

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-52598

**AUTOMATIZOVANÉ GENEROVANIE
APLIKAČNÉHO
ROZHRANIA V PROCESNE ORIENTOVANÝCH
SYSTÉMOCH
DIPLOMOVÁ PRÁCA**

2019

Juraj Kubričan

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-52598

**AUTOMATIZOVANÉ GENEROVANIE
APLIKAČNÉHO
ROZHRANIA V PROCESNE ORIENTOVANÝCH
SYSTÉMOCH
DIPLOMOVÁ PRÁCA**

Študijný program:	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	prof. RNDr. Gabriel Juhás, PhD.
Konzultant:	Ing. Milan Mladoniczky

Bratislava 2019

Juraj Kubričan

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Juraj Kubričan
Diplomová práca:	Automatizované ge- nerovanie aplikačného rozhraniavproces- neorientovaných systémoch
Vedúci záverečnej práce:	prof. RNDr. Gabriel Juhás, PhD.
Konzultant:	Ing. Milan Mladoniczky
Miesto a rok predloženia práce:	Bratislava 2019

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean et est a dui semper facilisis. Pellentesque placerat elit a nunc. Nullam tortor odio, rutrum quis, egestas ut, posuere sed, felis. Vestibulum placerat feugiat nisl. Suspendisse lacinia, odio non feugiat vestibulum, sem erat blandit metus, ac nonummy magna odio pharetra felis. Vivamus vehicula velit non metus faucibus auctor. Nam sed augue. Donec orci. Cras eget diam et dolor dapibus sollicitudin. In lacinia, tellus vitae laoreet ultrices, lectus ligula dictum dui, eget condimentum velit dui vitae ante. Nulla nonummy augue nec pede. Pellentesque ut nulla. Donec at libero. Pellentesque at nisl ac nisi fermentum viverra. Praesent odio. Phasellus tincidunt diam ut ipsum. Donec eget est. A skúška mäččėňov a dlžnov.

Klíčové slová: klíčové slovo1, klíčové slovo2, klíčové slovo3

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Juraj Kubričan
Master's thesis:	Automatic generation of application interface in processor oriented systems
Supervisor:	prof. RNDr. Gabriel Juhás, PhD.
Consultant:	Ing. Milan Mladonický
Place and year of submission:	Bratislava 2019

On the other hand, we denounce with righteous indignation and dislike men who are so beguiled and demoralized by the charms of pleasure of the moment, so blinded by desire, that they cannot foresee the pain and trouble that are bound to ensue; and equal blame belongs to those who fail in their duty through weakness of will, which is the same as saying through shrinking from toil and pain. These cases are perfectly simple and easy to distinguish. In a free hour, when our power of choice is untrammelled and when nothing prevents our being able to do what we like best, every pleasure is to be welcomed and every pain avoided. But in certain circumstances and owing to the claims of duty or the obligations of business it will frequently occur that pleasures have to be repudiated and annoyances accepted. The wise man therefore always holds in these matters to this principle of selection: he rejects pleasures to secure other greater pleasures, or else he endures pains to avoid worse pains.

Keywords: keyword1, keyword2, keyword3

Pod'akovanie

I would like to express a gratitude to my thesis supervisor.

Obsah

Úvod	1
1 Analýza problému	2
1.1 Petriho siete	2
1.2 Petriflow	2
1.3 Aplikačné rozhranie	2
2 Špecifikácia	4
2.1 Funkcionálne požiadavky	4
2.2 Nefunkcionálne požiadavky	4
3 Návrh	6
3.1 Prípady použitia	6
3.1.1 Registrácia modelu	6
3.1.2 Získanie informácie o prechode	6
3.1.3 Spustenie prechodu	6
3.1.4 Autentifikácia	6
3.2 Architektúra	6
3.3 Bezpečnosť	7
4 Implementácia	9
4.1 Použité technológie	9
4.1.1 Kotlin	9
4.1.2 Gradle	10
4.1.3 Spring boot	10
4.1.4 Spring cloud	10
4.2 Relay Service	11
4.3 Generator service	11
4.3.1 Prijatie požiadavky	12
4.3.2 Čítanie súborov Petriflow a súborov s používateľmi	13
4.3.3 Registrácia používateľov	13
4.3.4 Príprava repozitára	13
4.3.5 Generovanie kódu	14
4.3.6 Kompilácia a spustenie relay	15
4.4 Auth Service	17

4.5 Gateway Service	17
5 Testovanie	18
Záver	19
Zoznam použitej literatúry	I
Prílohy	I
A XSD súbor súboru s používateľmi	II
B Algoritmus	III
C Výpis sublime	IV

Zoznam obrázkov a tabuliek

Obrázok 1	Úloha	3
Obrázok 2	Architektúra rozhrania	7
Obrázok 3	Autorizácia	18

Zoznam algoritmov

B.1	Vypočítaj $y = x^n$	III
-----	---------------------	-----

Zoznam výpisov

1	Ukážka funkcie typov v jazyku Kotlin	9
2	Príklad súboru s používateľmi	12
3	Príklad vygenerovanej funkcie	14
4	Príklad vygenerovanej funkcie	15
5	Príklad vygenerovanej funkcie	16
C.1	Ukážka sublime-project	IV

Úvod

Tu bude krásny úvod s diakritikou atď.

