FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Rozšíření systému pro sběr dat z OPC UA serverů
Student:	Bc. Dominik Codl
Vedoucí:	Ing. Ladislav Šťastný, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem diplomové práce je navázat na bakalářskou práci "Systém pro sběr dat s využitím OPC UA". Na základě nových informací a požadavků bude upraven návrh systému. Komponenta pro podporu OPC UA bude z konzolové aplikace přetvořena do serveru s REST API a budou doplněny nové funkcionality. Rovněž se o nové prvky rozšíří knihovna Scalable OPC UA. Vzhledem k tomu, že software je implementovaný v jazyce Scala a s ním spjatými technologiemi, práce rovněž bude obsahovat představení těchto technologií.

Postup:
1. Představte použité technologie.
2. Popište dosavadní stav vývoje a navrhněte úpravu systému podle nových požadavků.
3. Implementujte nové funkcionality komponenty pro OPC UA a knihovny Scalable OPC UA.
4. Zhodnoťte výsledek a navrhněte možná vylepšení do budoucna.

Literatura:
CODL, D.: Systém pro sběr dat s využitím OPC UA. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.
MAHNKE, W.; LEITNER, S.-H.; DAMM, M.: OPC Unified Architecture. Berlin: Springer-Verlag Berlin Heidelberg, 2009, ISBN 978-3-540-68898-3.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 31. října 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Rozšíření systému pro sběr dat z OPC UA serverů

Bc. Dominik Codl

Katedra softwarového inženýrství
Vedoucí práce: Ing. Ladislav Šťastný, Ph.D.

29. dubna 2024

Poděkování

Rád bych poděkoval své rodině a blízkým za podporu a motivaci během mých studií, jenž postupně dospěly do stádia napsání této závěrečné práce. Taktéž bych rád poděkoval Ing. Ladislavovi Šťastnému, PhD. za odborné konzultace a vedení při tvorbě diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 29. dubna 2024 . . . . . .

Odkaz na tuto práci

Codl, Dominik. *Rozšíření systému pro sběr dat z OPC UA serverů.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Abstrakt

Diplomová práce se zaměřuje na pokračování vývoje systému pro sběr dat společnosti ModemTec. Práce analyzuje výchozí řešení a nabízí nový návrh či úpravy vzhledem k novým požadavkům a doméně. Ze systémových komponent je implementován adaptér pro podporu OPC UA protokolu. Současně jsou přidány funkcionality do knihovny Scalable OPC UA. Pro kód komponent je vybrán jazyk Scala, jehož frameworky a knihovny jsou v práci představeny.

Klíčová slova OPC UA, Scala, REST, IoT, funkcionální paradigma

Abstract

The diploma thesis is dedicated to the next development of the data collection system for the company ModemTec. The thesis analyzes the original solution and offers a new design or modifications due to the new requirements and domain. An OPC UA adapter is implemented from system components. At the same time, functionalities are added to the Scalable OPC UA library. The Scala language is chosen for the component code, whose frameworks and libraries are presented in the thesis.

Keywords OPC UA, Scala, REST, IoT, functional paradigm

Obsah

Úvod **1**

1 Cíl práce **3**

2 Použité technologie **5**
2.1 Cats . . . 5
2.1.1 Type Classes . . . 6
2.1.2 Monads . . . 7
2.1.3 Monad Transformers . . . 8
2.1.4 Další konstrukty a využití . . . 9
2.2 Pekko/Akka . . . 9
2.2.1 Actors . . . 10
2.2.2 Streams . . . 12
2.3 Slick . . . 13
2.3.1 Databázové schéma . . . 14
2.3.2 Vložení dat . . . 15
2.3.3 Dotazování . . . 15
2.3.4 Kombinování akcí . . . 16
2.4 Play Framework . . . 17
2.4.1 Serializace objektů . . . 17
2.4.2 Guice a injektáž . . . 18
2.4.3 Tvorba kontroléru . . . 19

3 Návrh systému **21**
3.1 Výchozí stav . . . 22
3.2 Doména . . . 23
3.2.1 Procesy . . . 24
3.3 Komponenty . . . 25
3.3.1 Adapter . . . 25
3.3.2 Data Store . . . 26

3.3.3 Diagnostic Tool . . . 26
3.4 Rozdělení do projektů . . . 26
3.5 Integrace komponent . . . 27
3.6 Funkční požadavky na API komponent . . . 28
3.6.1 Adapter API . . . 28
3.6.2 Data Store API . . . 28
3.7 Uživatelské rozhraní . . . 29
3.8 Zvolené technologie . . . 30
3.9 Nasazení systému . . . 30

4 Scalable OPC UA **31**
4.1 Výchozí stav . . . 31
4.1.1 Jazyk . . . 31
4.1.2 Funkcionality . . . 31
4.1.3 Klient . . . 32
4.2 Úpravy . . . 33
4.2.1 Jazyk . . . 33
4.2.2 Funkcionality . . . 33
4.2.3 Klient . . . 34
4.3 Testování . . . 36
4.4 Ukázky použití . . . 36
4.4.1 Vyčtení hodnoty . . . 37
4.4.2 Zavolání metody . . . 38
4.4.3 Vyčtení historie . . . 38

5 Adapter API **41**
5.1 Návrh . . . 41
5.1.1 Verzování . . . 41
5.1.2 Specifikace zdroje . . . 42
5.1.3 Operace . . . 44
5.1.4 Chybové stavy . . . 47
5.1.5 Objekty . . . 48
5.1.6 Serializace objektů . . . 54
5.1.7 Bezpečnost . . . 54
5.2 Knihovna ve Scale . . . 54
5.2.1 Návratové hodnoty a monády . . . 55
5.2.2 Ukázky použití klienta . . . 56

6 OPC UA Adapter **61**
6.1 Výchozí stav . . . 61
6.1.1 Datový model . . . 61
6.1.2 Architektura . . . 62
6.1.3 Implementace . . . 63
6.2 Důsledek a požadavky pro OPC UA Adapter . . . 64

6.3 Doménový model 64
6.4 Architektura 66
6.4.1 Prezentační vrstva 68
6.4.2 Byznys vrstva 68
6.4.3 Datová vrstva 68
6.4.4 Cross-cutting concerns 70
6.5 Implementace 70
6.6 Testování 70
6.7 Nasazení 72

7 Další vývoj a možné vylepšení **73**
7.1 Dokumentace 73
7.2 Uložení hesel a certifikátů 73
7.3 Optimalizace datových typů v paměti 73
7.4 UI a zbylé komponenty systému 74

Závěr **75**

Literatura **77**

A Seznam použitých zkratek **81**

B Obsah přiloženého média **83**

Seznam obrázků

2.1 Vizualizace příkladu 10

3.1 Výchozí návrh komponent 22
3.2 Vizualizace systému 23
3.3 Komponenty systému 25

4.1 Struktura balíčků knihovny (UML) 32

5.1 Model specifikace zdroje 42

6.1 Databázové schéma původní aplikace 62
6.2 Původní architektura aplikace (UML) 63
6.3 Doménový model (UML) 65
6.4 Architektura OPC UA Adapter (UML) 67
6.5 Schéma relační databáze (UML) 69

Seznam tabulek

5.1 Atributy AdapterInfo 49
5.2 Atributy ResourceConfig 49
5.3 Atributy AuthConfig 50
5.4 Výčet SecurityPolicy 50
5.5 Atributy ReadResult 50
5.6 Atributy WriteRequest 50
5.7 Atributy WriteResult 51
5.8 Atributy CallMethodRequest 51
5.9 Atributy CallMethodResult 51
5.10 Atributy HistoryReadRequest 51
5.11 Atributy HistoryReadResult 52
5.12 Atributy ResourceSpec 52
5.13 Atributy NodeSpec 52
5.14 Atributy VariableSpec 52
5.15 Atributy MethodSpec 53
5.16 Atributy SchemaSpec 53
5.17 Atributy AdapterError 53
5.18 Atributy CertData 53

Úvod

V současné době již nejsou Big Data pouze data, které vznikají lidskou činností, například nákupy, logistika, byrokracie, ale jsou to i data, jenž jsou tvořena různými výrobky, měřiči, či roboty. IoT v průmyslu nebo v každodenně užívané elektronice hraje nezastupitelnou roli. Tato diplomová práce se soustředí na sběr dat z IoT zařízení společnosti ModemTec, která se specializuje na diagnostiku elektrických sítí. Práce navazuje na bakalářskou práci, ve které se položily základy řešení systému sběru dat. Tato práce výchozí řešení upravuje a rozvíjí do podoby, kdy je vytvořen nový návrh systému a implementována komponenta pro OPC UA jako server s pečlivě navrženým REST API. Text práce se skládá z následujících kapitol.

První kapitola představuje cíl práce, který je rozčleněn do dílčích cílů. Každý z nich obsahuje podrobnější popis a případnou motivaci.

Druhá kapitola popisuje použité technologie ze světa jazyka Scala. Text podává základy a principy knihovny Cats, Slick, Akka a frameworku Play. Rovněž během výkladu jsou vysvětleny některé návrhové vzory používané ve funkcionálním programování.

Třetí kapitola se věnuje vykreslení výchozího stavu systému a poskytuje upravený návrh. Vychází z analýzy domény a procesů. Dále sbírá funkční požadavky na jednotlivé komponenty. Nastiňuje integraci a vybrané technologie.

Čtvrtá kapitola popisuje výchozí funkcionality knihovny Scalable OPC UA a následně představuje funkcionality nově přidané. Taktéž zobrazuje ukázky kódu, kde jsou vybrané nové funkce demonstrovány.

Pátá kapitola se zaměřuje na definice Adapter API, jenž je společným rozhraním pro komponenty, jenž vyčítají data pomocí různých IoT protokolů. Krom popisu objektů a operací, které API dodává, také nastíní implementaci klienta pro toto API. A nakonec na vybraných ukázkách ukazuje použití klienta.

Šestá kapitola se týká vývoje komponenty OPC UA Adapter, která imple-

mentuje Adapter API pro protokol OPC UA. Popisuje výchozí stav a následně se vrhá na nový návrh. Po návrhu následuje implementace a nasazení.

Sedmá kapitola se závěrem se věnují myšlenkám na možné úpravy systému. Dále nabízejí další možné kroky vývoje. V závěru jsou shrnuty výsledky této diplomové práce.

Kapitola 1

Cíl práce

Diplomová práce pokračuje ve vývoji systému pro sběr dat, který započala bakalářská práce „Systém pro sběr dat s využitím OPC UA“. Systém je vyhotoven pro společnost ModemTec, která působí na poli diagnostických řešení elektrického vedení jak nízkonapěťového, tak vysokonapěťového.[1] Cílem práce je rozvinout systém ze stavu, kdy se jedná CLI aplikaci s databází, jenž vyčítá data z OPC UA serverů, do stavu, kdy se jedná o plnohodnotnou komponentu, jenž může být zařazena do celkového diagnostického řešení. Výše zmíněné lze rozčlenit do čtyř dílčích cílů:

- Prvním dílčím cílem je představení použitých technologií naprogramovaných ve Scale. Byť Scala je jazykem používaným hojně ve zpracování Big Data, nedosahuje velikostí své komunity na mainstreamové jazyky, jakými jsou Java, C# nebo JavaScript. Tento bod tedy cílí na seznámení se základními informacemi a principy, kde je znát moderní využití funkcionálního a objektového paradigma.
- Druhým dílčím cílem je popis výchozího stavu vývoje, zvážení nových požadavků na systém a upravení návrh systému. Je tedy cíleno na analýzu domény s procesy, které nyní zahrnují sběr dat ze zařízení, které poskytují rozhraní i v jiných protokolech než OPC UA, a dle tohoto sestavit návrh řešení a zvolit technologie.
- Třetí dílčím cílem je implementace komponenty pro podporu OPC UA protokolu a přidání funkcionalit do knihovny Scalable OPC UA, jenž je komponentou využívána. Důraz je kladen na popis návrhu a API. Na případných ukázkách kódu je demonstrováno použití jazyka Scala v praxi.
- Čtvrtým dílčím cílem a závěrem práce je seskupení myšlenek ohledně dalšího vývoje a provedení zhodnocení výsledku.

Kapitola 2

Použité technologie

Kapitola představuje technologie, které jsou použity při implementaci řešení v jazyce Scala. K práci se rovněž hodí mít přehled o protokolu OPC UA, ten je však popsán v bakalářské práci „Systém pro sběr dat s využitím OPC UA“ v kapitole 3 „Komunikační standard OPC UA“.[2] Jak se ukazuje v praxi, Scala a k ní přilehlé technologie, knihovny a frameworky mívají obvykle pomalu rostoucí učící křivku. Autor předpokládá, že to je v důsledku vyšších nároků kladených na vývojáře. Kombinace objektově-orientovaného paradigma s funkcionálním sice vytváří platformu pro vysokou expresivitu jazyka, vhodnost použití ve vícevláknových programech nebo možnost přirozeného užití doménově specifických jazyků, ale daní budiž, že vývojář musí rovněž zvládat více abstraktnější způsob myšlení a funkcionální programování není vždy zařazeno ve vysokoškolské výuce. Tato kapitola si klade za cíl poskytnout úvod do vybraných projektů.

2.1 Cats

Knihovna Cats přidává do Scaly prvky funkcionálního programování, které se nevyskytují ve standardní knihovně. Název Cats vychází ze zkrácení slov „Category theory“ alias hezky česky Teorie kategorií, jenž se zabývá matematickými strukturami a jejich vzájemnými vztahy. Tato teorie se využívá ve vícero oblastech informatiky, zejména však ve funkcionálním programování a sémantice programovacích jazyků. Často lze narazit na zkoumání kategorií, objektů, morfismů, funktorů či aplikací transformací na matematické objekty.[3]

Dalším jazykem krom Scaly, kde se této teorie využívá, je Haskell, který taktéž podporuje funkcionální paradigma. Knihovna Cats v mnoha částech vychází z Haskellu. Tento text se zaměřuje na čtyři oblasti, jenž autor považuje za nedílnou součást znalostí Scala vývojáře: typové třídy (*Type Class*), monády (*Monad*) a jejich transformery (*Monad Transformer*). Ostatně taktéž zbylé technologie budou často využívat stejné principy, a tak knihovna Cats může

mimo jiné sloužit i jako ukázka implementace částí Teorie kategorií.[3] Scala 3 ekvivalenty k pojmům jsou následující:

- trait: type class;
- implicitní hodnoty (viz *given*): type class instance;
- implicitní parametry (viz *using*): použití type class;
- implicitní třídy (viz *extension*): pomocné utility pro práci s typovými třídami, respektive rozšíření tříd.

Pro úplnost: knihovna Cats byla vytvořena primárně pro Scalu 2, současná verze (2.10.0) je kompatibilní se Scalou 3. A tak i příklady kódu, které budou následovat jsou napsány ve Scale 3.[4]

2.1.1 Type Classes

Typová třída představuje rozhraní, respektive množinu funkcionalit, která je očekávána k implementaci. Níže definice type class alias trait pro serializaci hodnoty typu A do JSON formátu.[3]

```
trait JsonWrite[A]:
  def write(value: A): Json
```

Výpis kódu 2.1: Příklad typové třídy

Instance typové třídy poté poskytuje implementaci tohoto rozhraní. Instance jsou vnímány (a taktéž programovány) jako implicitní hodnoty.[3]

```
case class Person(name: String, email: String)

object JsonWriterInstances:

  given stringWriter: JsonWriter[String] with
    def write(value: String): Json = JsString(value)

  given personWriter: JsonWriter[Person] with
    def write(value: Person): Json =
      JsObject(Map(
        "name" -> JsString(value.name),
        "email" -> JsString(value.email)
        ))

  // etc...
```

Výpis kódu 2.2: Příklad instance typové třídy

Instance typové třídy je použita skrze implicitní parametry. Zatímco ve Scale 2 se čtenář setká s doslova *implicit* parametry v samostatném seznamu argumentů funkce, ve Scale 3 se používá pojem kontext.[4]

```
def serialize[A](value: A)(using writer: JsonWriter[A]): A = ...

val person = Person("Pepa", "pepa@mail.cz")
serialize(person)(using JsonWriterInstances.personWriter)
```

Výpis kódu 2.3: Příklad použití instance typové třídy

Knihovna Cats poskytuje plno defaultních typových tříd pro datové typy jazyka Scala a jejich instance, například *List*, *Option* nebo *Int*.[3]

```
package cats

trait Show[A]:
  def show(value: A)

...

import cats.instances.int.given Show[A]  // for Show[A]
```

Výpis kódu 2.4: Příklad importu defaultní instance typové třídy

2.1.2 Monads

Ačkoliv monáda je velmi abstraktní pojem a budiž typickými příklady *Future[A]*, *Option[A]*, ale taky i *List[A]* nebo *Map[A, B]*, vývojář si pro drtivou většinu případů vystačí s pouhým: monáda je návrhový vzor na tvorbu sekvence výpočetních operací. Dvěmi důležitými metodami v rozhraní monády *Monad[F[A]]* je metoda *flatMap* a metoda *pure*. Metoda *flatMap* vezme mezivýsledek typu A a aplikuje na něj funkci *func*, jenž vrací výsledek typu *F[B]*. Metoda *pure* je konstruktor pro nový monadický kontext pro čistou hodnotu typu A.[3]

```
trait Monad[F[A]]:
  def pure[A](value: A): F[A]
  def flatMap[A, B](value: F[A])(func: A => F[B]): F[B]
```

Výpis kódu 2.5: Obvyklé rozhraní monády

Pro metody *pure* a *flatMap* by měly platit následující Monad Laws[3]:

1. *Levá identita*

   Volání *pure* a následná transformace skrze *flatMap* s funkcí *func* je ekvivalentní volání funkce *func*.

```
pure(a).flatMap(func) == func(a)
```

Výpis kódu 2.6: Levá identita monády

2. *Pravá identita*

   Použití *pure* v rámci *flatMap* je ekvivalentní jako neudělat nic.

```
m.flatMap(pure) == m
```

Výpis kódu 2.7: Pravá identita monády

3. *Asociativita*

   Aplikace *flatMap* s funkcí f a následně *flatMap* s funkcí g je ekvivalentní aplikace *flatMap* s funkcí f a vnořenou aplikací *flatMap* s funkcí g.

```
m.flatMap(f).flatMap(g) == m.flatMap(x => f(x).flatMap(g))
```

Výpis kódu 2.8: Asociativita monády

Jazyk Scala nabízí konstrukt tzv. for comprehension, kdy za pomoci řídící struktury for-yield programátor může zřetězit operace *flatMap* a toto zřetězení zapsat imperativně. Při komplexnějším použití monád a delších sekvencích toto tzv. syntactic-sugar může zlepšit čitelnost kódu.[5]

```
for
  x <- Some(10)
  y <- Some(20)
yield x + y
```

Výpis kódu 2.9: Imperativní zápis sekvence flatMap

Za zmínku dále stojí, že v případě, kdy dojde k chybě, například v kódu výše by na jednom řádku byla hodnota *None*, je tato hodnota obvykle vrácena jako výsledek. Pod pojmem „obvykle“ se schovává fakt, že často je monáda implementována tak, aby se sekvence operací přerušila. Nicméně jsou i monády, které naopak tento vedlejší efekt (chybu, kód, či jinou hodnotu) akumulují. Příkladem budiž validace formuláře, kdy uživatel chce nasbírat všechny chybně vyplněné položky.[3]

2.1.3 Monad Transformers

Co ovšem může v kódu nastat při použití monád, je jejich vnořování. Příkladem budiž, že metoda vrací návratovou hodnotu s typem *Future[Either[A, B]]*. V tomto okamžiku při zpracování vnořených monád se může začít v kódu vyskytovat čím dál více vnořených for comprehension nebo *flatMap*. Monad

Transformers nabízí cestu ven skrze zabalení do speciálních monád, které poskytnout přímo práci s hodnotou. Pro zmíněné *Future[Either[A, B]]* knihovna Cats nabízí *EitherT[Future, A, B]*. Obsahuje však i plno jiných.[3]

