



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Webové prostředí pro správu a konfiguraci IoT projektů
Student:	Bc. Jakub Vejr
Vedoucí:	Ing. Stanislav Vítek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1. Navrhňte webovou aplikaci, která umožní správu IoT projektů založených na sensorových sítích. Aplikace bude obsahovat klientskou část, serverovou část a bude implementováno REST API pro správu. Součástí aplikace budou také nástroje pro vizualizaci dat.
2. Aplikace může pracovat s různými datovými formáty: JSON, strukturovaná textová informace nebo binární data. Pro všechny datové formáty budou navrženy mapovací funkce a filtry.
3. Aplikace má víceúrovňový uživatelský přístup o nejméně dvou úrovních. Administrátor může konfigurovat veškerá nastavení a přístup pro další uživatele, běžný uživatel má přístup k nasbíraným datům z vybraných senzorů.
4. Pro implementovanou aplikaci a její API vytvořte integrační a unit testy. Pokud to bude možné, pokryjte testy i klientskou část.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 11. února 2019



**FAKULTA
INFORMAČNÍCH
TECHNOLGIÍ
ČVUT V PRAZE**

Diplomová práce

Webové prostředí pro správu a konfiguraci IoT projektů

Bc. Jakub Vejr

Katedra softwarového inženýrství

Vedoucí práce: Ing. Stanislav Vítek, Ph.D.

4. května 2019

Poděkování

Rád bych poděkoval Ing. Stanislavu Vítkovi, Ph.D za vedení práce a všem lidem, kteří mne při vypracování této práce podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

Prague dne 4. května 2019

.....

České vysoké učení technické v Praze
Fakulta informačních technologií

© 2019 Jakub Vejr. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Vejr, Jakub. *Webové prostředí pro správu a konfiguraci IoT projektů*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019. Dostupný také z WWW: (<https://github.com/vejrak/jv-dp>).

Abstrakt

Diplomová práce se zabývá návrhem a implementací webové aplikace pro správu a konfiguraci IoT projektů. Její dílčí cíle jsou umožnit, aby aplikace mohla shromažďovat data z IoT senzorů, ukládat tato data a následně je prezentovat uživatelům aplikace. Řešení bylo naprogramováno v jazyku JavaScript za použití frameworků Next.js a Express.

Klíčová slova IoT, shromažďování dat, webová aplikace, Node.js, Next.js

Abstract

The goal of this master thesis is to design and implement a web application for managing and configuration IoT projects. The partial goal is to enable collecting and saving data from IoT sensors and present these data to the application users. The result was developed in JavaScript language using Next.js and Express frameworks.

Keywords IoT, data collecting, web application, Node.js, Next.js

Obsah

Úvod	1
1 Rešerše	3
1.1 Existující platformy	3
1.2 Závěr	8
1.3 Cíl práce	8
2 Internet of things	9
2.1 Typy senzorů	9
2.2 Komunikace	10
3 Návrh řešení a použité technologie	11
3.1 Návrh řešení	11
3.2 Uživatelé a jejich oprávnění	13
3.3 Použité technologie	14
3.4 Použité knihovny	17
4 RESTful API	21
4.1 Prostředí	21
4.2 Návrh modelů	25
4.3 Routy a controllery	29
4.4 Přihlašování	31
4.5 Mappery	34
4.6 Použití mapperů	36
4.7 Poskytování dat	37
5 Frontend aplikace	39
5.1 Prostředí	39
5.2 Adresářová struktura	42
5.3 Stránky	43

5.4	Komponenty a jejich organizace	44
5.5	Prezentace dat	44
6	Testování	47
6.1	Testování backend aplikace	47
6.2	Testování frontend aplikace	50
6.3	Test rychlosti	52
	Závěr	55
	Literatura	57
	A Seznam použitých zkratk	59
	B Obsah příloženého CD	61

Seznam obrázků

1.1	Ukázka UI thinger.io	4
1.2	Ukázka grafu v thinger.io	4
1.3	Ukázka IBM Watson	5
3.1	Návrh aplikace	12
3.2	Struktura aplikace	13
3.3	Důležité části Node.js [1]	14
3.4	Průběh CI u aplikace	16
3.5	React lifecycle metody [2]	18
3.6	Redux	19
4.1	Databázové schéma skupin	25
4.2	Databázové schéma uživatelů	26
4.3	Databázové schéma metrik a jednotek	27
4.4	Databázové schéma senzoru	28
4.5	Databázové schéma dat	28
4.6	JWT proces	31
4.7	Databázové schéma JsonMapperu	34
4.8	Databázové schéma BinaryMapperu	35
4.9	Databázové schéma BinaryMixedMapperu	36
5.1	Ukázka vyhledávače	45
5.2	Ukázka detailu senzoru	45
5.3	Ukázka mapy s detailem senzoru	46
6.1	Přihlašovací komponenta	50

Seznam tabulek

1.1 Shrnutí řešení	7
------------------------------	---

Úvod

Aplikace se zabývá řešením problému, kdy uživatel má k dispozici sadu senzorů, ze kterých by chtěl hromadně shromažďovat data. Každý z těchto senzorů může zasílat data v různých formátech, jako jsou JSON nebo text, či dokonce v binární podobě a mnoho dalších. Hodnoty těchto dat se dále mohou lišit v metrikách nebo pouze v jejich jednotkách. Lze očekávat velký nárůst množství těchto dat, a tím pádem bylo třeba uzpůsobit návrh tomu, aby nedocházelo s postupem času k poklesu výkonu. Aplikace dále řeší, jak tato data přehledně zobrazit. K tomu se využívají grafy, mapa nebo různé statistiky. Dále je umožněno, aby tato data mohla být zobrazena i jinými uživateli. Každý uživatel vidí ale pouze data, ke kterým mu dá správce přístup. Uživatel není oprávněn jakkoliv měnit nastavení aplikace. Správci je umožněno přidávat či měnit senzory, metriky a její jednotky.

Rešerše

Po zmapování prostředí platforem nabízejících správu IoT produktů byla nalezena již existující řešení, avšak s různými poli působnosti. Tato práce se zabývá hlavně zpracováním dat z IoT senzorů, čemuž je uzpůsobeno i zkoumání podobných platforem.

1.1 Existující platformy

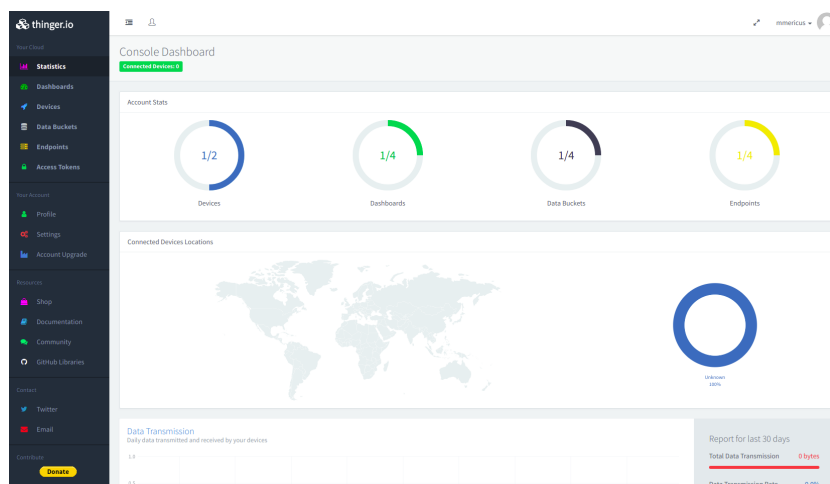
1.1.1 Thinger

Platforma je poskytována volně ke stažení. Pro využití jejich hardware a údržby je nutné platit [3]. Zdrojové kódy serverové části nejsou zveřejněny. Dále je možné nainstalovat mobilní aplikaci nebo klientskou část pro sledování dat. U klientské části a mobilní aplikace jsou již zdrojové kódy zveřejněny na GitHub.

Platforma umožňuje spravovat a shromažďovat data ze senzorů. Lze se propojit přímo se senzorem a komunikovat s ním obousměrně a o data si žádat. Bohužel však umožňuje zpracovávat data pouze ve formátu JSON. Dále není umožněn převod jednotek. Uživatel si pouze vytvoří widget, kde definuje, jaká jednotka ze senzoru přichází. Sensory se mohou zanášet pomocí koordinát na mapu a zaznamenat tak jeho přesnou polohu. Dále je umožněno data vkládat do takzvaných *data-buckets*. *Data-buckets* je virtuální úložiště, kam uživatel může uložit sérii nebo část dat (například teplotu), kterou poté může zobrazovat v grafu nebo exportovat.

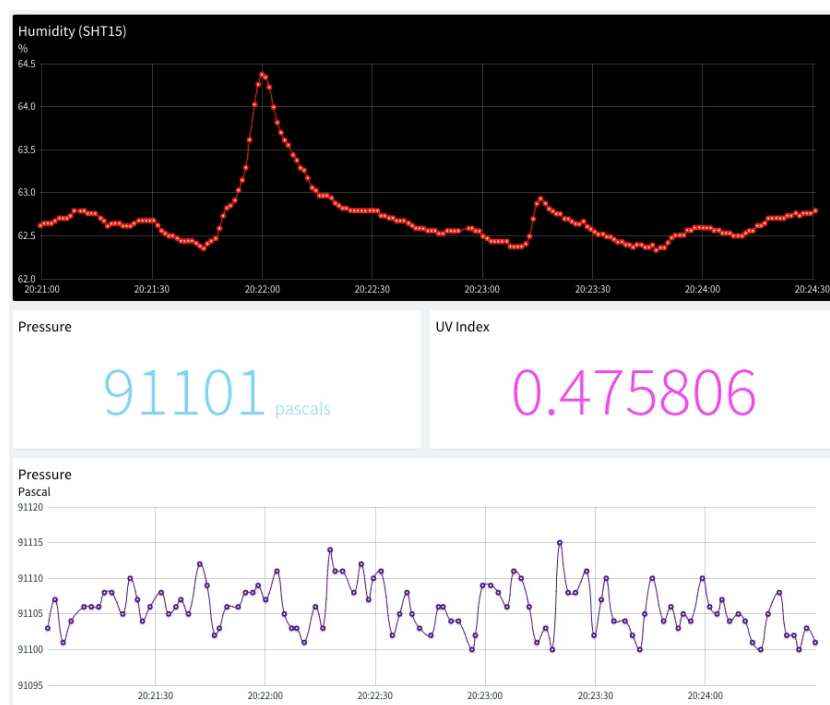
1. REŠERŠE

Obrázek 1.1: Ukázka UI thinger.io



Platforma je přizpůsobena práci v real-time režimu, kdy je možné například sledovat vývoj vlhkosti vzduchu živě, jako je na následujícím obrázku.

Obrázek 1.2: Ukázka grafu v thinger.io



Výhody

- Řešení přímo pro shromažďování a prezentaci dat.
- Možnost nainstalovat na vlastní servery.
- Při zaplacení zákaznická podpora a údržba.

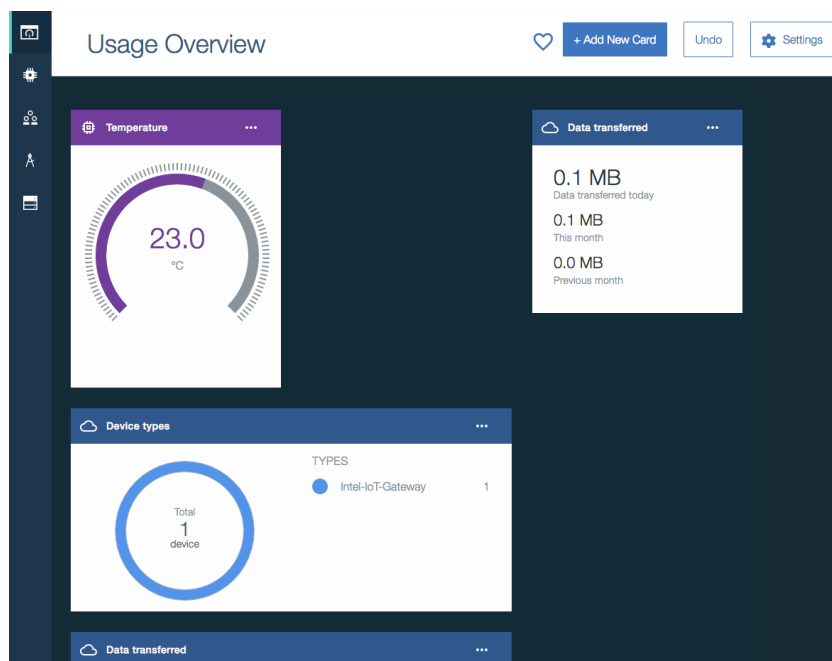
Nevýhody

- Zdrojové kódy pro server nejsou dostupné a nelze je tedy modifikovat.
- Není možná konverze jednotek.
- Podpora pouze JSON formátu.

1.1.2 IBM Watson IoT Platform

Po dobu 30 dní je možné využívat platformu v omezeném režimu. Po uplynutí této doby je nutné za platformu platit podle požadované funkcionality a kapacity [4]. Platforma umožňuje spravovat senzory a zařízení, rozdělovat je do skupin a mnoho dalšího. Umožňuje zpracovávat data v binárním, textovém a JSON formátu [5].

Obrázek 1.3: Ukázka IBM Watson



Výhody

- Umožňuje zpracovávat více formátů (JSON, binární a textový) [5].
- Udržovaná platforma od známé a stabilní společnosti.
- Poskytování široké funkcionality v oblasti IoT. Nezaměřuje se pouze na senzory, ale i na ostatní zařízení.
- Lze očekávat neustálý vývoj a rozšiřování.
- Zákaznická podpora.

Nevýhody

- Placená služba.
- Nelze volně rozšiřovat a modifikovat.

1.1.3 Amazon

Po dobu 12 měsíců je možné platformu zdarma používat v omezeném režimu. Po uplynutí této doby se platí za provoz, kapacitu a funkcionality, která převyšuje zdarma poskytované služby [6]. Umožňuje přidávat senzory a ostatní zařízení do skupin a spravovat je. Data jsou zpracovávána v JSON formátu [7].