A možno aj viac riadkový úvod.

1 Analýza problému

1.1 Petriho siete

Carl Adam Petri založil koncept Petriho sietí v roku 1962 vo svojej dizertačnej práci - Komunikácia s Automatmi - na Technickej Univerzite v Darmstadte. Ďalším výskumom sa z pôvodného konceptu ktorý bol určený na modelovanie analýzu komunikačných systémov vyvinul nástroj, ktorý sa používa naprieč mnohými oblasťami najmä na modelovanie paralelných a distribuovaných systémov.

Základné Petriho siete pozostávajú z prechodov, miest a hrán.

1.2 Petriflow

[petriflow_clanok]

Formalizmus Petriflow je rozšírenie Petriho sietí, ktoré bolo navrhnuté na modelovanie komplexných biznis procesov. Je vyvíjaný spoločnosťou Netgrif na základe dlhodobých skúseností s klientami, ktorý pomocou tohto formalizmu modelujú svoje biznis procesy, ktoré sú následne zavedené do používania.

Formalizmus Petriflow rozširuje Petriho siete o ďalšie komponenty. Ako základ berie Petriho siete obohatené reset, inihibitor a read hrany. Aby sa dali modelovať moderné biznis procesy pridáva Petriflow do tohto modelu roly, dátové polia a akcie.

Roly definujú kto je oprávnený spúšťať rôzne prechody.

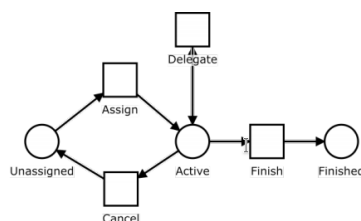
Dátové polia definujú štruktúru dát ktoré každá inštancia procesu obsahuje počas svojho behu.

Akcie definujú vzťahy a interakcie medzi jednotlivými dátovými poľami a prechodmi.

V klasických Petriho sieťach je spustenie prechodu vždy atomická operácia. Petriflow obsahuje 2 typy prechodov udalostné prechody, ktoré sú rovnako ako v klasických PN sieťach atomické, avšak vždy ich spúšťa nejaká osoba(používateľ) v systéme. Druhý typ prechodu je úloha. Úlohu môžeme vnímať ako podsieť na obrázku 1. Na začiatku je úloha nepridelená, ako prvý krok je potrebné ju niekomu (aj sebe) prideliť. Následne môže toto pridelenie zrušiť, prideliť inej osobe, alebo úlohu dokončiť

1.3 Aplikačné rozhranie

Majme procesne orientovaný systém ktorý implementuje procesy, popísane v Petriflow. S takýmto systémom používatelia interagujú iba presne popísaným spôsobom a to spúšťaním prechodov ktoré majú podľa svojich rolí oprávnenie spúšťať (delegovať, dokončiť,...). Pri dokončení úlohy (prechodu) musí používateľ poskytnúť dáta popísané v



Obr. 1: Úloha

dátových poliach.

Keďže formalizmus Petriflow je schopný takýmto spôsobom popísať všetky interakcie používateľa so systémom je možné na jeho základe vygenerovať aj aplikačné rozhranie. Toto rozhranie sprístupní systém mimo jeho domény, zaručí autorizáciu podľa rolí, poskytne dokumentáciu o svojej štruktúre (a tým pádom aj štruktúre procesu) a zabezpečí validáciu dát, ktoré do systému používateľ odošle.

Ak by rozhranie držalo informáciu o značkovaní, a teda spustiteľnosti prechodov, mohol by nasť stav kedy aplikačné rozhranie je v stave, ktorý naznačuje, že prechod sa nedá spustiť no procesný server už je v stave, kedy je prechod spustiteľný. Kvôli udržaniu konzistencie dát a predídeniu race conditions nemôže aplikačné rozhranie držať informáciu o značkovaní a teda ani spustiteľnosti prechodu.

Aktuálna verzia Petriflow poskytuje relatívne granulárne informácie o autorizácii na vykonávanie akcií pomocou rolí, neobsahuje však informáciu o používateľoch ani o tom ktorý používateľ má pridelené aké roly. Na implementáciu funkčného aplikačného rozhrania teda bude nutné dorobiť systém ktorý túto informáciu bude obsahovať.

2 Špecifikácia

V tejto kapitole najprv stručne opíšeme hlavnú funkcionálnu navrhovanej aplikácie, potom zadefinujeme funkcionálne a nefunkcionálne požiadavky na aplikáciu.

Softvér, ktorý sme sa rozhodli implementovať bude slúžiť ako rozhranie medzi procesným serverom a internetom. Bude umožňovať klientovi pripojiť sa na procesný server cez internet, autentifikovať sa, získať informácie o dátach v prechodoch petriho siete a bude umožňovať modifikovať stav siete (procesu) spúšťaním prechodov.

