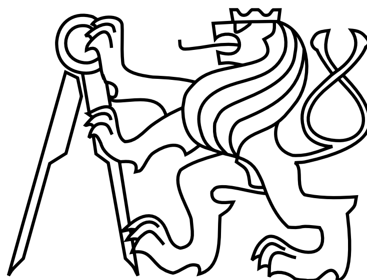
České vysoké učení technické v Praze
Fakulta strojní



Diplomová práce

Aplikace pro Industrial Edge

Bc. Jakub Znamenáček

Vedoucí práce: doc. Ing. Josef Kokeš, CSc.

Studijní program: Automatizační a přístrojová technika

Obor: Automatizace a průmyslová informatika

27. května 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Znamenáček** Jméno: **Jakub** Osobní číslo: **456392**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Automatizační a přístrojová technika**

Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Aplikace pro Industrial Edge

Název diplomové práce anglicky:

Application for Industrial Edge

Pokyny pro vypracování:

1) Úvod, definice problematiky, vymezení cílů a výstupů práce
2) Přehled a popis významných knihoven, frameworků, protokolů atd. důležitých pro backend aplikace
3) Vytvoření funkčního backendu aplikace v jazyce Java s použitím frameworku Spring Boot monitorující lisovací proces porovnáváním naměřené a referenční křivky lisovacího procesu a ukládající případné problémy do databáze PostgreSQL
4) Napsání vhodných unit testů pro důležité části kódu backendu
5) Vytvoření dockerového obrazu připraveného pro běh na systému SIEMENS Industrial Edge

Seznam doporučené literatury:

[1] COSMINA, Iuliana, Rob HARROP, Clarence HO a Chris SCHAEFER. Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools. 5th ed. 2017. Imprint: Apress, 2017. ISBN 9781484228081.
[2] GULATI, Shekhar a Rahul SHARMA. Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5. Imprint: Apress, 2017. ISBN 9781484230145.
[3] MIELL, Ian a Aidan Hobson SAYERS. Docker in practice. Second edition. Shelter Island, New York: Manning Publications, [2019]. ISBN 9781617294808.
[4] POLLARD, Barry. HTTP/2 in action. Shelter Island, NY: Manning Publications Co., [2019]. ISBN 9781617295164.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Josef Kokeš, CSc., U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **30.04.2021** Termín odevzdání diplomové práce: **10.06.2021**

Platnost zadání diplomové práce: ______________

doc. Ing. Josef Kokeš, CSc. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Michael Valášek, DrSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání	Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Poděkování

Rád bych zde poděkoval hlavnímu vývojáři knihovny Milo, Kevinu Herronovi, který si na mě, i přes jeho vytížení, udělal čas a pomohl mi pochopit základní principy této knihovny. Dále mé díky patří Manuelovi Aguado Puertasovi, za organizování a koordinaci schůzek s vývojářem frontendu, což mělo pozitivní vliv na rychlost vývoje aplikace. V neposlední řadě bych rád poděkoval Ronnymu Lehmannovi, který byl tak ochotný a prováděl testy aplikace s reálnými stroji. Jeho konstruktivní připomínky byly velmi přínosné. Též nemohu opomenout všechny ty, kteří mě na Stack Overflow nasměrovali k řešení některých problémů, s kterými jsem se v rámci vývoje backendu potýkal.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně s využitím zdrojů, které jsou všechny v práci uvedeny.

V Praze dne 10. 5. 2021 ....

Abstract

The content of this diploma thesis is the development of an application backend for the Industrial Edge system in the Java programming language. This application is designed to monitor the metal forming process via the OPC UA protocol and provides detection of deviations from the user-specified tolerance, feedback to the PLC which is controlling press and possible persistance of detected problems in the database for later display. In addition to the development of the backend itself, part of the work is also devoted to the creation of a Docker image of the entire application, which can be used with the Industrial Edge system.

Key words: Industrial Edge, Java, backend, press, metal forming, OPC UA, Docker

Abstrakt

Obsahem této diplomové práce je vývoj backendu aplikace pro systém Industrial Edge v programovacím jazyce Java. Tato aplikace je určena pro monitorování lisovacího procesu prostřednictvím protokolu OPC UA a zajišťuje detekci odchylek od uživatelem stanovené tolerance, zpětnou vazbu pro PLC řídící lis a případné ukládání zjištěných problémů do databáze za účelem pozdějšího zobrazení. Kromě samotného vývoje backendu je část práce věnována též tvorbě Dockerového obrazu celé aplikace, jenž je možný použít se systémem Industrial Edge.

Klíčová slova: Industrial Edge, Java, backend, lis, tváření kovů, OPC UA, Docker

Obsah

Seznam použitých zkratek **xi**

1 Úvod **1**
1.1 Automatizace v průmyslu . . . 1
1.2 Industrial Edge . . . 2
1.2.1 Spojení s cloudem . . . 3
1.2.2 Lokální aplikace . . . 3
1.3 Cíl diplomové práce . . . 3

2 Přehled použitých technologií **5**
2.1 OPC UA . . . 5
2.1.1 Milo . . . 7
2.2 HTTP . . . 7
2.2.1 HTTPS . . . 8
2.2.2 Struktura HTTP . . . 8
2.2.2.1 Stavové kódy . . . 9
2.2.2.2 Hlavičky . . . 10
2.2.2.3 Metody . . . 10
2.3 WebSocket . . . 10
2.3.1 STOMP . . . 11
2.4 Java . . . 11
2.5 Maven . . . 13
2.5.1 Správa závislostí . . . 13
2.6 Git . . . 15
2.7 Spring framework . . . 16
2.7.1 Spring Boot . . . 18
2.8 Testování kódu . . . 18
2.8.1 JUnit . . . 18
2.8.2 Mockito . . . 19
2.8.3 AssertJ . . . 20
2.9 Lombok . . . 20
2.10 MapStruct . . . 21
2.11 Práce s databází . . . 22
2.11.1 Hibernate . . . 22
2.11.2 JPA . . . 23

2.11.3 Spring Data JPA . . . 24
2.12 Docker . . . 24

3 Implementace 27
3.1 Testování kódu . . . 27
3.2 Základní architektura backendu . . . 28
3.3 API . . . 29
3.3.1 REST . . . 30
3.3.1.1 Správa PLC . . . 31
3.3.1.2 Správa nástrojů . . . 32
3.3.1.3 Správa logů . . . 34
3.3.2 WebSocket . . . 36
3.3.2.1 Změna stavu připojení PLC . . . 36
3.3.2.2 Změna používaného nástroje . . . 37
3.3.2.3 Detekce nového nástroje . . . 37
3.3.2.4 Stav výpočtu referenční křivky . . . 37
3.3.2.5 Vytvoření nového logu . . . 37
3.3.3 Zabezpečení komunikace . . . 37
3.4 Aplikační vrstva . . . 38
3.4.1 Zpracování dotazů zaslaných prostřednictvím API . . . 38
3.4.2 Automatická aktualizace informací . . . 38
3.4.3 Vyhodnocování cyklu . . . 39
3.4.3.1 Tvorba referenční křivky . . . 39
3.4.3.2 Validace naměřené křivky . . . 40
3.5 Komunikace s PLC . . . 41
3.5.1 Reprezentace spojení . . . 41
3.5.1.1 Připojení k PLC . . . 41
3.5.1.2 Odpojení od PLC . . . 42
3.5.2 Struktura dat v PLC . . . 42
3.5.3 Definice vlastních struktur . . . 43
3.5.4 Vytvoření spojení . . . 44
3.5.5 Automatické připojení při startu aplikace . . . 45
3.6 Databázová vrstva . . . 46
3.6.1 Aktuální implementace . . . 46
3.6.1.1 Struktura databáze . . . 46
3.6.1.2 Entity . . . 48
3.6.1.3 Repositáře . . . 49
3.6.1.4 Vlastní mapování tříd . . . 50
3.6.1.5 Vytváření logů . . . 51
3.6.2 Možná vylepšení . . . 52
3.6.2.1 Nevýhody a výhody současné implementace . . . 52
3.6.2.2 SCD . . . 52
3.6.2.3 Hibernate Envers . . . 52

4 Tvorba aplikace pro Industrial Edge **54**
4.1 Struktura aplikace 54
4.2 Tvorba Dockerových obrazů 54
4.2.1 Aplikace 54
4.2.2 Databáze 56
4.3 Docker-compose 56
4.4 Nahrání aplikace do zařízení 57

5 Závěr **58**

Literatura **60**

A Obsah přiloženého disku **63**

Seznam obrázků

1.1 Vývoj průmyslu . . . 1
1.2 Správa Edge zařízení . . . 2

2.1 Architektura OPC UA . . . 6
2.2 Reprezentace dat na serveru . . . 6
2.3 Odběr změn informací na serveru . . . 7
2.4 Struktura http zprávy . . . 8
2.5 Komunikace přes WebSocket [19] . . . 11
2.6 STOMP - message broker . . . 11
2.7 Popularita programovacích jazyků [14] . . . 12
2.8 Java Virtual Machine . . . 12
2.9 Správa závislostí . . . 14
2.10 Základní práce s Gitem . . . 16
2.11 Větvení . . . 16
2.12 Spring kontejner . . . 17
2.13 Integrace JUnit v IDE IntelliJ IDEA . . . 19
2.14 Zpracování anotací . . . 20
2.15 Hibernate ORM . . . 23
2.16 Porovnání kontejnerů a virtuálních strojů . . . 25
2.17 Architektura Dockeru . . . 25
2.18 Dockerový obraz . . . 26

3.1 Organizace testů . . . 27
3.2 Pokrytí testy . . . 28
3.3 Návrhový vzor pozorovatel . . . 29
3.4 Přehled REST API . . . 30
3.5 Výpočet referenční křivky . . . 39
3.6 Validace naměřené křivky . . . 40
3.7 Struktura dat v PLC . . . 43
3.8 Vytvoření spojení . . . 44
3.9 Využití paralelních vláken pro připojení PLC . . . 46
3.10 Struktura části databáze pro aktuální data . . . 47
3.11 Struktura části databáze pro logy . . . 47

4.1 Architektura kontejnerizované aplikace pro Industrial Edge . . . 54
4.2 Multi-stage build . . . 55

4.3 Proces nahrávání aplikace do Edge zařízení 57

5.1 Testování aplikace s reálným lisem 59

Seznam použitých zkratek

API: Application Programming Interface

AWS: Amazon Web Services

CI/CD: Continuous Integration/Continuous Deployment

CLI: Command Line Interface

CRUD: Create, Read, Update, Delete

DBMS: Database Management System

DTO: Data Transfer Object

EJB: Enterprise Java Beans

ER: Entity Relationship

HQL: Hibernate Query Language

HTTP: Hypertext Transfer Protocol

HTTPS: Hypertext Transfer Protocol Secure

HW: Hardware

IDE: Integrated Development Environment

J2EE: Java 2 Enterprise Edition

JAR: Java Archive

JPA: Java Persistence API

JPQL: Java Persistence Query Language

JVM: Java Virtual Machine

NPM: Node Package Manager

OPC UA: Open Platform Communications Unified Architecture

ORM: Object Relational Mapping

PLC: Programmable Logic Controller

REST: Representational State Transfer

SCADA: Supervisory Control And Data Acquisition

SCD: Slowly Changing Dimension

SQL: Structured Query Language

STOMP: Simple Text Oriented Message Protocol

SW: Software

TCP: Transmission Control Protocol

TDD: Test Driven Development

URL: Uniform Resource Locator

WAR: Web Application Resource

XML: Extensible Markup Language

YAML: YAML Ain't Markup Language

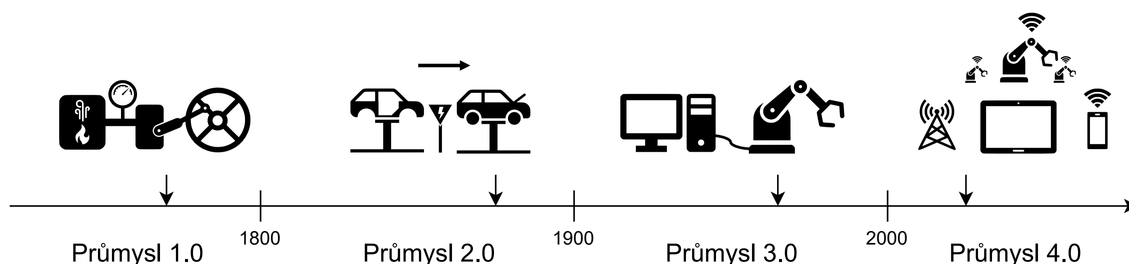
Kapitola 1

Úvod

1.1 Automatizace v průmyslu

Rychlý technologický vývoj v posledních letech a desetiletích nám každý den zjednodušuje a zpříjemňuje život. Stále výkonnější zařízení a pokročilejší algoritmy se zasluhují o zvýšenou úroveň automatizace, ať už se jedná o chytré rozsvícení žárovek, když dorazíme domů, či optimalizaci a řízení výrobních linek. Moderní technologie se rozšiřují závratným tempem a stávají se tak nedílnou součástí výroby téměř jakýchkoli produktů.

V počátcích zavádění automatizace výrobních procesu docházelo k inovaci pouze v malých segmentech výroby. Ty se pak chovaly jako samostatné jednotky, které neměly informace o tom co se děje v jiných částech. Avšak s postupujícím tlakem na zvýšení kvality, zkrácení výrobního času a redukci nákladů je nutné zajistit komunikaci jak mezi jednotlivými částmi továrny, tak i mezi továrnami samotnými. V současnosti se tak setkáváme s chytrými stroji, které neustále monitorují, detekují a predikují možné poruchy, o kterých informují a umožňují tím předejít zastavení výroby. Jakákoli nečinnost produkce totiž často výrobci může způsobit škody v řádech milionů.



Obrázek 1.1: Vývoj průmyslu

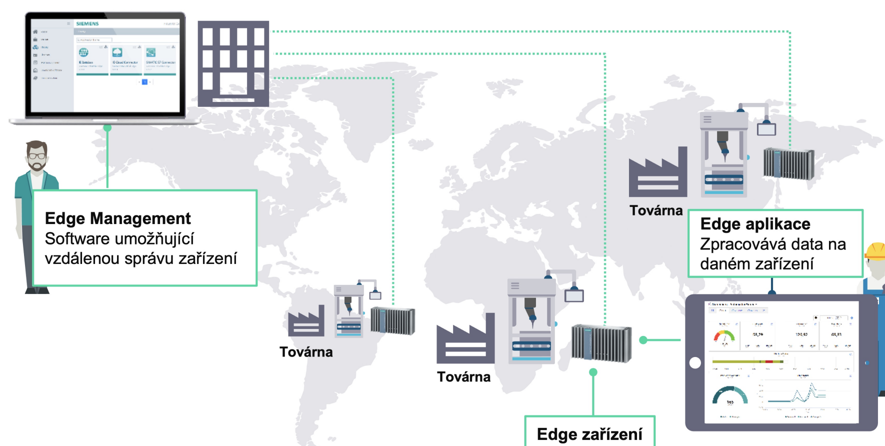
Všudy přítomná komunikace mezi zařízeními a sběr dat z nich je často brána za základy průmyslu 4.0 [2, str. 2]. Nacházíme se tak na hraně další velké průmyslové revoluce, která jako všechny předešlé výrazně vylepší kvalitu a efektivitu výroby. Většina zdrojů [13, str. 2], [8, str. 12] se shoduje a za první průmyslovou revoluci, tedy průmysl 1.0, je označován přechod od využívání lidské či zvířecí síly k využití parních strojů a celková mechanizace. Za průmysl 2.0

je po té považováno období devatenáctého století, kdy došlo k velkému rozmachu využívání elektrické energie a rozšíření masové výroby. Využívání výpočetní techniky a vznik PLC poté započal třetí průmyslovou revoluci. Celkový vývoj průmyslu je znázorněn na obrázku 1.1.

Za středobod automatizace by se zajisté stále dalo považovat PLC, které je dosud využíváno více než 95% [8, str. 15] výrobci automatizovaných linek. Jeho oblíbenost plyne z jednoduchého programování a spolehlivosti. PLC mají své místo hlavně u rychle vyhodnocovaných dat ze senzorů a následného řízení akčních členů. Do této kategorie tak může spadat řízení ramene robota v automobilovém průmyslu či řízení teploty pece v průmyslu potravinářském. Komplikované je naopak použití PLC pro zpracovávání velkého množství dat pomocí náročných algoritmů a zajištění bezpečné správy přes internet. Zde najde své uplatní nový systém Industrial Edge od firmy SIEMENS.

1.2 Industrial Edge

Podstata Edge zařízení je ta, že se nacházejí tak blízko zdroji dat, jak je to jen možné. Pracují na zabezpečené firemní síti připojené k jednomu či více PLC a umožňují sběr, zpracování a vyhodnocení dat z právě probíhajícího procesu či jejich odeslání do cloudu. Platforma Industrial Edge nabízí jednoduchou správu všech těchto zařízení z jednoho místa viz obrázek 1.2.



Obrázek 1.2: Správa Edge zařízení

Toto je velkou výhodou při instalaci a aktualizaci aplikací v továrnách po celém světě. Software pro tento systém je založen na technologii kontejnerizovaných aplikací, což přináší značnou řadu výhod. Mezi ty mimo jiné patří například jednoznačné oddělení jednotlivých aplikací od sebe či nezávislost na použitém programovacím jazyce. Většina softwarových vývojářů tak nebude mít problém napsat svou vlastní aplikaci, což ve výsledku povede k jejich velké rozmanitosti a tím tak pokrytí širokého množství problémů. Dalším benefitem, který

tato platforma přináší je obchod s aplikacemi, tak jak jej známe z moderních operačních systémů. Odtud je možné si rychle a pohodlně stáhnout dostupný placený a bezplatný software, nebo provést jeho aktualizaci pomocí jednoho kliknutí.

1.2.1 Spojení s cloudem

Aplikace zajišťující spojení PLC s cloudovou platformou jako je například AWS, Azure, nebo MindSphere přináší mnoho nových možností. Lze pomocí nich například zajistit přístup k aktuálním i historickým datům odkudkoli na světě, nebo využít možnosti pokročilé analýzy a umělé inteligence. Na základě těchto informací je pak možné zlepšit efektivitu a profitabilitu produkce ve veškerých odvětvích výroby.

Zasílání všech dat do cloudu může být i pro malou výrobní linku komplikované, neboť je rychle dosaženo obrovského množství informací, které by zbytečně přetěžovaly síť a zabíraly velké místo v datových uložištích. Výhodné je tak sbíraná data předzpracovávat a třídit pomocí aplikací běžících lokálně a pouze informace, které jsou důležité, poté zasílat do cloudu.

1.2.2 Lokální aplikace

Naopak lokální aplikace jsou aplikace zamýšlené pro funkci v zabezpečené firemní síti. Zajišťují zpracování a vyhodnocení dat týkajících se výroby, ke kterým mají prakticky okamžitý přístup. Právě nízká odezva je jednou z výhod oproti cloudovým řešením. Nebývá tak problém vyhodnocovat časově kritické informace a výsledky předat zpět v podobě zpětné vazby či je sdílet s jinými stroji. Tyto aplikace většinou nejsou nezbytné pro běh celého systému, a tak jejich výpadek či chyba nezpůsobí zastavení celé linky.

Bezpečnost je též jedním z důvodů existence aplikací běžících lokálně. I přes sebevětší snahu poskytovatelů cloudových služeb o jejich zabezpečení zde může dojít k úniku dat. Nemusí se jednat nutně o vadu v softwaru, ale problémem může být i lidská chyba v podobě tzv. sociálního hackování. V případě zpracování dat přímo na zařízení umístěném v továrně, které je připojeno pouze k lokání síti toto nebezpečí nehrozí, a tak může být preferováno u obzvláště citlivých dat.

1.3 Cíl diplomové práce

Velmi důležitou součástí při lisování je nástroj, který je v přímé interakci s tvářeným materiálem. Je proto důležité, aby byl odolný, což odráží často i jeho vysoká cena. Jakékoli poškození proto může znamenat velké náklady a to jak z důvodu nutnosti vyměnit nástroj (cena nástroje + doba, kdy výrobní linka nemohla běžet), tak kvůli tomu, že než dojde k detekci vady nástroje, tak jsou vyráběny výrobky nedostatečné kvality.

Cílem této diplomové práce je vytvořit základní backend pro lokální Industrial Edge aplikaci zajišťující monitorování lisovacího procesu. Výsledná aplikace by měla sloužit hlavně jako ukázka možností systému Industrial Edge a jako podklad pro další vývoj v této oblasti.

Samotné monitorování bude probíhat na základě porovnání referenční a právě naměřené křivky. Data o naměřené křivce budou získávána společně s ostatními důležitými informacemi

přes protokol OPC UA z PLC obsluhujícího daný lis. Referenční křivka naopak bude vytvářena v rámci backendu z uživatelem předvoleného počtu cyklů lisovacího procesu. Jakákoli odlišnost mezi těmito křivkami vyšší než navolená tolerance poté bude zaznamenána do databáze pro pozdější kontrolu. Informaci o detekci problému bude dále možné zaslat do PLC, které na ni bude moci odpovídajícím způsobem reagovat. Toto by mělo ve výsledku výrobci, který se rozhodne tuto aplikaci využívat, ušetřit náklady spojené s nesprávným během lisu.

V rámci vývoje backendu bude navrženo smysluplné API pro správu všech důležitých částí výsledné aplikace, jako například připojení k PLC, zapnutí, čí vypnutí automatického monitorování pro jednotlivé nástroje či spuštění kalkulace referenční křivky. S tímto API bude interagovat frontend, který bude spolu s backendem v rámci kontejnerizované aplikace možné nahrát do Edge zařízení.

Kapitola 2

Přehled použitých technologií

Tato kapitola je věnována nejdůležitějším technologiím, které byly použity při vytváření aplikace. Budou zde zmíněny jak protokoly použité ke komunikaci, tak i knihovny a frameworky zjednodušující tvorbu či testování programu. Jejich výběr byl dán již zadáním diplomové práce, či se jedná nejvyužívanější open source možnosti v dané oblasti. Rozhodnutí využít tyto standardy bylo učiněno z důvodu velmi dobré dokumentace, aktivní komunity a velkého množství návodů k použití. Případné významné alternativy pro danou technologii budou zmíněny v konkrétních podkapitolách.

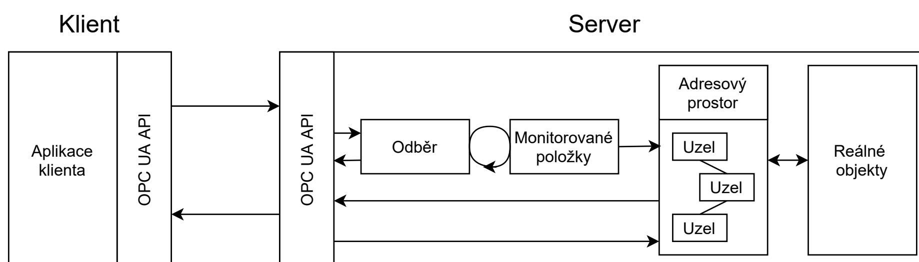
Tato sekce nemůže v žádném případě obsáhnout všechny informace o dané technologii, což dokazuje i nespočet knih napsaných o jednotlivých možnostech. Budou zde tak vyzdviženy ty nejdůležitější vlastnost a oblasti, které přímo souvisí s tvorbou aplikace, či jsou důležité k pochopení dané problematiky.

