VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

WEBOVÉ ROZHRANÍ PRO KOMUNIKACI S UZLY BEZDRÁTOVÉ SENZOROVÉ SÍTĚ

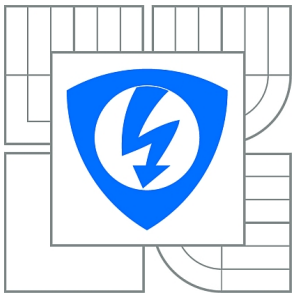
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. PETR ČERNOCKÝ
AUTHOR

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

WEBOVÉ ROZHRANÍ PRO KOMUNIKACI S UZLY BEZDRÁTOVÉ SENZOROVÉ SÍTĚ

WEB USER INTERFACE FOR COMMUNICATING WITH THE WIRELESS SENSOR NODES

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

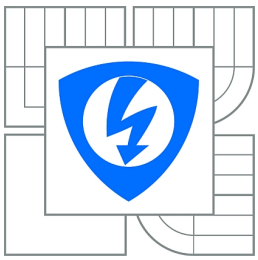
AUTOR PRÁCE — Bc. PETR ČERNOCKÝ

AUTHOR

VEDOUCÍ PRÁCE — Ing. VLADIMÍR ČERVENKA

SUPERVISOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Petr Černocký	***ID:***	73072
Ročník:	2	***Akademický rok:***	2010/2011

NÁZEV TÉMATU:

Webové rozhraní pro komunikaci s uzly bezdrátové senzorové sítě

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je prostudovat problematiku SQL databází a přístupu k datům pomocí REST architektury. Navrhnout a vytvořit webovou aplikaci, která bude zajišťovat vizualizaci získaných dat ze vzdálené SQL databáze i přímo z jednotlivých uzlů sítě. Aplikace bude schopna načítat podkladové materiály ve vektorovém formátu, vizualizovat naměřená data v podobě grafu a ukládat do databáze relativní polohu jednotlivých uzlů v prostoru. Zobrazované informace bude možné vymezit datem a dále pak exportovat do souboru.

DOPORUČENÁ LITERATURA:

[1] DUNKELS, Adam; VASSEUR, Jean-Philippe. Interconnecting Smart Objects with IP : The Next Internet . California : Morgan Kaufmann, 2010. 432 s. ISBN 0123751659.

[2] The Contiki Operating System [online]. 2010-09-06 [cit. 2010-10-11]. Dostupné z WWW: <http://www.sics.se/~adam/contiki/docs/main.html>.

Termín zadání:	7.2.2011	***Termín odevzdání:***	26.5.2011

Vedoucí práce: Ing. Vladimír Červenka

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Cílem této diplomové práce je navrhnout a vytvořit webovou aplikaci, která bude sloužit k analýze a dohledu bezdrátových senzorových sítí. Její hlavní funkcí je načtení mapového podkladu, ve kterém je zobrazena poloha jednotlivých uzlů sítě. Těm může být následně přiřazeno, jakou měřenou veličinu budou zobrazovat. Naměřené údaje mohou být získávány přímo od senzorových uzlů nebo prostřednictvím databázového serveru. Zobrazení údajů za uživatelem definované časové období je možné prostřednictvím implementovaného grafu. Aplikace samotná je naprogramována v jazyce JavaFX podle pravidel REST architektury. Pro zjednodušení komunikace s databázovým serverem je využit mapovací nástroj MyBatis.

KLÍČOVÁ SLOVA

Bezdrátová senzorová síť, JavaFX, MyBatis, REST, JSON, webová aplikace

ABSTRACT

The aim of this master's thesis is to design and create web application that allows analysing and monitoring wireless sensor networks. The main function is to load plan of given area and shown position of the sensor network nodes. These nodes allow user to set which measured phenomenon will be display on them. The measured values can be loaded directly from sensor nodes or from database server. Visualized values for user defined time period can be displayed in the implemented graph. The application itself is programmed in JavaFX language and follows the rules of REST architecture. For easier communication with database server application uses the MyBatis framework.

KEYWORDS

Wireless sensor network, JavaFX, MyBatis, REST, JSON, web application

ČERNOCKÝ P. *Webové rozhraní pro komunikaci s uzly bezdrátové senzorové sítě*. Brno: Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. 2011. 59 s. Vedoucí diplomové práce Ing. Vladimír Červenka.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Webové rozhraní pro komunikaci s uzly bezdrátové senzorové sítě“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

(podpis autora)

Poděkování

Děkuji vedoucímu práce Ing. Vladimíru Červenkovi za užitečnou metodickou pomoc a cenné rady při zpracování diplomové práce.

V Brně dne ....

podpis autora

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Díky stále se zdokonalujícím technologiím a miniaturizaci v oblasti mikrokontrolerů a pamětí, vzniká v posledních letech odvětví, jehož možnosti se do budoucna jeví jako neomezené. Tímto odvětvím jsou bezdrátové senzorové sítě (WSN). Tyto sítě bývají složené z většího množství miniaturních bezdrátových senzorových uzlů, které jsou na svou velikost poměrně výkonné. Oproti jiným odvětvím, zde však neplatí paradigma „čím výkonnější tím lepší“, ale naopak i výkon samotný je podřízen jinému požadavku, kterým je co nejnižší spotřeba elektrické energie. Té jsou podřízeny nejen hardwarové, ale také softwarové prvky celé sítě. Abychom i my webovou aplikací dodrželi nízkoenergetickou „symbiózu“ všech prvků systému, musí být pečlivě zváženo jaká architektura a protokoly budou použity při komunikaci se senzorovými uzly.

Jelikož senzorové sítě často pokrývají svou rozlohou celé budovy či otevřená prostranství o velké rozloze, je při jejich správě a dohledu nad nimi velice důležité vědět jak jsou rozmístěny a kde se konkrétní uzly nacházejí. K tomuto účelu bude sloužit navrhovaná aplikace, která načte mapový podklad oblasti, ve které bude síť umístěna a zobrazí v něm rozmístění jednotlivých uzlů. Jelikož však cílem této práce není pouhé vytvoření elektronické mapy uzlů, ale nástroje pro analýzu a dohled nad sítí, budou schopnosti aplikace rozšířeny o možnost komunikace s databázovým serverem. Zde budou uloženy veškeré informace o síti a jednotlivých uzlech. Zároveň s nimi se na server budou v předem daných intervalech ukládat i naměřené hodnoty jednotlivých senzorů, což umožní nejen jejich neprodlené zobrazení v aplikaci, ale také zpracování dat za uživatelem zvolené časové období. Pro případy, kdy bude potřeba zjistit skutečně aktuální hodnotu měřené veličiny, bude implementována funkce pro přímou komunikaci se senzorovým uzlem. Ten bude pomocí instalovaného webového serveru schopen reagovat na HTTP dotazy aplikace a vracet zpět požadovanou hodnotu.

Cílem diplomové práce je analyzovat a porovnat možnosti webových služeb, komunikačních protokolů a datových formátů sloužících pro přenos dat počítačovými sítěmi. Nejvhodnější z nich budou následně popsány a implementovány do jazyka JavaFX, ve kterém bude celá aplikace napsána. Následující kapitoly pak budou věnovány rozdělení aplikace na jednotlivé části a jejich detailnímu popisu. Největší důraz bude kladen na princip fungování aplikace, použité metody a komunikační rozhraní, které je pro její správné fungování stěžejní.

1 WEBOVÉ SLUŽBY

Pojem webové služby lze chápat jako sadu technik sloužících pro vývoj distribuovaných aplikací využívajících pro svou funkci webových protokolů (např. HTTP). V dnešní době jsou webové služby, specifikované skupinou W3C v [1], děleny do dvou hlavních kategorií na SOAP (Simple Object Access Protocol) a REST (Representational State Transfer).

1.1 REST vs. SOAP

Přestože oba přístupy využívají pro přenos dat protokol HTTP, značně se odlišují v jeho používání. Na rozdíl od SOAP, které umožňuje definici vlastních metod, mají webové služby používající principy REST architektury (tzv. RESTful služby) jasně dané metody (GET, PUT, POST, DELETE), které pro komunikaci používají.

Jak je v [2] podrobněji popsáno, hlavní nevýhodou SOAP oproti RESTu je jeho těžkopádnost. Chce-li například odesílatel poslat příjemci SOAP zprávu, musí její obsah zabalit do HTTP protokolu, protože jinak by se mohlo stát, že některé firewally by samotnou SOAP zprávu nepropustily. S tímto souvisí také problematika přenosu dat v jiném formátu než XML. Pokud bude aplikace používat dnes stále populárnější formát JSON, nelze jím XML jednoduše nahradit, ale musí dojít ke vložení JSON souboru do XML zprávy a až poté je možné ji odeslat. Oproti tomu REST, založený čistě na HTTP protokolu, nabízí sto procentní průchodnost firewally a možnost výběru formátu přenášených dat deklarovanou již v hlavičce každé zprávy.

Hlavní výhodou webových služeb založených na SOAP je jejich velké rozšíření, s čímž souvisí i jejich vysoká podpora vývojářů a firem, které nabízejí programy značně zjednodušující vývoj webových aplikací.

1.2 Shrnutí

Webové služby založené na výše popsaných přístupech mají své výhody i nevýhody. Proto i zde, jako u vývoje jakékoliv jiné aplikace, je při rozhodování hlavní, kde bude daná služba použita. V tomto případě je rozhodování zjednodušeno faktem, že aplikace má sloužit pro komunikaci s uzly bezdrátové senzorové sítě, jejichž hlavním požadavkem je co nejnižší energetická náročnost.

Vzhledem k výše uvedenému je jasné, že v zájmu úspory energie musí být redundance přenášených zpráv co nejmenší a operace s daty (např. jejich parsování) omezeny na nutné minimum.

Jak již bylo dokázáno v [3], má RESTful webová služba (běžící na operačním systému Contiki) při provedení stejné operace přibližně 2,5krát nižší spotřebu energie a čas nutný pro dokončení operace je téměř 4krát nižší. Díky těmto faktům a již provedeným experimentům budou podrobněji popsány a následně implementovány služby založené na REST architektuře.

1.3 REST

Representational State Transfer je na technologii nezávislá architektura, obsahující soubor pravidel používaných při komunikaci v distribuovaném prostředí. Každý systém splňující daná pravidla nazýváme RESTful systémem.

Díky své jednoduchosti a využívání HTTP protokolu pro přenos zpráv, je myšlenka RESTful systémů často prezentována na principu Webu. Do webového prohlížeče (klient) se zadá URL adresa s požadovaným obsahem a pomocí HTTP protokolu se odešle webovému serveru (server). Ten dotaz zpracuje a v odpovědi vrátí dokument ukrývající se pod zadanou URL adresou. Aby bylo možné takto jednoduše přenášet data mezi účastníky komunikace, je nutné se při vývoji RESTful systémů držet základních pravidel a doporučení. Níže uvedené parametry jsou základem REST architektury podle [4].

- *Klient-server* - oddělení uživatelského rozhraní od serveru značně přispívá k mobilitě klienta skrz různé platformy a vylepšuje škálovatelnost zjednodušením komponent serveru.

- *Bezestavovost* - výrazně přispívá k rychlejší obsluze požadavků klientů a uvolňování zdrojů. Hlavní nevýhodu je vytěžování linky díky nutnosti opětovně vyslat celou sérii dat i v případě chyby pouze několika bitů.

- *Vyrovnávací paměť* - některé odpovědi generované serverem mohou být označeny příznakem cacheable, což umožňuje jejich uložení do vyrovnávací paměti a pozdější použití v případě volání stejného požadavku.

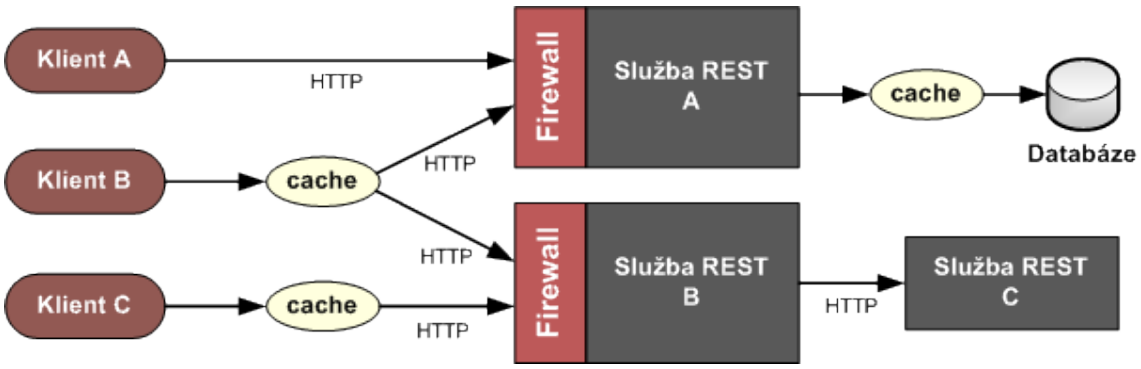
- *Jednotné rozhraní* - přispívá k celkovému zjednodušení architektury a předchází vzniku problémů při komunikaci. Příkladem může být aplikace využívající pro získávání dat ze serveru pouze HTTP příkazy GET, POST, PUT a DELETE. Jeho hlavní nevýhodou je degradace efektivity, způsobená nutností převádět data získaná z aplikace do standardizované formy vhodné k přenosu.

- *Systém vrstev* - stejně jako většina jiných architektur využívá i REST pro zjednodušení implementace hierarchický systém vrstev. Díky tomu může každá

vrstva komunikovat pouze s vrstvou sobě podřízenou a výsledky své práce předávat vrstvě nadřazené. Proti zřejmým výhodám toho uspořádání stojí zpoždění způsobené průchodem dat jednotlivými vrstvami.

