VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA WORKFLOW NA PLATFORMĚ JAVASCRIPT

WORKFLOW SUPPORT ON THE JAVASCRIPT PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MIROSLAV VÁLKA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Válka Miroslav**
Program: Informační technologie
Název: **Podpora workflow na platformě JavaScript**
Workflow Support on the JavaScript Platform
Kategorie: Informační systémy

Zadání:

1. Seznamte se s koncepty workflow, standardem BPMN a způsobem jeho XML reprezentace.
2. Prostudujte technologii GraphQL a způsoby jejího využití.
3. Po konzultaci s vedoucím navrhněte architekturu jednoduchého workflow systému pro zpracování a řízení podnikových procesů.
4. Implementujte navržený systém na platformě Node.js s využitím GraphQL pro realizaci aplikačního rozhraní.
5. Ověřte funkčnost řešení na jednoduché demonstrační aplikaci.
6. Zhodnoťte vytvořené řešení a jeho limity. Zhodnoťte nedostatky a výhody GraphQL API.

Literatura:

- Swicegood, T.: Programming Node.js, O´Reilly, 2012
- Žára, O.: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015
- Shapiro, R.M. et al.: BPMN 2.0 Handbook, Future Strategies Inc, 2010,http://www.conradbock.org/white-bpmn2-process-bookmark-web.pdf

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Burget Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 21. října 2019

Abstrakt

Cílem této bakalářské práce je navrhnout a vytvořit jednoduchý workflow systém pro zpracování a řízení podnikových procesů. Workflow systém vychází z jedné části specifikace BPMN 2.0, která je zaměřena na model podnikových procesů. Konkrétně je zaměřena na soukromé podnikové procesy. Implementovaný workflow systém je určen k provozu na platformě Node.js a jeho aplikační rozhraní je realizováno pomocí GraphQL. Pro implementaci byl zvolen jazyk TypeScript. Práce také zhodnocuje použitelnost technologie GraphQL v kombinaci s jazykem TypeScript.

Abstract

The goal of this bachelor thesis is to design and create simple workflow system for processing and managing business processes. The workflow system is based on one part of BPMN 2.0 specification, which is focused on business process model. Specifically, it focuses on private business processes. Developed simple workflow system is designed to operate on Node.js platform and its application interface is implemented using GraphQL. TypeScript language was chosen for implementation. This thesis also evaluates usability of GraphQL technology in combination with TypeScript language.

Klíčová slova

Workflow systém, BPMN, JavaScript, TypeScript, GraphQL, Node.js

Keywords

Workflow system, BPMN, JavaScript, TypeScript, GraphQL, Node.js

Citace

VÁLKA, Miroslav. *Podpora workflow na platformě JavaScript*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

Podpora workflow na platformě JavaScript

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Miroslav Válka
27. května 2020

Poděkování

Rád bych poděkoval svému vedoucímu panu Ing. Radku Burgetovi, Ph. D za pomoc a odborné vedení při vypracování této bakalářské práce.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce je navrhnout a vytvořit jednoduchý workflow systém neboli systémů pro zpracovávání a řízení podnikových procesů, jenž je určený k provozu na platformě Node.js. Tyto systémy slouží firmám k zefektivnění, kontrole a vynucování jejich interních procesů a postupů, které vznikly za účelem úspory peněz, lidských zdrojů či času. V současnosti většina dostupných populárních řešení workflow systémů využívá virtuálního stroje Java a jsou implementovány v jazyce Java.

Nepředpokládá se, že vytvořený systém bude plně schopný konkurovat existujícím komerčním řešením. Přínosem této práce pro veřejnost je převážně demonstrace použitelnosti platformy Node.js pro tento typ systému, ale pro mne je hlavním přínosem získání zkušeností s pokročilým vývojem na platformě Node.js s využitím pro mne nových technologií.

V první části práce v kapitole 2 probíhá seznámení s konceptem workflow, zasvěcení do problematiky podnikových procesů a uvádí specifikaci Business Process Model and Notation (BPMN).

Následující část práce kapitola 3 se zaměřuje na technologii GraphQL. Probíhá zde představení a uvedení GraphQL, jenž je následováno bližším seznámením s ním a popisem jeho klíčových částí.

Kapitola 4 se zabývá analýzou systému pro řízení a zpracování podnikových procesů a zaměřuje se na návrh jednoduchého workflow systému, jenž vychází ze specifikace BPMN.

Po návrhu jednoduchého workflow systému následuje jeho realizace, které je věnována kapitola 5. V této kapitole postupně dojde k seznámením s jednotlivými klíčovými částmi systému, ze kterých se vyvíjený workflow systém skládá. Během popisu klíčových částí systému jsou zmiňovány a rozebírány i problémy a jejich řešení, ke kterým došlo při jejich vývoji.

Poslední část, která se nachází v kapitole 6, se věnuje testování a ověřování funkčnosti vyvíjeného workflow systému a jeho částí za využití automatizovaných testů a demonstrační klientské aplikace.

(Demonstrační klientská aplikace je dostupná na webové adrese `https://mwe.mwarcz.cz/`*. Rozhraní vyvíjeného workflow systému je dostupné skrze webovou adresu* `https://mwe.mwarcz.cz/graphql` *a pro účely přímého procházení rozhraní je dostupný i nástroj Playground na adrese* `https://mwe.mwarcz.cz/play`*.)*

Kapitola 2

Workflow a BPMN

Tato kapitola vás seznámí s pojmem workflow, jeho vztahem k řízení podnikových procesů a také ke specifikací BPMN (Business Process Model and Notation). K nalezení zde jsou informace o účelu programu nazývaného workflow engine a i proč nezaměňovat pojmy pracovní postup (Workflow) a obchodní pravidla (Business rules).

2.1 Workflow a workflow engine

Pracovní postup neboli taktéž workflow je řada úkolů. Nejedná se přitom o jednorázové úkoly, ale o úkoly, které se ve firmě nebo společnosti opakují ať už pravidelně či nepravidelně. Tato řada úkolů vede k dosažení určitého obchodního cíle. Aby byla zajištěna správnost dokončení všech úkolů pracovního postupu je potřeba programu, který bude sledovat celý proces pracovního postupu a zajistí, že všechny úkoly budou probíhat podle plánu. [13] [12]

Takovýto program je nazýván *workflow engine*. Někdy je označován taktéž synonymy jako *software workflow* nebo *workflow system*[13]. Tento program pomáhá vynucovat pracovní postupy, které společnost zavedla pro své vnitřní procesy. Zároveň také slouží k automatizaci pracovních postupů firmy a jejich zefektivnění, protože workflow engine usnadňuje tok informací nejen mezi pracovníky, ale i mezi vedením firmy, řídí správné provádění pořadí úkolů prováděnými lidmi či automatizovanými úkoly a událostmi. [13] [12]

Nejen v minulosti, ale i nyní v přítomnosti je většina pracovních postupů dokončována ručně a to skrze e-mailovou komunikaci nebo papírovou formou[12] nechávána ke schválení a k podpisu než se může přejít k dalšímu úkolu pracovního postupu. Při takovémto postupu dochází ke zpomalování prováděných úkolů z důvodu jejich fyzického cestování mezi lidmi, čekáním a nejistotou, jestli někdo objeví e-mail s požadavkem anebo může docházet k nepochopením při rozdělení zodpovědnosti za zpracování jednotlivých úloh.

Tyto problémy se právě snaží program workflow engine řešit a minimalizovat. Program podporuje zpracování pracovních postupů, obsahující sofistikovaná pravidla včetně možností opakování úloh, větvení toku, paralelního provádění toků a jiné.[21, str.1–2]

Firmy potažmo uživatelé tak mohou dostat nástroj, který jim poskytne například možnost zobrazit si termín pro zpracování úkolu, kdo je za úkol zodpovědný, přístup k potřebným dokumentům, jak probíhají úkoly pro daný pracovní postup.

Základními vlastnosti, kterými disponuje workflow engine je směrování, řízení a monitorování stavů úkolů. Mezi tyto úkoly může patřit například přijmutí nové objednávky, vyplnění formuláře, odeslání upomínky na e-mail klienta, upozornění skupiny pracovníků na nový úkon a další nejrůznější úkoly, které se mohou v pracovních postupech firem vysky-

tovat. Zároveň řeší organizační změny, kdy se zaměstnanci mohou vyhýbat novým postupům ve firmě nebo je plnit špatně. Díky programu budou informováni o změně pracovního postupu a program zajistí dodržování nového pracovního postupu a neumožní zahájit starý postup práce. [13] [12]

Existují tři hlavní funkce, které by měl workflow systém obstarávat. Ověření aktuální stavu úkolu a zda je možné nad ním provést zvolenou akci. Určení, zda je uživatel oprávněn provádět danou akci nad úkolem. A taktéž následně musí vykonat akci nad úkolem a vrátit, zda skončila úspěchem a nebo neúspěchem. V případě neúspěchu by se měl zachovat jedním z následujících chováním. Mohou se zahodit změny týkající se úkolu a úkol se vrátí zpět do stavu před provedením akce. Nebo se nastartuje zpracovávání chyby, kdy pro pokračování v úkolu musí být nejprve chyba vyřešena.