2.1 Funkcionálne požiadavky

1. Rozhranie bude umožňovať administrátorovi registrovať používateľov a priradovať im roly
2. Umožní administrátorovi za behu pridať nové koncové body a meniť, alebo vymazať aktuálne nasadené koncové body
3. Umožní prihlásenie používateľa pomocou štandardného autentifikačného protokolu.
4. Autentifikovaným používateľom umožní prístup k dátam z tých prechodov ktoré majú právo čítať podľa ich roly.
5. Autentifikovaným používateľom umožní spúšťať prechody ktoré majú právo spúšťať podľa ich roly.
6. Pri spúšťaní prechodu prebehne validácia vstupných dát. V prípade nevalidných alebo nekompletných dát nepovolí spustenie prechodu.
7. Rozhranie poskytne online dokumentáciu prechodov v sieti, táto dokumentácia bude zahŕňať URL prechodu, potrebné dátové polia na spustenie prechodu a roly, ktoré sú oprávnené prechody spúšťať.
8. Rozhranie poskytne aplikačné rozhranie viacerým sieťam s rôznou štruktúrou. A viacerím inštanciam týchto sietí.

2.2 Nefunkcionálne požiadavky

1. Rozhranie bude škálovateľné
2. Rozhranie bude zabezpečené štandardnými bezpečnostnými prvkami

3. Rozhranie bude bežať na serveri s operačným systémom Ubuntu 18.04.2 LTS Bionic Beaver poskytnutom Fakultou Elektrotechniky a Informatiky na Slovenskej Technickej Univerzite.

3 Návrh

3.1 Prípady použitia

Zo špecifikácie vyplývajú nasledovné interakcie používateľa s naším systémom.

3.1.1 Registrácia modelu

Prvý prípad použitia je registrácia modelu. Aktér ktorý môže registrovať modely je len administrátor. Pri tejto akcii administrátor poskytne nášmu systému informácie o štruktúre procesu vo formáte petriflow, zoznam používateľov im prislúchajúcich rolí v XML a unikátny identifikátor siete.

Pokiaľ chce administrátor upraviť nasadený proces spustí proces registrácie nanovo s upravenými údajmi o sieti, používateľoch a rolách.

Administrátor taktiež môže sieť vymazať.

3.1.2 Získanie informácie o prechode

Keď už sú sieť aj používatelia úspešne zaregistrovaný, si môžu používatelia, vyžiadať ?? informácie o prechode. Tieto informácie budú poskytnuté len používateľovi s rolou oprávnenou na čítanie dát z daného prechodu.

3.1.3 Spustenie prechodu

Používatelia s príslušnými rolami môžu taktiež spúšťať ?? prechody. Pri spustení prechodu poskytne používateľ dátové polia potrebné na spustenie prechodu.

3.1.4 Autentifikácia

Z požiadavky na bezpečnosť aplikácie vyplýva ešte prípad použitia, kedy sa používateľ autentifikuje, aby nadobudol identitu rozpoznanú našim systémom a boli mu pridelené roly.

3.2 Architektúra

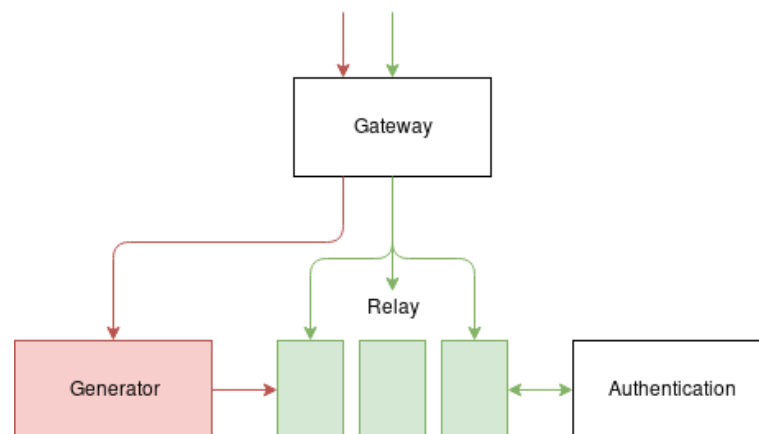
Aby sme splnili požiadavku na jednoduché škálovanie aplikácie a pre prehľadnenie architektúry zvolili sme si architektúru mikroservisov. Táto architektúra pozostáva z viacerých oddelených častí, každá z týchto častí má svoju jasne definovanú funkciu. Takéto mikroservisy sú jednoducho testovateľné, dajú sa nasadzovať postupne a nezávisle od seba a softvér navrhnutý v tejto architektúre býva spravidla robustný a vysoko škálovateľný.

Softvér sa bude skladať z troch hlavných služieb: generátor, relay bod a autentifikačná služba.

- Generator je služba, ktorá bude registrovať a zostavovať koncové body. Je to služba, na ktorú sa pripojí adiministátor, zadá štruktúru siete a zoznam používateľov a rolí.