2.1.4 Další konstrukty a využití

Utility knihovny Cats se využívají i pro tvorbu kombinátorů, které se hodí při implementaci různých validací, parsování nebo doménově specifického jazyka. Další významnou oblastí jsou efekty (*Effects*). Cats rozlišuje definice effects a side-effects. Side-effects je něco, co se stane mimo jiné během výpočetních operací, například logování nebo změna vnitřního stavu objektu, zatímco effects je popis side-effects. Uživatel tak získává kontrolu nad vedlejšími efekty. Může je paralelizovat, může rozhodnout, že se vůbec nevykonají, či je může transformovat. Doposud popsané konstrukty umožňuje objektově-orientované programování ztvárnit pomocí návrhového vzoru *Template Method*. Avšak s rostoucí komplexitou řešení, respektive s rostoucím množství kroků, vzniká nespočet tříd. Funkcionální programování oproti tomu využije snadné dosazení funkcí, potažmo anonymních funkcí, a lze tak celý návrh operací popsat ad hoc a zacházet s ním, jako s hodnotou, kterou lze někam předat nebo uložit.[3]

2.2 Pekko/Akka

Projekt Akka původem vychází ze standardní knihovny Scaly, která implementovala Actor model. S postupem času se knihovna rozvíjela o další funkcionality, až se nakonec vyčlenila jako samostatný open-source software pod společností Lightbend. Dnes Akka poskytuje nejen řešení paralelizace přes Actor model, ale taktéž i řešení distribuovaného prostředí, cluster, streamování, knihovny pro komunikaci přes HTTP či gRPC protokoly, ba i samotnou implementaci mikroslužeb. Framework dodává jak open-source knihovny, tak enterprise edici a služby.[6]

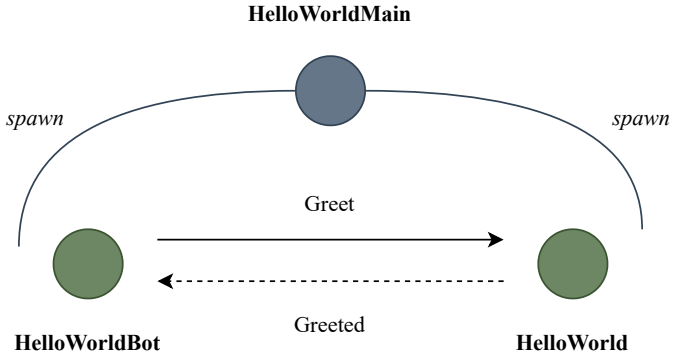
V názvu této sekce je však ještě slíben projekt Pekko, který vznikl v důsledku přechodu Akka pod licenci BSL (*Business Source License*) v1.1. Tato licence komercializuje aktuální verzi Akka a firmy s určitým finančním příjmem musí za tuto licenci platit, nebo pokud se jedná o komerční projekt, ke kterému Akka bude využita, minimálně požádat o obdržení licence. Licence však má vždy pro danou verzi omezenou platnost na 3 roky, a tak po uplynutí této doby daná verze přejde pod licenci Apache 2.0 a může tak být bez omezení použita širokou komunitou.[7] Projekt Pekko reagoval na změnu v licenci přijetím Akka verze 2.6.x a její vlastním rozvíjením. Pekko získalo sponzory pro rozvoj a některé projekty velmi rozšířené v komunitě, např. Play Framework, přešly z Akka na Pekko. Vzhledem k fork Akka v 2.6.x stačí pouze v kódu změnit import tříd z *akka* na import z *pekko*.[8]

Tento text se zaměří na přiblížení dvou fundamentálních částí knihovny a to aktory (*Actors*) a proudy (*Streams*). Ukázky kódu pocházejí od Akka, ale pro Pekko jsou takřka ekvivalentní.

2.2.1 Actors

Model aktorů vychází z principů objektového modelu – aktoři jsou objekty, které spolu asynchronně komunikují pomocí zpráv. Aktoři současně mohou odesílat konečný počet zpráv, vytvářet konečný počet aktorů a přijímat zprávy, na základě kterých mohou změnit své chování pro další zprávy.[9]

Následující příklad demonstruje použití aktorů na obligátním „Hello World“ programu. Aktor reprezentující hlavní program *HelloWorldMain* vytvoří dva aktory *HelloWorldBot* a *HelloWorld*. Ti si mezi sebou vyměňují zprávy *Greet*, pozdrav od bota, a *Greeted*, odpověď na pozdrav.[9] Řečenou situaci zobrazuje obrázek 2.1.



Obrázek 2.1: Vizualizace příkladu

Nejprve je definován aktor *HelloWorld* s pozdravy. *HelloWorld* obsahuje chování (*Behaviour*) pro zprávu typu *Greet*, jenž přidá do logu přijatou zprávu a následně odpoví zprávou *Greeted*. Po odeslání odpovědi zůstane chování stejné jako doposud (*Behaviors.same*).[9]

```
object HelloWorld:
  case class Greet(whom: String, replyTo: ActorRef[Greeted])
  case class Greeted(whom: String, from: ActorRef[Greet])

  def apply: Behavior[Greet] = Behaviors.receive: (context, message) =>
    context.log.info("Hello {}!", message.whom)
    message.replyTo ! Greeted(message.whom, context.self)
    Behaviors.same
```

Výpis kódu 2.10: Definice aktora HelloWorld

Definice aktora *HelloWorldBot* již demonstruje změny v chování. Realizace vychází z funkcionálního paradigma, kdy každé chování, respektive stav, aktora zůstává immutable. Aktor tak při změně přechází z jednoho chování/stavu, do druhého chování/stavu. Z principu je takovýto přístup thread-safe a programátor ví, že žádné jiné vlákno mu nezmění obsah proměnných, se kterými zachází. Aktor *HelloWorldBot* v kódu níže reaguje na přijatou zprávu typu *Greeted* inkrementací počítadla a v případě dosáhnutí maxima *max* se zastaví (*Behaviors.stopped*), jinak vyšle pozdrav.[9]

```
object HelloWorldBot:

  def apply(max: Int): Behavior[HelloWorld.Greeted] = bot(0, max)

  def bot(greetingCounter: Int, max: Int): Behavior[HelloWorld.Greeted] =
    Behaviors.receive: (context, message) =>
      val n = greetingCounter + 1
      context.log.info2("Greeting {} for {}", n, message.whom)
      if n == max then
        Behaviors.stopped
      else
        message.from ! HelloWorld.Greet(message.whom, context.self)
        bot(n, max)
```

Výpis kódu 2.11: Definice aktora HelloWorldBot

Následuje definice aktora *HelloWorldMain*. Metoda *Behaviors.setup* je provedena při vytvoření aktora a v tomto případě vytváří aktora/potomka *HelloWorld*. Na obdržení zprávy reaguje vytvořením aktora *HelloWorldBot*, který si vymění tři zprávy s *HelloWorld*.[9]

```
object HelloWorldMain:

  case class SayHello(name: String)

  def apply: Behavior[SayHello] =
    Behaviors.setup: context =>
      val greeter = context.spawn(HelloWorld(), "greeter")

      Behaviors.receiveMessage: message =>
        val replyTo = context.spawn(HelloWorldBot(max = 3), message.name)
        greeter ! HelloWorld.Greet(message.name, replyTo)
        Behaviors.same
```

Výpis kódu 2.12: Definice aktora HelloWorldMain

A konečně je celý systém aktorů spuštěn, jak demonstruje kód níže.

```
val system: ActorSystem[HelloWorldMain.SayHello] =
  ActorSystem(HelloWorldMain(), "hello")

system ! HelloWorldMain.SayHello("World")
system ! HelloWorldMain.SayHello("Akka")
```

Výpis kódu 2.13: Spuštění příkladového systému aktorů

Pro aktory existuje celá škála funkcí a návrhových vzorů poskytovaná knihovnou Akka. Utility se pohybují od interakčních vzorů pro komunikaci přes routování po životní cyklus aktora.[9]

2.2.2 Streams

Akka pomocí aktorů implementuje reaktivní streamy a krom jejich standardizovaného API, poskytuje i API vlastní, které více odpovídá funkcionálnímu programování. Pomocí monád uživatel vytváří řetězec výpočetních operací, ba dokonce obecně graf, které zpracovávají posloupnost entit. Entity jsou zpracovány paralelně a nezávisle na sobě.[10] Než funkce knihovny budou demonstrovány na ukázce, dojde na přehled základních pojmů, na které může čtenář narazit při dalším studiu Akka Streams.

- *Stream*

  Proces zajišťující pohyb a zpracování dat.

- *Element*

  Jednotka zpracovávaných dat nebo-li entita.

- *Back-pressure*

  Způsob řízení toku dat, kdy konzumenti informují producenty o rychlosti zpracování dat a případně zvýší/sníží výkon producentů.

- *Graph*

  Popis topologie streamů.

- *Operator*

  Označení pro jakýkoliv stavební blok grafu – od funkcí pro transformaci dat po sjednocení/rozdělení grafu. Může i nemusí mít vstupy/výstupy.

- *Source*

  Operátor s pouze jedním výstupem – posílá data dále do streamu, kdykoliv konzument jej může přijímat.

- *Sink*

  Operátor s pouze jedním vstupem – žádá a přijímá data. Může snížit výkon producenta.

- *Flow*

  Operátor s s jedním vstupem a s jedním výstupem – spojuje dvě části streamu a případně transformuje data.

- *Runnable Graph*

  Stream s přiřazeným začátkem (*Source*) a koncem (*Sink*) je spustitelný graf. Na *RunnableGraph* lze pohlížet jako na popis grafu, který musí být teprve materializován. Proces materializace zahrnuje si vzít popis grafu a alokovat pro něj výpočetní prostředky, nad kterými je spuštěn. Typicky tvorba aktorů.[10]

Na příkladu sumy níže je ukázán vznik *Source* z definice seznamu, jenž bude naplňovat stream daty. Ty potečou do *Sink*. Akka nabízí celou řadu funkcí pro transformaci v operátorech. Zde namísto tvorby *Flow* s operací sčítání je sumarizace rovnou provedena v *Sink*. Tím je vytvořen *RunnableGraph*, jenž je následně spuštěn dvakrát v jednom okamžiku. Tudíž vzniknou dvě různé instance monády *Future[Int]* v pravděpodobně různých časech se stejným výsledkem.[10]

```
val sink = Sink.fold[Int, Int](0)(_ + _)
val runnable: RunnableGraph[Future[Int]] =
  Source(1 to 10).toMat(sink)(Keep.right)

val sum1: Future[Int] = runnable.run()
val sum2: Future[Int] = runnable.run()
```

Výpis kódu 2.14: Příklad streamu pro sumu čísel

2.3 Slick

S rozvojem databází, zejména relačních, přišla i potřeba práce s databázemi v rámci aplikace. Navíc se rozšířilo objektově-orientované programování. A tak vzniklo ORM, tedy objektově-relační mapování, které cílí na mapování objektů na relační tabulky. K tomu typicky frameworky přidávají funkcionality na enkapsulaci databázových dotazů. Oproti tomu knihovna Slick představuje FRM, tedy funkcionálně-relační mapování. Zakládá si na třech oblastech[11]:

- Využití DSL pro modelování databázového schéma.
- Pohled na tabulku jako na kolekci řádků včetně typických operací pro kolekce: filtrování, mapování, třídění atp.
- Monadický pohled na dotazy nad databází, tj. sekvence dotazů, resp. transakce, je akce skládající se z dílčích akcí.

Rozdílem oproti běžným ORM implementovaných v Jave je asynchronicita – použití *Future[A]* pro návratovou hodnotu. Dalšími významnými konstrukcemi je možnost streamování a použití efektů. Díky standardizaci API pro reaktivní streamy si lze vybrat pro streamování knihovnu dle svého uvážení. Autor tohoto textu využívá vždycky Akka (příp. Pekko) Streams. Slick rovněž podporuje evoluci databázového schéma a generování kódu. Nadcházející části představí pár základních příkladů užití.[12]

2.3.1 Databázové schéma

Na učebnicovém příkladu níže je demonstrováno možné modelování dat. Řádek tabulky je reprezentován třídou *Message*, tabulka samotná třídou *MessageTable*. Definice tabulky obsahuje sloupečky s názvy a datovými typy – Slick rovněž podporuje označení klíčů včetně autoinkrementace. Dalším zajímavým aspektem při používání Slick je využití compile-time dependency injection, které lze realizovat pomocí typových tříd/traits a Cake Pattern. Zmíněný návrhový vzor, jenž je primárně zkonstruován pro Scalu, představil Martin Odersky ve svém článku „Scalable Component Abstractions“.[11]

```
trait DatabaseSchema:
  self: HasDatabaseConfigProvider[JdbcProfile] =>
  import profile.api.*

  case class Message(
    sender:  String,
    content: String,
    id:      Long = 0L
  )

  class MessageTable(tag: Tag) extends Table[Message](tag, "message"):
    def id      = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def sender  = column[String]("sender")
    def content = column[String]("content")
    def * = (sender, content, id)<>((Message.apply _).tupled, Message.unapply)

  lazy val messages = TableQuery[MessageTable]
```

Výpis kódu 2.15: Příklad modelování schéma

Dále se hodí si vytvořit pomocnou funkci pro spuštění dotazů nad databází. Samotnou konfiguraci připojení k databázi pro odpovídající stroj je ponecháno laskavému čtenáři. Stojí však za zmínku, že Slick podporuje tvorbu vlastních rozšíření funkcionalit JDBC profilů, např. existuje rozšíření pro PostgreSQL, které podporuje datový typ jsonb, utility pro PostGIS nebo práci s časem.[11]

```
trait QueryExecutor:
  self: HasDatabaseConfigProvider[JdbcProfile] =>
  import profile.api.*

  def exec[A](action: DBIO[A]): Future[A] = db.run(action)
```

Výpis kódu 2.16: Spuštení dotazu

2.3.2 Vložení dat

Vložení dat lze po jednotlivých řádcích nebo najednou. V obou případech je možné vytvořit jednu akci, která se spustí jen tak nebo jako transakce. Ačkoliv Slick podporuje streamování dat z databáze, pro cestu tam existuje jen „standardní“ insert.[11]

```
val insert = messages ++= Seq(
  Message("Dave", "Hello, HAL. Do you read me, HAL?"),
  Message("HAL", "Affirmative, Dave. I read you."),
  Message("Dave", "Open the pod bay doors, HAL."),
  Message("HAL", "I'm sorry, Dave. I'm afraid I can't do that."))

exec(insert)
```

Výpis kódu 2.17: Příklad vložení kolekce řádků

2.3.3 Dotazování

Samotné dotazy se snaží přiblížit pohledu na tabulku jako na kolekci řádků. Odtud existují očekávané funkce pro filtraci, mapování, agregaci atp. Taktéž Slick dovoluje rozšířit API o vlastní funkce, např. chceme-li dodat operátor, co není ve standardní výbavě SQL. Joiny jde však ztvárnit přístupem připomínající SQL (viz první příklad), nebo monadicky (viz druhý příklad).[11] Výhoda monadického přístupu je subjektivně čistý kód a využití for comprehension. Na druhou stranu SQL-like přístup může být pro náhodného čtenáře kódu více srozumitelnější.

```
val select = messages.join(users).on(_.senderId === _.id)
```

Výpis kódu 2.18: Příklad join (SQL-like)

```
val select = for
  msg <- messages
  usr <- users if usr.id === msg.senderId
yield (usr.name, msg.content)
```

Výpis kódu 2.19: Příklad join (monadicky)

2.3.4 Kombinování akcí

Poslední příklad je převzat z implementace OPC UA Adapter. Jak bylo řečeno, akce se dají kombinovat do větších akcí či transformovat. Bližší informace k implementaci monád a kombinátorů může čtenář nalézt v části věnované knihovně Cats. Slick poskytuje k I/O akcím alias *DBIO[+A]*, za kterým se schovává *DBIOAction[A, NoStream, All]*. Tedy akce bez streamování a efektů, jelikož obě tyto funkce Slick podporuje. Kód níže demonstruje implementaci funkce pro vyčtení identifikátoru v OPC UA *UaId* pro daný uzel s daným databázovým *id* a funkci pro vyčtení obecného uzlu. Nakonec je definována funkce pro vyčtení uzlu typu objekt, ve které lze nahlédnout využití for comprehension konstrukce pro složení sekvenční posloupnosti vykonaných akcí do jedné.[11]

```
def readNodeId(id: Long): DBIO[UaId] =
  val rows = for
    ident <- Ids if ident.id === id
    ns    <- Namespaces if ns.id === ident.namespaceId
  yield (ident.index, ns.uri)

  rows.result.head.map: (index, uri) =>
    UaId(index, uri)


def readCommonNode(nodeId: UaId): DBIO[NodeRow] =
  val rows = for
    ident <- Ids        if ident.index === nodeId.index.toInt
    ns    <- Namespaces if ident.namespaceId === ns.id &&
                           ns.uri === nodeId.namespaceUri.value
    node  <- Nodes      if node.id === ident.id
  yield node

  rows.result.head


def readObject(nodeId: UaId): Future[UaObjectNode] =
  val selection = for
    n      <- readCommonNode(nodeId)
    parent <- readNodeId(node.parentId)
  yield UaObjectNode(nodeId, parent, n.browseName, n.displayName, n.description)

  execute(selection)
```

Výpis kódu 2.20: Příklad kombinace akcí

2.4 Play Framework

Projekt Play umožňuje tvorbu škálovatelných webových aplikací v Jave nebo Scale – programy v Play konzumují nižší množství zdrojů, jakými jsou CPU, paměť nebo vlákna, a podporují predikovatelnost jejich konzumace při zátěži. Interně Play používá knihovny Akka (Play 2.x a nižší) a Pekko (Play 3.x a vyšší), jenž poskytují asynchronní výpočetní model. Vytvořené webové aplikace jsou bezestavové s REST API v protokolech HTTP a WebSockets, včetně Comet a EventSource. K tomu navíc JSON je zde first-class citizen a framework nabízí knihovny pro serializaci dat. Tento celý balíček je zastřešen architekturou Model-View-Controller (MVC) a doporučenou strukturou výsledného projektu.[13]

Nespornou výhodou je, že framework umožňuje přistup k Actor System a dalším funkcionalitám Pekko/Akka na pozadí, a tak uživatel může aplikaci přímočaře rozšířit, např. o další aktory. Play rovněž ponechává na uživateli, zda použije dependency injection formou užití knihovny, např. Guice, nebo Cake Pattern pro compile-time DI. Stejně tak vývojář není omezen výběrem ORM – lze použít knihovny od klasické JPA pro Javu po Slick. Není zde tedy vazba na relační databáze a aplikace může přistupovat i k NoSQL databázím. Vysoký výkon v kombinaci s volbou ORM nabízí platformu pro práci s Big Data.[13]

Následující části demonstrují základní použití.