2.2 Souvislost mezi Workflow a Business Rules

Pracovní postup neboli workflow a obchodní pravidla čili business rules jsou dvě zcela rozdílné věci. Pro obě existují specifické programy nazývané workflow engine a business rules engine. Tyto programy bývají velmi často lidmi zaměňovány. I když oba dva poskytují uživatelům možnost změnit chování procesu ve firmě, tak ale mají mezi sebou daleko více rozdílů, než kolik mají mezi sebou podobností. [12]

Business rules engine je program obsahující sadu podmínek neboli pravidel, který na základě splnění těchto podmínek nebo určité části podmínek spustí odpovídající kód programu [12]. Tato sada pravidel tedy na základě široké řady kritérií definuje chování programu. Jeho účelem je uživatelům pomoci při rozhodování o postupech firmy, kdy program je schopen oproti člověku udělat rychlá a spolehlivá rozhodnutí nad velkém množstvím údajů, které by člověk nemusel vždy správně vyhodnotit [12]. Uživatelé, kteří mnohdy ani nemusejí znát logiku stojící za rozhodováním, zadají programu údaje vyplývající z jejich obchodních požadavků a program rozhodnutí učiní za ně.

V tabulce 2.1 je uvedeno stručné srovnání programů workflow engine a business rules engine.

Workflow Engine	**Business Rules Engine**
Založen na návrhu procesu	Řízen pravidly
Řídí přechod z jednoho úkolu na další a tok informací	Při splnění sady podmínek provede konkrétní rozhodnutí
Vytváří spojení mezi pracovníky nebo týmy pro efektivní práci	Šetří čas strávený složitým rozhodováním
Umožňuje jednoduchou automatizaci úkolů a řízení či kontrolu pracovního postupu	Zpracovává složité scénáře, provede komplexní rozhodnutí a před schválením zkontroluje velké množství pravidel
Pomáhá při provádění obchodního procesu	Pomáhá při vytváření obchodních rozhodnutí

Tabulka 2.1: Tabulka srovnání programů workflow engine a bussiness roles engine. Vychází z informací uvedených v následujících zdrojích [4, 12]

2.3 Existující nástroje

Existuje mnoho komerčních i nekomerčních řešeních poskytujících služby pro vybudování a správu pracovních postupů a podnikových procesů. Většina těchto řešení obsahuje mimo programu/modulu pro řízení pracovního postupu nebo pro řízení podnikových procesů (Nazýván též jako motor pro řízení procesů) také další nástroje pro podporu práce s procesy. Poskytují nástroje od nástrojů pro tvorbu a modelování procesů přes statistické nástroje až po nástroje pro správu uživatelů nebo správu úkolů pro podnikové uživatele.

Jedním ze standardů, které podporuje velká část z nich, je specifikace BPMN 2.0. Jedná se o rozšířenou specifikaci, která je pro programy z této kategorie téměř standardem. Více o specifikaci BPMN lze nalézt v kapitole 2.4.

Existující motory pro řízení procesů nepodporují kompletní specifikaci BPMN 2.0, ale pouze její části. V dokumentaci k nim je uvedeno, které elementy podporují a popřípadě i jejich chování. Pokud v dokumentaci je uvedena fráze *"Program vychází ze specifikace BPMN."* nebo podobná, tak nemusí být vstupní a výstupní soubory přenositelné mezi různými řešeními tj. kompatibilní pro zpracování motorem od jiného subjektu.

Camunda

Camunda BPM je platformou s otevřenými zdrojovými kódy *(open source)* vydané pod licencí *Apache License 2.0*. Platforma Camunda poskytuje prostředky pro automatizaci pracovních postupů a procesů. Motorem pro řízení procesů je nativní modul napsány v jazyce Java. Modul motoru pro řízení procesů vychází ze specifikace BPMN 2.0. Mimo tento modul obsahuje i řadu dalších modulů a nástrojů. K dispozici jsou nástroje pro modelování a návrh podnikových procesů, aplikace pro správu a plnění úkolů, aplikace pro správu uživatelů a skupin a další. [3]

Camunda za účelem řízení podnikových procesů, vytváření formulářů a práci s daty rozšiřuje elementy definované specifikací BPMN 2.0 o další sadu elementů. Elementy definované nástroji Camunda lze v souborech BPMN identifikovat prostřednictvím jmenného prostoru `http://camunda.org/schema/1.0/bpmn`.

Jimi vyvíjený systém workflow podporuje následující elementy ze specifikace BPMN[3]:

Participants Pool

Subprocesses Subprocess, Call Activity, Event Subprocess, Transaction

Tasks Service Task, User Task, Script Task, Business Rule Task, Manual Task, Receive Task, Send Task

Gateways Parallel Gateway, Inclusive Gateway, Exclusive Gateway, Event Gateway

Events None, Message, Timer, Conditional, Link atd.

Bonita

Bonita je platforma, která umožňuje vytvářet aplikace založené na procesech. Procesy na této platformě mohou být automatizované nebo částečně automatizované. Server Bonita se skládá ze dvou součástí z motoru pro řízení provádění procesů a z webového portálu, který poskytuje webové rozhraní koncovým uživatelům a administrátorům. Aplikace pro server je napsána v jazyce Java. Aplikace je distribuována pro spuštění na aplikačním serveru Java *(JVM = Java Virtual Machine)*. [1, Kap. 1]

jBPM

Sada nástrojů **jBPM** je určená pro vytváření podnikových aplikací, které pomáhají automatizovat podnikové procesy a rozhodnutí. Kromě hlavního modulu jBPM, obsahující jádro motoru umožňující provádět podnikové procesy definované ve specifikaci BPMN 2.0, má k dispozici řadu funkcí a nástrojů pro podporu podnikových procesů v průběhu celého jejich životního cyklu. Zdrojové kódy jsou volně dostupné *(open source)* a jsou vydány pod licencí *Apache Software License*. Moduly jBPM jsou napsány v jazyce Java a běží na jakémkoli JVM *(Java Virtual Machine)*. [24, Kap. 1.1]

2.4 Co to je BPMN?

Business Process Model and Notation neboli zkráceně jen **BPMN** je označení pro specifikaci vyvíjenou a vydanou organizací **The Object Management Group** *(OMG)*.

Primárním cílem specifikace je poskytnout snadno srozumitelný prostředek pro zápis podnikových procesů. Při návrhu specifikace BPMN byl kladen důraz na srozumitelnost pro všechny podnikové uživatele, kteří se účastní nebo jinak interagují s podnikovými procesy. Specifikace udává prostředky pro vizualizaci podnikových procesů.[19, str. 1 (pdf str. 31)][18, str. 1 (pdf str. 25)] Definuje grafickou notaci čili obsahuje grafické prvky s popisem jejich vlastností a chováním. Určuje vzájemné možné vztahy mezi grafickými prvky.

Sekundárním cílem specifikace je umožnit vizualizovat informace a data určené pro nástroje k provádění podnikových procesů zvané **WSBPEL** *(Web Services Business Process Execution Language)* způsobem srozumitelným pro podnikové uživatele a s obchodně orientovanou notací.[19, str. 31] WSBPEL používají soubory napsané nebo generované ve značkovacích jazycích patřících do podmnožiny jazyka XML *(eXtensible Markup Language)*. Tyto jazyky totiž nepatří mezi nesrozumitelnější prostředky pro široké spektrum uživatelů.

Práce na specifikaci BPMN začaly po sloučení dvou organizací OMG a BPMI *(Business Process Management Initiative)* v roce 2005. Roku 2007 [20] byla specifikace přijata organizací OMG jako standart a byla zveřejněna finální podoba první veze. Specifikace prodělala během času několik změn.

Současná specifikace BPMN 2.0 byla vydána v roce 2014 [20]. Velkou změnou oproti předchozím verzím je, že specifikace BPMN 2.0 definuje i způsob/pravidla reprezentace diagramu v souboru. S dokumentem obsahující textový popis specifikace BPMN 2.0 byly uvolněny i dokumenty určené pro strojové zpracování, umožňující kontrolu souborů s uloženými diagramy podnikových procesů. Tyto dokumenty jsou ve formátech **CMOF** *(Meta-Object Facility)*, **XSD** *(XML Schema Definition)* nebo **XSLT** *(eXtensible Stylesheet Language Transformations)*. [20]

V předchozí vydaná specifikace BPMN obsahovala pouze popis a definici grafických elementů pro modelování podnikových procesů. Za účelem sjednotit a standardizovat způsob předávání dat mezi nástroji a systémy, které využívají specifikaci BPMN, byla zvolena varianta ukládání do textového souboru ve formátu XML[19, str. 1, kapitoly 'Package XML Schemas']. V praxi se u těchto textových souboru ve formátu XML nesoucí informace o diagramu BPMN používá koncovka `.bpmn` nebo `.bpmn2` či `.bpmn20` namísto koncovky `.xml`, kterou běžně jsou označovány soubory ve formátu XML. Dále v textu budou tyto soubory nazývány zkráceně jako **soubory BPMN**.

Kapitola 3

GraphQL

V této kapitole je popsána technologie GraphQL. Lze zde nalézt odpovědi na otázky jako: "Odkud GraphQL vzešlo?", "Co bylo motivem vzniku GraphQL?", "Proč se stalo silnou konkurencí pro REST API?".

V jednotlivých podkapitolách se nachází nejen zodpovězení těchto otázek, ale i popsání základní struktur a principů pro tvorbu rozhraní za pomocí technologie GraphQL. Rozebrána je i problematika zabezpečení takto vytvořeného rozhraní a některých problémů či nevýhod, které GraphQL má oproti své konkurenci.

3.1 Co je GraphQL?

Otázka "Co je GraphQL?"je velice častá. Pro mnohé je GraphQL pouhá konkurence nebo častěji nazývána jako náhrada za dlouhou dobu používané a zažité rozhraní REST (Representational State Transfer). Toto tvrzení je ovšem jen částečná pravda.

REST je architektura rozhraní, která umožňuje přistupovat k datům pomocí standardních metod protokolu HTTP. Oproti tomu GraphQL je specifikace složená z mnoha částí. GraphQL je nazýváno tvůrci i komunitou jako dotazovací jazyk sloužící ke komunikace klienta se serverem [7]. Poskytuje úplný a srozumitelný popis dat pro aplikační programovatelné rozhraní.

