci či provedení resetování stavu konkrétních vstupních bodů či dat v kontextovém úložišti.

Splnění požadavku na vhodnou transformaci dat do stanovené výstupní podoby bylo nedílnou součástí implementace jednotlivých vstupních bodů. Každý ze vstupních bodů, který byl při psaní této práce implementován, tento požadavek splňuje.

Posledním zmíněným požadavkem byla dostatečná modularita aplikace, které bylo, na základě teoretických poznatků a průběžných konzultací s pověřenými osobami spolupracujícími na projektu datové platformy, taktéž docíleno. Tato skutečnost je zároveň demonstrována v následujících kapitolách.

5.2 Nasazení a další vývoj aplikace

Zdrojový kód byl v několika etapách postupně nahrán na produkční server univerzity, kde následně probíhal několikaměsíční provoz, v průběhu kterého byly postupně laděny podrobnější aspekty projektu. V době psaní této části práce je zdrojový kód aplikace nasazen v ostrém provozu více než půl roku, přičemž je skrze něj realizováno stahování a transformace dat vybraných zařízení.

Taktéž bylo rozhodnuto, že se tento prototyp vstupního uzlu datové platformy osvědčil a nadále se pracovalo (a bude pracovat) na jeho dalších úpravách a transformacích. Vzhledem k tomu, že se ve výsledku jedná o stále se vyvíjející část projektu datové platformy univerzity, která je průběžně aktualizována dle nových požadavků, autor této práce se rozhodl v praktické části práce nezachycovat aktuální stav aplikace, nýbrž stav takový, který byl na počátku definován úvodními požadavky. V následujícím textu jsou stroze zachyceny vybrané změny aplikace, které jsou buď ve fázi testovacího provozu či ve fázi návrhu a které dále rozšiřují či upravují funkcionalitu aplikace na základě nových, dodatečných požadavků.

5.2.1 Technologie LoRaWAN a centrální databáze překladačů zařízení

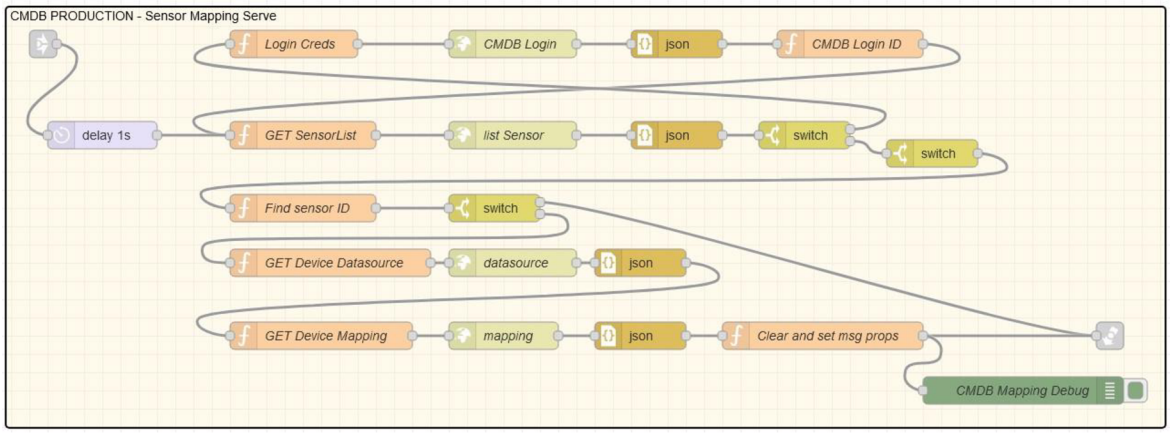
Jak bylo zmíněno v kapitole *4.5.5 (Realizace řešení: Vstupní bod LoRaWAN)*, zpracování příchozích dat v rámci daného vstupního bodu se zpravidla liší u každého typu zařízení, které je do školní sítě LoRaWAN zapojeno. Do budoucna je plánováno skrze tuto strukturu připojit větší množství nových zařízení a předpokládá se, že se daná struktura stane stěžejní pro příjem většinového podílu dat zapojených zařízení. To vedlo k definování úprav funkcionální části aplikace, jelikož stávající struktura přístupového bodu LoRaWAN byla vyhodnocena jako dlouhodobě neudržitelná a příliš náročná na údržbu. Kvůli tomu byly navrženy a implementovány dodatečné úpravy, které jsou popsány níže. Díky vhodné modularizaci funkcionální části aplikace tyto změny nijak nenarušily fungování zbytku vstupních bodů či jiných částí aplikace.