2.4.1 Serializace objektů

Částí Play je knihovna pro manipulaci s JSON. Při procesu serializace a deserializace jsou data převedena do JSON reprezentace *JsValue*. Nad ní probíhá případná validace. Knihovna nabízí automatickou serializaci základních datových typů pro case classes. Nicméně uživatel má možnost naprogramování i vlastních konvertorů a validátorů. Nadefinované formáty pro serializaci jsou pak použity implicitně.[13] Níže ukázka pro třídu, která obsahuje rovněž i výčtový typ. Nejdříve jsou třídy *Priority* a *Message* nadefinovány, následně vytvořen vlastní popis formátování pro výčet *Priority* a nakonec předvedena serializace a deserializace.

```
trait Priority

object Priority:
  case object High extends Priority
  case object Low extend Priority

case class Message(content: String, priority: Priority)
```

```
given formatPriority: Format[Priority] = new Format[Priority]:
  overrides def writes(value: Priority): JsValue = value match
    case Priority.High => JsString("high")
    case Priority.Low  => JsString("low")

  overrides def reads(json: JsValue): JsResult[Priority] = json match
    case JsString(value) if value == "high" => JsSuccess(Priority.High)
    case JsString(value) if value == "low"  => JsSuccess(Priority.Low)
    case _ => JsError("Unknown format")

given formatMessage: Format[Message] = Json.format[Message]

...

val obj = Message("haha", "low")

val ast = Json.object(
  "content" -> JsString("haha"),
  "priority" -> JsString("low"))

val str = """{ "content": "haha", "priority": "low" }"""

Json.toJson(obj)   // == ast
Json.parse(str)    // == obj
Json.fromJson(ast) // == obj
```

Výpis kódu 2.21: Formát pro JSON

2.4.2 Guice a injektáž

Pro demonstraci injektáže je zde použita knihovna Guice. V její filozofii je aplikace rozdělena do modulů, z nichž každý může mít svou implementaci metody *configure*, která se spustí hned po startu aplikace.[13]

```
class Module extends AbstractModule:
  override def configure(): Unit =
    bind(classOf[InitApp]).asEagerSingleton()
```

Výpis kódu 2.22: Příklad modulu

Guice používá anotace *Singleton*, *ImplementedBy* pro označení singleton objektů a v případě traitů pro označení, která třída je implementuje. Další častou anotací je *Inject*, která předchází konstruktoru třídy a označuje tak metodu, jenž bude volána s patřičnými dosazenými hodnotami. Play umožňuje dosadit přes injektáž konfiguraci aplikace, databázové spojení, Actor System od Pekko/Akka aj.[13]

```
@ImplementedBy(classOf[ReceiverImpl])
trait Receiver:
  receive(message: Message, to: String): Unit

...

@Singleton()
class ReceiverImpl @Inject(val config: Configuration):
  overrides def receive(message: Message, to: String) =
    println(s"Received message: ${message.content}, to: $to")
```

Výpis kódu 2.23: Příklad rozhraní a definice

Nakonec je v ukázce níže ukázán malý trik. Kód obsažený v definici třídy ve Scale je spuštěn celý, nebo-li všechen kód spadá pod konstruktor, který má parametry uvedené v kulatých závorkách za názvem třídy (viz taktéž metoda *apply*).[5] Chce-li tedy uživatel napsat kód, jenž bude spuštěn po rozeběhnutí aplikace Play, lze k tomu využít bindování *asEagerSingleton* a kód mít v rámci konstruktoru třídy.[13] Hodí se to jak pro inicializaci dat, tak pro tvorbu úloh, které poběží na pozadí, např. v rámci Pekko/Akka scheduleru.

```
class InitApp @Inject() (val config: Configuration):

  println("I am initialized !")
```

Výpis kódu 2.24: Příklad inicializace

2.4.3 Tvorba kontroléru

V poslední ukázce je řešeno naprogramování kontroléru, který přijme v parametru HTTP metody POST hodnotu *to* a v těle JSON objekt *Message*. Kontrolér předá zprávu službě *Receiver* a vrátí poděkování s HTTP statusem 200 (OK). Akce poskytované kontrolérem jsou všechny asynchronní.[13]

```
@Singleton()
class Controller @Inject() (
  val cc: ControllerComponents,
  val service: Receiver
) extends BaseController:

  def receive(to: String): Action[Message] =
    Action(parse.json[Message]): request =>
      val message = request.body
      service.receive(message, to)
      val json = JsString("Thanks")
      Ok(json)
```

Výpis kódu 2.25: Příklad kontroléru

V souboru */conf/routes* se nachází seznam cest. Pro zdejší příklad stačí cesta jedna pro HTTP metodu POST. Formát cest dovoluje užití parametrů a dalších vychytávek. Pokud uživatel s funkcionalitami, které defaultní implementace routeru poskytuje, není spokojen, je zde možnost i implementace vlastního formátu a routeru. Stejně tak k routerům je možné přidat zabezpečení. Narozdíl od mainstreamových JVM frameworků typu Spring je kód ve Scale takřka bez anotací a vše je řešeno skrze rozšiřování s využitím implicits a funkcionálního přístupu, respektive monád a kombinátorů.[13]

```
POST    /mailbox/:to    Controller.receive(to: String)
```

Výpis kódu 2.26: Příklad seznamu cest

Kapitola 3

Návrh systému

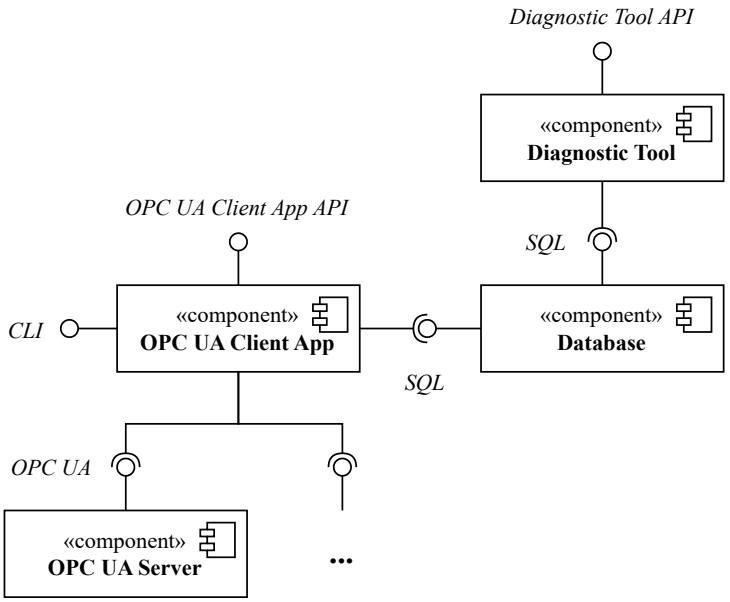
Tato kapitola popisuje problémovou doménu, ve které se řešení pohybuje, procesy, které v ní probíhají a návrh řešení. Rovněž se zaměřuje na funkční požadavky, které jsou kladeny na komponenty výsledného systému, a nasazení systému. Dále zmiňuje výchozí stav řešení. Popsané výsledky analýzy a návrhu provedené v této práci jsou shrnuty následovně:

- *Doména*
  - sběr naměřených dat z IoT zařízeních a jejich poskytnutí dalším nástrojům;
  - tok dat lze rozčlenit do procesů: sběr dat, zpracování dat a analýza dat.
- *Návrh systému*
  - systém je rozdělen do komponent (services/micro-services, UI aplikace);
  - komponenta Data Store: řeší správu úlohu na vyčtení dat a uložení nezpracovaných dat;
  - komponenty <Protokol> Adapter: implementuje Adapter API používané Data Store, mapuje IoT protokol na Adapter API;
  - dále obsahuje komponenty pro uživatelské rozhraní a zabezpečení.
- *Projekty*
  - komponenty jsou rozděleny do samostatných projektů;
  - znovupoužitelný kód je vyčleněn do knihoven;
  - dále existují projekty určené jako šablona projektu, dokumentace nebo testovací servery.

- *Technologie*
  - backend: Scala (Play Framework, Akka/Pekko, Cats, Slick);
  - frontend: JavaScript/TypeScript (React);
  - databáze: PostgreSQL, SQLite;
  - integrace: point-to-point (REST API, Websockets);
  - nasazení: Docker, Nginx;
  - zabezpečení: Nginx, OAuth Server.

3.1 Výchozí stav

Návrh systému po dokončení bakalářské práce „Systém pro sběr dat s využitím OPC UA“ se nachází ve stavu, kdy je zhotovena konzolová aplikace „OPC UA Client App“, která na základě příkazů z CLI provede danou operaci nad OPC UA servery a obdržená data uloží do databáze. Při analýze domény a požadavků na systém návrh počítal s nasazením pouze standardu OPC UA na IoT zařízeních, a tak dávalo smysl mít pouze správu úloh na vyčítání dat a správu dat pouze přes jednu aplikaci. Další diagnostické nástroje by si braly data přímo z databáze. Řečený plán znázorňuje UML diagram komponent na obrázku 3.1. [2]



Obrázek 3.1: Výchozí návrh komponent

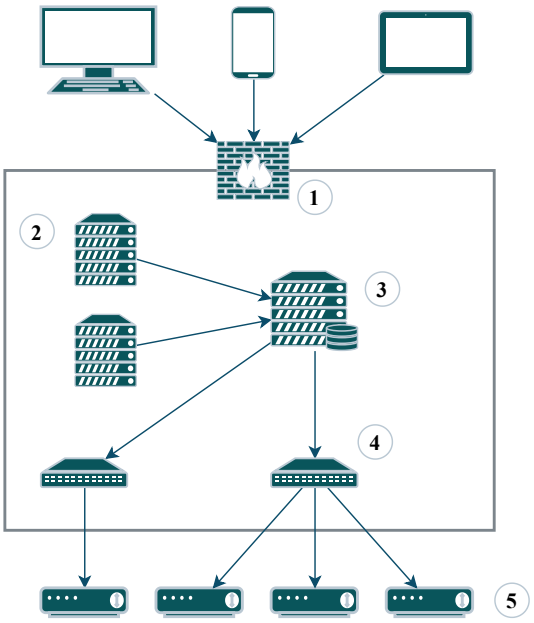
Ukázalo se však, že orientace na podporu pouze jednoho IoT protokolu není dostatečná a současně uložení dat včetně metadat lze udělat natolik obecně,

že by systém mohl být použitelný i pro vícero situací. Ze sesbíraných úvah a nových požadavků tak vznikl následující popis domény a návrh systému.

3.2 Doména

Systém pro sběr dat je nástroj pro zisk dat z IoT zařízení navržený pro společnost ModemTec, s.r.o. Společnost se orientuje na měření fyzikálních veličin převážně v oblasti elektrických sítí. Z naměřených dat vytváří statistické modely a predikuje chování, např. jakou životnost mají elektrické komponenty, nakolik je bezpečné je používat, kdy je již vyřadit nebo opravit, či v jakém stavu je izolace vodiče. K odpovědím na tyto otázky využívá zejména výzkum v oblasti částečných výbojů.[1]

Vzhledem k dlouhodobé predikci se neočekává od systému real-time vytížení - naopak se očekává dlouhodobé dotazování na uložená data v zařízeních. Nicméně některé hodnoty jsou získávány real-time, ale jedná se o menšinově zastoupenou situaci. Na obrázku 3.2 se nachází vizualizace možného použití systému. Systém je nasazen na serveru, jak backend, tak frontend. Uživatel skrze grafické UI zadává úlohy na vyčítání dat či manipulaci s daty. Je-li systém rozdělen do jednotlivých komponent, není nutné, aby se všechny nacházely na jednom výpočetním uzlu, mohou být rozděleny do vícero. V budoucnu lze lépe navrhnout optimalizaci a škálování. Doména první verze je tak omezena na jedno místo, které se stará o rozvrhnutí vyčítání a skladování dat a další místa, které se starají o přístup k IoT zařízením.



Obrázek 3.2: Vizualizace systému

Popis jednotlivých bodů:

1. zabezpečený přístup k systému – autentizace a autorizace;
2. diagnostické nástroje zpracující data a poskytující analytické informace uživatelovi;
3. správa úloh pro sběr dat a uložení syrových, tj. nezpracovaných, dat;
4. transformace dat ze zařízeních s různými IoT protokoly do podoby, ve které mohou být dále poskytnuty a uloženy;
5. IoT měřící zařízení s rozhraním v různých protokolech, jakými jsou OPC UA nebo ModemTec proprietární protokol pro komunikaci s PLC modemy.

3.2.1 Procesy

Zahrneme-li do domény i širší kontext než pouhý sběr syrový dat, je možné rozdělit manipulaci s daty do třech obecných procesů, respektive fází.

1. *Sběr dat*

   Sběr dat z IoT zařízení a transformace do datového schéma v databázi. Uložená data jsou tzv. syrová, tedy nejsou nijak zpracovány pro další použití, jsou ve stavu, v jakém byly vyčteny. K tomu obsahují metadata potřebná či vhodné pro další manipulaci.

2. *Zpracování dat*

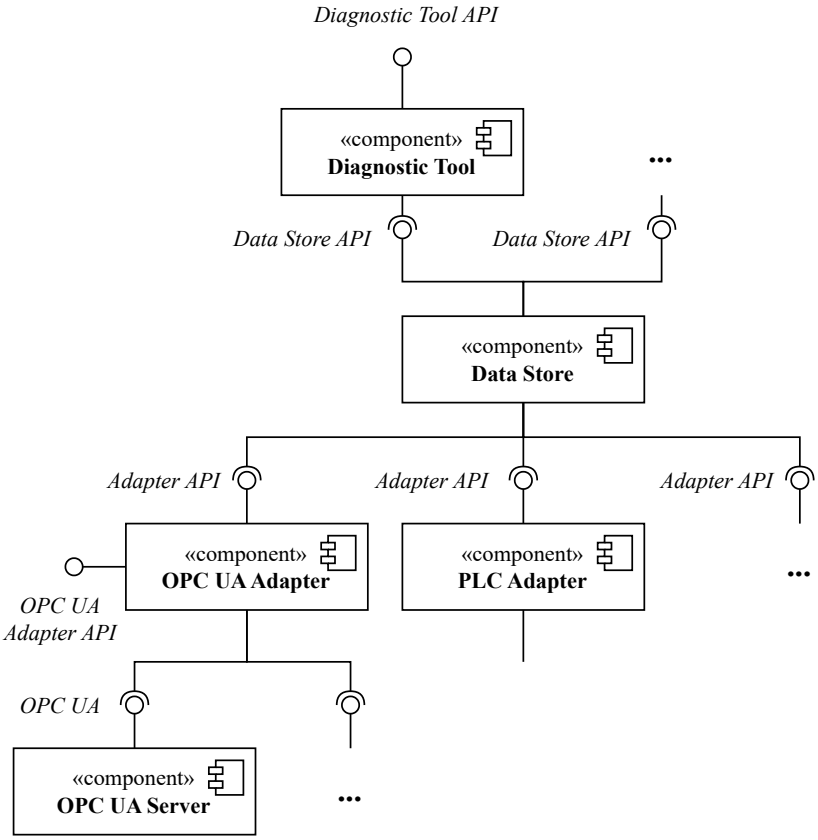
   Diagnostický nástroj využije datový sklad - místo, kde jsou syrová data uložena - a transformuje je do podoby využitelné k analýze. Může být více nástrojů, co stejná data transformují do různých podob. Proto je vhodné mít jak uložiště pro syrová data (v rámci systému pro sběr dat) a uložiště v diagnostickém nástroji, kde jsou data po transformaci.

3. *Analýza dat*

   Na konci datového toku jsou data nachystaná k prezentaci uživateli. Typicky jsou zobrazeny různé grafy či diagramy, případně tabulky. Diagnostický nástroj obvykle umí i výsledky exportovat do standardizovaných formátů.

3.3 Komponenty

Níže jsou popsány systémové komponenty. Na obrázku 3.3 se nachází UML diagram komponent, kde lze krom samotných komponent i vidět rozhraní, které poskytují, a jak se komponenty navzájem využívají.



Obrázek 3.3: Komponenty systému

3.3.1 Adapter

Adapter poskytuje mapování mezi daným IoT protokolem a Adapter API. Tím pádem lze k manipulaci s IoT zařízením použít univerzální rozhraní adaptéru bez ohledu na použitý protokol na zařízení. Což umožňuje vytvořit uživatelské rozhraní použitelné na jakýkoliv adaptér, či vytvořit komponentu sbírající data zkrze adaptéry. Stinnou stránkou může být, že Adapter API není natolik univerzální, aby pokrylo veškeré funkcionality nutné k používání adaptéru. Například OPC UA Adapter rozšiřuje Adapter API o pár metod spravující podporované informační modely, které mohou být volány při inicializaci systému nebo v průběhu, pokud uživatel zjistí, že potřebuje vložit

do OPC UA Adapteru další OPC UA NodeSety. Ale kupříkladu PLC Adapter si vystačí pouze s Adapter API. Na vyčítání/zápis dat je tedy Adapter API dostačující. Dále Adapter poskytuje správu připojení k IoT zařízením.

3.3.2 Data Store

Data Store spravuje úlohy na vyčítání/zápis dat z IoT zařízení pomocí adaptérů. Dále poskytuje uložení nasbíraných dat a rozhraní k jejich dotazování. Souvisejíc se sběrem rovněž Data Store spravuje připojení k adaptérům a skrze ně může spravovat připojení k IoT zařízením. Rozhraní je pojmenováno Data Store API. Původně Data Store měl implementovat i Adapter API, ale při návrhu je od toho opuštěno, jelikož by v implementaci došlo ke zbytečně komplexnímu „ohýbání“ řešení, např. přidat adaptér se zařízením do cesty k metodě v Adapter API. Případné pokusy o implementaci Adapter API jsou ponechány do budoucna a Data Store tak poskytuje Data Store API, které je přímo určené pro jeho požadované funkcionality. Adapter API poskytuje rovněž specifikaci, resp. metadata, jak vypadají operaci či datový model na IoT zařízení, což Data Store využívá k dotazování nebo poskytnutí

3.3.3 Diagnostic Tool

Komponenta Diagnostic Tool zastupuje jakoukoliv aplikaci, která využívá Data Store ke sběru dat. Na získaná data aplikuje různé funkce, např. agregace, statistické modely, jejichž výsledky poskytuje uživateli. Narozdíl od Data Store, kde jsou uložena syrová data a metadata, obecný Diagnostic Tool si ukládá již zpracovaná data. To umožňuje dlouhodobé uložení syrových dat v Data Store a vznik dalších diagnostických nástrojů, které mohou, ale i nemusí, zpracovávat stejná data.

3.4 Rozdělení do projektů

Každá komponent má svůj projekt, který umožňuje nezávislý vývoj. V rámci standardizace vývoje je definovaná obecná struktura projektu a co má obsahovat *readme.md* soubor. Níže seznam projektů a rozdělení do adresářů či skupin v případě, že je použit systém pro správu verzí, např. GitLab.

- */tools*

  Adresář s podpůrnými nástroji, např. OPC UA NodeSet Editor pro úpravu informačních modelů OPC UA.

- */docs*

  Adresář s dokumentacemi, např. dokumentace k řešení systému pro sběr dat.

- */adapter*

  Adresář obsahující komponenty implementující Adapter API.

  - *Adapter API*

    Projekt s návrhem Adapter API a s implementacemi klientů pro Adapter API.

  - *OPC UA Adapter*

    Projekt realizující adaptér určený pro protokol OPC UA.

- */libs*

  Adresář s knihovnami, které různé projekty využívají.

  - *Scalable OPC UA*

    Knihovna implementující asynchronního klienta OPC UA a práci s daty OPC UA.

- *Project Template*

  Projekt s šablonou pro projekty. Rovněž obsahuje popis, co má obsahovat *readme.md* a jak má být projekt strukturován.

- *Data Store*

  Komponenta s schedulerem pro úlohy, které mají za úkol vyčítat data skrze adaptéry, a databází pro uložení získaných dat.

- *OPC UA Test Server – open62541*

  Testovací server pro OPC UA implementovaný v knihovně open62541, ve které jsou implementovány servery běžící na IoT zařízeních.

- *OPC UA Test Server – Eclipse Milo*

  Testovací server pro OPC UA implementovaný v knihovně Eclipse Milo, kterou využívá knihovna Scalable OPC UA.

3.5 Integrace komponent

Přestože point-to-point integrace bývá problematická pro velké systémy, v tomto případě je použita. Je možné ho použít vzhledem k nízkému počtu komponent, kdy zvýšená komplexita event-driven architektur či message-oriented middleware by přinesla větší pracnost bez užitku, které přináší v podobě zmizení závislostí mezi komponentami a rozložení zátěže.[14] Přeci jen systém je určen pro jednotky uživatelů a je orientovaný více na analytické zpracování (OLAP) než na zpracování velkého množství transakcí (OLTP). Cílová IoT zařízení jsou různé měřící nástroje pro dlouhodobé sledování hodnot, a tak

sběr dat je typicky rozvržen na delší časové úseky. Stejně tak je třeba brát v potaz zatížení sítě mezi IoT zařízeními. Tedy provoz na síti přidává nějaké požadavky na systém a rozložení zátěže musí probíhat zde. Očekává se, že síť mezi komponentami systému snese větší provoz než síť mezi zařízeními. Na druhou stranu je nutné při návrhu a implementaci jednotlivých komponent systému myslet na to, že se situace může změnit a může dojít někdy v budoucnu k využití message queues. Nicméně jednotlivá API jsou beztak vhodná pro využití frontendovými aplikacemi.

3.6 Funkční požadavky na API komponent

Z návrhu a popisu domény výše vyplývají následující požadavky na funkcionality jednotlivých komponent, resp. požadavky na jejich API. Rozhraní pro obecný Diagnostic Tool není rozebráno, jelikož silně zavisí na jeho účelu – nejedná se o obecné řešení, které nabízí Data Store s připojenými Adapter API.

3.6.1 Adapter API

Adapter API poskytuje rozhraní v následujících třech oblastech:

1. *správa připojení/zdrojů*
    - přidání, odebrání, úprava připojení/zdroje;
    - podporované certifikáty;
2. *specifikace připojení/zdroje*
    - generování specifikace pro dané připojení/zdroj – ve formátu podobném velmi zjednodušenému informačnímu modelu OPC UA, jelikož na objektový model, resp. strom objektů, lze namapovat jak protokoly, jenž jsou objektově orientované, tak protokoly obsahující pár funkcí.
3. *zpřístupnění funkcí*
    - čtení, zápis hodnoty, vyčtení historie hodnoty, zavolání metody.