O GraphQL by se dalo i říci, že se jedná o protokol. O protokol umožňující na straně serveru definovat a sdělit klientovi, jaká data potažmo služby jsou na něm dostupná. A na druhé straně klientovi umožňuje žádat po serveru jen ta data, která skutečně požaduje. S funkčností a principy týkajících se GraphQL proběhne seznámení v kapitole 3.3.

Název GraphQL není zvolen náhodně. Nese v sobě hlavní myšlenku. Nepřemýšlejme o datech z pohledu adres URL a jejich formátu, ale spíše jako o grafech a stromech vzniklé průchodem grafu[2]. Aplikace pracují s daty, která mají mezi sebou vazby. Takže data, se kterými aplikace pracují, tvoří datovou strukturu grafu. Název GraphQL je tedy odvozen/složen ze slov Graph (graf), Query (dotazovací), Language (jazyk). Takže se jedná o dotazovací jazyk nad daty uloženými ve struktuře grafu.

3.2 Vznik GraphQL

Za vznikem GraphQL stojí jeden z velkých hráčů na poli informačních technologií, a to firma Facebook. Aplikace Facebook určená pro mobilní operační systém iOS měla problémy. Jednou z hlavních příčin problémů bylo příliš vysoké využití sítě k získávání dat. [2]

Proto Facebook v roce 2012 zahájil vývoj, jehož cílem bylo přepracování nativních mobilních aplikací Facebooku. Z počátku aplikace byli jen obalem nad mobilní verzí webu. Bohužel s přibývajícími funkcemi se aplikace Facebooku staly složitějšími. Měly špatný výkon a častokrát havarovali. [2]

Zlomovým okamžikem ve vývoji byla práce s rozhraním News Feed. Vysoká složitost potřebných dotazů na server, jak k získání dat pro zobrazení klientovi, tak i pro zápis dat na server, vedla k frustraci vývojářů. Tento stav frustrace inspiroval několik vývojářů firmy Facebook k zahájení projektu, ze kterého vzešlo GraphQL. [2]

K veřejnému odhalení a uvolnění GraphQL došlo až v roce 2015. V tomto roce na konferenci ReactEurope 2015 představil hlavní vývojář Lee Byron GraphQL[1]. Byl také vydán pracovní návrh specifikace GraphQL a implementace referenční knihovny napsané v jazyce Javascript s názvem GraphQL.js[6]. Po zveřejnění začaly komunity různých programovacích jazyků vytvářet vlastní implementace knihovny pro daný programovací jazyk.

Až do roku 2018 spadal projekt GraphQL pod firmu Facebook, ale dne 7. listopadu 2018 byl projekt přesunut od firmy Facebook pod nově založenou nadační organizaci GraphQL Foundation. Nadační organizace GraphQL Foundation spadá pod neziskové technologické konsorcium The Linux Foundation. [17]

3.3 Seznámení s GraphQL

V následujícím textu jsou popsány hlavní vlastnosti technologie GraphQL a je nastíněno, jakým způsobem GraphQL funguje. Nejprve je potřeba seznámit se s klíčovými komponentami, které jsou v GraphQL používané.

Zdrojem informací pro většinu této podkapitoly jsou jednotlivá vydání oficiální specifikace GraphQL z roků 2015 a 2018[6, 7], volně dostupná dokumentace pro vývojáře k naučení GraphQL [8], původní článek a prezentace od vývojáře Lee Byron[2] a moje znalosti/zkušenosti nabyté během mého zájmu o GraphQL.

Mezi klíčové komponenty používané v GraphQL patří:

- Query (Dotazy)
  - Dotaz je požadavek klienta posílaný na server o vykonání určité akce či akcí.
  - Dotaz má podobu textového řetězce, kde textový řetězec je napsán v dotazovacím jazyce GraphQL.
  - Dle struktury dotazu jsou serverem vrácena data provedené akce či akcí.
  - Vrácená data serverem jsou ve formátu JSON.
  - V oficiální specifikaci bývá požadavek klienta na službu GraphQL označován jako dokument.
- Resolvers (Řešitelé)
  - Řešitelé interpretují dotazy poslané na server.
  - Tvoří prostředníka mezi dotazem klienta a vnitřní logikou a daty aplikace.
  - Objevuje se v nich komunikace s databází, zabezpečení, doprovodné výpočty a logika aplikace.
  - Každý řešitel vrací data.

[1] Videozáznam je dostupný na Youtube: https://www.youtube.com/watch?v=WQLzZf34FJ8

  - Výsledná data jsou posílány klientovi ve formátu JSON.

- Schema (Schéma)
  - Schéma popisují klientovi, které akce jsou na serveru dostupné, jaké datové typy může očekávat a jak je možné strukturou grafu procházet.
  - Základní stavební jednotkou schématu je typ.
  - Schema je zpracováváno typovým systémem GraphQL.
  - Podrobný popis schématu a jeho tvorby bude popsán v kapitole 3.3.

Důležitých vlastností GraphQL je celá řada. Jedna z nich je, že definuje tvar dat. Podobnost ve struktuře dotazů a struktuře vrácených dat není náhodná. Toto chování usnadňuje předvídat tvar dat vrácený serverem klientovi. Tvar dat dotazu a odpovědi nesedí jen v určitých případech. [2] [7, kap.7]

Případy, kdy struktura dat v odpovědi nemusí sedět se strukturou dotazu jsou následující[7, kap.7]:

- Došlo k lexikální či syntaktické chybě při zpracování dotazu klienta.
- Došlo k sémantické chybě při zpracování dotazu. Může se jednat o nesprávný datový typ v parametru nebo o nedefinovaný datový typ, který není definován ve schématu.
- Řešitelé nenalezli požadovaná data při průchodu.
- Řešitelé vrátily chybu, způsobenou vnitřní logikou aplikace. Například nedostatečná oprávnění, chybné rozsahy parametrů a jiné.

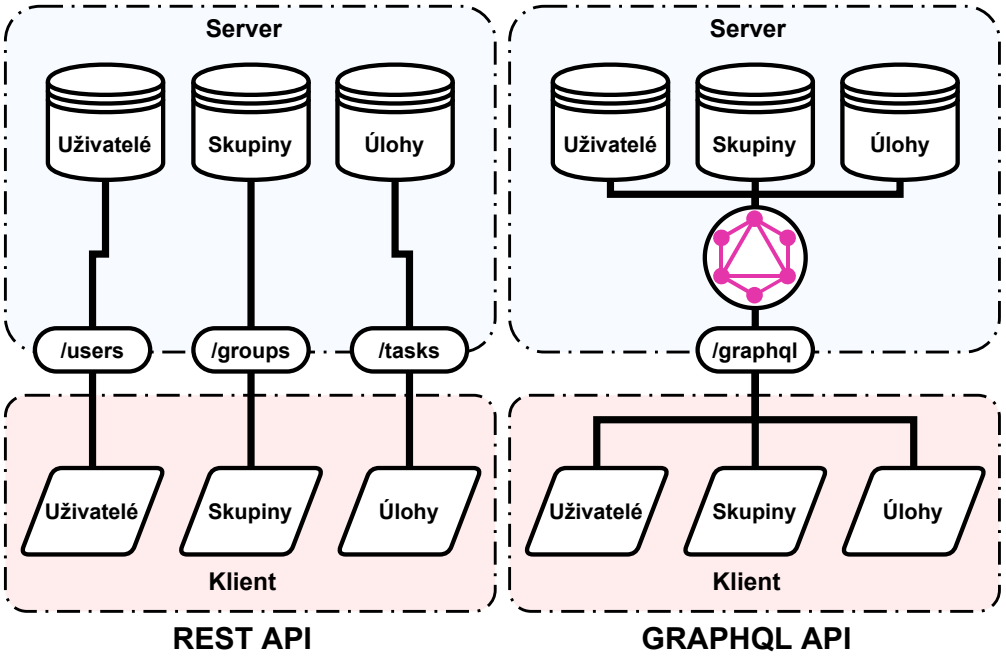
Další aspekt, který má GraphQL, je, že je silně typovaný. Každá úroveň dotazu odpovídá určitému typu, který je definován ve schématu. Každý typ popisuje sadu v něm dostupných polí a jakého typu jsou daná pole. [2] [7, kap.3]

GraphQL má hierarchistickou povahu. Je schopné sledovat vztahy mezi objekty. To pro vývojáře znamená, že je jim dána jistá volnost pohybu při procházení grafem dat a mohou si být jisti, že data dostanou v pořadí, v jakém je procházeli grafem dat. [2] [7, kap.2–3]

GraphQL není úložiště dat pro klienta, ale pouze poskytuje protokol pro komunikaci[2]. Každé pole ve schématu má vlastní funkci (řešitele) na serveru, který dostává a poskytuje data. Nediktuje vývojáři na serveru způsob, odkud má brát data, ani jakým způsobem je má získávat a dokonce ani nezakazuje jejich modifikaci. Data může nechat generovat programem, číst ze souborů nebo je může získávat z databází jako MySQL, Oracle Database, Neo4j, MongoDB a dalších. Díky tomu je možné GraphQL nasadit i na již fungujících serverech s existující obchodní logikou (Business Logic) a zaběhlými databázemi[2].

Na obrázku 3.1 je znázorněno umístění zpracování GraphQL při komunikaci mezi klientskou aplikace a serverovou aplikací. Zároveň si zde je možné všimnout, že klient přes GraphQL k datům přistupuje skrze jeden přístupový bod, ve kterém může požádat o všechna pro něj potřebná data a to i v jednot požadavku. Oproti tomu při komunikaci klienta se serverem přes rozhraní REST musí klient pro získání dat v některých případech poslat na server více požadavků, než získá všechna potřebná data.