Táto služba následne zaregistruje používateľov a ich roly, vygeneruje kód pre službu relay a túto službu spustí.

- Relay je služba, ktorá obsahuje vygenerovaný kód koncových bodov. Táto služba bude poskytovať dokumentáciu dostupných koncových bodov. bude poskytovať samotné koncové body a pri zavoľaní koncového bodu sa bude starať o validáciu prijatých dát.
- Autentifikačná služba sa stará o autentifikáciu používateľov a pridelenie rolí používateľom
- Gateway služba poskytuje rozhranie medzi internetom a našou doménou. Vonkajšiemu svetu prezentuje len služby ktoré majú byť dostupné z internetu. Zároveň poskytuje funkcionality load balancera.



Obr. 2: Architektúra rozhrania

3.3 Bezpečnosť

Autentifikáciu v našej aplikácii budeme riešiť pomocou štandardnej knižnice, ktorá poskytuje OAuth2.0. Táto služba bude bežať v samostatnom kontajnere a prístup do nej bude len pomocou protokolu OAuth2.0 na overovanie klientov a prostredníctvom interného volania na registráciu používateľov.

Ochrana pred neoprávneným prístupom bude ďalej realizovaná pomocou služby gateway. Táto služba bude jediný bod ako sa dá prístupíť do našej domény z internetu. Služba bude povoľovať len dotazy ktoré špecifikujeme. Ochrana proti neoprávnenému prístupu bude taktiež realizovaná na vrstve operačného systému pomocou firewallu.

Ochrana proti DOS útokom bude taktiež realizovaná pomocou gateway služby. Táto služba bude slúžiť ako load balancer, a rovnomerne rozdeľovať dotazy medzi inštalácie relay servisu tak, aby boli rovnomerne vyťažené. Služba Gateway bude taktiež poskytovať ochranu proti priveľkému počtu dotazov od jedného klienta(rate limiting).

4 Implementácia

4.1 Použité technológie

4.1.1 Kotlin

Kotlin je relatívne nový programovací jazyk, projekt Kotlin bol po prvý krát zverejnený v roku 2011 spoločnosťou JetBrains(Andrey Breslav). Bol vyvinutý ako moderný staticky typovaný jazyk, ktorý podporuje rýchlu kompiláciu do javy. V roku 2017 vyhlásil Google podporu pre Kotlin v operačnom systéme Android.

Medzi jeho hlavné výhody patrí menší boilerplate (menej zbytočného kódu), a vylepšený systém typovania premenných. V Kotlinu môžu byť premenné nulovateľné, to znamená, že kompilátor počíta s prípadom, že nulovateľná premenná ešte nie je inicializovaná. Kompilátor Kotlinu využíva pokročilú logiku na to, aby zistil v ktorých vetvách kódu premenné môžu obsahovať null a v ktorých nie. V prípade že podmienkou ošetríme nulový prípad kompilátor v druhej vetve programu zmení typ premennej tak, že už nie je nulovateľná 1. Teda v kóde(pokiaľ aktívne neobídeme typovú ochranu) nemôže nastať null pointer exception.

```
// returns null or String
fun maybeGetString():String?

// type: String?
val variable:String? = maybeGetString()

// compile time error
variable.length

if(variable == null){
    variable = ""
}

// type: String
// no compile time error
variable.length
```

Listing 1: Ukážka funkcie typov v jazyku Kotlin

Kotlin je jazyk ktorý sa dá buď transpilovať do jazyka java, alebo JavaScript, alebo sa dá kompilovať priamo do spustiteľného binárneho súboru

pre všetky bežné Operačné systémy (Linux, Windows, ANdroid, iOS). My budeme používať Kotlin kompilovaný do jazyka Java lebo nám to dovoľuje využívať výhody jazyka a zároveň využívať všetky knižnice, ktoré sú dostupné pre jazyk Java.

4.1.2 Gradle

Gradle je voľne šíriteľný nástroj na automatizáciu zostavovania softvéru. Je stavaný na to aby bol schopný zostaviť takmer ľubovoľný program. Podporuje jazyky ako java, C++ Python, a mnoho ďalších. V našom projekte sa gradle použijeme na manažment závislostí, kompiláciu kódu a spustenie samotného skompilovaného programu. Konfigurácia nástroja prebieha pomocou konfiguračného súboru napísaného v jazyku Groovy, tieto konfiguračné súbory sa v našom prípade použitia ukázali ako veľmi prehľadné a ľahké na použitie. Gradle taktiež používa pokročilú techniku memoizácie procesu zostavovania softvéru takže jeho výkon je pri opakovanej kompilácii vyšší.