2.1 OPC UA

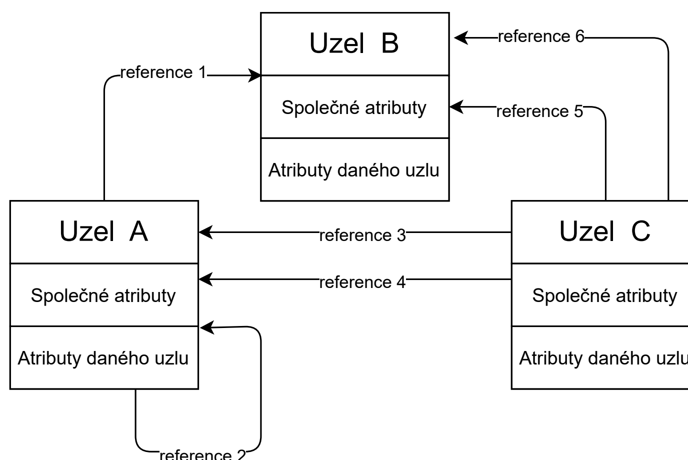
OPC UA je komunikační protokol nezávislý na platformě pro průmyslovou automatizaci, který byl vyvinut OPC Foundation a publikován v roce 2008 [37]. Jedná se o nástupce klasického OPC, které bylo založeno na technologiích Microsoftu COM a DCOM, což zamezovalo použití na jiných strojích než počítačích s operačním systémem Windows. Architektura OPC UA je postavena na principu klient server [40, str. 783] viz obrázek 2.1. Každý systém využívající tohoto komunikačního protokolu může mít více klientů a serverů, které mezi sebou mohou komunikovat. Jedno zařízení může být dokonce zároveň klientem a serverem. Protokol OPC UA má široké využití kam se řadí SCADA systémy, řídící panely, distribuované řídící systémy, PLC či MES systémy.

Jak je uváděno v [24, str. 11], komunikace mezi klientem a serverem může probíhat jak po lokální síti tak přes internet. Po lokální síti se kvůli nižší odezvě volí optimalizovaný binární TCP protokol, kdežto při komunikaci přes internet se používají standardy jako XML a HTTP umožňující snazší průchod přes firewall. Jelikož je komunikační model abstraktní, volba konkretní implementace neovlivní jakoukoli jinou část systému.

OPC UA při mapování reálných objektů využívá objektově orientovaných principů jako je například dědičnost. Toto mapování probíhá vždy na straně serveru viz obrázek 2.1. Organizace informací v adresovém prostoru je řešena sítí na sebe navzájem ukazujících uzlů.



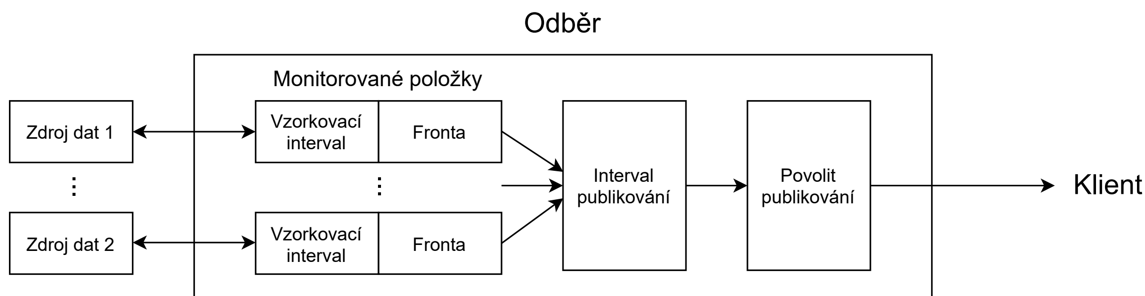
Obrázek 2.1: Architektura OPC UA



Obrázek 2.2: Reprezentace dat na serveru

Tyto uzly mohou být různých druhů, například existují uzly představující jednotlivé instance či uzly reprezentující jednotlivé datové typy. Každý uzel je popsán množinou atributů, viz obrázek 2.2. Některé jsou specifické pro daný uzel, jiné jsou všem uzlům společné. Mezi společné atributy patří například BrowseName, který definuje název jenž je klientovi zobrazen při procházení adresového prostoru, NodeClass udávající typ uzlu, nebo NodeId. Právě NodeId je nejdůležitějším atributem, neboť jednoznačně identifikuje daný uzel na serveru viz [24, str. 22] a dovoluje tak například specifikovat uzel jehož informace chce klient číst. Tento informační model umožňuje stejný přístup ke všem druhům informací a zároveň dovoluje data reprezentovat ve více různých hierarchiích.

Kromě klasického čtení a zápisu dat OPC UA umožňuje přihlásit se k odběru jejich změn. Monitorované položky viz obrázek 2.1 představují zdroje informací, které klienta zajímají. Odběr pak tvoří větší skupinu monitorovaných položek. Nejdůležitějšími nastaveními monitorovaných položek jsou vzorkovací interval, který určuje jak často bude kontrolován zdroj dat a fronta, jejíž velikost odpovídá počtu zapamatovaných hodnot, které klientovy mohou být odeslány. Pro odběr je pak volen interval publikování, tedy rozestupy mezi tím, kdy jsou všechna data o změnách zasílána klientovy a pak to, zda je vůbec publikování informací umožněno viz obrázek 2.3.



Obrázek 2.3: Odběr změn informací na serveru

V [24, str. 132-138] lze dohledat, že připojení klienta k dostupnému serveru probíhá přes takzvaný discovery server, ke kterému se jednotlivé OPC UA servery registrují. Tento pomocný server klientovi poskytuje informace o dostupných přístupových bodech. Ty definují síťový protokol, který má být použit ke komunikaci, adresu serveru a vyžadované zabezpečení. Každý OPC UA server je zároveň i discovery serverem, což zajišťuje stejnou funkcionalitu i tehdy, pokud je tento OPC UA server na síti jediný.

2.1.1 Milo

Milo je open source implementace UPC UA v jazyce Java vytvořená Kevinem Herronem [12]. První přidání kódu na GitHub proběhlo v roce 2016 [11] a jak sám autor uvádí, jednalo se o alternativu k oficiální verzi od OPC Foundation a placeným SDK jako Prosys. Oproti oficiální verzi však kromě základní implementace (přenos zpráv, kódování či šifrování) nabízí i SDK, které vývojáři usnadňuje práci s ní. V současné době již OPC Foundation ukončila vývoj Java implementace a udržuje pouze .NET verzi [26], takže je Milo jedinou stále aktivní open source implementací a proto se jí dostává stále větší pozornosti od velkých firem jako je například Bosch.

Architektura Milo se skládá z stack-core, stack-client, stack-server, sdk-core, sdk-client a sdk-server. Stack představuje elementární implementaci komunikace přes protokol OPC UA, kdežto SDK staví na těchto základech a usnadňuje jejich využití v konečné aplikaci. Jednotlivé části jsou rozděleny zvlášť pro klienta a pro server s tím, že společné části se nachází v core. Při vytváření aplikace, která bude sloužit jako OPC UA client stačí importovat pouze část stack-client, neboť všechny potřebné části jsou v ní obsaženy jako její závislosti.

Jedinou nevýhodou Milo je, že bohužel stále není hotová její dokumentace. Naštěstí díky široké a aktivní komunitě a rozsáhlé sbírce příkladů nahraných přímo v repositáři na GitHubu [11] není tento problém tak veliký, ale existence dokumentace by zajisté novým vývojářům, kteří s Milo pracují usnadnila začátky.

2.2 HTTP

Jak je možné dočíst se v [23, str. 1-5] historie HTTP (Hypertext Transfer Protocol) započala roku 1989, kdy jen Tim Berners-Lee poprvé použil pro zajištění komunikace mezi počítači

v evropské organizaci pro jaderný výzkum, která je známa pod svou zkratkou CERN. Tento protokol byl zprvu používán pouze k přenosu hypertextových dokumentů, které umožňovaly odkazovat se na jiný soubor. V současné době má však tento standard značně rozsáhlejší využití a často se tak můžeme setkat s tím, že je používán pro zasílání nejrůznějších datových souborů.

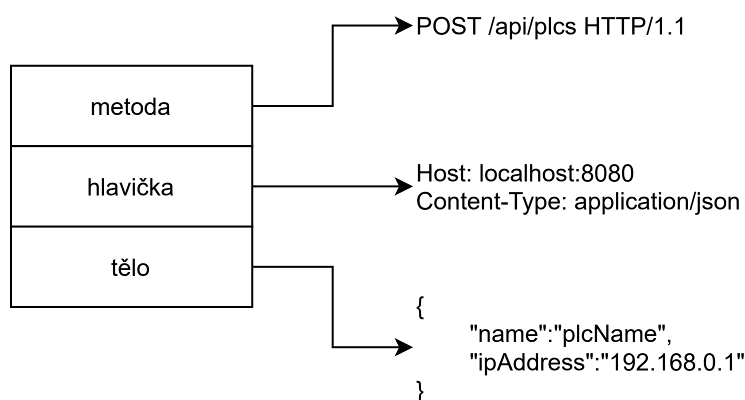
2.2.1 HTTPS

HTTP jako takové zasílá data nešifrovaně (plain-text) [27, str. 28], což může být v určitých situacích velký problém. Typickým příkladem je přihlašování na webové stránce, kdy klient předává serveru přihlašovací údaje. Ty mohou být na síti zachyceny útočníkem a jelikož nejsou šifrované, tak je lze snadno zneužít.

V [27, str. 28-31] je možné dohledat, že HTTPS (Hypertext Transfer Protocol Secured) je šifrovaná verze HTTP, která využívá TLS (Transport Layer Security) protokolu. Nedochází zde pouze jen k šifrování dat samotných, ale též ke kontrole toho, zda zaslaná data nebyla po cestě nikterak pozměněna (data jsou digitálně podepsána) a ověření toho, že server je opravdu ten, za koho se vydává. Využívá se zde asymetrické kryptografie. Při prvním připojení klienta k serveru zašle server veřejný klíč v podobě certifikátu. Tento certifikát je podepsán certifikační autoritou, což zaručuje autenticitu serveru. Pokud totiž klient důvěřuje certifikační agentuře, pak může důvěřovat i samotnému certifikátu. Výchozí certifikáty jsou poskytovány dodavateli softwaru, mezi které patří například Apple, Microsoft, Opera a tak dále. [6].

2.2.2 Struktura HTTP

Struktura HTTP dotazu se dá na základě [35, str. 47-49] rozdělit do tří hlavních sekcí a to do metod, hlaviček a samotného těla zprávy.



Obrázek 2.4: Struktura http zprávy

Jak je patrné z obrázku 2.4, každá zpráva začíná specifikováním dané metody (POST) a verzí protokolu (HTTP 1.1). Po té následuje sekce s HTTP hlavičkami. Tato sekce může

být vynechána či v ní může být více hlaviček. Jelikož je však využita verze protokolu 1.1 je hlavička „Host“ povinná [27, str. 23]. Poslední částí je pak tělo zprávy. To v tomto případě obsahuje JSON se zasílanými daty. Z vyobrazené zprávy je tedy zřejmé, že slouží k vytvoření nového objektu s názvem „plcName“ a IP adresou "192.168.0.1".

Odpovědi zasílané pomocí HTTP mají pak podobnou strukturu jako dotazy, jen je zde nahrazena část s metodou. Její místo zde zastávají stavové kódy.

2.2.2.1 Stavové kódy

Stavové kódy, které server posílá v odpovědi klientovy se dají rozdělit do pěti kategorií viz tabulka 2.1.

stavový kód	význam
100-199	informační charakter
200-299	úspěšná operace
300-399	přesměrování
400-499	chyba na straně klienta
500-599	chyba na straně serveru

Tabulka 2.1: Kategorie stavových kódů [35, str. 53]

Z tabulky vyplývá, že počet číselných hodnot, kterých stavový kód může nabývat je značný. Z přehledu stavových kódu v [41, str. 22-31] je však jasné, že zdaleka ne všechny možnosti jsou definované. I přes to je jejich počet velký a proto tu budou zmíněny jen ty nejzákladnější využité v samotném backendu.

200-299 Z této oblasti, tedy úspěšných operací stojí za zmínění 200 OK. Jedná se o standardní odpověď pro úspěšný HTTP požadavek a je tak například zasílána při úspěšném smazání entity. 201 Created po té zasílá server při úspěšném vytvoření entity, tedy jako reakci na požadavek s metodou POST.

400-499 Zde je nutné zmínit asi nejznámější stavový kód a to 404 Not Found, který klient obdrží, pokud dotazovaný cíl nebyl nalezen. Často se také můžeme setkat s kódem 401 Unauthorized, který informuje o tom, že dotaz nemohl být zpracován kvůli tomu, že nebyl autorizován či s 400 Bad Request, který je reakcí na nesprávnou syntaxi dotazu. Posledním zmíněným z této kategorie pak je 409 Conflict, který jak již název napovídá signalizuje konflikt, který dotaz vyvolal.

500-599 Pokud je klientem na server zaslán platný dotaz, který splňuje všechny podmínky, které server vyžaduje, a je spuštěna operace při níž dojede k chybě obdrží klient s největší pravděpodobností stavový kód 500 Internal Server Error. Ten jej informuje právě o tom, že na straně serveru nastala chyba, která mohla být způsobena například chybou v programu.

2.2.2.2 Hlavičky

V [35, str. 53-57] je možné se dočíst, že hlavičky, zasílané v HTTP zprávě, je možné rozdělit do čtyř kategorií a to obecné, hlavičky požadavku, hlavičky odpovědi a hlavičky těla zprávy.

Pomocí hlaviček požadavku specifikuje klient například jaké typy souborů je schopen číst na základe čehož mu server zasílá odpověď. Hlavičky odpovědi po té mohou sloužit k informování klienta o stáří žádaného souboru či softwaru, který na serveru implementuje HTTP. Hlavičky těla zprávy zajišťují dodatečné informace o zasílané entitě. To může být například typ obsahu či jeho délka.

2.2.2.3 Metody

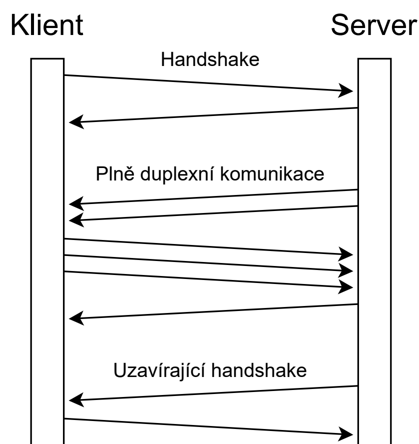
Z HTTP metod zde budou popsány ty, které jsou typické pro REST API. Mezi ně patří GET, PUT, POST, DELETE. Metoda GET je nejjednodušší z jmenovaných a slouží k vyžádání dat ze serveru. Jejím pravým opakem je metoda DELETE, která umožňuje klientovy požádat o smazání daného souboru na serveru. Zbývající metody PUT a POST jsou si podobné v tom, že obě umožňují klientovy zaslat data na server. Zásadní rozdíl však představuje interpretace požadavku. Zatímco v případě POST server vytvoří nová data, tak v případě PUT dojde k vložení dat již do existujícího objektu. Zjednodušeně řečeno metoda POST je využívána k vytvoření nového objektu, kdežto PUT slouží k jeho aktualizaci. Podrobnější informace lze dohledat v [41, str. 10-22].

2.3 WebSocket

WebSocket je velice jednoduchý, nízkoúrovňový protokol, který umožňuje okamžitou obousměrnou (plně duplexní) komunikaci přes TCP. Neslouží jako náhrada HTTP protokolu, ale spíše jako jeho doplněk tam, kde je zapotřebí zajistit nízkou odezvu spojení či odeslat informaci ze serveru klientovi bez toho, aby byl zaslán požadavek.

Jak se lze dočíst například v [19], před existencí WebSocketu byly tyto situace řešeny na základě HTTP protokolu pomocí opakovaného zasílání požadavků na server v pravidelných intervalech tzv. pooling. Další alternativou byl long pooling, tedy zaslání dotazu s tím, že TCP spojení zůstane otevřené dokud na serveru nedojde ke změně, která je zaslána zpět a spojení musí být klientem znovu otevřeno pomocí nového dotazu. Poslední možností bylo HTTP streamování, kdy dochází stejně jako u long pooling s prvním dotazem k otevření spojení, ovšem to není uzavřeno odpovědí serveru, ale zůstává neustále otevřené. Nevýhodou streamování je, že je pouze jednostranné (poloviční duplex) a data tak mohou téci pouze ze serveru.

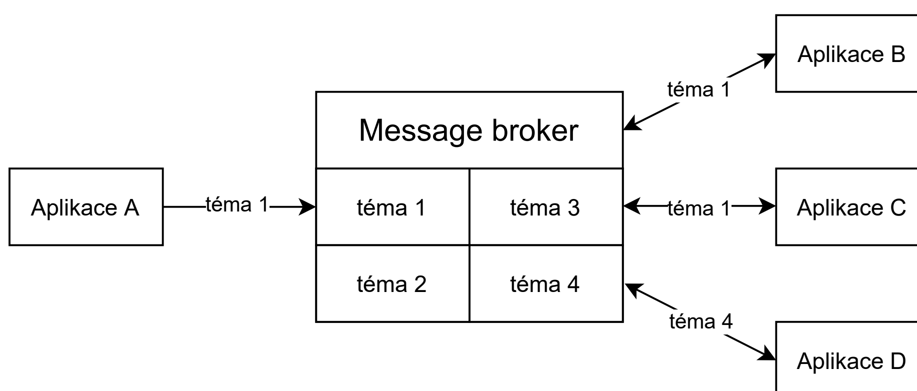
Komunikace přes WebSocket začíná handshakem [21, str. 114], který je zajišťován pomocí HTTP protokolu zasláním požadavku Upgrade se speciální hlavičkou Sec-WebSocket-* pro změnu protokolu viz obrázek 2.5. Obdobně jako u HTTP zde existuje zabezpečena verze WebSocket sercure, která slouží ke stejnému účelu a též využívá TLS [21, str. 79].



Obrázek 2.5: Komunikace přes WebSocket [19]

2.3.1 STOMP

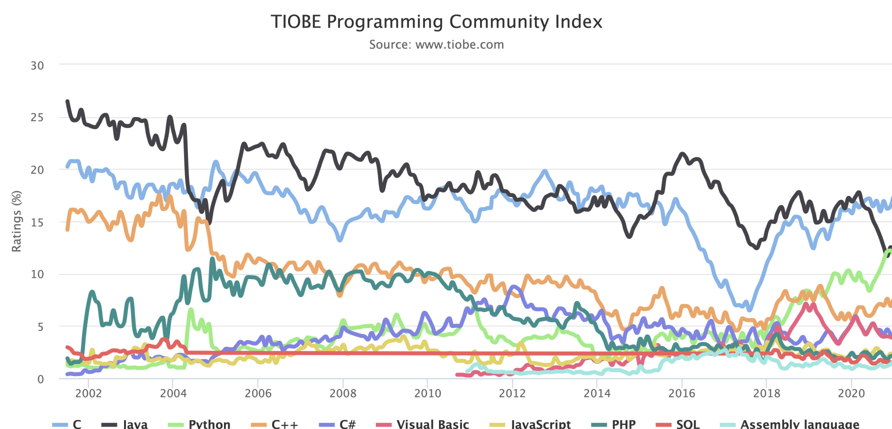
Z [20] vyplývá, že WebSoket jako takový nedefinuje formát zpráv a proto je vhodné využívat subprotokoly. Jedním ze subprotokolů, které mohou být využity je STOMP. STOMP není omezen pouze na WebSocket, ale může fungovat přes jakýkoli plně duplexní protokol. Komunikace neprobíhá přímo mezi jednotlivými aplikacemi, ale přes prostředníka viz obrázek 2.6. Aplikace, které chtějí získávat informace na dané téma se přihlásí k jeho odběru a jakmile je na něj zaslána zpráva, tak jsou všechny aplikace odebírající toto téma informovány.



Obrázek 2.6: STOMP - message broker

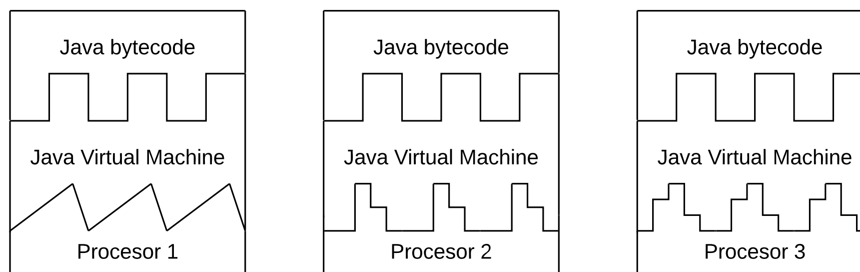
2.4 Java

Java je vedle C, C++ či Pythonu jedním z nejpoužívanějších a nejoblíbenějších programovacích jazyků na světě viz obrázek 2.7.



Obrázek 2.7: Popularita programovacích jazyků [14]

Jak se lze dočíst v sekci věnované historii jazyka v [32], byla vytvořena týmem společnosti Sun Microsystems pod vedením Jamese Goslinga jako programovací jazyk nezávislý na nekonkrétním hardwaru. Prvotního největšího rozšíření nabyla díky tomu, že se stala jazykem internetu a její aplety oživily do té doby statické webové stránky. V současnosti ji v této oblasti již nahradil JavaScript, ovšem Java je dnes využívána pro programování aplikací pro operační systém Android (zde ji ovšem vytlačuje Kotlin, který se stal v roce 2018 [16] výchozím programovacím jazykem pro Android) a podnikových aplikací. Umožnila vznik takových softwarů jako jsou OpenOffice, Gmail či Atlassian [34]. V současnosti je Java vlastněná společností Oracle.



Obrázek 2.8: Java Virtual Machine

Jedná se o kompilovaný jazyk, který však není kompilován do strojového kódu jako je tomu například u jazyku C, ale do bytecodu, což jsou v podstatě instrukce pro JVM (Java Virtual Machine). Tento virtuální stroj běží nad samotným hardwarem a umožňuje tak funkci stejného programu na jakémkoli procesoru viz obrázek 2.8. Takovéto řešení v sobě kombinuje výhody kompilovaného jazyku (rychlost) a výhody jazyku interpretovaného (možnost běžet na jakémkoli hardwaru) a bylo na svou dobu unikátní. I přes to, že JVM bylo zamýšleno pouze pro tento programovací jazyk, v současné době podporuje mnoho

skriptovacích a programovacích jazyků jako jsou například Scala [18, str. 175], Groovy [18, str. 401] či Kotlin [18, str. 326], který je preferovaným jazykem pro Android. Právě proto bývá JVM často považováno za nejdůležitější součást ekosystému Javy [33].

Mezi další typické vlastnosti patří to, že se jedná o objektově orientovaný jazyk. Kód je tak organizován do tříd obsahujících metody a atributy. Z těchto tříd, které slouží jako jakési šablony, jsou vytvářeny objekty. Toto zaručuje lepší organizaci větších projektů. Dále je též staticky a silně typovaný. Typ proměnné je tedy volen explicitně a je stálý. Díky tomu je kód programu sice delší, ovšem vzniká méně chyb.

Podobně jako u programovacího jazyku Swift je zde automaticky spravováno přiřazování a čištění paměti pomocí tzv. grabage collectoru. Ten se stará o to, že objekty, které již nejsou používány (neexituje na ně reference) jsou automaticky odstraněny.

Stejně jako jazyky, kterými se dorozumíváme, i programovací jazyky prochází neustálým vývojem. Nejinak tomu je i zde. Java od verze 8 získala spousty vylepšení. Nyní jsou podporovány streamy, které umožňují zápis dlouhých nejčastěji používaných typů smyček jedním řádkem kódu. Příkladem může být třeba seřazení listu objektů podle jednoho jejich parametru. Dále došlo k vylepšení a zjednodušení paralelního programování pomocí CompletableFuture či přidání lambda výrazů, které umožňují jednoduší předávání funkcionality mezi jednotlivými třídami. Více podrobností lze nalézt v [38].

2.5 Maven

Jedná se o open source framework určený ke správě softwarových projektů v jazyce Java. Maven vývojáři usnadňuje samotnou tvorbu aplikace, její testování a organizaci závislostí. Dále se mimo toho též zaručil za standardizaci struktury projektu, což umožňuje jednodušší přechod mezi jednotlivými projekty. Byl vytvořen neziskovou organizací Apache Software Foundation v roce 2004 [39, str. 1]. Mezi jeho alternativy se řadí Ant v kombinaci s Ivy či Gradle. Mavem má však stále dominantní postavení, čemuž vděčí hlavně své jednoduchosti používání.