GraphQL dále umožňuje takzvanou introspekci. Klient se může serveru dotazovat na typy/akce, které server poskytuje. Tato vlastnost dává prostor pro vznik a použití pomocných nástrojů, které zjednoduší vývoj aplikací.[2] Existuje již celá řada nástrojů, které vlastnost



Obrázek 3.1: Ukázka rozdílné komunikace klienta a serveru při získávání dat u rozhraní REST a GraphQL. (Diagram byl vytvořen v nástroji draw.net.)

introspekce využívají. Jedním typem nástrojů jsou nástroje sloužící ke generování kódu primárně pro statické programovací jazyky. Tím odpadá nutnost ručně psát datové typy pro uložená data. Dále velmi užitečnými nástroji jsou nástroje pro tvorbu dotazů na server. Tyto nástroje šetří vývojářům klientských aplikací čas a nervi při učení a prozkoumávání rozhraní serveru. Většinou poskytují grafické rozhraní, našeptávač pro snazší procházení grafem dat a zvýraznění syntaxe včetně lexikální a syntaktické kontroly. [7, kap.4]

Poslední významná vlastnost GraphQL by se dala nazvat *rozšiřitelnost*. Rozhraní vytvořené v GraphQL není nutné tzv. *verzovat* [2] po provedení změn, jako je tomu u rozhraní REST. Klient svým dotazem přesně definuje tvar vrácených dat serverem. To dává možnost přidávat nové typy, funkce nebo rozšířit stávající typy o nová pole, aniž by došlo k ovlivnění ostatních klientských aplikací, které aktuálně využívají rozhraní serveru. Tímto způsobem se sice může stát, že některá odpovídající pole budou zastaralá, ale i nadále budou fungující.[2] Díky takovému postupnému vývoji rozhraní je usnadněn proces zachování kompatibility. Zpětná kompatibilita je velmi důležitá převážně ve chvílích, kdy uživatelé nejsou agresivním způsobem nuceni k pravidelným aktualizacím klientských aplikací.

Schéma

Následující obsah podkapitoly vychází z informací obsažených ve specifikacích GraphQL[7, kap.3][6, kap.3] a částečně i z popisu prvotní knihovny GraphQL[9] *(Jedná se o moji interpretaci a pochopení. Pro celkové a korektnější vysvětlení doporučuji projít oficiální zdroje napsané v anglickém jazyce.).*

[2]Přístupový bod k rozhraní na serveru bývá dělen do několika různých větví dle vydané verze daného rozhraní[2]. Důvodem je převážně zpětná kompatibilita klientských aplikací.

Schéma je jednou z klíčových komponent použitou v GraphQL. Schéma je složeno ze sady datových typů.

Tato sada datových typů kompletně popisuje všechny možné formáty dat, které je možné od serveru získat pomocí dotazu zaslaného klientem.

Pro vytvoření schématu byl vytvořen jednoduchý jazyk tzv. jazyk schématu GraphQL. Jazyka schématu GraphQL je velmi intuitivní pro vývojáře. Jeho syntaxe nám může připadat podobná jako syntaxe v jazyce TypeScript pro definování typů. V GraphQL má vývojář možnost deklarovat několik druhů datových typů od skalárních typů přes výčtové typy až po složené typy.

V prvé řadě budou popsány skalární typy, které by se měly nalézt v každé implementaci knihovny GraphQL. Tyto skalární typy jsou někdy také označovány jako listové typy. Je to z důvodu, že v grafu (Při reprezentaci dat aplikace.) či stromu (Při reprezentaci dotazu.) se jedná o listy.

- **Int**
  - Jedná se o znaménkové 32bitové celé číslo.
- **Float**
  - Jde o znaménkové číslo s pohyblivou desetinnou čárkou s dvojitou přesností.
- **String**
  - Představuje textový řetězec. Znaky textového řetězce jsou v kódování UTF-8.
- **Boolean**
  - Logický datový typ nabývající dvou pravdivostních hodnot. Možné hodnoty jsou true a false.
- **ID**
  - Představuje jedinečný identifikátor.
  - Stejně jako typ String se jedná o textový řetězec s kódováním znaků UTF-8.
  - Narozdíl od typu String se u něj nepředpokládá, že bude čitelný pro člověka.
  - Je používán převážně pro nové načtení oběktu nebo jako klíč do mezipaměti.

Zde je nutné zmínit, že různé implementace mohou obsahovat i další skalární typy. Ve většině implementací je možné i vytvořit vlastní skalární typy. Na obrázku 3.2 je ukázáno, jakým způsobem se vytvářejí tyto vlastní skalární typy.

```
scalar MySuperScalarType
scalar Date
scalar Time
```

Obrázek 3.2: Ukázka kódu pro vytvoření vlastních skalárních typů *MySuperScalarType, Date, Time*

.

Datový typ výčet je dalším datovým typem, jenž je v GraphQL dostupný. Jedná se o zvláštní druh skalárního typu. Tento datový typ je omezen na určitou množinu hodnot, kterých může nabývat. Obrázek 3.3 ukazuje, jak může vypadat deklarace nového typu za pomocí datového typu výčet. V konkrétních implementacích knihoven pro poskytnutí GraphQL může být interpretace datového typu výčet interně odlišná od specifikace v GraphQL. Například v jazyce JavaScript je datový typ výčet z GraphQL interpretován jako celočíselná hodnota. Podobně tomu může být v ostatních jazycích, kde neexistuje datový typ výčet. Nicméně z pozice klientské aplikace nehraje interní implementace na server žádnou roli.

```
enum TaskState {
    UNKNOWN
    WAITING
    ONGOING
    INTERRUPTED
    CANCELED
    FINISHED
}
```

Obrázek 3.3: Ukázka kódu pro vytvoření vlastního datového typu pomocí datového typu výčet. Vytváří typ *TaskState*, který může nabývat jen šesti pro něj definované hodnoty.

Dalšími datovými typy jsou datový typ objekt a datový typ pole. Typ objekt patří mezi nejzákladnější typy, které se při tvorbě schématu používají. Na obrázku 3.4 lze vidět deklaraci dvou typů Query a Task, které jsou deklarované jako datový typ objekt. Datový typ objekt se skládá z názvu typu a ze sady položek, kde každá položka má uveden svůj jedinečný název v rámci objektu, jakého datového typu může nabývat a volitelně sadou parametrů, které přijímá. Nastal čas ukázat si na obrázku 3.4 ukázat použití datového typu pole. Datový typ pole je homogenní strukturovaný datový typ. Jeho použití lze vidět u typu Query u položky *tasks*. Datový typ pole je zapisován do hranatých závorek a vně závorek je uvedeno, jaký datový typ dané pole obsahuje (např. [Int]).

```
type Query {
    tasks: [Task!]
    task(id: ID!): Task
}
type Task {
    id: ID!
    name: String!
    state: TaskState!
}
```

Obrázek 3.4: Ukázka kódu obsahující dva datové typy *Query* a *Task*. Takto vypadá deklarace typu objekt.

Existuje několik speciálních názvů pro datové typy objekt. S typem Query byl již dříve zmiňován. Spolu s typy Mutation a Subscription patří mezi speciální názvy jejichž existence ve schematu má speciální význam. Schéma GraphQL musí obsahovat minimálně typ Query.

Typ Query, Mutation a Subscription jsou vstupními body serveru neboli vstupní uzly grafu, které budou tvořit kořen dotazu. Dotaz klienta, který dotaz vytváří, bude začínat právě jedním z této trojice typů. Existence tří různých typů, které je možné použít jako kořen pro vytvářený dotaz naznačuje, že budou mezi nimi drobné rozdíly při zpracování a interpretaci dotazů.

Typ Query má býti používán pro získávání dat. Jednotlivé položky příchozího dotazu s kořenem Query mohou být zpracovávány asynchronně. Není zaručeno, že položky dotazu budou provedeny ve stejném pořadí jako jsou uvedeny v dotazu. Proto by neměl typ Query obsahovat položky, které mohou měnit data či stav serveru. Pro tyto účely zde existuje právě typ Mutation. U typu Mutation probíhá zpracování všech položek příchozího dotazu sériově. Je tedy zaručeno, že první položka v dotazu se zpracuje dříve než se zahájí zpracování druhé položky. V typu Mutation se mohou vyskytovat položky, které vytváří nová data, upravují stávající data nebo mažou existující data. Na druhou stranu by se zde neměly vyskytovat položky pro pouhé získávání dat. Důvod je prostý. Sériové zpracovávání dotazu je pomalejší, než když je dotaz zpracováván paralelně. (Zda typ Query bude dotazy zpracovávat paralelně záleží na konkrétní implementaci knihovny GraphQL.)

Ještě nebyl zmíněn typ Subscription. K vysvětlení typu Subscription se hodí využít příklad jako například klientské aplikace, které implementují chat. Pokud jedna z nich vloží na server novou zprávu, tak ostatní aplikace chtějí být upozorněny na tuto novou zprávu. Aplikace musí serveru tuto informaci, že je má upozornit, nějak sdělit. Typ Subscription by měl obsahovat právě položky, které provádí takovouto akci. Klientův dotaz s kořenem Subscription nedojde k vrácení dat ihned jako je tomu u Query. Server posílá data klientovi pokaždé, když dojde k dané události na niž klient chce být upozorněn.

Řešitelé

Tato podkapitole čerpá z informací uvedených v rámci popisu prvotní knihovny GraphQL[9] a výukových lektcí o GraphQL[8] v kombinaci s informacemi ze specifikace GraphQL[7, kap.6]

Aby celé GraphQL mohlo fungovat a bylo možné klientovi poslat data o která žádá, je na serveru nezbytná existence logiky pro provedení dotazu, mapování a získávání dat.