- *Kód na vyžádání* - umožňuje klientovi vyžádat si od serveru kód v podobě appletu nebo skriptu a ten následně vykonat. Přínos této operace je především v přenesení zátěže ze serveru na klienta. Jelikož však tato možnost značně nabourává požadavek na průhlednost komunikace, je jako jediná z předešlých pouze volitelná.

Pro lepší představu je na obrázku 1.1 ukázáno, jak může vypadat síť při aplikaci pravidel REST architektury podle [5]. Jak je z něj patrné jsou použity vyrovnávací paměti jak pro klienty, tak pro databázi. Služby REST běžící na serveru jsou chráněny firewally, díky kterým dochází k zahazování nechtěného provozu, což zde díky použitému protokolu HTTP nehrozí. Důležitá vlastnost, která nebyla zatím uvedena, je schopnost služeb volat jiné služby, což částečně umožňuje přenést zátěž z klienta na server.



Obr. 1.1: Příklad sítě při aplikaci REST architektury.

V aplikacích využívajících REST architekturu jsou podle [6] klíčové následující principy práce se zdroji.

- *Identifikace zdrojů* - jednotlivé zdroje jsou v žádosti identifikovány pomocí URL adres. Jako takové jsou požadované zdroje odděleny od jejich reprezentace, která je odeslána klientovi. Například pošle-li klient na server žádost s adresou databáze, nedostane v odpovědi celou databázi, ale pouze reprezentaci požadovaných záznamů v podobě HTML, XML nebo JSON dokumentu.

- *Manipulace se zdroji skrze jejich reprezentace* - pokud klient obdrží reprezentaci zdroje včetně přiložených metadat, má dostatek informací k tomu, aby

mohl data zdroje ze serveru smazat nebo je modifikovat. To lze však učinit pouze za předpokladu, že k tomu má příslušná oprávnění.

- *Hlavička zprávy* - každá zpráva vrácená ke klientovi obsahuje hlavičku, ve které je popsáno jak danou zprávu správně zpracovat. Například informace jaký parser má být použít pro získání dat ze zprávy, je uvedena v poli Internet media type.

- *Odkazy na související zdroje* - pokud je pravděpodobné, že klient bude chtít přistupovat ke zdrojům souvisejícím s aktuální reprezentací, měly by v ní být uvedeny jejich URL adresy.

Z výše uvedených vlastností REST architektury podle [7] vyplývá, že aplikace navržené v jejím duchu se vyznačují velkou flexibilitou formátu přenášených dat a účinným využitím možností HTTP protokolu. To má za následek zvýšení efektivity komunikace klienta se serverem a s tím spojenou i jejich menší zátěž.

2 DATOVÉ FORMÁTY PRO PŘENOS DAT

Pokud dojde při tvorbě webové aplikace na potřebu přenosu dat v uspořádané struktuře, nabízí se hned několik datových formátů, které mohou být použity. Mezi dva dnes nejrozšířenější patří JSON (JavaScript Object Notation) a XML (Extensible Markup Language).

2.1 JSON vs. XML

Jednou ze společných výhod těchto formátů oproti ostatním je jejich přehlednost a čitelnost jak pro počítače, tak i pro lidské oko.

První z výše jmenovaných formátů JSON, je sice derivátem programovacího jazyka JavaScript, ale i přes tuto příbuznost je díky velkému množství nástrojů použitelných pro jeho zpracování jazykově nezávislý. Právě tato nezávislost mu zajišťuje stále větší oblibu na úkor rozšířenějšího XML [9]. Hlavním důvodem náskoku XML je jeho dřívější uvedení na trh a fakt, že byl již od svého počátku vyvíjen jako platformě nezávislý jazyk, díky čemuž je dnes jeho podpora obrovská.

Jak vyplývá z porovnání v [10], největší výhodou JSON je jeho menší velikost při stejném objemu přenášených dat a s tím spojená i vyšší rychlost jejich zpracování. To je dáno nutností zpracovat XML dokument předtím, než mohou být data v něm obsažená použita. Tato robustnost XML je však zcela záměrná a má za následek jeho větší bezpečnost a také možnosti přizpůsobení potřebám dané aplikace.

2.2 Shrnutí

Stejně jako při porovnání REST a SOAP, i zde byla zvolena sice ne tak zavedená, ale zato jednoduší varianta, kterou je JSON. Hlavním důvodem tohoto kroku je nutnost použít pro zpracování XML dokumentu „parser“ běžící na uzlu senzorové sítě, což by mělo negativní vliv na spotřebu uzlu. Jelikož uzly senzorové sítě většinou nemají možnost síťového napájení, jeví se odlehčený a na zpracování méně náročný JSON jako vhodnější volba.

2.3 JSON

Po obecném představení vlastností JSON dokumentu bude tato kapitola zaměřena na popis jeho struktury. Tu je možné si prohlédnout na výpisu 2.1.

Výpis 2.1: Ukázka JSON dokumentu

```
{   "Senzor":"Uzel1",
    "Parametry":[
        {
         "Teplota":28
        },
        {
         "Stav":"Zapnuto"
         "Energie":70,
        }
    ]
}
```

Data obsažená v JSON dokumentu mohou být organizována v objektech nebo polích. Každý objekt je ohraničen složenými závorkami ({ }) a obsahuje nesetříděnou množinu párů název/hodnota oddělenou dvojtečkou (:). Oproti tomu polem, ohraničeným hranatými závorkami ([ ]), je označována setříděná kolekce hodnot oddělených čárkami. Příklad objektu a do něj vnořeného pole je uveden na výpisu 2.1. Vstupní data mohou mít téměř libovolný formát (číslo, boolean, řetězec, objekt), avšak výstupem budou vždy data uspořádaná v řetězci. Složitost a počet vnoření vstupních dat není nijak omezen. Všechny zde uvedené parametry a podrobnější popis JSON se nachází v [8].

3 OBJEKTOVĚ-RELAČNÍ MAPOVÁNÍ

Při vytváření aplikací vyžadujících pro svou činnost spolupráci s databází, se často vyskytne nutnost převodu dat uložených v tabulkách na objekty. Tento proces se nazývá objektově-relační mapování (ORM) a více o něm je uvedeno v [15]. Mezi nejčastěji používané rámce sloužící k ORM patří Hibernate, jehož největším konkurentem je MyBatis. Přestože oba vykonávají principiálně stejnou činnost, není zařazení MyBatisu mezi ORM rámce zcela správné, jelikož nemapuje objektový model na relační databázi, ale pouze SQL příkazy na jednoduché objekty v Javě označované jako POJO (Plain Old Java Object).

3.1 MyBatis vs. Hibernate

V dnešní době rozšířenější Hibernate definovaný v [16], je naprogramován v jazyce Java a jeho hlavním úkolem je udržovat data mezi aplikací a databází perzistentní. Perzistence neboli stálost mezi dvěma aplikacemi znamená, že jsou spolu vzájemně provázané a dojde-li ke změně jedné z nich, převede se automaticky tato změna i na druhou. K zajištění tohoto stavu využívají jak Hibernate, tak MyBatis prostředků mapování či anotací.

Hlavní rozdíl mezi oběma metodami je v přístupu k databázi. Hibernate je lepší použít v případě, je-li hlavní důraz při vývoji kladen na aplikaci, podle které je následně vytvořena struktura databáze a naopak MyBatis je vhodnější použít v případě, je-li dána struktura databáze podle které bude vyvíjena aplikace. Tento postup je samozřejmě možné použít i pro Hibernate, ale je nutné počítat s podstatně složitější a časově náročnější implementací. Další podstatný rozdíl je v jazyce, pomocí kterého jsou data získávána. Zatím co MyBatis využívá standardních SQL příkazů, definuje rámec Hibernate vlastní jazyk nazvaný Hibernate Query Language (HQL) odvozený z SQL.

Obecně tedy z porovnání [16] a [17] vypívá, že MyBatis je jednodušší, rychlejší a méně náročný na implementaci než Hibernate, který naopak vyniká svou komplexností podpořenou doplňkovými nástroji a umožňuje uživateli vyhnout se jakémukoliv psaní SQL kódu.

3.2 Shrnutí

Aplikace, která je předmětem této práce, byla vyvíjena s vědomím, že nebude pracovat jako samostatná a nezávislá jednotka, ale bude součástí většího systému určeného

pro práci se senzorovými sítěmi. Tato skutečnost je patrná již ze schématu komunikace se senzorovými uzly, kde jsou data stahována z databázového serveru a je tedy zapotřebí jiné aplikace, která zajistí jejich nahrání. To naznačuje, že hlavním prvkem systému je databáze, na které jsou závislé další aplikace a je tedy nutné, aby nedocházelo k jejím modifikacím, jelikož by mohly způsobit pád celého systému. Z toho vyplývá také hlavní důvod proč byl v práci použit rámec MyBatis, který umožní aplikaci plně přizpůsobit struktuře databáze. Další vlastností této metody, která je často označována za nevýhodnou, je nutnost ručního zadání SQL příkazů pro stahování dat. Zde je tato vlastnost využita spíše ve prospěch aplikace, jelikož umožňuje načítat data po blocích. V nich jsou obsaženy pouze informace, které uživatel aktuálně vyžaduje a nedochází tak ke zbytečné zátěži databáze či aplikace zpracováním nepotřebných dat.

3.3 MyBatis

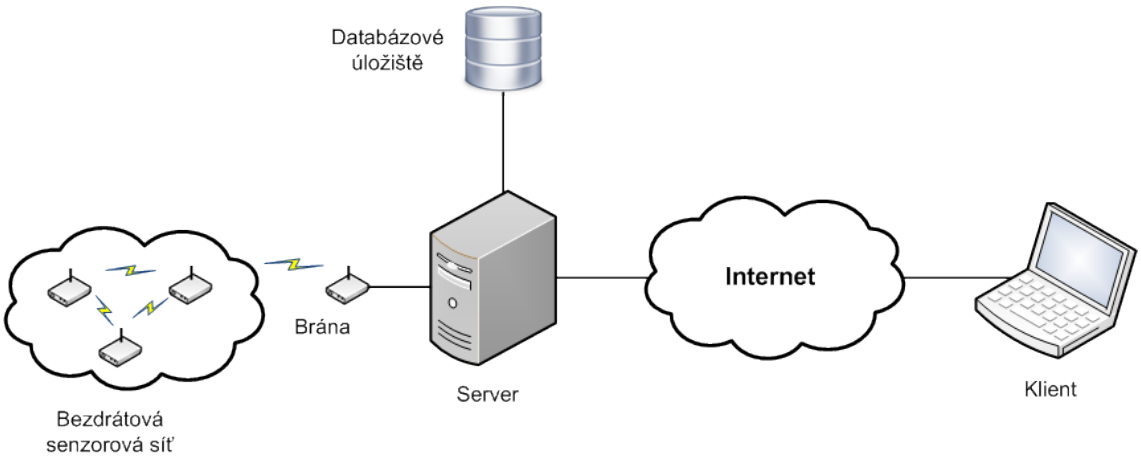
IBatis, který musel být z licenčních důvodů přejmenován na dnešní MyBatis, je aplikační rámec umožňující nejen mapování záznamů SQL databází na Java objekty, ale také na objekty .NET, více viz [17]. Přestože je aplikace napsána v jazyce JavaFX, který MyBatis nepodporuje, využívá pro stahování dat na pozadí vlákno napsané v Javě, které jeho využití umožní. Jelikož ukazovat strukturu Mybatisu na obecných třídách není moc přehledné a všechny důležité vlastnosti byly již popsány v předešlém srovnání, bude jeho podrobnější popis obsahem kapitoly 6.2.2. Zde bude jeho funkce podrobně vysvětlena na reálném příkladu použitém v této aplikaci.

4 APLIKACE

Nástin funkčnosti webové aplikace byl uveden již v úvodu. V této kapitole bude tedy pozornost zaměřena na její zařazení do přenosového řetězce a popis struktury.

Jazyk JavaFX, ve kterém je aplikace napsána, rozšiřuje její vlastnosti o multiplatformnost. Díky tomu ji bude možné spouštět nejen ve webovém prohlížeči, ale také jako desktopovou aplikaci pod operačními systémy Windows XP/Vista a MacOS. Další možností je zobrazení v mobilních telefonech, které JavaFX také podporuje. Ty však nejsou pro účely zde vyvinuté aplikace vhodné.

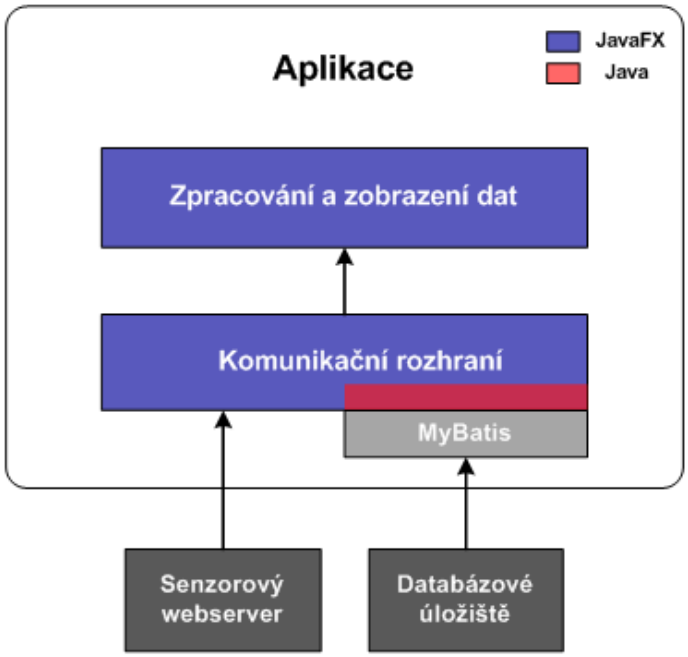
Struktura celého přenosového řetězce, začínajícího uzly bezdrátové senzorové sítě a končícího webovou aplikací běžící na klientské stanici, je zobrazena na obrázku 4.1. Jak je z ní patrné, může komunikace probíhat přímo se senzorovými uzly prostřednictvím brány, nebo s databázovým úložištěm na které jsou měřené hodnoty průběžně ukládány.