3.6.2 Data Store API

Od Data Store API se očekává, že poskytne rozhraní ve čtyřech následujících oblastech:

1. *správa úloh*

    - přidání, odebrání, úprava úlohy;
    - podpora jednorázových úloh, podpora úloh běžících po delší časový úsek;
    - úlohy odpovídající návrhu Adapter API: zavolání metody, vyčtení hodnoty, zápis hodnoty, vyčtení historie;
    - asynchronní chování úloh - je možné nahlédnout na současný stav, ve kterém se úloha nachází.

2. *správa dat*

    - možnost jednorázové odpovědi, ale i streamu dat;
    - dotazy na specifikaci/metadata připojení/zdroje;
    - dotazy na získané hodnoty a návratové hodnoty metod pro dané připojení/zdroj.

3. *dotazování nad daty*

    - migrace dat: přidání, extrakce dat;
    - možnost zcela vyčistit databázi.

4. *správa adaptérů*

    - přidání, odebrání, úprava adaptéru;
    - správa podporovaných certifikátů.

3.7 Uživatelské rozhraní

Konkrétní uživatelské rozhraní je předmětem dalšího vývoje. Nicméně současný návrh počítá spíše s variantou micro frontends. Na tuto architekturu UI lze nahlížet jako na skládání více UI komponent či projektů do jednoho. Což je možné využít stylem, že je vytvořený frontend pro dotyčné API a následně cílová UI aplikace je složením těchto frontendů.[15] Příkladem budiž OPC UA Adapter, jehož rozhraní se skládá z Adapter API a specifického OPC UA Adapter API.

3.8 Zvolené technologie

Rozdělení řešení do vícero projektů umožňuje vícejazyčné prostředí. Přesto je snaha proces vývoje a technologie unifikovat, aby se zúžily požadavky na paletu schopností, kterou musí vývojáři oplývat.[16] Hlavním jazykem pro psaní backend komponent je jazyk Scala. K němu patří užité frameworky a knihovny jakými jsou Play Framework, Slick, Cats nebo Akka, případně ekvivalentní Pekko. Scala se vyznačuje kombinací OOP a funkcionálního programování a s technologiemi v ní vytvořenými je tak ideálním nástrojem pro psaní programů na zpracování dat.[5] Oproti tomu fronendovým jazykem je JavaScript, eventuálně TypeScript. Očekávanou technologií pro tvorbu UI je React. React kombinuje funkcionální prvky a stavbu UI jakožto komponent.[17] Velkou výhodou všech zmíněných technologií je i jejich široká komunita a dostupnost různých návodů [5][17] a jejich zařazení do výuky na některých vysokých školách, např. FIT ČVUT.[18]

Databáze jsou poněkud tradičně vedeny v relacích - PostgreSQL nebo SQLite dle potřebné velikosti. Nicméně v závislosti na vývoji a dalšího rozvoje funkcionalit je možné použít Neo4j pro grafové problémy, v tomto případě typicky struktura sítě nebo grafu objektů, či MongoDB pro Big Data, kde se očekává rychlost. Ale i v případě databází byť moderní technologie mohou být velmi lákavé, je nutné zvážit i dostupnost studijních materiálů a vhodnost pro daný problém.

API jednotlivých komponent je určeno pro point-to-point komunikaci. V současné chvíli je kladen důraz na jednoduchost a odladitelnost. Pro API jsou tedy zvoleny standardy REST a WebSockets. Snadný vývoj však může přinést nevýhody v podobě efektivity, např. velké objemy dat ve formátu JSON, či synchronní komunikace zvyšující latenci. V případě, že by časem bylo třeba optimalizovat, nabízí se řešení skrze Message Queue nebo gRPC protokol.

3.9 Nasazení systému

Nasazení je provedeno s využítím technologie Docker a případně jeho funkcionality Docker Compose, jelikož se jedná pouze o jednotky softwarových komponent. Nasazení lze provést v závislosti na požadovaných funkcionalitách, například je možné nasadit pouze OPC UA Adapter a odpovídající frontend pro Adapter API, nebo celý systém, to je Data Store, frontendové aplikace a jeden či více OPC UA Adaptérů. Současně lze při nasazení využít Nginx a OAuth server pro zabezpečení systému.

Kapitola 4

Scalable OPC UA

Tato kapitola začíná výchozím stavem knihovny Scalable OPC UA s lehkým zhodnocením. Knihovna podporuje práci s daty a klienta OPC UA v jazyce Scala. Klient je naprogramován nad Java knihovnou Eclipse Milo. Následně kapitola pokračuje představením nových funkcionalit a implementace.

4.1 Výchozí stav

Na následujících řádcích je stručně popsán výchozí stav knihovny Scalable OPC UA. Podrobnější informace lze nastudovat v kapitole 6 „Scalable OPC UA“ v bakalařské práci „Systém pro sběr dat s využitím OPC UA“. Knihovna samotná podporuje OPC UA v1.04 a neimplementuje zcela celý protokol, jelikož se zaměřuje na transformaci dat mezi formáty, parsování a připojení k OPC UA serveru. Neméně důležitou součástí je i datový model.[2]

4.1.1 Jazyk

Jazykem je zvolena Scala verze 2.13. Zde nutno podotknout, že v době vývoje (rok 2022) byla Scala 2.13 nejaktuálnější verze s dlouhodobou podporou, tzv. LTS. Scala 3 byla teprve v počátcích vývoje, stejně jako nástroje ji podporující. A taktéž teprve vznikaly materiály, tutoriály a knihy. Plno projektů, včetně stěžejních pro tuto práci, tj. Play Framework, Slick, Akka, neměly ještě plnou podporu pro Scalu 3.[5]

4.1.2 Funkcionality

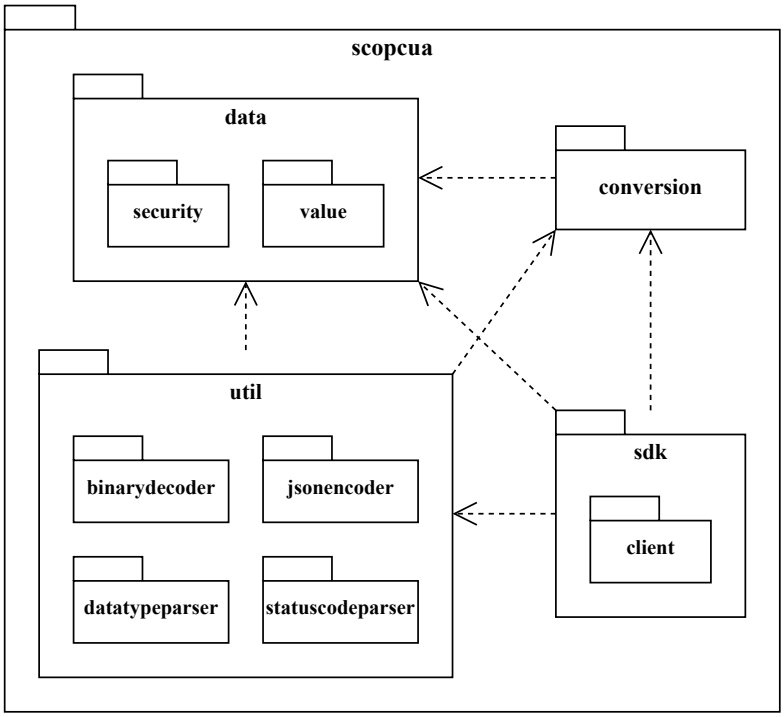
Knihovna ve výchozím stavu poskytuje následující funkcionality:

- *UaBinaryDecoder* – transformace ze standardního OPC UA binárního kódování do datového modelu;
- *UaJsonEncoder* – transformace do JSON kódování z datového modelu;

- *UaStatusCodeParser* – parsování CSV formátu návratových kódů OPC UA;

- *UaDataTypeParser* – parsování XML souboru s informačním modelem za účelem získání datových typů.

Přehled jednotlivých balíčků a jejich závislostí je zobrazen níže. UML digram je převzat z [2].



Obrázek 4.1: Struktura balíčků knihovny (UML)

4.1.3 Klient

OPC UA klient je naprogramován jako synchronní. Navíc jsou implementovány pouze operace vyčtení hodnoty, přečtení datového typu proměnné a zjištění všech jmenných prostorů na serveru. A samozřejmě připojení a odpojení. Zabezpečení připojení pomocí jméno, heslo a PKI již v této verzi je rovněž naprogramováno. Nicméně toto vše poskytuje pouze možnost si vyzkoušet připojení k OPC UA serveru a jednoduché vyčtení hodnoty.[2] Na nasazení v praxi je to málo.

4.2 Úpravy

Vývoj knihovny především reaguje na požadavky aplikace OPC UA Adapter, vývoj jazyka Scala a k němu odpovídajících technologií. V úpravách je změněn jazyk, klient a přidány/upraveny funkcionality. Struktura projektu, tj. rozdělení kódu do balíčků na principu diagramu 4.1, zůstává zachována. Dále je proveden refactoring kódu.

4.2.1 Jazyk

V současné době (rok 2024) již Scala 3 má LTS verzi a to 3.3.1, která je podporována rozšiřujícím se množstvím projektů, zejména Akka a Play Framework, ale i knihovna Cats. Tudíž prvním úpravou je převod projektu ze Scaly 2 do 3.[5] Vzhledem k určitým nekompatibilitám je přechod proveden částečně automatizovaně dle oficiálního návodu a částečně ručním přepisem kódu. Ruční přepis je proveden v souladu s návrhem pro celý systém, kdy je pro Scalu použit bez-závorkový (braceless/indentation) styl.

4.2.2 Funkcionality

Co se týče funkcionalit, jsou zachovány následující:

- *UaBinaryDecoder*,
- *UaJsonEncoder*,
- *UaStatusCodeParser*.

Modul *UaDataTypeParser* je zrušen, resp. je přetvořen z parsování datových typů z XML souboru obsahující informační model OPC UA na parsování všech uzlů včetně datových typů.[19] Seznam funkcionalit je rozšířen o níže uvedené:

- *UaBinaryEncoder* – transformace datového modelu do standardního binárního kódování OPC UA;
- *UaJsonDecoder* – transformace do datového modelu z JSON kódování;
- *UaJsonValidator* – validace JSON hodnoty podle definice datového typu;
- *UaJsonSchemaGenerator* – generování JSON schéma podle definice datového typu;
- *UaNodeSetExtractor* – parsování XML souboru z informačním modelem OPC UA a zisk uzlů s datovými typy;
- *UaTrustListManager* – správce certifikátů a klíčů pro PKI;

- *UaValueCreator* – tvorba defaultních hodnot na základě definice datového typu.

Dále jsou doplněny implicitní konverze mezi třídami, resp. datovými typy, které nebyly součástí předešlého řešení.

4.2.3 Klient

Při úpravě klienta se rozšířily podporované operace na většinu standardních pro OPC UA, to jest:

- *readValue* – vyčtení aktuální hodnoty proměnné;
- *writeValue* – zápis hodnoty proměnné;
- *callMethod* – zavolání metody s parametry;
- *readHistory* – vyčtení historie hodnot v daném časovém úseku;
- *readNamespaces* – vyčtení jmenných prostorů na serveru;
- *createSubscription* – vytvoření odběru novinek;
- *createMonitoredItem* – vytvoření monitorované položky v rámci daného odběru novinek;
- *deleteMonitoredItem* – zrušení monitorované položky;
- *deleteSubscription* – zrušení odběru novinek;
- *readDataTypeId* – vyčtení datového typu proměnné;
- *connect* – připojení klienta k serveru;
- *disconnect* – odpojení klienta od serveru.

Nicméně hlavní úpravou je změna synchronního chování na asynchronní. Asynchronní API klienta může využít potenciál funkcionálního programování skrze monádu *Future[A]* a současně se stane neblokujícím kódem. Lze tak využít výhody jazyka Scaly, kdy díky monadickému přístupu a implicitnímu kontextu, např. *ExecutionContext* pro *Future[A]*, je možné lépe (s větší mírou optimalizace) řídit více vláknové programy. Narozdíl od Javy, kde panuje ideologie mnoha vláken a blokujícího kódu, Scala se ubírá směrem neblokujícího kódu a méně vláken.[5]

Jak takový neblokující kód pro operaci *readValue* může být naprogramovaný funkcionálně je vidět na ukázce kódu níže. Předtím ovšem je uvedena ještě synchronní varianta ve Scale 2.13, aby si čtenář mohl udělat srovnání s původním řešením.[2]

```
override def readValue(id: UaId): (UaValue, UaDateTime) = {
  val nodeId = converter.uaId2NodeId(id)
  val value =
    client
      .getAddressSpace
      .getVariableNode(nodeId)
      .readValue()

  val dateTime = value.getSourceTime
  val variant = value.getValue
  val typeId = readDataTypeId(id)

  val resultValue = converter.variant2UaValue(variant, typeId)
  val resultDateTime = converter.dateTime2UaDateTime(dateTime)
  (resultValue, resultDateTime)
}
```

Výpis kódu 4.1: Synchronní readValue

```
override def readValue(id: UaId): Future[UaReadValue] =
  for
    typeId <- readDataTypeId(id)
    value  <- readValue(id, typeId)
  yield value

override def readValue(id: UaId, typeId: UaId): Future[UaReadValue] = guard:
  val nodeId = converter.uaIdToNodeId(id)
    client
      .readValue(0.0, TimestampsToReturn.Both, nodeId)
      .asScala
      .map: dataValue =>
        val value = converter.variantToUaValue(dataValue.getValue, typeId)
        val sourceTime =
          converter.dateTimeToUaDateTime(dataValue.getSourceTime) match
            case Some(value) => if value.value == 0 then None else Some(value)
            case None        => None
        UaReadValue(value, sourceTime)
```

Výpis kódu 4.2: Asynchronní readValue

Pro vysvětlenou v obou ukázkách proměnná *client* představuje klienta z Java knihovny Eclipse Milo, nad kterou je implementován klient knihovny Scalable OPC UA. Odtud transformace Java *Future* do ekvivalentu ve Scale pomocí metody *asScala*. Za poukázání na další detail stojí ještě funkce *guard*, což je funkce, která přijímá kód (funkci) vracející *Future[A]* jako parametr. Vyskytne-li v prováděných instrukcích výjimka, je transformována do formátu výjimek knihovny Scalable OPC UA.

4.3 Testování

Způsob testování zůstal zachován. To jest automatizované jednotkové testy pro pomocné třídy a automatizované integrační testy pro moduly. Současně testování klienta probíhá nad běžícím OPC UA serverem, aby se daly zjistit případné chyby v komunikaci či použití knihovny Eclipse Milo. Rovněž je nutné brát na zřetel, že i Eclipse Milo je v nějaké fázi vývoje, tudíž i tato knihovna může obsahovat chyby. Pro účely testování je použit OPC UA server „Eclipse Milo OPC UA Demo Server“ a server vytvořený pomocí knihovny open62541, který se snaží simulovat použité funkcionality produkčního OPC UA serveru.[2] Pod produkčním serverem si lze představit například měřící zařízení, jehož informační model obsahuje metody podporující měření fyzikálních veličin a proměnné, které uchovávají naměřená data a případně další technické parametry.

4.4 Ukázky použití

Následující ukázky použití knihovny jsou vypůjčeny z automatických testů klienta využívající knihovnu ScalaTest. Pro úplnost je nejdříve definován trait pro testování klienta s definicí níže. Účelem této třídy je poskytnout metody pro připojení/odpojení klienta v případě dokončení série operací nebo při vzniku chyby. Současně lze z obrázku 4.4 nahlédnout, jak by mohla být programována třída zajišťující proxy pro klienta.

```
trait UaMiloClientTest extends AnyFunSuite with Matchers:

  given context: ExecutionContext = ExecutionContext.global

  extension [A](future: Future[A])
    def asSync: A = Await.result(future, 240.seconds)

  def createClient: UaMiloClient

  def withClient[A](code: UaMiloClient => A): A =
    val client = createClient
    var isConnected = false

    try
      client.connect().asSync
      isConnected = true
      code(client)
    finally
      if isConnected then client.disconnect().asSync
```

Výpis kódu 4.3: Trait pro testování klienta

A následně je definována již samotná třída pro testování klienta s anonymním připojením.

```
class UaAnonymousMiloClientTest extends UaMiloClientTest:

  override def createClient: UaMiloClient =
    val types = DataTypeDataSet.All
    val config =
      UaClientConfig(
        name       = "Scalable Test Client",
        uri        = "urn:scalable:test:client",
        timeout    = 5000,
        url        = "opc.tcp://127.0.0.1:4840",
        auth       = None,
        encryption = None,
        policy     = UaSecurityPolicy.None)

    UaMiloClient.from(config, types)
```

Výpis kódu 4.4: Třída pro testování anonymního klienta

4.4.1 Vyčtení hodnoty

V testu, který se nachází v kódu 4.4.1, je domonstrováno vyčtení hodnoty. Jedná se o strukturu Quality ve standardu IEC 61580-7-3.[20] Na ukázce je možné vidět, jak lze v knihovně vytvořit instanci struktury, případně enumerace a jak zavolat *readValue* s dalším možným zpracováním.

```
test("Read data of node /PDM1/LLN0/Beh/q", NeedsRunningServer):
  val expected =
    UaStructure(
      "validity" -> UaEnumeration(1, "invalid"),
      "detailQual" ->
        UaStructure(
          "overflow"     -> UaBoolean(true),
          "outOfRange"   -> UaBoolean(false),
          "badReference" -> UaBoolean(true),
          "oscillatory"  -> UaBoolean(false),
          "failure"      -> UaBoolean(true),
          "oldData"      -> UaBoolean(false),
          "inconsistent" -> UaBoolean(true),
          "inaccurate"   -> UaBoolean(false)),
        "source" -> UaEnumeration(0, "process"),
        "test" -> UaBoolean(false),
        "operatorBlocked" -> UaBoolean(true))

  val result = withClient: client =>
    val id = UaId(6001, "http://www.modemtec.cz/PD/")
    val readResult = client.readValue(id).asSync
    readResult.value.get

  result shouldBe expected
```

Výpis kódu 4.5: Vyčtení hodnoty v testech

4.4.2 Zavolání metody

V nadcházející ukázce 4.4.2 lze spatřit, jak zavolat metodu na objektu. API knihovny poskytuje volání metody bez nutnosti znalosti pořadí vstupních argumentů – jako parametr metody *callMethod* je použit slovník *Map* s názvy argumentů a k nim přiřazených hodnot. Opět zde vidíme transformaci asynchronní metody *callMethod* na synchronní pomocí implicitní funkce *asSync*. Výsledek je vrácen jako slovník názvů výstupních argumentů s jejich hodnotami.

```
test("Call Method /PDM1/Conf/changePass - result OK", NeedsRunningServer):
  val objectId = UaId(5016, "http://www.modemtec.cz/PD/")
  val method = "changePass"

  val input = Map(
    UaString("username") -> UaString("username"),
    UaString("password") -> UaString("password"))

  val expected = Map[UaString, UaValue]("result" -> UaEnumeration(0, "Ok"))

  val result = withClient: client =>
    client.callMethod(objectId, method, input).asSync

  result shouldBe expected
```

Výpis kódu 4.6: Zavolání metody v testech

4.4.3 Vyčtení historie

V ukázce 4.4.3 je možné vidět použití metody *readHistory* pro zisk historických hodnot zvolené proměnné. Zde je využita vlastnost Scaly umožňující, aby metoda měla více seznamů argumentů.[5] Tudíž *readHistory* má první seznam obsahující určení uzlu a od-do časové známky dle úseku historie, který uživatel chce, a druhý seznam argumentů obsahující funkci/callback pro zpracování jedné hodnoty s časem.

```
test("Read history of node /PDM1/SPDC1/Beh/q", NeedsRunningServer):
  val expected = Vector(/* test data */)

  val result = withClient: client =>
    var collected = Vector.empty[UaValue]
    val id = UaId(6058, "http://www.modemtec.cz/PD/")
    val from = stringToUaDateTime("1601-01-01T00:00:00Z")
    val to = stringToUaDateTime("2025-01-01T00:00:00Z"))

    val action = client.readHistory(id, from, to): readValue =>
      collected = collected :+ readValue.value.get
```

```
    Await.result(action, 60.second)
    collected

  result shouldBe expected
```

Výpis kódu 4.7: Vyčtení historie v testech

Kapitola 5

Adapter API

Kapitola popisuje návrh Adapter API, který vznikl z myšlenek na úpravu systému. Text se podrobně věnuje jednotlivým operacím, přenášeným objektům, chybám, verzování a serializaci, jelikož v praxi se ukazuje, že kvalita návrhu jakéhokoliv API může významně usnadnit nebo ztížit celkový vývoj projektu. Kapitola dále obsahuje stručný komentář ke knihovně implementující klienta pro zmíněné API a ukazuje její použití.