Existuje celá řada různých implementací knihoven, které dávají možnost vybudovat a použít GraphQL na serveru. První veřejně dostupná implementace knihovny pro GraphQL byla napsána v jazyce JavaScript a byla určena pro platformu Node.js. V současnosti naleznete pro každý známější programovací jazyk, který se používá pro budování serverových aplikací (Nazývané anglickým slovem Backend.), minimálně jednu implementaci této knihovny.

Velmi důležitou roli v těchto knihovnách hrají tzv. Resolvers neboli česky Řešitelé. Až na některé výjimky naleznete v každé implementaci možnost neli nutnost vytvořit vlastní řešitele. V následujícím popisu se bude hovořit o základu většiny implementací a až později bude zmíněno pár výjimek.

Ve schematu GraphQL jsou uvedené veškeré typy a jejich položky. Každá položka všech typů ve schématu GraphQL je reprezentována funkcí. Řešitelé (Resolvers) je označení právě těchto funkcí. Při provádějí dotazu GraphQL dochází k postupnému volání řešitelů, který odpovídajících právě zpracovávané položce. Řešitelé mohou vrátit skalární hodnotu nebo objekt. V případe vrácení objektu bude pokračovat dále zpracování objektu. Dojde k volání dalších řešitelů pro získání dat z objektu dle hierarchie typů definovaných ve schématu

GraphQL. Tento proces zpracovávání a zanořování se do objektu končí ve chvíli, kdy řešitel vrátí skalární hodnotu.

Výsledek řešení dotazu čili výsledná data by měly býti v objektu se strukturou stromu v níž skalární hodnoty vrácené řešiteli tvoří listy stromu.

Řešitele programuje vývojář a je tedy na vývojáři, jak danou funkci naprogramuje. Vně funkce vývojář řeší získání dat. Data můžou býti získávána z libovolné databáze, mohou býti generována, je možné libovolně je transformovat či je nejrůzněji modifikovat. Dále zde vývojář řeší případné restrikce a způsoby zabezpečení dat například dle rolí uživatele.

Aby mohl vývojář provádět ve funkci výše zmiňované a ještě více, tak každá funkce dostává při zavolání parametry. Každá funkce řešitel by měla obdržet následující parametry/informace.

- Rodiče
  - Rodičovský objekt neboli též předchozí objekt.
  - Jedná se o referenci na objekt, z něhož se přišlo k řešení současné položky.
  - Je častokrát využíván pro řízení přístupu skrze hrany grafu.
- Argumenty
  - Parametry předané klientem v dotazu u dané položky.
- Kontext
  - Objekt obsahující kontextové informace (Sdílené informace poskytované všem řešitelům).
  - Obsahuje například informace jako aktuálně přihlášený uživatel, přijení k databázi nebo zvolené restrikce.
- Infofmace
  - Objekt s informacemi, které jsou předpřipraveny knihovnou.
  - Můžou se zde nalézat specifické informace pro daný dotaz, jaký typ by funkce měla vracet, typ rodiče a další.
  - Tato část je individuální pro různé implementace knihoven.

Fauna

Jednou ze zajímavých výjimek z knihoven poskytujících GraphQL je Fauna. Fauna je tzv. serverless databáze. Serverless je filozofii, kdy se buduje klientská aplikace bez budování vlastní serverové aplikace. Takováto klientská aplikace využívá služeb, které poskytují jiné specializované servery.

Servery Fauna poskytují databázové služby a možnosti řízení přístupu k datům. Administraci je možné provádět skrze jejich webovém rozhraní. Pro komunikaci klientské aplikace se servery Fauna je používáno právě GraphQL. Při používání služeb Fauna vývojáři aplikací neřeší implementaci řešitelů, neboť v tomto typu řešitelé již existují. Dotazy GraphQL, které klient posílá, jsou zpracovány aniž by vývojáři museli sami programovat získávání dat z databáze, autorizaci a autentizaci. [5]

Neo4j

Další velice zajímavou výjimkou je i integrace GraphQL do Neo4j.

Projekt Neo4j zahrnuje stejnojmennou databázi a nástroje pro práci s touto databází. Databáze Neo4j se odlišuje od ostatních běžně známých databází, kterými jsou relační databáze Oracle, MySQL, SQLite, MariaDB nebo zástupci dokumentově orientovanými databázemi MongoDB, Cosmos DB. Databáze Neo4j se řadí mezi tzv. grafové databáze. Data v grafových databázi jsou ukládána do struktury grafu. Pro komunikaci s databází Neo4j se používá jazyk Cypher Query Language.[14]

Co vás napadne existuje-li grafová databáze Neo4j a dotazovací jazyk GraphQL pro dotazování nad daty uloženými v grafu, který se stává více a více populárním? Zde přichází ke slovu právě dodatečná implementace přidávající možnost využít GraphQL v databázi Neo4j. Neo4j interně převádí přijatý dotaz v jazyce GraphQL na dotaz v jazyce Cypher Query Language.[14]

Hasura

Fakta v následujícím odstavci pochází z webu Hasura [11].

Podobně jako vznikaly projekty, které přímo propojovaly rozhraní REST s databází, tak vznikají podobné projekty i pro GraphQL. Jedním takovým způsobem je například převedení požadavku REST na odpovídající dotaz v jazyce SQL. Hasura dělá velice podobnou věc. Hasura převádí dotaz GraphQL na dotaz v jazyce SQL. Samozřejmě tento způsob má jistá pravidla a omezení, ale jedná se o zajímavé alternativní použití, které si najde své příznivce.

Kapitola 4

Analýza a návrh systému

Cílem projektu je vytvořit jednoduchý workflow systém pro zpracování a řízení podnikových procesů. Pro jasnou identifikaci bude dále v textu vyvíjený jednoduchý workflow systém zmiňován pod jménem ***My Workflow Engine*** nebo častěji pod zkráceným názvem ***systém MWE***. Při návrhu a vývoji systému MWE se počítá s částečnou kompatibilitou se specifikací *BPMN 2.0* a je vycházeno převážně z dané specifikace *BPMN 2.0* obohacené o některé elementy nezbytné pro správné řízení procesů. Systém MWE je cílen na platformu *Node.js*, která poskytuje běhové prostředí pro aplikace napsané v jazyce *JavaScript*. Pro demonstraci funkčnosti systému MWE je vyvíjena i jednoduchá demonstrační klientská aplikace.

Projekt je rozdělen na dva na sobě částečně závislé podprojekty s oddělenými repositáři typu git.

Jedním z podprojektů je právě systém MWE, který bude realizován jako serverová aplikace (Anglicky označována termínem *backend*.), která s klientem komunikuje za pomocí rozhraní *GraphQL* vytvořeného pomocí dostupných nástrojů a technologií z ekosystému Node.js. Zdrojové kódy systému budou napsány v jazyce TypeScript, neboť TypeScript umožňuje typovou kontrolu, která má při vývoji rozsáhlejších projektů pozitivní dopad na redukci chyb. Jelikož se bude jednat o serverovou aplikaci, tak bude tento podprojekt pojmenován ***MWE Server*** (*Server MWE*) a stejnojmenně bude pojmenován i repositář typu git.

Druhým podprojektem je již zmiňovaná jednoduchá demonstrační klientská aplikace. Bude se jednat o klientskou aplikaci, která bude z větší části pouze grafickým panelem či terminálem pro práci a manipulaci se serverem MWE. Klientská aplikace nemusí přímo řešit oprávnění a práva uživatelů k aktivitám a akcím, neboť kontrola oprávnění a práv k provádění akcí bude plně kontrolována na straně serveru MWE. Pro zachování jednoznačného pojmenovaní bude zvolen název ***MWE Client*** (*Klient MWE*) jako označení pro klientskou aplikaci a její repositář typu git.

4.1 Interní datový model pro systém

Specifikace BPMN 2.0 udává celou řadu elementů pro modelování podnikových procesů a také jakým způsobem elementy ukládat/načítat do/ze souboru. Formát souboru je podmnožinou textového souboru ve formátu *XML*. Bližší informace ke specifikaci BPMN jsou v kapitole 2, anebo přímo ve specifikaci BPMN 2.0[1].

[1]Specifikace BPMN 2.0 https://www.omg.org/spec/BPMN/2.0/PDF

Jednotlivé elementy, ze kterých jsou vymodelovány podnikové procesy, je nezbytné klasifikovat do skupin, zobecnit a vytvořit interní datový model, aby bylo možné s elementy v systému MWE efektivně pracovat. Ve specifikaci jsou uvedeny elementy obstarávající různé typy úloh (*např. script task, service task, manual task, user task aj.*), událostí (*např. start event, end event, intermediate throw event, intermediate catch event*), toky a brány (*např. parallel gateway, inclusive gateway, exclusive gateway aj.*) dále také elementy pro deklaraci procesů a spolupracujících objektů. Interní datový model by byl příliš robustní, kdyby měl obsahovat samostatné tabulky pro každý typ elementu definovaného ve specifikaci BPMN.

Na první pohled lze vidět, že elementy je možné zobecnit do několika základních skupin jako jsou *procesy, úlohy, události, brány, sekvenční toky, datové objekty* a *datové toky*.

Individuální rozdílné atributy konkrétních elementů s jejich hodnotami v rámci jedné skupiny elementů budou ukládány do jednoho textového pole. Jejich individuální rozdílné vlastnosti a jejich hodnoty budou v poli ukládány v serializované formě textu ve formátu JSON. Data uložená ve formát JSON jsou snadno čitelná z pohledu aplikace psané v jazyce JavaScript. S těmito vlastnostmi elementů nebude potřeba přímo provádět akce v databázi, ale pouze při běhu v systému MWE.