Obr. 4.1: Struktura přenosového řetězce.

Jelikož tvorba aplikace je dosti náročný proces, při kterém může dojít k velkému množství chyb, bude rozdělena na dvě části zobrazené na obrázku 4.2. První z nich, pojmenovaná „Zpracování a zobrazení dat“, popíše všechny funkce aplikace z hlediska jejich principu a použitých řešení. Ve druhé části, jak už její název „Komunikační rozhraní“ napovídá, se pozornost zaměří na komunikační kanály aplikace a představení nástrojů, které jejich implementaci usnadňují. Pro lepší přehlednost dojde k oddělené obou částí pomocí tříd tak, aby docházelo k co nejmenšímu počtu možných kolizních míst a data mezi nimi byla předávána hierarchicky. I když se toto rozdělení může zdát jako velice hrubé, jsou spolu všechny prvky aplikace provázány

tak pevně, že rozdělení na více vrstev by bylo složité a stejně by se jejich popis prolínal.



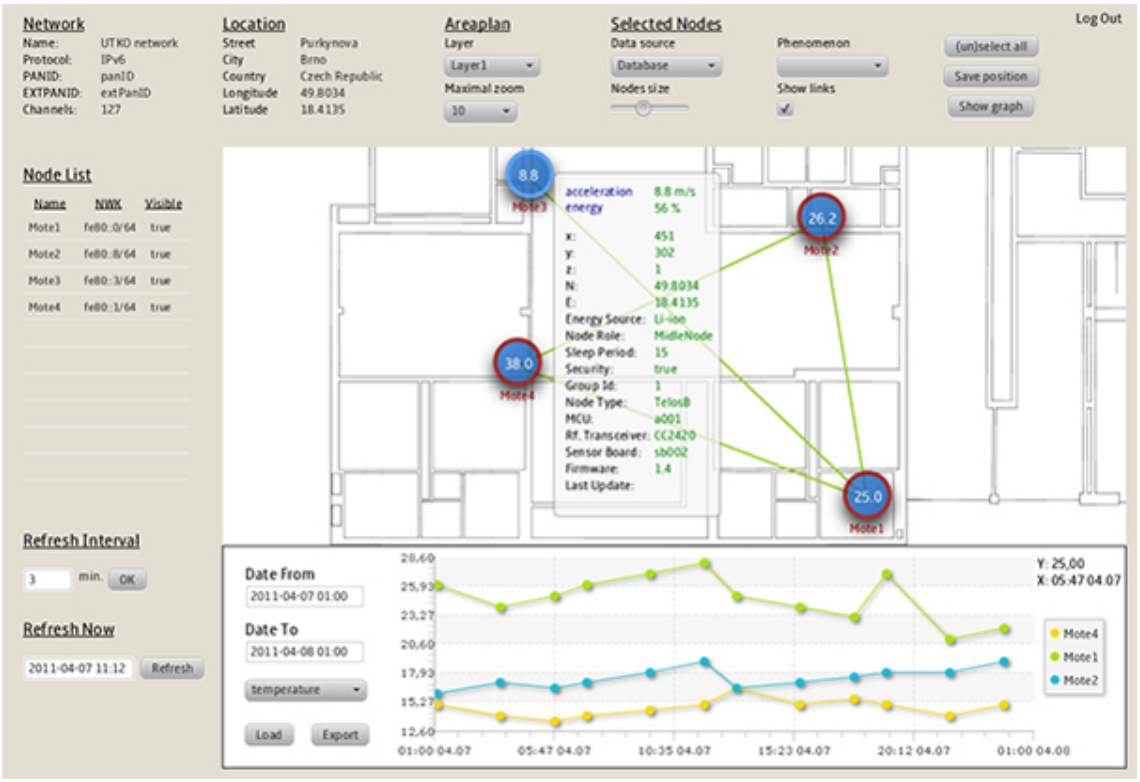
Obr. 4.2: Rozdělení aplikace do vrstev.

5 ZPRACOVÁNÍ A ZOBRAZENÍ DAT

Úkolem této vrstvy bude přijímat příkazy od uživatele, zpracovávat požadovaná data a výsledky přehledně prezentovat uživateli pomocí grafických prvků aplikace. Prezentace výsledků se bude odvíjet od charakteru dat, tedy statické zůstanou zobrazeny po celou dobu běhu aplikace v neměnných tabulkách a dynamické budou vykreslovány do grafů, či aktualizovány v uživatelem definovaných intervalech.

Obrázek 5.1 zobrazuje grafické prostředí aplikace, které je koncipováno jako jednostránkové. To znamená, že všechny důležité údaje a nastavení jsou zobrazeny na jedné stránce a není potřeba mezi nimi přepínat pomocí záložek. Toto rozložení výrazně ulehčuje práci s aplikací a zajišťuje neustálý přehled nad děním v připojené senzorové síti.

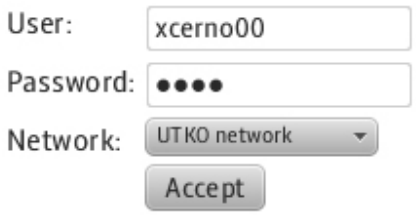
Následující popis aplikace nebude zaměřen čistě na ovládací prvky či práci s prostředím, ale přehledně shrne postup při vývoji jednotlivých částí, jejich funkčnost, použité principy a také problémy, které musely být během jejího vývoje vyřešeny. Prvním krokem, ke kterému aplikace po spuštění uživatele vyzve je přihlášení.



Obr. 5.1: Grafické prostředí aplikace.

5.1 Přihlášení

Přestože aplikace umožňuje pouze základní získávání dat a není pomocí ní možné senzorovou síť ovládat či modifikovat, je vhodné data v ní zobrazovaná alespoň částečně chránit. K tomuto účelu slouží úvodní okno aplikace, ve kterém je uživatel vyzván k zadání přihlašovacího jména a hesla (obr. 5.2). Po jejich vyplnění a následném potvrzení tlačítkem „Accept“, dojde k jejich porovnání se seznamem oprávněných uživatelů umístěným v databázi. Z důvodů větší bezpečnosti nejsou přihlašovací údaje porovnávány na straně aplikace, ale na straně databázového úložiště, které v případě nalezené shody vrátí identifikační číslo uživatele. Není-li uživatel v databázi nalezen, či neshoduje-li se zadané heslo, vrátí databáze nulu a přístup do aplikace bude odepřen.



Obr. 5.2: Úvodní okno aplikace.

5.2 Načítání informací o síti a seznamu uzlů

Kromě zadání přihlašovacích údajů slouží úvodní přihlašovací okno také k výběru sítě. Dostupné senzorové sítě se z databáze načtou ihned po spuštění aplikace a výběrem jedné z nich, stisknutím tlačítka „Accept“, dojde ke spuštění série příkazů sloužících k načtení potřebných údajů z databáze. Pokud uživatel při přihlášení nevybere žádnou z nabízených sítí, aplikace ho sice pustí do hlavního okna, ale to nebude obsahovat žádná data a informace k zobrazení.

Prvním blokem dat načítaným po přihlášení jsou statické informace o síti. Výraz „statické“ je v tomto případě myšlen tak, že se během doby spuštění aplikace nepočítá se změnou těchto údajů (na rozdíl od údajů jednotlivých uzlů). Načítané informace jsou rozděleny dle následujících kritérií.

- *Informace o síti* - název, protokol, PANID, EXTPANID a kanál na kterém vysílá.

- *Informace o poloze sítě* - ulice, město, stát a údaje o zeměpisné délce a šířce.

Druhým a nejdůležitějším blokem, k jehož načtení dojde, je seznam uzlů patřících do dané sítě (obr. 5.3). Tyto uzly jsou z databáze načítány spolu se svými základními informacemi jako id, název, logická adresa a souřadnice x, y, z. Hodnota id udává pořadí uzlu v databázové tabulce a je tudíž jeho jedinečným identifikátorem. Tohoto faktu je využito i v aplikaci, kde se pro práci a identifikaci uzlů využívá právě jejich id.

Obr. 5.3: Výpis seznamu uzlů.

Zvláštními údaji, které jsou jako jediné nejen načítány, ale i vkládány pomocí aplikace jsou souřadnice x, y, které jsou vztaženy ke zde použitému mapovému podkladu. Jelikož k prvotnímu vkládání uzlů do sítě dochází na straně databáze, nemůže osoba toto provádějící znát rozměry podkladu a tudíž ani polohu uzlu v něm. Tato situace je ošetřena tak, že pokud se vkládá do databáze nový uzel, jsou mu nastaveny souřadnice x, y na nulu. Tento stav aplikace při načítání uzlu detekuje a nezobrazí ho v mapovém podkladu, ale pouze v seznamu uzlů sítě, kde je tento stav avizován nastavením parametru *visible* na *false*. Chce-li uživatel daný uzel umístit do mapového podkladu, tak jej nejdříve pomocí dvojkliku na jeho název zobrazí a po přesunutí na správné místo tlačítkem „Save position“ uloží jeho polohu. Od této chvíle se při každém spuštění aplikace bude uzel nacházet na určené pozici.

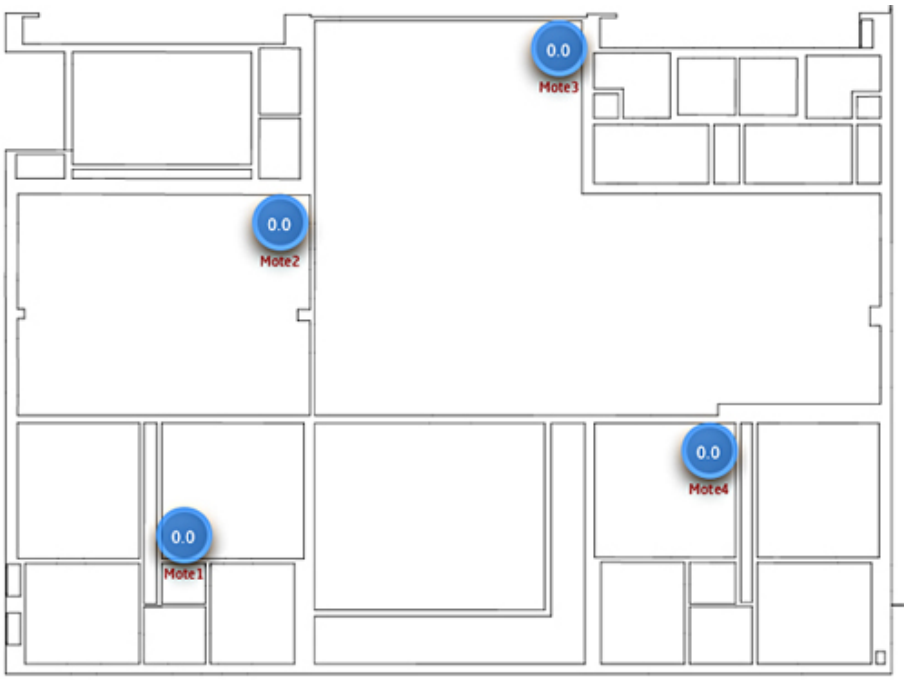
5.3 Poloha uzlů v mapovém podkladu

Jelikož hlavní a nejnáročnější částí programu je zobrazování polohy uzlů v mapovém podkladu, byla většina problémů vzniklých při tvorbě aplikace spojena právě s touto

částí. Prvním z nich byl již výběr formátu dispozičního podkladu. Jelikož rozlohou senzorové sítě často pokrývají celé budovy, byly požadavky na podklad následující.

- Vysoký rozsah přiblížení bez ztráty kvality.
- Malá velikost souboru.
- Podpora v jazyce JavaFX.

Po ujasnění těchto požadavků, které vylučují všechny zástupce rastrové grafiky, vyšel jako nejvhodnější řešení vektorový grafický formát SVG. Jeho hlavní výhodou proti formátům EPS (PostScript), PDF (Portable Document Format) a CDR (Corel Draw), je jeho otevřenost a velké množství volně dostupných nástrojů pro jeho editaci. Aby bylo možné využít všechny výhody jazyka JavaFX a usnadnila se práce s podkladem, převedou se soubory SVG pomocí programu SVG converter (součást JavaFX production suite) do formátu FXZ. Tento postup umožní načíst soubor FXZ do aplikace a zde s ním pracovat jako s objektem. Příklad zobrazení senzorových uzlů v mapovém podkladu se nachází na obrázku 5.4.



Obr. 5.4: Zobrazení senzorů v mapovém podkladu.

Operace, které budou s půdorysem budovy či jiným podkladem prováděny, jsou posunutí a přiblížení (oddálení). K zvětšení či zmenšení jakýchkoliv objektů v JavaFX slouží příkazy *scaleX* a *scaleY*, které udávají, kolikrát bude zvětšen objekt

oproti své původní velikosti. Nastavením hodnoty *scale* menší než jedna, dojde ke zmenšení objektu. Pro aplikaci je dolní hodnota *scale* omezena na 0,5, jelikož při zmenšení pod tuto hranici docházelo při posunu podkladu k nepředvídatelným výchylkám. Maximální zvětšení je explicitně nastaveno na hodnotu 10, ale z důvodů různé podrobnosti a velikosti podkladového materiálu ji lze upravit pomocí komponenty „Maximal zoom“ až na hodnotu 30. K další požadované operaci, tedy posunutí v obou osách, slouží příkazy *translateX* a *translateY*. Pro určení velikosti a směru posunutí se vychází z výpočtu proměnných, zobrazeném ve výpisu 5.1. Ten je nutné aplikovat jak na posunutí v ose x, tak na posunutí v ose y.