Klíčovým souborem je pom.xml, který obsahuje konfigurační informace pro Maven. Tento soubor se nachází v rootu projektu. Jsou zde specifikované například informace o daném projektu, formát v jakém má být vytvořen, výčet závislostí či jaké profily mohou být použity.

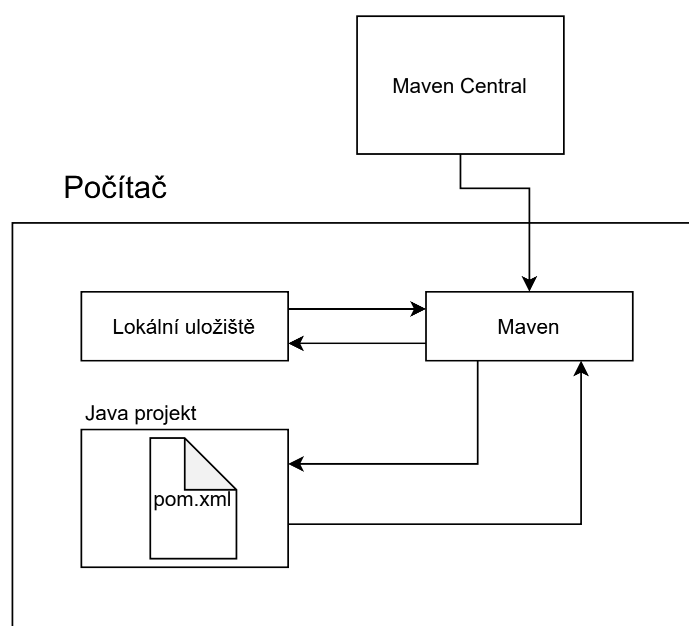
2.5.1 Správa závislostí

Jednou z nejdůležitějších funkcí Mavenu je správa závislostí. Většina nově tvořených aplikací závisí na softwaru, který někdo již vytvořil. Před používáním nástrojů jako je Maven tak bylo třeba jít na stránky softwarového vývojáře a danou knihovnu či framework si odtamtud stáhnout v podobě JAR souboru. Tento soubor však mohl záviset na jiných knihovnách, které byly opět nutné získat ručně. Problém však nekončil prvotním stažením potřebného softwaru, ale opakoval se při každé jeho aktualizaci.

Maven tuto situaci vyřešil specifikováním závislostí pomocí groupId (identifikuje vývojáře či organizaci), artifactId (identifikuje konkrétní software) a version (verze softwaru) v pom.xml viz níže.

```
<dependencies>
    <!--OPC-UA COMMUNICATION-->
    <dependency>
        <groupId>org.eclipse.milo</groupId>
        <artifactId>sdk-client</artifactId>
        <version>0.5.3</version>
    </dependency>
</dependencies>
```

Tato trojice, známá také jako GAN, jednoznačně určuje na čem daný projekt závisí a co je pro jeho správnou funkci nutné stáhnout. Výchozím místem odkud jsou soubory stahovány je Maven Central. Maven nejprve zkontroluje jaké závislosti jsou potřebné podle pom.xml uloženém v Java projektu, zjistí zda jsou dostupné lokálně ve složce .m2 a pokud ne, tak si je stáhne z Maven Central viz obrázek 2.9. Podrobnější informace lze dohledat v [39, str. 23-35]



Obrázek 2.9: Správa závislostí

Kromě definování jaké závislosti jsou potřebné Maven umožňuje určit i konkrétní fázi vývoje, kdy jsou používané. Ne vždy je totiž nutné do finálního souboru přibalovat všechny závislost. Příkladem může být framework JUnit, který je potřebný pouze při testování a však ve výsledném JAR souboru by byl zbytečný a pouze zvětšoval jeho velikost. Toto je voleno pomocí scope u dané závislosti v pom.xml. Maven umožňuje volit ze šesti možných vývojových fází. Pokud není specifikována žádná konkretní je výchozí možností požití závislosti ve všech fázích vývoje.

Jak již bylo zmíněno výše, samotné závislosti daného projektu mívají často své závislosti, které jsou automaticky získávaný s nimi. Toto chování je ve většině případů žádoucí, někdy

však chceme specifikovat části, které chceme vynechat. Tohoto je dosaženo pomocí exclusion specifikovaného u dané závislosti. Příkladem může být využití spring-boot-starter-test, který obsahuje jak JUnit 4 (starší verze) tak JUnit 5 (aktuální verze). Pokud tedy vývojář ví, že bude používat pouze novou verzi, může Mavenu říci, ať nestahuje verzi starší.

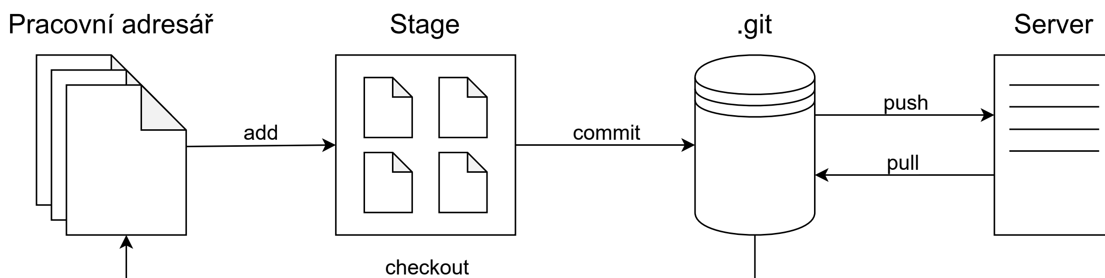
2.6 Git

Při vývoji téměř jakéhokoli softwaru je potřeba udržovat jeho různé verze. Důvodem může být snaha zajistit si možnost návratu k funkčnímu programu v případě, že aktuální verze obsahuje nějaký problém či mít místo, kde lze testovat nové funkcionality. Organizace různých verzí je často velmi komplikovaná pokud na projektu pracuje jeden vývojář, ale při spolupráci celého týmu je ruční správa verzí téměř nemyslitelná.

Řešením tohoto problému jsou různé verzovací systémy, které tuto organizaci řeší za vývojáře. Dle [36, str. 7-10] a [5, str. 1-4] je lze rozdělit do třech hlavních kategorií podle toho, kde se software spravující verze nachází. První možností jsou systémy lokální, které běží na počítači vývojáře. Jejich výhodou je, že je lze používat bez přístupu k internetu, naopak nevýhodou je, že pokud by došlo k selhání počítače dojde k celkové ztrátě dat. Navíc neumožňují spolupráci více lidí na jednom projektu. Alternativou k nim jsou centralizované verzovací systémy, které běží na jednom serveru, kde veškerá správa probíhá. Toto řešení umožňuje spolupráci vývojářů, ovšem zachovává si nevýhodu jediného možného bodu selhání. Navíc je není možné používat bez připojení k internetu. Poslední kategorií jsou decentralizované verzvací systémy, mezi které spadá i Git. Ty běží na počítačích všech co se podílí na vývoji a tak nabízí velkou ochranu před ztrátou dat, neboť každý má kopii celé historie. Není problém je používat bez připojení k internetu, ovšem nabízí i možnost využívat sdíleného serveru, přes který dochází k synchronizaci změn mezi účastníky. Tento server se svou strukturou nijak neliší od běžného uživatele. Příkladem takového serveru je například GitHub.

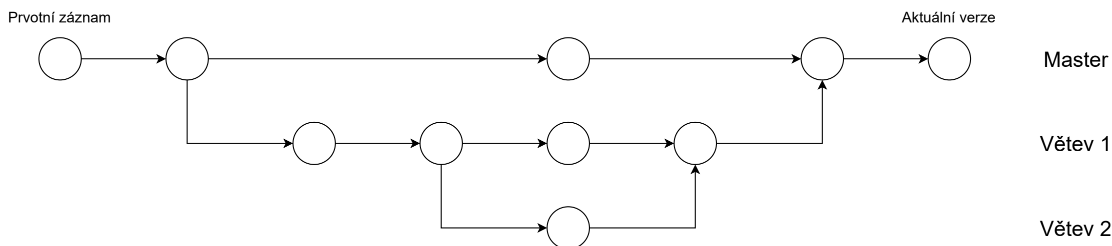
Git byl vytvořen Linusem Torvaldsem v roce 2016 [36, str. 10] jako open source alternativa k tehdy velmi rozšířenému BitKeeperu, který byl zpoplatněn. Jádrem je skrytá složka .git, ve kterém jsou uchovávány všechny informace. Tato složka vznikne při vytváření úložiště spravovaného Gitem a je unikátní pro každý projekt. Jak se lze dočíst v [5, str. 6-7], na rozdíl od většiny ostatních verzovacích systémů Git neukládá pouze změny, které nastaly oproti předchozí verzi, ale snímky aktuálního stavu. Díky tomuto rozdílu je rychlejší než jiné alternativy. Rozpoznávání změn probíhá na základě kontrolního součtu a pokud nedošlo ke změně tak soubor není ukládán znovu, ale je uložen pouze odkaz na něj.

Práce s Gitem se dá rozdělit do čtyř stupňů viz obrázek 2.10. Prvním stupněm je pracovní adresář, což je složka v níž se nachází .git spolu se soubory, na kterých je aktuálně pracováno. Pokud zde dojde ke změnám, které mají být zaznamenány, je nutné příkazem add přesunout tyto soubory do druhého stupně. Zde se nachází soubory, ze kterých bude vytvořen snímek pomocí příkazu commit a tím dojde k jeho uložení do lokální databáze. Z lokální databáze lze soubory sdílet pomocí příkazu push a naopak získávat změny, které byly nahrány jinými vývojáři pomocí pull. Návrat k předchozí verzi, či přesun do jiné větve je prováděn příkazem checkout, kdy dojde k načtení uloženého snímku z lokální databáze. Veškeré ovládání lze provádět z příkazové řádky, ale moderní IDE již s Gitem spolupracují a tak umožňují využít jejich uživatelského rozhraní.



Obrázek 2.10: Základní práce s Gitem

Důležitým prvkem práce s Gitem je větvení, které zaručuje jednoduchou spolupráci mezi vývojáři. Běžně je za hlavní větev považována větev Master, která se však od ostatních ničím neliší. Bývá dobrým zvykem v ní udržovat poslední funkční verzi programu. V rámci vývoje softwaru si vývojáři před implementací nové funkce vytvoří svou větev založenou na konkrétním snímku. V této větvi pak provádí potřebné úpravy a když jsou hotovy tak změny spojí s hlavní větví. Nové větve nemusí vznikat pouze z hlavní větve, ale mohou být vytvářeny z jakéhokoli snímku viz obrázek 2.11. Spojování větví probíhá automatiky, pokud nedošlo v obou větvích k úpravě stejné části kódu. V opačné případě musí vývojář tento konflikt vyřešit ručně.



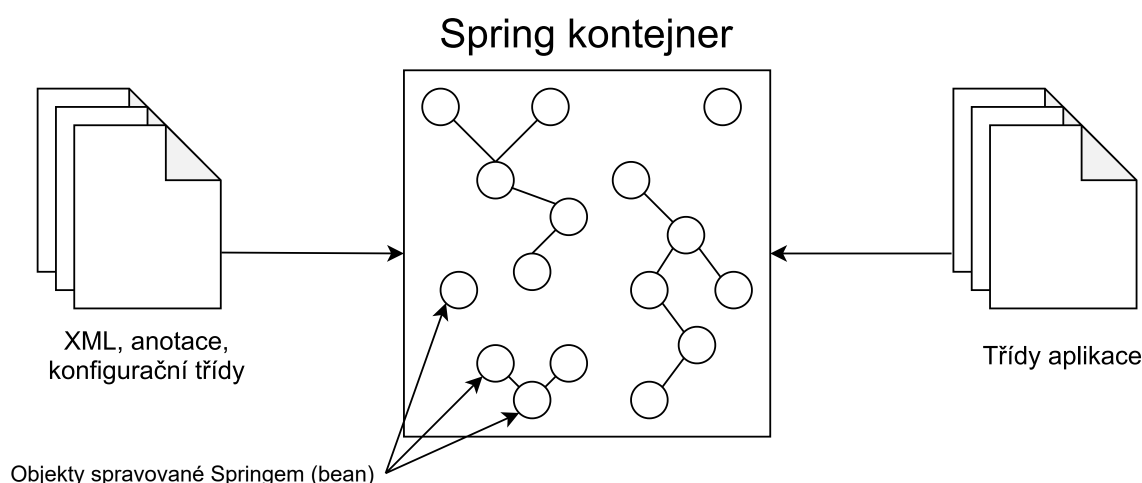
Obrázek 2.11: Větvení

2.7 Spring framework

Spring je open source framework vytvořený Rodem Johnsonem v roce 2002[7, str. 1], jako odpověď na komplikovanost tehdejší J2EE. Lze pomocí něj vytvářet libovolné aplikace, velké obliby však získal hlavně mezi aplikacemi webovými a v současnosti patří mezi jeden z nejznámějších frameworků v Javě. Pro jeho pochopení je důležité porozumět tomu co znamenají pojmy IoC a DI.

IoC je princip programování, při kterém průběh volání funkcí není řízen programem samotným, ale je volán z vnějšku. Toto je mimo jiné hlavní vlastnost, která odlišuje framework od knihovny, kdy knihovna je balíček funkcí, které programátor ve svém programu volá sám, kdežto framework volá kód napsaný programátorem. Dochází tak inverzi kontroly v porovnání s klasickým procedurálním programováním.

Návrhový vzor dependency injection (DI) poté udává to, že jednotlivé objekty by na sebe měly být vázány přes rozhraní a neměly by v sobě vytvářet konkrétní implementace. Tyto závislosti jsou do daného objektu vpravovány až pomocí konstruktoru či setterů. Třída tak není vázána na konkrétní implementaci, ale může být použita jakákoli implementace splňující definované rozhraní. Tímto je zajištěna lepší možnost testování kódu, neboť lze testovat funkčnost právě vytvářené třídy bez toho, aby bylo nutné mít hotový kód pro její závislosti. Při využití setterů lze také jednoduše měnit chování objektu za běhu programu. Jedná se tak o jednu z možností jak dosáhnout inverze kontroly (IoC). Použití tohoto návrhového vzoru vede ve výsledku k lepší rozšiřitelnosti, flexibilitě a celkově robustnější architektuře programu.



Obrázek 2.12: Spring kontejner

Jak se lze dočíst například v [29], jádrem Springu je Spring kontejner, který získává metadata z XML souborů, anotací, nebo z konfiguračních tříd a na základě těchto metadat vytváří objekty z konkrétních tříd, které v nich jsou uvedeny. Mezi objekty, které Spring kontejner spravuje, mohou existovat vztahy viz obrázek 2.12 a Spring tak musí zajistit jejich vytváření ve správném pořadí. Při inicializaci lze závislosti do jednotlivých objektů vpravovat buď pomocí konstruktoru, využitím setteru, či aplikováním reflexe. Poslední jmenovaný způsob není doporučovaný, neboť komplikuje psaní testů. Springem spravované objekty se nazývají bean a jsou znázorněny kroužky v obrázku 2.12. Přímý přístup k ním je v případě potřeby umožněn skrze rozhraní ApplicationContext.

Spring framework byl původně pouze jednoduchý IoC kontejner, který se ale do současnosti rozrostl o mnoho částí. Skládá se zhruba z 20 modulů jako jsou například Data Access/Integration, Web či Test [7, str. 21-23]. S rozrůstajícími možnostmi vzrostla i komplikovanost konfigurace, a tak bylo před během i jednoduché aplikace nutné provést spousty přípravných kroků, což bylo problematické hlavně pro nové uživatele.

2.7.1 Spring Boot

Konfigurační problém vyřešil až Spring Boot, který umožňuje vytvoření jednoduché aplikace během pár minut a bez jakýchkoli XML konfiguračních souborů. Nemusí však zůstat pouze u jednoduchých aplikací, Spring Boot využívají i velké firmy jako například Netflix [15].

Aplikace je konfigurována automaticky na základě použitých závislostí. Pro povolení této funkcionality je potřeba třídu obsahující statickou metodu main označit pomocí anotace `@EnableAutoConfiguration` či pomocí `@SpringBootApplication`, která je sama anotována prvně jmenovanou. `@SpringBootApplication` v sobě nese též anotaci `@ComponentScan`, která způsobí to, že Spring vyhledá všechny třídy označené anotací `@Component` a vytvoří z nich beany. Třídy nemusí být označeny přímo anotací `@Component`, ale jednou z anotací, která tuto anotaci nese v sobě. Těmi jsou například anotace `@Repository` používaná pro označování tříd zajišťujících přístup k databázi, `@Service` označující třídy nesoucí hlavní logiku programu, `@Controller` využívaná pro označení tříd sloužících ke zpracování požadavků z frontendu či `@Configuration` označující konfigurační třídy.

Aby vše bylo ještě jednodušší, lze použít tzv. Spring Starters, což jsou balíčky závislostí, které jsou nejčastěji používány pro daný typ úkolu. Existuje tak například spring-boot-starter-web, spring-boot-starter-test či spring-boot-starter-data-jpa, které obsahují závislosti vhodné pro vytváření webových aplikací, testování kódu či komunikaci s databází. Z těchto balíčků si lze automaticky vygenerovat základní strukturu aplikace a to pomocí Spring Initializr, který lze nalézt buď na internetových stránkách https://start.spring.io či je přímo integrován do IDE.

Takto vygenerovaný projekt obsahuje kromě souboru pom.xml definujícího zvolené závislosti také složku resources, ve které se nachází soubor application.properties, kde je možné nastavit jednotlivé parametry aplikace jako například port web serveru, na kterém se aplikace má spouštět či úroveň logování v aplikaci. Často je však potřeba mít různá nastavení pro vývoj a pro produkční aplikaci. Tohoto je dosaženo pomocí definování spring profilů, které poté využívají jim příslušných .properties souborů.

Další výhodou je, že Spring Boot nabízí vestavěný webový server, který umožní rychlé vytvoření spustitelného JAR souboru, což je výhodné při vývoji samostatných aplikací a jejich následné kontejnerizaci. Není totiž nutné stahovat webový server a aplikaci na něj instalovat jako WAR. Výchozím serverem je Tomcat, lze jej však v případě potřeby změnit například na Jetty.

2.8 Testování kódu

Tato část je věnována dvěma frameworkům a jedné knihovně, neboť jsou na sebe velmi vázány a často jsou používány spolu. Tvoří základ pro psaní jednoduchých, rychlých a přehledných testů.

2.8.1 JUnit

Skvělým nástrojem pro TDD je open source framework sloužící k psaní automatizovaných testů v programovacím jazyce Java JUnit. Jeho první verze byla vydána v roce 1997 [10, str.

8] a zasloužili se o ni vývojáři Kent Beck a Erich Gama. Od té doby se stala prakticky standardem pro testování Java aplikací. JUnit nabízí skvělou integraci se všemi populárními IDE a s nástroji jako je Maven. Na obrázku 2.13 je vyobrazeno rozhraní testů. Je zde přehledně ukázáno, jaké testy proběhly v pořádku a jaký čas zabraly.

<= TOOL SERVICE SPECIFICATION => 51 ms
UPDATE TOOL BY PLC ID AND TOOL ID 37 ms
throws ToolNotFoundException when tool with given ID was not found 23 ms
throws PlcNotFoundException when plc with given ID was not found 2 ms
PLC AND TOOL EXIST 12 ms
throws ToolUniqueConstrainException when tool number was changed to one which already exists 2 ms
throws ToolNumberUpdateException when tool number was updated for autodetected tool 1 ms
updates tool's attributes 9 ms
CREATE TOOL 5 ms
DELETE ONE TOOL FROM PLC BY PLC ID AND TOOL ID 3 ms
FIND ALL TOOLS 3 ms
FIND ALL TOOLS FROM ONE PLC BY PLC ID 3 ms

Obrázek 2.13: Integrace JUnit v IDE IntelliJ IDEA

Velkou roli v JUnit hrají anotace, na základě nichž jsou nastavovány všechny parametry testů. Mezi nejdůležitější patří anotace pro inicializaci testů jako `@BeforeAll` či `@BeforeEach`, jimiž označené metody proběhnou před spuštěním všech testů v dané třídě resp. před během každého z nich. Každá testovací metoda pak musí být označena `@Test` a již nezáleží na jejím pojmenování jak tomu bylo u dřívějších verzí. Při velkém množství testů je vhodně je seskupovat viz. obrázek 2.13. Toho je dosaženo vytvořením vnitřní třídy označené anotací `@Nested`. Poslední důležitou anotací je `@DisplayName`, která umožňuje nastavit zobrazované jméno testu.

2.8.2 Mockito

Při psaní testů může nastat situace, kdy ještě nejsou k dispozici některé třídy, které jsou k testování potřebné a je tedy nutné simulovat chování daného objektu. Právě toto umožňuje framework Mockito.

Nejdůležitějšími prvky při používání Mockita jsou statické metody `mock()`, `when()` a `thenReturn()`. První zmíněná metoda slouží k vytvoření simulovaného objektu. Kombinací metod `when()` a `thenReturn()` je po té možné určit, co se stane pokud bude na nově vytvořeném simulovaném objektu zavolána daná metoda. Alternativou ke statické funkci `mock()` pak může být použití anotace `@Mock`, které má obdobnou funkci. Například tedy pokud by bylo potřeba simulovat metodu `getZnamka()` třídy Student, bylo by toho dosaženo pomocí následujícího kódu.

```
@Mock
Student student;
when(student.getZnamka()).thenReturn(1);
```

Framework též umožňuje sledovat počty volání konkrétních metod na simulovaných objektech, což se může hodit v situaci, kdy je třeba ověřit zda konkrétní metoda byla v průběhu testu zavolána či nikoli. Pokud by bylo potřeba podobného výsledku dosáhnout s konkrétní instancí třídy, tak Mockito nabízí statickou metodu `spy()`.

2.8.3 AssertJ

Posledním dílkem skládačky při vytváření automatizovaných testu je knihovna AssertJ. Ta umožňuje zápis předpokládaných výsledků testů v jazyce co nejbližším tomu přirozenému a díky tomu je dosaženo lepší přehlednosti a čitelnosti testů. Nejdůležitější metodou této knihovny je statická metoda `assertThat()` v překladu do češtiny tedy „předpokládej že“.

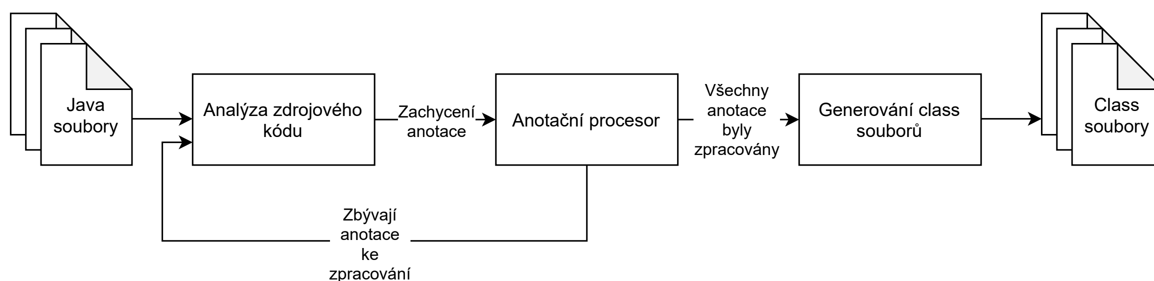
Pokud by tedy výsledkem testu mělo například být, že jméno daného studenta začíná na „Ja“ a končí „b“, dalo by se to pomocí AssertJ zapsat doslova jako „předpokládej, že jméno začíná Ja a končí b“. V kódu by toto mělo podobu následující.

```
assertThat(student.getName()).startsWith("Ja").endsWith("b");
```

Jak je patrné z předešlé ukázky, použití této knihovny je velice jednoduché a intuitivní. Nabízí velké množství metod a v kombinaci s jejich návrhy v IDE je tvorba testů velmi rychlá. Právě díky tomu si získala obliby velkého množství vývojářů.