I přes uvedené rozdělení elementů do skupin by byly vztahy mezi skupinami příliš propletené a komplikované. Při návrhu interního datového modelu dojde ještě ke zredukování počtu skupin elementů ještě na základě podobnosti při zpracování. Tím dojde ke sloučení některých skupin elementů. Konkrétně dojde ke sloučení skupin *úlohy, události* a *brán* do jedné skupiny **uzly**. Skupina *uzly* obsahuje všechny elementy, které vykonávají nějakou činnost či přímo svojí existencí ovlivňují sekvenční toky v procesech.

Ze souboru BPMN načtené informace o procesech popisují podnikové procesy, které plní úlohu šablony či vzoru. Ze šablony procesu budou vytvářeny individuální instance procesu. V následujícím popisu interního modelu dat je patrné rozdělení do tří hlavních částí:

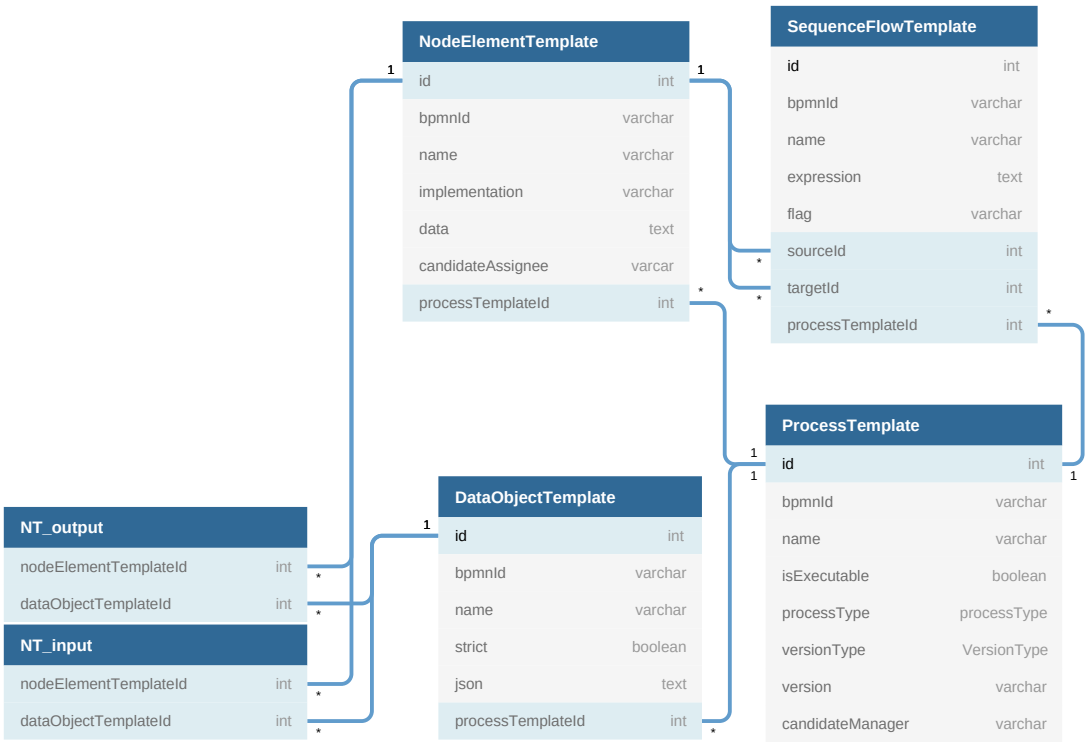
- Šablony
- Instance
- Uživatelé a skupiny

Obrázek 4.1 s ER diagramem ukazuje tabulky/entity představující dříve zmiňované skupiny elementů a jejich vzájemné vztahy. Všechny entity na obrázku 4.1 patří mezi **šablony**, které dohromady nesou informace o *šablonách podnikových procesů*.

Entita `ProcessTemplate` je šablona podnikového procesu. Každá šablona procesu je kořenovým elementem, na který se odkazují všechny elementy, který daný proces tvoří. Nejdůležitější atributy entity pro běh systému MWE mimo jednoznačného identifikátoru (*id*) patří i příznak spustitelnosti (*isExecutable*) a název skupiny obsahující kandidáty na pozici manager pro daný proces (*candidateManager*).

Entita `NodeElementTemplate` představuje skupinu uzlů neboli elementů ovlivňující svým přímím chováním proces. Šablona uzlu obsahuje čtveřici důležitých atributů. Mezi ně patří jednoznačný identifikátor šablony uzlu (*id*), název implementace pro obsluhu daného uzlu (*implementation*), individuální vlastnosti elementů a jejich hodnoty (*data*) a název skupiny obsahující potencionální nabyvatele uzlu (*candidateAssignee*). Uzly mají mezi sebou orientované vztahy přechodu z jednoho uzlu na druhý takzvané sekvenční toky.

Tyto sekvenční toky jsou představovány entitou `SequenceFlowTemplate`. Přechod z jednoho uzlu na druhý může být podmíněn nějakým výrazem (*expression*) nebo příznakem (*flag*).



Obrázek 4.1: ER diagram obsahující část interního datového modelu, která představuje entity ze skupiny šablon a jejich vzájemné vztahy. (Diagram byl vygenerován nástrojem dbdiagram.io.)

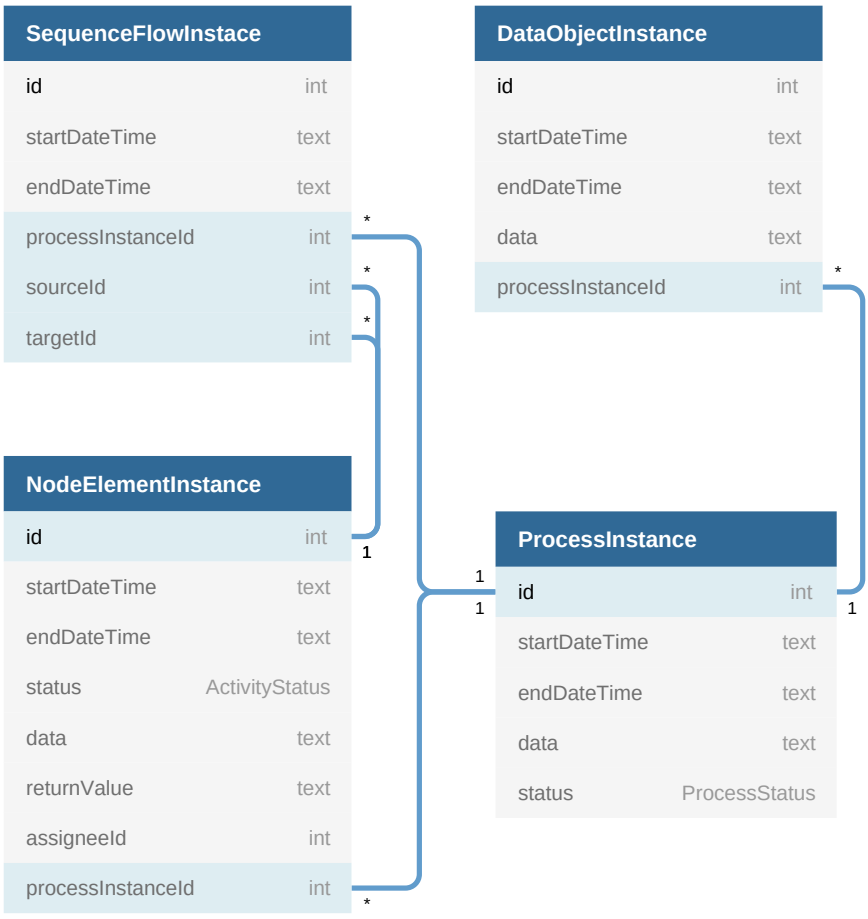
Jednou z posledních entit je `DataObjectTemplate`. Jedná se o entitu představující šablonu datového objektu. Pro systém MWE jsou nezbytné atributy název datového objektu (*name*), příznak striktnosti ukládání dat (*strict*) a výchozí data datového objektu (*json*).

Poslední dvojice entit z obrázku 4.1 představuje vztahy vstupních a výstupních dat pro uzly neboli datové toky v podnikových procesech.

Na obrázku 4.2 lze nalézt ER diagram vyobrazující entity, které zastupují instance. Instance podnikového procesu sice vychází ze šablony podnikového procesu, ale instance procesu nemusí obsahovat stejný počet instancí uzlů nebo sekvencí jako počet šablon uzlů či sekvencí, které obsahuje šablona procesu, z níž vychází. I dvě instance procesu vycházející z jedné šablony procesu se nemusí svých složením shodovat. Pro toto chování je vysvětlení následující:

- Není potřeba při vytvoření instance procesu vytvářet instance všech jejích částí z níž může být instance procesu složena.
  - Instance uzlů a instance datových objektů postačuje vytvářet až ve chvíli, kdy jsou skutečně potřeba.
  - Např. Instance uzlu před dokončením řekne, které další instance uzlů se mají vytvořit a s nimi jaké instance sekvenčního toku.
  - Např. Instance datových objektů budou vytvořeny až při první změně dat v nich obsažených.

- Instance procesu nemusí sletovat všechny sekvenční toky a do některých větví se nemusí nikdy dostat.
  - Je zbytečné vytvářet instance uzlů k jejichž zpracování se během procesu není možné vlivem řízení toku procesu dostat.
- Cyklus vně procesu znamená opakovaný přechod do nějakého již jednou provedeného uzlu a není vhodné zahodit informace o předchozím zpracování.
  - V takovémto případě je na místě vytvořit více instancí daných uzlů a u každého uchovávat jejich individuální informace, aby bylo možné sledovat celou historii zpracování procesu.
- Při vytvoření instance procesu bude vždy vytvořena jedna instance uzlu.
  - První instancí uzlu bude startující událost.