Výhody

- Udržovaná platforma od známé a stabilní společnosti.
- Poskytování široké funkcionality v oblasti IoT. Nezaměřuje se pouze na senzory, ale i na ostatní zařízení.
- Lze očekávat neustálý vývoj a rozšiřování.
- Zákaznická podpora.

Nevýhody

- Placená služba.
- Nelze volně rozšiřovat a modifikovat.
- Neumožňuje zpracovávat jiný formát než JSON [7].

1.1.4 Azure IoT Hub

Tato platforma umožňuje zdarma vyzkoušení oblíbené funkcionality po dobu 12 měsíců. Po dovršení této doby je placená [8]. Podobně jako předešlá korporátní řešení umožňuje širokou funkcionalitu v oblasti IoT od její správy až po prezentaci dat. Podporuje však pouze JSON formát pro zpracování dat ze senzorů.

Výhody

- Udržovaná platforma od známé a stabilní společnosti.
- Poskytování široké funkcionality v oblasti IoT. Nezaměřuje se pouze na senzory, ale i na ostatní zařízení.
- Lze očekávat neustálý vývoj a rozšiřování.
- Zákaznická podpora.

Nevýhody

- Podporuje pouze JSON formát pro zpracování.
- Placená služba.
- Nelze volně rozšiřovat a modifikovat.

Tabulka 1.1: Shrnutí řešení

Platformy	Podporované formáty	Cenová dostupnost	Volně dostupné zdrojové kódy
Thingier	JSON	Zdarma (při použití vlastního hardware) jinak placené	Ne
IBM Watson	JSON, textový a binární	Placené	Ne
Amazon	JSON	Placené	Ne
Azure IoT Hub	JSON	Placené	Ne

1.2 Závěr

Žádné z nalezených řešení neposkytuje požadovanou funkcionalitu jako open-source a jejich plná verze je dostupná pouze za poplatek. Některé z nich umožňují zdarma jejich využívání v omezeném režimu. Jiné zase neposkytují potřebné funkcionalitu v oblasti zpracování dat. Tato práce se bude zabývat hlavně zpracováním dat IoT senzorů různých formátů a bude poskytována jako open-source. Každý si bude moci aplikaci nainstalovat na vlastní server a volně ji modifikovat.

1.3 Cíl práce

Kombinací zadání s provedenou analýzou bylo rozhodnuto, že výsledná aplikace musí splňovat následující body.

- Měla by mít webové prostředí.
- Volně dostupná.
- Snadno modifikovatelná.
- Komunikovat pomocí HTTP protokolu a zpracovávat požadavky ze senzorů nebo jiných platforem z oblasti IoT.
- Mít uživatelský systém umožňující správcům nastavovat a řídit aplikaci a uživatelům zobrazovat data.
- Umožňovat vkládat a spravovat senzory.
- Umožnit vytvářet metriky a jednotky.
- Umožnit definici zpracování příchozích požadavků dle typů formátů (text, JSON, binární).
- Umožnit zanášení senzorů do mapy.
- Zobrazovat data ve grafu.
- Možnost exportovat data do CSV nebo podobného formátu.
- Data budou zobrazena jen uživatelům, kteří dostanou oprávnění.
- Ovládání aplikace pomocí interaktivního UI.

Internet of things

Internet of things (Internet věcí) je nastupující trend digitalizování nejrůznějších fyzických zařízení, která jsou vybavena senzory, elektronikou, softwarem a síťovými prvky. Tato zařízení dokážou mezi sebou komunikovat a vyměňovat si data. Využívají se pro optimalizaci procesů a pro výzkum.

2.1 Typy senzorů

V organizacích a průmyslu se používá celá řada typů senzorů. Každý z těchto senzorů snímá různé metriky [9].

- Teplotní senzory - Zařízení používané k měření tepelné energie, což umožňuje detekovat fyzické změny v teplotě konkrétního zdroje energie a poskytuje tyto údaje zařízení nebo uživateli.
- Sensory pro měření tlaku - Zařízení, které snímá tlak a konvertuje ho do elektrického signálu. Jeho síla závisí na úrovni nasnímaného tlaku.
- Sensory měření kvality vody - Tento typ senzoru slouží k měření kvality vody v potrubí, vodních zdrojích, atd. Sensory se dále dělí podle toho, jak a co ve vodě v rámci její kvality sledují.
- Sensory pro měření koncentrace chemických látek - Tyto senzory jsou používány v nejrůznějších odvětvích. Jejich hlavním cílem je pozorovat chemické změny (například ve vzduchu). Výsledky mohou pomáhat sledovat prostředí ve velkých městech a chránit tak obyvatelstvo.
- Sensory pohybu - Elektrické zařízení pro snímání fyzického pohybu. Pohyb jakéhokoliv objektu nebo člověka je převáděn na elektrický signál. Lze využít například pro zajištění bezpečnosti.
- Gyroskopy - Senzor detekující úhel a rychlost, jakou je vykonáván pohyb.

- Akcelerometr - Senzor využívající se k měření fyzického zrychlení objektu, které bylo způsobeno nějakou inicializační silou. To je poté převedeno opět na elektrický signál.
- IR senzory - Tento senzor slouží k měření infračerveného záření, které vyzařují okolní objekty. Lze pomocí něho snímat tepelnou energii, kterou vyzařují objekty.
- Sensory kouře - Snímají koncentraci kouře v prostředí. Používají se v průmyslu i v oblasti bezpečnosti běžného života (například v domácnosti). Při detekci určité koncentrace lze spouštět bezpečnostní mechanismy jako například zavolání pomoci.
- Obrazové senzory - Jsou využívány k převádění obrazu na elektrický signál. Používají se v kamerách, v lékařství, ve vybavení pro noční vidění.
- Sensory vlhkosti - Sledují množství vodní páry obsažené ve vzduchu nebo jiných plynech.
- Optické senzory - Měří množství světelných paprsků a převádí je na elektrický signál, který lze snadno číst uživatelem nebo zařízením.

2.2 Komunikace

Senzory se dále liší ve způsobu komunikace.

- Cellular
- Wifi
- NFC
- Bluetooth
- Z-Wave
- Zigbee
- RFID
- Smartdust
- MEMS
- TCP/IP
- HAN

Návrh řešení a použité technologie

Tato kapitola popisuje návrh řešení a technologie, pomocí kterých bude tento návrh implementován.

3.1 Návrh řešení

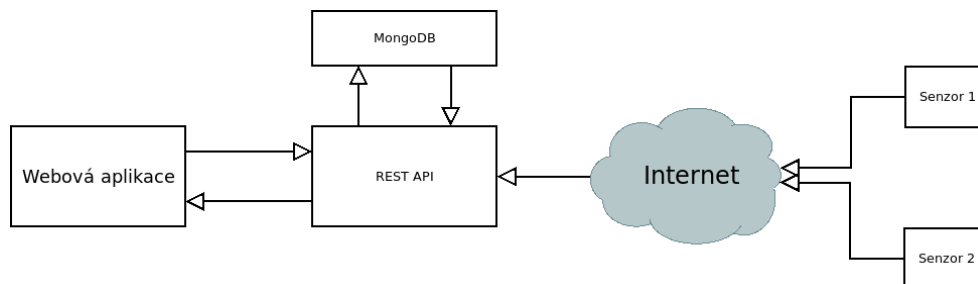
Dle cíle práce má být zhotovena aplikace, která bude mít dvě hlavní funkce. První bude shromažďování dat ze senzorů a jejich ukládání do databáze. Druhá hlavní funkce je filtrování těchto dat a jejich prezentace.

Na základě cíle práce bylo rozhodnuto, že pro komunikaci pomocí HTTP protokolu je vhodná architektura REST. REST je sada pravidel a principů, které pokud jsou dodržovány, umožní dosáhnouti dobrého designu a interoperability systému [10]. V REST architektuře se přistupuje k resource (zdroji) pomocí HTTP metod GET, PUT, PATCH, DELETE, POST. Každý zdroj je identifikován pomocí URI.

Z důvodů škálovatelnosti výkonu a lepší údržby kódu bylo rozhodnuto, že bude lepší mít dvě rozdělené aplikace. První aplikace bude shromažďovat a ukládat data, která poté bude jako RESTful API poskytovat frontendové aplikaci. Ta se bude starat o jejich prezentaci. Tímto rozdělením lze také získat větší svobodu při výběru technologií, jelikož nyní není nutné, aby aplikace běžely na stejném fyzickém stroji. Další potencionální výhodou je, že se shromážděná data dají poskytovat třetím stranám.

Vzhledem k nastupujícímu trendu bylo rozhodnuto, že obě aplikace budou napsány v JavaScriptu. To umožňuje, aby člověk se znalostí jednoho jazyka byl schopen spravovat obě aplikace. Dále je možné využívat stejné knihovny a zdrojové kódy na obou stranách a vyhnout se tak duplicitám kódu a jejich složité modifikaci. Zároveň se v aktuální době jedná o moderní a velmi podporovanou cestu. RESTful API psané pomocí Node.js totiž umožňuje celou

Obrázek 3.1: Návrh aplikace



řadu výhod, které budou popsány dále.

Obě aplikace budou využívat flow-type z důvodů větší kvality kódu a menší náchylnosti na nechtěné chyby. Flow-type totiž umožňuje, u jinak dynamicky typovaného jazyku JavaScript, deklarovat i typy. Dodržování jednotného stylu kódu bude hlídat knihovna ESLint.

3.1.1 Frontend aplikace

Jelikož bude aplikace využívána pouze lidmi z technického oboru, tak lze předpokládat, že nebude vytížena častými uživatelskými dotazy na server. Bylo proto rozhodnuto, že bude velmi výhodné napsat aplikaci pomocí Next.js a knihovny React. Next.js vykresluje obsah stránky na straně serveru (server-side rendering). React komponenty budou tedy předvykresleny už tehdy, když uživatel obdrží obsah webové stránky. Tím odpadá nevýhoda Reactu na pomalejších zařízeních, jelikož uživatel musel čekat, dokud prohlížeč neprovede vykreslení. Server side rendering má tedy za výhodu vyšší rychlost na mobilních zařízeních. Nevýhodou je, že se zátěž přesouvá od uživatele na server. Jak již bylo ale řečeno, u aplikace se nepředpokládá vysoká uživatelská zátěž.

Pro bundling je využít Webpack. Jedná se o aktuálně nejpoužívanější a technicky nejlepší variantu. Ke stylování webové stránky bude využita knihovna Bootstrap. Data budou vykreslena v grafech a jednotlivé senzory bude možné zanést do mapy s jejich přesnou polohou.

3.1.2 Databáze

Jako nejlepší databáze k výše zmíněným technologiím bylo zvoleno MongoDB. Lze očekávat veliké množství dat, které bude od senzorů shromažďováno. S tímto nárůstem je tedy nutné počítat a tím pádem i se snižováním výkonu při jejich zpracovávání. MongoDB jako NoSQL databáze dosahuje v tomto případě dobrých výsledků. Je nutné tomu ale přizpůsobit její návrh, jelikož nelze postupovat jako při návrhu relační databáze.

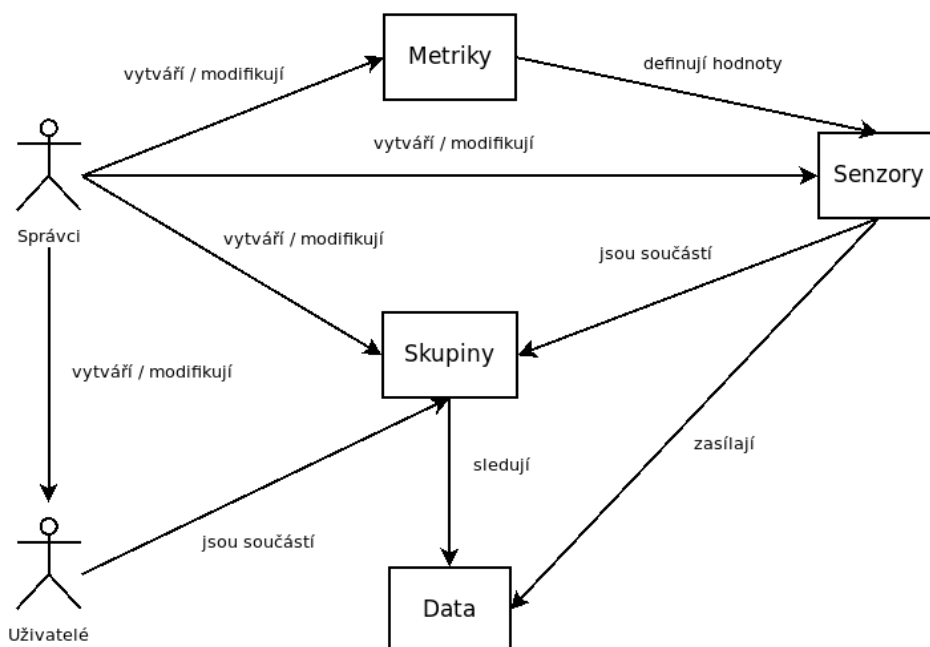
3.1.3 Backend aplikace

Backend aplikace bude realizována pomocí Node.js a frameworku Express. Data budou zapisována a čtena z MongoDB. Správci bude umožněno vytvářet metriky a jednotky, do kterých budou přichozí hodnoty řazeny. V rámci této práce se vypracují funkce, které dokážou zpracovávat požadavky ve formátech JSON, text a vybrané případy binárních dat. API bude poskytovat endpoint, kam budou moci služby a senzory zasílat svá data. Tato data bude zároveň možné převádět na stejné jednotky a filtrovat podle času vytvoření. Zároveň bude poskytovat stručný přehled a jednoduché statistiky doposud přijatých dat každého senzoru.