5.1 Návrh

V této části je popsán návrh Adapter API. Jedná se o textový popis rozhraní adaptéru. V projektu se nachází i adresář */schemas*, kde lze dohledat různá schémata ve strojově čitelných formátech, např. */schemas/openapi.yaml* obsahuje popis HTTP části API ve formátu OpenAPI v3.0.0.

Adaptér spravuje zdroje v daném protokolu a Adapter API poskytuje jednotné rozhraní pro všechny adaptéry. Úlohou adaptéru je udržovat připojení nebo naopak jejich zánik, poskytnout operace pro zisk dat ze zdrojů či jejich zápis. Adapter API se snaží o vytvoření co nejuniverzálnějšího rozhraní. Čerpá z požadavků, které jsou obvykle kladeny na sběr dat z IoT zařízení. API však lze zobecnit na jakýkoliv server nebo síť.

5.1.1 Verzování

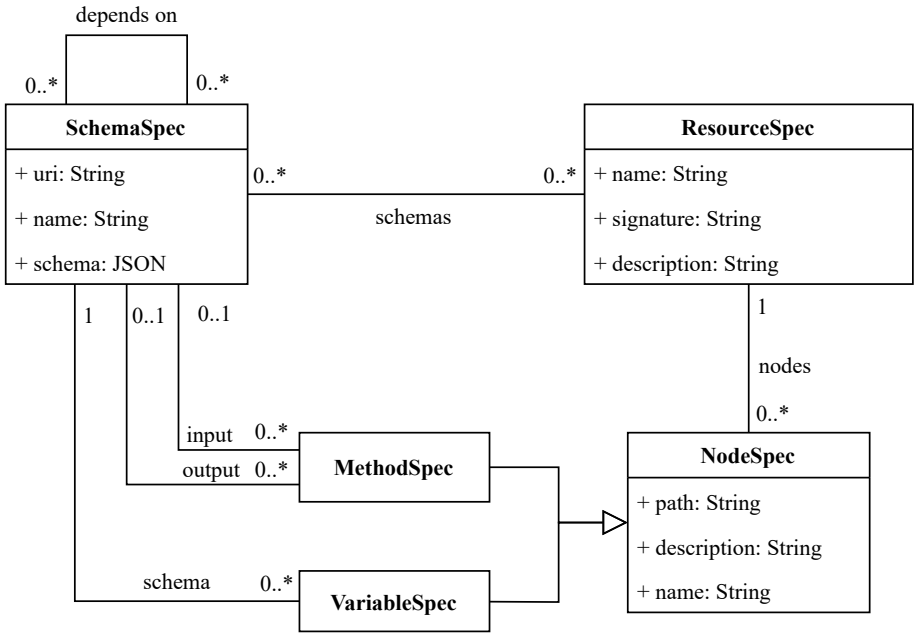
Existují dvě základní strategie pro verzování API využívající URL pro určení endpointu, což jsou například protokoly vystavěné nad HTTP či SOAP. První možností jak verzovat je vložit číslo verze do společné části URL všech endpointů. Druhou možností je poskytnutí endpointu, který nabízí informace o daném serveru, např. verzi API, název, popis.[14] Adapter API používá druhou strategii. HTTP endpoint s cestou {*URL serveru*}*/about/info* při volání metody GET vrací strukturu *AdapterInfo*, která mimo jiné obsahuje číslo verze Adapter API a číslo verze samotného adaptéru. Lze tak dostat komplexní in-

formaci o verzování a případně v budoucnu mít možnost rozšíření struktury o další data.

5.1.2 Specifikace zdroje

Zdroj (*Resource*) v řeči adaptéru představuje cokoliv, co může vlastnit endpointy, nad kterými lze volat operace. Typickým zdrojem je například server, ale může se jednat i o síť. Z toho důvodu objekt *ResourceConfig* obsahuje i velmi obecný atribut *connstr* (dlouze rozepsáno *connection string*), kde lze uložit kupříkladu URL serveru či pouze název portu, je-li zdrojem sériový port.

Zdroj popisují metadata inspirované protokolem OPC UA, jenž jsou nezavislé na použité technologii pro implementaci Adapter API. Data s popisem zdrojem představuje objekt *ResourceSpec*. Základním stavebním kamenem je uzel (Node), který představuje bod pro komunikaci. Uzlem může být metoda (Method) nebo proměnná (Variable). V OPC UA existuje dále také i objekt.[19] Nicméně se nejedná o koncový bod API, na kterém by bylo možné volat nějakou metodu nebo spravovat proměnnou. Tudíž je od objektů opuštěno. Přesto ale lze objektově modelovat a to díky atributu cesta, který každý uzel vlastní a který v rámci specifikace zdroje je unikátní. Navíc lze mít cesty, aniž by se za nimi schovával nějaký objektový model - tedy jako identifikátor nebo klasická cesta v rámci URL. Uzly popisují objekty *MethodSpec* a *VariableSpec*, jenž dědí od *NodeSpec*.



Obrázek 5.1: Model specifikace zdroje

Operace nad uzly jsou rovněž inspirované OPC UA. Pro uzel *Variable* lze použít čtení (*Read*), zápis hodnoty (*Write*), vyčtení historie hodnoty (*HistoryRead*) nebo odběr nových hodnot (*Subscribe*). Nad uzlem *Method* je možné volat metodu (*CallMethod*). Datové typy proměnných nebo vstupní/výstupní argumenty metod jsou popsány pomocí JSON Schema. To je obsaženo v objektu *SchemaSpec*. Identifikací schéma je jeho URI.

Obrázek 5.1 zobrazuje UML diagram tříd specifikace zdroje. Bližším informace k atributům se nacházejí v části *Objekty*.

Příklad

Nechť P je libovolný protokol, který implementuje server vytvořený uživatelem a uživatel chce funkcionality serveru namapovat na rozhraní adaptéru. Budiž to server na měřiči teploty, který poskytuje teplotu ve stupních celsia (*temperature*) a metodu „kalibruj“ (*calibrate*). Aby to nebyl úplně triviální příklad, endpoint *temperature* poskytuje pole 5 posledních hodnot typu Int a *calibrate* má dva vstupní argumenty *number* a *otherNumber* s typy taktéž Int. Dané dva endpointy můžou být namodelovány jako dva uzly:

1. *Variable temparature* s cestou */temperature* a se schéma identifikovaný pomocí URI *http:/www.measure.com/temperature*;
2. *Method calibrate* s cestou */tools/calibrate* a se schéma identifikovaným pomocí URI *http:/www.measure.com/tools/calibrate/Input*.

Daná schémata vypadají následovně:

- Datový typ „Teplota“.

```
{
   "$schema" : "https://json-schema.org/draft/2020-12/schema",
   "$id" : "http:/www.measure.com/Temp",
   "title" : "Temp",
   "type" : "number"
}
```

Výpis kódu 5.1: Schéma příkladu datového typu

- Schéma proměnné „teplota“.

```
{
  "$schema" : "https://json-schema.org/draft/2020-12/schema",
  "$id" : "http:/www.measure.com/temperature",
  "title" : "VariableValue",
```

```
        "type" : "array",
        "items" : {
              "$ref" : "http:/www.measure.com/Temp"
        },
        "minItems" : 0,
        "maxItems" : 5
}
```

Výpis kódu 5.2: Schéma příkladu proměnné

- Schéma vstupních argumentů metody „kalibruj“.

```
{
  "$schema" : "https://json-schema.org/draft/2020-12/schema",
  "$id" : "http:/www.measure.com/tools/calibrate/Input",
  "title" : "MethodInput",
  "type" : "object",
  "properties" : {
    "number" : {
      "type" : "number"
    },
    "otherNumber" : {
      "type" : "number"
    }
  },
  "required" : [ "number", "otherNumber" ]
}
```

Výpis kódu 5.3: Schéma příkladu metody

Za pozornost stojí, že vstupní argumenty metody jsou zde modelovány jako *JSON Object* a že proměnná není skalární. Vstupní argumenty jako objekt jsou užitečné pro zachování názvů argumentů. Jinými slovy hodnoty argumentů nejsou dosazovány „naslepo“, přesněji v pořadí, které nám říká dokumentace k danému protokolu P, ale podle jejich názvů. Proměnná, která není skalární, může mít jedno schéma, nebo mít dvě schémata, kde jedno je schéma pro skálární datový typ a druhé pro proměnnou, která obsahuje pole hodnot tohoto typu.

Specifikace zdroje může být využita ve vícero situacích. Jmenovitě například generování tříd/datových typů podle JSON Schéma nebo řešení uživatelského vstupu ve frontendu, případně zobrazení datových struktur uživateli. Další text bude zaměřen na bližší popis operací.

5.1.3 Operace

Současné API adaptéru poskytuje dva protokoly pro komunikaci - HTTP a WebSockets. HTTP pro restful část a WebSockets pro streamování dat.

HTTP

Na následujících řádcích je popsána část API využívající protokol HTTP. Nejdříve pár poznámek společných pro všechny operace.

- Cesty zdrojů v HTTP (nikoliv zdrojů v řeči Adapter API) jsou zapsány relativně vůči výsledné URL, která však může mít různou adresu v závislosti, kde se nachází adaptér.
- Atribut *guid* použitý v cestách je shodný s atributem *guid* v *ResourceConfig* objektu. Představuje globální unikátní identifikátor zdroje. Atribut *nodepath* představuje libovolnou cestu k uzlu, která vznikla během transformace protokolu zdrojů do Adapter API. Některé adaptéry mohou mít neměnnou množinu cest k uzlům, ale existují i adaptéry, které mají množinu cest závislou od aktuálně dostupných zdrojů. Např. OPC UA Adapter mapuje cesty podle informačních modelů konkrétních OPC UA serverů, ke kterým vlastní klienty.
- Všechny úspěšně provedené operace vrací HTTP kód 200 (OK). Přístup k ostatním návratovým kódům je rozvinut v části *Návratové kódy*.
- Veškerá data v tělech HTTP zpráv jsou ve formátu JSON. Bližší informace lze nahlédnout v části *Serializace objektů*.
- Implementace metod PUT, DELETE musí dodržovat idempodenci. POST a GET zmíněný požadavek nemá. Což vzhledem k využití POST pro operaci *CallMethod* je na místě, jelikož není garantováno, že zavolaná metoda bude vždy vracet stejný výstup na stejný vstup. Obdobně POST pro *HistoryRead* se může nalézt v situaci, kdy část historie byla smazána v době mezi dvěma voláním této HTTP metody se stejnými parametry. Oproti tomu GET by podle obecně známých doporučení měl být idempodentní. Avšak adaptér typicky spravuje zařízení, kterými jsou různé měřiče, jenž přirozeně mohou v čase měnit hodnotu proměnné.

A konečně výčet operací.

- **/about/health**
  - GET: vrátí OK (200), pokud služba je dostupná.
- **/about/info**
  - GET: vrátí informace o adaptéru, např. verze nebo použitý protokol (*AdapterInfo*).
- **/node/{guid}/{nodepath}**

  - PUT: zapíše hodnotu v těle zprávy (*WriteRequest*), pokud je uzlem proměnná, a vrátí zpět hodnotu (*WriteResult*).
  - POST: zavolá metodu se vstupními argumenty reprezentovanými jako objekt v těle zprávy (*CallMethodRequest*), pokud je uzlem metoda, a vrátí výsledné návratové hodnoty (*CallMethodResult*).
  - GET: vrátí aktuální hodnotu a čas v těle zprávy (*ReadResult*), pokud je uzlem proměnná.

- **/node/history/{guid}/{nodepath}**
  - POST: vrátí hodnoty s časy (*HistoryReadResult*) z historie podle parametrů (*HistoryReadRequest*).

  Návratovou hodnotou ale nemusí být *HistoryReadResult*. Server vrátí chunked response a je na klientovi, zda ji agreguje do *HistoryReadResult* nebo s ní bude zacházet formou streamu.

- **/resource/{guid}**
  - GET: vrátí konfiguraci (*ResourceConfig*) zdroje pro zvolený server.
  - PUT: vytvoří/modifikuje daný zdroj/konfiguraci klienta (*ResourceConfig*).
  - DELETE: odstraní zdroj.

- **/resouce/{guid}/spec**
  - GET: vrátí specifikaci daného zdroje (*ResourceSpec*).

- **/resouce/guid/spec/signature**
  - GET: vrátí atribut *signature* z *ResourceSpec* pro daný zdroj.

  Operace *GET /resouce/{guid}/spec* může být časově nákladná, tudíž se hodí pomocí atributu *signature* ověřit, zda již danou specifikaci nevlastníme.

- **/resource/all**
  - GET: vrátí všechny uložené konfigurace zdrojů.

- **/pki/trusted/{guid}**
  - GET: vrátí certifikát s daným GUID (*CertData*).
  - PUT: vytvoří/modifikuje certifikát s daným GUID (*CertData*).
  - DELETE: smaže certifikát s daným GUID.

- **/pki/all/trusted/guid**
  - GET: vrátí GUID všech uložených certifikátů.

WebSocket

WebSocket umožňuje posílání zpráv/streamování oběma směry. Nicméně Adapter API dodržuje client-server komunikaci. Tudíž současná verze podporuje pouze half-closed kanály, kde tečou data od adaptéru k jeho klientovi. Jinými slovy režim Publisher-Subscriber. Jakmile klient, jakožto subscriber, ztratí o odběr dat zájem, ukončí komunikaci. Nutno ještě dodat, že WebSocket pro zahájení komunikace používá HTTP handshake, a tak lze operaci namapovat na klasickou HTTP cestu.[21] Níže přehled podporovaných operací.

- **/node/subscribe/guid/{nodepath}**

  Představuje operaci *Subscribe*, kdy adaptér v každé zprávě pošle novou hodnotu (*ReadResult*) proměnné, jenž je určena cestou *nodepath*. Na druhé straně cokoliv pošle klient adaptéru, je ignorováno.

5.1.4 Chybové stavy

Veškeré návratové kódy jsou popsány ve zmíněném dokumentu OpenAPI. Zde jsou rozebrány pouze predefinované chyby, které mohou nastat, u takřka libovolné operace. Návratové kódy jsou pro HTTP část, a tudíž odpovídají obvyklému užití HTTP kódů. WebSocket vrací vždy korektní hodnoty, ale může dojít k chybě při tvorbě spojení, nebo během spojení atp. – v těchto případech se řídí návratové hodnoty protokolem WebSocket.

Nicméně nedostatkem návratových kódů je jejich obecnost. Příkladem budiž „404 Not Found“ pro jakoukoliv operaci. Z tohoto statusu nelze zjistit, zda-li nebyl nalezen zdroj, nebo uzel. Adapter API takovéto situace řeší dodatečnou informací (*AdapterError*) v těle odpovědi. Objekt *AdapterError* obsahuje URI, které pomáhá identifikovat chybu napříč systémem, název chyby, zprávu s popisem chyby a kód, typicky HTTP status. Níže je slíbený rozbor předdefinovaných chyb.

Certifikát nenalezen

Pro kód je použit HTTP status „404 Not Found“ a pro URI *http://modemtec.cz/adapter-api/errors/CertNotFound* – chyba je na straně klienta. Certifikát se zadaným GUID nemůže adaptér najít.

Zdroj nenalezen

Pro kód je použit HTTP status „404 Not Found“ a pro URI *http://modemtec.cz/adapter-api/errors/ResourceNotFound* – chyba je na straně klienta. Zdroj se zadaným GUID nemůže adaptér najít.

Uzel nenalezen

Pro kód je použit HTTP status „404 Not Found“ a pro URI *http://modemtec.cz/adapter-api/errors/NodeNotFound* – chyba je na straně klienta. Uzel na daném zdroji a s danou cestou nemůže adaptér najít.

Nelze se připojit ke zdroji

Pro kód je použit HTTP status „503 Service Unavailable“ a pro URI *http://modemtec.cz/adapter-api/errors/ResourceNotConnected* – chyba je obvykle na straně serveru, ale špatnou konfiguraci zdroje mohl zaslat dříve již klient. Jedná se o situaci, kdy se adaptér nemůže připojit ke zdroji.

Nepodporovaná operace

Pro kód je použit HTTP status „405 Method Not Allowed“ a pro URI *http://modemtec.cz/adapter-api/errors/UnsupportedOperation* – chyba je na straně klienta. Uzel nepodporuje danou operaci, např. volá se vyčtení hodnotu na metodě.

Chyba v průběhu operace

Pro kód je použit HTTP status „500 Internal Server Error“ a pro URI *http://modemtec.cz/adapter-api/errors/ErrorDuringOperation* - chyba je na straně serveru. Vznikne v případě chyby v průběhu vykávání operace. Problém blíže specifikuje atribut zpráva.

Chybná vstupní data

Pro kód je použit HTTP status „400 Bad Request“ a pro URI *http://modemtec.cz/adapter-api/errors/BadInputData* – chyba je na straně klienta. Může se jednat o chybný formát dat v těle žádosti nebo chybnou hodnotu na vstupu operace.

Neznámá chyba

Pro kód je použit HTTP status „500 Internal Server Error“ a pro URI *http://modemtec.cz/adapter-api/errors/Unknown* – chyba je obvykle na straně serveru. Typicky se jedná o situaci, že není implementován převod chyby na *AdapterError*. V atributu zpráva se vyskytuje popis převzatý například ze zachycené výjimky.

5.1.5 Objekty

Následuje definice objektů, které operace využívají. Vzhledem k verzování API pomocí objektu *AdapterInfo* je možné tyto objekty měnit, aniž by se musely

měnit názvy nebo URI schémat. Budou-li navíc změny pouze ve smyslu přidání atributů, zachová se zpětná kompatibilita. Tedy objekty nové verze budou použitelné i pro starší verze Adapter API. Tudíž definice jsou minimální možné pro plnohodnotné používání adaptérů. Pro popis datových typů je použito následující značení:

- *Option[A]* značí hodnotu typu A, která může ale nemusí být přítomna;
- *Map[String, B]* značí slovník, kde klíč je typu *String* a hodnota typu B;
- *Any* značí libovolnou hodnotu, která musí být přítomna - pro hodnotu, jenž nemusí být přítomna, je použito značení *Option[Any]*;
- *Array[A]* značí pole hodnot typu A;
- *Set[A]* značí množinu hodnot typu A.