Obrázek 4.2: ER diagram obsahující část interního datového modelu, která představuje entity ze skupiny instancí a jejich vzájemné vztahy. (Diagram byl vygenerován nástrojem dbdiagram.io.)

Budou blíže představeny entity z obrázku 4.2. Všechny entity instancí mají některé společné atributy. Jsou jimi jednoznačný identifikátor a také datum a čas jejich vytvoření a dokončení.

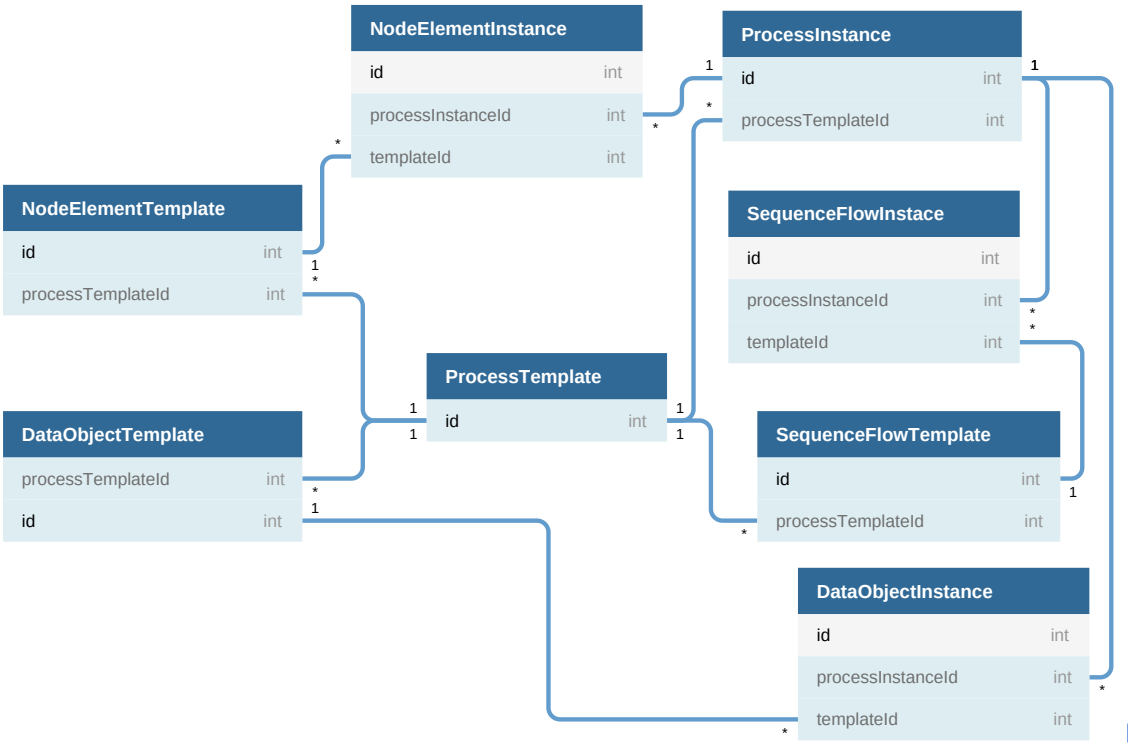
Entita `ProcessInstance` představuje instanci procesu vytvořenou ke konkrétní šabloně procesu. Atribut obsahující hodnotu aktuálního stavu, ve kterém se proces nachází (*status*) je velmi důležitým jak pro systém MWE, tak i pro uživatele. Vně instance procesu se nachází i prostor pro individuální data dané instance procesu (*data*), které budou k dispozici instancím uzlů při zpracování. Podobně jako u individuálních dat u šablon i zde se jedná o textovou položku obsahující serializovaný text ve formátu JSON.

Entita `DataObjectInstance` obsahuje datové objekty, ve kterých již byli pozměněny data, která nesou. Uložená data (*data*) jsou opětovně realizovány textovou položkou obsahující text ve formátu JSON.

A úplně stejně je tomu i u entity `NodeElementInstance`, která obsahuje data (*data*), která jsou individuální pro danou instanci uzlu. V instanci uzlu jsou pro systém MWE důležité atributy pro uchování aktuálního stavu instance (*status*) a identifikátor uživatele (*assigneeId*), který s uzlem naposledy manipuloval *(Pro uživatele uvedeného u instance uzlu je v textu užíváno označení nabyvatel, neboť uživatel u takové instance nabývá práva s ní manipulovat a vkládat do ní data.)*.

Poslední ještě nezmíněnou entitou je `SequenceFlowInstance`, jejíž role je pouze nést informaci o průchodu do dané instance uzlu skrze daný sekvenční tok.

Již bylo naznačeno, že mezi šablonami a instancemi jsou vztahy. Tyto vztahy jsou vyobrazeny na obrázku 4.3, jehož ER diagram zobrazuje chybějící vztahy, které nebylo možné postihnout u předchozí dvojice obrázků 4.1 a 4.2 z důvodu přehlednosti.



Obrázek 4.3: ER diagram ukazuje vztahy mezi entitami ze skupiny šablon a entitami ze skupiny instancí. (Diagram byl vygenerován nástrojem dbdiagram.io.)

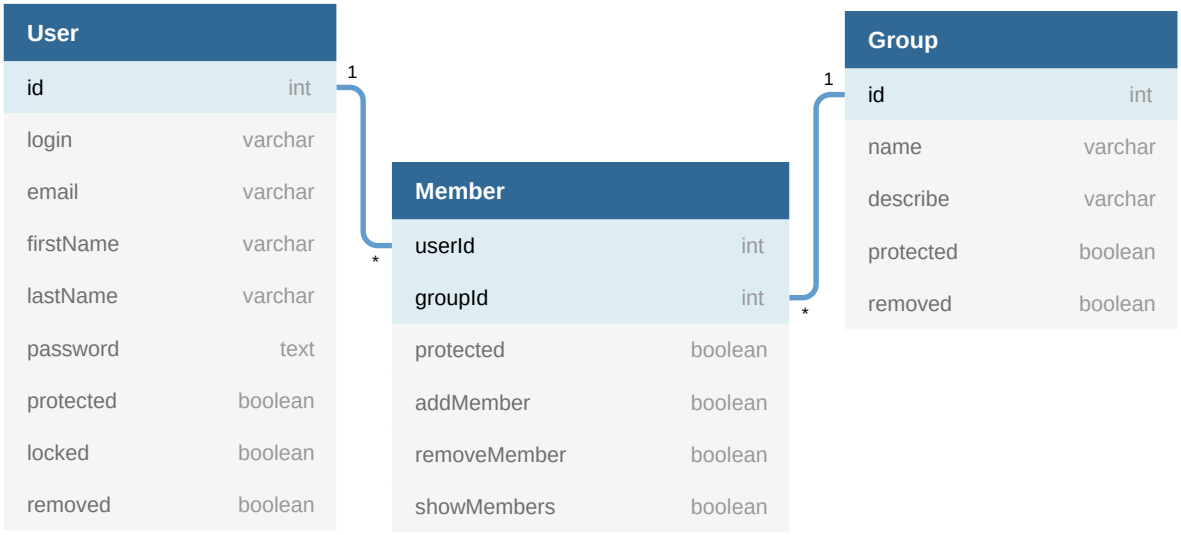
V systému MWE je potřeba mimo samotných podnikových procesů mít i informace o uživatelích, skupinách a jejich členstvím v jednotlivých skupinách. Někteří uživatelé a skupiny budou mít v systému MWE speciální postavení. Budou to takzvaně *chráněné skupiny, chránění uživatelé* a *chráněná členství*. Práva k provádění interakcí v systému MWE budou vázaná na členství uživatelů v konkrétních skupinách. Práva k provádění interakcí budou dvojího typu:

- Globální
- Lokální

Globální práva jsou navázaná na členství uživatelů v *chráněný skupinách*. Jedná se například o práva vytvářet, odebírat, mazat, upravovat uživatele či skupiny. Nebo nahrávání nových šablon procesů, jejich mazání či úprava.

Lokální práva jsou vázaná na konkrétní objekty, se kterými chce uživatel manipulovat na základě uživatelova *členství* ve skupině. Mezi takovéto případy patří obsloužení instance uzlu nebo spuštění procesu ale i vidění dalších členů ve stejné skupině či jejich přidávání do konkrétní skupiny.

V podkapitole 4.2 *Služby a interakce se systémem* je k nalezení podrobnější rozbor práv a oprávnění včetně jejich vlivu na možné interakce se systémem.



Obrázek 4.4: ER diagram vyobrazuje vztahy entit spadajících do skupiny uživatelé a skupiny. (Diagram byl vygenerován nástrojem dbdiagram.io.)

Obrázek 4.4 popisuje entity nesoucí právě tyto informace. Entita `User` zastupuje uživatele systému. Mimo atributy obsahující osobní informace o uživateli je zde i dvojice atributů pro přihlašování uživatelů do systému. Jedná se o unikátní přihlašovací jméno (*login*) a heslo (*password*), které nejlépe nebude uchováváno v prostém tvaru. Dále zde je trojice příznaků určující, zda je uživatel chráněn před smazáním (*protected*), jestli se může přihlašovat do systému (*locked*) a příznak o odstranění uživatele (*removed*). Příznak o odstranění uživatele je nedestruktivní způsob smazání uživatele. V praxi se jedná o jeho skrytí. Hlavním důvodem a účelem je zachování historie zpracování a obsloužení podnikových procesů.