Výpis 5.1: Příklad výpočtu posunutí

```
translate = translate + (delta/scale)
delta = end - start
//end - souradnice konecneho bodu posunuti (pixely)
//start - souradnice pocatecniho bodu posunuti (pixely)
//scale - meritko podkladu
```

Jak je z výpisu 5.1 patrné, je nutné do výpočtu promítnout také hodnotu *scale*, která zajistí správný posun při proměnné hodnotě zvětšení.

Kvůli problémům vznikajícím při přímém navázání zvětšení uzlu na zvětšení podkladu jsou tyto dvě komponenty separovány. Díky tomu je možné pomocí komponenty „Node size“ měnit velikost (zvětšení) uzlů již výše popsaným způsobem a zcela nezávisle na velikosti podkladu. Komplikace však nastávají při výpočtu posunutí uzlů, které musí udržovat stále stejnou polohu v podkladu nezávisle na zvětšení obou prvků. Dojde-li tedy ke změně velikosti podkladu, musí být poloha uzlu přepočítána dle funkce z výpisu 5.2.

Výpis 5.2: Funkce pro posunutí uzlu v závislosti na přiblížení podkladu

```
xPosition = (loadedX+ areaTranX+zoomPosX);
yPosition = (loadedY+ areaTranY+zoomPosY);

if(prewZoom < (zoom)){
    zoomPosX += (xPosition -400)/( zoom-0.5)*0.5;
    zoomPosY += (yPosition -300)/( zoom-0.5)*0.5;
}else if(prewZoom > (zoom)){
    zoomPosX -= (xPosition -400)/( zoom+0.5)*0.5;
    zoomPosY -= (yPosition -300)/( zoom+0.5)*0.5;
}
prewZoom = (zoom);
```

Nejdříve tedy dojde k výpočtu proměnných *xPosition* a *yPosition*, které udávají aktuální polohu uzlu. Ta se pro osu x skládá ze souřadnice *loadedX* (načtené z databáze), *areaTranX* (posunutí mapového podkladu) a *zoomPosX* (posunutí způsobené zvětšením podkladu). Následně v závislosti na tom, byl-li podklad zvětšen nebo zmenšen, se provede výpočet posunutí uzlu tak, aby kopíroval změnu rozměrů úměrnou provedené operaci. Hodnoty *400* a *300* nejsou ve funkci uvedeny náhodně, ale značí souřadnice x, y tzv. pivotu, což je bod podle kterého se změna velikosti podkladu provádí. Proměnná *zoom* udává aktuální hodnotu zvětšení, která je upravována číslem 0,5 značícím krok, po kterém je změna velikosti prováděna.

Jak je z výše uvedených a popsaných funkcí patrné, dochází při operacích změny polohy a hlavně velikosti k několika složitějším výpočtům, které musí mít vliv na přesnost výsledků. Tato nepřesnost se projevuje při opakované změně velikosti mapového podkladu a následném uložení pozice uzlu, kdy se po restartu aplikace uzel nepatrně vychýlí ze své původní pozice. Jelikož se však toto vychýlení pohybuje v setinách pixelu a po následné korekci se již neprojevuje, můžeme ho považovat pro naše účely za zanedbatelné.

5.4 Výběr uzlů a nastavení měřené veličiny

Jelikož se v aplikaci počítá se zobrazováním mnoha uzlů najednou a ne vždy chce uživatel zobrazovat na všech stejné informace, vyvstala během programování potřeba individualizovat jednotlivé uzly a dát uživateli nástroj, díky kterému si bude moci sám volit, se kterými chce pracovat. To je umožněno výběrem uzlu pomocí jednoduchého kliknutí na něj tlačítkem myši. K výběru všech uzlů na ploše lze využít tlačítko „(un)select all“. Takto označené uzly jsou od ostatních odlišeny změnou barvy okraje (obr. 5.5).

Obr. 5.5: Zobrazení nevybraného (vlevo) a vybraného (vpravo) uzlu.

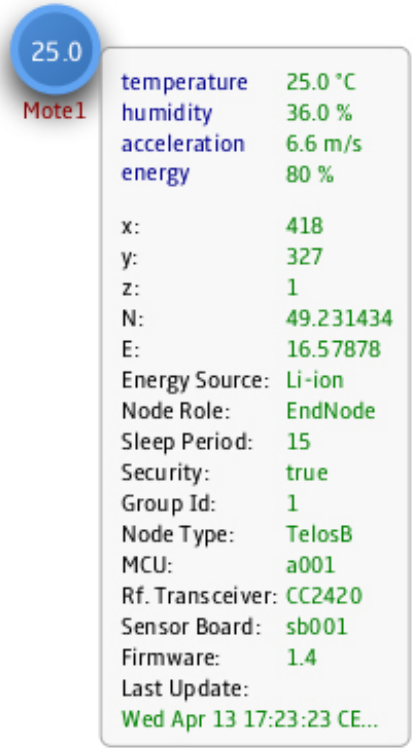
Jak je z obrázku patrné, kromě indikace výběru umožňují uzly také zobrazení jejich názvu a nejaktuálnější hodnoty vybrané veličiny. K výběru zobrazované veličiny slouží komponenta „Phenomenon“, která nám po jejím zvolení nabídne seznam

veličin, které jsou schopny senzory uzlu měřit. Tento seznam se získává pro každý uzel zvlášť, při jejich načítání z databáze. Jelikož se velice často stává, že senzory na více uzlech nejsou totožné a tím pádem i veličiny, které jsou schopné měřit se různí, je nutné při vícenásobném výběru uzlů pomocí algoritmu zjistit jejich společné veličiny a až ty následně nabídnout uživateli k výběru. Pokud při tomto procesu není nalezena žádná shoda, zůstane komponenta „Phenomenon“ prázdná.

5.5 Informační tabulka uzlu

Kromě výše popsaných funkcí umožňují jednotlivé uzly také zobrazení tabulky, která skýtá podrobnější informace o každém z nich. Příklad obsahu této tabulky je možné si prohlédnout na obrázku 5.6, přičemž její vyvolání se provádí pomocí dvojkliku na plochu uzlu. Hodnoty v ní uvedené jsou rozděleny do dvou bloků. První z nich, modře podbarvené, značí hodnoty měřené pomocí senzorů uzlu a jsou tedy dynamicky měněny za běhu programu. Změna těchto parametrů, plus změna hodnoty energie, která je do této skupiny také zahrnuta, se provádí v závislosti na uživatelském nastavení obnovovacích intervalů, které budou podrobněji popsány níže. Jak už bylo v textu jednou zmíněno, měřené veličiny (teplota, vlhkost atd.) se mohou uzel od uzlu lišit a proto ani zde nejsou v tabulce zadány napevno, ale jsou načítány z databáze pro každý uzel zvlášť. Aby nedocházelo k mylné interpretaci naměřených hodnot, jsou stejným způsobem, tedy z databáze, získávány také jednotky měřených veličin. Díky tomu je zaručena naprostá nezávislost aplikace na typu zobrazovaných dat.

Černým podbarvením jsou charakterizovány informace spadající do druhého bloku údajů, které není možné při zobrazené tabulce dynamicky měnit. Jsou zde zahrnuty informace o použitém hardwaru (Energy Source, Node Type, MCU, Rf. Transceiver, Sensor Board), vlastnostech uzlu (Node Role, Sleep Period, Security, Group Id) a jeho poloze (x, y, z, N, E). Jediným údajem z tohoto výčtu, jehož význam nemusí být na první pohled zcela patrný, je souřadnice z, která udává vrstvu mapového podkladu, v níž se daný uzel nachází. Speciálním údajem zahrnutým v tomto bloku, ale přímo vázaným na data bloku prvního, je položka „Last Update“. Důležitým parametrem, který nesmí být při tvorbě aplikace opomenut, je platnost zobrazovaných dat. Jak je z obrázku 5.6 patrné, obsahuje jeden uzel několik (zde 4) dynamicky načítaných hodnot, přičemž každá z nich mohla být změřena v jiný časový okamžik. Jelikož výpis všech těchto údajů by narušoval koncepci tabulky, je informace o platnosti každé z nich skryta do položky „Last Update“. Přepínání mezi jednotlivými časovými údaji probíhá automaticky po najetí kurzoru myši na změřený údaj. Chce-li tedy uživatel zjistit, kdy byla změřena teplota 25 °C uvedená na obrázku,



Obr. 5.6: Informační tabulka uzlu.

stačí najet kurzorem myši na číslo 25 a v položce „Last Update“ se automaticky tento údaj zobrazí. Pro úplnost je nutné podotknout, že časové známky a data jsou z databáze stahovány současně.

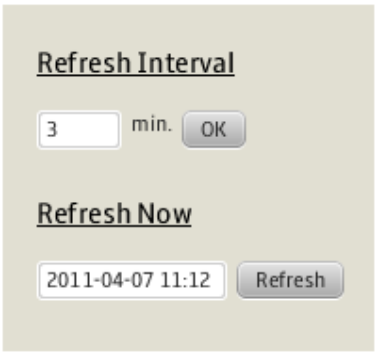
Pokud už nebude tabulka pro práci potřeba, může být opětovným dvojklikem na plochu uzlu uzavřena. Přes výše zmíněnou nemožnost dynamické změny údajů druhého bloku dat při otevřené tabulce, jsou tato data načítána vždy při jejím otevření. Z toho vyplývá, že chce-li uživatel tato data aktualizovat, stačí tabulku zavřít a znovu otevřít.

5.6 Interval obnovy dat

Jednou ze stěžejních funkcí, které byly do aplikace implementovány, je funkce obnovy zobrazovaných dat, díky které není aplikace pouhým zobrazovačem jednorázově stažených dat, ale umožňuje neustálou interakci s databázovým serverem. To uživateli zajišťuje neustálý přísun aktuálních informací, které jsou průběžně na server ukládány.

Po spuštění aplikace jsou všechna políčka určená pro zobrazení naměřených hodnot nulová, což je logické, protože aplikace neví, jaká data bude chtít uživatel načíst.

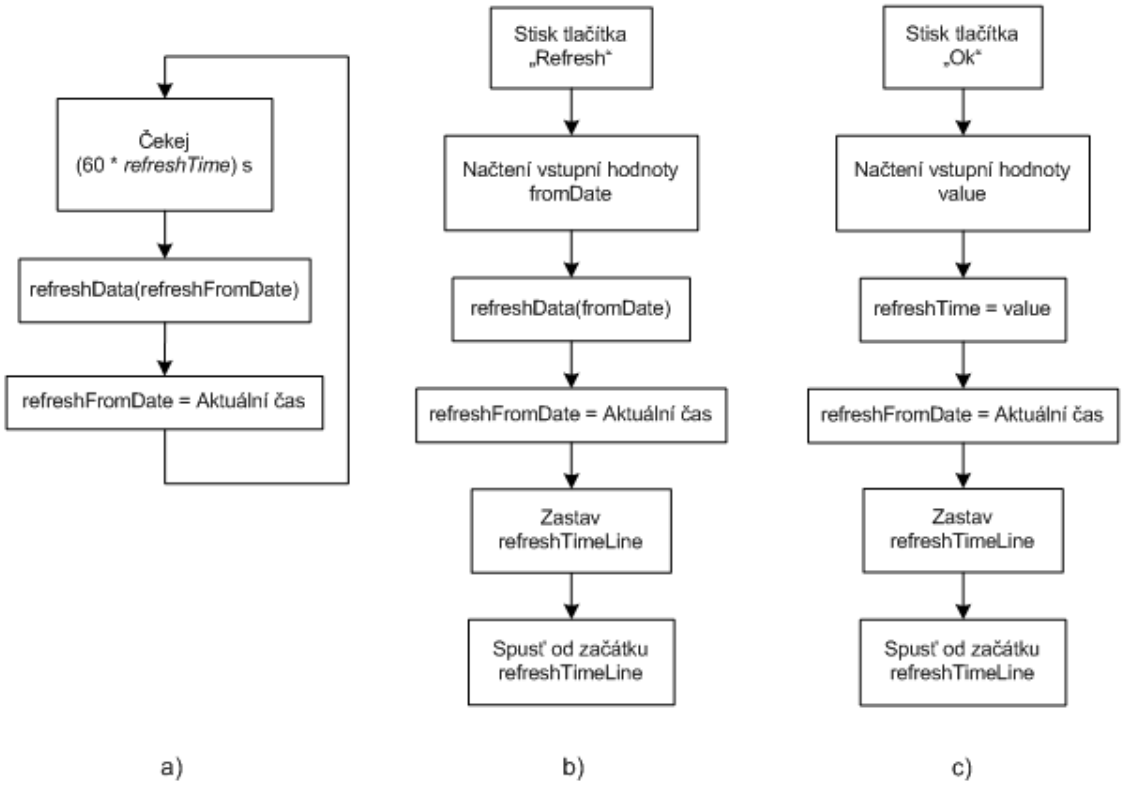
Jak z předchozí věty vyplývá, nejsou během aktualizace a prvotního načítání stahována všechna data, ale pouze ta, která jsou tzv. viditelná. Tato „viditelnost“ je určována dle funkce, která po jejím zavolání prochází uzel po uzlu a zjišťuje, zda mají nastavenou veličinu zobrazenou na jejich těle (implicitně nenastavena) a otevřenou tabulku s informacemi o uzlu. Po doběhnutí funkce se pro všechny viditelné údaje záznamy aktualizují. Díky tomuto způsobu aktualizace nejsou stahována nepotřebná data a nedochází tak k zbytečné zátěži databázového úložiště.