AdapterInfo

Objekt obsahuje obecné informace o adaptéru. Hodí se k registraci adaptéru nebo zjištění, který protokol transformuje.

atribut	typ	význam
protocol	String	protokol v rámci URL, např. opc.tcp, http
name	String	název adaptéru, např. OPC UA Adapter
adapterVersion	String	verze adaptéru, např. 0.2.0
apiVersion	String	verze Adapter API, které adaptér implementuje

Tabulka 5.1: Atributy AdapterInfo

ResourceConfig

Objekt obsahuje data potřebná k vytvoření spojení se zdrojem.

atribut	typ	význam
guid	String	globální identifikátor zdroje
connstr	String	řetezec obsahující informace k připojení
policy	Option[SecurityPolicy]	bezpečnostní politika
auth	Option[AuthConfig]	autentizace uživatele

Tabulka 5.2: Atributy ResourceConfig

AuthConfig

Objekt obsahuje data potřebná k autentizaci uživatele.

atribut	typ	význam
username	String	uživatelské jméno nebo alias
password	String	heslo

Tabulka 5.3: Atributy AuthConfig

SecurityPolicy

Objekt odpovídá enumeračnímu typu. Vyjadřuje zvolené šifrování.

enum hodnota
None
Basic256
Basic256Sha256
Aes128Sha256RsaOaep
Aes256Sha256RsaPss

Tabulka 5.4: Výčet SecurityPolicy

ReadResult

Objekt obsahuje data vyčtená v daném čase. Pokud atribut *time* je None, není znám čas vzniku hodnoty. Objekt využívá jak operace *Read*, tak operace *HistoryRead* nebo *Subscribe*.

atribut	typ	význam
value	Any	hodnota v daném čase
time	Option[DateTime]	čas hodnoty

Tabulka 5.5: Atributy ReadResult

WriteRequest

Objekt obsahuje vstupní data pro operaci *Write*, tedy data pro zápis hodnoty.

atribut	typ	význam
value	Any	nová hodnota

Tabulka 5.6: Atributy WriteRequest

WriteResult

Objekt obsahuje úspěšný výsledek operace *Write*, tedy data, jenž byla zapsána.

atribut	typ	význam
value	Any	zapsaná hodnota

Tabulka 5.7: Atributy WriteResult

CallMethodRequest

Objekt obsahuje vstupní data pro operaci *CallMethod*, tedy data reprezentující vstupní argumenty dané metody.

atribut	typ	význam
input	Map[String, Any]	vstupní parametry

Tabulka 5.8: Atributy CallMethodRequest

CallMethodResult

Objekt obsahuje úspěšný výsledek operace *CallMethod*, tedy data reprezentující výstupní argumenty dané metody.

atribut	typ	význam
output	Map[String, Any]	výstupní parametry

Tabulka 5.9: Atributy CallMethodResult

HistoryReadRequest

Objekt obsahuje vstupní data pro operaci *HistoryRead*, tedy filtr na výběr dat z historie.

atribut	typ	význam
from	DateTime	čas, včetně, odkud se má začít
to	DateTime	čas, včetně, kde se má skončit aktuální hodnoty

Tabulka 5.10: Atributy HistoryReadRequest

HistoryReadResult

Objekt obsahuje úspěný výsledek operace *HistoryRead*.

atribut	typ	význam
values	Array[ReadResult]	hodnoty v daném období

Tabulka 5.11: Atributy HistoryReadResult

ResourceSpec

Objekt obsahuje specifikaci zdroje: schémata datových typů a popis uzlů.

atribut	typ	význam
name	String	název
signature	String	unikátní řetězec pro specifikaci
description	String	popis
nodes	Map[String, NodeSpec]	popis uzlů
schemas	Map[String, SchemaSpec]	popis použitých datových typů

Tabulka 5.12: Atributy ResourceSpec

NodeSpec

Objekt představující společné atributy specifikace uzlů. Často se implementuje jako abstraktní třída nebo trait.

atribut	typ	význam
path	String	absolutní cesta k uzlu, např. ”/a/b/c”
description	String	popis uzlu
name	String	název uzlu, poslední položka v cestě, např. ”c”

Tabulka 5.13: Atributy NodeSpec

VariableSpec

Objekt popisuje specifikaci uzlu typu proměnná. Dědí od *NodeSpec*.

atribut	typ	význam
path	String	absolutní cesta k uzlu, např. ”/a/b/c”
description	String	popis uzlu
name	String	název uzlu, poslední položka v cestě, např. ”c”
schema	String	URI Json Schema pro datový typ proměnné

Tabulka 5.14: Atributy VariableSpec

MethodSpec

Objekt popisuje specifikaci uzlu typu metoda. Dědí od *NodeSpec*.

atribut	typ	význam
path	String	absolutní cesta k uzlu, např. ”/a/b/c”
description	String	popis uzlu
name	String	název, poslední položka v cestě, např. ”c”
input	Option[String]	URI Json Schema pro vstup
output	Option[String]	URI Json Schema pro výstup

Tabulka 5.15: Atributy MethodSpec

SchemaSpec

Objekt popisuje specifikaci datového typu.

atribut	typ	význam
uri	String	unikátní URI (garantováno v rámci zdroje)
name	String	název
schema	Any	JSON Schema – URI je shodné s *uri*
dependencies	Set[String]	množina URI datových typů

Tabulka 5.16: Atributy SchemaSpec

AdapterError

Objekt popisuje výjimku/chybu na začátku, v průběhu či na konci vykonávání operace.

atribut	typ	význam
uri	String	unikátní URI (garantováno v rámci adaptéru)
code	Number	kód chyby, typicky odpovídající HTTP Status Code
name	String	název
message	String	zpráva/další informace

Tabulka 5.17: Atributy AdapterError

CertData

Objekt popisuje X.509 certifikát a přidružená metadata.

atribut	typ	význam
cert	String	X.509 certifikát v DER formátu v Base64 řetězci
guid	String	unikátní řetězec pro specifikaci

Tabulka 5.18: Atributy CertData

5.1.6 Serializace objektů

Objekty jsou kódovány v JSON formátu. Názvy atributů objektů odpovídají názvům atributů JSON objektů. Pro atributy a jejich hodnoty jsou aplikovány následující pravidla:

- jestliže *Option[A]* má přitomnou hodnotu, pak je kódován pro hodnotu typu *A* - naopak není-li přítomna, atribut není ve výsledném *JSON Object* přítomen;
- *Map[String, B]* je kódován jako *JSON Object* - výjimku tvoří slovníky v objektu *ResouceSpec*, které jsou serializovány jako *JSON Array* s hodnotami kódovanými pro typ *B*;
- *DateTime* je kódován jako *JSON String* s formátem podle normy ISO 8601 s UTC: *"yyyy-MM-ddTHH:mm:ssZ"*, např. *"2022-06-27T15:16:00Z"*.
- *Array[A]* je kódován jako *JSON Array* s hodnotami kódovanými pro typ *A*;
- *Set[A]* je kódován jako *JSON Array* s hodnotami kódovanými pro typ *A*;
- *Any* je kódováno podle těchto pravidel;
- jinak je použito obvyklé JSON kódování, např. *String* je kódován jako *JSON String*.

U JSON formátu, narozdíl od binárních kódování, nezáleží na pořadí atributů v objektu.

5.1.7 Bezpečnost

Otázka bezpečnosti je ponechána čistě na implementaci adaptéru. Adaptér nemusí být zabezpečen vůbec, nebo může použit HTTPS protokol či JSON Web Token.

5.2 Knihovna ve Scale

Projekt Adapter API obsahuje také i knihovny s klienty pro Adapter API a základní adaptéry, např. OPC UA Adapter. V současné chvíli se na seznamu nachází pouze klient implementovaný v jazyce Scala 3.

Klient napsaný v jazyce Scala implementuje Adapter API verze 0.1.0. Využívá především projekty Play Framework a Pekko. Z Play si bere implementaci klienta pro webové služby a podporu JSON formátu. Což umožňuje hladké začlenění knihovny do aplikací postavených nad Play Framework pracující s HTTP. Pekko, kterou rovněž využívá i Play, dodává podporu streamů a protokolu WebSocket.[8]

5.2.1 Návratové hodnoty a monády

Hlavním rysem implementace je užití funkcionálních prvků, nejčastěji monády. Pomocí monád se ve Scale řeší asynchronní kód (třída *Future[A]*) nebo zpracování chybových stavů (lze pomocí tříd *Either[A, B]*, *Try[A]* nebo *Option[A]*). V případě této knihovny se pro chybové stavy užívá monáda *Either[AdapterError, A]*, kde *A* je návratová hodnota v případě, že nedošlo k chybě. *AdapterError* obsahuje popis chyby (viz. návrh API). Vzhledem k tomu, že operace klienta jsou vyhodnocovány asynchronně, výsledkem může být typ *Future[Either[AdapterError, A]]*. Nicméně takto vnořené monády mohou být obtížnější pro programátora na zápis. Řešením jsou tzv. Monad Transformers, které umožňují přístup přímo k zanořené hodnotě - v tomto případě instanci *A*. Implementaci Monad Transformers nabízí knihovna Cats, která krom nich obsahuje i další funkce a třídy užitečné pro funkcionální paradigma.[3] *EitherT[Future, A, B]* implementuje transformace monády *Future[Either[A, B]]*. Lze tak používat i for-comprehension pro sekvenční zápis monád (viz část *Ukázky použití klienta*). Níže ukázka využití v definici typu.

```
object MonadTransformer extends MonadTransformer

trait MonadTransformer:

  type FutureEither[L, R] = EitherT[Future, L, R]

  extension [L, R](self: Future[Either[L, R]])
    def asFutureEither: FutureEither[L, R] = EitherT[Future, L, R](self)

  object FutureEither:

    def unit[L]: FutureEither[L, Unit] =
      Future.successful(Right(())).asFutureEither

    def right[L, R](value: R): FutureEither[L, R] =
      Future.successful(Right(value)).asFutureEither

    def left[L, R](value: L): FutureEither[L, R] =
      Future.successful(Left(value)).asFutureEither

    def sequence[L, R](items: Iterable[FutureEither[L, R]])
      (using context: ExecutionContext): FutureEither[L, Seq[R]] =
      items
        .map(_.value)
        .pipe: futures =>
          Future.sequence(futures)
        .map: eithers =>
          eithers.foldLeft(Right(Seq.empty): Either[L, Seq[R]]): (acc, either) =>
            for
              items <- acc
              item  <- either
```

```
            yield items :+ item
        .asFutureEither
```

Výpis kódu 5.4: Definice třídy MonadTransformer

5.2.2 Ukázky použití klienta

Níže je uvedeno pár příkladů použití knihovny. Části kódu jsou převzaty z testů OPC UA Adapter.

Tvorba klienta

K vytvoření klienta je nutný webový klient z Play, port, kde se služba nachází a IP adresa služby. Z nich je vytvořen prefix pro operace v protokolu HTTP a WebSocket. Implicitními parametry jsou ExecutionContext pro správu asynchronního výpočtu a ActorSystem – ten běží na pozadí Play aplikace, nicméně zde je potřeba kvůli streamům a WebSocket.

```
class CommonAdapterClient(
  val webservice: WSClient,
  val port: Int,
  val address: String
)(using val executionContext: ExecutionContext, val system: ActorSystem)
  extends AdapterClient:

  protected val httpPrefix: String = s"http://$address:$port"
  protected val webSocketPrefix: String = s"ws://$address:$port"

// code ...
```

Výpis kódu 5.5: Parametry konstruktoru CommonAdapterClient

Například při testovaní adaptéru v rámci aplikace v Play lze inicializovat testy v sadě následovně.

```
"Test name" in new WithServer(app = AppWith("test.conf"), port = testPort):
  override def running(): Unit =
    given actors: ActorSystem = app.actorSystem
    val guid = "Some server identifier"
    val ws = app.injector.instanceOf[WSClient]
    val client = OpcUaAdapterClient(ws, port, "localhost")

    // code ...
```

Výpis kódu 5.6: Inicializace klienta v testech Play

Což vytvoří aplikaci s danou konfigurací v *test.conf* na daném portu. K tomu Play umožní vytvořit klienta pro webovou službu, nad kterým je naprogramován klient pro adaptér.

Vyčtení historie (*HistoryRead*)

Operace je definována následovně.

```
trait AdapterClient:

  /** Reads variable's history values.
    *
    * @param guid Resource identifier.
    * @param path Path to variable.
    * @param request Request data.
    * @return Stream of history values.
    */
  def historyRead(guid: String, path: String, request: HistoryReadRequest):
    FutureEither[AdapterError, Source[ReadResult, _]]

  // other methods ...
```

Výpis kódu 5.7: Deklarace metody historyRead

Historie je vyčítána postupně a vracena přes tzv. chunked HTTP response. Klient stream dat rozčlení do jednotlivých hodnot, deserealizuje z JSON a poskytne tento stream. Na straně uživatele je poté možné vytvořit plán streamu - co a jak se s ním má provést.

```
//  client creation ...

val expected = // expected values ...

val config =
  ResourceConfig(
    guid    = guid,
    connstr = "opc.tcp://localhost:4840/",
    policy  = None,
    auth    = None)

val request =
  HistoryReadRequest(
    from = SqlTimestamp.valueOf("2000-01-01 00:00:00"),
    to   = SqlTimestamp.valueOf("2025-01-01 00:00:00"))

val sink =
  Sink
    .fold[HistoryReadResult, ReadResult](HistoryReadResult.empty):
      (result, value) =>
        result.copy(values = result.values :+ value)

val action = for
  _      <- client.createOrUpdateResource(config)
  source <- client.historyRead(guid, "/PDM1/SPDC1/Beh/q", request)
yield source
```

```
val future =
  action
    .value
        .flatMap:
          case Left(error)   => throw error
          case Right(source) => source.runWith(sink)

val history = awaitFor(future)

val result = history.values.foldLeft(Vector.empty[JsValue]):
  (values, readResult) =>
  println(readResult)
  values :+ readResult.value

result mustBe expected
```

Výpis kódu 5.8: Vyčtení historie v testech Play

Odběr novinek (*Subscribe*)

Operace *Subscribe* využívá Pekko WebSockets. Protokol je obousměrný – vytvoří se stream od klienta k serveru a od serveru ke klientovi. Možným úskalím implementace v Pekko je, že komunikace skončí v okamžiku, kdy jeden z těchto streamů je uzavřen. *Subscribe* však streamuje data pouze ze strany serveru, tzv. half-closed WebSockets. Aby se předešlo ukončení odběru, klient vytvoří instanci *Promise*. Stream dat od klienta k serveru je tedy v nekonečném očekávání odeslání hodnoty. Pokud-li uživatel chce *Subscribe* operaci ukončit, úspěšně dokončí *Promise* přes metodu *promise.success(None)*.

```
trait AdapterClient:

  /** Subscribes variable's value.
    *
    * @param guid Resource identifier.
    * @param path Path to variable.
    * @param sink Stream's sink.
    * @tparam Mat Materialized type.
    * @return Promise used as cancellation switch -
    *         call {{{promise.success(None)}}} in order to cancel subscription.
    */
  def subscribe[Mat](guid: String, path: String, sink: Sink[ReadResult, Mat]):
    FutureEither[AdapterError, Promise[Option[Message]]]

  // other methods ...
```

Výpis kódu 5.9: Deklarace metody subscribe

V kódu níže lze vidět celý příklad použití streamování a způsob ukončení.

```
//  client creation ...

val config:  =
  ResourceConfig(
    guid    = guid,
    connstr = "opc.tcp://localhost:4840/",
    policy  = None,
    auth    = None)

val expected =
  Vector(JsNumber(0), JsNumber(1), JsNumber(2), JsNumber(3), JsNumber(4))
var values = Vector.empty[ReadResult]

val sink = Sink.foreach[ReadResult]: message =>
  values = values :+ message

val cancellation = for
  _       <- client.createOrUpdateResource(config)
  promise <- client.subscribe(guid, "/PDM1/SPDC1/OpCnt/stVal", sink)
yield promise

val promise = awaitFor(cancellation.value) match
  case Left(error)  => throw error
  case Right(value) => value

Thread.sleep(10000)

promise.success(None)

values.map(_.value) containsSlice expected mustBe true
```

Výpis kódu 5.10: Odběr novinek v testech Play

Vyčtení hodnoty (*Read*)

Pouhé vyčtení aktuální hodnoty proměnné představuje metoda *readValue*.

```
trait AdapterClient:

  /** Reads variable's value.
    *
    * @param guid Resource identifier.
    * @param path Path to variable.
    * @return Read value.
    */
  def readValue(guid: String, path: String):
    FutureEither[AdapterError, ReadResult]

// other methods ...
```

Výpis kódu 5.11: Deklarace metody readValue

Níže je k náhlednutí kód, který tuto metodu zavolá.

```
//  client creation ...

val config: ResourceConfig =
  ResourceConfig(
    guid    = guid,
    connstr = "opc.tcp://localhost:4840/",
    policy  = None,
    auth    = None)

val expected =
  Json
    .obj("value" -> JsNumber(3), "name" -> JsString("test"))
    .asRight

val action = for
  _      <- client.createOrUpdateResource(config)
  result <- client.readValue(guid, "/PDM1/LLN0/Beh/stVal")
yield result.value

val result = awaitFor(action.value)

result mustBe expected
```

Výpis kódu 5.12: Vyčtení hodnoty v testech Play

Kapitola 6

OPC UA Adapter

Tato kapitola se nejdříve věnuje popisu a lehkému hodnocení výchozího stavu projektu aplikace, jenž měla za úkol organizovat úlohy pro vyčítání dat a poskytovat uložiště pro tyto data. V dalších částech kapitoly je text věnovaný provedeným úpravám dle nového návrhu systému. Jednotlivé úkony vývoje od návrhu doménového modelu, architektury přes implementaci a nasazení jsou uvedeny v chronologickém pořadí.

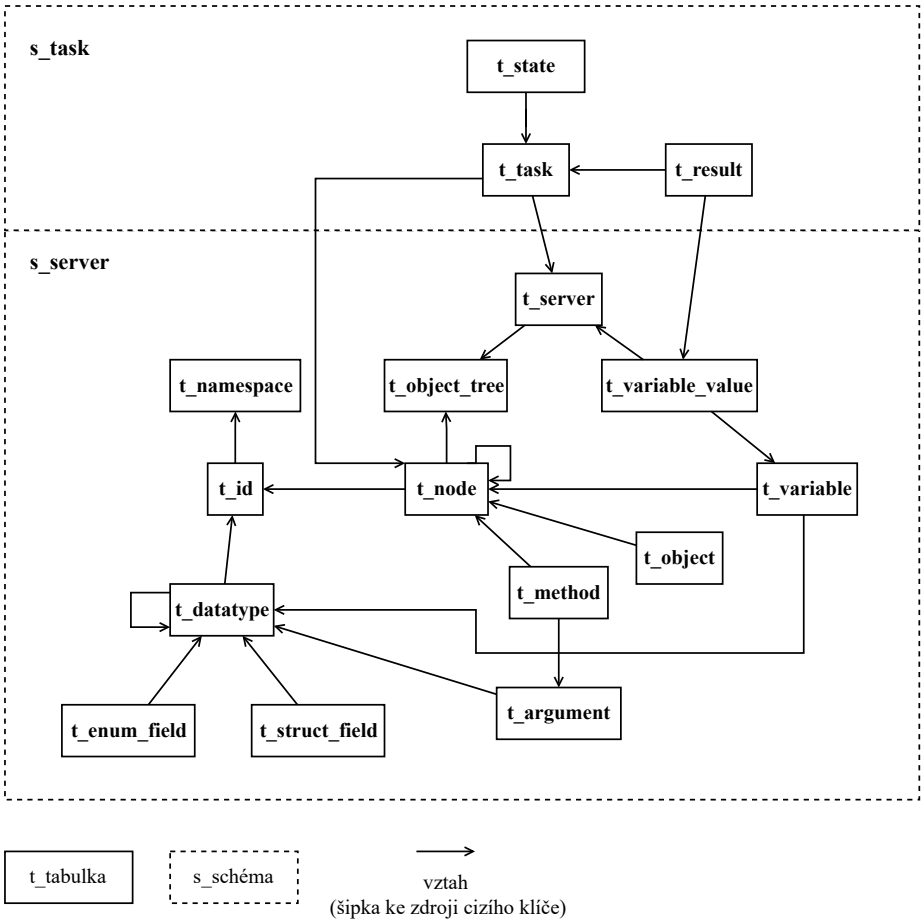
6.1 Výchozí stav

Původně projekt je koncipován jako konzolová aplikace, která poskytuje funkce správy úloh pro vyčítání dat a uložiště pro nasbíraná data. Veškerá data a podporované protokoly jsou pouze z rodiny standardů OPC UA. Nicméně dá se říci, že první verze je spíše verzí tzv. prošlapující cestu – vyzkoušení způsobu řešení, zjištění vzniku dalších požadavků na systém nebo otestování dílčích projektů a knihoven. Nejedná se tedy o finální produkt. Na dalších řádkách je stručně připomenut návrh a implementace výchozí aplikace. Podrobnější informace lze nastudovat v kapitole 7 „Aplikace“ a kapitole 5 „Databáze“ v bakalářské práci „Systém pro sběr dat s využitím OPC UA“.[2]

6.1.1 Datový model

Datový model vychází z popisu původní domény. Skládá se ze tří logických částí. První oblast zájmu je uložení informačního modelu OPC UA jako takového. Druhou částí je zaměření se na reprezentaci informace ohledně připojení k serverům a naměřené hodnoty pro daný uzel z informačního modelu, který využívá daný server. A konečně třetí částí budiž správa úloh pro sběr dat včetně reprezentace stavů, ve kterých se úloha může nacházet. Z tohoto posléze vznikají samotné třídy v rámci zdrojového kódu a schéma relační databáze. Obrázek 6.1 zobrazuje celkový pohled na databázové schéma původní