3.2 Uživatelé a jejich oprávnění

Uživatelské oprávnění bude víceúrovňové. Na jedné straně budou správci, kteří budou spravovat nastavení (senzorů, metrik, ostatních uživatelů a formáty přichozích dat), a na druhé straně uživatelé, kteří budou sledovat nashromážděné výsledky. Každý uživatel by však měl vidět pouze ta data, ke kterým mu dá správce oprávnění. Toto oprávnění bude ověřováno na obou stranách, tzn. frontendovou aplikací i API. Ověření bude realizováno pomocí tokenu, které API poskytne po přihlášení a které bude frontend aplikace posílat spolu s požadavky, které ho budou vyžadovat.

Obrázek 3.2: Struktura aplikace

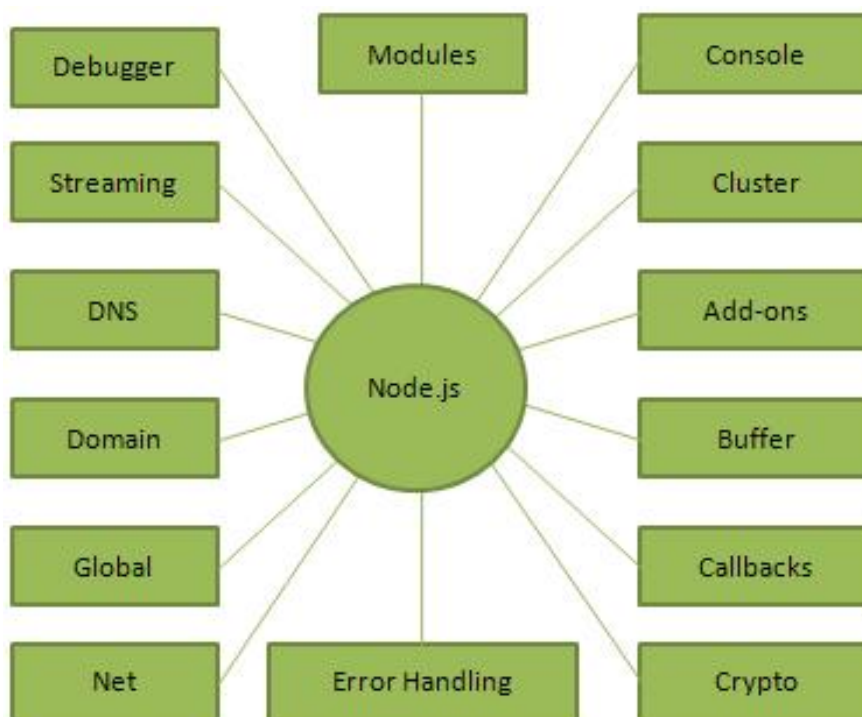


3.3 Použité technologie

3.3.1 Node.js

Node.js je framework sloužící k psaní JavaScriptového kódu mimo prostředí prohlížeče. Node.js prostředí umožňuje přístup k různým systémovým funkcím jako přístup k souborům a síťovým prvkům (viz obrázek 3.3). Je postaven na engine V8 pro Chrome. V dnešní době je Node.js populární k psaní serverové části aplikací reagujících na I/O události. Node.js všechna data zpracovává po chunkách, a tudíž nikdy žádná neukládá do bufferu [11].

Obrázek 3.3: Důležité části Node.js [1]



3.3.2 Next.js

Next.js je univerzální framework běžící na serveru i v prohlížeči. Slouží k psaní React aplikací, které jsou vykresleny už na straně serveru (server-side rendering). Na serverové straně Next.js využívá Node.js a framework Express.js.

V klasickém případě přijde uživateli webový obsah, do kterého se následně vykreslí všechny React komponenty. Mezitím je ale uživatel nucen čekat. S server-side renderingem však uživateli přijde již vykreslená webová stránka a on vidí příslušné komponenty hned, jak se stáhne počáteční obsah. Poté dojde k inicializaci Reactu i na straně prohlížeče, ale to už uživatel přímo nevidí.

To spatřuje výhodu u pomalejších zařízení, která se dostanou k počátečnímu obsahu rychleji. Jednou z nevýhod ale je pochopitelně nutnost většího výkonu na straně serveru.

3.3.3 MongoDB

MongoDB je dokumentově orientovaná multiplatformní NoSQL databáze. Dokumenty jsou psány ve formátu BSON, který vychází ze známého formátu JSON. Oproti tradičním relačním databázím podporuje dynamické databázové schéma. Zodpovědnost za toto schéma je totiž většinou ponechána na aplikační logice. Při přidání nového atributu máme dvě možnosti. První je, že se musíme spokojit s datovou heterogenitou a řešit problém pomocí aplikační vrstvy. Druhá možnost je doplnit zbývající záznamy o nový atribut, což ale může být výpočetně velmi náročné u mnoha záznamů.

U MongoDB se musí vývojář oprostít od technik používaných při návrhu relačních databází, protože v případě MongoDB je normalizace databáze kontraproduktivní. Přílišná normalizace vede k více databázovým dotazům a ty jsou zde časově náročnější [12].

Výhody:

1. Flexibilní schéma - Na rozdíl od relačních databází je *schema-less*, což usnadňuje situaci v některých případech.
2. Škálovatelnost - Rychlost čtení lze jednoduše škálovat pomocí replik (více serverů obsahujících stejná data). Zápis lze škálovat pomocí shardingu (rozdělení specifických dat mezi více serverů).
3. Rychlost - Oproti relačním databázím dosahuje MongoDB lepších výkonnostních výsledků, pokud jsou dodržena pravidla při jeho návrhu jako například denormalizace.

3. NÁVRH ŘEŠENÍ A POUŽITÉ TECHNOLOGIE

Nevýhody:

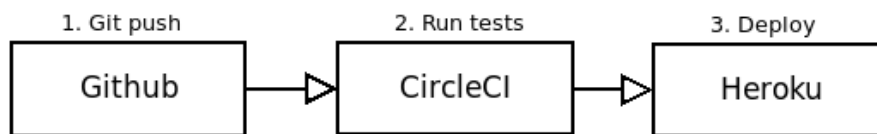
1. Flexibilní schéma - Přesouvá zodpovědnost z databáze na aplikaci a programátora.
2. Složitější dotazování (absence JOIN, atd.).
3. Neumožňuje transakce.
4. Velikost - Kvůli optimalizaci rychlosti dochází často k duplicitám dat.

3.3.4 CircleCI

Webová platforma pro realizaci continuous integration a continuous delivery. Lze velmi jednoduše propojit s GitHub, na kterém jsou i obě aplikace této diplomové práce. CircleCI nabízí již připravené docker obrazy, ze kterých si stačí pouze vybrat podle používané technologie.

Continuous integration slouží k urychlení vývoje a udržení kvality kódu. Při propojení s GIT repositářem si platforma vytvoří build, na kterém provede testy a ověří jeho správnost. V případě této práce spustí testy aplikace (integrační a unit testy), ESLintu a Flow. Pokud celý tento proces proběhne v pořádku, tak dojde k nahrání nové verze aplikace na produkční nebo testovací prostředí (continuous delivery).

Obrázek 3.4: Průběh CI u aplikace



3.3.5 Yarn

Yarn je alternativa k npm. Nabízí výhody jako například yarn.lock (v nejnovější verzi npm již existuje také), kdy má vývojář jistotu, jaká verze knihoven bude nainstalovaná na produkční prostředí. Další z výhod je offline cache. Při instalaci knihoven pomocí *yarn install* se tyto balíčky uloží na disk a při jejich opětovné instalaci se už nemusí znovu stahovat.

3.4 Použité knihovny

3.4.1 React

Dnes velmi populární knihovna pro vytváření uživatelských rozhraní v JavaScriptu. Její největší přínos je specifická manipulace s DOMem. DOM je zjednodušeně řečeno stromová reprezentace komponent, které uživatel vidí v prohlížeči a navíc definuje metody pro manipulaci s ním. Častá manipulace s DOMem může vést ke zpomalení aplikace a právě tento problém pomáhá řešit React. Ten programátora odstiňuje od práce s ním tak, že si vytváří svůj vlastní virtuální DOM, který pomocí různých specializovaných technik porovnává a hledá rozdíly, které pak co nejefektivnějším způsobem aktualizuje [13]. Používáme-li React nebo JSX (templátovací jazyk používaný v Reactu), tak nesmíme nikdy zapomenout React naimportovat na začátku souboru. React se dnes používá velmi často spolu s knihovnou Redux.

3.4.1.1 Props a State

Stav komponenty v Reactu můžeme definovat pomocí *props* a *state*. *State* je vnitřní stav komponenty, do kterého se můžou vkládat libovolná data. Tento *state* se mění pomocí funkce *setState()*.

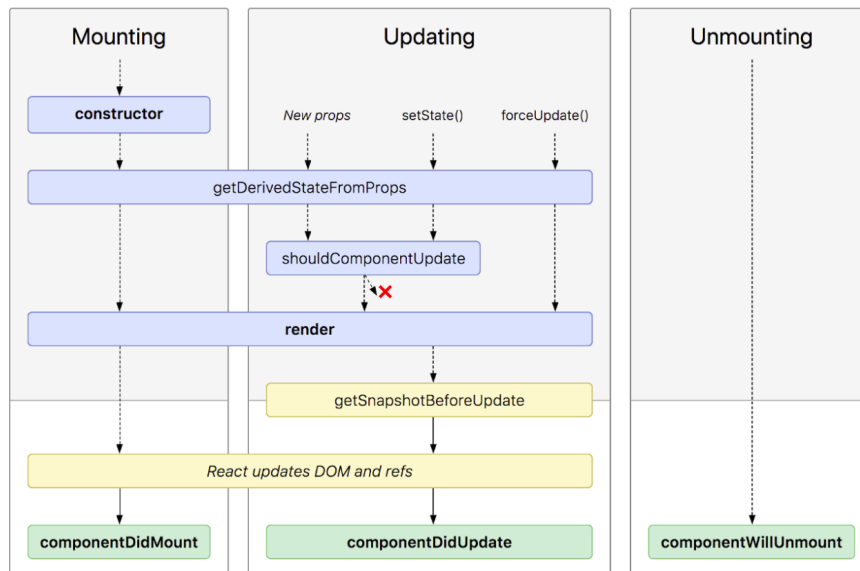
Pokud komponenty do sebe vnořujeme, tak je potřeba mezi nimi předávat informace. K tomu slouží právě *props*, které ale lze předávat pouze skrz komponenty ze shora dolů. Potomek jej nemůže tedy využít k zaslání dat svému rodiči. Tento problém však jde vyřešit pomocí callback funkcí, které potomek zavolá.

3.4.1.2 Lifecycle metody

Pomocí lifecycle metod můžeme kontrolovat, co se s komponentou děje po celý její životní cyklus (od *pre-mount* do *unmount*). Každá z následujících metod se liší v čase, kdy jsou volány (viz obrázek 3.5).

1. *static getDerivedStateFromProps* - Modifikuje *state* na základě nově přichozích *props* před vykreslením.
2. *shouldComponentUpdate* - Metoda, která určuje, zda dojde k překreslení (vrací *true*, nebo *false*).
3. *render* - Metoda vykreslující komponentu.
4. *componentDidMount* - Metoda volaná hned po prvním vykreslení.
5. *componentDidUpdate* - Volaná po každém překreslení komponenty.
6. *componentWillUnmount* - Metoda volaná před zničením komponenty.

Obrázek 3.5: React lifecycle metody [2]



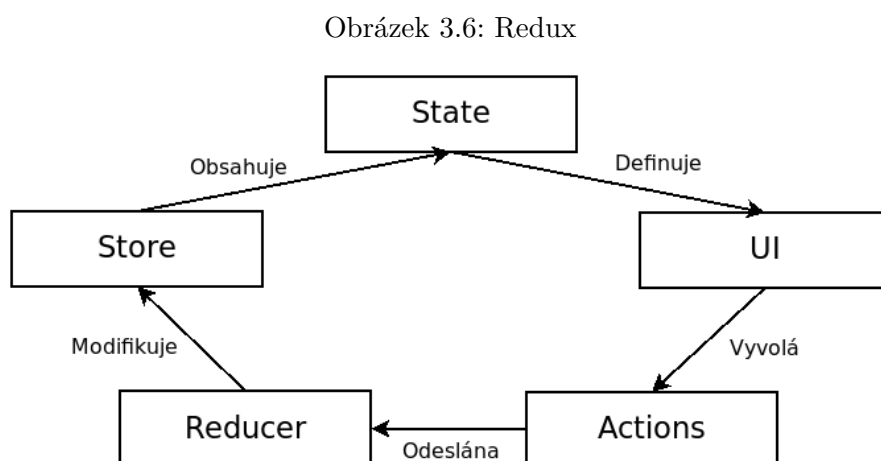
3.4.1.3 Vykreslování

Každá komponenta má jako jedinou povinnou metodu `render`. Tato metoda definuje, co bude vykresleno. Vykresleny mohou být jiné komponenty nebo HTML tagy. Kdy bude komponenta znovu vykreslena, závisí na jejím typu, ale po většinou je znovu vykreslena po každém použití funkce `setState` nebo pokud přijdou nové `props`. Toto chování však závisí na typu komponenty a implementaci `componentShouldUpdate`. React umožňuje vytvářet tři typy komponent:

1. Statefull komponenta - Je překreslena po každé změně stavu (`state` a `props`). Metoda `componentShouldUpdate` vždy vrací `true` (pokud není předefinovaná).
2. Stateless komponenta - Nemá vnitřní stav, řídí se pouze změnami v `props`.
3. Pure komponenta - V `componentShouldUpdate` se provádí mělké porovnávání aktuálních a nadcházejících stavových objektů. Díky tomu se překreslení neprovádí vždy, jako tomu je u klasické komponenty, ale jen když se `state` nebo `props` liší. Tyto objekty pochopitelně nemohou mít složité vnořené struktury.

3.4.2 Redux

Redux je JavaScriptová knihovna určená pro správu stavu (*state*) aplikace. Lze ho chápat jako aktuální stav aplikace od jejího počátku spolu se všemi změnami, které byly v jejím průběhu provedeny [14]. Způsob manipulace se stavem aplikace lze popsat pomocí následujícího obrázku.