Obr. 5.7: Komponenty pro obnovu dat.

Impulz k provedení obnovy dat může být vydán dvěma způsoby. Pomocí komponent umístěných pod označením „Refresh Now“, nebo pomocí komponent skrývajících se pod označením „Refresh Interval“. Vývojový diagram prvně jmenovaného je možné si prohlédnout na obrázku 5.8b. Jak je z obrázku 5.7 patrné, skrývají se pod označením „Refresh Now“ dvě komponenty a to sice textové pole a tlačítko. Tento způsob aktualizace slouží především k prvotnímu stažení dat po spuštění aplikace, kdy uživatel do textového pole zadá časový údaj (v diagramu označen fromDate) o tom jaké nejstarší záznamy je ochoten tolerovat a vše spustí kliknutím na tlačítko „Refresh“. Tím se rozběhne výše popsaný proces výběru údajů, které budou aktualizovány s tím rozdílem, že z databáze budou vybrány pouze záznamy novější než uživatelem zadané mezní datum. Ty následně databáze seřadí podle data uložení a uživateli nabídne pouze ty nejaktuálnější. Nebudou-li žádné novější záznamy nalezeny, zůstanou zobrazené údaje na své původní hodnotě. Tímto způsobem lze tedy obnovovat data okamžitě, což je velice výhodné zejména při jejich nárazových kontrolách.

Bude-li však chtít uživatel využít aplikaci pro dlouhodobější dohled nad sítí, je neustálá nutnost klikání na tlačítko „Refresh“ nepříjemná. Z tohoto důvodu jsou implementovány komponenty skrývající se pod označením „Refresh Interval“, které zaručují pravidelnou a plně automatizovanou obnovu záznamů. Jako v předchozím případě, i zde se nachází textové pole a tlačítko „Ok“, které slouží k potvrzení



Obr. 5.8: a) Smyčka „refreshTimeLine“, b) Diagram „Refresh Now“, c) Diagram „Refresh Interval“

zadaného časového intervalu, na obrázku 5.8c označeného jako *value*. Definice časové smyčky v jazyce JavaFX je uvedena na výpisu 5.3 a její názornější blokové schéma na obrázku 5.8a.

Výpis 5.3: Definice časové smyčky

```
var refreshTimeline: Timeline = Timeline{
    repeatCount: Timeline.INDEFINITE;
    keyFrames: [
        KeyFrame {
            time: bind (60s*refreshTime)
            action: function (){
                refreshData(refreshFromDate);
                getTimeData();
            }
        }
    ]
}
```

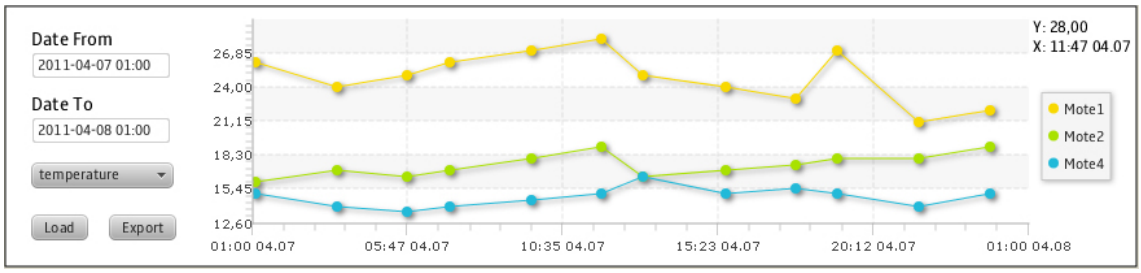
Hlavní výhodou tohoto řešení je, že se jedná o časovou linii používanou u animací, díky čemuž ji lze za běhu programu libovolně spouštět, zastavovat a přerušovat. Při podrobnějším prozkoumání výpisu 5.3, je možné si všimnout dvou důležitých parametrů *repeatCount* (počet opakování) a *time* (konečný čas smyčky). Počet opakování je v aplikaci nastaven na nekonečno a doba trvání časové smyčky je násobkem 60 sekund a proměnné *refreshTime*, jejíž nastavování se provádí v bloku označeném „Refresh Interval“. Díky tomuto uspořádání lze nastavovat libovolný interval obnovy dat. Spuštění smyčky se provede v momentě první aktualizace dat pomocí „Refresh Now“ komponent a poté už běží po celou dobu chodu aplikace. Výjimkami, kdy dojde k restartování smyčky jejím zastavením a opětovným spuštěním je situace, kdy uživatel provede okamžitou obnovu dat tlačítkem „Refresh“ nebo změní-li interval obnovy dat. Na obrázcích 5.8b a 5.8c, je tato situace ošetřena posledními třemi bloky, které mohou smyčku *refreshTimeLine*, zobrazenou na obrázku 5.8a, přímo ovládat.

Na rozdíl od uživatele, kterému stačí vědět, že obnova dat proběhne například každých pět minut, potřebuje databázový server jasně definovat od jakého data a času má záznamy porovnávat. K tomuto účelu slouží dosud nezmíněný parametr *refreshFromDate*, vyskytující se ve všech třech diagramech obrázku 5.8. Princip je takový, že po každém vypršení časovače nebo před restartem smyčky, se do proměnné *refreshFromDate* uloží aktuální čas, který se v následujícím cyklu použije jako mezní údaj. Od něj bude poté databáze záznamy porovnávat. Aby se předešlo problémům s nastavením systémových hodin různých počítačů a přechodem mezi zimním a letním časem, je údaj o aktuálním datu a čase získáván přímo z databázového serveru.

5.7 Graf

Doposud bylo řešeno, jak získávat z databázového serveru co nejaktuálnější data a poněkud stranou stál hlavní důvod tohoto uspořádání. Tím je zobrazení dlouhodobých dat. Jelikož pro správné vyhodnocení funkčnosti senzorové sítě je potřeba porovnávat měřené údaje za delší období, je do aplikace implementována funkce grafu. Okno grafu je možné si prohlédnout na obrázku 5.9.

Nejdůležitější a také nejproblémovější, částí této sekce jsou osy grafu, jejichž popisky a hodnoty musí být generovány automaticky a nezávisle na zobrazované veličině. První z nich je osa x, která zobrazuje časový údaj a jejíž počátek a konec definují parametry „Date From“ a „Date To“. Aby bylo možné rozdělit takto definovaný interval na kratší úseky, využívá aplikace funkci *getTime()* obsaženou v balíku



Obr. 5.9: Okno grafu.

java.util.date. Ta slouží k převodu data z formátu RR-MM-DD HH:mm na počet milisekund od 1. ledna, 1970, 00:00:00 GMT. Jak je uvedeno na výpisu 5.4, takto převedené hodnoty se od sebe odečtou, čímž se získá počet milisekund od počátku do konce uživatelem definovaného časového období. To už se dle potřeby rozdělí na požadovaný počet dílků. Kód ve výpisu 5.4 zabalený do funkce *for*, slouží ke zpětnému převodu na pro člověka lépe čitelný tvar. Zde je nutné dát pozor, aby nedošlo k převodu pouze vypočteného časového kroku, ale celkového počtu milisekund daného úseku. Aby nedošlo v tomto bodě k nejasnostem, je důležité zmínit, že zpětný převod je zde pro ukázku a v aplikaci se používá pouze pro generování popisků osy x. Pokud tedy mají být data zobrazena správně, musí být jejich x souřadnice uvedena v milisekundách. Z tohoto důvodu se při vstupním zpracování záznamů stažených z databázového serveru a určených pro zobrazení v grafu, přepočítává jejich časový údaj již výše popsaným způsobem.

Výpis 5.4: Převod formátu data na milisekundy a zpět

```
function setXaxis(fromDate:Date, toDate:Date){
    var diff:Float;
    lowBoundX = fromDate.getTime();
    upBoundX = toDate.getTime();
    diff = upBoundX - lowBoundX;
    tickUnitX = diff/5;

    for(f in [0..5]){
        var resultdate:Date = new Date(fromDate.getTime()+
        +(diff*f));
        println(format.format(resultdate));
    };
}
```

Po podrobnějším pohledu na obrázek 5.9, je možné si všimnout, že se zde nachází stejná komponenta pro výběr měřené veličiny, jako byla již popsána v kapitole 5.4. I když jsou tyto dva ovládací prvky zcela totožné, nejsou na sobě nijak závislé či spolu provázané. Aby bylo možné zobrazit data v grafu, je nejen nutné definovat časové období, do kterého mají záznamy spadat, ale také typ požadovaných dat. Na rozdíl od časového údaje, který má pro všechny měřené veličiny stejný tvar a rozsah, se mohou naměřené hodnoty jednotlivý senzorů pohybovat od desetin až po tisíce. Z tohoto důvodu nemůže být rozsah osy y pevně definován, ale musí se dynamicky přizpůsobit aktuálním údajům. Toho je docíleno tak, že při vstupu do aplikace jsou všechny záznamy z databáze procházeny a porovnávány mezi sebou, dokud se nenajde nejmenší a největší hodnota. Ty poté vytvoří hranice pro rozsah osy y. Jelikož senzory umístěné na uzlech mohou během měření zaznamenat chybu, která svou hodnotou převyšuje ostatní změřené veličiny, je pro osu y implementována funkce pro manuální nastavení maximální zobrazované hodnoty. To se provádí pomocí okna zobrazeného na obrázku 5.10, které je vyvoláno kliknutím tlačítka myši na osu y a je chráněno proti zadání jiného než číselného údaje.

Obr. 5.10: Okno pro nastavení maximální hodnoty osy y.

Jelikož okno pro zobrazení grafu není příliš velké, což činí vyčítání hodnot pomocí os velice obtížným, byla do něj pro zlepšení uživatelského komfortu implementována funkce, díky které je každý vložený bod zvýrazněn. Po kliknutí na tento bod se v pravém horním rohu okna zobrazí jeho hodnoty x a y. Pokud uživateli nebude ani tato funkce dostačovat může si zobrazená data pomocí tlačítka „Export“ uložit do formátu CSV a následně pro podrobnější analýzu či zobrazení exportovat do jiného programu.

5.8 Načtení aktuálních dat ze senzorů

Při provádění všech doposud popsaných operací a úkonů, se vždy využívala data získaná z databázového serveru. Nyní tedy bude objasněno, jak je možné navázat spojení s konkrétními uzly a získat od nich aktuální měřené hodnoty. Hlavním prvkem, který slouží pro výběr zdroje dat, je komponenta pojmenovaná „Data Source“.

Ta obsahuje dvě možnosti výběru a to sice „Database“, pro komunikaci s databázovým serverem a „Realtime Data“, pro přímou komunikaci se senzorovými uzly. Po přepnutí do režimu „Realtime Data“, se na první pohled nic nezmění, ale jakmile klikneme na tlačítko „Refresh“, dojde k načtení aktuálních hodnot pro všechny uzly. Zde je nutné podotknout, že z důvodu úspory energie uzlů se nenačítají všechny veličiny, které je uzel schopen měřit, ale pouze ta, která je zobrazená přímo na uzlu. Samotná komunikace mezi aplikací a senzorovým uzlem je věcí komunikačního rozhraní a bude podrobněji probrána v kapitole 6.1.

5.9 Odhlášení

Funkce pro odhlášení je implementována zejména proto, aby umožnila uživateli bezpečné opuštění hlavního okna s možností přihlášení se pod jiným jménem, či do jiné sítě bez nutnosti uzavření celé aplikace. Jelikož po odhlášení a opětovném přihlášení nedocházelo k přepsání některých proměnných a načtených grafů, muselo toto být ošetřeno příslušným kódem jako reakce na stisk tlačítka „Log Out“.

6 KOMUNIKAČNÍ ROZHRANÍ

Bezesporu nejdůležitější částí aplikace sloužící pro přenos a zobrazení dat ze senzorových sítí je její komunikační rozhraní. Aplikace nejen že je schopna získávat data „online“ přímo od uzlů sítě, ale také umožňuje stahovat data z databázového úložiště, kam jsou průběžně ukládána. Toto průběžné ukládání je zajišťováno programem běžícím na serveru, který má za úkol v daných časových intervalech shromažďovat data ze senzorů a ukládat je do předem vytvořených tabulek v databázovém úložišti. Díky tomuto programu, se aplikace nemusí zabývat tříděním a ukládáním dat, ale pouze si je stáhne z předem daných pozic v databázi.

Aby byly informace poskytované webovou aplikací co nejúplnější, nebudou v uživatelském rozhraní data získaná přímo od uzlu nijak oddělena od dat získaných z databázového úložiště. Naopak, budou se vzájemně prolínat a doplňovat tak, aby je uživatel mohl vzájemně porovnávat a získal tak kompletní přehled nad stavem připojené senzorové sítě.

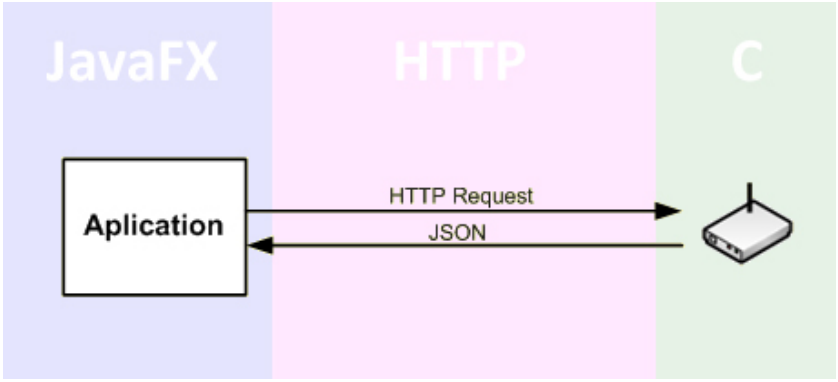
Obě rozhraní spolu s použitými principy budou popsána zvlášť, neboť komunikace s nimi i řetězec předávání dat je pro oba přístupy velice odlišný.