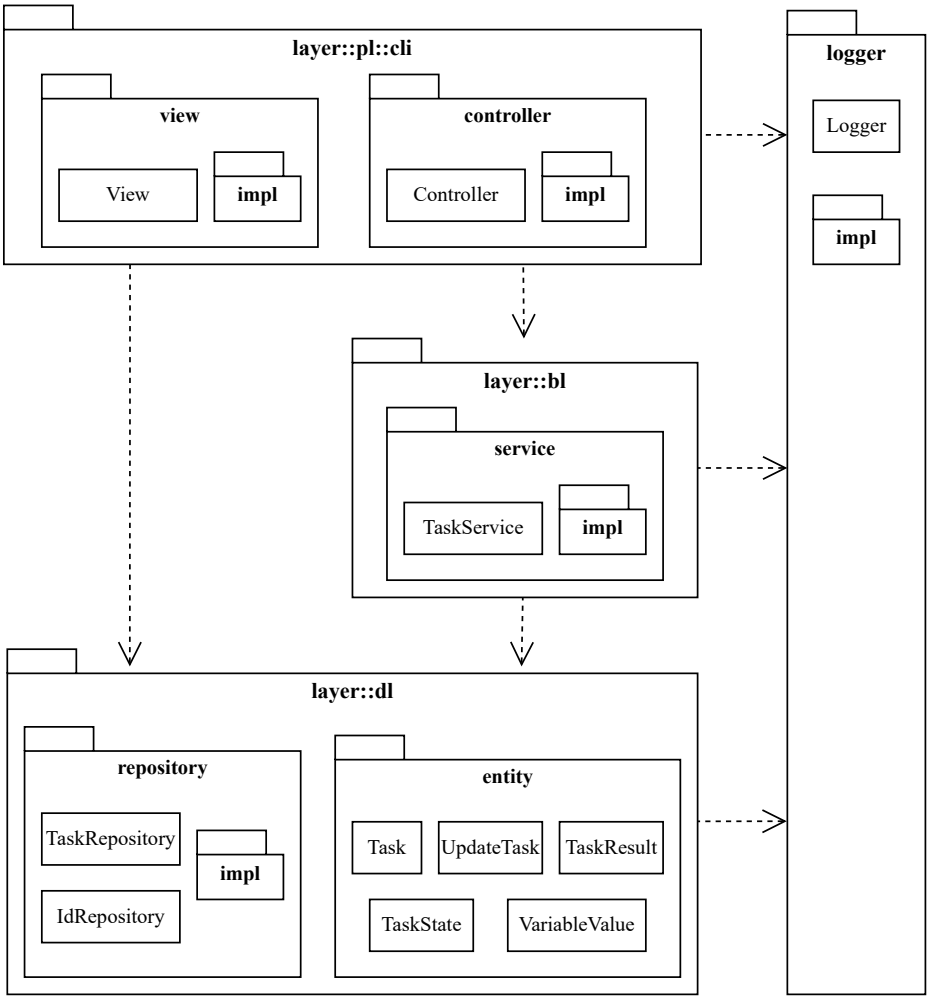
aplikace. Třídy v rámci aplikace jsou více či méně ekvivalentní (někde se nejedná o třídu, nýbrž o abstraktní třídu atp.).[2]



Obrázek 6.1: Databázové schéma původní aplikace

6.1.2 Architektura

Architektura je navržena jako třívrstvá. Spodní, datová, vrstva představuje přístup k databázi. Prostřední, byznys, vrstva zaštiťuje připojení k OPC UA serverům a vykonání úloh. Zatímco horní vrstva, prezentační, nabízí rozhraní pomocí CLI. Současně existuje tzv. cross-cutting concern v podobě logování napříč vrstvami. Na tomto místě je třeba dále uvést, že byť řešení obsahovalo správu úloh, aplikace poskytovala synchronní rozhraní. K tomu navíc, připojení k vybranému OPC UA serveru probíhalo vždy dočasně na dobu provádění operace.[2] Což by při vysoké míře operací nad jednom serverem znamenalo, že bude alokováno zbytečně mnoho zdrojů pro jeden server. Obrázek 6.2 níže zobrazuje strukturu balíčků a jejich závislosti.



Obrázek 6.2: Původní architektura aplikace (UML)

6.1.3 Implementace

Začne-li se od spodních vrstev architektury. Pro databázi je použit server PostgreSQL. Aplikace napsaná v jazyce Scala 2.13 využívá knihovnu Slick pro funkcioálně-relační mapování. Knihovna neposkytuje příliš automatizovaný kód v podobě anotací jako různé javovské frameworky pro mapování objektů na relační tabulky. Nicméně umožňuje namapování záznamů tabulek na kolekce. Vrstva byznys logiky využívá knihovnu Scalable OPC UA pro připojení k OPC UA serverům synchronním způsobem, jak je popsáno v předchozí části. Prezentační vrstva je pouhé naprogramování příkazů do CLI. Celkově aplikace pro svou jednoduchost nevyužívá ani framework pro dependency injection. Injektáž objektů je řešena manuálně jednotkami řádek kódu.[2]

6.2 Důsledek a požadavky pro OPC UA Adapter

V konečném důsledku je OPC UA Adapter navrhnut a implementován tzv. „na zelené louce“, jelikož původní aplikace je pouhým vyzkoušením, případně nástinem, možného řešení[2] a v návrhu systému se ukazuje, že je nutno celý systém dekomponovat do vícero dílčích komponent. Z toho důvodu OPC UA Adapter již nepodporuje správu úloh a uložení vyčtení dat. Pouze poskytuje API společné pro všechny adaptéry, na které je namapován OPC UA protokol. Dalším požadavkem na adaptér je jeho tvorba jako webový server, tudíž již se nejedná o konzolovou aplikaci. A konečně vzhledem k REST API adaptér poskytuje správu klientů k OPC UA serverům – jejich automatickou tvorbu či mazání na základě uložených informací k připojení v adaptéru.

6.3 Doménový model

OPC UA Adapter implementuje Adapter API, s čímž souvisí i převzetí datových tříd pro specifikaci zdrojů, uzlů a operací v rámci Adapter API. K tomu navíc je nutné udržovat metadata o uložených informačních modelech a datových typech OPC UA z vícero důvodů. Za prvé identifikace uzlu podle cesty. Za druhé vyhledávat datové typy a transformovat podle definic data z/do binárního formátu OPC UA. Za třetí schopnost generovat specifikace zdrojů a uzlů z informačních modelů, aby uživatel nebo jiný program komunikující s adaptérem měl přístup k metadatům o konkrétní struktuře API poskytované OPC UA Adapterem. Na diagramu 6.3 je představen UML diagram tříd domény. Lze ho rozlišit na dvě oblasti zájmu: uložení informačního modelu (uzly, datové typy, jmenné prostory) a připojení k OPC UA serverům (zdroje, certifikáty, jmenné prostory využívané zdroji).

Namespace

Třída *Namespace* reprezentuje jmenný prostor v rámci OPC UA informačního modelu. Jmenné prostory zajišťují odlišení stejných názvů uzlů/datových typů napříč informačním model, jenž takovýchto jmenných prostorů může obsahovat vícero.[19]

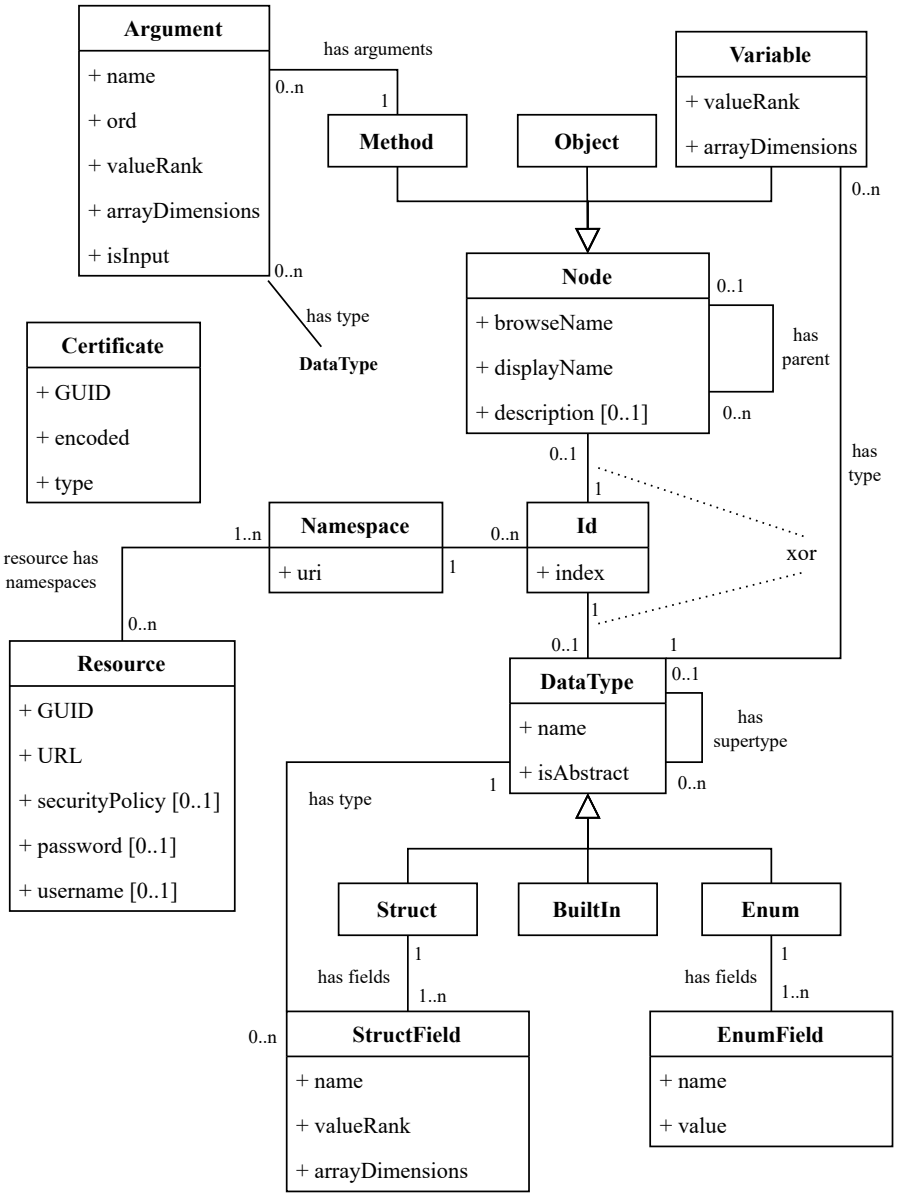
Id

Třída *Id* představuje identifikaci uzlu nebo datového typu v rámci OPC UA. Index je číselná hodnota unikátní rámci jmenného prostoru, do kterého *Id* je přiřazeno.

DataType

Třída *DataType* je abstraktní a schovává se pod ní hierarchie datových typů pro tuto aplikaci. Rozlišuje se na enumerační datový typ (*Enum*), struktu-

rovaný datový typ (*Struct*) a vestavěný (*BuiltIn*). Vestavěný typ patří mezi základní typy OPC UA. Byť se u něj může jednat o strukturu nebo enumeraci, standard uvádí, že v informačních modelech není třeba ho definovat, stačí pouze uvést. Pro potřeby automatizovaného generování specifikace zdroje však adaptér potřebuje tuto definici znát.



Obrázek 6.3: Doménový model (UML)

Node

Třída *Node* je abstraktní a vyrůstá z ní hierarchie typů uzlu. Adaptér si pro potřeby generování specifikace zdroje v rámci Adapter API ukládá pouze uzly typu *Method*, *Object* a *Variable*. Transformace informačního modelu OPC UA do domény je pouze jednosměrná.

Resource

Třída *Resource* reprezentuje zdroj z Adapter API, kde je i blíže popsána.

Certificate

Třída *Certificate* obsahuje certifikát, který adaptér může potřebovat během připojení ke zdroji, resp. OPC UA serveru. V rámci systému lze certifikát identifikovat jednoznačně přes uživatelem definované GUID.

6.4 Architektura

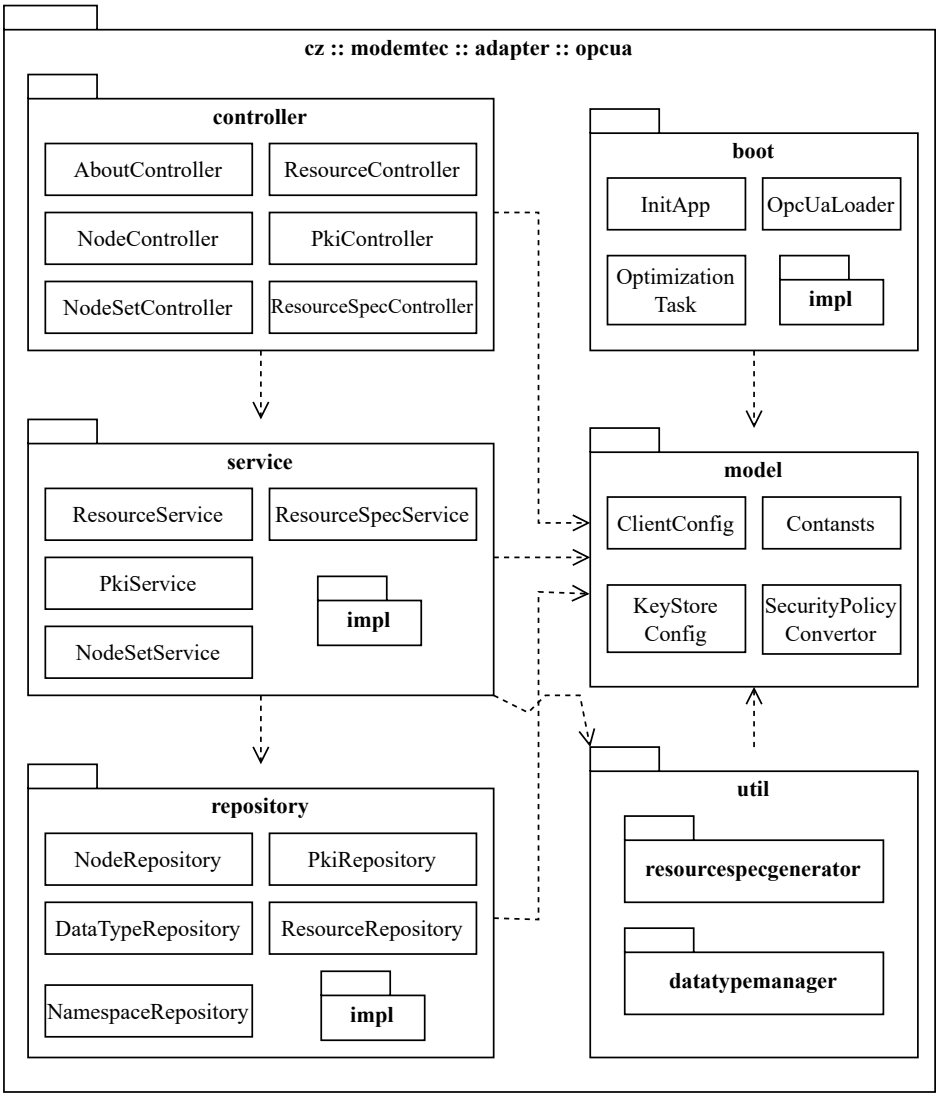
Architekturou OPC UA Adapteru je zvolena tří vrstvá architektura, jelikož základní rozdělení funkcionalit, resp. tříd, je do balíčků, které nevyžadují nic složitějšího než pouhý přístup k datové perzistenci, provést logiku aplikace a poskytnout komunikační rozhraní. Krom tohoto je zvláště dán balíček pro utility poskytované napříč aplikací (například různé generátory či pomocné třídy) a balíček pro inicializace aplikace – načtení výchozích dat do databáze či spuštění optimalizačních úloh na pozadí. Pod optimalizační úlohou si čtenář může představit pravidelné čištění paměti aplikace, která může mít nakešované definice datových typů OPC UA, nebo pravidelné odpojení a smazání nepoužívaných klientů OPC UA. Jedním z cílů při návrhu Adapter API je právě i enkapsulace optimalizačních funkcí, které závisí čistě na implementaci adaptéru a uživatel o nich nemusí vědět.

Přiřazení balíčkům k vrstvám je uvedeno níže. Pro úplnost cross-cutting concerns se jako vrstva nevnímá, ale je na místě tuto část zmínit, jelikož prostupuje celou aplikací.

- prezentační: *cz.modemtec.adapter.opcua.controller*;
- byznys: *cz.modemtec.adapter.opcua.service*;
- datová: *cz.modemtec.adapter.opcua.*{*repository, model*};
- cross-cutting: *cz.modemtec.adapter.opcua.*{*boot, util*}.

Na obrázku 6.4 je zobrazen UML diagram balíčků. Balíčky nejsou explicitně rozděleny dle názvů jednotlivých vrstev, ale přímo dle použití, jak je

tomu obvyklé u webových aplikací, např. byznys vrstva obsahuje balíček *service*. Na diagramu lze rovněž vidět rozhraní jednotlivých komponent, konkrétní implementace jsou zařazeny do balíčků *impl* nacházející se ve stejném balíčku, jako rozhraní. Výjimku tvoří kontroléry a datové modely, kde jsou přímo zobrazeny komponenty, resp. třídy. Dále lze vyčíst z diagramu závislosti mezi balíčky ve směru šipek.



Obrázek 6.4: Architektura OPC UA Adapter (UML)

6.4.1 Prezentační vrstva

Prezentační vrstva implementuje Adapter API. Nicméně v tomto případě je nutné rozšířit API o část, která přidává podporu pro zobrazení, přidání a odebrání informačního modelu. Informační model je přenášen v NodeSet XML souborech dané standardem OPC UA[19]. OPC UA Adapter z nich získává jednak popis uzlů a jednak definice datových typů, které jsou nutné pro de/serializaci dat z/do binárního formátu, jenž používají produkční OPC UA servery. Rozšíření Adapter API je definováno níže.

- **/nodeset**
  - PUT: uloží uzly/jmenný prostor obsažený v NodeSet XML, které se nachází v těle zprávy.
- **/nodeset/all**
  - GET: vrátí množinu URI jmenných prostorů.
- **/delete/nodeset**
  - POST: smaže daný jmenný prostor, pokud na něj neexistují závislosti z jiných uložených jmenných prostorů. Vzhledem k tomu, že URI není možné přenášet v rámci URL operace, je obsaženo v těle zprávy.