Jak je naznačeno na obrázku, tak veškerá modifikace *statu* se provádí pomocí akcí, které reducer přijímá. Reducer určuje, jak se stav aplikace změní po vyvolání specifické akce. Všechny tyto modifikace by měly být imutabilní z důvodů správného překreslování komponent a správného fungování knihovny. Další důležitou částí Reduxu je store. Store je objekt obsahující celý strom stavů a několik řídicích funkcí.

3.4.3 Bootstrap 4

Bootstrap je sada nástrojů pro vývoj webových aplikací. Ulehčují programátorovi práci v oblasti UI a webdesignu. Bootstrap totiž disponuje celou řadou šablon, připravených komponent (například pro formuláře) a animací. Vývojář při jeho využívání nepoužívá klasické CSS styly, ale pouze již definované CSS třídy. Některé jeho funkce jsou stále plně funkční pouze s knihovnou jQuery. Nejnovější verze Bootstrapu (verze 4) je psána v jazycích Sass a JavaScript.

3.4.3.1 Sass

Je kompilovaný jazyk rozšiřující syntaxi CSS o celou řadu dalších prvků jako proměnné, podmínky, cykly a mixin. Jeho hlavní výhodou oproti CSS je, že zpřehledňuje kód a šetří čas [15].

3.4.4 **Lodash**

Funkcionální knihovna, která obsahuje mnoho pomocných funkcí pro zpracování polí a objektů. Tyto funkce jsou velmi užitečné při filtrování a grupování dat podle určitých podmínek. Další přidanou hodnotou je, že se funkce dají composovat.

3.4.5 **Recharts**

Důležitou funkcionalitou frontend aplikace je zobrazení dat v grafu. Knihovna Recharts poskytuje celou řadu grafů, které jsou napsány jako React komponenty. To má za výhodu snadné modifikování vykreslovaných dat a pomocí již implementovaných animací a designových prvků tak může poskytovat příjemný uživatelský dojem a užitečné funkcionality.

3.4.6 **React Google Maps**

Jednotlivé senzory bude možné zanášet do mapy. Nejrozšířenější a funkcionálně nejbohatší řešení poskytuje Google Maps Platform [16] a jejich implementace pro React s názvem React Google Maps. Pro jejich využívání je však potřeba API klíč, který je poskytován za poplatek. Při vývoji je však možné využít vývojářský klíč. Knihovna umožňuje na základě koordinát (zeměpisné šířky a délky) zanášet na mapu značky, které budou v případě této aplikace reprezentovat právě senzory.

Google Maps dále umožňují rozšířenou geolokaci na základě názvu objektu, včetně hledání známých míst. Tato funkcionalita ale není v prvotní verzi této práce využita.

3.4.7 **Mapbox**

Z důvodu poplatků za používání Google Maps Platform je použita i další alternativa a to Mapbox [17]. Ten na rozdíl od Google Maps poskytuje zdarma token na 50 000 zobrazení za měsíc. Správce aplikace se může sám rozhodnout, kterou z platforem používat. Jaká mapa bude použita, bude rozhodnuto na základě toho, jaký API klíč správce při instalaci aplikace zadá.

Používání této platformy umožňuje knihovna *React map gl*, která opět obsahuje připravené React komponenty.

RESTful API

Tato část se věnuje implementaci RESTful API, jehož funkcionality byla popsána v předešlých kapitolách. Jak bylo naznačeno, toto API je implementováno pomocí frameworku Express a využívá MongoDB databázi. Vývoj probíhal v postupných krocích, které budou v této kapitole popsány.

4.1 Prostředí

Nejprve je nutné nastavit počáteční prostředí. Jako první je třeba nainstalovat všechny potřebné knihovny. Zde je výčet s popisem těch nejdůležitějších z nich:

1. Babel - JavaScriptový compiler používaný k převedení různých dosud plně nepodporovaných funkcí a syntaxe na jednotnou a podporovanou podobu.
2. Chai - Knihovna pro testování JavaScriptových aplikací. Obsahuje všechny potřebné funkce k psaní unit testů a integračních testů.
3. ESLint - Nástroj pro udržování kvality kódu. Stará se o dodržování pravidel jeho vzhledu a struktury jako například délky řádků a odsazení. Umožňuje definování vlastních pravidel, či využívání pravidel třetích stran.
4. Flow - Rozšíření syntaxe umožňující v dynamicky typovaném JavaScriptu deklarovat typy. Nástroj flow-bin zároveň ověřuje jejich dodržování napříč kódem.
5. Istanbul - Nástroj pro zjišťování, jak je kód pokryt testy.
6. Mocha - JavaScriptový framework běžící na Node.js a v browseru pro asynchronní testování.
7. Nodemon - Nástroj usnadňující vývoj v Node.js. Restartuje server po každé změně kódu.

4. RESTFUL API

8. Prettier - Automaticky formátuje kód podle vybraných pravidel.
9. Bcrypt - Šifrovací knihovna, která je použita k šifrování hesel.
10. Body-parser - Middleware pro Node.js používaný k extrakci obsahu příchozích požadavků.
11. Express - Framework pro Node.js, který obsahuje mnoho funkcí ulehčujících vývoj webových aplikací.
12. Lodash - Knihovna poskytující celou řadu funkcí pro usnadnění práce programátora. Zároveň podporuje funkcionální paradigma.
13. Mongoose - Knihovna umožňující práci s MongoDB v Node.js.
14. Morgan - Slouží k logování HTTP požadavků v Node.js.
15. Passport - Autentizační middleware pro Node.js. Obsahuje i rozšíření podporující JSON web token autentizaci, která je používána v této práci.

4.1.1 Adresářová struktura

Adresářová struktura odpovídá klasické MVC architektuře. Stěžejní je složka *app* a *tests*. Ve složce *app* se nachází veškerá struktura aplikace a ve složce *tests* všechny integrační a unit testy. Ve složce *configs* budou všechny konfigurační soubory včetně konfigurace pro nastavení MongoDB a tajný klíč pro šifrování atd. Tyto konfigurace se prioritně získávají z proměnných operačního systému a poté z JavaScriptových souborů v podsložce *env* (podle aktuálního prostředí).

Za zmínku stojí složka *.circleci*, která obsahuje konfigurační soubor pro CircleCI. Zde jsou nastaveny Node.js a MongoDB docker obrazy, které jsou potřeba k otestování a zprovoznění aplikace. Dále jsou zde definovány dvě workflow, první je testování aktuálního buildu, kdy se spustí Flow, ESLint a testy. Pokud vše proběhne v pořádku, tak se spustí druhá workflow, která otestovaný build nahraje na Heroku.

build	výsledný build aplikace a testů
app	
├─ config.....	konfigurace globálních proměnných
├─ controllers	controllery pro zpracování požadavků
├─ lib.....	pomocné knihovny
├─ middlewares.....	middlewarey pro zpracování požadavků
├─ models.....	modely a definice MongoDB dokumenty
├─ routes.....	routy pro zpracování požadavků
├─ services.....	pomocné služby
scripts	pomocné skripty
test	složka pro integrační testy a unit testy
consts.js	konstanty
.babelrc	nastavení Babel
server.js	soubor, který spouští a nastavuje aplikaci

4.1.2 Nastavení Babel

Jako kompilátor je použit Babel, u kterého je nutné nastavit kompilace flow-type a ECMAScript 6 do podporovaného ECMAScript 2015. Dále byly přidány pluginy umožňující používání modernější syntaxe. Následující zdrojový kód ukazuje Babel konfiguraci uloženou v souboru *.babelrc*.

```
{
  "presets": [
    ["@babel/preset-env", {
      "targets": {
        "node": "current"
      }
    }],
    "@babel/preset-flow"
  ],
  "plugins": [
    "babel-plugin-dynamic-import-node",
    "@babel/plugin-transform-flow-strip-types",
    "@babel/plugin-proposal-object-rest-spread"
  ]
}
```

Dále bylo nutné definovat příkazy, které budou vykonávat buildování a spuštění aplikace. Následující úryvek ze souboru *package.json* zobrazuje definici takových akcí.

```
"scripts": {
  "heroku-postbuild": "babel app -d build/app",
  "test": "./scripts/test",
  "build": "babel app -d build/app",
  "start": "NODE_ENV=production node build/app/server.js",
  "dev": "nodemon --exec babel-node app/server.js "
},
```

Každý z příkazů definovaných v úryvku z *package.json* má specifické funkce a je využit v následujících případech

1. heroku-postbuild - Skript spuštěný Heroku platformou pro vybuildování aplikace.
2. test - Skript starající se o build aplikace, testů a jejich následné spuštění.
3. build - Vykonává build aplikace do složky *build/app/*.
4. start - Spuštění posledního buildu ze *build/app/*. Tento příkaz je používán pro produkční prostředí.
5. dev - Spuštění nástroje Nodemon, který se využívá na vývojovém prostředí. Aplikaci není nutné po každé změně kódu restartovat.

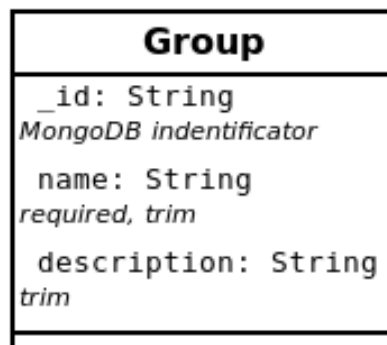
4.2 Návrh modelů

Jako další krok před samotným programováním byl nutný návrh struktury celé aplikační logiky vycházející z obrázku 3.2. Bylo nutné navrhnout jednotlivé modely tak, aby splňovaly zadání a aby byly stále aktuální, a to i pokud byl model vnořen do jiného modelu a došlo k jeho modifikaci. Zde je nutné mít na paměti požadavky na optimální MongoDB návrh, kdy se musí programátor vyvarovat přílišné normalizaci. Jako první MongoDB dokumenty (modely) byly navrženy ty, do kterých není nutné vnořovat jiné dokumenty. Všechny následující struktury obsahují automaticky generované unikátní ID.

4.2.1 Skupiny

V zadání je definováno, že je nutné umožnit uživatelům vidět vybraná data ze senzorů, které definuje správce. Řešení tohoto bodu je navrženo tak, že samotní správci mají přístup k datům všech senzorů. Uživatelé budou přidělováni do skupin, které budou přidělovány jednotlivým senzorům. Uživatel ve skupině A a B tedy uvidí data senzorů, které jsou ve skupině A nebo B. Dokument skupiny bude mít následující jednoduchou strukturu.

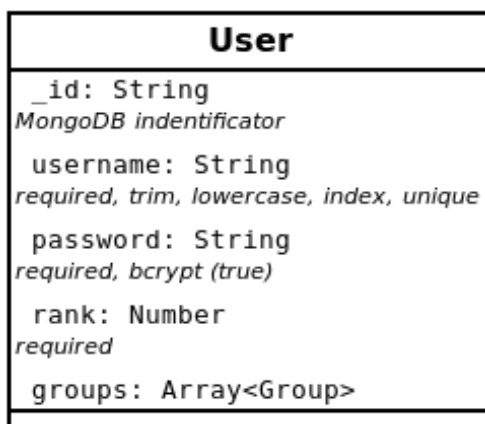
Obrázek 4.1: Databázové schéma skupin



4.2.2 Uživatelé

Nyní už je možné vytvořit uživatelský dokument. Kromě uživatelského jména a hesla bude dokument obsahovat pole identifikátoru skupin, kterých je součástí. Zároveň musíme přidat atribut poskytující informaci o tom, zda se jedná o uživatele, správce nebo libovolnou jinou entitu. Tento atribut je pojmenován *rank* a bude mít číselnou hodnotu. Heslo bude automaticky hashováno pomocí knihovny Bcrypt, kde je pouze potřeba ji povolit u atributu *password* a vložit importovanou knihovnu do uživatelského schématu.

Obrázek 4.2: Databázové schéma uživatelů



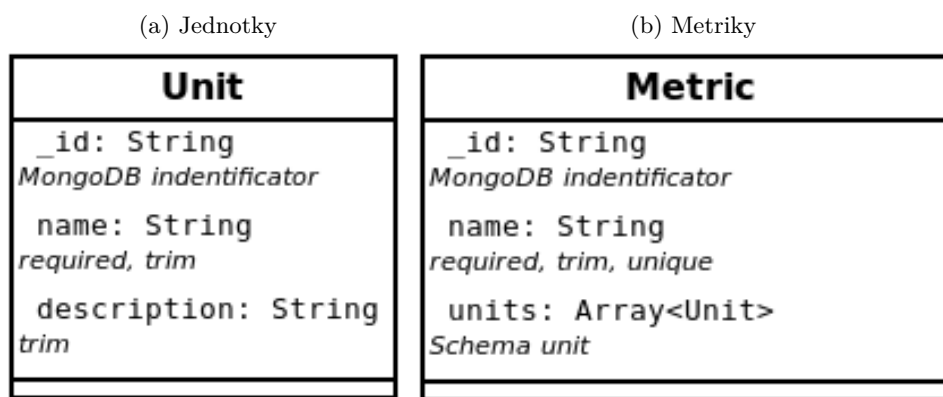
Aplikace bude mít tři uživatelské role. První role je hlavní správce. Jeho účet bude vytvořen při inicializaci aplikace a je zodpovědný za její provoz. V jeho pravomoci je vytvářet ostatní uživatele a případně další správce. Spolu s ostatními správci může vytvářet metriky, senzory a skupiny. Senzory a uživatelé mohou přiřazovat do skupin, ale sami vidí veškerá data, aniž by těchto skupin byli součástí.

Každá z těchto tří rolí je reprezentována rankem, který má číselnou hodnotu. Hlavní správce je reprezentován hodnotou nula a ostatní role mají vyšší hodnotu, dle důležitosti v hierarchiích. Je-li například dáno, že akce je povolena pouze správcům, tak rank uživatele musí být menší než dva.

4.2.3 Metriky a jednotky

Před definicí senzorů a mapperů je nutné definovat metriky a jednotky. Každá metrika by měla mít jednotky. Maximálně jedna jednotka může být označena jako kotevní. Pomocí číselného koeficientu jsou na ni ostatní jednotky převoditelné, což znamená, že se poté dají jednotky převádět i mezi sebou. Tento koeficient bude nepovinný, ale bez něj a kotevní jednotky nebude možné převod provést. Jednotky budou samostatný dokument, který bude vnořen do metrik jako pole.