4.1.3 Spring boot

Spring Boot je voľne šíriteľný framework založený na jazyku Java. Je vyvíjaný a udržiavaný tímom Pivotal. Je určený na vytváranie nezávislých, produkčných aplikácií a mikroservisov. Spring boot je robustná platforma so širokou podporou pre všetky štandardné operácie ktoré budeme potrebovať pri vývoji webového rozhrania. Poskytuje podporu pre vytváranie štandardných RESTful koncových bodov, ďalej poskytuje podporu pre štandardnú autentifikáciu pomocou OAuth2.0 a integráciu s OpenAPI 3.0 [**openapi3**] pomocou balíčka Swagger [**swagger**]

Všetky Spring Boot kontajnery sme inštalovali pomocou spring initializr [**initializr**] tento nástroj vygeneruje zip súbor so založeným projektom vo frameworku Spring Boot. Pri vytváraní projektu je možné si vybrať Jazyk v ktorom bude projekt založený a nástroj ktorý bude projekt zostavovať

4.1.4 Spring cloud

Spring Cloud je framework, ktorý obsahuje bohatú sadu nástrojov na vytváranie mikroservisov a cloudových riešení. Medzi nástroje Spring Cloudu patí:

- Cloud config - nástroj na distribúciu konfiguračných súborov medzi kontajnermi mikroservisov
- Service discovery - nástroj na registráciu a monitorovanie mikroservisov
- Gateway - Nástroj na routovanie a load balancing v rámci mikroservisov
- Cloud Authentication - Nástroj na riešenie komplexnej autentizácie a autorizácie v rámci

4.2 Relay Service

Vygenerovať túto službu je cieľom našej práce. Po vygenerovaní koncových bodov pre dané procesné modely bude táto služba prijímať požiadavky od klientov, autorizované požiadavky od klientov validovať a následne preposielať na procesný server. Okrem toho bude poskytovať dokumentáciu k daným koncovým bodom.

Pred tým ako vygenerujeme koncové body špecifické pre daný procesný model, je relay service iba prázdny Spring Boot kontajner so základnou konfiguráciou. Pre poskytnutie základnej funkcionality webového koncového bodu obsahuje kontajner balíček Spring Web Starter [**webstarter**], tento balíček poskytuje funkcionality, ktorá umožňuje pomocou anotácií mapovať funkcie tried na rôzne URL, špecifikovať, HTTP metódu, dátové polia vstupu a formát výstupu. Tieto mapovania následne oživí pomocou priloženého webového servera Apache Tomcat [**tomcat**]

Na poskytnutie dokumentácie vo formáte OpenAPI 3.0 [**openapi3**] využívame balík Swagger. Tento balík dokáže čítať anotácie balíčka Spring Web Starter, a na ich základe vygeneruje základnú dokumentáciu vo formáte JSON, ktorú následne vyprezentuje klientom. Okrem základnej dokumentácie, ktorá popisuje prístupné adresy a HTTP metódy ktorými sú dostupné obsahuje balíček aj dodatočné anotácie pomocou ktorých vieme konkrétnejšie opísať danú volanú funkciu. Tieto anotácie si bližšie priblížime v kapitole

Na poskytnutie základnej validácie vstupov použijeme regulárne výrazy a formáty vstupov

4.3 Generator service

Tento servis je základný satavebný kameň nášho rozhrania, jeho úlohou je na základe vstupu v podobe procesného modelu vo formáte Petriflow vygenerovať a spustiť inštancie relay service, ktorý bude prijímať samotné požiadavky od klienta. Základná operácia vygenerovania koncového bodu má nasledovné kroky:

- prijatie požiadavky,
- čítanie súborov Petriflow a súborov s používateľmi,
- registrácia používateľov,
- príprava repozitára,
- generovanie kódu
- a kompilácia a spustenie relay.

```

<?xml version="1.0" encoding="UTF-8"?>
<document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="./users_schema.xsd">
  <user name="admin" password="SecureAdminPass">
    <role id="client"/>
    <role id="bureau_agent"/>
    <role id="loan_officer"/>
    <role id="underwriter"/>
    <role id="property_appraiser"/>
    <role id="account_clerk"/>
  </user>
  <user name="user1" password="SecureUser1Pass">
    <role id="bureau_agent"/>
    <role id="underwriter"/>
    <role id="account_clerk"/>
  </user>
  <user name="user2" password="SecureUser2Pass">
    <role id="client"/>
    <role id="loan_officer"/>
    <role id="property_appraiser"/>
  </user>
</document>

```

Listing 2: Príklad súboru s používateľmi

4.3.1 Prijatie požiadavky

Pre funkciu pridania požiadavky je v našom systéme vyhradená základná URL `"/[POST]"`, na tejto URL je funkcia, ktorá berie ako parametre súbor s procesným modelom, súbor s používateľmi a unikátny identifikátor siete. Súbor s modelom je v štandardnom formáte Petriflow. Súbor s používateľmi je vo formáte XML so štruktúrou popísanou XSD súborom v prílohe A. Tento XSD súbor je poskytnutý aj v rámci dokumentácie v OpenAPI 3.0. Na príklade súboru v ukážke kódu 2 môžeme vidieť že súbor obsahuje zoznam používateľov s menom, heslom a zoznamom ich rolí.