2.9 Lombok

Programovacímu jazyku Java je často vytýkána jeho výřečnost. Neustále se opakující kód jako jsou například konstruktory, gettery, settery, nebo toString metody spojené téměř s každou novou třídou tomu jistě nepomáhají. Lombok je open source knihovna, která spolupracuje s IDE a nástroji jako je Maven a umožňuje tak automatickou generaci často opakujícího se kódu pomocí anotací. Třídy bez takovýchto částí jsou často přehlednější, což umožňuje vývojáři více se soustředit na jejich hlavní logiku.



Obrázek 2.14: Zpracování anotací

I když většina moderních IDE nabízí generaci kódu podobně jako Lombok, tyto části jsou přidány přímo do dané třídy. V [22] se lze dočíst, že Lombok využívá funkce zpracování anotací, která byla do Java kompilátoru přidána ve verzi 5. V průběhu kompilace dojde k

zachycení anotace, která je předána Lomboku, který již upravuje finální bytecode viz obrázek 2.14. Samotná třída, kde byl Lombok použit, tak zůstává prázdná.

Následující kód se tak postará o vytvoření getterů a setterů pro každý atribut (jmeno, prijmeni, znamka), přidá konstruktor obsahující zmiňované atributy a vygeneruje toString metodu vracející „Student(jmeno=Jakub, prijmeni=Znamenáček, znamka=1)“. Dále nastaví všechny atributy jako private.

```
@Setter
@Getter
@AllArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE)
@ToString
public class Student {
    String jmeno;
    String prijmeni;
    int znamka;
}
```

Jak je patrné, kód psaný ručně či generovaný pomocí IDE by se pravděpodobně nevešel ani na tuto stránku. Kromě tohoto Lombok též umožňuje pomocí jednoduché anotace `@Slf4j` vytvořit instanci loggeru v dané třídě, což opět umožní zaměření kódu na ty nejdůležitější věci.

2.10 MapStruct

Při vývoji API je často vhodné oddělovat vnitřní strukturu kódu od té, která je prezentována klientovi. Tato separace zajistí funkční API i při změnách v kódu a možnost redukovat množství požadavků zasláním více informací dohromady. Často jsou za tímto účelem využívány takzvané DTO. Jedná se o třídy, jejichž jediným účelem je přenášet data mezi klientem a serverem.

Psaní metod, které by zajistily konverzi z modelových tříd na DTO, je běžně rutinní a zdlouhavá práce. Mapování bývá mnohdy jednoduché a jména atributů v jedné třídě odpovídají atributům ve třídě druhé, pouze jsou nějaké vynechány. Situace je tak více než vhodná pro automatické generování kódu a zde přichází na řadu MapStruct. Jedná se o anotační procesor podobně jako Lombok, který je zaměřen na kopírování informací mezi třídami. MapStruct využívá čisté Javy co nejblíže tomu, jak by kód napsal vývojář, není zde tedy využita reflexe.

Jediné co je potřeba pro použití MapStructu udělat, je napsání mapovacího rozhraní označeného anotací `@Mapper` implementaci metod již zajistí MapStruct sám při kompilaci. Příkladem tak může být konverze modelové třídy Student na StudentDto kvůli tomů, aby nedošlo ke sdílení hesla. Pro zkrácení kódu je zde využito Lomboku, který se stará o generování všech potřebných věcí jako jsou například gettery a setter.

```
@Data
public class Student {
    String jmeno;
    String prijmeni;
    String heslo;
}


@Value @Builder
public class StudentDto {
    String jmeno;
    String prijmeni;
}


@Mapper
public interface DtoMapper {
    StudentDto toDto(Student student);
    Student fromDto(StudentDto studentDto);
}
```

V nějakých případech není pojmenování atributů stejné, či je potřebná ruční implementace. Pro všechny takovéto situace MapStruct nabízí vhodné anotace či je možné do daného rozhraní dopsat ručně defaultní metodu.

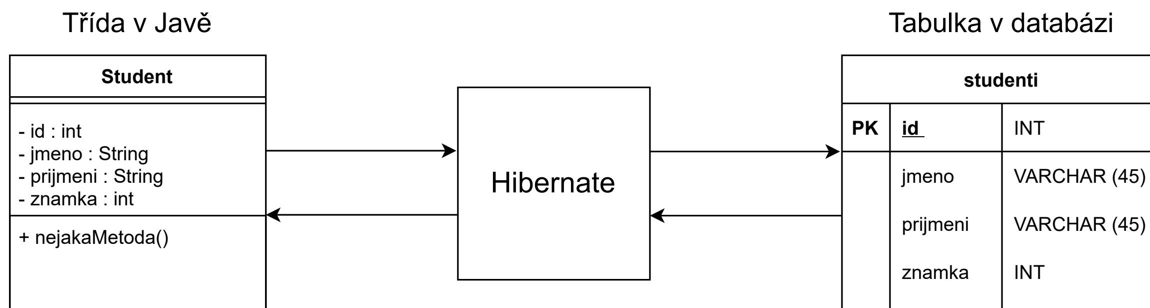
2.11 Práce s databází

Většina aplikací má potřebu ukládat svá data tak, aby i po restartování aplikace byla dostupná. Častým řešením bývají relační databáze, které ukládají informace do tabulek. K získávání a zápisu dat slouží SQL, bohužel SQL může mít různé dialekty v závislosti na zvoleném DBMS. V programovacím jazyce Java je pro přístup k databázi pomocí SQL využíváno JDBC.

Jelikož je Java objektově orientovaný programovací jazyk jsou zde s oblibou využívaný ORM frameworky. ORM je programovací technika zajišťující automatické mapování tříd a datových typů daného programovacího jazyka na tabulky a datové typy v databázi viz obrázek 2.15. Vedle prostého mapování řeší ORM též problém dědičnosti, která se typicky vyskytuje v objektově orientovaných programovacích jazycích a není podporována relačními databázemi.

2.11.1 Hibernate

Ve světě Javy je nerozšířenějším ORM frameworkem Hibernate. Jedná se o open source software vytvořený Gavinem Kingem v roce 2001[3, str. 14], který vývojáři umožní pracovat s databází bez toho, aby musel ručně psát SQL. To vede ke zvýšení produktivity a a usnadnění údržby kódu, neboť funkčně stejná část programu lze napsat pomocí méně řádků.



Obrázek 2.15: Hibernate ORM

Velkou výhodou oproti použití čistého JDBC je též nezávislost na konkretním DBMS. Toto je zajištěno díky tomu, že při práci s daty uloženými v databázi je vývojářem místo SQL využíváno vlastního objektově orientovaného dotazovacího jazyku HQL. Ten je po té Hibernatem automaticky převáděn na běžné SQL, pomocí nějž jsou dotazy zasílány. Je tak možné změnit DBMS bez nutnosti úpravy kódu.

I přes to, že se najdou situace, kdy může být využití JDBC a ruční napsání SQL dotazu rychlejší, neustálá optimalizace, kterou Hiberanate provádí ve výsledku vede k lepším celkovým rychlostem [4, str. 17]. Velkou zásluhu na tomto má ukládání naposledy získaných informací do mezipaměti, které snižuje počet dotazů na databázi a zrychluje tak odezvu.

Hibernate je v současné době více než pouhým ORM frameworkem a spadají pod něj mimo jiné i Hibernate Envers sloužící k auditování a udržování historie databáze, či Hibernate Validator. Poslední jmenovaný slouží k vytvoření požadavků, které má daná třída či její atribut splňovat. Pomocí těchto omezení, která jsou definována skrze anotace, lze například specifikovat, že daný atribut nesmí být nulový (`@NotNull`), či určit rozsah, kterého může nabývat (`@Size(min = 2, max = 50)`). Jejich kontrolu lze buď provádět ručně či automaticky před ukládáním do databáze.

2.11.2 JPA

Stejně jako se v objektově orientovaném programování preferuje programování do rozhraní za účelem zachování nezávislosti na konkrétní implementaci, tak i u používání různých ORM frameworků je výhodné využít abstrakce, aby je bylo možné v budoucnu v případě nutnosti jednoduše zaměnit. Za tímto účelem bylo v roce 2006 [17, str. 18] vytvořeno i JPA. Jedná se pouze o specifikaci, která sama o sobě nenese žádnou logiku. Tu musí dodat až příslušné implementace jako jsou Hibernate, Toplink, nebo EclipseLink. Zajímavé je si povšimnout, že JPA bylo vytvořeno až po Hibernate. Jelikož se jednalo o velmi populární framework byla spousta jeho vlastností převzata právě jako specifikace pro JPA. Platí tak například, že JPQL, objektově orientovaný dotazovací jazyk definovaný v JPA, je podtřídou HQL. Lze tak vše zapsané v JPQL použít jako HQL, to samé však neplatí naopak.

JPA, podobně jako většina moderních frameworků v Javě, využívá ve velkém anotací a výchozích nastavení. Pomocí nich lze tak definovat, že daná třída bude ukládána do databáze (`@Entity`), do jaké tabulky (`@Table`), jaké budou názvy sloupců odpovídajících jednotlivým

atributům (`@Column`), jaký atribut má být považován na primární klíč (`@Id`) či vztahy mezi jednotlivými třídami (`@OneToOne`,`@OneToMany`, `@ManyToMany`). Pokud nejsou například názvy sloupců či tabulky specifikovány, bere se jako výchozí možnost název založený na pojmenování atributu respektive třídy. Ukázka jednoduché třídy namapované do databáze je níže.

```
@Entity @Table(name = "studenti")
public class Student{
    @Id
    private Integer id;
    private String jmeno;
    private String prijmeni;
}
```

2.11.3 Spring Data JPA

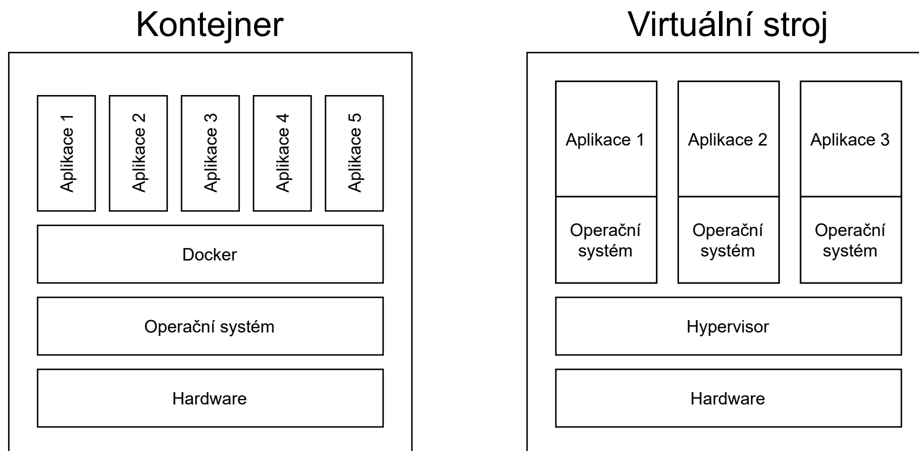
Spring Data JPA je část Spring frameworku, která staví na JPA a zjednodušuje práci hlavně v oblasti získávání dat z databáze. Spring Data JPA využívá návrhového vzoru repository, ovšem na rozdíl od jeho ručního psaní nabízí generická rozhraní, která stačí využít a jejich implementace bude vygenerována automaticky. Tato rozhraní nabízí základní CRUD operace. Pokud je potřeba přidat specifičtější metody, tak Spring Data JPA umožňuje automatické generování jejich implementace pouze na základě jejich názvu. Například lze tak přidat metodu `findStudentByJmeno(String jmeno)` tedy „najdi studenta na základě jména" a její implementace bude automaticky vygenerována. Pro komplikovanější dotazy, které by nešly na základě této konvence napsat či by bylo jméno metody moc dlouhé, lze použít anotaci `@Query` nad metodou libovolného jména a do ní vyplnit JPQL, či SQL dotaz přímo.

2.12 Docker

Docker vznikl v roce 2013 [28, str. 15], od té doby nabýval rychle na popularitě a stále více a více se o něj zajímaly velké firmy. Dokonce to došlo až tak daleko, že v roce 2016 [31] umožnil Microsoft tvorbu kontejnerů založených na operačním systému Windows. Do té doby byl totiž možný pouze běh kontejnerů založených na linuxu, díky čemuž na serveru s Windows musela běžet virtualizace linuxu, nad níž právě tyto kontejnery fungovaly. V současnosti se stal Docker prakticky standardem pro vytváření, práci a distribuci kontejnerů.

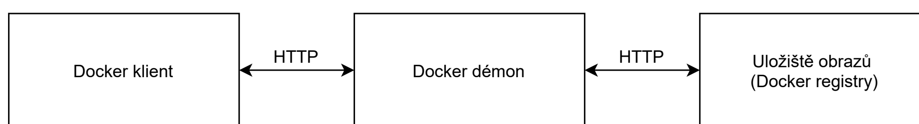
K porozumění tomu co je to Docker je důležité pochopení pojmů jako virtuální stroj (známý spíše pod anglickým názvem virtual machine), kontejner a jaký je mezi nimi rozdíl. Jak se lze dočíst v [28, str. 8-10], virtuální stroj umožňuje běh více virtualizovaných operačních systémů na jednom hardwaru, což dovoluje jednodušší distribuci programů, jejich izolaci od ostatních aplikací a nezávislost na konkrétním prostředí, kde chceme aplikaci spouštět. Nevýhodou daného řešení je naopak značná náročnost na hardwarové požadavky jak v podobě výkonu CPU či velikosti RAM, tak i velikost obsazeného místa na pevném disku, neboť každý běžící virtuální stroj si sebou nese i celý operační systém, který tyto zdroje spotřebovává. Tento problém řeší právě kontejnerizace aplikace, která si zachovává hlavní výhody virtuálních strojů, ale odstraňuje nutnost běhu jednoho operačního systému

pro jednu aplikaci viz obrázek 2.16. Díky tomu, že kontejnery sdílí jeden operační systém, tak mohou být menší a méně náročné což ve výsledku vede i k jejich rychlejšímu spouštění oproti virtuálním strojům.



Obrázek 2.16: Porovnání kontejnerů a virtuálních strojů

Docker je open-source software, který umožnil jednoduché vytváření, standardizaci, distribuci kontejnerů a jejich běh. Jak je uváděno v [25, str. 20-23], typická architektura Dockeru je složená z klienta, démona a úložiště obrazů jednotlivých kontejnerů, které mezi sebou komunikují přes REST API pomocí protokolu HTTP viz obrázek 2.17 a to jak lokálně tak po internetu. Nejdůležitější částí je démon, který běží na pozadí a stará se o jednotlivé operace s kontejnery, obrazy, sítěmi a podobně. Pro jeho obsluhu je klíčový klient. Jedná se o konzolovou aplikaci, která pomocí příkazů umožňuje ovládat démona. Úložiště obrazů (Docker registry) po té slouží k ukládání a šíření obrazů. Jeho typickým příkladem je Docker Hub sloužící jako veřejné úložiště všemožných obrazů a lze zde najít obrazy softwarů jako jsou například PostgreSQL, Ubuntu či OpenJDK.



Obrázek 2.17: Architektura Dockeru

Dockerové obrazy si lze představovat jako třídy v objektově orientovaném programování. Jedná se tedy o jakýsi vzor, na základě nějž vytváříme jednotlivé instance, v tomto případě kontejnerizované aplikace. Obrazy jsou tvořeny jednotlivými vrstvami, které vznikají nabalováním specifičtějších závislostí pro konkrétní aplikace. Toto umožňuje menší obsazení paměti neboť jednotlivé vrstvy obsahují pouze změny oproti výchozímu obrazu viz obrázek 2.18 a stejné spodní vrstvy lze sdílet mezi rozdílnými obrazy. Více informací lze dohledat v [28, str. 74-103].

Součástí Dockeru je i Docker Compose. Jedná se o nástroj na vytváření složitějších apli-

n-tá vrstva	Spuštění aplikace	0 MB
⋮	⋮	⋮
2. vrstva	Zkopírování aplikace	30 MB
1. vrstva	Výchozí obraz	200 MB

Obrázek 2.18: Dockerový obraz

kací složených z více kontejnerů na základě YAML souboru. Původně se jednalo o Pythonovský nástroj vytvořený společností Orchid nazvaný Fig využívající API Dockeru. Díky jeho popularitě se společnost stojící za Dockerem v roce 2014 [28, str. 157] rozhodla jej skoupit.

Kapitola 3

Implementace

3.1 Testování kódu

Vývoj backendu byl založen na přístupu programování řízené testy (TDD), nejprve tedy byly psány testy, které reprezentovaly požadované chování dané třídy a až poté byla implementována potřebná logika. Veškeré testy se nachází v balíčku test, jehož struktura je téměř stejná jako struktura samotného kód nacházejícího se v balíčku main. Každé testované třídě přísluší třída specifikující její vlastnosti. Název třídy s testy je shodný s názvem testované třídy pouze má příponu Spec.

Kvůli lepší organizaci jsou testy shlukovány podle daných tříd, metod či požadovaných vlastností. Skupina označená symboly <==> představuje kompletní balíček všech testů týkajících se jedné třídy. Podkategorie jsou pak psány velkými písmeny a samotné testy následně písmeny malými. Tato hierarchie je znázorněna na obrázku 3.1 pro testy třídy ToolController.

Test	Čas
▼ ✔ <= TOOL CONTROLLER SPECIFICATION =>	302 ms
▼ ✔ UPDATE TOOL BY PLC ID AND TOOL ID	199 ms
▼ ✔ allows CORS from all origins	50 ms
✔ [1] origin=http://localhost:4200	44 ms
✔ [2] origin=http://localhost:4201	3 ms
✔ [3] origin=http://random-page.com	3 ms
▶ ✔ INVALID TOOL DTO	115 ms
▶ ✔ VALID TOOL DTO	34 ms
▶ ✔ CREATE TOOL BY PLC ID	42 ms
▶ ✔ DELETE TOOL BY PLC ID AND TOOL ID	28 ms
▶ ✔ FIND ALL TOOLS FROM ONE PLC	18 ms
▶ ✔ FIND ALL TOOLS	15 ms

Obrázek 3.1: Organizace testů

V rámci celé aplikace bylo vytvořeno přes 400 testů, které při každé kompilaci kontrolují její funkčnost. Celkové porytí jednotlivých tříd je přes 80%. Podrobnější rozložení testů je znázorněno na obrázku 3.2, kde první sloupec představuje procentuální pokrytí tříd balíčku, druhý pokrytí jednotlivých metoda a třetí pak pokrytí samotných řádků kódu. Ze statistiky je patrné, že nejméně jsou testovány balíčky zajištující komunikaci, neboť byly zajišťovány externími knihovnami.

annotations	100% (0/0)	100% (0/0)	100% (0/0)
connection	44% (8/18)	39% (45/114)	39% (135/339)
controller	100% (3/3)	100% (16/16)	100% (26/26)
domain	100% (5/5)	85% (18/21)	94% (55/58)
dto	83% (44/53)	93% (137/146)	87% (312/358)
entity	100% (21/21)	96% (115/119)	89% (284/318)
enumerated	100% (3/3)	100% (6/6)	100% (11/11)
exception	84% (16/19)	72% (26/36)	70% (61/87)
repository	100% (0/0)	100% (0/0)	100% (0/0)
service	100% (7/7)	100% (36/36)	95% (225/235)
validators	100% (1/1)	100% (3/3)	64% (11/17)
webSocket	0% (0/1)	0% (0/3)	0% (0/6)
ApiConfiguration	100% (1/1)	100% (2/2)	100% (10/10)
ApplicationStartupManager	100% (1/1)	100% (3/3)	100% (13/13)

Obrázek 3.2: Pokrytí testy

Při testování metod, kde může dojít k selhání na více místech, jsou pro rychlejší lokalizaci chyby psány popisy, které se zobrazí s výsledky testů. Toto je velmi výhodné například u mapovacích metod, kde je u neprocházejícího testu zobrazen atribut, který nebyl správně zkopírován. Je tak snadné tento problém rychle dohledat a odstranit.

Kvůli jednoduchému psaní testů je většina tříd vytvářena jako implementace konkrétních rozhraní. Toto usnadnilo jak samotné psaní kódu předtím, než existovala třída nesoucí logiku, tak vytváření simulace tříd pomocí frameworku Mockito, na kterých testovaná třída závisela, ale v danou chvíli ještě nebyly k dispozici.

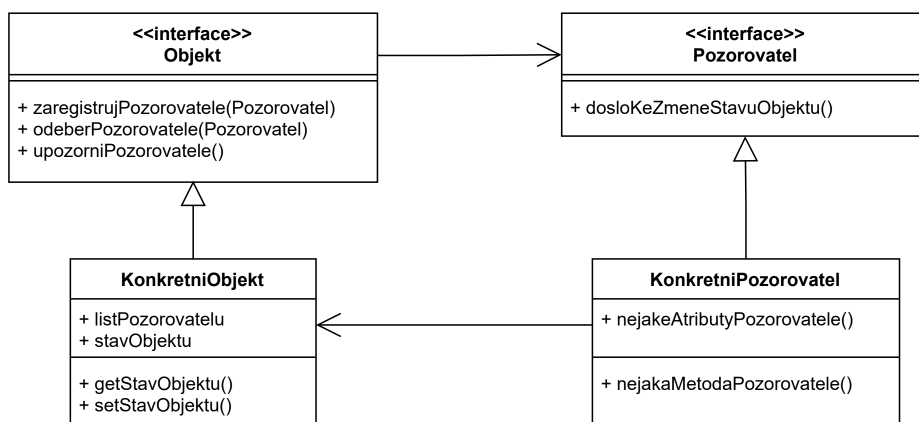
3.2 Základní architektura backendu

Backend je založen na frameworku Spring Boot, takže je ve velké míře využíváno tvorby beanů, které jsou pak při jeho startu automaticky vytvářeny. Též je použit vestavěný webový server Tomcat, jehož port je v application.properties nastaven na hodnotu 3000. Po kompilaci kódu a následném spuštění je tedy na tomto portu možné na backend zasílat HTTP dotazy prostřednictvím API. Kvůli odlišnostem při vývoji a při běhu v produkčním prostředí byly vytvořeny dva profily a to dev a prod. Rozdíl mezi nimi je hlavně v nastavení databáze, kdy v profilu dev dochází při každém startu aplikace ke smazání uložených dat, kdežto v reálném provozu musí být data samozřejmě zachována. Tyto profily se taktéž liší úrovní nastaveného logování, která je pro produkční verzi nastavená na úroveň info, kdežto ve verzi určené k vývoji je nastaveno podrobnější debug.

Architektura aplikace by se dala rozdělit na tři základní části. První z nich je část controllerů. Všechny třídy, které sem spadají jsou zařazeny do stejnojmenného balíčku. Tato vrstva slouží ke zpracovávání požadavků z frontendu, ověření jejich struktury a zasílání správných odpovědí popřípadě chybových hlášek. Formuje se zde tedy struktura API, přes které je aplikace ovládána. Další důležitou částí je servisní vrstva nacházející se v balíčku service. Ta v sobě nese všechnu důležitou logiku, jako je například vytváření spojení s PLC, kontrola platnosti zasílaných dotazů, převod tříd aplikace na DTO, které jsou v rámci API zasílány a tak dále. Poslední vrstvou je datová vrstva, která se nachází rozděleně ve dvou balíčcích a

to v entity a repository. V prvním zmiňovaném jsou třídy reprezentující tabulky v databázi, druhý balíček pak obsahuje třídy, které se starají o ukládání a získávání dat z databáze.

Za vyzdvižení stojí implementace připojení k PLC, která využívá návrhového vzoru pozorovatel, který je zobrazen na obrázku 3.3. Tento návrhový vzor je využíván pokud je potřeba zajistit reakci pozorovatele na změnu objektu bez toho, aby docházelo ze strany pozorovatele k pravidelným dotazům na stav objektu. Jeho základním principem je, že se pozorovatel u objektu zaregistruje k odběru informací o tom, že se objekt změnil. Jakmile dojde ke změně stavu objektu, je zavolána příslušná metoda pozorovatele. Jeden objekt může být najednou pozorován více pozorovateli, ti jsou pak ukládáni do listu pozorovatelů a všichni jsou na změnu stavu upozorněni. Pokud pozorovatel již nemá zájem být informován o změnách objektu, požádá jen objekt aby jej smazal se seznamu pozorovatelů.