Obrázek 4.3: Databázové schéma metrik a jednotek



4.2.4 Senzor

Každý senzor nese informaci o svém vlastním výrobním identifikátoru, názvu a popisu. Aby mohl být senzor zobrazen na mapě, tak je třeba dále správci umožnit vyplnit jeho souřadnice *lat* a *lng* (zeměpisnou šířku a délku). Nejdůležitější částí nejen senzoru, ale i celé aplikace, budou mappery, které budou parsovat příslušný obsah z příchozích požadavků. Prozatím pouze určíme, že každý senzor bude obsahovat pole mapperů pro příslušný typ požadavku (JSON, text, atd.). Tato logika bude podrobně vysvětlena v následujících kapitolách.

Obrázek 4.4: Databázové schéma senzoru

Sensor
<code>_id: String</code> <i>MongoDB indentificator</i>
<code>identificator: String</code> <i>required,, index, unique</i>
<code>description: String</code>
<code>datasource_type: String</code> <i>required</i>
<code>separator: String</code>
<code>lat: Number</code>
<code>lng: Number</code>
<code>groups: Array<Group></code>
<code>jsonMappers: Array<BasicMapper></code>
<code>textMappers: Array<BasicMapper></code>
<code>binaryMappers: Array<BinaryMapper></code>
<code>binaryMixedMappers: Array<BinaryMixedMapper></code>

4.2.5 Data

Tento model bude obsahovat již zpracovaná data ze senzorů. Každá položka dokumentu bude obsahovat hodnotu, ID senzoru, ID metriky, jednotku a statusový příznak. Tento dokument bude obsahovat nejvíc položek a bude používán pro výpis a prezentaci ve frontendové aplikaci.

Obrázek 4.5: Databázové schéma dat

Data
<code>_id: String</code> <i>MongoDB indentificator</i>
<code>sensor_id: ObjectId reference</code> <i>Reference na sensor</i>
<code>unit: String</code>
<code>value: String</code>
<code>is_status: Boolean</code>
<code>metric: ObjectId reference</code> <i>Reference na metriku</i>

4.3 Routy a controllery

Nejprve je vhodné vysvětlit, jak funguje vytváření rout a controllerů, jelikož později z toho vychází celá řada middlewarových funkcí, jako například přihlášení.

4.3.1 Routy a controllery v Express

Routa směřuje příchozí požadavky na příslušné akce controlleru. Routa se pomocí frameworku Express vytváří následovně.

```
app.get(route_path, ...callbacks);
```

Pro všechny HTTP metody (GET, POST, PUT, PATCH, DELETE) existují v Express funkce, které jako argumenty přijmou na prvním místě cestu, jakou routa zpracovává a poté libovolný počet callback funkcí. Tyto funkce obdrží jako argumenty *Request*, *Response* a *NextFunction*.

1. Request - HTTP požadavek extrahovaný do objektu, který obsahuje všechny náležitosti jako metodu, URL parametry, body, atd.
2. Response - Objekt reprezentující budoucí HTTP odpověď. Tento objekt je modifikován a následně odeslán metodou *send*.
3. NextFunction - Metoda říkájící, že se má pokračovat v průchodu dalšími callback funkcemi. Bez jejího zavolání je řetězec průchodů ukončen aktuální funkcí.

Tyto callback funkce se dají využít jako middleware, který může například provádět autentizaci nebo jen modifikovat požadavek o další informace. Controller odpovídající na validní požadavek je tedy většinou poslední callback funkce v tomto řetězci.

```
app.get(route_path, ...middlewares, controller.action);
```

4.3.1.1 Extrakce URL parametrů

Express provádí extrakci URL parametrů, který se nachází v *PATH* za dvojtečkou. Všechny extrahované URL parametry se poté nachází s totožným názvem v objektu *request.params*, který přichází jako parametr již zmíněných callback funkcí.

```
GET /users/:parametr -> req.params.parametr
```

4. RESTFUL API

4.3.2 Implementace rout a controllerů

Jako první je třeba vytvořit controllery s routami pro již navržené modely, aby je mohl uživatel a správce skrz aplikaci vytvářet a číst. Jedná se o controllery pro skupiny, uživatele, metriky a senzory. Tyto controllery sdílí mnoho, mají podobné chování i strukturu. Každý controller obsahuje stejnou sadu základních rout, která se od následující ukázky liší pouze v názvu příslušné entity.

```
GET /sensors/:_id
GET /sensors
PATCH /sensors/:_id
POST /sensors
DELETE /sensors/:_id'
```

Příslušné akce mají jednoduchou implementaci, kdy se většinou pracuje pouze s Mongoose. Následující ukázka zobrazuje způsob, jakým je implementován proces routování až směrem k akcím controlleru. Routy pro příslušné controllery se nachází ve složce *routes*.

```
// routes/metrics.js

app.post('/metrics', MetricController.post);

// controllers/MetricController.js

export const post = (req: $Request, res: $Response) => {
  Metric.create(req.body)
    .then(unit => {
      res.status(200).json(unit.toJSON());
    })
    .catch(err => {
      res.status(400).json(err);
    });
};
```

Všechny routy jsou dynamicky importované v souboru *server.js* pomocí knihovny Glob. Glob umožňuje synchronně načítat soubory pomocí patternů.

```
// server.js

glob.sync('routes/**/*.js',
  { cwd: __dirname }).map(filename => {
  import('./'.concat(filename)).then(module => {
    module.default(app);
  });
});
```

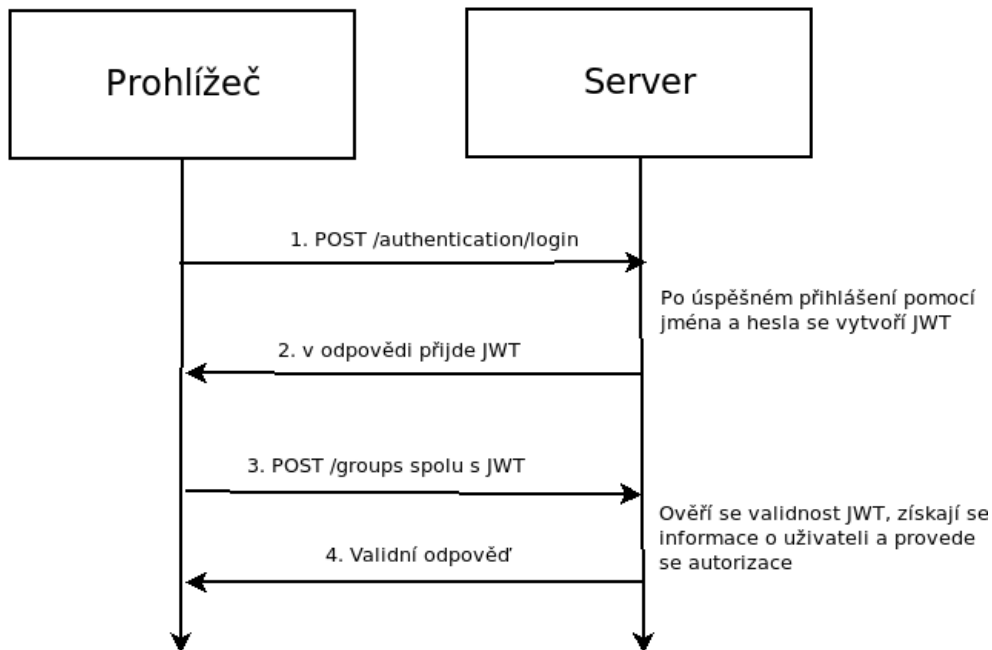
4.4 Přihlašování

Při implementaci přihlašování se používá JavaScriptová knihovna Passport, která poskytuje middleware pro Node.js podporující celou řadu různých strategií přihlašování, a to i přes různé systémy třetích stran, jako například GitHub, Facebook, Google, atd. V této aplikaci se bude používat strategie využívající JSON web token.

4.4.1 JSON web token

JSON web token je standard splňující RFC 7519 [18], který se používá k zabezpečení komunikace mezi API a klientem. Bezpečně přenáší informace mezi komunikujícími stranami jako JSON objekt [19]. Věrohodnost tohoto objektu může být ověřována, jelikož je podepsán příslušným algoritmem. JWT může být podepsán tajným klíčem, například pomocí HMAC. Passport v JWT objektu uchovává informace identifikující uživatele a čas expirace tokenu. Proces přihlašování při použití JWT lze popsat pomocí následujícího obrázku.

Obrázek 4.6: JWT proces



4.4.2 Vytvoření tokenu (přihlášení)

Přihlášení je vyvoláno endpointem s routou *POST /authentication/login*, kde v body je zasláno uživatelské jméno a heslo. Tuto routu zpracovává příslušný controller, který získá uživatele odpovídajícího příslušnému uživatelskému jménu. Knihovna Bcrypt poté provede porovnání hashů hesla z databáze a hesla z požadavku. Pokud je zjištěna shoda, tak se zavolá metoda, která vytvoří JWT. Nejelegantnější umístění této metody je přímo v definici modelu uživatele, protože se poté dá zavolat na každou instanci daného uživatele.

```
// models/User.js

UserSchema.methods.getJwt = function() {
  const expiration_time = parseInt(config.jwt_expiration);
  return (
    'Bearer ' + jwt.sign(
      { user_id: this._id },
      config.jwt_secret,
      { expiresIn: expiration_time }
    )
  );
};

// controllers/AuthController.js

export const login = (req: $Request, res: $Response) => {
  User.findOne(
    { username: req.body.username },
    (error, user) => {
      const isPasswordValid = bcrypt.compareSync(
        req.body.password,
        user.password,
      )
      if (!isPasswordValid)
        return res
          .status(401)
          .send({ auth: false, token: null })

      res.status(200).json({
        auth: true,
        token: user.getJwt(),
        username: user.username,
        rank: user.rank,
      })
    },
  )
}
```


4.4.3 Passport strategie

Pro používání passport je třeba implementovat vlastní JWT strategii, kde se nastaví, pomocí jaké funkce získat token a jak získat uživatele z informací v něm uložených. K implementaci takovéto funkcionality je potřeba doinstalovat strategii pro JWT s názvem *passport-jwt*.

Získávat token bude podle strategie bearer authentication scheme, které token obsahuje v hlavičce authentication s předponou *bearer*. V překladu toto schéma lze popsat jako "Dát přístup držiteli (bearer) tohoto tokenu"[20].

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

Zdrojový kód strategie vypadá následovně. Nejprve se získá token a poté se vloží spolu s *jwt_secret* (uloženým v konfiguračním souboru) do objektu, pomocí něhož se vytvoří nová passport strategie. Tato strategie pomocí uživatelského identifikátoru, který byl uložen v tokenu, získá existujícího uživatele. Pokud jakýkoliv krok skončí neúspěchem, tak dojde k chybné autorizaci a odeslání příslušné odpovědi.

```
export default (passport: Passport) => {
  var opts = {};
  opts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();
  opts.secretOrKey = config.jwt_secret;

  passport.use(
    new Strategy(opts, async function(jwt_payload, done) {
      User.findById(jwt_payload.user_id)
        .lean()
        .exec((err, user) => {
          if (err) throw err;

          if (user) {
            return done(null, user);
          } else {
            return done(null, false);
          }
        });
    })
  );
};
```

4.5 Mappery

Další důležitou částí aplikace jsou mappery. Jejich úkolem bude mapovat a parsovat data z požadavků příslušným metrikám. Mapper bude sloužit ke zpracování určitého formátu obsahu požadavku. Tato práce se zabývá hlavně zpracováním textového, JSON a binárního formátu reprezentovaného v hexadecimální soustavě.

Každý mapper může obsahovat místo metriky a jednotky statusový příznak *is_status*. To je booleovská hodnota, která říká, že daná zpracovaná hodnota nese kód, v jakém stavu se senzor nachází nebo proč nemohla dorazit validní data.

Senzor bude mít jeden typ mapperů a bude určen ke zpracování jednoho formátu dat. Každý typ mapperu se bude nacházet v modelu senzoru v poli a každý mapper v tomto poli bude sloužit k nalezení jedné hodnoty.

Pokud přijde požadavek na příslušný endpoint, nejprve je nutné zjistit, o který senzor se jedná, aby byly použity příslušné mappery. To bude probíhat tak, že přijde-li požadavek, tak bude zpracován globálními mappery pro detekci senzoru podle identifikátoru. Zjednodušeně řečeno aplikace bude hledat identifikátor v požadavku na místech, která definuje správce.

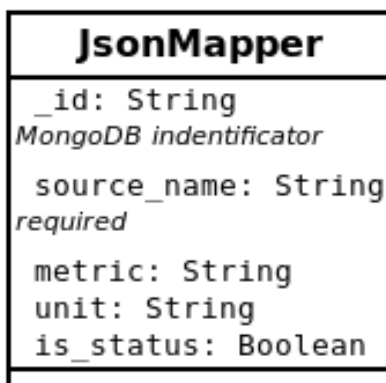
4.5.1 JSON mapper

Model mapperu pro JSON bude obsahovat 4 atributy. Cestu k hledané hodnotě, ID metriky, jednotku a statusový příznak. Cesta k požadované hodnotě budou klíče potřebné k dosažení hodnoty oddělené tečkou.

```
{ "key": { "key2": 10 } } => key.key2  
{ "key": [{ "key2": 10 }, { "key3": 10 }] } => key[1].key3
```

Schéma takového mapperu bude vypadat následovně:

Obrázek 4.7: Databázové schéma JsonMapperu



4.5.2 Text mapper

Textový formát je zasílán v podobě textového řetězce key-value hodnot oddělených separátorem. Separátor je uložen v modelu senzoru. Ke zpracování tohoto formátu je plně postačující schéma mapperu, který byl definován pro JSON formát. *JsonMapper* můžeme tedy přejmenovat na *BasicMapper*, jelikož bude využíván pro svoji univerzalitu více formáty.