Princip nové implementace

Při příjmu zprávy skrze vstupní bod LoRaWAN je na základě jedinečného identifikátoru zařízení nejprve nahlédnuto do kontextového úložiště NodeRED, kde proběhne pokus o vyhledání skriptu překladače určeného pro dané konkrétní zařízení. Daný skript obsahuje veškeré strojové instrukce, které jsou zapotřebí k překladu přijatých dat ze surové do výstupní podoby (narozdíl od předešlé implementace každý z překladačů obsahuje i vlastní metodu pro transformaci hodnot do požadované podoby pro datovou platformu). Pokud je skript překladače nalezen, zpráva může být přeložena a zpracována.

Pokud skript překladače nalezen není, je vyvolán požadavek na centrální databázi, ve které jsou uchovány informace ohledně jednotlivých zařízení. Nejprve je v databázi nalezena

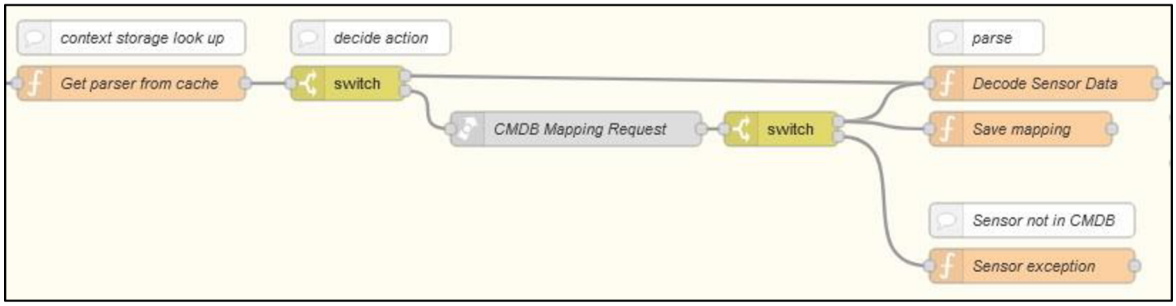
datová instance pro konkrétní zařízení, ve které je mimo jiné nalezen klíč, který odkazuje na konkrétní skript překladače (který je s tímto zařízením v databázi spárován). K realizaci komunikace s centrální databází byla v prostředí NodeRED realizována dodatečná struktura, viz následující obrázek.



Obrázek 34 - náhled na dodatečně realizovanou strukturu v NodeRED; Zdroj: vlastní

Skript překladače je následně společně s dalšími důležitými parametry zařízení odeslán v odpovědi zpět do funkcionální části aplikace, kde jsou pro aktuálně zpracovávané zařízení uloženy řídící parametry, včetně skriptu překladače pro budoucí zpracovávání. Ukládání do místního úložiště NodeRED je realizováno především z důvodu úspory síťové komunikace (některá zařízení mohou posílat data například každých 5 minut a v takovém případě by bylo velmi neefektivní při každém zpracování zprávy opětovně stahovat celý skript překladače pro dané zařízení).

Pokud byl skript překladače úspěšně nalezen či stažen, je spuštěn skrze vhodný funkcionální uzel nově upravené struktury realizující transformaci zpráv přístupového bodu LoRaWAN, viz obrázek a ukázka zdrojového kódu dále.



Obrázek 35 - ukázka nové struktury vstupního bodu LoRaWAN; Zdroj: vlastní

```
if(msg._dataMapping == null){
    msg.sensorOutputData = msg.sensorMsgData;
    return msg;
} else {
    let decoderFunction = new Function("input", msg._dataMapping);
    let _output = decoderFunction(msg.sensorMsgRaw);

    _output.source = msg._vendorCode + '.' + msg._vendorID;
    msg.sensorOutputData = _output;
    return msg;
}
```

Zdrojový kód 28 - ukázka uzlu načítajícího skript překladače; Zdroj: vlastní

Výstupem jsou data, která lze přesměrovat na výstup do datové platformy. Tato změna funkcionality efektivně řeší problém s nutností aplikování různorodých překládacích metod pro jednotlivé typy zařízení. V rámci připojení nového zařízení stačí provést jeho registraci v centrální databázi a přiřazení vhodného skriptu překladače. Pokud má být připojeno zařízení nového, zatím nepoužívaného typu, je tento proces rozšířen o sepsání a vložení na míru upraveného skriptu překladače do centrální databáze dle definované šablony, doporučených postupů a poskytnuté dokumentace k zařízení.