Funkciu registrácie môže spustiť iba administrátor systému (Viac v kapitole Authentication)

Tieto súbory sa uložia do priečinka na servery s názvom `Net[ID siete]` a `Users[ID siete]`. V prípade, že už existujú súbory s rovnakým ID tieto súbory sa prepíšu a tým zaručujeme funkcionality zmeny siete.

Ukladanie súboru s heslami v textovej forme je v produkčnej aplikácii neprípustné, preto pred uložením vytvoríme hash hesla štandardným spôsobom kompatibilným so Spring Security.

Okrem požiadavky na pridanie siete môže administrátor spustiť aj požiadavku na vymazanie siete. V takomto prípade sa vymažú súbory s ID danej siete a ďalšie kroky postupujú rovnako ako pri registrácii.

4.3.2 Čítanie súborov Petriflow a súborov s používateľmi

Na čítanie XML súborov používame štandardnú knižnicu JAXB [**jaxb**]. Táto knižnica podľa XSD súborov vygeneruje triedy, so rovnakou štruktúrou ako majú entity v XML.

Vygenerovali sme teda dva balíky, jeden balík s triedami v jazyku java z definície Petriflow a jeden s triedami z definície nášho súboru s používateľmi.

Následne pri čítaní súborov použijeme triedu Unmarshaller balíka JAXB, ktorý prečíta textový súbor XML a vráti java objekty s dátami zo súborov. Keďže jazyk kotlin, v ktorom pracujeme je plne kompatibilný s jazykom java, s týmito objektami môžeme ďalej pracovať ako so štandardnými kotlin dátovými objektami.

4.3.3 Registrácia používateľov

Keď už máme dáta s používateľmi a modelom, zaregistrujeme používateľov do našej autentifikačnej služby.

Registrácia používateľov sa realizuje pomocou HTTP volania na Authentication service. Servisu poskytneme prihlasovacie meno a heslo používateľa. Aby sme predišli konfliktom v menách používateľov medzi jednotlivými sieťami, pred prihlasovacie mená pridáme ID siete a tým pádom sú všetci používatelia jednej siete v samostatnom menovom priestore.

4.3.4 Príprava repozitára

Na spustenie Spring boot kontajnera treba okrem kódu so samotnými koncovými bodmi aj základnú konfiguráciu. Pre zjednodušenie procesu generovania kódu stiahneme predpripravenú konfiguráciu zo separátneho git repozitára [**dp_relay**]. V tomto repozitári je predpripravený Spring Boot projekt s podporou pre Kotlin do ktorého budeme vkladať naše súbory s vygenerovanými triedami. Na stiahnutie projektu používame knižnicu JGit [**jgit**] čo je implementácia git klienta v jazyku java.

Funkcia na prípravu repozitára najprv skontroluje či v dočasnom priečinku už repozitár neexistuje, ak existuje zavolá jgit ekvivalent funkcie "git reset origin/master" čím zmení lokálne súbory na aktuálnu verziu z repozitára. Súbory obsiahnuté v .gitignore sa ponechajú bez zmeny, a tým pádom akékoľvek cache súbory z predošlej kompilácie ostatnú nedotknuté čo urýchli kompiláciu kódu. Ak git reset z nejakého dôvodu zlyhá, funkcia vymaže akékoľvek pozostatky súborov v dočasnom priečinku a repozitár znovu naklonuje.


```

@PostMapping("1/{instanceId}/assign")
@RolesAllowed("ROLE_EXAMPLE_CLIENT")
@ApiOperation(
    value = "Transition1",
    notes = "Allowed roles: [ROLE_EXAMPLE_CLIENT]" )
fun assign1(@PathVariable("instanceId") instanceId: String): ResponseEntity<String> {
    processServerRequest.assign("Example", "1", instanceId)
    return ResponseEntity("", OK)
}

```

Listing 3: Príklad vygenerovanej funkcie

4.3.5 Generovanie kódu

Keď už je projekt stiahnutý a pripravený môžeme začať s generovaním kódu koncových bodov. Na generovanie kódu využívame knižnicu kotlipoet [kotlipoet]. Táto knižnica nám poskytuje nástroje na generovanie kódu v jazyku Kotlin. Tieto nástroje pozostávajú z builder funkcií, pomocou ktorých vieme vyskladať komplexný, syntakticky správny kód.

Najprv pre každý procesný model vygenerujeme vlastnú triedu. Túto triedu vygenerujeme s nasledovnými anotáciami:

- @Controller Zabezpečuje spustenie funkcionality HTTP controllera
- @EnableResourceServer Zabezpečuje funkcionality autentifikácie pre danú triedu
- @RequestMapping Pridá predponu s ID siete pred URL koncových bodov danej triedy

Následne vygenerujeme samotné funkcie triedy. Pre každý prechod v procesnom modeli vygenerujeme šesť funkcií. každá z nich reprezentuje jednu operáciu, ktorú môžeme s prechodom(úlohou) podľa Petriflow robiť.