Následující ukázka zobrazuje, jaké hodnoty *source_name* a *separator* jsou potřeba pro získání hodnoty *val1* klíče *key1* z příslušného řetězce.

```
key1=val1;key2=val2 => source_name="key1=" separator=";"
```

4.5.3 Bin mapper

Na fungování binárního mapperu se lze dívat z mnoha úhlů. Může nám přijít JSON, kde pouze vybraná hodnota bude obsahovat binární data. Další možností je, že nám například přijde *octet-stream* požadavek, který bude kompletně v binární podobě. V každém případě je tedy v konečném důsledku nutné zpracovat nějaká binární data, ale je nespočetně mnoho způsobů, jak je v požadavku nalézt. Tato aplikace v prvotní verzi bude řešit pouze dva případy. Přijdou-li data v *octet-stream* požadavku nebo v JSON, budou se zpracovávat po bytech, jelikož se zatím věnujeme zpracování hexadecimálního formátu. Z těchto důvodů je tedy nutné, aby měl základní binární mapper následující položky.

1. Od jakého bytu hodnota začíná.
2. V jakém bytu hodnota končí.
3. Jakou má hodnota metriku a jednotku nebo zda má statusový příznak.

Obrázek 4.8: Databázové schéma BinaryMapperu

BinaryMapper
_id: String <i>MongoDB indentificator</i>
from_byte: Number <i>required</i>
to_byte: Number <i>required</i>
unit: String
metric: String
is_status: Boolean

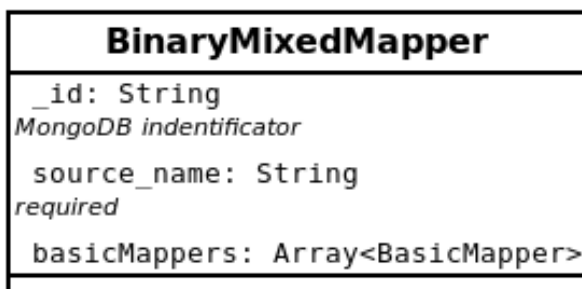
4. RESTFUL API

Za pomoci těchto položek lze získat hodnotu z binárních dat pomocí jednoduchého zpracování řetězce.

```
parseInt(binaryValue.substr(from_byte, to_byte), 16)
```

Zároveň je umožněno tento binární mapper vkládat do jiných mapperů, které budou mít pouze za úkol extrahovat hodnotu z jiného formátu požadavku jako například JSON a zpracování této hodnoty nechají na binárním mapperu.

Obrázek 4.9: Databázové schéma BinaryMixedMapperu



4.6 Použití mapperů

Přijde-li požadavek na endpoint `/datasource`, tak je proces zpracování rozdělen do následujících kroků.

4.6.1 Získání senzoru

Nejprve je požadavek zpracován middleware, který se pokusí zjistit identifikátor senzoru pomocí mapperů k tomu určených. Podle *content-type* se však vyberou pouze mappery pro příslušný formát požadavku. Přijde-li požadavek s *content-type json/application*, tak je funkcí *SensorIdParser* zpracován pomocí *IdParseru* pro JSON. Pokud dojde k získání identifikátoru, tak se funkce pokusí nejprve získat senzor podle databázového identifikátoru a pokud senzor nebude nalezen, tak se pokusí vyhledat pomocí atributu *identificator*.

```
if (headers['content-type'] === consts.contentType.JSON) {
  const sensorIdMapper = await SensorIdMapper.find({
    content_type: consts.contentType.JSON})
  if (!sensorIdMapper) return null

  const sensorId =
    JsonIdParser(sensorIdMapper, requestBody)
  const sensor =
    await Sensor.findOne({identificator: sensorId})
  if (!sensor) return await Sensor.findById(sensorId)

  return sensor
```

Poté se nalezený senzor vloží k objektu požadavku pod atribut *sensor* a pokračuje se ve vykování řetězce dalších middlewarů, pokud nějaké jsou, až k samotnému *DatasourceControlleru*. V opačném případě je zaslána odpověď *400 Bad Request*.

4.6.2 Získání dat

Požadavek je následně controllerem zaslán ke zpracování *DataParseru*, který podle *content-type* a *datasource* atributu, již nalezeného senzoru z předešlého kroku, zvolí správný parser. Atributy *content-type* a *datasource* se ověřují společně, protože binární data mohou přijít i v *content-type* JSON, jak bylo vysvětleno dříve.

```
if (
  headers['content-type'] === consts.contentType.JSON &&
  sensor.datasource_type === consts.datasource.JSON_BINARY
) {
  return JsonBinParser(requestBody, sensor)
}
```

Příslušný parser následně vrátí již rozparovaná data v poli, ve kterém bude objekt ve formátu Data modelu (navrženého v předešlých kapitolách), který se následně uloží do MongoDB.

4.7 Poskytování dat

Data jsou poskytována oprávněným uživatelům na endpointu *GET /data*. V prvotní verzi aplikace jsou navrženy dvě podoby dat, které budou uživateli zaslány.

První data jsou přehledová. Jsou to data specifické metriky a senzoru, která jsou poté vypisována po označení senzoru na mapě, jak je zobrazeno na obrázku 5.3. V tomto detailu se nachází základní informace o senzoru jako jeho identifikátor a popis. Dále jsou zde charakteristiky jeho dat jako minimální a maximální hodnota a průměr. Pro získání těchto dat je nutné provolat endpoint

```
GET data/overview/:metricId/:sensorId
```

4. RESTFUL API

Jako odpověď poté přijde pole přehledových dat specifické metriky a senzoru. Tato data se ale nejprve převedou na stejnou jednotku. Následující kód vytváří přehled podle popsaných specifikací.

```
const getOverview = async ( data: Array<Data> ) => {
  const convertedData = await convertValuesUnit(
    data, metricId, unit,
  )

  const {
    min, max, sum, numberOfValidItems,
  } = getOverviewData(convertedData)

  return {
    min,
    max,
    mean: getMean(sum, numberOfValidItems),
    unit,
  }
}
```

Jak již bylo řečeno, každá položka obsahuje minimální a maximální hodnotu, průměr a jednotku, ve které jsou tyto hodnoty. To vše je stahováno po každém označení senzoru na mapě z důvodu lepšího výkonu. Jinak by se musela stahovat všechna data na počátku a to by bylo časově náročné.

Druhý endpoint je provoláván s ID senzoru. Jeho výsledkem jsou všechna data, která jsou pro daný senzor uložena.

```
GET /data/:sensorId
```

To je následně exportovatelné do CSV nebo vykresleno v grafu. Endpoint umožňuje zaslat další parametry, které zpřesní nebo modifikují výsledek.

Následující požadavek získá data senzoru určité metriky a případně zkonvertuje na stejnou jednotku (pokud je to možné).

```
GET /data/:sensorId/:metricId?convertUnit=unit
```

Tento požadavek pošle data za poslední rok, měsíc, týden nebo den. Akceptuje pouze konstanty *years*, *months*, *weeks* a *days*.

```
/data/:sensorId?lastDate=lastDateUnit
```

Všechny zmíněné parametry lze vzájemně kombinovat. Dále je možné požádat o statusová data. To jsou, jak již bylo řečeno, data, která mají místo metriky a jednotky statusový příznak. Díky těmto datům uživatel zjistí, v jakém stavu je senzor nebo zda nenastala nějaká chyba. K získání takových dat je vystaven endpoint

```
GET /data/:sensorId/status}
```

Frontend aplikace

Tato část bude stručně popisovat řešení frontendové aplikace, která využívá API zpracované v předešlé kapitole. Tato aplikace bude zhotovena pomocí frameworku Next.js a knihoven React a Redux. Ke stylování se bude používat Bootstrap navržený speciálně pro React a obsahující i navíc užitečné React komponenty.

5.1 Prostředí

Jak již bylo řečeno, tak k vývoji je použit framework Next.js. To usnadňuje inicializaci a prvotní nastavení projektu, jelikož většina základních konfigurací je již v základním nastavení po jeho instalaci. K získání údajů, jako například URL adresa API, se využívají opět systémové proměnné, ale na rozdíl od API se poté na druhém místě berou ze souboru `.env`.

Oproti API byly použity následující knihovny a pluginy.

1. babel/polyfill - Poskytuje funkce, které vývojář očekává, že jsou ve všech prohlížečích nativně. Některé funkce, jako například *Promise*, stále nejsou jednotně nativně implementovány napříč všemi prohlížeči.
2. babel/preset-react - Babel preset pro kompilaci Reactu.
3. zeit/next-sass - Jelikož v této aplikaci je použit Bootstrap 4, který je napsán v Sass, tak je nutné stáhnout rozšíření pro Next.js, které bude Sass soubory zpracovávat.
4. axios - Axios je používán ke komunikaci mezi aplikací a RESTful API. Ulehčuje zasílání HTTP požadavků v browseru a Node.js, které se poté zpracovávají asynchronně jako promisy.
5. dotenv - Knihovna umožňující pohodlné načítání proměnných z `.env` souborů spolu se systémovými proměnnými.

5. FRONTEND APLIKACE

6. moment - Moment usnadňuje práci s časem a datумы.
7. Jest - Testovací knihovna.
8. react-test-renderer - Knihovna potřebná pro psaní snapshot testů.
9. next-redux-wrapper - Wrapper pro Next.js integrující Redux do React.
10. react - Knihovna pro psaní UI v JavaScriptu.
11. react-bootstrap - Rozšíření pro React integrující knihovnu pro stylování bootstrap. V této práci je použit nejnovější Bootstrap verze 4, který je napsán v Sass.
12. react-google-maps - Pro zobrazení umístění senzorů jsou použity Google mapy. Pro React je možné využít již odladěnou komponentu.
13. redux a react-redux - Správa stavu aplikace.
14. recharts - React komponenty pro vykreslování grafů.
15. redux-thunk - Redux-thunk umožňuje využívat asynchronní akce.
16. universal-cookie - Knihovna usnadňuje práci s cookie.

5.1.1 Redux

Konfigurace Reduxu se bude nacházet v základním souboru `./src/redux.js`. V tomto souboru se budou i shlukovat reducery pomocí funkce `combineReducer`. Akce budou vykonávány asynchronně pomocí knihovny `redux-thunk`. Jedna z akcí včetně definice návratového typu může vypadat následovně:

```
type LoginAction =
  | { type: 'LOGIN_PENDING' }
  | { type: 'LOGIN_FULFILLED', username: string }
  | { type: 'LOGIN_REJECTED', payload: ErrorResponse }

export const login = (data: Object) => async ({
  dispatch,
  apiClient,
}: (ActionDeps) => Promise<LoginAction>) => {
  // implementace
}
```


Každá akce je následně zpracována příslušným reducerem. U každého reduceru se definuje typ daného stavu spolu s inicializačními hodnotami.

```
export type UserState = {
  // datové typy jednotlivých položek
}
const INITIAL_STATE = {
  // počáteční stav
}

const reducer = (
  state: UserState = INITIAL_STATE,
  action: UserAction) =>
{
  switch (action.type) {
    case 'LOGIN_PENDING':
      // modifikace stavu
    case 'LOGIN_REJECTED':
      // modifikace stavu
    case 'LOGIN_FULFILLED':
      // modifikace stavu
    default:
      return state
  }
}

export default reducer
```

5.2 Adresářová struktura

Struktura je přizpůsobena frameworku Next.js, který jednotlivé stránky automaticky zpracovává ze složky *pages*. Výsledný build Next.js ukládá do složky *.next*, která se i automaticky při buildu vytvoří. Stejně jako u API je zde složka *.circleci* s nastavením CircleCI. Samotné zdrojové kódy včetně React komponent se poté nachází ve složce *src*.

```
.circleci.....konfigurace CircleCI
├── .next ..... build aplikace
├── __tests__.....složka pro testy
├── pages.....implementace stránek pro Next.js
│   ├── _app.js
│   └── _document.js
├── src.....složka se zdrojovými kódy
│   ├── components.....zde budou často používané komponenty
│   ├── libs.....externí zdrojové kody jako např. Bootstrap
│   ├── helpers.....často používané funkce
│   ├── redux.js.....základní Redux soubor
│   └── types.js.....Definování typů pro Flow
├── static
├── tests
├── .babelrc.....konfigurace Babel
├── .env.....globální proměnné
├── .eslintrc.js.....konfigurace ESLint pravidel
├── .prettierrc.....definice pravidel pro Prettier
├── heroku-postbuild.sh.....post-build skript pro Heroku
└── next.config.js.....konfigurace Next.js
```

5.3 Stránky

Stránky frontend aplikace převážně reflektují rozvržení controllerů u API. V Next.js má každá stránka soubor ve složce *pages*. URL stránky je dáno názvem souboru. Implementaci stránky *users* najdeme tedy v souboru */pages/users.js*. Seznam všech stránek této aplikace je následující:

1. index - Základní stránka, která je také zobrazena nepřihlášenému uživateli.
2. login - Přihlašovací stránka.
3. users - Vytváření a mazání uživatelů, které je přístupné pouze správci.
4. changePassword - Stránka pro změnu hesla.
5. metrics - Správa metrik a jednotek (pouze pro správce).
6. sensors - Správa senzorů a mapperů (pouze pro správce).
7. sensorIdMappers - Správa mapperu pro parsování ID (pouze pro správce).
8. groups - Vytváření a mazání skupin (pouze pro správce).
9. data - Presentace nashromážděných dat přístupná oprávněným uživatelům.

Pokud chceme všechny stránky rozšířit například o nastavení *favicon.ico* nebo pozměnit základní rozvrstvení HTML stránky, tak je možné využít soubor */pages/_document.js*. Tento soubor je v Next.js šablonou pro všechny stránky, do kterého se následně vkládají.

Obdobně funguje i React v Next.js. Každá stránka je JavaScriptový soubor, který obsahuje React komponenty, které jsou vždy vykresleny v souboru *app*, který tvoří Reactovou nejvrchnější vrstvu v Next.js. V tomto souboru se například integruje Redux. V případě této aplikace se zde taktéž kontroluje, zda je uživatel přihlášen.

Pro základní vzhled stránky včetně umístění menu byla vytvořena komponenta *Page*, do které se obalují ostatní komponenty u většiny stránek.