Obrázek 3.3: Návrhový vzor pozorovatel

V aplikaci jsou vytvořena rozhraní pro jednotlivé informace z PLC, u kterých je důležité si udržovat jejich aktuální hodnotu. Příkladem může být změna nástroje (ToolDataSource), změna stavu připojení (ConnectionStatusSource) a podobně. Tato rozhraní jsou implementována abstraktní třídou PlcData, která v sobě nese veškerou logiku týkající se návrhového vzoru pozorovatel. Jsou v ní tak definovány jednotlivé metody umožňující přihlášení konkrétních pozorovatelů k odběru odpovídajících dat, či metody pro zrušení odběru. Třída PlcData navíc umožňuje též zrušení odběru pro všechny pozorovatele, čehož je využíváno pokud má dojít ke smazání spojení s PLC. Tato třída je abstraktní, neboť v sobě nenese informaci o konkrétním typu připojení. Takovéto řešení přináší zásadní výhodu pokud by mělo přijít na to, že by došlo ke změně či přidání protokolu, přes který aplikace s PLC komunikuje. V takovém případě totiž stačí přidat jen konkrétní implementaci třídy PlcData a na samotné struktuře aplikace se nic nezmění, neboť všechen kód závisí právě na abstraktní třídě a nikoli na třídách konkrétních.

3.3 API

API aplikace je rozděleno na dvě kategorie, první je REST, která je založena na HTTP dotazech, které jsou zasílány na server, jenž na ně zasílá příslušné odpovědi. Prostřednictvím

této části probíhá klasické ovládání aplikace, zajištující základní činnosti. Druhá část je založena na protokolu WebSocket v kombinaci se subprotokolem STOMP. Ta slouží k předávání informací v reálném čase bez toho, aby muselo dojít k zaslání dotazu na server. Tato část není nezbytná pro ovládání aplikace, pouze jen zpříjemňuje její používání. Data jsou pro obě možnosti zasílána ve formátu JSON, který je dnes prakticky standardem.

Za účelem oddělení vnitřní struktury backendu od struktury API jsou využívány DTO třídy. Instance těchto tříd vzniknou namapováním potřebných dat a jsou po vytvoření neměnné, neboť představují snímek informací v daný okamžik. K jejich lepšímu vytváření je použit návrhový vzor builder, který umožní nastavit parametry jednotlivě a jakmile je vše správně zvoleno vytvořit neměnnou instanci dané třídy. Veškeré DTO se nachází ve stejnojmenném balíčku a pro lepší organizaci byly sdruženy pomocí vnitřních tříd a vytváří tak přehledné jmenné prostory.

V případě, že je na server zaslán špatný dotaz, či dojde k nějakému problému je klientovi kromě odpovídajícího stavového kódu zasílán též JSON představující sjednocený formát pro chybová hlášení API. Ta obsahují status kód, zkrácenou zprávu, detail popisující problém a případně seznam všech chyb, které nastaly. Tato hlášení jsou reprezentována třídou ApiException a její instance jsou vytvářeny tehdy, pokud dojde k zachycení vnitřního chybového hlášení backendu, které má být zasláno klientovi.

3.3.1 REST

Logs
GET /api/logs
DELETE /api/logs
GET /api/logs/{id}
PUT /api/logs/{id}

PLCs
GET /api/plcs
POST /api/plcs
PUT /api/plcs/{plcId}
DELETE /api/plcs/{plcId}

Tools
GET /api/plcs/{plcId}/tools
POST /api/plcs/{plcId}/tools
GET /api/plcs/tools
PUT /api/plcs/{plcId}/tools/{toolId}
DELETE /api/plcs/{plcId}/tools/{toolId}

Obrázek 3.4: Přehled REST API

Struktura REST API byla navrhována tak, aby odpovídala používaným zvyklostem [30]

a byla logická. Pro všechny URL platí, že za adresou serveru následuje předpona api. Při přístupu k více položkám je využíváno množného čísla, tedy pro získání všech PLC stačí zaslat požadavek na /api/plcs. Pokud je vyžadována akce na konkrétním objektu následuje v adrese jeho identifikátor (ID). Pro ještě jednodušší práci s API je využito frameworku Swagger, který vygeneruje přehled celého API, který je následně zpřístupněn na adrese /api/docs-ui. Tento přehled je znázorněn na obrázku 3.4.

3.3.1.1 Správa PLC

Získání všech PLC uložených v databázi Při zaslání GET dotazu na adresu /api/logs dojde k zaslání odpovědi se stavovým kódem 200 OK a všemi PLC, které jsou obsaženy v databázi. Jako tělo zprávy je zasílán list PlcDto.Response.Overview, jehož formát je zobrazen níže. Obsahuje id, pomocí nějž je možné PLC jednoznačně identifikovat, IP adresu na které je PLC dostupné, jméno, informace o hardwaru PLC a informace o stavu připojení včetně data, kdy došlo k poslední změně.

```
[
    {
        "id": 14,
        "name": "PlcSim",
        "ipAddress": "192.168.0.84",
        "hardwareInformation": {
            "serialNumber": "10S C-92z452z442",
            "firmwareNumber": "V02.05.00"
        },
        "connection": {
            "lastStatusChange": "2021-03-25T12:58:29.926+00:00",
            "status": "CONNECTED"
        }
    }
]
```

Vytvoření nového PLC Pro vytvoření nového PLC je třeba zaslat PlcDto.Request.Create, jehož struktura je naznačena dále, na adresu /api/plcs. Získání odpovědi může trvat až 5 sekund neboť je čekáno, než vyprší čas, za který bude PLC označeno za nepřipojené.

```
{
    "name": "plcName",
    "ipAddress": "192.168.0.84"
}
```

Pokud vše proběhne v pořádku je navrácen stavový kód 201 Created a jako tělo je zasláno zpět vytvořené plc ve formátu jaký je používán při získávání všech PLC z databáze. Pokud je však zaslán požadavek pro vytvoření PLC se stejným jménem či IP adresou, která se schoduje s již existujícím PLC, odpoví server status kódem 409 Conflict a zašle zpět

příslušnou chybovou zprávu vysvětlující, kde došlo k problému. Obdobně při zaslání IP adresy v chybném formátu či nevyplněného jména. V takovém případě je však zasílán status kód 400 Bad Request.

Aktualizace informací o již existujícím PLC Pro aktualizaci informací o PLC je na adresu /api/plc/1, kde 1 představuje id PLC, které má být upraveno, zaslán stejný formát dat jako při vytváření. Rozdílem je použitá metoda, kdy místo POST je využito metody PUT. Opět proběhne kontrola platnosti zaslaných dat. Navíc zde nastává situace, kdy je zadán špatný identifikátor PLC, v takovém případě je zasílán zpět status kód 404 Not Found s příslušnou zprávou vysvětlující problém. Pokud vše proběhne v pořádku je zasílán status kód 200 OK s aktualizovanými informacemi opět ve stejném formátu, jako tomu bylo u vytváření.

Smazání existujícího PLC Smazání PLC je nejjednodušší operací, mohou totiž nastat pouze dvě situace, buď je plc s daným id v databázi nalezeno, pak dojde k jeho smazání a je zaslán status kód 200 OK, nebo není nalezeno a v takovém případě je obdobně jako u pokusu o aktualizaci neexistujícího PLC zasílán zpět status kód 404 Not Found.

3.3.1.2 Správa nástrojů

Struktura API pro správu nástrojů je velmi podobná části pro správu PLC. Opět je zde možné získat všechny nástroje z databáze, vytvářet je, upravovat či mazat. Navíc přibyla pouze možnost získání nástrojů pro konkrétní PLC.

Získání všech nástrojů Zasláním požadavku GET na adresu /api/plcs/tools je možné získat list všech nástrojů, které se nachází v databázi. Tento list je zasílán v následujícím formátu.

```
[
    {
        "id": 18,
        "plcId": 14,
        "toolNumber": 3,
        "name": "Tool3MACHINE20spm",
        "numberOfReferenceCycles": 7,
        "calculateReferenceCurve": false,
        "tolerance": {
            "type": "ABSOLUTE",
            "torqueTolerance": 10.0,
            "speedTolerance": 5.0
        },
        "stopReaction": "DO_NOTHING",
        "referenceCurveIsCalculated": true,
        "automaticMonitoring": false,
        "toolStatus": "AUTODETECTED",
        "isActive": true
    }
]
```


Jsou zde obsaženy všechny podstatné informace, jako id nástroje, id PLC, kterému nástroj náleží, číslo nástroje, jméno nástroje, počet cyklů potřebných k výpočtu referenční křivky, to zda je kalkulace referenční křivky pro daný nástroj spuštěna, tolerance, typ zpětné vazby pro PLC v případě nesplnění tolerancí, to zda je referenční křivka vypočítána, to zda je zapnuta automatická kontrola lisovacího cyklu, stav nástroje určující, zda byl přidán uživatelem, či rozpoznán aplikací a to, jestli je aktuálně používán.

Získání všech nástrojů pro jedno PLC Pro získání pouze nástrojů pro konkrétní plc stačí v rámci adresy specifikovat id PLC, například takto /api/plcs/1/tools. Pokud PLC s tímto identifikátorem existuje jsou zaslány zpět jeho nástroje ve stejném formátu jako dříve. Pokud takovéto PLC neexistuje je zaslán status kód 404 Not Found společně s příslušnou chybovou hláškou, která uživateli vysvětlí, kde nastala chyba.

Vytvoření nového nástroje Při zaslání nového nástroje v následujícím formátu na adresu /api/plcs/1/tools dojde k přidání nástroje pro PLC s identifikátorem 1.

```
{
    "toolNumber":"12",
    "name":"toolName",
    "numberOfReferenceCycles":"10",
    "tolerance": {
        "type":"ABSOLUTE",
        "torqueTolerance": 10,
        "speedTolerance":10
    },
    "stopReaction":"DO_NOTHING",
    "automaticMonitoring":"false",
    "calculateReferenceCurve":"false"
}
```


Jako odpověď je v případě bezchybného průběhu zasílán status kód 201 Created s aktuálními informacemi ve stejném formátu, který je zasílán při získávání nástrojů. Pokud však již existuje nástroj se stejným číslem je vracen status kód 409 Conflict a tělo obsahující zprávu, která vysvětluje problém. Když dojde k zaslání nástroje ve špatném formátu, což může nastat pokud chybí číslo nástroje, počet požadovaných cyklů pro výpočet referenční křivky neleží v intervalu 1 až 100, či hodnoty pro toleranci nejsou platné, zasílá server 400 Bad Request s konkrétní zprávou, kde došlo k problému, aby jej uživatel mohl odstranit.

Aktualizace informací o již existujícím nástroji Obdobně jako u části API pro správu PLC je i zde adresa pro aktualizaci stejná jako adresa pro vytvoření nástroje, pouze je rozšířena o identifikátor konkrétního nástroje. Má tedy tvar /api/plcs/1/tools/1 a jsou na ni zasílána data ve stejném formátu jako při tvorbě. Nová data opět projdou stejnou validací. Navíc zde je pouze kontrola toho, že u automaticky detekovaného nástroje není upravováno jeho číslo. Pokud by k takovému pokusu došlo, je zasílán klientovi status kód 403 Forbidden se zprávou vysvětlující proč došlo k zamítnutí požadavku.

Též zde jako u správy PLC může nastat situace, kdy je zaslán špatný identifikátor nástroje či PLC. Jestliže k tomuto případu dojde, klient obdrží status kód 404 Not Found s vysvětlením, zda v databázi nebylo nalezeno požadované PLC či nástroj.

Smazání existujícího nástroje Cesta pro smazání nástroje je stejná jako cesta pro jeho úpravu, pouze je nutné použít metodu DELETE. Shodné je i řešení neplatnosti identifikátorů pro PLC či nástroj. Navíc zde však může nastat situace, že dojde k požadavku na smazání právě používaného nástroje. Tato akce není povolena a proto je jako odpověď na tento požadavek zasílán status kód 403 Forbidden se zprávou vysvětlující proč tuto akci nelze provést.

3.3.1.3 Správa logů

Samotné API nenabízí možnost vytvořit nové logy, ty jsou totiž vytvářeny pouze backendem. Možnost jejich úpravy je též velmi omezená. Lze u nich měnit pouze komentář a nikoli samotné informace, které log nese.

Získání všech logů pro konkrétní nástroj Adresa pro získání logů daného nastroje je /api/logs a id nástroje je předáváno jako parametr dotazu tedy ?tool-id=1. Není zde tudíž zadáváno ID v cestě, neboť se nejedná o ID samotného logu, ale dochází k požadavku na filtrování všech logů a zaslání pouze těch, které se vážou k nástroji se zasílaným identifikátorem. Odpovědí na takovýto požadavek je seznam logů, které mají podobu DTO LogDto.Response.Overview. Jehož struktura je zobrazena dále.


```
[
    {
        "id": 128,
        "createdOn": "2021-03-31T09:04:39.322+00:00",
        "numberOfCollisions": 12,
        "plcInformation": {
            "name": "PlcSim",
            "ipAddress": "192.168.0.84",
            "serialNumber": "10S C-92z452z442",
            "firmwareNumber": "V02.05.00"
        },
        "toolInformation": {
            "toolId": 1,
            "toolNumber": 0,
            "name": "Tool0MACHINE44spm",
            "stopReaction": "DO_NOTHING",
            "tolerance": {
                "type": "ABSOLUTE",
                "torqueTolerance": 10.0,
                "speedTolerance": 11.0
            }
        }
    }
]
```


Jelikož může být jako odpověď na tento požadavek zasíláno velké množství logů, jsou posílány pouze důležité informace. Mezi ty patří identifikátor logu, čas kdy byl log vytvořen, počet bodů naměřené křivky, které se nacházely mimo toleranci, informace o PLC a informace o daném nástroji.

Získání detailních informací o konkrétním logu Jak již bylo zmíněno, s dotazem na získaní více logů se zasílají pouze ty nejdůležitější informace. Pokud uživatel potřebuje získat všechna dostupná data může tak učinit zasláním dotazu na adresu /api/logs/1, kde 1 představuje identifikátor konkrétního logu. Pokud vše proběhne bez komplikací, je mu zaslán v těle odpovědi následující JSON.

```
{
    "id": 1,
    "createdOn": "2021-03-31T09:04:39.322+00:00",
    "measuredCurve": {
        "points": [ ... ]
    },
    "motorCurve": {
        "points": [ ... ]
    },
    "referenceCurve": {
        "points": [ ... ]
    },
    "collisionPoints": [ ... ],
    "plcInformation": { ... },
    "toolInformation": { ... }
}
```

Kvůli přehlednosti byly vynechány informace o PLC a nástroji, které se shodují s informacemi zasílanými při získávání více logů najednou. Dále byly vynechány též body, reprezentující naměřenou křivku, křivku představující omezení rozsahu motoru, křivku referenční a body nacházející se mimo toleranci.

Smazání logu Při mazání logů je předpokládáno, že může probíhat po větším množství. Z toho důvodu je na adresu /api/logs zasíláno tělo obsahující list identifikátorů těch logů, které mají být odstraněny. Všechny logy, které jsou v databázi nalezeny pod danými identifikátory jsou smazány a pokud nějaké nebyly nalezeny je jejich seznam zaslán zpět v těle zprávy se status kódem 404 Not Found.

3.3.2 WebSocket

Připojení přes WebSocket probíhá na adrese /api/ws a následně přihlášením k pěti možným tématům. Přes tento protokol jsou zasílány informace, které se mohou měnit a není možné předpovědět, kdy ke změně dojde. Mezi taková data patří například aktuální stav připojení PLC, změna nástroje, detekce nového nástroje, stav spuštěného výpočtu referenční křivky či vytvoření nového logu. Veškeré tyto informace lze získat i zasláním HTTP dotazu, ovšem udržet si informace neustále aktuální by vyžadovalo cyklické zasílání dotazů.

3.3.2.1 Změna stavu připojení PLC

Pokud se klient přihlásí k odběru informací o aktuálním stavu připojení PLC na tématu /topic/plcs/connection-status, tak pokaždé, kdy dojde ke změně stavu připojení, mu bude zaslán identifikátor PLC, aktuální stav připojení a kdy ke změně došlo. Předpona topic je běžně používána při využívání subprotokolu STOMP se Spring Bootem viz [1, str. 65].

3.3.2.2 Změna používaného nástroje

Pro přehled o tom, jaký nástroj je aktuálně používán je zapotřebí přihlásit se k odběru tématu /topic/plcs/current-tool. Zde klient při změně nástroje získává identifikátor PLC, na kterém ke změně došlo, a číslo aktuálně používaného nástroje. Tato informace je důležitá například při výpočtu referenční křivky, kdy při změně nástroje dojde k přerušení výpočtu.

3.3.2.3 Detekce nového nástroje

Uživatel má možnosti si přidat nové nástroje ručně pomocí HTTP požadavku, jak bylo zmiňováno dříve, pokud je však detekován nástroj, který nebyl uživatelem přidán, tak dojde k jeho automatickému uložení. Nový nástroj by se tak objevil až tehdy, jakmile by došlo k novému načtení všech nástrojů. Aby však bylo možné získat tuto informaci co nejdříve, je třeba se přihlásit k odběru tématu /topic/plcs/new-tool, kde je klientovi zasílán nový nástroj ve stejném tvaru, jako by tomu bylo u HTTP požadavku.

3.3.2.4 Stav výpočtu referenční křivky

Velmi důležitou informací může být stav výpočtu referenční křivky a to hlavně, pokud se má křivka skládat z velkého počtu cyklů. Jakmile by totiž uživatel spustil výpočet, nevěděl by za jak dlouho dojde k jeho dokončení. Pokud se ovšem před spuštěním přihlásí k tématu /topic/tools/calculation-status, tak je mu s každým novým cyklem zasílána informace o kolikátý cyklus se jedná z daného počtu požadovaných cyklů.

Jednoduše se dá i poznat, jestli nedošlo k nějaké poruše (postup by se zastavil) a uživatel by pak mohl pomocí klasického HTTP požadavku výpočet zrušit. Taktéž lze z této informace dobře odhadnout hrubý čas výpočtu referenční křivky, a pokud by byl příliš dlouhý, výpočet zastavit a spustit jej znovu s nižším počtem požadovaných cyklů.

3.3.2.5 Vytvoření nového logu

Poslední téma, které je k dispozici je /topic/logs/new-log, které slouží k zasílání nově vytvořených logů. Jelikož jsou logy generovány automaticky backendem při nesrovnalosti mezi naměřenou a referenční křivkou, není možné odhadnout, kdy budou vytvořeny a proto je pro jejich co nejrychlejší detekci nutné využít protokolu WebSocket.

3.3.3 Zabezpečení komunikace

Komunikace probíhá po nezabezpečených verzích protokolů HTTP a WebSocket. Nedochází zde k zasílání uživatelských jmen či hesel, ale data z produkce mohou být stejně důležitá a jejich únik by mohl způsobit velké problémy. Důvodem tedy není to, že by nebyla posílána citlivá data, ale to, že zabezpečení poskytuje samotná platforma, ve které má aplikace běžet.

Pro interakci s aplikací je tak nutné přihlásit se do systému Industrial Edge a na základě tohoto je uživateli umožněn přístup ke všem jeho nainstalovaným aplikacím. Veškerá komunikace mezi jednotlivými částmi tak může probíhat nezabezpečeně a vše co jde mimo tento systém je zasíláno zabezpečenou cestou pomocí protokolů HTTPS a WebSocket Secure. Tento přístup má výhodu v tom, že není potřeba zajišťovat a obnovovat certifikáty pro jednotlivé aplikace a zároveň implementovat autentizace pro každou aplikaci zvlášť.

3.4 Aplikační vrstva

3.4.1 Zpracování dotazů zaslaných prostřednictvím API

Jakmile má zasílaný dotaz správný formát (prošel přes vrstvu kontrolérů), tak je poté předáván servisní vrstvě. Jmenovitě se jedná o třídy PlcService, ToolService a LogService, které zařizují veškerou logiku týkající se splnění obdrženého požadavku. Dále v nich probíhá převod mezi zasílanými DTO a vlastními třídami aplikace, s kterými je dále zacházeno. Tyto části servisní vrstvy jsou spravovány Spring frameworkem, jenž vytváří jejich instance při startu aplikace a ty poté existují pouze jednou. Aby mohly vykonávat svou činnost, musí mít k dispozici příslušné objekty zajištující komunikaci s databází a mapování na DTO. Ty jsou též spravovány Springem a jelikož bez nich tato část servisní vrstvy nemůže korektně fungovat jsou při vytváření jednotlivých instancí vpravovány jako závislosti v konstruktoru.

I když dotaz projde přes kontrolu formátu, mohou nastat konflikty, které se pouze ze zasílaných dat nedají odhalit. Jedním z takových je například existence stejnojmenného PLC v databázi při vytváření nového. V takovýchto případech je zapotřebí vytvořit speciální výjimku, která je zachycena třídou GlobalExceptionHandler. V ní dochází ke konverzi zachycené výjimky na formát ApiException, který je pak zasílán zpět klientovi. Důležitá je i tvorba jednoduché a jasné zprávy, jenž je ve výjimce předávána. Ta totiž informuje proč daná situace nastala a umožňuje se tomuto problému v budoucnu vyhnout.

Prostřednictvím API je možné vytvořit nové PLC, aktualizovat IP adresu starého či existující PLC smazat. Všechny tyto požadavky jsou vázány na zajištění či zrušení komunikace s reálným hardwarem. Práci se spojením zajišťuje v PlcService objekt typu PlcConnector, který disponuje metodami connect a disconnect. Ty kromě zajištění samotného připojení či odpojení od PLC též aktualizují jednorázově informace získávané prostřednictvím této komunikace, což umožní klientovy ve zpětné odpovědi zaslat všechny potřebné detaily.

3.4.2 Automatická aktualizace informací

Aktualizace informací v databázi je obstarávána instancí třídy AutomaticUpdateService. Tato třída implementuje rozhraní z návrhového vzoru pozorovatel, což jí umožňuje přihlášení se k odběru jednotlivých změn, které nastaly v objektu typu PlcData. Z těchto rozhraní dědí metody onFirmwareNumberChange, onSerialNumberChange, onToolDataChange, onMotorCurveChange a onConnectionStatusChange, které jsou zavolány jakmile dojde k nastavení nové hodnoty příslušných atributů.

Jediný objekt této třídy, který je vytvořen při startu aplikace frameworkem Spring, se stará o aktualizaci veškerých informací v databázi. Je tedy nutné, aby odebíral data ze všech instancí třídy PlcData, které reprezentují jednotlivá spojení s konkrétními PLC. K registraci dochází při vytváření nového PLC či změně IP adresy již existujícího. Tato instance kromě samotného uložení současných dat do databáze též při detekci změny odesílá aktuální stav přes protokol WebSocket skrze odpovídající témata. S tímto je spojena i příslušná konverze na DTO, která zde musí probíhat.

Při zavolání jakékoli metody je prvním krokem načtení uložených dat z databáze a ověření, zda v ní existuje PLC s odpovídající IP adresou reprezentující datové spojení. Pokud

nedošlo k nějaké chybě, tak by takovýto záznam měl být vždy nalezen. U získaného PLC je pak jednoduše aktualizován konkrétní atribut a změna je uložena zpět do databáze.

Komplikovanější proces nastává u detekce změny nástroje. Zde je nutné zkontrolovat, zda nově získané informace o nástroji odpovídají již existujícímu objektu, či je zapotřebí vytvořit nový. Pokud je detekována shoda, tak je nastaven tento nástroj jako aktuální. V případě, že však nebyl žádný odpovídající objekt nalezen, dojde k vytvoření nového nástroje, přiřazení k příslušnému PLC a jeho zvolení jako aktuálně používaného. Tvorba takovéhoto nástroje je naznačena v následující části kódu.