Ve výsledku tím bylo docíleno stavu, kdy při zapojování nového zařízení komunikujícího na technologii LoRaWAN není potřeba provádět žádné dodatečné úpravy v rámci funkcionální části aplikace, což velmi zjednodušilo celý proces zapojení nových zařízení a zvýšilo míru udržitelnosti řešení. Tato implementace je v době psaní této části práce v pokročilé fázi procesu testování, přičemž bude pravděpodobně následovat její širší nasazení.

Zároveň lze v budoucnu implementovat automatizované čištění dat kontextového úložiště, při kterém by byly mazány předpřipravené (stažené) informace o zařízeních tohoto vstupního bodu. Pro aktivní zařízení by se poté při jejich následující komunikaci opět stáhla potřebná data z centrální databáze, mezitím co neaktivní zařízení by již nezabírala místo v kontextovém úložišti NodeRED.

5.2.2 Širší využití nové implementace vstupního bodu LoRaWAN

Generická struktura překladače popsána v předešlé kapitole by mohla být použita i v případě realizace zbylých vstupních bodů, nicméně zde vyvstává několik komplikací.

První komplikací je nutnost udržení jednoznačné a unikátní identifikace jednotlivých zařízení napříč vícero vstupními body. Některé aktuálně zapojené vstupní body ovšem v zasílaných datech nezahrnují jednoznačnou identifikaci zařízení, případně je unikátní označení zařízení součástí zakódovaných datových struktur, tudíž není možné identifikovat zařízení před samotným překladem takových dat, což není slučitelné s aktualizovanou implementací generického překladače, který nyní struktura vstupního bodu LoRaWAN využívá.

Druhou komplikací jsou příchozí zprávy obsahující data k vícero zařízením či vícero časovým značkám, viz kapitola *4.5.4 Realizace řešení: Vstupní bod typu http GET*. Generická struktura překladače byla navržena pro zprávy obsahující data vztahující se k jednomu zařízení a jedné časové značce.

Tyto komplikace lze řešit například realizací takzvaného "preprocesoru zpráv". Preprocesor by fungoval na velmi podobném principu jako samotný generický překladač dat. Pro zpracování příchozí zprávy by byl opět využit skript, jehož obsah by byl na míru upraven konkrétnímu vstupnímu bodu a který by mohl být uložen v centrální databázi zařízení, stejně jako skripty překladače. Obsahem tohoto skriptu preprocesoru by byla logika zajišťující jednoznačnou identifikaci zařízení a především dělení příchozích zpráv na dílčí zprávy vztahující se k jednomu zařízení a jedné časové značce.

Poté, co by byla zpráva zpracována ve struktuře preprocesoru zpráv, byla by standardně odeslána do již zhotovené generické struktury překladače, kde by byla dále zpracovávána.

6 Závěr

Cílem práce bylo vytvoření struktury a realizace funkcionality vstupního uzlu pro datovou platformu univerzity, skrze který by mohla být přijímána naměřená data z různorodých IoT zařízení. Dosažení tohoto cíle bylo podmíněno dodržením předem stanovených limitací a přizpůsobením výsledného řešení konkrétním požadavkům. Navržením a následnou realizací řešení, které je zachyceno v kapitole pojednávající o praktické části práce, byl tento cíl naplněn v plném rozsahu včetně dílčích cílů.

V rámci teoretické části práce jsou čtenáři poskytnuty základy problematiky různých softwarových odvětví a technologií, které byly součástí následného vypracování vlastní práce. V kontextu této práce se jednalo především o principy realizace zdrojového kódu, základy dynamických webových stránek a technologie pro jejich realizaci (Javascript, Vue.js, NodeRED a další).