Jednoduché funkcie sú assign, cancel a finish tieto funkcie sú na svojich URL podľa návrhu, nemajú žiadne parametre a pokiaľ nenastane chyba vracajú prázdnu odpoveď. Do tela funkcie pridáme príslušné volanie na procesný server. Na ukážke kódu 3 vidíme vygenerovaný kód pre funkciu assign. Kód funkcií cancel a finish je analogický

Funkcia View, navyše vracia objekt s dátami ktoré sa na danom prechode v danej inštancii nachádzajú. na ukážke kódu 4 vidíme ukážku vygenerovaného koncového bodu pre View

```

@GetMapping("/{instanceId}/view")
@RolesAllowed("ROLE_EXAMPLE_CLIENT")
@ApiOperation(
    value = "Transition1",
    notes = "Allowed roles: [ROLE_EXAMPLE_CLIENT]" )
fun view1(@PathVariable("instanceId") instanceId: String): get1Result {
    return processServerRequest.get("Example", "1", instanceId)
}

```

Listing 4: Príklad vygenerovanej funkcie

Zložitejšia funkcia je dáta. Táto funkcia používateľovi umožňuje poslať dáta pre dátové polia prechodov.

Tejto funkcii vygenerujeme toľko vstupných argumentov koľko dátových polí sa v prechode nachádza. Argumenty, okrem vstupných súborov, vygenerujeme typu String a následne do tela funkcie vygenerujeme validačný kód, ktorý pomocou regulárnych výrazov validuje tieto reťazce. Pre dátový typy enum a multichoice generujeme regulárny výraz podľa možností z definície poľa v Petriflow. Pre polia s typom date používame regulárny výraz pre dátum podľa štandardu ISO 8601. v prípade, že jedno alebo viac polí nemá správny formát alebo sa nebolo poskytnuté a je povinné vrátime používateľovi HTTP chybu 400 s výpisom v ktorých poliach je chyba.

Pri vstupných súboroch a poliach s typom string sa kontroluje iba ich prítomnosť, ak sú povinné.

Na ukážke kódu 5 vidíme príklad vygenerovanej funkcie.

Po vygenerovaní kódu uložíme súbory tried do priečinka v našom dočasnom priečinku v ktorom je inicializovaná Spring Boot aplikácia.

4.3.6 Kompilácia a spustenie relay

Po uložení vygenerovaných tried do správneho priečinka môžeme spustiť príkaz "gradle build". Tento proces skontroluje syntax a typy v našom vygenerovanom Kotlin kóde, následne transpiluje Kotlin kód do javy a tento kód skompiluje. Ak prebehla kompilácia bez chyby vypneme staré procesy relay servisu a spúšťame príkaz "gradle bootRun" ktorý spustí skompilovaný relay servis. Tento príkaz spustíme viac krát aby sme vytvorili viac inštancií relay servisu. Pokiaľ počas kompilácie nastala chyba vrátime výpis tejto chyby.

```

@PostMapping("35/{instanceId}/data")
@RolesAllowed("ROLE_EXAMPLE_CLIENT")
@ApiOperation(
    value = "Transition35",
    notes = "Allowed roles: [ROLE_EXAMPLE_CLIENT]"
)
fun data35(
    @PathVariable("instanceId") instanceId: String,
    @RequestParam(value = "floor") @ApiParam(required = true) floor: String,
    @RequestParam(value = "type") @ApiParam(required = true, allowableValues = "[House, Flat, Bungalow, Cabin]") type: String,
    @RequestParam(value = "years", defaultValue = "") @ApiParam(required = false) years: String,
    @RequestParam(value = "photo", defaultValue = "") @ApiParam(required = false) photo: MultipartFile ):
    ResponseEntity<String> {
        val errors = mutableListOf<String>()
        if (!Regex("\\d+").matches(floor)) {
            errors.add("floor should match \\d+")
        }
        if (!Regex("House|Flat|Bungalow|Cabin").matches(type)) {
            errors.add("type should match House|Flat|Bungalow|Cabin")
        }
        if (years != "" && !Regex("\\d+").matches(years)) {
            errors.add("years should match \\d+")
        }
        if (errors.isNotEmpty()) {
            return ResponseEntity(errors.toString(), BAD_REQUEST)
        }
        processServerRequest.data("Example", "35", instanceId, mapOf("floor" to floor, "type" to type,
            "years" to years, "photo" to photo ))

        return ResponseEntity("", OK)
    }
}

```

Listing 5: Príklad vygenerovanej funkcie

4.4 Auth Service

Na implementáciu autorizačného servera, ktorý dokáže spolupracovať so zvyškom nášho systému sme použili balíček Spring Cloud Security [`cloud_security`]. Tento balíček podobne ako Spring Boot Security poskytuje základnú funkcionality umožňujúcu sa prihlásiť pomocou mena, hesla, alebo aj OAuth2.0 a uskladnenie profilov používateľov. Na rozdiel od základného balíčka Spring Boot Security poskytuje funkcionality ktorá umožňuje v centralizovať autentifikáciu do jedného kontajnera.