6.1 Webserver

Aby bylo možné získávat data přímo z uzlů bezdrátové senzorové sítě, musí v sobě mít každý z nich nahrán webový server, díky němuž bude možné přidělit uzlu pevnou IP adresu. V tomto případě je na uzlech instalován RESTful web server běžící pod operačním systémem Contiki. Ten v sobě obsahuje odlehčený HTTP server, který umožňuje přijímat a odesílat HTTP zprávy a v reakci na ně získávat od operačního systému požadovaná data. Tyto data následně zpracuje a předá protější straně v požadovaném formátu. Server je nastaven tak, aby pracoval v rámci pravidel REST architektury a návratové hodnoty ukládal do formátu JSON. Celkové schéma komunikace je na obrázku 6.1.

Pokud aplikace bude chtít komunikovat s konkrétním uzlem, zjistí si jeho IP adresu a na ní bude zasílat HTTP dotazy. Na rozdíl od komunikace s databázovým serverem je získávání aktuálních dat přímo od uzlů závislé na předchozím načtení IP adresy a seznamu dostupných senzorů z databáze. Nepodaří-li se tyto údaje načíst, nebude aplikace vědět na jakou URL adresu má dotaz poslat.

Jak z výše uvedeného vyplývá, jsou stěžejními funkcemi této kapitoly vyslání HTTP dotazu a zpracování vráceného JSON dokumentu.



Obr. 6.1: Přenosový řetězec pro komunikaci s uzly.

6.1.1 HTTP dotazy

Základním kamenem správné funkčnosti komunikace mezi aplikací a webovým serverem jednotlivých uzlů, je bezchybný přenos HTTP zpráv. V této kapitole bude proto více přiblížena struktura a možnosti třídy *HttpRequest*, která tuto funkci v jazyce JavaFX zajišťuje.

HttpRequest je základní třídou balíčku *javafx.io.http* [12], který umožňuje posílat HTTP dotazy na server a přijímat od něj data ve formě odpovědi.

Jednoduchou funkci, kterou využívá aplikace pro získání dat ze serveru, je možné si prohlédnout na výpisu 6.1. Zde dojde k vytvoření instance *request* třídy *HttpRequest*, která je na konci výpisu příkazem *request.start()* zavolána. Po zavolání této instance dojde k poslání GET dotazu na URL adresu uvedenou v parametru *location*. Adresa uvedená v příkladu obsahuje dvě proměnné *address* a *phenom*. První z nich udává logickou adresu cílového uzlu a druhá měřenou veličinu, jejíž aktuální hodnota bude zjišťována. Bude-li žádost kladně vyřízena, obdrží aplikace zpět požadovaná data, která si uloží do proměnné *os*. Následně je prostřednictvím funkce *processJSON()* předá k dalšímu zpracování (bude popsáno v následující kapitole).

Výpis 6.1: Příklad použití HttpRequest

```
function getRealtimeData(phenom:String, address:String, k:Integer){
    var os = new ByteArrayOutputStream();
    var request = HttpRequest{
        location: "http://{address}/appdata/phenomenon/{phenom}/"
        sink: os
        onDone: function():Void{
            processJSON(os, k);
        }
```

```
    }
    request.start();
}
```

Aby byla práce s daty kompletní, obsahuje třída *HttpRequest* možnost je na server odesílat (příkazy PUT a POST) nebo mazat (příkaz DELETE). Vlastnosti metody GET použité v aplikaci jsou tedy následující.

Metoda GET

Během svého „životního“ cyklu prochází žádost HTTP GET následujícími fázemi popsanými v [13].

1. Proměnné *location* a *method* objektu *HttpRequest* jsou nastaveny.
2. Spuštění operace zavoláním funkce *start()*.
3. Zahájení operace ve vlákně běžícím na pozadí.
4. Pokus o spojení se serverem.
5. Pokud je spojení úspěšné, pokračuje se ve vykonávání kódu, pokud ne je ohlášena chyba.
6. Žádost je spolu s HTTP hlavičkou, uloženou v proměnné *headers*, odeslána serveru.
7. Přečtení odpovědi serveru (začíná se od HTTP hlavičky).
8. Získání kódu a zprávy z HTTP hlavičky.
9. Pokud server zprávu odmítl, je ohlášena chyba.
10. Dokončení zpracování hlavičky.
11. Přečtení těla odpovědi a zpřístupnění obsahu aplikaci.
12. Ohlášení dokončení zpracování odpovědi.

Z předešlého výpisu je jasně patrné, že žádost *HttpRequest* prochází mnoha stavy, což společně s ne vždy plně spolehlivou sítí, může vést ke vzniku nežádoucích chyb. Aby bylo v případě jakéhokoliv selhání chybu možné lépe nalézt a opravit, jsou během cyklu zpracování žádosti aktualizovány stavy několika proměnných a jim odpovídajících funkcí. Ty umožňují sledovat, v jakém stavu se proces nachází. Seznam

Tab. 6.1: Návratové hodnoty při volání *HttpRequest* [12].

Proměnná	**Funkce**	**Počáteční hodnota**	**Popis**
started	onStarted	false	Indikace začátku zpracování HTTP request.
connecting	onConnecting	false	Spojování se serverem.
doneConnect	onDoneConnect	false	Spojeno se serverem.
readingHeaders	onReadingHeaders	false	Čtení hlavičky odpovědi.
responseCode	onResponseCode	0	Čtení kódu HTTP odpovědi.
responseMessage	onResponseMessage	null	Čtení zprávy HTTP odpovědi.
error	onError	null	Zpráva o chybě od serveru.
doneHeaders	onDoneHeaders	false	Dokončení čtení hlavičky odpovědi.
reading	onReading	false	Začátek čtení těla odpovědi.
toRead	onToRead	0	Velikost dat dostupných pro čtení.
read	onRead	0	Velikost právě čtených dat.
input	onInput	null	InputStream poskytuje přístup do těla odpovědi.
doneRead	onDoneRead	false	Dokončení čtení těla odpovědi.
done	onDone	false	Dokončení HTTP request.
exception	onException	null	Zpráva o nastalé výjimce.

těchto funkcí se stručným popisem, proměnou a hodnotami které může nabývat, je obsahem tabulky 6.1.

Proměnná *exception* uvedená v tabulce jako poslední je nastavena vždy, pokud dojde k neočekávané výjimce v jakékoliv části procesu. Nastane-li tato výjimka, jsou všechny další operace zastaveny. Posledním krokem po dokončení celé žádosti *HttpRequest* je nastavení proměnné *done* na *true* a zavolání funkce *onDone*. K tomuto musí dojít i v případě neúspěšného volání a obdržení chybové hlášky.

Stejně jako všechny ostatní objekty v JavaFX i *HttpRequest* musí být vytvořena a používána výhradně v hlavním vlákně aplikace. Jedině proces připojování, čtení a zápisu dat na server může běžet ve vlákně na pozadí, ovšem i zde platí, že pokud chceme nastavovat proměnné nebo zpracovávat funkce o aktuálním stavu žádosti

HttpRequest, musíme tak činit v hlavním vlákně aplikace. Navzdory svému jménu může být třída *HttpRequest* použita pro čtení dat z jakéhokoliv zdroje disponujícího URL adresou a protokolem podporovaným Java platformou. Příkladem takových protokolů jsou *file:* pracující s lokálními soubory a *ftp:* sloužící k přenosu dat mezi klientem a FTP serverem.

6.1.2 Zpracování JSON

Jelikož struktura a vlastnosti JSON dokumentu byly již popsány, zaměří se tato kapitola na zpracování přijatého dokumentu v aplikaci vytvořené pomocí jazyka JavaFX. K procházení dokumentu a ukládání dat do proměnných slouží objekt *parser*, který je instancí třídy *PullParser* [12]. Jak třída *PullParser* prochází dokumentem, vrací zprávy o aktuálním stavu. Ošetření těchto zpráv, jejichž přehled se nachází v tabulce 6.2, se provádí ve funkci *onEvent* [5].

Tab. 6.2: Návratové hodnoty při volání *PullParser* [12].

Zpráva	**Popis**
START_DOCUMENT	Začátek JSON dokumentu.
END_DOCUMENT	Konec JSON dokumentu.
START_ELEMENT	Začátek JSON objektu.
END_ELEMENT	Konec JSON objektu.
START_VALUE	Začátek hodnoty objektu.
END_VALUE	Konec hodnoty objektu.
START_ARRAY	Začátek pole.
END_ARRAY	Konec pole.
START_ARRAY_ELEMENT	Začátek elementu uloženého v poli.
END_ARRAY_ELEMENT	Konec elementu uloženého v poli.
TEXT	Indikuje textový řetězec.
INTEGER	Indikuje číslo z rozsahu integer.
NUMBER	Indikuje číslo z rozsahu floating-point.
TRUE	Hodnota true z rozsahu boolean.
FALSE	Hodnota false z rozsahu boolean.
NULL	Hodnota null.

Nyní, když je známa použitá třída a stavové zprávy, je možné uvést jednoduchý příklad zpracování JSON dokumentu z výpisu 6.2. Ten byl vrácen serverem po přijetí *HttpRequest* z předchozí kapitoly. Jak je z obsahu výpisu 6.2 patrné, byl vrácen po

dotazu na teplotu změřenou senzorem a jediný údaj, který bude aplikaci zajímat je hodnota proměnné *val*.

Výpis 6.2: JSON dokument

```
{
 "val": "25.7",
 "seq" : "9856"
}
```

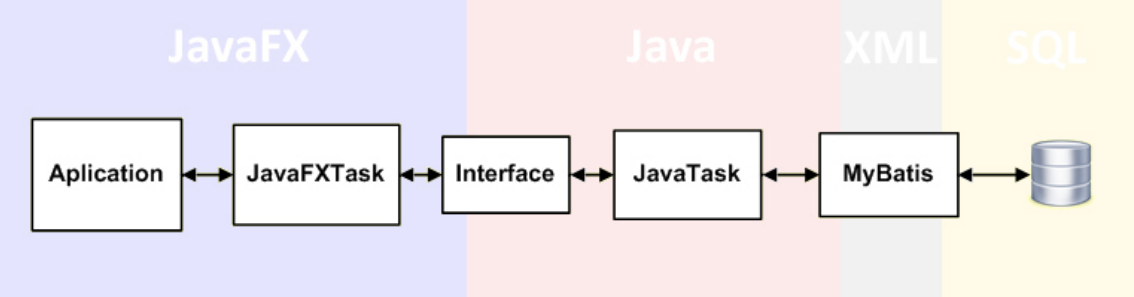
Jak je z výpisu 6.3 patrné, ještě než zpracování začne, definuje se proměnná *value* do které bude uložena získaná hodnota. Poté následuje inicializace instance *parser*, které musí být zadán vstupní dokument a specifikován jeho formát. Po dokončení těchto přípravných opatření může být přistoupeno k samotnému prohledávání dokumentu, které je založeno na porovnávání jmen přidělených k proměnné. Dojde-li během tohoto procesu ke shodě, načte se hodnota příslušející názvu „val“ a uloží se do proměnné *value*, definované na začátku funkce. Tímto způsobem se projde celý dokument a získá se tak požadovaná hodnota, která je příkazem na posledním řádku přidělena příslušnému uzlu.

Výpis 6.3: Příklad parsování JSON dokumentu

```
function processJSON(os:ByteArrayOutputStream, k:Integer){
    var vstupniDokument = new ByteArrayInputStream(
                                os.toByteArray());
    var value: String;
    var parser = PullParser {
            input: vstupniDokument
            documentType: PullParser.JSON
            onEvent: function(event: Event) {
                    if (event.type == PullParser.TEXT){
                            if(event.name == "val"){
                                    value = event.text;
                            }
                    }
            }
    }
    parser.parse();
    nodeObj[k].value = value;
}
```

6.2 Databázové úložiště

Databázové úložiště je hlavním zdrojem dat aplikace, bez jejichž správného načítání nebude umožněn její korektní chod. Hlavním rozdílem oproti komunikaci s webovým serverem je možnost uložit do databáze statická data a doplňkové informace, které nám senzor samotný není schopen poskytnout. Informace, které aplikace získává z předpřipravené databáze, lze vyčíst z jejího schématu uvedeného v příloze A. Při vývoji byla použita standardní MySQL databáze, ale po příslušných změnách v nastavení rámce MyBatis, může být použita jakákoliv databáze podporující dotazovací jazyk SQL. Schéma celého komunikačního řetězce je možné si prohlédnout na obrázku 6.2.



Obr. 6.2: Přenosový řetězec pro komunikaci s databázovým serverem.

Jak je ze schématu patrné již na první pohled, získávání dat z databázového serveru je mnohem komplexnější a složitější než přímé dotazování senzorových uzlů. Tato komplexnost je podpořena množstvím použitých jazyků, z nichž každý má své opodstatnění ať už pro zjednodušení práce (XML) nebo pro nezbytný chod základních funkcí aplikace (Java).

V následujícím textu bude uvedený řetězec rozdělen na dvě části a jejich princip podrobně popsán.

6.2.1 Asynchronní přístup k databázi

Cílem každé aplikace pracující s daty ze vzdáleného serveru je, aby stahování těchto dat probíhalo na pozadí a uživatel tak mohl současně aplikaci ovládat. Pokud by stahování probíhalo ve vláknu hlavním, mělo by to za následek zamrznutí aplikace po celou dobu komunikace se serverem.