5.4 Komponenty a jejich organizace

Každá stránka má ve složce *app* vlastní podsložku s React komponentami. Každá taková podsložka má také vlastní Redux akce a reducery, které se poté pomocí funkce *combineReducer()* v souboru *redux.js* shluknou dohromady. Většina těchto komponent je psána jako stateless nebo pure, (pokud není vyžadována vlastní implementace funkce *componentShouldUpdate*).

Některé z komponent využívají funkci *connect*, která mapuje *state* a akce z Reduxu do *props*. Tento proces se však provádí v odděleném souboru, který až posléze pošle všechny tyto *props* příslušné komponentě. To se v této aplikaci dělá z důvodů snadného testování, protože pokud samotná komponenta není připojena na Redux, ale data z něj chodí přes *props* z jiné komponenty, tak je pak možné tyto *props* mockovat a provádět snapshot testy.

Následující příklad ukazuje soubor mapující *state* metriky a akci *getMetrics* z Redux do komponenty *Mapper*. Při testování této komponenty je pak velice snadné tyto hodnoty nahradit vlastními smyšlenými, které budou nezávislé na stavu Reduxu.

```
import { connect } from 'react-redux'
import Mapper from './Mapper'
import { getMetrics } from '../../metric/actions'

export default connect(
  ({ metric: { metricList } }) => ({
    isFetching: metricList.isFetching,
    metrics: metricList.metrics,
  }),
  { getMetrics },
)(Mapper)
```

5.5 Prezentace dat

Jak bylo zmíněno, většina komponent pouze spravuje jednotlivé MongoDB dokumenty a nejsou nijak zajímavé. Dvě komponenty se však liší a plní onu prezentační funkci.

Na stránce pro vizualizaci dat si může uživatel vybrat, zda chce vidět detail senzoru nebo všechny senzory dané metriky. Jelikož lze očekávat větší množství senzorů, tak je k dispozici jednoduchý vyhledávač podle identifikátoru senzoru 5.1.

Obrázek 5.1: Ukázka vyhledávače

Sensors

Search sensor

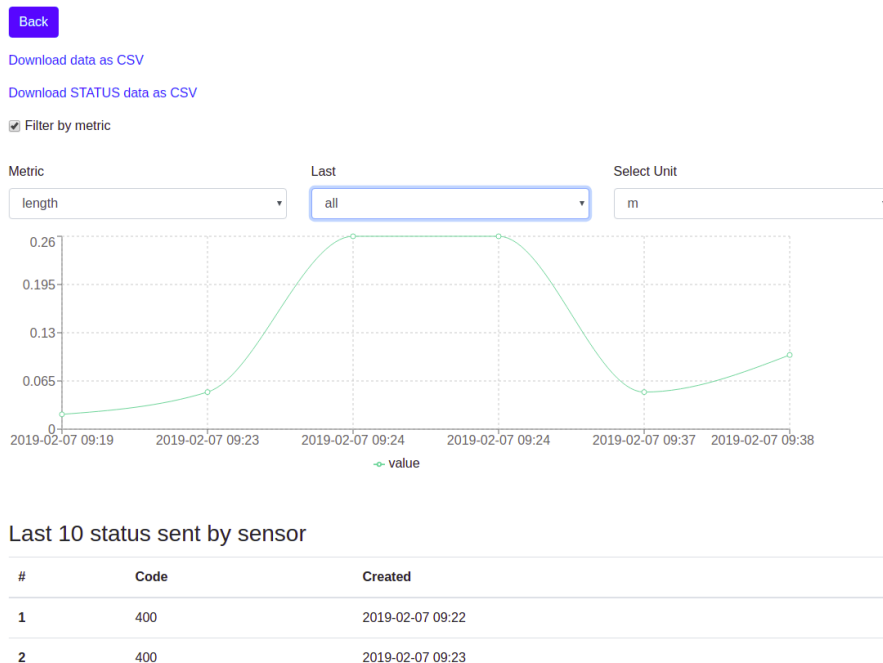
Identificator	Description
test123456	teeest

[Show](#)

5.5.1 Detail senzoru

První taková komponenta je detail senzoru. Jak lze vidět na obrázku 5.2, tak na detailu senzoru je k dispozici stažení dat v CSV a filtry, pomocí kterých si uživatel může data vybrat. Pokud to nastavení metriky umožní (data mají kotevní jednotku a jsou validní), tak je vykreslen graf za vybraný časový horizont. Jako poslední se ve spodní části nachází výpis posledních deseti statusů zasláných senzorem (jejich kompletní výpis je opět stáhnutelný v CSV). K získání dat potřebných k zobrazení této komponenty je využit endpoint `/data/sensorId`.

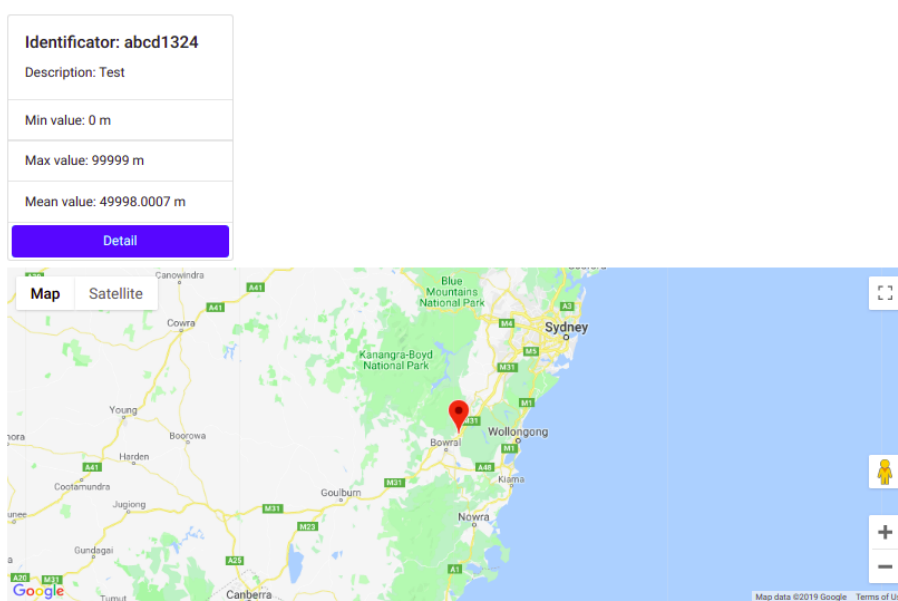
Obrázek 5.2: Ukázka detailu senzoru



5.5.2 Přehled senzorů na mapě

Druhá komponenta (obrázek 5.3) umožňuje zobrazit všechny senzory specifické metriky na mapě (pokud mají vyplněny *lat* a *lng* parametry). Dále je po kliknutí na značku senzoru na mapě zobrazena jeho stručná charakteristika s možností přeměrování na jeho detail. Charakteristika obsahuje minimum, maximum a průměr, ale jen pokud je možný převod na kotevní jednotku. Pro detail senzoru se využívá endpoint `/data/overview/:metricId/:sensorId`

Obrázek 5.3: Ukázka mapy s detailem senzoru



Tato komponenta je implementována pro dvě platformy poskytující online mapy, a to pro Google Maps a Mapbox. Použita je ta, která má v konfiguračním souboru `.env` vyplněn API klíč.

Testování

Testování je stejně jako řešení rozděleno na dvě části, testování frontendu a backendu. Jelikož se většina logiky provádí na backendu, tak mu byla věnována větší pozornost.

Backend aplikace je pokryta unit testy, které například ověřují funkčnost mapovací logiky. Všechny controllery jsou pokryty integračními testy, kde se ověřuje celková funkčnost aplikace. Dále je ověřena správnost řešení otestováním výkonu pomocí zátěžových testů.

U frontend aplikace se testují React komponenty pomocí snapshot testů.

6.1 Testování backend aplikace

Jak již bylo řečeno, tak backend aplikace je pokryta integračními a unit testy. K tomu jsou v této práci použity knihovny Mocha a Chai.

Mocha je testovací framework, který může být spuštěn v Node.js i v browseru. Mocha poskytuje funkcionalitu umožňující testování synchronního i asynchronního kódu. Spolu s Mocha je ale potřeba také použít knihovnu pro porovnání výsledků (assertion library). K tomu je použita knihovna Chai.

6.1.1 Integrační testy

Integrační testy pokrývají na backendu všechny controllery se všemi používanými akcemi. Jako příklad implementace je použita nejpoužívanější akce *Login*.

Nejprve je nutné definovat takzvaný test suit, který poté bude obsahovat jednotlivé testy (test cases). K vytvoření test suit se v Chai používá metoda *describe* a k vytvoření test case metoda *it*. To vše jde do sebe vnořovat jako callback funkce. Každý test case obdrží v callback funkci metodu *done*, která je zavolána na konci každého testu.

6. TESTOVÁNÍ

```
describe('AuthController tests', () => {
  describe('test login process', () => {
    it('it should login and receive token', (done) => {})
  })
})
```

Aby všechny testy byly na sobě navzájem nezávislé, tak je potřeba, aby měly stejná počáteční data, i pokud je test v rámci svého procesu modifikuje. K tomu se nejčastěji používají metody *before* a *beforeEach*. Metoda *before* je spuštěna pouze jednou v rámci své scope, zatímco *beforeEach* před každým jednotlivým testem. V metodě *before* je většinou v této aplikaci vytvářen uživatel, který poté využívá všechny endpointy. Ten se totiž ve většině testů pouze využívá, ale nemodifikuje.

```
before((done) => {
  const userData = { data testovacího uživatele }
  User.deleteMany({}) // smazání všech uživatelů
  .then(() => {
    return User.create(userData)
  })
  .then(() => {
    done()
  })
  .catch((err) => {
    throw err
  })
})
```

Nyní už je v databázi připraven uživatel, pomocí kterého se může otestovat přihlašovací proces. K tomu je využita metoda *getRequest*, která umožňuje odeslat libovolný požadavek. Na ni v případě přihlášení zavoláme metodu *post*, která přijme jako parametr cestu daného endpointu. V poslední metodě *send* zašleme data potřebná k přihlášení. Poté je už pouze nutné definovat metodu *end*, která jako parametr přijímá callback funkci, která bude zavolána po dokončení požadavku. Tato callback funkce přijme atributy *error* a *response*. V případě chyby jsou v atributu *error* zaslány podrobnosti o příslušném neúspěšném požadavku, zatímco v *response* je zaslána odpověď na validní požadavek. Tato odpověď je následně ověřena pomocí Chai na to, zda obsahuje token, který má v případě úspěšného přihlášení přijít. Na samém konci nesmíme zapomenout zavolat metodu *done*.


```
describe('test login process', () => {
  it('it should login and receive token', (done) => {
    getRequest()
      .post('/auth/login')
      .send({ username: 'user1', password: 'test1' })
      .end((err, res) => {
        res.should.have.status(200)
        res.body.should.be.a('Object')
        expect(res.body.token).to.be.a('string')
        done()
      })
  })
})
```

Podobným způsobem jsou testovány všechny ostatní controllery. Jelikož je ale u většiny z nich potřeba přihlášení, tak je předchozí logika přenesena do pomocné funkce *loginAs*, která provádí přihlášení před každým testem.

6.1.2 Unit testy

Unit testy využívají Mocha a Chai stejně jako integrační testy. Unit testy pokrývají veškerou logiku jako konvertování jednotek, parsery, atd. Následující ukázka zobrazuje jeden z testů JSON parseru.

```
it('Test Valid JSON ID parser', function() {
  const body = '{"id": "idval", "key": {"value": "4"}}'
  const result = JsonIdParser(
    sensorIdMappers,
    JSON.parse(body),
  )
  assert.equal(result, 'idvalue')
})
```

6.2 Testování frontend aplikace

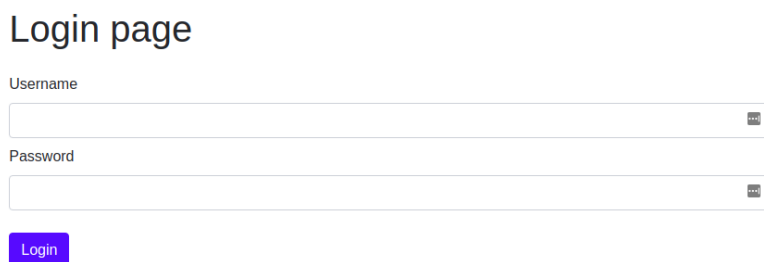
Frontend aplikace je složena z mnoha React komponent. K jejich otestování jsou použity snapshot testy. Pomocné funkce jsou otestovány pomocí klasických Unit testů.

6.2.1 Snapshot testy

Snapshot testy slouží k testování React komponent. Pro snapshot testování se využívá testovací knihovna Jest a `react-test-renderer`, který slouží k vykreslení komponenty, aniž by byla potřeba DOM.

Snapshot testy fungují tak, že při prvním spuštění se vytvoří ve složce `__snapshots__` soubor, který obsahuje vykreslenou React komponentu v HTML. Vývojář ověří, zda je vygenerovaná správně. Pokud poté dojde k její změně, tak je hlídáno, zda je vygenerovaná opět ve stejné podobě. Pokud však provedeme chtěnou změnu, tak musíme daný snapshot přegenerovat. K tomu slouží Jest příkaz `jest updateSnapshot`.

Obrázek 6.1: Přihlašovací komponenta



The image shows a simple login form titled "Login page". It contains two input fields: "Username" and "Password", each with a small eye icon on the right side. Below the fields is a blue "Login" button.

Pro ukázkou byl vybrán jeden krátký snapshot test přihlašovací stránky. Pro již zhotovenou React komponentu s názvem `Login` (na obrázku 6.1), obsahující jednoduchý formulář se vstupem pro přihlašovací jméno a heslo, bude v Jestové defaultní složce `__tests__` vytvořen soubor `login.react.test.js`. Poté je nutné nainportovat React, výše zmíněný renderer a komponentu `Login`. Poté stačí pouze pomocí metody `create` vykreslit komponentu a porovnat ji s již uloženým snapshotem.

```
import React from 'react'
import renderer from 'react-test-renderer'
import Login from '../src/user/login/Login'

it('renders correctly', () => {
  const tree = renderer.create(<Login />).toJSON()
  expect(tree).toMatchSnapshot()
})
```

6.2.2 Unit testy

Ačkoliv většina logiky se odehrává na API, některé funkce frontendové aplikace vykonávají logiku, kterou je potřeba otestovat.