Autentifikáciu má na starosti len autorizačný server. Ak sa chce používateľ autentifikovať jeho dopyt je presmerovaný na autorizačný server a tento mu v prípade poskytnutia správnych prihlasovacích údajov udelí autorizáciu(v prípade OAuth2.0 je to autorizačný token).

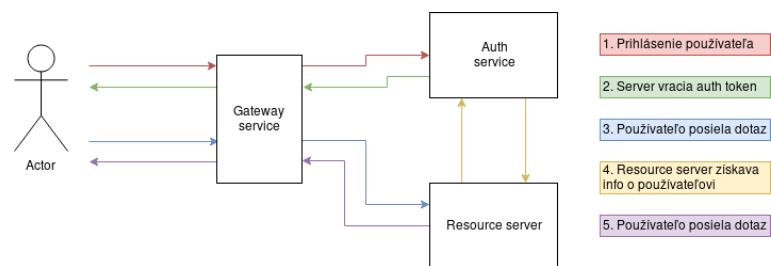
Resource server, nedrží informácie o používateľoch, ak však príde dopyt na funkciu, ktorá si vyžaduje autorizáciu, vie tento server získať profil používateľa od autorizačného servera. Ak nie je tento používateľ autorizovaný na vykonanie odpytu vracia resource server chybu 403.

Na spustenie autorizačného servera stačí importovať balíky Spring Cloud Security a Spring Cloud OAuth, napísať základnú konfiguráciu a autorizačný server sa dá spustiť.

Na integráciu so zvyškom nášho systému sme importovali tieto balíky aj do ostatných servisov a v prípade, kedy si funkcionality vyžiadala autorizáciu pridali sme potrebné anotácie. Aby sme dosiahli granulórnú kontrolu nad prístupom k funkcionalite, ku každej funkcií sme špecifikovali aké roly k nim majú prístup pomocou príslušnej anotácie.

Pre jednoduchosť implementácie sme na ukladanie profilov používateľov a tokenov využili in memory store. To znamená, že sa pri každom reštarte auth servisu tieto informácie zmažú. Keďže však používame architektúru mikroservisov, a servery sú od seba do veľkej miery nezávislé, nie je potrebné reštartovať všetky servery pri zmene jedného. Pri vývoji teda fakt, že používatelia boli uložený vo volatilnej pamäti nebol priveľkou prekážkou. Pre nasadenie systému do produkcie však bude nutné pridať do auth servisu perzistentné úložisko používateľov.

4.5 Gateway Service



Obr. 3: Autorizácia

5 Testovanie

Na testovanie sme použili softvér Insomnia REST Client[**insomnia**]. Tento program je určený na testovanie REST a GraphQL služieb. Je to voľne šíriteľná alternatíva známeho programu Postman, postavená na platforme Electron s použitím knižnice React. Insomnia nám dovolí vytvoriť a uložiť viacero testovacích dopytov, ktoré môžeme neskôr spustiť a overiť ich správne fungovanie. Insomnia taktiež podporuje autentifikačný protokol OAuth2, takže nám stačí iba zadať prístupové údaje a autorizačný token si stiahne sama. Taktiež v prípade vypršania autorizačného tokenu ho automaticky obnoví.

Pomocou tohto softvéru sme testovali funkcionality všetkých servisov.

Záver

Táto architektúra však nie je vhodná na menšie projekty, lebo réžia vzniknutého softvéru býva spravidla vysoká, lebo si vyžaduje viacero spustených inštancií servisov. Tiež nie je vhodná v prípadoch, kde sa čakávajú väčšie zmeny biznis logiky aplikácie. Pri väčšej smene biznis logiky je často nutné prerábať viacero servisov a zmena protokolu, ktorým medzi sebou komunikujú.

Prílohy

A	XSD súbor súboru s používateľmi	II
B	Algoritmus	III
C	Výpis subline	IV

A XSD súbor súboru s používateľmi

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="xs3p.xsl"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  version="1.1.0">
  <!-- ===== DOCUMENT ELEMENTS ===== -->
  <xs:element name="user">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="role" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute type="xs:string" name="name" use="required"/>
      <xs:attribute type="xs:string" name="password" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="role">
    <xs:complexType>
      <xs:attribute type="xs:string" name="id" use="required"/>
    </xs:complexType>
  </xs:element>
  <!-- ===== DOCUMENT ===== -->
  <xs:element name="document">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="user" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```


B Algoritmus

Algorithm B.1 Vypočítaj $y = x^n$

Require: $n \geq 0 \vee x \neq 0$

Ensure: $y = x^n$

$y \leftarrow 1$

if $n < 0$ **then**

$X \leftarrow 1/x$

$N \leftarrow -n$

else

$X \leftarrow x$

$N \leftarrow n$

end if

while $N \neq 0$ **do**

if N is even **then**

$X \leftarrow X \times X$

$N \leftarrow N/2$

else $\{N$ is odd $\}$

$y \leftarrow y \times X$

$N \leftarrow N - 1$

end if

end while

C Výpis sublime

```
../.. / fei .sublime-project
```

Listing C.1: Ukážka sublime-project