V této kapitole bude podrobněji popsán způsob tvorby třídy, která se pomocí rámce MyBatis připojí k databázovému serveru, ve vlákně běžícím na pozadí přečte

požadované údaje a doručí je pomocí funkce zpětného volání do hlavního vlákna aplikace. Jednotlivé kroky procesu jsou následující.

1. Vytvoření vlákna běžícího na pozadí.
2. Inicializace instance pro navázání spojení s databází.
3. Zavolání funkcí běžících na pozadí, které pomocí rámce MyBatis zajistí stažení dat z databáze.
4. Ukončení spojení s databází.
5. Předání dat přes rozhraní do hlavního vlákna aplikace.
6. Úprava a následné zobrazení dat.

Při realizaci popsaného postupu v praxi vyvstává několik překážek vycházejících z faktu, že JavaFX je jednovláknový jazyk. První z nich je nemožnost přístupu k proměnné v jazyku JavaFX z jiného než hlavního vlákna, což přímo odporuje požadavku z bodu 3 a druhou překážkou je nemožnost vykonávat JavaFX kód v jiném než hlavním vlákně aplikace. Aby tedy nedocházelo při komunikaci k problémům, musí být kód běžící ve vlákně na pozadí napsaný v jazyce Java.

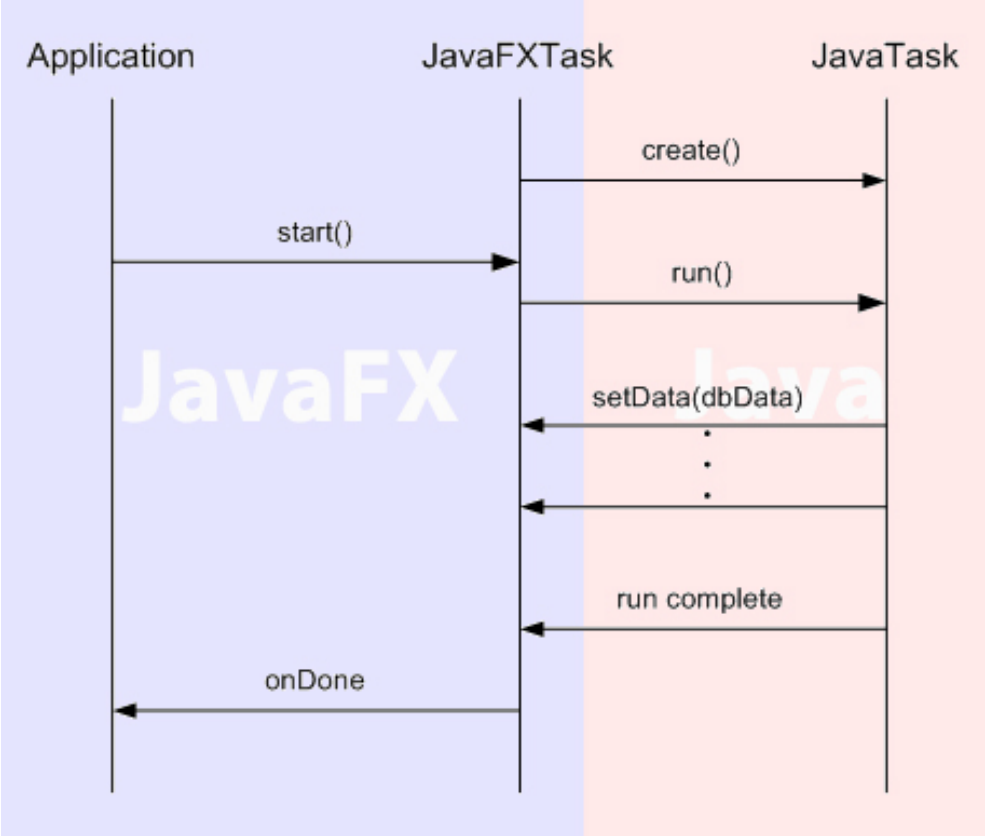
Pro komunikaci s databází jsou potřeba následující třídy [14].

- *JavaTask* (Java) - třída běžící ve vlákně na pozadí. Využívá metodu *run()*, obsahující kód, který je spuštěn na pozadí. Implementuje rozhraní *RunnableFuture* (viz níže).
- *JavaFXTask* (JavaFX) - rozšiřuje *JavaTaskBase* (viz níže). Přepisuje metodu *create()* a vytváří/vrací třídu *JavaTask*.
- *JavaObject* (Java) - slouží k ukládání řádků tabulky dat přijatých ze serveru. Objekt obsahující data je poté konvertován na *JavaFXObject*.
- *JavaFXObject* (JavaFX) - obsahuje stejné proměnné jako *JavaObject*.
- *Interface* (Java) - Javové rozhraní, které slouží jako prostředník při převodu objektu z Java do JavaFX. Zpřístupňuje funkci pro volání.

Pro správnou funkčnost aplikace je výše uvedené třídy nutné doplnit o třídy již implementované v jazyce JavaFX. Jak je uvedeno v [12], k podpoře asynchronního přenosu dat je určen balíček *javafx.async*, obsahující následující třídy.

- *JavaTaskBase* - umožňuje spustit kód Javové aplikace ve vlákně běžícím na pozadí. Tento kód je do *JavaTaskBase* předán pomocí implementace abstraktní funkce *create()*, která vrací objekt podporující Javové rozhraní *RunnableFuture*.
- *Task* - umožňuje spustit, zastavit a sledovat kód běžící na pozadí.
- *RunnableFuture* - rozhraní pro Javový kód, který bude běžet v budoucnu ve vlákně na pozadí. JavaFX kód by neměl být spouštěn z objektu, který podporuje toho rozhraní.

Přehlednější zobrazení spouštěných metod a hodnoty nastavovaných proměnných jsou zobrazeny na obrázku 6.3.



Obr. 6.3: Vytvoření vlákna běžícího na pozadí.

Implementaci třídy *JavaFXTask*, která zajišťuje pro aplikaci komunikaci s vláknem běžícím na pozadí, je možné si prohlédnout na výpisu 6.4.

Výpis 6.4: Třída *JavaFXTask*

```
public class JavaFXTask extends JavaTaskBase, Interface {
    public-init var networkNumber:Integer;
    public-read var nodesData:NodesFX[];
    public var onNodes:function(NodesFX[]):Void;

    protected override function create():RunnableFuture {
          new JavaTask(networkNumber, this);
    }

    public override function setNodesData(data:Object) {
        var javaObjects = data as nativearray of Nodes;
        for (javaObject in javaObjects) {
            insert NodesFX {

        //prevod objektu z Java do JavaFX
            } into nodesData;
        }
        if (onNodes != null) {
            onNodes(nodesData); //vrati data do hlavniho vlakna
        }
}}
```

Jak je možné si všimnout na výpisu 6.4, tato třída rozšiřuje třídu *JavaTaskBase* a rozhraní *Interface*. Implementovaná funkce *create()*, vytváří a vrací instanci *JavaTask*, které předává adresu databáze a parametry nutné pro přihlášení. Zbylá funkce *setNodesData()* je vyžadována rozhraním *Interface* a slouží k navrácení dat získaných z databáze. Tato funkce je volána s argumentem, který tvoří Java pole objektu *JavaObject*. Obsah tohoto pole je převeden na několik objektů *JavaFXObject*, jenž jsou poté uloženy do proměnné *nodesData*. Ta je následně zpřístupněna hlavní aplikaci voláním funkce *onNodes()*.

Hlavní třída vykonávající kód běžící na pozadí a sloužící k návratu získaných dat přes rozhraní *Interface* se nazývá *JavaFXTask*. Jak je z výpisu 6.5 patrné, hlavní kód běžící ve funkci *run()*, slouží ke komunikaci mezi aplikací a databázovým serverem pomocí rámce MyBatis. Jakmile je spojení navázáno dojde ke stažení dat a následnému odpojení. Stažená data, jsou poté vložena do seznamu a po převodu na pole zavoláním funkce *returnNodesData()* předána prostřednictvím rozhraní *Interface* třídě *JavaFXTask*. Převod dat na pole probíhá z důvodu jejich konverze mezi jazyky Java a JavaFX, která bez této operace není možná.

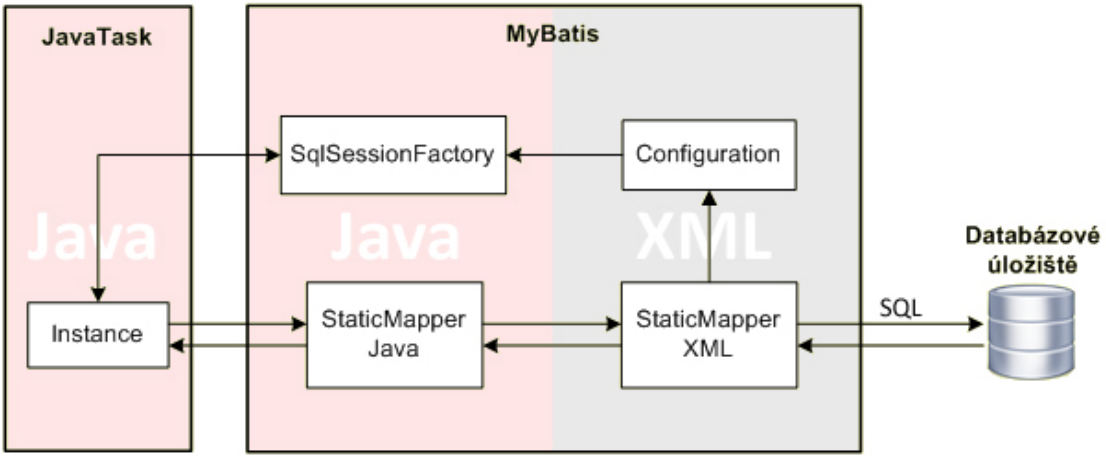
Výpis 6.5: Třída *JavaTask*

```
class JavaTask implements RunnableFuture {
    private final Integer networkNumber;
    private final Interface task;

    JavaTask(Integer networkNumber, Interface task) {
        this.networkNumber = networkNumber;
        this.task = task;
    }

    @Override public void run() throws Exception {
      SqlSession session = SqlSessionFactory.
      .getSqlSessionFactory().openSession();
      try {

          // stazeni dat pomoci MyBatis

        session.commit();
      } catch (Exception e) {
      } finally {
        session.close();
      }
}

private void returnNodesData(final Nodes[] nodesResults) {
        FX.deferAction(new Function0() {
            @Override public Void invoke() {
                task.setNodesData(nodesResults);
                return null;
            }
        });
}
```

I když se na první pohled zdá implementace asynchronní komunikace velice složitá je nutné si uvědomit, že většinu výše uvedeného kódu tvoří definice proměnných a objektů použitých pro předávání dat mezi jednotlivými vlákny. Po prvotní implementaci je tak rozšiřování předávaných informací mezi oběma vlákny aplikace spíše rutinní záležitostí. Přesto, že je výše uvedený kód trochu rozsáhlejší, je nutné ho v práci uvést pro lepší pochopení předávání dat v aplikaci, které je pro její funkci kritické.

6.2.2 Implementace MyBatis

Jak již bylo dříve avizováno, bude tato kapitola zaměřena na samotnou strukturu získávání dat a použité instance. Hlavním bodem aplikací využívajících MyBatis je instance třídy *SqlSessionFactory*, která je získávána pomocí třídy *SqlSessionFactoryBuilder*. Pro lepší přehlednost je možné si všechny instance nutné pro správnou funkci předávání dat prohlédnout na obrázku 6.4.



Obr. 6.4: Struktura rámce MyBatis.

SqlSessionFactory

Pro nastavení spojení je možné využít konfigurační údaje z připojeného XML souboru označeného jako „Configuration“ nebo je definovat přímo v Java kódu. Zde bude použito načtení z XML, které je přehlednější a také jednodušší.

Výpis 6.6: Příklad obsahu adresáře SqlSessionFactory

```
Reader reader = Resources.getResourceAsReader("app/
mybatis/xml/Configuration.xml");
FACTORY = new SqlSessionFactoryBuilder().build(reader);
```

Ve výpisu 6.6 je možné si všimnout obsahu třídy *SqlSessionFactory*, která se skládá pouze z načtení konfiguračního souboru a vytvoření nové instance třídy *SqlSessionFactoryBuilder*.

Configuration

Konfigurační XML soubor je zřejmě nejdůležitějším prvkem celé soustavy z obrázku 6.4. Obsahuje nastavení pro jádro systému MyBatis, včetně adresy databáze, jejích ovladačů a přihlašovacích údajů. Zároveň definuje, jak má spojení probíhat a zajišťuje jeho kontrolu. Dojde-li tedy k migraci databázového systému nebo změně přihlašovacích údajů, je nutné tento soubor aktualizovat, jinak nebude možné s databází navázat spojení.