Během vypracovávání řešení vlastní práce byly k dosažení cílů využity nabyté teoretické poznatky, které vycházely ze zdrojů či informací uvedených v teoretické části práce. Zpracování práce bylo zaměřeno především na funkční stránku aplikace, zejména na distribuci a zpracování různorodých dat. V první části vlastní práce byla rozvržena struktura aplikace na 3 dílčí části. Každá z těchto dílčích částí byla následně jedna po druhé implementována. Vybrané části aplikace a konkrétní postupy zpracování dat byly popsány v jednotlivých kapitolách takovým způsobem, aby byla daná problematika zachycena v odpovídajícím rozsahu úměrném náročnosti daných dílčích úkolů.

Zdrojový kód a výstupy, které vznikly v průběhu vypracovávání této práce, byly následně testovány a úspěšně začleněny do projektu datové platformy ČZU, kde nadále dochází k úpravám a dalšímu vývoji této části projektu. Začlenění aplikace do ekosystému datové platformy lze považovat za důkaz validního zpracování zadaných cílů práce. Na základě toho se vývojové prostředí NodeRED, na kterém byla tato práce vystavěna, jeví jako vhodné vývojové prostředí pro realizaci projektů z oblasti IoT.

7 Seznam použitých zdrojů

1. IBM - VIRTUALIZACE. Www.ibm.com [online]. [cit. 2023-11-29]. Dostupné z: https://www.ibm.com/topics/virtualization

2. DOCKER ENGINE. Docs.docker.com [online]. [cit. 2023-11-29]. Dostupné z: https://docs.docker.com/

3. IOT - INTERNET VĚCÍ. Www.oracle.com [online]. [cit. 2023-11-29]. Dostupné z: https://www.oracle.com/internet-of-things/what-is-iot/

4. NODERED - VÝVOJ V OBLASTI IOT. Nodered.org [online]. [cit. 2023-11-29]. Dostupné z: https://nodered.org/

5. DOKUMENTACE NODERED - STRUKTURA TOKU DAT. Nodered.org/docs [online]. [cit. 2023-11-29]. Dostupné z: https://nodered.org/docs/developing-flows/flow-structure

6. MCFEDRIES, Paul. Web Coding & Development All-in-One For Dummies (1. edice), elektronická verze [online]. In: . John Wiley & Sons, 2018, Kniha 7, Kapitola Planning a web app, strana 1 [cit. 2023-11-29]. ISBN 9781119473923.

7. MCFEDRIES, Paul. Web Coding & Development All-in-One For Dummies (1. edice), elektronická verze [online]. In: John Wiley & Sons, 2018, Kniha 7, Kapitola Planning a web app, strany 2 - 6 [cit. 2023-11-29]. ISBN 9781119473923.

8. BRITANNICA - DEFINICE HTML. Www.britannica.com [online]. [cit. 2023-11-29]. Dostupné z: https://www.britannica.com/technology/HTML

9. MOZILLA.ORG - DEFINICE KASKÁDOVÝCH STYLŮ. Developer.mozilla.org [online]. [cit. 2023-11-29]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/CSS/First_steps/What_is_CSS

10. RANJAN Alok, Abhilasha SINHA a Ranjit BATTEWAD. JavaScript for Modern Web Development , elektronická verze [online]. In: BPB Publications, 2020, strana 313 [cit. 2023-11-29]. ISBN 9789389328721.

11. RANJAN Alok, Abhilasha SINHA a Ranjit BATTEWAD. JavaScript for Modern Web Development , elektronická verze [online]. In: BPB Publications, 2020, strana 309 [cit. 2023-11-29]. ISBN 9789389328721.

12. RANJAN Alok, Abhilasha SINHA a Ranjit BATTEWAD. JavaScript for Modern Web Development , elektronická verze [online]. In: BPB Publications, 2020, strana 312 - 313 [cit. 2023-11-29]. ISBN 9789389328721.