Jako příklad frontedového unit testu je případ, kdy je potřeba imutabilně modifikovat pole jednotek metriky. K tomu slouží funkce *immutableEditUnit*, která jako parametry přijímá pole jednotek metriky, jednotku, kterou chceme modifikovat a na kterém indexu pole. Následně se musí otestovat nejen požadovaný výstup, ale i jestli nebyl modifikován objekt vstupního pole.

```
it('immutable edit of unit on specific index', () => {
  const units = [
    { name: '1', anchor: false, to_anchor: 0.2, },
    { name: 'test2', anchor: true, to_anchor: 1, },
  ]

  const unitForEdit =
    { name: 'test2', anchor: false, to_anchor: 0.1, }

  const expectedResult = [
    { name: '1', anchor: false, to_anchor: 0.2, },
    { name: 'test2', anchor: false, to_anchor: 0.1, },
  ]

  expect(immutableEditUnit(units, unitForEdit, 1))
    .toEqual(expectedResult)
  expect(units[1].anchor).toBe(true)
  expect(units[1].to_anchor).toBe(1)
  expect(units[1].name).toBe('test2')
})
```

6.3 Test rychlosti

Jako poslední byl prováděn test rychlosti. Na API bylo sledováno, jak rychle budou požadavky obsluhovány. Na to může mít vliv počet požadavků za sekundu a také velikost databáze. Na frontendu se testoval výkon při větším množství dat, které může mít vliv na vykreslování grafů a plynulost celé aplikace.

6.3.1 Frontend

Testování plynulosti frontendové aplikace bylo zaměřeno hlavně na to, jak budou prezentační funkce jako grafy a export do CSV plynulé při větším množství dat. Při tomto testování bylo odhaleno, že knihovna pro vykreslování grafů Recharts má výkonnostní problémy při větším množství dat (řádově při pěti tisících a výše). To mělo za následek velké množství výpočetních operací. Z toho důvodu byly od určitého počtu dat (tři tisíce) vypnuty všechny animace a různé designové funkce knihovny. To vedlo k mnohem větší plynulosti vykreslování. Export do CSV a další funkce nebyly velkým počtem dat zásadně ovlivněny.

6.3.2 Backend

U dat přicházejících ze sensorů lze očekávat jejich velký nárůst. Z toho důvodu bylo otestováno, jak bude API pracovat, až data budou mít větší množství položek. Přesněji bylo zkoumáno, jak dlouho bude API trvat zpracovat požadavek. Data se totiž nenačtou pouze z databáze a následně neodešlou, ale může na ně být aplikována celá řada funkcí a filtrů. Mezi ně patří:

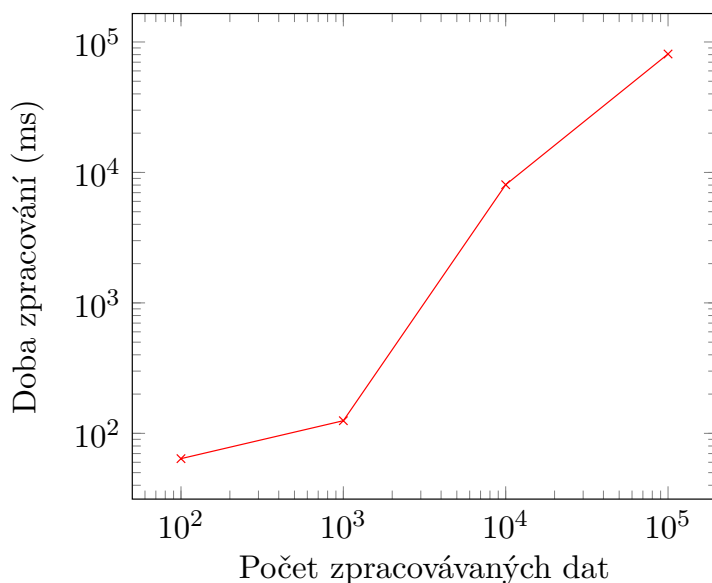
1. Očištění samotných dat od databázového identifikátoru a dalších pro frontend aplikaci nepotřebných údajů.
2. Převedení dat na stejnou jednotku (potřebné u vykreslení grafu).
3. Vyfiltrování dat za určitý časový horizont.
4. Stručná charakteristika dat (minimum, maximum a průměr).

Byla zkoumána doba zpracování požadavku API, a to ve výpočetně nejnáročnějším případě. To je, když jsou aplikovány všechny výše zmíněné filtry a funkce. Měření probíhalo na lokálním prostředí.

Schopnost backendu shromažďovat data byla zároveň otestována pomocí senzoru, který poskytl vedoucí práce.

6.3.3 Test detailu senzoru

Na detailu senzoru (obrázek 5.2) se provedla měření s počtem až do sto tisíc položek dat stejné metriky, stejného senzoru a byl zaznamenáván čas. Měření byla u každého množství (100, 1000, 10 000 a 100 000 položek) prováděna desetkrát bez jakékoliv cache a následně zprůměrována. Jako první byl testován endpoint pro získání všech dat za časový horizont jednoho roku převedených na centimetry `/data/sensorId/metricId?convertUnit=cm&lastDate=years`.



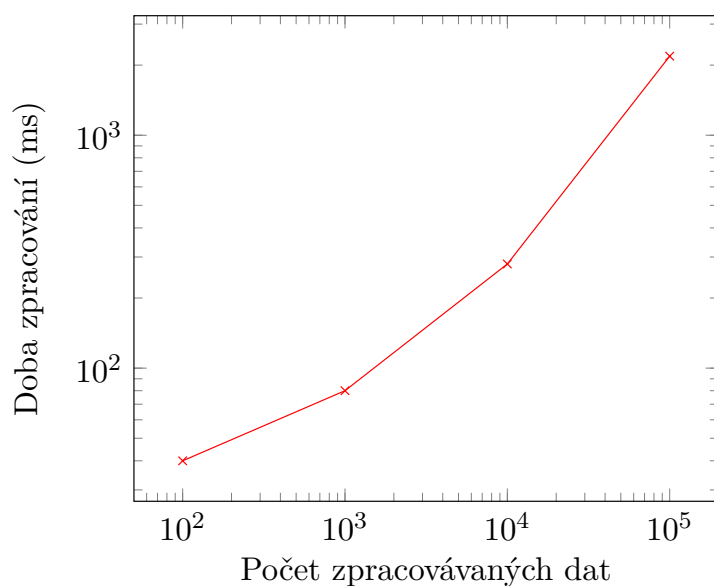
Z výše uvedeného grafu naměřených dat je vidět zhruba lineární nárůst, což je oproti čtení těchto dat z databáze podstatné zhoršení. Jeden z důvodů je, že všechny filtry a funkce se aplikují na sobě nezávisle, a tak se provádí mnoho iterací nad těmito daty. To bylo navrženo, aby se funkce mohly aplikovat v libovolném pořadí a počtu.

Frontend aplikace si však s tímto problémem poradila, jelikož všechna data se načítají asynchronně a uživatel si počká pouze na načtení dílčích částí. V budoucnu je ale rozhodně prostor pro lepší optimalizaci. Dále je nutno podotknout, že dotaz na data senzoru překračující počet několika tisíc položek není pravděpodobný.

6.3.4 Test mapy

Druhý test byl proveden u endpointů vracejících všechny senzory a základní statistiky jejich dat (průměr, maximum, minimum), které jsou využity například u mapy (obrázek 5.3). Už při začátku testu bylo odhaleno, že řešení je absolutně nevyhovující. Začala se zpracovávat data všech senzorů, i když nebyla uživatelem označena pro zobrazení detailu. Při deseti senzorech o 1000 položek dat (každý) trvalo načtení přibližně 30 sekund. Dále se v testu tedy nepokračovalo a přepracoval se návrh celého procesu.

Ve druhé verzi byl pozměněn kompletně proces získání senzorů a dat. Nejprve si frontend aplikace požádá o všechny požadované senzory. Poté, co uživatel označí senzor na mapě pro detail, tak se asynchronně odešle požadavek o data vybraného senzoru. Toto řešení se ukázalo jako ideální a testovatelné. Dále se tedy podobně jako u testu detailu provedl test rychlosti na vzorcích 100, 1000, 10 000 a 100 000 položek. Provolávaný endpoint je `/data/overview/metricId/sensorId`.



Díky menšímu počtu možných filtrů dosahuje zkoumaný endpoint lepších výsledků než ten z předešlého testu a spolu s novým řešením celého procesu je výsledek uspokojivý.

Závěr

Byly vytvořeny dvě aplikace umožňující správu a konfiguraci IoT projektů. První z nich je navržena v Node.js a je schopna shromažďovat příchozí data ve formátech JSON, text a v binárním formátu, který je reprezentován hexadecimálně. Tato příchozí data jsou ukládána pomocí MongoDB databáze. Aplikace umožňuje vytvářet a modifikovat informace o skupinách, uživateli, senzorech, metrikách a mapperech, které umožňují příchozí data zpracovávat. Dále je umožněno data zobrazovat jen povolaným osobám v rámci jejich ranku a skupin. Tato aplikace je obsluhována jako RESTful API. Druhá aplikace toto RESTful API využívá a umožňuje jeho snadné používání pomocí interaktivního UI implementovaného v Next.js a React. Zároveň prezentuje výsledná data pomocí grafů a mapy. Obě aplikace jsou otestovány tak, jak je definováno v zadání. Zároveň proběhl test použitím skutečného senzoru, který poskytl vedoucí práce.

Tyto aplikace mohou být dále rozšiřovány o další funkcionality. V budoucnu je například možnost umožnit zpracovávat další formáty jako XML a další reprezentace binárních dat, či optimalizovat výkon zpracování dat. Dále je možné rozšířit aplikaci o možnost, existence více správců, kteří mohou spravovat své senzory, aniž by do nich ostatní správci mohli zasahovat, či zpřístupnit tuto možnost dokonce uživatelům.

Literatura

- [1] Node.js - Introduction. [online], 2019, [cit. 2019-02-12]. Dostupné z: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm
- [2] React lifecycle methods diagram. [online], 2019, [cit. 2019-02-12]. Dostupné z: <http://projects.wojtekmaaj.pl/react-lifecycle-methods-diagram/>
- [3] Thinger.io. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://thinger.io/>
- [4] Top 15 Sensor Types Being Used in IoT. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://www.ibm.com/cz-cs/internet-of-things/solutions/iot-platform/watson-iot-platform>
- [5] MQTT messaging. [online], 2019, [cit. 2019-02-12]. Dostupné z: https://console.bluemix.net/docs/services/IoT/reference/mqtt/index.html?pos=2&cm_mc_uid=73631728288815474708924&cm_mc_sid_50200000=15790891548158405640#message-payload&cm_sp=dw-bluemix--nospace--answers
- [6] AWS IoT Device Management. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://aws.amazon.com/iot-device-management/>
- [7] How AWS IoT Works. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html>
- [8] Azure IoT Hub. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://azure.microsoft.com/cs-cz/services/iot-hub/>

LITERATURA

- [9] Sharma, R.: Top 15 Sensor Types Being Used in IoT. [online], 2018, [cit. 2019-02-12]. Dostupné z: <https://www.finoit.com/blog/top-15-sensor-types-used-iot/>
- [10] doc. Ing. Tomáš Vitvar, P.: Web 2.0 REST Architecture. [Neveřejná přednáška], 2018, [cit. 2019-01-06].
- [11] Malý, M.: Node.js – s JavaScriptem na server. [online], 2010, [cit. 2019-02-12]. Dostupné z: <https://www.zdrojak.cz/clanky/node-js-s-javascriptem-na-server/>
- [12] Irena, H.: *Big Data a NoSQL databáze*. Grada Publishing, a.s., listopad 2015, ISBN 978-80-247-5466-6.
- [13] React úvod. [online], 2016, [cit. 2019-02-12]. Dostupné z: <https://www.dzejes.cz/react-uvod.html>
- [14] Randus, T.: Redux + React – Redux. [online], 2016, [cit. 2019-02-12]. Dostupné z: <https://www.zdrojak.cz/clanky/redux-react-23-redux/>
- [15] Bittner, J.: Úvod do CSS preprocesoru Sass. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://www.itnetwork.cz/html-css/webove-portfolio/tutorial-moderni-webove-portfolio-sass>
- [16] Google maps platform. [online], 2019, [cit. 2019-03-02]. Dostupné z: <https://cloud.google.com/maps-platform/>
- [17] Mapbox. [online], 2019, [cit. 2019-03-02]. Dostupné z: <https://mapbox.com/>
- [18] RFC 7519. [online], 2015, [cit. 2019-02-12]. Dostupné z: <https://tools.ietf.org/html/rfc7519/>
- [19] Json web token. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://jwt.io/introduction/>
- [20] Bearer - authentication. [online], 2019, [cit. 2019-02-12]. Dostupné z: <https://swagger.io/docs/specification/authentication/bearer-authentication/>

Seznam použitých zkratk

API Application Programming Interface

CSS Cascading Style Sheets

CSV Comma-separated values

JSON JavaScript Object Notation

SASS Syntactically Awesome Style Sheets

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

HTML Hypertext Markup Language

IoT Internet of Things

CI Continuous integration

BSON Binary JavaScript Object Notation

UI User interface

DOM Document Object Model

JWT JSON web token

URL Uniform Resource Locator

MVC Model-view-controller

Obsah přiloženého CD

readme.txt.....	Popis obsahu CD
src	Složka se zdrojovými kódy
├─ README.....	Manuál k instalaci a obsluze aplikací
├─ frontend.....	Zdrojové kódy frontend aplikace
├─ backend	Zdrojové kódy backend aplikace
├─ thesis.....	Složka s \LaTeX zdrojovými kódy práce
text	Text diplomové práce
├─ thesis.pdf.....	Text diplomové práce v PDF formátu