Výpis 6.7: Příklad obsahu adresáře configuration

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
 <environments default="development">
  <environment id="development">
   <transactionManager type="JDBC"/>
   <dataSource type="UNPOOLED">
    <property name="driver" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
   </dataSource>
  </environment>
 </environments>
 <mappers>
  <mapper resource="app/mybatis/xml/StaticMapper.xml"/>
 </mappers>
</configuration>
```

MyBatis umožňuje široké nastavení spojení, pro které také definuje rozsáhlou sadu parametrů, jejichž podrobný popis se nachází v [17]. Zde bude uvedeno pouze základní nastavení, které je nutné pro správný chod aplikace. Jak je z výpisu 6.7 patrné, kromě již zmíněné definice ovladače, url adresy a přihlašovacích údajů, obsahuje konfigurační soubor XML hlavičku, která je vyžadována pro určení verze a použitého kódování XML dokumentu.

Jedním z rozšiřujících parametrů, obsaženým v MyBatisu až od verze 3.0, je schopnost nastavení různých běhových prostředí. To přináší možnost definovat v

jednom konfiguračním souboru několik prostředí, z nich každé může obsahovat rozdílné nastavení spojení. Mezi nimi se následně přepíná pomocí jednoduchého příkazu během vytváření instance. Seznam použitých mapovacích souborů, zde však pouze jeden, je uvozen tagy *<mappers> </mappers>*. V nastavení je nutné vždy použít celou cestu k souboru nebo využít parametru *typeAliases*, který jí přiřadí zkrácený název. Zde však použit není.

StaticMapper(XML)

V doposud probraných instancích nebylo stále uvedeno, jak jsou data ze serveru získávána. To bude objasněno pomocí výpisu 6.8, který zobrazuje příklad mapovacího souboru. Stejně jako u konfiguračního souboru i zde je možné vybrat, zda pro definici příkazů bude použit formát XML nebo anotace. Jelikož však anotace mají svá omezení a nehodí se pro složitější konstrukce SQL příkazů, které jsou v aplikaci použity, zaměří se další text pouze na mapování pomocí XML souboru. Jako první musí být uvedena obligátní XML hlavička, na níž navazuje konstruktor značící začátek mapování s parametrem namespace. Spojení parametrů *namespace* a *id*, který slouží jako identifikátor sady SQL příkazů, umožní volat mapovací sekvenci přímo pomocí jejího plně kvalifikovaného jména (např. app.mybatis.mapper.StaticMapper.selectNetwork).

Výpis 6.8: Příklad obsahu adresáře staticMapper (XML)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="app.mybatis.mapper.StaticMapper" >

  <select id="selectNodes" parameterType="int" resultType=
  "app.mybatis.bean.Nodes">
      SELECT
      Node.id,
      Node.name as "nodeName",
      Node.logicalAddress,

      FROM Node, Location, CartesianPoint

      WHERE Node.networkId = #{actualNetwork}
  </select>

</mapper>
```

Jak je vidět na obrázku 6.4, nacházejí se v přenosovém řetězci dva soubory pojmenované StaticMapper, z nichž jeden je typu XML a druhý Java. Druhý z jmenovaných slouží jako rozhraní pro přímé volání SQL příkazů mapovacího souboru z nově vytvořené instance umístěné ve vláknu *JavaTask*. StaticMapper (Java) je mapován do StaticMapper (XML) pomocí již zmíněného parametru *namespace* a jeho struktura s návazností na parametry *parameterType* a *resultType* bude popsána níže.

Kromě ve výpisu uvedeného příkazu pro vybrání dat z databáze (select), umožňuje MyBatis také jejich vkládání (insert), aktualizaci (update) a mazání (delete). Každá tato operace musí být avizována příslušnou hlavičkou s následujícími parametry.

- *id* - Jak již bylo uvedeno, slouží tento parametr k identifikaci jednotlivých sad příkazů pro vkládání, výběr, aktualizaci či mazání databázových dat. Zde je pro ukázku uveden pouze jednou, ale například mapovací soubor aplikace obsahuje čtrnáct bloků příkazů volaných dle aktuálně prováděné operace.

- *parameterType* - Při výměně dat s databází často vyvstane potřeba předat jí vstupní parametr, podle kterého má být příslušná operace provedena. Pro lepší pochopení je na výpisu 6.8 uveden příklad, kdy aplikace chce z velkého množství uzlů v databázi získat pouze ty, které patří do vybrané sítě. Jelikož vstupním parametrem bude číslo sítě, uvede se jako *parameterType* typ *integer* (int). Do následujícího SQL kódu je pak toto omezení implementováno příkazem *WHERE Node.networkId = actualNetwork*.

- *resultType* - Jelikož ve většině případů budou od databáze očekávána návratová data, je potřeba definovat jejich typ. Zde však, často nastane problém, kdy navrácených hodnot je více a jsou každá jiného typu. Z tohoto důvodu nelze použít jednoduché datové typy jako *int*, *float* a *String*, ale musí být použity Java objekty. Cesta k tomuto objektu je pak předána mapovacímu souboru pomocí parametru *resultType*.

Oba parametry (*resultType* a *parameterType*) pracují na stejném principu a je tedy možné uplatnit postupy uvedené u jednoho i na druhý. To umožní vyřešit situaci, kdy je potřeba předat databázi více vstupních parametrů, stejně jako v případě navrácení více hodnot z databáze, tedy pomocí Java objektu.

StaticMapper(Java)

Jak již bylo zmíněno, vytváří tato třída rozhraní mezi SQL příkazy mapovacího

souboru a nově vzniklou instancí. Toto rozhraní obsahuje seznam funkcí, které mají stejné názvy jako bloky SQL kódu, což umožňuje jejich jednoduché volání. Vstupní hodnota typu *pameterType* je mapovacímu souboru předána jako parametr volané funkce a navrácená data jsou následně uložena do objektu stejného typu jako funkce.

Výpis 6.9: Příklad obsahu adresáře staticMapper(Java)

```
public interface StaticMapper {
    List <Nodes> selectNodes(int actualNetwork);
}
```

Výpis 6.9 obsahuje funkci *selectNodes*, která nese jako vstupní hodnotu číslo sítě a navrácená data ukládá do objektu *Nodes*. Jelikož jeden objekt obsahuje informace pouze o jednom uzlu, kterých bude navráceno několik, je třeba příchozí objekty vkládat do seznamu definovaného komponentou *List*.

Instance

Po podrobném popisu všech nastavení a použitých tříd, bude v této části zaměřena pozornost na samotnou instanci implementovanou do aplikace prostřednictvím třídy *JavaTask*.

Pro vytvoření instance *SqlSession* mohou být použity dva přístupy, ze kterých zde však bude uveden pouze ten přehlednější a lépe pochopitelný. Jak je patrné z prvního řádku výpisu 6.10, pro otevření spojení se použije třída *SqlSessionFactory*, která načte všechny konfigurační údaje a naváže spojení s databází. Proběhne-li vše správně, předá toto spojení zpět vytvořené instanci a ta bude pokračovat vytvořením proměnné *mapper*, do které se načte cesta k rozhraní *StaticMapper*. Přidáním této proměnné k názvu funkce (shodným s názvem bloku SQL příkazů) a jejím zavoláním, dojde k provedení daného SQL kódu a navrácení požadovaných záznamů.

Výpis 6.10: Příklad vytvoření a použití instance

```
SqlSession session = SqlSessionFactory.getSqlSessionFactory().
.openSession();
StaticMapper mapper = session.getMapper(StaticMapper.class);
List<Nodes> nodes = mapper.selectNodes(networkNumber);
```

Seznam objektů s těmito záznamy se uloží do proměnné *nodes*, se kterou je možné dále pracovat dle potřeby.

Jak již bylo zmíněno jsou výše uvedené parametry a nastavení pouze nezbytným základem pro funkční komunikaci mezi aplikací a databázovým serverem. Kompletní popis doplňkových parametrů je uveden v [18]. Přes zdánlivou složitost je nastavení a implementace MyBatis jednodužší než implementace rámce Hibernate. Po nezbytné konfiguraci, která je pro drtivou většinu aplikací stále stejná, je jeho zprovoznění otázkou zadání přístupových údajů k databázi, bloků SQL kódu a názvů funkcí pro přímé volání.

7 ZÁVĚR

Úvodní dvě kapitoly diplomové práce jsou věnovány popisu dostupných webových služeb a datovým formátům sloužícím k přenosu dat. V první z nich jsou porovnány dvě dnes nejrozšířenější architektury REST a SOAP. Díky nižší energetické náročnosti a rychlejšímu zpracování informací dokázanému v [3], byla nakonec při návrhu zvolena implementace REST architektury. Ze stejného důvodu, tedy zaručení co nejnižší zátěže senzorových uzlů, byl ze srovnání datových formátů pro přenos dat vybrán JSON. Ten byl porovnán s rozšířenějším formátem XML, který se však kvůli náročnějšímu zpracování do navrhované aplikace nehodil.

Aplikace samotná umožňuje načítání mapového podkladu ve vektorovém formátu, což zaručí zachování jeho kvality jak při posunu, tak při přiblížení. Jako výchozí byl díky své otevřenosti a velkému množství volně dostupných editačních nástrojů zvolen formát SVG. Ten umožňuje vytvářet vícevrstvé soubory, které dokáže vytvořená aplikace rozdělit a zobrazovat pouze uživatelem požadované vrstvy. Do nich jsou následně vkládány senzorové uzly, jejichž prostřednictvím je uživateli předávána informace o naměřených hodnotách. Pro vizualizaci dlouhodobých dat je implementována funkce grafu, která nejen že umožňuje jejich vymezení datem, ale také nastavení maximální měřené hodnoty.

Po podrobném popisu funkcí aplikace a použitých principů je pozornost zaměřena na komunikační rozhraní, které se skládá hned ze dvou částí. Tou první a z hlediska předávání dat i složitější, je přístup k databázovému serveru. Složitost tohoto přístupu je způsobena jednovláknovostí jazyka JavaFX, díky které je možné provádět v daný okamžik pouze jednu činnost, což způsobovalo při stahování dat z databáze zablokování celé aplikace. Tento problém byl potlačen vytvořením vlákna běžícího na pozadí (v jazyce Java), které umožňuje během stahování dat bezproblémovou práci s aplikací. Pro snadnější komunikaci s databází je implementován rámec MyBatis, pomocí kterého jsou získávána data po předem definovaných blocích. Druhou částí komunikačního rozhraní je získávání aktuálních dat pomocí dotazů na webový server, běžící na každém uzlu. I když je tento princip v práci také podrobně popsán, má v jazyce JavaFX velkou podporu a s jeho implementací tak nebyl žádný problém.

Z celkového pohledu vyvinutá aplikace obsahuje všechny požadované funkce, čímž se podařilo splnit zadáním stanovené cíle.

LITERATURA

[1] BOOTH, David, et al. *Web Services Architecture* [online]. 11. 2. 2004 [cit. 2011-05-19]. Dostupné z URL: <http://www.w3.org/TR/ws-arch>.

[2] KADLEC, Jiří. *REST a webové služby v jazyce Java* . Brno, 2010. 73 s. Diplomová práce. Masarykova univerzita.

[3] YAZAR Dogan; DUNKELS Adam. *Application Integration in IP-Based Sensor Networks* [online]. 2009 [cit. 2011-05-19]. Swedish Institute of Computer Science. Dostupné z URL: <http://www.sics.se/ adam/yazar09efficient.pdf>.

[4] FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce. University of California. Irvine, 2000.

[5] CLARKE, Jim; CONNORS, Jim; BRUNO, Eric. *Developing Rich Internet Applications*. Prentice Hall, 2009. 359 s. ISBN 013701287X.

[6] SCRIBNER, Kenn; SEELY, Scott. *Effective REST Services Via .NET: For .NET Framework 3.5*. FT Press, 2009. 431 s. ISBN 0321613252.

[7] ALLAMARAJU, Subbu. *RESTful Web Services Cookbook*. O'Reilly Media, 2010. 314 s. ISBN 9780596801687.

[8] CROCKFORD, Douglas. *The application/json Media Type for JavaScript Object Notation (JSON)* [online]. 2006 [cit. 2011-05-19]. Dostupné z URL: <http://www.ietf.org/rfc/rfc4627.txt>.

[9] w3schools *XML Tutorial* [online]. 2010 [cit. 2011-05-19]. Dostupné z URL: <http://www.w3schools.com/xml/default.asp>.

[10] JSON *The Fat-Free Alternative to XML* [online]. [cit. 2011-05-19]. Dostupné z URL: <http://www.json.org/xml.html>.

[11] VAJSAR, Pavel. *Webové rozhraní pro monitoring senzorového pole*. Brno, 2010. 60 s. Diplomová práce. Vysoké učení technické v Brně.

[12] Oracle *JavaFX Documentation* [online]. 2010 [cit. 2010-11-28]. Dostupné z URL: <http://download.oracle.com/docs/cd/E17802_01/javafx/javafx/1.3/docs/api/index.html>.

[13] TOPLEY, Kim. *JavaFX Developer's Guide*. Addison-Wesley, 2010. 1150 s. ISBN 0321601653.

[14] Oracle blogs *JavaFX 1.2 Async* [online]. 2009 [cit. 2011-05-19]. Dostupné z URL: <http://blogs.sun.com/baechul/entry/javafx_1_2_async>.

[15] VALÍČEK, Arnošt. *Objektově-relační mapování v Javě* . Brno, 2007. 91 s. Diplomová práce. Masarykova univerzita.

[16] KING, Gavin; BAUER, Christian; BERNARD, Emmanuel; EBERSOLE, Steve. *Hibernate Getting Started Guide* [online]. 6. 4. 2011 [cit. 2011-05-19]. Dostupné z URL: <http://docs.jboss.org/hibernate/core/3.6/quickstart/en-US/html/>.

[17] Apache *MyBatis 3 - User Guide* [online]. 2011 [cit. 2011-05-19]. Dostupné z URL: <http://mybatis.googlecode.com/files/MyBatis-3-User-Guide.pdf>.

[18] BEGIN, Clinton; GOODIN, Brandon; MEADORS, Larry. *iBATIS in Action*. Manning Publications, 2007. 381 s. ISBN 1932394826.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

CSV – Comma Separated Values

HTTP – Hypertext Transfer Protocol

JSON – JavaScript Object Notation

ORM – Object-Oriented Programming

REST – Representational State Transfer

SOAP – Simple Object Access Protocol

SQL – Structured Query Language

SVG – Scalable Vector Graphics

URL – Uniform Resource Locator

XML – Extensible Markup Language

SEZNAM PŘÍLOH

A SCHÉMA DATABÁZE