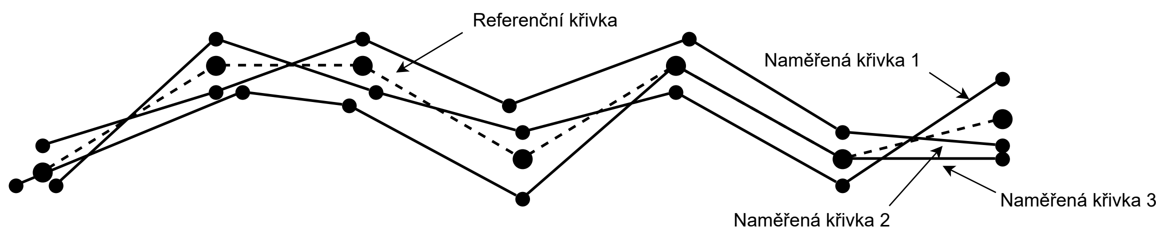
```
Tool autodetectedTool = Tool.builder()
                        .toolNumber(plcData.getToolData().getToolNumber())
                        .nameFromPlc(plcData.getToolData().getToolName())
                        .toolStatus(ToolStatusType.AUTODETECTED)
                        .automaticMonitoring(false)
                        .calculateReferenceCurve(false)
                        .build();
```

Kromě nastavení všech informací jako je číslo nástroje, originální jméno nástroje, vypnutí automatického monitorování, vypnutí kalkulace referenční křivky, je zde nastaven i status nástroje jako AUTODETECTED, čímž je dáno najevo, že nemůže dojít ke změně čísla nástroje uživatelem.

3.4.3 Vyhodnocování cyklu

3.4.3.1 Tvorba referenční křivky

Jeden lisovací cyklus je reprezentován křivkou závislosti točivém momentu na rychlosti danou 360 body, které jsou měřeny ve stejných fázích procesu. Referenční křivka je počítána na základě uživatelem zadaného počtu cyklů jako aritmetický průměr sobě odpovídajících bodů. Tento postup je naznačen pro výpočet ze tří křivek na obrázku 3.5 . Spojení mezi jednotlivými body jsou zde vyobrazeny pouze kvůli přehlednosti.



Obrázek 3.5: Výpočet referenční křivky

Veškerá logika týkající se výpočtu se nalézá ve třídě ReferenceCurveCalculationService-Impl, která se chová jako pozorovatel stavu a v případě, že došlo k získání nově naměřené křivky z PLC je zavolána metoda onMeasuredCurveChange. V rámci ní jsou načteny informace z databáze a proběhne kontrola toho, zda má být na základě nově příchozích dat

spočítána křivka referenční. Pokud tomu tak je, následuje pokus o získání rozpracovaného výpočtu z atributu calculations typu HashMap. Klíč v něm představuje IP adresa PLC, ke které je přiřazena příslušná kalkulace reprezentovaná instancí třídy ReferenceCurveCalculation. Jestliže atribut calculations žádný takový záznam neobsahuje, dojde k vytvoření nového objetu ReferenceCurveCalculation, který vznikne pomocí konstruktoru, kterému je zapotřebí předat požadovaný počet cyklů. Takto nově vytvořený výpočet je poté vložen do calculations a aktuálně naměřená křivka je do něj zařazena. Daný postup se opakuje dokud nedojde k dosažení požadovaného počtu cyklů. Informace o aktuálním stavu jsou zasílány přes WebSocket na téma `/topic/tools/calculation-status`. Jakmile je výpočet křivky dokončen proběhne její uložení do databáze a odstranění odpovídajícího záznamu z hash mapy.

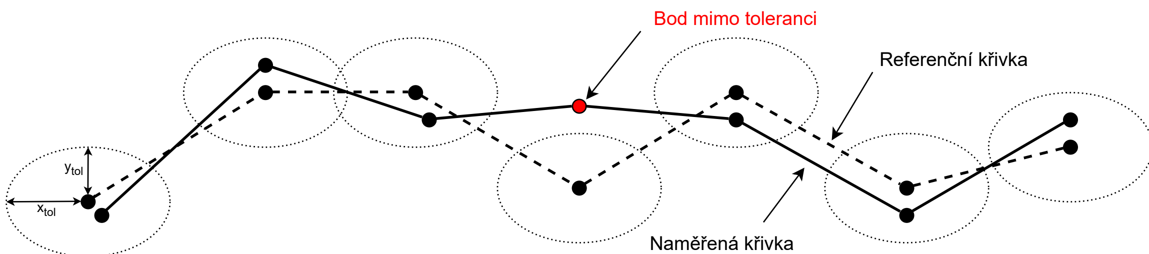
Referenční křivka je vytvářena pouze z po sobě následujících cyklů a nemůže tak v průběhu výpočtu dojít ke změně nástroje. Pokud by takováto situace nastala je zavolána metoda onToolDataChange a všechny doposud zaznamenané cykly jsou smazány. Jakmile se po té nástroj opět změní zpět na původní, tak začne celý výpočet znovu.

3.4.3.2 Validace naměřené křivky

K vyhodnocení toho, jestli naměřená křivka splňuje předepsané tolerance dochází na základě postupu naznačeného na obrázku 3.6. Jednotlivé body referenční křivky jsou porovnávány s body křivky naměřené tak, že kolem bodu referenční křivky je vytvořena elipsa jejíž délky hlavní a vedlejší poloosy odpovídají předepsaným tolerancím (na obrázku x_{tol} a y_{tol}). Bod naměřené křivky tedy musí splňovat rovnici 3.1, kde x a y jsou souřadnice bodu naměřené křivky, x_{ref} a y_{ref} jsou souřadnice bodu referenční křivky a x_{tol} a y_{tol} jsou předepsané tolerance.

$$\frac{(x - x_{ref})^2}{x_{tol}^2} + \frac{(y - y_{ref})^2}{y_{tol}^2} \leq 1 \tag{3.1}$$

Jinými slovy pokud tedy bod naměřené křivky leží uvnitř či na okraji elipsy je vše v pořádku, v opačném případě je tento bod vyhodnocen jako problematický.



Obrázek 3.6: Validace naměřené křivky

Implementaci dříve popsaného postupu je možné nalézt ve třídě AutomaticMonitoringServiceImpl, která sleduje, zda nedošlo k získání nových dat z PLC. Jakmile je detekována změna naměřené křivky, proběhne ověření toho, zda je na daném nástroji sepnutá automatická kontrola. Pokud tomu tak je, je naměřená křivka vyhodnocena oproti křivce referenční

s požadovanou tolerancí. Toto porovnání probíhá uvnitř třídy CurveValidationServiceImpl, jež má metodu validate přijímající, naměřenou křivku, referenční křivku a požadovanou toleranci. Její návratovou hodnotou je množina bodů, které předepsanou toleranci nesplnily. Pokud existuje alespoň jeden takový bod, tak dojde k vytvoření záznamu v databázi, jenž obsahuje všechny důležité informace o současném stavu. Zároveň jsou tato data zaslána přes WebSocket na téma /topic/logs/new-log. Uživatel též může nastavit, že v případě nesplnění tolerance je dána zpětná vazba PLC, které na ni již může reagovat, například zastavením lisovacího procesu, aby nedošlo k poškození nástroje.

3.5 Komunikace s PLC

3.5.1 Reprezentace spojení

Centrem pro zajištění komunikace s PLC je třída PlcConnector. Spolu se spuštěním aplikace je automaticky vytvořena její implementace, která v sobě spravuje všechna existující připojení v podobě hash mapy, kde klíčem je IP adresa PLC v podobě Stringu a jí příslušný objekt třídy PlcData, reprezentující dané spojení.

3.5.1.1 Připojení k PLC

Při zavolání metody connect, která jako jako parametr přijímá objekty třídy Plc (a ty též vrací), dojde na základě IP adresy ke kontrole, zda již dané spojení existuje. Pokud tomu tak není musí dojít k vytvoření nového objektu typu PlcData. Jak již bylo zmiňováno PlcData je abstraktní třída sloužící jako rozhraní pro různé druhy připojení. Její konkrétní implementace je tak poskytována Springem spravovaným objektem typu PlcDataProvider, který při zavolání metody getPlcData vytvoří reprezentaci příslušného spojení. V současné chvíli je možné využít připojení pouze přes protokol OPC UA. Tento proces tak zajišťuje nyní jen lepší rozšiřitelnost do budoucnosti a pokaždé navrací instanci třídy PlcDataOpcua. Takto nově získaný objekt typu PlcData je spolu s příslušnou IP adresou uložen do hash mapy.

Pokud došlo k vytvoření nového spojení je dalším krokem přihlášení všech potřebných pozorovatelů, jako jsou automaticUpdateService, automaticMonitoringService a ReferenceCurveCalculationService k odběru aktuálních dat z PLC. Toto je prováděno pomocí privátních tříd, které zajistí, registraci k potřebným atributům třídy PlcData specifických pro dané objekty.

Posledním krokem je aktualizace informací o PLC získaných na základě tohoto připojení. Pokud spojení bylo úspěšné je PLC označeno za připojené, jsou nastavena data o jeho hardwaru a aktuálně používaném nástroji. V opačném případě dojde ke zvolení stavu DISCONNECTED. Toto PLC s aktuálními informacemi je poté navraceno zpět, aby s ním bylo možné dále pracovat. Celý předešlý proces je naznačen v následující části kódu.

```
public Plc connect(Plc plc){
    PlcData plcData = plcDataMap.get(plc.getIpAddress());

    if(plcData==null){
        plcData = plcDataProvider.getPlcData(plc.getIpAddress());
        plcDataMap.put(plc.getIpAddress(), plcData);
        registerForAutomaticPlcInformationUpdate(plcData);
        registerForCurveValidation(plcData);
        registerForReferenceCurveCalculation(plcData);
    } else {
        log.warn("Connection for PLC with IP address {} already exist.",
        plc.getIpAddress());
    }
    return updatePlcInformation(plc,plcData);
}
```

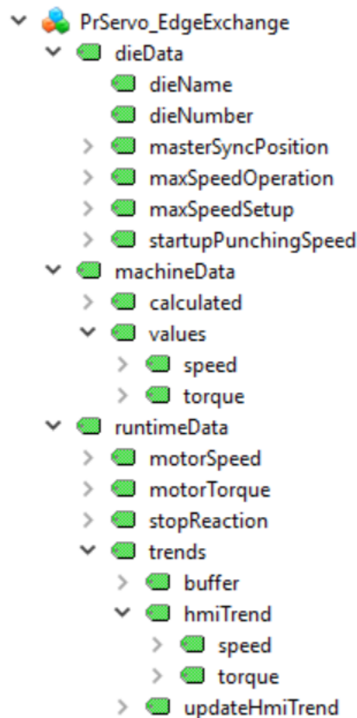
3.5.1.2 Odpojení od PLC

Odpojení od PLC je značně jednodušší než jeho připojení. Při zavolání metody disconnect dojde k prohledání hash mapy na základě IP adresy, a pokud je pro tuto IP adresu nalezeno spojení, je provedeno zrušení odběru informací u všech registrovaných pozorovatelů. Následujícím krokem je po té označení PLC jako odpojeného. Pokud je zapotřebí změnit IP adresu PLC je nutné jej nejprve odpojit pomocí metody disconnect a následně jej připojit za pomoci nové IP adresy.

3.5.2 Struktura dat v PLC

Každé PLC, které obsluhuje lisovací proces a má spolupracovat s touto aplikací musí obsahovat data blok PrServo_EdgeExchange, který je znázorněn na obrázku 3.7. Data jsou zde rozdělena do třech kategorií, první jsou dieData, která obsahují informace o právě používaném nástroji. Druhou kategorií je machineData, tedy informace o samotném lisu. Ty obsahují například záznamy o limitech motoru, který je používán. Poslední je pak runtimeData, kde se nachází informace o aktuálním cyklu a proměnná pro zpětnou vazbu. Detaily o PLC jsou pak načítány ze standardních lokací OPC UA serveru.

K získání těchto dat přes protokol OPC UA je nutné znát jejich NodeId, které se skládá z NamespaceIndexu a Identifieru. Tyto detaily jsou uloženy v souboru opcua-production.properties, aby byla jejich změna možná na jednom místě a nebylo potřeba je hledat v kódu. Při startu aplikace proběhne jejich načtení do objektu typu OpcuaConfiguration, kde jsou z nich následně automaticky vytvořeny instance typu NodeId pro knihovnu zajišťující komunikaci přes tento protokol. Vpravování konfigurace do samotných objektů typu PlcData po té probíhá v PlcDataProvideru prostřednictvím konstruktoru.



Obrázek 3.7: Struktura dat v PLC

3.5.3 Definice vlastních struktur

K zajištění komunikace mezi backendem a PLC přes protokol UPC UA je využívána knihovna Milo. Ta umožňuje automatické čtení proměnných základních typů a jejich polí jako jsou například int, float, nebo String. Této funkcionality bylo využito při zjišťovaní informací o sériovém čísle či firmwaru v PLC. Avšak takovýmto způsobem nemohou být získávána všechna data. Nejen, že by daný postup nebyl vždy nejefektivnější, ale například při čtení právě naměřené křivky nelze přečíst pouze pole hodnot točivého moment a po té zvlášť pole rychlostí, neboť by mohla nastat situace, kdy by mezi jednotlivými požadavky na získání dat mohlo dojít ke změně a polovina hodnot by tak náležela jiné křivce. Podobná situace nastává i při načítání informací o aktuálně používaném nástroji.

Ze struktury dat na obrázku 3.7 je patrné, že je možné získávat celé proměnné jako například dieData či hmiTrend. Aby toto mohlo být realizováno, byly vytvořeny jejich reprezentace pro knihovnu Milo, které po té jsou registrovány do příslušného dekodéru. Došlo tak k napsání třídy CurveStructure, reprezentující datový typ křivky a ToolDataStructure, představující strukturu informací o nástroji. Obě tyto třídy implementují rozhraní knihovny UaStructure, které zajišťuje, že nesou informaci o tom, kde se v OPC UA serveru nachází daný datový typ a jak jej dekódovat.

Příklad realizace metody decode pro datový typ křivky je zobrazen v následující části kódu. Jelikož jsou data zasílána v binární podobě je důležité v jakém pořadí budou načítána. Jméno položky v závorce se uplatňuje pouze tehdy, pokud by data byla získávána ve formátu XML. V průběhu vývoje prohození těchto řádků způsobilo velké zdržení, neboť kvůli

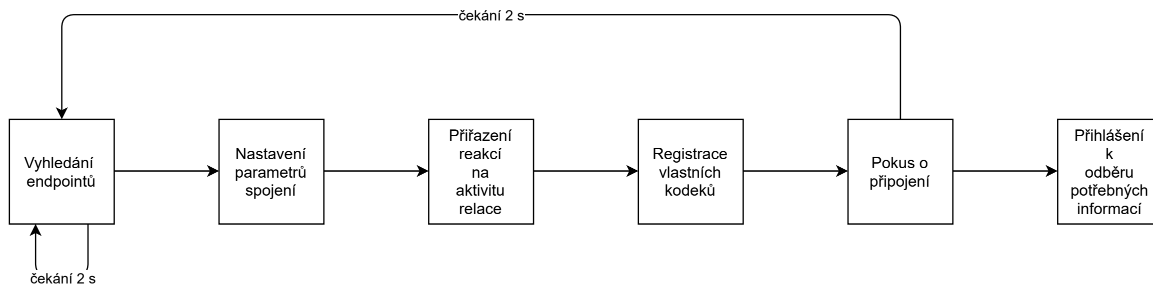
neexistující dokumentaci knihovny Milo nebylo možné tuto chybu jednoduše odhalit. K její opravě došlo až po kontaktování hlavního vývojáře knihovny, který vše uvedl na správnou míru.

```
public CurveStructure decode(
            SerializationContext context,
            UaDecoder decoder) throws UaSerializationException {

        //The order of those is important!!
        Float[] speedArray = decoder.readFloatArray("speed");
        Float[] torqueArray = decoder.readFloatArray("torque");

        return new CurveStructure(torqueArray,speedArray);
}
```

Obdobným způsobem je poté implementována i metoda encode, která však v rámci programu není využívána, neboť jsou pro data pro křivky z PLC pouze čtena.

3.5.4 Vytvoření spojení

K realizaci samotného spojení s PLC dochází vytvořením objektu třídy PlcDataOpcua. V rámci jejího konstruktoru dojde k vyhledání připojovacích endpointů na základě poskytnuté IP adresy a konfigurace. Současná verze programu nepodporuje zabezpečenou komunikaci a proto je pokaždé vybrán endpoint bez zabezpečení. Následně se nastaví parametry spojení jako například jméno aplikace či zvolený přístupový bod. Po té jsou pro dané připojení zvoleny reakce na změnu aktivity relace, kdy při její neaktivitě dojde k přepnutí statusu připojení na odpojeno a naopak při její aktivitě na připojeno. Jelikož všechny settery třídy PlcData, přes které takovéto aktualizace probíhají, automaticky oznamují všechny změny pozorovatelům, dojde po nastavení stavu připojení k jeho okamžitému předání zainteresovaným objektům. Dalším krokem je zaregistrování vlastních kodeků, které byly popsány v předchozí sekci. V konečné fázi je realizován pokus o připojení a přihlášení k odběru všech potřebných informací. Celý tento proces je pro přehlednost znázorněn na obrázku 3.8.



Obrázek 3.8: Vytvoření spojení

V rámci tohoto postupu může dojít k tomu, že se nebude dát připojit k discovery serveru (první krok) či selže připojení k samotnému PLC (předposlední krok) v obou těchto pří-

padech je nastaven stav připojení na DISCONNECTED a následuje vytvoření rekurzivního paralelního cyklu, který se bude pokoušet o připojení každé dvě sekundy. Celkový čas tohoto procesu je ukládán a logován.

Jakmile proběhne prvotní úspěšné připojení, tak jsou již jakékoli výpadky vyřizovány knihovnou Milo. Jediné co je zapotřebí zajistit je, že při znovupřipojení dojde ke smazání starých odběrů a jsou vytvořeny odběry nové.

Jelikož je celá knihovna Milo založena na paralelním programování pomocí CompletableFuture, byl problém dosáhnout toho, aby bylo pozastaveno aktuální vlákno, než dojde k získání první hodnoty po vytvoření odběru. Vše nakonec bylo vyřešeno pomocí napsání vlastní metody subscribe, která obalovala metodu knihovny a přidávala následující dva řádky kódu.

```
ManagedDataItem.DataValueListener listener =
    managedDataItem.addDataValueListener(result::complete);
result.whenComplete((v,e) ->
    managedDataItem.removeDataValueListener(listener));
```

Zde je v první části k odebíraným informacím přidán pozorovatel a jakmile dojde k jeho zavolání je označen proces za dokončený. V témže kroku následuje odebrání pozorovatele, který je již zbytečný. Po té je tak možné pomocí metody get, zavolané na návratovou hodnotu vytvořené metody subscribe, pozastavit aktuální běh vlákna dokud nedojde k vyřízení požadavku.

3.5.5 Automatické připojení při startu aplikace

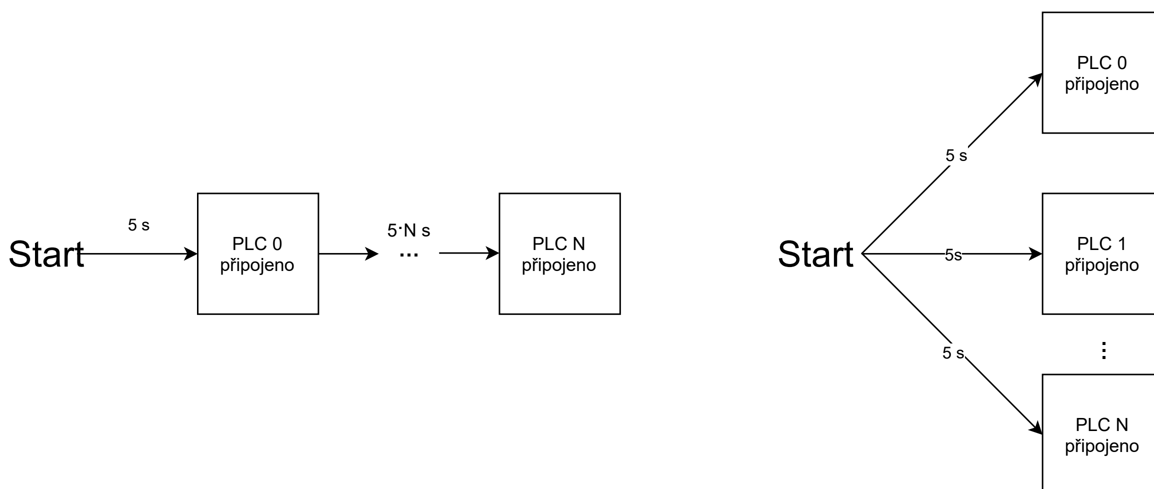
Při startu aplikace bylo nutné zajistit, aby došlo k připojení všech PLC, které se nachází v databázi a v rámci tohoto procesu aktualizovat uložená data. Tento poměrně jednoduchý úkol nelze řešit pomocí klasické for smyčky, neboť jedno připojení může trvat až 5s. Takový je totiž časový limit než dojde k označení daného PLC za odpojené. Pokud by se tedy v databázi nacházel velký počet PLC zabralo by toto připojování spoustu času viz obrázek 3.9 vlevo.

Nabízí se tak využití paralelního programování, kdy dojde ke spuštění připojování pro veškeré PLC najednou a po té se počká jakmile budou všechny tyto procesy dokončeny. V nejhorší situaci tak může celá tato akce zabrat nejvýše 5 sekund viz obrázek 3.9 vpravo.

Tato procedura probíhá v instanci třídy ApplicationStartupManager, která je označena anotací `@Component`, což frameworku Spring říká, že je nutné její instanci při startu automaticky vytvořit. Jakmile k tomuto dojde je zavolána její jediná metoda onApplicationStartup, v které dojde k načtení všech PLC z databáze a ty jsou pomocí následujícího kódu připojeny.

```
List<Plc> plcs = plcRepository.findAll().stream()
             .map(plc -> CompletableFuture.supplyAsync(() ->
                 plcConnector.connect(plc)))
             .map(CompletableFuture::join)
             .collect(Collectors.toList());
```

Je zde využito převodu získaného listu PLC z databáze na stream, kdy jsou jednotlivé objekty namapovány na paralelní procesy zajištující jejich připojení a následně jsou výsledky těchto částí navráceny opět v podobě listu. Jelikož je pro připojování využívána instance třídy PlcConnector, dojde pro jednotlivá PLC i k nastavení současných hodnot jejich atributů. Posledním krokem je uložení těchto aktualizovaných PLC zpět do databáze.