13. MOZILLA.ORG - FORMÁT JSON. Developer.mozilla.org [online]. [cit. 2023-11-29]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

14. NODEJS.ORG - PROSTŘEDÍ NODE.JS. Nodejs.org [online]. [cit. 2023-11-29]. Dostupné z: https://nodejs.org/en/about

15. VUEJS.ORG - JAVASCRIPT FRAMEWORK VUEJS. Vuejs.org [online]. [cit. 2023-11-29]. Dostupné z: https://vuejs.org/

16. VUEJS.ORG - DOKUMENTACE - FUNKCIONÁLNÍ KOMPONENTY. Vuejs.org [online]. [cit. 2023-11-29]. Dostupné z: https://v2.vuejs.org/v2/guide/render-function#Functional-Components

17. VUEJS.ORG - ZÁKLADNÍ STRUKTURA VUE.JS KOMPONENTY. Vuejs.org [online]. [cit. 2023-11-29]. Dostupné z: https://v2.vuejs.org/v2/guide/components

18. VUEJS.ORG - DEFINOVÁNÍ SDÍLENÝCH VLASTNOSTÍ. Vuejs.org [online]. [cit. 2023-11-29]. Dostupné z: https://vuejs.org/guide/components/props.html

19. W3SCHOOLS.COM - FUNKCE EMIT(). *W3Schools.com* [online]. [cit. 2024-01-04]. Dostupné z: https://www.w3schools.com/vue/vue_emit.php

20. VUEJS.ORG - PODMÍNĚNÉ VYKRESLOVÁNÍ OBSAHU. Vuejs.org [online]. [cit. 2023-11-29]. Dostupné z: https://vuejs.org/guide/essentials/conditional.html

21. VUEJS.ORG - ITERATIVNÍ VYKRESLOVÁNÍ OBSAHU. Vuejs.org. Online. Dostupné z: https://vuejs.org/guide/essentials/list.html [cit. 2023-11-29].

22. MICROSERVICES.IO - POPIS PRINCIPU MICROSERVICES. Microservices.io [online]. [cit. 2023-11-29]. Dostupné z: https://microservices.io/patterns/microservices.html

23. UNIXOVÁ ČASOVÁ ZNAČKA. Unixtimestamp.com [online]. [cit. 2023-11-29]. Dostupné z: https://www.unixtimestamp.com/

24. OMG.ORG - SPECIFIKACE BPMN 2.0. Omg.org [online]. [cit. 2023-11-29]. Dostupné z: https://www.omg.org/spec/BPMN/2.0/

25. NODERED - CRON-PLUS ROZŠÍŘENÍ. *Npmjs.com* [online]. [cit. 2024-01-22]. Dostupné z: https://www.npmjs.com/package/node-red-contrib-cron-plus

26. NODERED ROZŠÍŘENÍ "UIBUILDER." Flows.nodered.org [online]. [cit. 2023-11-29]. Dostupné z: https://flows.nodered.org/node/node-red-contrib-uibuilder

27. VUE ROUTER. Router.vuejs.org [online]. [cit. 2024-01-20]. Dostupné z: https://router.vuejs.org/

28. HTTP-VUE-LOADER. *Npmjs.com* [online]. [cit. 2024-01-20]. Dostupné z: https://www.npmjs.com/package/http-vue-loader

29. BOOTSTRAP VUE. *Bootstrap-vue.org* [online]. [cit. 2024-01-20]. Dostupné z: https://bootstrap-vue.org/

8 Seznam obrázků a textových polí

8.1 Seznam obrázků

8.2 Seznam textových polí

9 Přílohy

Vytvořené skripty a soubory obsahující zdrojový kód projektu jsou dostupné v přiloženém .zip souboru, případně na poskytnutém paměťovém médiu. Ačkoliv tyto soubory tvoří funkční celek zaměřené části projektu datové platformy, pro spuštění je nutno využít instance běhového prostředí NodeRED, jež vyžaduje konfiguraci odpovídající tomuto konkrétnímu projektu. Zdrojový kód v přiložených souborech je upravenou verzí produkčního zdrojového kódu, přičemž přiložená verze zdrojového kódu se liší především v absenci přihlašovacích údajů či v absenci konkrétní konfigurace připojení k dostupným přístupovým bodům (API). Úpravy byly provedeny kvůli minimalizaci bezpečnostního rizika. Funkčnost zdrojového kódu lze ovšem stále ověřit na testovacích datech a strukturách, které jsou součástí této verze zdrojového kódu.