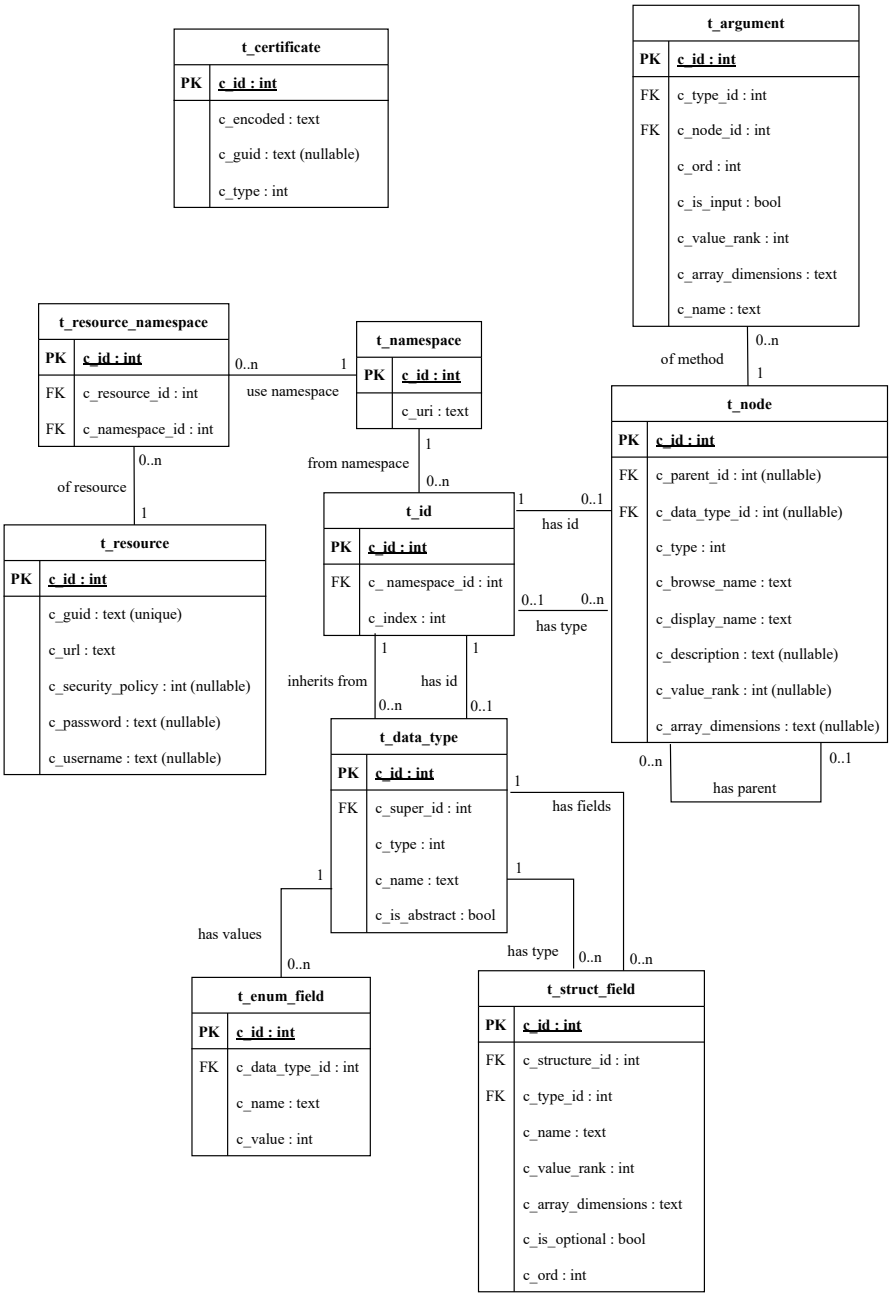
6.4.2 Byznys vrstva

Vrstva aplikační logiky poskytuje jednak enkapsulaci přístupu k datové vrstvě pro prezentační vrstvu, jednak správu OPC UA klientů (nebo jinak řečeno zdrojů) a správu podporovaných datových typů. Rovněž na této úrovni probíhá optimalizace. V současné verzi se týká množství alokovaných klientů. OPC UA protokol při komunikaci vytváří spojení, které je udržováno (keep-alive zprávy), je k němu spjatý stav, kódování atp., čili vytvořit toto spojení je relativně nákladné oproti restful API.[19] Je tedy výhodné si klienta držet nějakou dobu v paměti a po určité čase zrušit spojení s OPC UA serverem a dealokovat ho. K tomu by měla sloužit optimalizační úloha (*OptimizationTask*), jenž se spustí v nakonfigurovaném časovém intervalu a zkontroluje, který z klientů neprovádí žádnou operaci a dealokuje ho.

6.4.3 Datová vrstva

Datová vrstva poskytuje uložení konfigurace připojení k OPC UA serverům, certifikátů a přihlašovacích údajů, dále pak uložení informačních modelů. Entitně vztahový model 6.5 vychází z doménového modelu 6.3, jelikož současný návrh počítá s lehkou relační databází typu SQLite. Nicméně stojí za zvážení

do budoucna, zda by nebyla vhodnější grafová databáze. Důvodem této úvahy je, že nejnáročnější operace nad databází je identifikovat uzel na základě cesty. K čemuž se využije buď to rekurzivní dotaz, má-li ho databáze ve svém jazyce, nebo opakované dotazování na rodiče/syna uzlu, či vyčíst celý informační model pro daný uzel naráz a nad ním provést hledání.



Obrázek 6.5: Schéma relační databáze (UML)

6.4.4 Cross-cutting concerns

Mezi typické cross-cutting concerns patří loggování a různé pomocné utility. V tomto případě do nich jsou zahrnuty třídy z balíčku *boot*, který obsahuje inicializaci databáze, datových typů OPC UA, klienty k OPC UA serverům a vznik/zánik optimalizační úlohy zmíněné v byznys vrstvě.

6.5 Implementace

Implementace je provedena v jazyce Scala 3 – aplikace používá framework Play a knihovnu Slick pro přístup k databázi. Databází je zvolena relační SQLite. Představu, jak použití zmíněných technologií vypadá, čtenář nabude v kapitole *Použité technologie*, a tak zde je zmíněno pár význačných informací. Implementace je provedena tzv. „na zelené louce“, jelikož významnými prvky oproti výchozímu stavu jsou asynchronní chování OPC UA klientů, změna prezentační vrstvy z CLI do REST serveru a DI pomocí frameworku. To vše se stalo předchozí verzi neslučitelné se životem. Projekt OPC UA Adapter využívá knihovnu Adapter API, kde se nachází klient a datové třídy (viz kapitola *Adapter API*). K tomu navíc i knihovnu Scalable OPC UA pro podporu OPC UA klienta. O ní si lze dočíst podrobnější informace ve stejnojmenné kapitole.

6.6 Testování

Během testování jsou kombinovány dva druhy testů: jednotkové a integrační. Zatímco jednotkové se používají k otestovaní funkcionalit pomocných tříd a funkcí, integrační jsou určeny pro testování vrstev vůči rozhraní. Současně existují i testy, kdy je sestavena celá aplikace a testy jsou spuštěny vůči serveru. Pro testování kódu ve Scale se používá defaultně knihovna ScalaTest, která poskytuje několik různých stylů dle cíleného typu testů, např. styl pro psaní akceptačních testů, styl pro funkcionální testování atd.[22]

Pro účely testování celé aplikace je použit styl *FlatSpec*, který vychází z tzv. *Behavior-Driven Development* (BDD), kdy pomocí DSL je daný test popsán způsobem, že čtenář si přečte, která utilita má, co dělat, jaké má mít vlastnosti atp.[22] V podobné duchu Play Framework nabízí *PlaySpec*, jenž je určena pro testování spuštěného serveru nebo kontrolérů.[13] Oproti tomu pro integrační testy rozhraní a jednotkové testy utilit je použit styl *FunSuite*, který je mezikrokem mezi tradičními javovskými xUnit a zmíněným BDD. Uvedené druhy existují jak v synchronní, tak v asynchronní variantě.[22]

Pro demonstraci *PlaySpec* je níže uveden test na vyčtení hodnoty z uzlu z rozeběhnutého serveru. Pro úplnost: funkce *WithApp* vytváří aplikaci s daným seznamem cest, routerem a vybranými komponentami (controllers, services, ...).

```
"Read value" in new WithServer(
  app = AppWith("config/test_opcua.conf"),
  port = testPort):

  override def running(): Unit =
    given actors: ActorSystem = app.actorSystem

    val guid = "testServer"
    val ws = app.injector.instanceOf[WSClient]
    val client = OpcUaAdapterClient(ws, port, "localhost")

    val config =
      ResourceConfig(
        guid    = guid,
        connstr = "opc.tcp://localhost:4840/",
        policy  = None,
        auth    = None)

    val expected = Json.obj(
      "value" -> JsNumber(3),
      "name" -> JsString("test")
    ).asRight

    val action = for
      _      <- client.createOrUpdateResource(config)
      result <- client.readValue(guid, "/PDM1/LLN0/Beh/stVal")
    yield result.value

    val result = awaitFor(action.value)
    result mustBe expected
```

Výpis kódu 6.1: PlaySpec pro test běžící aplikace

Na dalším příkladě je zobrazen *AsyncFunSuite* styl, jelikož OPC UA Adapter je implementován asynchronně a návratové hodnoty metod komponent vrací *Future[A]*, případně Monad Transformer obsahující taktéž *Future[A]*. Metoda *createService* vytváří potřebné komponenty pro integrační testování služby *ResourceService*.

```
test("Create resource and read node /PDM1/LLN0/Beh/stVal", NeedsRunningOpcUa):
  val service = createService()
  val guid = "test-server-open62541"
  val path = "PDM1/LLN0/Beh/stVal"

  val expected = Json.obj(
    "value" -> JsNumber(3),
    "name" -> JsString("test")
  ).asRight
```

```
val config = ResourceConfig(
  guid    = guid,
  connstr = "opc.tcp://127.0.0.1:4840",
  policy  = None,
  auth    = None)

val action = for
  _      <- service.createOrUpdateResource(config)
  _      <- service.checkResource(guid)
  nodeId <- service.readNodeId(guid, path)
  result <- service.read(guid, nodeId)
  _      <- service.deleteResource(guid)
yield result.value

action
  .value
  .map: result =>
    result shouldBe expected
```

Výpis kódu 6.2: AsyncFunSuite pro integrační test ResourceService

6.7 Nasazení

K nasazení je použit Docker. Přesto zde pro účely prvotního vyzkoušení a případného solitérního nasazení je uveden způsob dockerizace aplikace v Play Frameworku. Nástroj sbt, jenž je typicky používán pro sestavování software naprogramovaný ve Scale, podporuje rozšíření pro Docker. Je tak možné pokyny pro dockerizaci napsat přímo v souboru *build.sbt*, který již poté vygeneruje patřičný *dockerfile*. V sbt terminálu pak lze přímo i volat operace, které provedou samotné nasazení do Dockeru.[23] Ve výňatku kódu ze souboru *build.sbt* se nachází pokyny pro dockerizaci OPC UA Adapteru.

```
import com.typesafe.sbt.packager.docker.DockerChmodType
import com.typesafe.sbt.packager.docker.DockerPermissionStrategy

dockerChmodType := DockerChmodType.UserGroupWriteExecute
dockerPermissionStrategy := DockerPermissionStrategy.CopyChown

Docker / maintainer := "dominik.codl@modemtec.cz"
Docker / packageName := "opc-ua-adapter"
Docker / version := sys.env.getOrElse("BUILD_NUMBER", "0")
Docker / daemonUserUid  := None
Docker / daemonUser := "daemon"

dockerExposedPorts := Seq(9000)
dockerBaseImage := "eclipse-temurin:17-jre-alpine"
dockerRepository := sys.env.get("ecr_repo")
dockerUpdateLatest := true
```

Výpis kódu 6.3: Dockerizace v rámci sbt

Kapitola 7

Další vývoj a možné vylepšení

Kapitola popisuje další oblasti směřování vývoje, případně nápady na možná vylepšení systému. Pro jednoduchost jsou dále představeny položky nezávisle na projektu, kterých se týkají. Některé se mohou týkat konkrétních projektů a některé systému jako celku.

7.1 Dokumentace

Tato diplomová práce lze použít i jako projektová dokumentace. Část informací, tedy popis OPC UA standardu a podrobný popis řešení, které zůstalo nezměněno či bylo pouze mírně upraveno, se nachází v bakalářské práci „Systém pro sběr dat s využitím OPC UA“.[2] Pro další vývoj je vhodné vytvořit jednotnou dokumentaci pro celý systém – ať už formou webové stránky nebo dokumentu. Umístí se tak na jedno místo jednak výklad teorie a technologií, jednak se sjednotí dokumentace k jednotlivým projektům. Vývojáři tak získají rychle dostupné materiály.

7.2 Uložení hesel a certifikátů

V OPC UA Adapteru jsou přihlašovací údaje a certifikáty k OPC UA serverům uloženy v databázi, aniž by byly jakkoliv chráněny. V současné chvíli je toto řešení dostačující. Předpokládá se zajištění ochrany v rámci Docker kontejneru, která ovšem není vše garantující. Tuto tématiku mimo jiné rozebírá i kniha „Using Docker“[24].

7.3 Optimalizace datových typů v paměti

V OPC UA Adapteru je aktuálně implementována pouze optimalizace připojených klientů k OPC UA serverům. Optimalizace definic datových typů v paměti nebyla předmětem hlubší diskuze a implementace, jelikož produkční servery

nyní používají pouze jeden informačních model. Tedy současná množina definic datových typů nepřináší zátěž. Pokud by bylo třeba optimalizovat, lze se vydat cestou opakovaného načítání definic typů z databáze a jejich kešování. Typicky některé budou používány velmi často (základní datové typy nebo typy proměnných, jenž obsahují měřené veličiny) a některé méně (různé pomocné proměnné a metody pro konfiguraci nebo diagnostiku stavu samotného zařízení).

7.4 UI a zbylé komponenty systému

V této práci se diskutovalo UI pouze z hlediska možné architektury či technologií a ostatní systémové komponenty se popsaly stručně. V okamžiku tvorby diplomové práce se teprve sbírají požadavky na uživatelské rozhraní a případně vznikají první Lo-Fi prototypy (wiframes, mock up atp.). Bylo zde tedy cíleno na návrh backendu a implementaci komponenty pro OPC UA protokol. Dalším krokem vývoje je komponenta Data Store, diagnostické nástroje a UI.

Závěr

Diplomová práce si položila za cíl rozšířit systém sběru dat z OPC UA serverů do plnohodnotné části diagnostického řešení společnosti ModemTec. Soustředila se na úpravu návrhu systému dle nových požadavků a implementaci komponenty zajišťující vyčítání dat pomocí OPC UA protokolu. Bylo dosaženo všech stanovených dílčích cílů.

V prvním z nich byly popsány knihovny a frameworky pro jazyk Scala, na které vývojář při implementaci systému narazí. Byla popsána knihovna Cats, které implementuje prvky teorie kategorií. Rovněž posloužila i pro vysvětlení základních funkcionálních návrhových vzorů, např. monáda. Dále byl uveden projekt Akka, jenž nabízí škálu knihoven pro tvorbu paralelních a distribuovaných programů, streamování dat a gRPC, HTTP servery. To vše postavené na implementaci modelu aktorů. Třetí knihovnou byl Slick, který dodává funkcionálně relační mapování pro SQL a NoSQL databáze. A konečně se představil Play Framework užívaný pro vytváření webových serverů. Současně s Play bylo demonstrováno použití knihovny Guice používanou pro dependency injection.

Druhým dílčím cílem bylo popsání výchozího stavu systému a návrhu úprav. Při práci na novém návrhu byla nejdříve provedena analýza domény a procesů, následně se vyhotovilo rozdělení do komponent, které spolu budou komunikovat point-to-point a budou nasazeny přes Docker. K nim byly vybrány patřičné technologie a shrnuly se funkční požadavky na jejich API. Celkově systém podporuje různé IoT protokoly, ke kterým přistupuje Data Store, kde jsou sesbírána data, přes univerzální Adapter API.

Třetí dílčí cíl se zaměřoval na implementaci komponenty pro OPC UA. Při návrhu se vytvořilo Adapter API, které bylo detailně definováno a ke kterému byl vytvořen klient. Aplikace pro OPC UA byla přetvořena do restful serveru OPC UA Adapter, jenž implementuje Adapter API a poskytuje tak přes univerzální rozhraní obsluhu OPC UA serverů. Architektura je třívrstvá. Rovněž byly přidány nové funkcionality do knihovny Scalable OPC UA, kterou OPC UA Adapter využívá.

Posledním dílčím cílem, tedy čtvrtým, bylo sestavení nápadů na možné rozšíření/další vývoj a samotný závěr. Tato práce má mimo jiné i jistý dokumentační charakter, a tak jedním z dalších kroků je seskupení informací do jedné systémové dokumentace. Dalším rozvojem systému je implementace komponenty Data Store a tvorba uživatelského rozhraní.

Literatura

[1] MODEMTEC S.R.O: *ModemTec*. [online], 2024, [cit. 2024-01-30]. Dostupné z: `https://modemtec.cz/cs/`

[2] Codl, D.: *Systém pro sběr dat s využitím OPC UA*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

[3] WELSH, N.; GURNELL, D.: *Scala with Cats*. Brighton: Underscore Consulting LLP, druhé vydání, 2020.

[4] ÉCOLE POPLYTECHNIQUE FEDERALE LAUSSANE: *Scala 3 Reference*. [online], 2024, [cit. 2024-02-04]. Dostupné z: `https://docs.scala-lang.org/scala3/reference/`

[5] ÉCOLE POPLYTECHNIQUE FEDERALE LAUSSANE: *The Scala Programming Language*. [online], 2024, [cit. 2024-01-20]. Dostupné z: `https://www.scala-lang.org/`

[6] LIGHTBEND INC.: *Akka*. [online], 2024, [cit. 2024-01-28]. Dostupné z: `https://akka.io/`

[7] BONÉR, J.: *Why We Are Changing the License for Akka*. [online], 2022, [cit. 2024-02-01]. Dostupné z: `https://www.lightbend.com/blog/why-we-are-changing-the-license-for-akka`

[8] APACHE PEKKO: *Pekko*. [online], 2024, [cit. 2024-02-01]. Dostupné z: `https://pekko.apache.org/`

[9] LIGHTBEND INC.: *Akka Actors*. [online], 2024, [cit. 2024-02-01]. Dostupné z: `https://doc.akka.io/docs/akka/current/typed/index.html`

[10] LIGHTBEND INC.: *Akka Streams*. [online], 2024, [cit. 2024-02-01]. Dostupné z: https://doc.akka.io/docs/akka/current/stream/index.html

[11] DALLAWAY, R.; FERGUSON, J.: *Essential Slick*. Brighton: Underscore Consulting LLP, 2019.

[12] LIGHTBEND INC.: *Slick*. [online], 2024, [cit. 2024-02-15]. Dostupné z: https://scala-slick.org/

[13] PLAY FRAMEWORK: *Play Framework*. [online], 2024, [cit. 2024-02-08]. Dostupné z: https://www.playframework.com/

[14] BYARS, B.: *Enterprise Integration Using REST*. [online], 2013, [cit. 2024-01-08]. Dostupné z: https://martinfowler.com/articles/enterpriseREST.html

[15] CAM, J.: *Micro Frontends*. [online], 2019, [cit. 2024-01-15]. Dostupné z: https://martinfowler.com/articles/micro-frontends.html

[16] FOWLER, M.; LEWIS, J.: *Microservices*. [online], 2014, [cit. 2024-01-15]. Dostupné z: https://martinfowler.com/articles/microservices.html

[17] META OPEN SOURCE: *React*. [online], 2024, [cit. 2024-01-20]. Dostupné z: https://react.dev/

[18] FIT ČVUT: *Oborové předměty Softwarové inženýrství*. [online], 2024, [cit. 2024-01-21]. Dostupné z: https://fit.cvut.cz/cs/studium/programy-a-obory/bakalarske/8325-softwarove-inzenyrstvi/oborove-predmety

[19] MAHNKE, W.; LEITNER, S.-H.; DAMM, M.: *OPC Unified Architecture*. Berlin: Springer-Verlag Berlin Heidelberg, 2009, ISBN 978-3-540-68898-3.

[20] OPC UA FOUNDATION: *OPC UA Online Reference, IEC61850-7-3*. [online], 2024, [cit. 2024-01-29]. Dostupné z: https://reference.opcfoundation.org/IEC61850-7-3/

[21] KANTOR, I.: *WebSocket*. [online], 2022, [cit. 2024-01-10]. Dostupné z: https://javascript.info/websocket

[22] ARTIMA INC.: *ScalaTest*. [online], 2024, [cit. 2024-02-07]. Dostupné z: https://www.scalatest.org/

[23] PLAY FRAMEWORK: *Dockerize the App - Play Framework Tutorial*. [online], 2020, [cit. 2024-02-6]. Dostupné z: https://dvirf1.github.io/play-tutorial/posts/dockerize-the-app/

[24] MOUAT, A.: *Using Docker*. Sebastopol: O'Reilly Media, Inc., první vydání, 2015, ISBN 978-1-491-91576-9.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

BDD Behavior-Driven Development

CLI Command Line Interfac

CPU Central Processing Unit

CSV Comma-separated Values

DI Dependency Injection

GUID Globally Unique Identifier

HTTP Hypertext Transfer Protocol

IEC International Electrotechnical Commission

I/O Input/Output

IoT Internet of Things

ISO International Organization for Standardization

JPA Jakarta Persistence

JSON JavaScript Object Notation

Lo-Fi Low Fidelity

LTS Long-term Support

MVC Model-View-Controller

OLAP Online Analytical Processing

OLTP Online Transaction Processing

PKI Public Key Infrastructure

SOAP Simple Object Access Protocol

UI User Interface

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTC Coordinated Universal Time

XML Extensible Markup Language

Příloha B

Obsah přiloženého média

- `readme.txt` ..... popis obsahu přílohy
- `text` ..... písemná část práce
  - `DP_Codl_Dominik_2024.pdf` ..... text práce ve formátu PDF
  - `latex_zdroj` ..... zdrojové soubory písemné části práce
  - `obrazky_zdroj` ..... zdrojové soubory obrázků
- `system` ..... nepísemná část práce – Systém pro sběr dat
  - `adapter` ..... Adapter API a jeho implementace
    - `Adapter_API` ..... dokumentace API a implementace klienta
    - `OPC_UA_Adapter` ..... implementace Adapter API pro OPC UA
  - `libs` ..... knihovny pro komponenty
    - `Scalable_OPC_UA` ..... knihovna pro OPC UA