Obrázek 3.9: Využití paralelních vláken pro připojení PLC

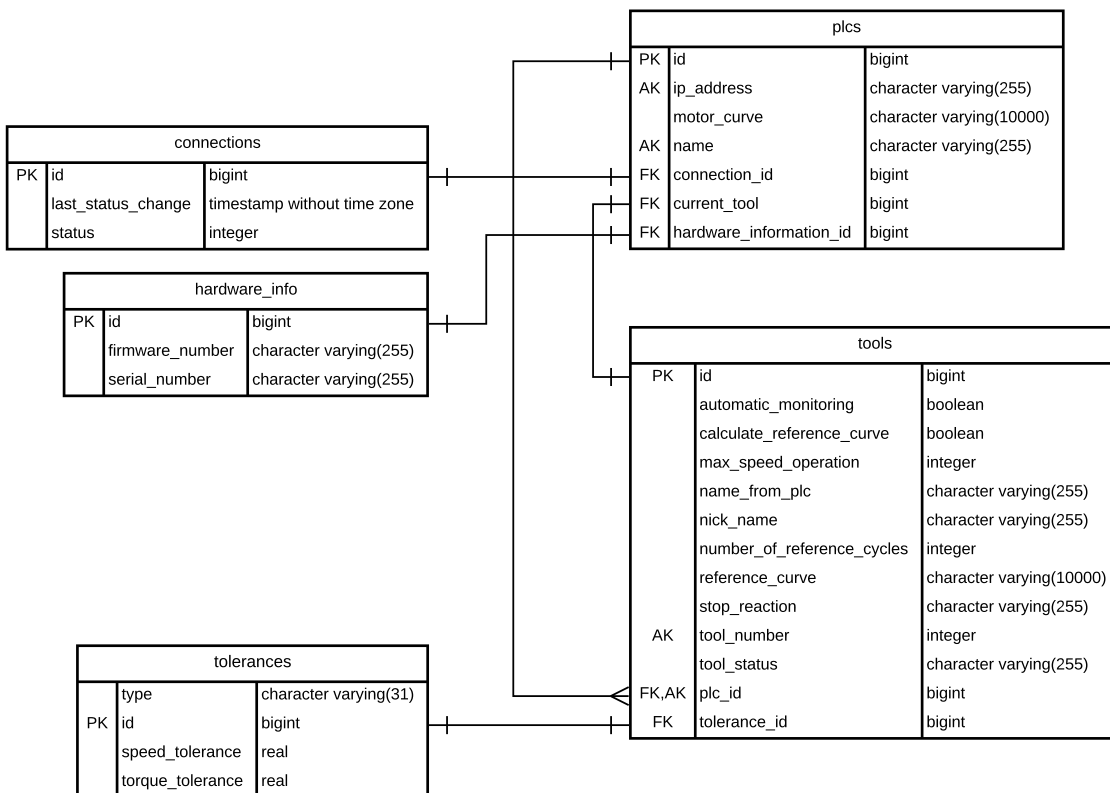
3.6 Databázová vrstva

3.6.1 Aktuální implementace

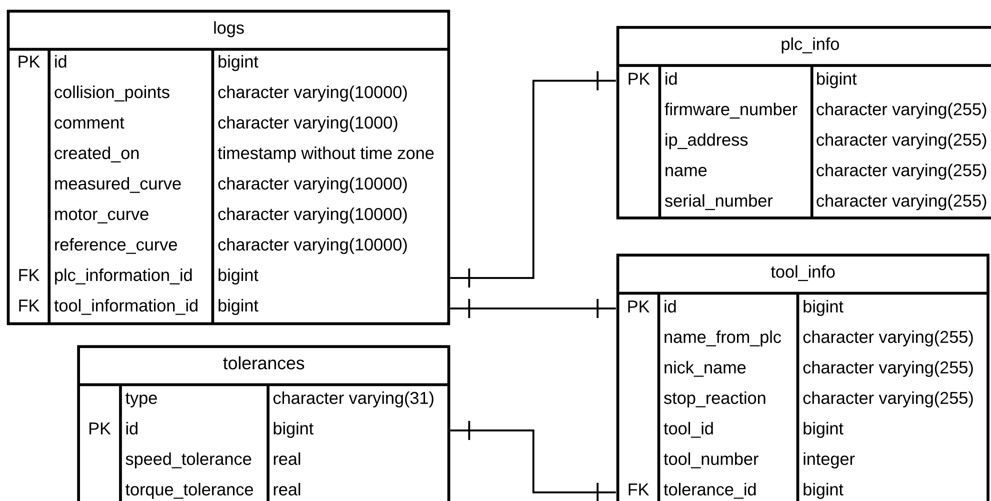
3.6.1.1 Struktura databáze

Současnou strukturu databáze, do které jsou data ukládána, lze rozdělit na dvě sekce. V první jsou zaznamenávána data o současně používaných PLC a jejich nástrojích. ER diagram pro tuto část je zobrazen na obrázku 3.10. Jak je patrné, tato část databáze se skládá z pěti na sebe navázaných tabulek. Za její centrum by se dala považovat tabulka plcs obsahující informace o všech PLC, které aplikace právě spravuje. Na ni jsou navázány tabulky connections a hardware_info, které mají s tabulkou plcs kardinalitu 1:1, tedy jednomu záznamu v tabulce plcs přísluší jeden záznam v obou zmiňovaných tabulkách. Jsou zde ukládány informace o stavu připojení jednotlivých PLC a informace o jejich HW. Stejnou kardinalitu má tabulka plcs i s tools pro přiřazení aktuálně používaného nástroje. Druhý vztah mezi těmito tabulkami má kardinalitu 1:N a reprezentuje všechny nástroje patřící k danému PLC. Každému nástroji pak náleží jedna tolerance, jejíž data jsou ukládána do tabulky tolerances.

Druhou částí databáze je část věnovaná ukládání logů znázorněná na obrázku 3.11. Jelikož má log představovat snímek stavu aplikace v čase jeho vytváření a ze své podstaty má být neměnný, jsou aktuální data o PLC překopírovávána do tabulky plc_info. To samé je provedeno pro informace o nástroji a jeho toleranci. Je možné si povšimnout, že tabulka tolerances se nachází v obou částech databáze. Neznamená to však, že by pro jeden nástroj



Obrázek 3.10: Struktura části databáze pro aktuální data



Obrázek 3.11: Struktura části databáze pro logy

tabulka tool_info odkazovala na stejný záznam v tabulce tolerances jako tabulka tools. Pokud by tomu totiž tak bylo, došlo by při změně tolerance na nástroji i ke změně informací o toleranci v logu, ten však jak již bylo řečeno má být neměnným záznamem historie. Veškeré tabulky v této části mají mezi sebou vztah s kardinalitou 1:1.

Struktura databáze není vytvářena na základě předem daného SQL skriptu, neboť se aplikace stále nachází ve vývoji a často dochází k úpravám. Její generování a aktualizace jsou zajišťovány frameworkem Hibernate při startu aplikace na základě tříd označených anotací `@Entity`. Jedinou nutnou podmínkou pro správnou funkci je tak existence databáze s daným jménem, které je specifikováno spolu s přístupovými údaji (jméno a heslo uživatele) a adresou DBMS v application.properties.

3.6.1.2 Entity

Každé tabulce databáze z dříve znázorněných ER diagramů odpovídá jedna třída umístěná v balíčku entity. Jsou zde definovány kaskádové typy, to jak je generováno id, omezení či typ načítání dat. Jelikož se jedná o velký počet tříd, budou zde vyzdviženy jen ty nejdůležitější.

Třída Plc Tato třída reprezentuje tabulku plcs. Stejně jako veškeré ostatní entity je zde atribut id odpovídající primárnímu klíči. Pro všechny tyto atributy je nastaveno automatické generování na AUTO, což zaručuje, že při ukládání do databáze jim jsou přiřazeny unikátní hodnoty. Pomocí této třídy jsou též nastavena omezení tabulky tak, že ip_address a name nemohou nabývat v odpovídajícím sloupci stejných hodnot a též nesmí být rovné null. Toto zaručuje, že i kdyby přes všechny kontroly, které jsou v rámci aplikace prováděny, došlo k požadavku na databázi pro uložení PLC s již existujícím jménem, dojde k vyvolání příslušné výjimky a data nebudou do databáze uložena.

Pro veškeré atributy představující vztahy tabulky plcs s ostatními tabulkami je použit kaskádový typ ALL. Což znamená, že jakékoli požadavky (vytvoření, úprava, smazání) aplikované na třídu Plc se promítnou jak do tabulky plcs tak, i do ostatních navázaných tabulek. Pokud je tedy smazáno příslušné PLC dojde k odstranění všech jeho nástrojů, informací o připojení a informací o hardwaru.

Jelikož nástroj nemůže existovat bez toho, aby byl přiřazen nějakému PLC, tak je list reprezentující všechny nástroje daného PLC nastaven tak, že neumožňuje existenci sirotků. Proto jakmile je odebrán nástroj z listu, a daná instance třídy Plc je uložena do databáze, dojde ke smazání příslušného osamoceného nástroje.

Získávání dat z databáze je nastaveno na LAZY, a to jak pro všechny nástroje, tak i pro křivku motoru. Tato data nejsou totiž vždy potřebná a jejich načtení by zvyšovalo dobu vyřízení požadavku. Důvodem je velikost dat v případe křivky motoru či nutnost získávat více hodnot z jiných tabulek v situaci načítání dat o nástrojích. Přístup k těmto informacím je tak možný pouze uvnitř metody označené anotací `@Transactional`, kdy dojde k dodatečnému dotazu na databázi, nebo ručním specifikováním toho, že data mají být v daný okamžik načtena.

Třída Tool Tabulce tools odpovídá třída Tool. Oproti třídě Plc je zde nastaveno větší množství restrikcí. Není tak možné uložit do tabulky tools dva nástroje, které by měly zároveň stejné plc_id a tool_number. Jak již totiž bylo řečeno, číslo nástroje je v rámci jednoho lisu, kterému odpovídá konkrétní PLC, unikátní. Dále je zde určeno, že sloupce odpovídající atributům plc, toolNumber, calculateReferenceCurve, stopReaction, automaticMonitoring a toolStatus nemohou nabývat hodnoty null. Pro některé tyto atributy jsou nastaveny výchozí hodnoty, jiné musí být navoleny před tím, než dojde k pokusu o uložení do databáze. Obdobně jako u třídy Plc je tu, kvůli rychlosti vyřízení dotazu, zvoleno získávání dat o referenční křivce na LAZY. Pro atribut tolerance, představující navázání tabulky tools na tabulku tolerances, je nastaven kaskádový typ na ALL, neboť je požadováno, že při smazání, uložení či aktualizaci dat o nástroji dojde k provedení stejné operace pro příslušnou toleranci.

Třída Log Třída Log představuje obdobně nazvanou tabulku logs. Pro všechny její atributy a jim odpovídající sloupce jsou nastavena dvě základní omezení, a to za prvé, že se nesmějí rovnat null a za druhé, že je nesmí být možné po vytvoření změnit. Jedinou výjimkou je atribut comment, pro který ani jedna z těchto restrikcí neplatí. Tato omezení jsou podpořena ještě tím, že třída Log obsahuje jeden jediný setter a to právě pro comment. Nastavení zbývajících atributů na final však není možné kvůli tomu, jak s nimi framework Hibernate zachází. Dojde totiž nejprve k vytvoření prázdné instance pomocí konstruktoru bez argumentů a následně jsou jednotlivé atributy nastavovány pomocí reflexe.

V rámci Logu jsou obsaženy i atributy představující referenční křivku, křivku motoru, naměřenou křivku a množina bodů, které nesplnily předepsanou toleranci. Všechny tyto informace nejsou načítány z databáze automaticky při získávání instance třídy Log, ale až při jejich vyžádání. Opět je to z důvodu, že například při zasílání přehledu všech logů tyto informace nejsou potřeba.

3.6.1.3 Repositáře

Pro získávání dat z databáze v podobě tříd popsaných v předchozí sekci slouží rozhraní definovaná v balíčku repository. Jejich implementace jsou poskytovány automaticky frameworkem Spring. Za vyzdviženi zde stojí například specifické metody, které byly definovány na základě JPQL. Příklad takovéto metody je znázorněn v následujícím kousku kódu.

```
@Query("select case when (count(plc) > 0)  then true else false end " +
        "from Plc plc where plc.ipAddress = ?1 and plc.id != ?2")
boolean existsByIpAddressIgnoringId(String ipAddress, Long id);
```

Jedná se o metodu rozhraní PlcRepository, která je používána při aktualizaci IP adresy PLC, neboť umožňuje ověřit zda v databázi již existuje PLC s danou adresou s tím, že ignoruje záznam s konkrétním id. Tento ručně psaný požadavek pomocí JPQL, který je vidět v kódu uvnitř anotace `@Query` určuje, že dojde ke spočítání všech záznamů v tabulce plcs, které mají IP adresu danou prvním parametrem metody a zároveň jejich id není shodné s id, jež je předáváno jako druhý parametr. Pokud byl nalezen alespoň jeden takový záznam, pak je navrácena hodnota true, v opačném případě metoda vrací false.

Pro některé dotazy je též třeba specifikovat, jaká data mají být z databáze získána. Toto je prováděno pomocí anotace `@EntityGraph`, která umožní zvolit jednotlivé položky, jež je třeba v rámci jednoho požadavku načíst. Výhoda tohoto postupu oproti získávání informací uvnitř transakční metody je, že je ušetřen minimálně jeden dotaz na databázi. V případě načítání dat v metodě označené anotací `@Transactional` totiž dojde k získání základních dat (těch, která nejsou načítána jako LAZY) a až po té, když jsou zbývající data v rámci kódu vyžadována, je vytvořen dodatečný požadavek na databázi. Pokud však je jasné, že tyto informace jsou pro danou část programu vždy nutné, je lepší využít jejich načtení v rámci jednoho dotazu. Příklad takovéto metody je zobrazen v následujícím kuse kódu.

```
@EntityGraph(attributePaths = {
    "collisionPoints",
    "plcInformation",
    "toolInformation",
    "toolInformation.tolerance",
    "motorCurve",
    "referenceCurve",
    "measuredCurve"})
Optional<Log> findById(Long id);
```

Tato metoda je definována v rozhraní LogRepository a je v rámci programu používána u zasílání detailů o jednom logu. Je zde využito toho, že Spring framework podle názvu metody automaticky vygeneruje její implementaci. Pokud by však nebyl specifikován EntityGraf, došlo by při získávání dat z databáze k více než jednomu dotazu, neboť ve třídě Log jsou některé atributy nastaveny na načítání LAZY. Z toho důvodu je zde v rámci anotace specifikováno, že všechny informace o dané třídě mají být získány z databáze jedním dotazem, v rámci kterého dojde ke spojení tabulek logs, plc_info, tool_info a tolerances. Ten sice bude trvat déle, než dotaz získávající pouze základní informace, ale pokud by byly započteny všechny dodatečné dotazy, tak by měl být ve výsledku rychlejší.

3.6.1.4 Vlastní mapování tříd

V prvních fázích vývoje backendu byla data reprezentující křivky ukládána do dvou tabulek. Jedna obsahovala samostatné body a druhá poté celé křivky složené z těchto bodů. Ovšem při načítání takto uložených informací z databáze, trvaly dotazy velmi dlouho, a proto byl definován vlastní konvertor, který příslušné třídy před ukládáním převede na text v následujícím formátu.

```
-39.20828,218.25061;51.578445,379.8077;57.121655,636.8528; ...
```

Tato konverze je implementována v rámci třídy CurveConverter, která dědí z generického rozhraní AttributeConverter a musí díky tomu obsahovat dvě metody. První slouží k převodu ukládané třídy na String a druhá pak k vytvoření instance na základě textu získaného z databáze. Implementace metody `convertToDatabaseColumn` je v celku jednoduchá. V rámci for smyčky dojde k projití všech bodů křivky, a ty jsou pak uloženy jako String. První je hodnota kroutícího momentu oddělená čárkou od hodnoty rychlosti, a následně je použit

středník pro separaci jednotlivých bodů. Jelikož se jedná o cyklické skládání Stringů, je použit StringBuilder. Využitím klasického + by totiž s každým přidáním informace docházelo k vytvoření nového String, což je způsobeno tím, že String je v Javě neměnný.

Implementace metody `convertToEntityAttribute`, která slouží ke konverzi textu na instanci dané třídy, je řešena rozdělením textu nejdříve pomocí středníku na pole Stringů představujících jednotlivé body. Ty jsou pak v rámci for smyčky rozloženy na dvě části reprezentující hodnoty točivého momentu a rychlosti. Následuje převod na desetinná čísla, ze kterých je pomocí konstruktoru vytvořen bod křivky. Ze všech takto získaných bodů je poté zkonstruována výsledná křivka.

Obdobným způsobem je též řešeno ukládání bodů nesplňujících toleranci v rámci logu. Jejich reprezentace pomocí Stringu je stejná jako u křivky, ovšem kvůli tomu, že se jedná o jinou třídu, pro ně musel být vytvořen samostatný konvertor.

Jelikož se délka takto vytvořeného textu pohybuje okolo 8000 znaků, byl pro dané sloupce tabulky zvolen datový typ character varying(10000). Tato hodnota je nastavena jak pro všechny křivky, tak i pro body nesplňující toleranci, neboť jejich počet v případě nastavení velmi malé tolerance může být maximálně roven počtu bodů křivky.

3.6.1.5 Vytváření logů

Pro vytváření logů slouží rozhraní LogCreator jehož implementace je generována automaticky pomocí generátoru kódu Mapstruct. V rámci tohoto rozhraní je definována metoda create, jež přijímá tři parametry typů Plc, Curve a Set<PointOfTorqueAndSpeed>. Ty reprezentují aktuálně naměřenou křivku, na které došlo k problémům, jí odpovídající PLC a množinu bodů, které nesplňovaly toleranci. Tato metoda je zobrazena v následujícím kódu.

```
@Mapping(target = "id", ignore = true)
@Mapping(target = "toolInformation", source = "plc.currentTool")
@Mapping(target = "referenceCurve",
    source = "plc.currentTool.referenceCurve")
@Mapping(target = "plcInformation", source = "plc")
@Mapping(target = "motorCurve", source = "plc.motorCurve")
@Mapping(target = "comment", ignore = true)
Log create(
    Plc plc,
    Curve measuredCurve,
    Set<PointOfTorqueAndSpeed> collisionPoints);
```

Pomocí anotací `@Mapping` je určeno, odkud kam mají být data kopírována. První řádek tedy říká, že id vytvářeného logu má být ignorováno a zůstane prázdné. Druhý řádek pak říká, že data, která mají být nakopírována do toolInformation, se nachází v atributu instance třídy Plc s názvem currentTool. Třetí řádek informuje o lokaci zdroje dat pro referenční křivku, čtvrtý pak určuje odkud mají být brána data o PLC. Pomocí pátého řádku je definován zdroj pro motorovou křivku a poslední řádek pak udává, že comment má zůstat prázdný. Zbylé atributy, které zde nejsou zmíněny, byl MapStruct schopen namapovat automaticky.

Jelikož je požadováno, aby při ukládání logu do databáze byla vytvořena kopie záznamu o toleranci, tak je nutné, aby nebylo zkopírováno i její id, neboť by poté došlo pouze k odkázání na původní hodnotu. Toto je zaručeno definováním metody pro konverzi tolerance, která je vidět v následující části kódu.

```
@Mapping(target = "id", ignore = true)
AbsoluteTolerance toAbsoluteTolerance(AbsoluteTolerance absoluteTolerance);
```

Použití takto definovaného LogCreatoru je poté již snadné, neboť při startu aplikace dojde k vytvoření instance spravované Springem. Ta je poté vpravena do všech potřebných tříd, které již využívají pouze její metodu create, jež předají potřebná data a jako výsledek získají nově vytvořený log, který je již možné uložit do databáze.

3.6.2 Možná vylepšení

3.6.2.1 Nevýhody a výhody současné implementace

Aktuální řešení databázové vrstvy je zcela funkční a má velkou výhodu v tom, že jej nebylo až tak komplikované implementovat. Problémem zde však je, že při vytváření logů v databázi mohou být stejná data ukládána neustále dokola. Při hypotetické situaci, kdy by za den bylo vytvořeno například 100 logů bez jakékoli změny na PLC či nástroji, tak by do databáze byla 100x uložena totožná informace o motorové a referenční křivce, PLC, nástroji a toleranci. Jediné co by dané logy odlišovalo, by byla data reprezentující naměřenou křivku, body nesplňující toleranci a čas, kdy byl log vytvořen.

Jak již bylo vysvětleno, tuto situaci nelze vyřešit pouhým odkázáním logu například na PLC v databázi, což by sice vyřešilo problém s ukládáním redundantních dat, ale při změně informací o PLC by nebylo možné získat log s původními daty. Tento problém by však mohlo vyřešit vytvoření nového řádku v tabulce plcs při každé změně již uložených záznamů.

3.6.2.2 SCD

Takovéto řešení popisuje SCD typu 2, kdy je tabulka rozšířena o sloupec obsahující datum a čas vytvoření daného záznamu a sloupec, od kdy daný řádek již není aktuální. Ruční implementace takovéhoto řešení není triviální a zkomplikuje i jednoduché operace jako je získávání všech PLC z databáze, neboť bude třeba filtrovat pouze ta aktuální.

Komplikací je též zajištění vytváření nových řádků při změně, neboť toto by mělo být řešeno v rámci DBMS pomocí akcí, které se spustí, jakmile dojde k aktualizaci záznamu. Podobná situace nastává též při mazání, kdy při odstranění logu nelze automaticky smazat i záznam o PLC, ale je nutné kontrolovat, zda je aktivní, či zda na něj neodkazuje nějaký jiný log.

3.6.2.3 Hibernate Envers

Jedno z možných řešení, jak se vyhnout ruční implementaci SCD, nabízí framework Hibenate se svými Envers. Jednoduchou anotací @Audited přidanou nad entitu, u které má být

v databázi ukládána historie, dojde k vytvoření historické tabulky, kam jsou zaznamenávány všechny změny. Toto řešení se tak vyhne tomu, že by při práci s aktuálními daty bylo potřeba filtrovat data historická, a zároveň nabízí jednoduché rozhraní pro získávání historických dat. Problémem zde však je, že Envers jsou určené pro auditování, tedy zachovávání všech změn, a není proto jednoduché historická data smazat.

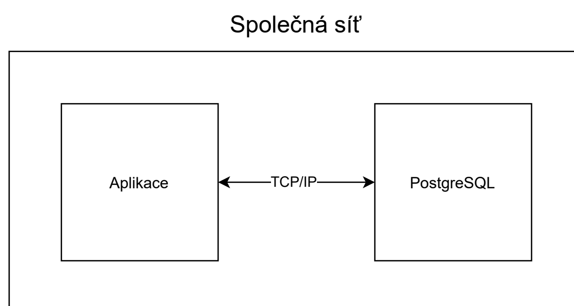
Jelikož je aktuální řešení plně funkční a není ještě napevno dána struktura databáze, nebyla tato optimalizace řešena. Jakmile však dojde k ustálení struktury databáze, bylo by vhodné se na toto téma zaměřit a vyzkoušet jak možnosti použití Hibernate Envers, tak manuální implementace SCD.

Kapitola 4

Tvorba aplikace pro Industrial Edge

4.1 Struktura aplikace

Aplikace pro Industrial Edge je složená ze dvou kontejnerů, databáze a samotné aplikace psané v jazyce Java, která kromě poskytování API slouží též jako sever pro frontend. Tuto funkcionalitu zajišťuje Spring Boot framework, který automaticky sdílí všechny soubory ze složky /src/main/resources/static, kam musí být kód frontendu umístěn. Tyto dva kontejnery spolu komunikují ve vlastní síti viz obrázek 4.1.



Obrázek 4.1: Architektura kontejnerizované aplikace pro Industrial Edge

4.2 Tvorba Dockerových obrazů

4.2.1 Aplikace

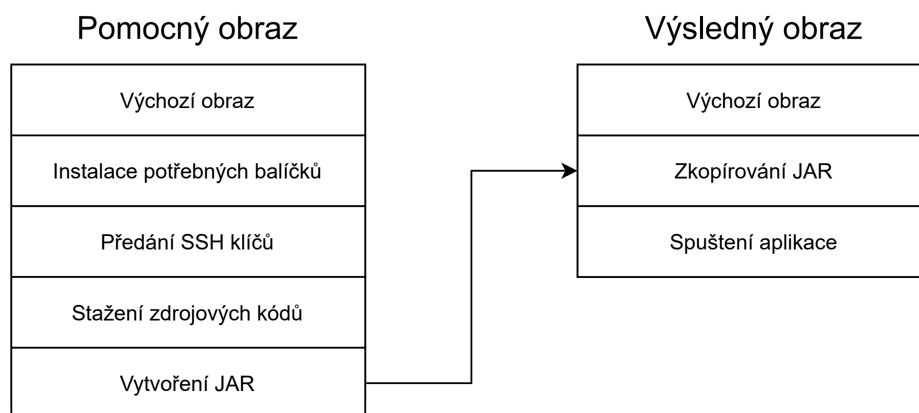
Struktura obrazu pro aplikaci je dána pomocí Dockerfile, který je umístěn v rootu projektu. Jedním z problémů při vytváření tohoto souboru bylo to, že při tvorbě obrazu je nutné získat zdrojový kód pro frontend ze soukromého repositáře za použití SSH klíčů. Není totiž možné vzít privátní SSH klíč a nakopírovat jej do obrazu, neboť jak bylo popsáno již dříve, s každým příkazem `COPY` je vytvořena nová vrstva obrazu, která by pak i přes smazání při jeho sdílení obsahovala dané klíče, což představuje velké bezpečnostní riziko. Řešením může

být využití tzv. multi-stage buildu, což umožní nakopírovat SSH klíč do obrazu, získat zdrojový kód ze soukromého repositáře a ten pak předat do jiného obrazu, který privátní klíč již neobsahuje a je jej tak bezpečné publikovat. Toto řešení je sice funkční, ale komplikované a vývojář si musí dávat pozor, aby kvůli nesprávné změně v Dockerfile nedošlo k úniku klíčů. Z toho důvodu Docker od verze 18 umožňuje využití klíčů počítače na kterém je obraz vytvářen viz dokumentace [9]. Stačí před každý příkaz, který potřebuje přístup k SSH klíči vložit `--mount=type=ssh`. I přes to, že je tato funkcionalita dostupná již delší dobu stále jí v době psaní této aplikace neimplementoval docker-compose a z toho důvodu bylo nutné využít multi-stage build a nakopírovat klíče do pomocného obrazu. Toto by bylo vhodné opravit jakmile docker-compose umožní využití SSH klíčů počítače.

Část kódu řešící problém s klíči je vidět níže. Klíče jsou předány kontejneru pomocí argumentu `SSH_PRIVATE_KEY`, který je zadán při tvorbě obrazu. Následně jsou nakopírovány do příslušné složky. Poslední dva řádky kódu přidávají veřejné klíče serverů ze kterých je zdrojový kód získáván do know_host, aby bylo předejito hlášce o nedůvěryhodnosti serveru.

```
ARG SSH_PRIVATE_KEY
RUN mkdir /root/.ssh \
&& echo "${SSH_PRIVATE_KEY}" > /root/.ssh/id_rsa \
&& chmod 600 /root/.ssh/id_rsa \
&& ssh-keyscan github.com >> /root/.ssh/known_hosts \
&& ssh-keyscan code.siemens.com >> /root/.ssh/known_hosts
```

Využití multi-stage buildu přináší i jiné výhody a to v podobě menší velikosti výsledného obrazu. Veškeré nástroje pro správu a automatizaci buildů jak pro frontend tak pro backend, zdrojové kódy či linuxové balíčky potřebné k vytvoření aplikace jsou pouze v dočasném obraze a do výsledného obrazu je nakopírován až finální JAR soubor. Přibližná struktura použitého multi-stage buildu je zobrazená na obrázku 4.2.



Obrázek 4.2: Multi-stage build

Při vytváření výsledného obrazu přišla v úvahu i možnost využít napsání skriptu, který by nahradil pomocný obraz a vytvořil JAR přímo a ten by byl opět nakopírován do finálního

obrazu. Toto řešení nakonec bylo zavrženo z důvodu toho, že by skript nemusel běžet pod různými operačními systémy a byl by závislý na přítomnosti nástrojů jako git, maven, npm a podobně. Jeho výhodou by sice byla jednoduchost při práci s SSH klíči, ovšem použité řešení nabízí možnost vytvoření výsledného obrazu na jakémkoli počítači bez nutnosti cokoli instalovat s tím, že práce s SSH klíči by měla být v brzké době vyřešena, jak již bylo zmiňováno výše.

4.2.2 Databáze

Obraz pro databázi je opět vytvořen na základě Dockerfile, který je umístěn ve složce database. Nyní se jedná pouze o čistý obraz databáze PostgreSQL. Šlo by tedy využít pouze ten, bez nutnosti vytváření příslušného souboru. Ovšem toto řešení je připraveno na budoucí nakopírování schématu databáze při její inicializaci. V současnosti je schéma generováno frameworkem Hibernate. Jedná se o plně funkční řešení, které umožňuje úpravu formátu tabulek přímo v kódu Java aplikace bez nutnosti zasahovat do jiných souborů. Což je v současném stavu vývoje rozumné řešení, neboť může dojít k drobným úpravám a bylo by komplikované upravovat schéma ručně. V pozdějších fázích vývoje by však bylo vhodné mít schéma databáze pevně dané.

4.3 Docker-compose

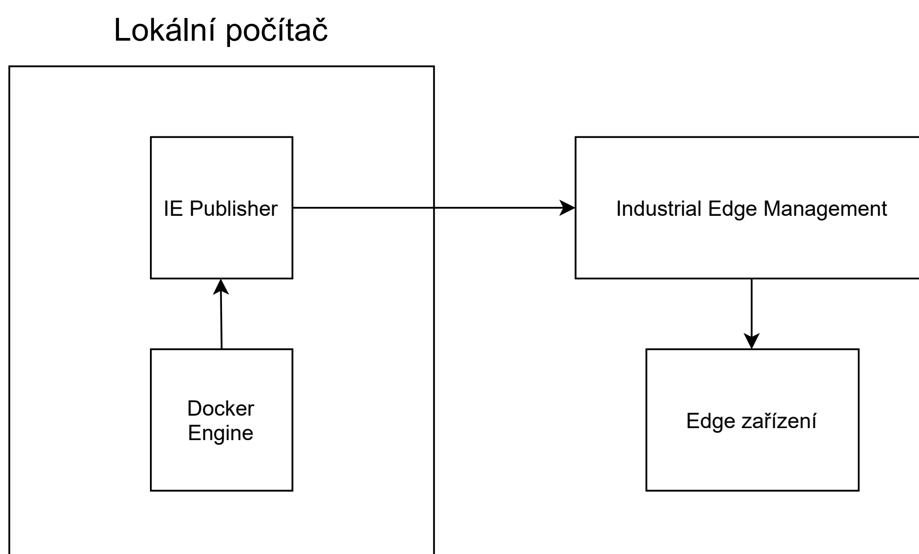
K dosažení funkčního celku složeného z databáze a samotné aplikace, byl využit nástroj docker-compose a k němu příslušný soubor docker-compose.yml, ve kterém je specifikovaná struktura aplikace. Soubor definuje dvě služby, a to app a db, pro které je implicitně vytvořena samostatná síť, ve které spolu mohou komunikovat. Jak již názvy napovídají, tyto služby představují aplikaci a databázi. Jelikož je nutné zajistit, aby databáze běžela dříve než, dojde ke spuštění aplikace, je pod službou app definováno `depends_on`, tedy v překladu závisí na, což zaručí správné pořadí spuštění obou služeb.

Jak pro aplikaci, tak pro databázi jsou definovány porty na kterých budou dostupné. Aplikace mezi sebou komunikují na vlastní síti, kde jim byly přiřazeny IP adresy. Jelikož se však tyto IP adresy mohou měnit s každým novým startem, je adresa databáze v kódu back-endu definována pomocí názvu služby, tedy db s příslušným portem. Aby nebylo nutné tuto hodnotu neustále přepisovat v application.properties, když je aplikace v rámci vývoje testována s lokálně běžící databází, lze volit z profilů dev a prod, kdy první slouží pro vývoj (development) a druhý je určen do produkce (production). Při vytváření obrazu aplikace je tak v Dockerfile uveden příkaz mavenu `mvn -P prod install`, který zaručí, že je použit produkční profil.

Dále je zde pro obě služby definován maximální počet zaznamenávaných logů. Pro databázi pak proměnné prostředí `POSTGRES_PASSWORD` určující přístupové heslo a `POSTGRES_DB` představující jméno výchozí databáze a pro aplikaci argument `SSH_PRIVATE_KEY`, který umožňuje předat SSH klíč. Tyto proměnné jsou potřebné pro vytvoření obrazů za použití `docker-compose build`.

4.4 Nahrání aplikace do zařízení

Jakmile jsou k dispozici obrazy obou částí aplikace, může být využit Industrial Edge Publisher, který se napojí na Docker Engine v počítači, čímž mu je umožněn přístup k lokálním obrazům. Při prvotním publikování aplikace je třeba vytvořit vytvořit konfiguraci podobnou výše vytvářenému docker-compose.yml, která je typická pro běh v prostředí Industrial Edge. Největším rozdílem je, že obsahuje definici pro přesměrování komunikace pomocí zabezpečených protokolů. Jakmile je tato konfigurace vytvořená jde na základě ní odeslat aplikaci do Industrial Edge Managemetu, který spravuje konkrétní Edge zařízení a je odtamtud možné aplikaci na jednotlivá zařízení nainstalovat. Tento postup je naznačen na obrázku 4.3. Jakmile dojde ke změnám v aplikaci (například vydání nové verze či záplaty) lze aplikaci přidat již na základě vytvořené konfigurace a pouze zvolit aktuální obrazy.



Obrázek 4.3: Proces nahrávání aplikace do Edge zařízení

Industrial Edge Publisher nabízí kromě verze s uživatelským rozhraním i verzi pro příkazový řádek, která je vhodná pro vytváření CI/CD pipeline a automatizování celého dříve popsaného procesu. Je tak například možné nastavit, že s novém přidáním kódu do master větve je spuštěn daný proces a aplikace je automaticky nahrána do Industrial Edge Management systému.

Automatizace nahrávání zatím nebyla implementována, neboť CLI verze byla v době tvorby aplikace novinkou a nebyla k ní ještě potřebná dokumentace. Raná fáze vývoje této verze též znamenala časté změny (vylepšení) a proto bylo rozhodnuto počkat jakmile se její implementace ustálí.

Kapitola 5

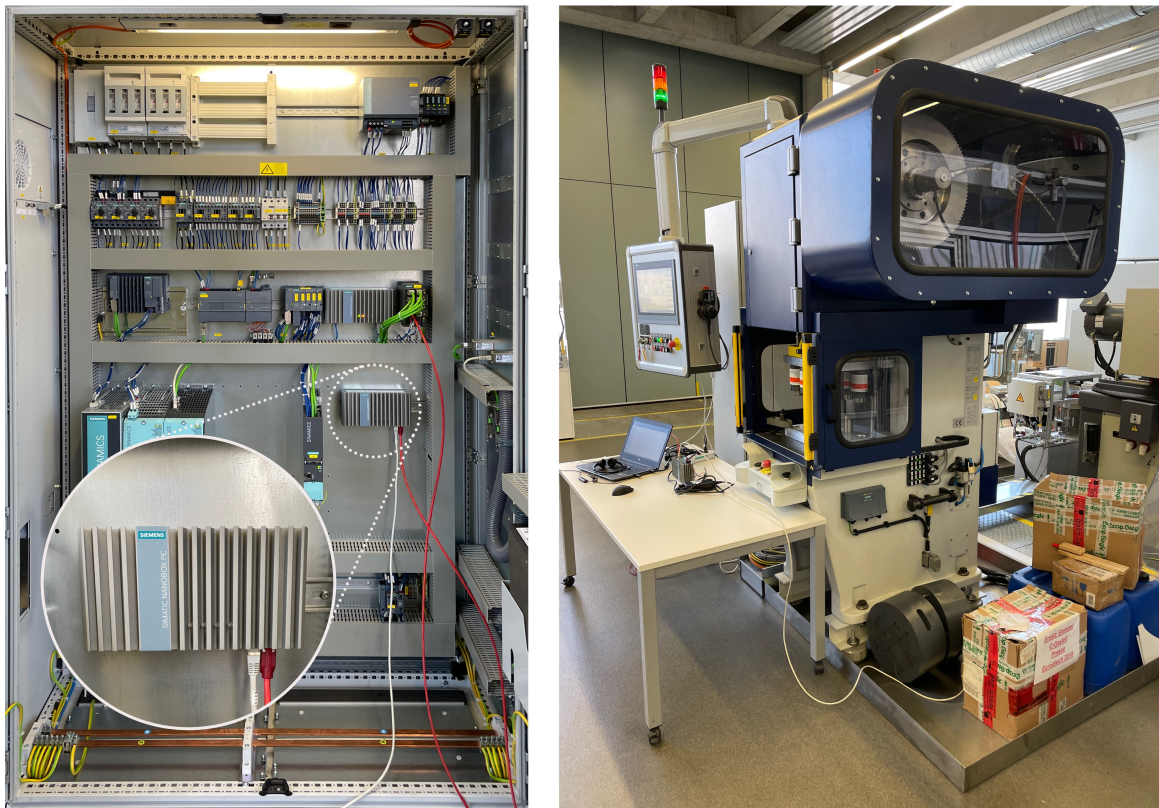
Závěr

V rámci této diplomové práce byl vytvořen funkční backend v programovacím jazyce Java pro Industrial Edge aplikaci monitorující lisovací proces. Tato část výsledné aplikace komunikuje s PLC řídícím lis přes protokol OPC UA a umožňuje pomocí API ovládat všechny důležité části aplikace, jako je například připojení a odpojení jednotlivých PLC, správu nástrojů či nastavení tolerancí monitorování. Backend též při detekci překročení tolerance odchylky průběhu lisovacího cyklu od referenční křivky uloží získaná data do databáze pro jejich pozdější zobrazení a analýzu. Kromě vývoje backendu byly v rámci této práce vytvořeny i předpisy pro tvorbu Dockerových obrazů, kombinujících databázi, backend a frontend. Ty poté umožňují běh aplikace na platformě Industrial Edge.

Vyvíjená část kódu byla psána technikou TDD, a proto je více než 80% tříd pokryto unit testy, které při každé kompilaci kontrolují funkčnost programu, což zaručuje jeho vysokou spolehlivost. Kromě testování kódu byla třetí stranou otestována i samotná aplikace v laboratoři společnosti SIEMENS v Německu, viz obrázek 5.1. V průběhu ověřování funkčnosti aplikace s reálným lisem nedošlo k odhalení žádného problému. V rámci diplomové práce tedy byly splněny všechny požadavky.

I přes bezchybnou funkčnost samotného backendu je zde prostor pro budoucí vylepšení. V prvé řadě by bylo dobré v databázi optimalizovat ukládání záznamů o cyklech lisovacího procesu, v rámci nichž došlo k překročení uživatelem nastavené tolerance. Možná řešení byla nastíněna na konci kapitoly „Implementace“. Budoucí pozornost si též zaslouží tvorba Dockerových obrazů. Zde by bylo vhodné, jakmile to bude možné, upravit práci s SSH klíči, jak je rozebráno v kapitole „Tvorba aplikace pro Industrial Edge“. Též pak implementovat CI/CD pipeliny pro automatickou tvorbu Dockerových obrazů a jejich následné nahrání do Industrial Edge Management systému s přidáním kódu do hlavní větve. Pro produkční použití aplikace je též nezbytné, aby bylo implementováno zabezpečení přes OPC UA, zajištující bezpečný přenos dat mezi PLC a backendem.

Již ne tak nutné, ale zajisté užitečné by bylo uživateli prostřednictvím API umožnit specifikaci lokace a názvu data bloku PLC, ze kterého jsou získávána veškerá data. V současné době je toto nutné nastavit před kompilací aplikace v souboru application.properties. Další rozšiřující funkcionalitou by mohlo být umožnění sdílet data o zaznamenaných problémech s ostatními aplikacemi či alespoň jejich stažení.



Obrázek 5.1: Testování aplikace s reálným lisem

Literatura

[1] Jorge Acetozi. *Pro Java clustering and scalability: building real-time apps with Spring, Cassandra, Redis, Websocket and RabbitMQ*. New York, NY: Springer Science+Business Media, 2017. ISBN: 9781484229842.

[2] A Arockiarajan, M Duraiselvam a Ramesh Raju. *Advances in industrial automation and smart manufacturing: select proceedings of ICAIASM 2019*. English. OCLC: 1204136291. 2021. ISBN: 9789811547393. URL: `http://public.eblib.com/choice/PublicFullRecord.aspx?p=6381204` (cit. 11. 03. 2021).

[3] Christian Bauer a Gavin King. *Hibernate in action*. OCLC: ocm56576271. Greenwich, CT: Manning, 2005. ISBN: 9781932394153.

[4] Christian Bauer et al. *Java persistence with Hibernate*. Second edition. OCLC: ocn930029401. Shelter Island, NY: Manning, 2016. ISBN: 9781617290459.

[5] Scott Chacon. *Pro Git*. Second edition. The expert's voice in software development. New York, NY: Apress, 2014. ISBN: 9781484200773.

[6] Jeremy Clark a Paul C Van Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, s. 511–525.

[7] Iuliana Cosmina et al. *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools*. 5th ed. 2017. Berkeley, CA: Apress : Imprint: Apress, 2017. ISBN: 9781484228081.

[8] Chanchal Dey a Sunit Kumar Sen. *Industrial automation technologies*. First edition. Boca Raton, FL: CRC Press, 2020. ISBN: 9780367260422.

[9] Inc. Docker. *Docker Engine 18.09 release notes*. en. Dub. 2021. URL: `https://docs.docker.com/engine/release-notes/18.09/` (cit. 16. 04. 2021).

[10] Shekhar Gulati. *Java unit testing with Junit 5: test driven development with Junit 5*. New York, NY: Springer Science+Business Media, 2017. ISBN: 9781484230145.

[11] Kevin Herron. *eclipse/milo*. original-date: 2016-05-06T13:20:04Z. Dub. 2021. URL: `https://github.com/eclipse/milo` (cit. 23. 04. 2021).

[12] Kevin Herron. *Milo*. en. Text. Ún. 2016. URL: `https://projects.eclipse.org/proposals/milo` (cit. 23. 04. 2021).

[13] Muhammad Ali Imran, Sajid Hussain a Qammer H. Abbasi, ed. *Wireless automation as an enabler for the next industrial revolution*. Hoboken, NJ: Wiley/IEEE Press, 2020. ISBN: 9781119552628 9781119552581.

[14] *index | TIOBE - The Software Quality Company.* URL: https://www.tiobe.com/tiobe-index/ (cit. 06. 03. 2021).

[15] Amrata Joshi. *Netflix adopts Spring Boot as its core Java framework.* en-US. 19. pros. 2018. URL: https://hub.packtpub.com/netflix-adopts-spring-boot-as-its-core-java-framework/ (cit. 28. 04. 2021).

[16] Miłosz Kaczorowski. *Java vs. Kotlin: Which One To Pick While Building An Android App?* en-us. 21. říj. 2020. URL: https://www.ideamotive.co/blog/java-vs-kotlin-which-one-to-pick-while-building-an-android-app (cit. 28. 04. 2021).

[17] Mike Keith. *Pro JPA 2 in Java EE 8: an in-depth guide to Java persistence Apis.* New York, NY: Springer Science+Business Media, 2018. ISBN: 9781484234198.

[18] Vincent van der Leun. *Introduction to JVM languages: Java, Scala, Clojure, Kotlin, and Groovy.* English. OCLC: 1030349325. 2017. ISBN: 9781787126589 9781787127944.

[19] Aliaksandr Liakh. *WebSockets With Spring, Part 1: HTTP and WebSocket.* en. Lis. 2020. URL: https://medium.com/swlh/websockets-with-spring-part-1-http-and-websocket-36c69df1c2ee (cit. 23. 04. 2021).

[20] Aliaksandr Liakh. *WebSockets With Spring, Part 3: STOMP Over WebSocket.* en. Lis. 2020. URL: https://medium.com/swlh/websockets-with-spring-part-3-stomp-over-websocket-3dab4a21f397 (cit. 23. 04. 2021).

[21] Andrew Lombardi. *Websocket.* First edition. OCLC: ocn922938414. Sebastopol, CA: O'Reilly, 2015. ISBN: 9781449369279.

[22] Miguel García López. *Write Fat-free Java Code with Project Lombok.* en. URL: https://www.toptal.com/java/write-fat-free-java-code-project-lombok (cit. 12. 03. 2021).

[23] Stephen Ludin a Javier Garza. *Learning HTTP/2: a practical guide for beginners.* "O'Reilly Media, Inc.", 2017.

[24] Wolfgang Mahnke, Stefan-Helmut Leitner a Matthias Damm. *OPC unified architecture.* OCLC: ocn268784080. Berlin: Springer, 2009. ISBN: 9783540688983 9783540688990.

[25] Ian Miell a Aidan Hobson Sayers. *Docker in practice.* Second edition. OCLC: on1060735743. Shelter Island, New York: Manning Publications, 2019. ISBN: 9781617294808.

[26] *OPCFoundation/UA-Java-Legacy.* original-date: 2015-11-11T15:38:48Z. Dub. 2021. URL: https://github.com/OPCFoundation/UA-Java-Legacy (cit. 23. 04. 2021).

[27] B. Pollard. *HTTP/2 in Action.* Manning Publications, 2019. ISBN: 9781617295164. URL: https://books.google.cz/books?id=Hhd1tgEACAAJ.

[28] Nigel Poulton. *Docker deep dive: zero to Docker in a single book.* English. OCLC: 1176226082. 2020. ISBN: 9781521822807.

[29] Zoltan Raffai. *How does spring work internally?* en-US. 3. čvc 2018. URL: https://www.zoltanraffai.com/blog/how-does-spring-work-internally/ (cit. 28. 04. 2021).

[30] Vinay Sahni. *Best Practices for Designing a Pragmatic RESTful API.* URL: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api (cit. 31. 03. 2021).

[31] Chris Saso. *Containers 101: Starting the Journey from OS Virtualization to Workload Virtualization*. en. Lis. 2017. URL: https://medium.com/@ITsolutions/containers-101-starting-the-journey-from-os-virtualization-to-workload-virtualization-1ce0b32df473 (cit. 28. 02. 2021).

[32] H. Schildt. *Java: The Complete Reference, Eleventh Edition*. McGraw-Hill Education, 2018. ISBN: 9781260440249. URL: https://books.google.cz/books?id=-W57DwAAQBAJ.

[33] Tom Smith. *Is the JVM the Most Important Element of the Java Ecosystem? - DZone Java*. en. 2017. URL: https://dzone.com/articles/is-the-jvm-the-most-important-element-of-the-java (cit. 23. 04. 2021).

[34] *The Good and the Bad of Java Programming*. en-US. URL: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-java-programming/ (cit. 06. 03. 2021).

[35] S.A. Thomas. *HTTP Essentials: Protocols for Secure, Scaleable Web Sites*. Wiley, 2001. ISBN: 9780471398233. URL: https://books.google.cz/books?id=ZQxHAAAAYAAJ.

[36] Mariot Tsitoara. *Beginning Git and GitHub A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer*. English. OCLC: 1156939418. 2020. ISBN: 9781484253137. URL: https://doi.org/10.1007/978-1-4842-5313-7 (cit. 25. 03. 2021).

[37] *Unified Architecture*. en-US. URL: https://opcfoundation.org/about/opc-technologies/opc-ua/ (cit. 07. 03. 2021).

[38] Raoul-Gabriel Urma, Mario Fusco a Alan Mycroft. *Modern Java in action: lambda, streams, functional and reactive programming*. OCLC: on1064559289. Shelter Island: Manning Publications, 2018. ISBN: 9781617293566.

[39] Balaji Varanasi. *Introducing Maven: a build tool for today's Java developers*. English. OCLC: 1126540141. 2019. ISBN: 9781484254103 9781484254097 9781484254110. URL: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=2286175 (cit. 12. 03. 2021).

[40] Bogdan M. Wilamowski a J. David Irwin, ed. *The industrial electronics handbook*. eng. 2. ed. The electrical engineering handbook series. Boca Raton, Fla.: CRC Press, 2011. ISBN: 9781439802892.

[41] Clinton Wong. *HTTP pocket reference*. 1st ed. Sebastopol, CA: O'Reilly, 2000. ISBN: 9781565928626.

Příloha A

Obsah přiloženého disku

- `metalForming`
  - `src`
    - `main` - *obsahuje zdrojové kódy backendu*
    - `test` - *obsahuje zdrojové kódy testů backendu*
  - `Dockerfile` - *předpis pro tvorbu Dockerového obrazu*
  - `docker-compose.yml` - *předpis pro vytvoření kompletní aplikace*
  - `README.md` - *popis jak zacházet s aplikací*
  - `pom.xml` - *obsahuje důležité informace pro Maven*
- `metalForming.jar` - *spustitelná verze backendu*
- `thesis.pdf` - *text diplomové práce*