Entita `Group` představuje skupiny, které pod sebou sdružují uživatele. Každá skupina má svoje unikátní jméno (*name*), které slouží k její identifikaci a popis, který by měl obsahovat stručný popis či charakteristiku skupiny. Podobně jako uživatelé i skupina má příznak ochrany před smazáním (*protected*) a příznak odstraněné skupiny (*removed*). Existence příznaků má stejný důvod jako u uživatelů.

Entitou, která tvoří přemostění mezi dvěma předchozím, je entita `Member`. Uživatel je členem v různých skupinách a skupiny mají členy. U členství jsou mimo příznaku pro ochranu před odstraněním (*protected*) i příznaky specifikující lokální práva daného člena skupiny pro danou skupinu v níž je členem. Tato lokální práva mohou dát uživatelovi možnost vidět další členy skupiny (*showMembers*), přidávat členy do skupiny (*addMember*) či odebírat členy ze skupiny (*removeMember*).

4.2 Dekompozice systému

Systém pro řízení a zpracování podnikových procesů je komplexním programem a je nezbytné rozložit jej na části, u kterých bude snazší popsat jejich funkcionalitu a zpřehlednit jejich vývoj.

Systém MWE bude jako vstup pro nahrání šablon procesů zpracovávat soubor BPMN. Ze souborů BPMN vytáhne nezbytná data a informace o podnikových procesech, které převede a uloží do jejich odpovídající reprezentace v interním datovém modelu. Ze šablon procesů budou vytvářeny instance procesů. Instance procesů a jejich uzly budou zpracovávány. Systém MWE na základě implementace udávané uzly dané instance uzlů zpracuje. Tato aktivita se bude opakovat a bude potřeba připravovat data pro zpracovaní uzlů a řízení chování systému na základě výsledků po zpracování uzlů.

Zároveň systém musí poskytovat své služby skrze server HTTP/HTTPS za použití rozhraní GraphQL. Služby, které bude systém poskytovat, je nutné zabezpečit a podmínit interní logikou, která zajistí správné použití při komunikaci s okolním světem.

Z krátkého popisu fungování systému MWE vyplývá rozdělení na následující části:

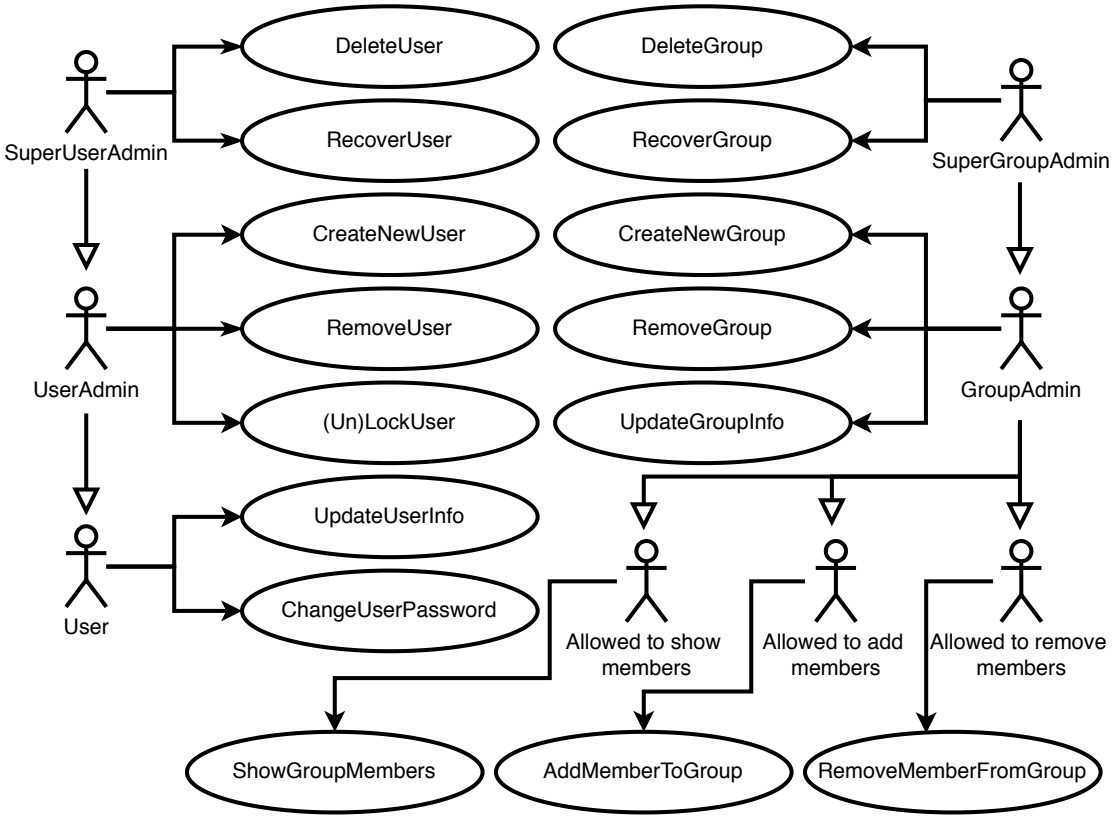
- Stavitel šablon procesů
  - Převod textu v souboru BPMN na data interního datového modelu.
- Hlavní běhové jádro
  - Zpracování instancí procesů a uzlů.
- Server pro běh zpracování procesů
  - Řízení postupného zpracování instancí uzlů.
- Server GraphQL
  - Poskytování rozhraní pro komunikaci skrze internet.
- API služeb
  - Zabalení služeb a interakcí, které jsou poskytovány systémem pro práci v něm.

Služby a interakce se systémem

Nezávisle na zvoleném prostředku (*REST, GraphQL, CLI apod.*) ke komunikaci uživatele se systémem je nezbytnou součástí vývoje definovat jaké služby a interakce bude uživatel moci po systému vyžadovat. Není žádoucí, aby jakýkoliv uživatel systému MWE měl možnost využívat všech služeb systému bez omezení. Z tohoto důvodu systém MWE bude rozlišovat různé role, které budou dovolovat obsluhu a rozhodovat o přístupnosti dostupných služeb a interakcí pro konkrétního uživatele. Interakce, které uživatelovi budou dostupné, lze rozdělit do částí:

- Interakce nepřihlášeného uživatele
- Interakce s uživateli a skupinami
- Interakce s podnikovými procesy

Systém MWE pracuje s podnikovými procesy jednoho podniku a neočekává se žádná přímá interakce anonymního uživatele neboli nepřihlášeného uživatele se systémem. Se systémem budou pracovat výhradně uživatelé, jejíž existence v systému bude chtěná podnikem. Ale je zde jedna výjimka. Existuje jedna interakce povolená pro anonymního uživatele. Jedná se o akci přihlášení (*LogIn*) při níž je ověřena identita anonymního uživatele.



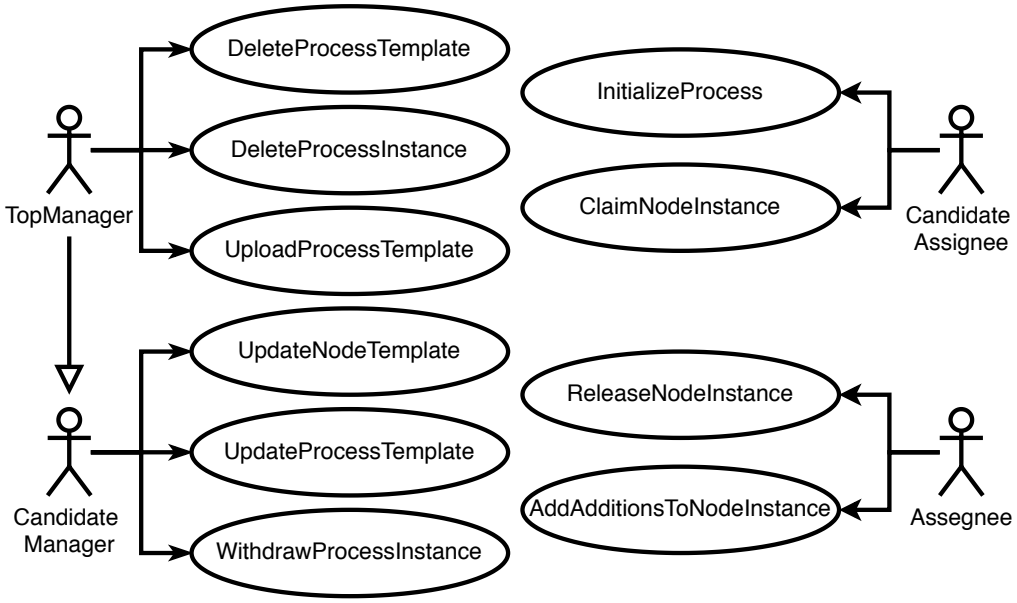
Obrázek 4.5: Diagram případů užití obsahuje aktéry, jenž představují role v systému, a interakce, které mohou vykonávat, v rámci administrace uživatelů a skupin v systému. (Diagram byl vytvořen v nástroji draw.net.)

Na obrázku 4.5 je diagram případů užití vyobrazující interakce s uživateli a skupinami. Je zde znázorněno celkově 8 rolí:

- `User`
  - Přihlášený uživatel
  - Role je udělena, každému přihlášenému uživateli.
  - Je mu dovoleno upravovat své osobní a přihlašovací údaje.
- `UserAdmin`
  - Administrátor uživatelů
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny `UserAdmin`.
  - Mezi jeho privilegia patří vytváření nových uživatelských účtů, odstraňování existujících uživatelských účtů nebo jejich uzamykání v případě potřeby.
  - Oproti roli 'User' je mu umožněno upravovat osobní a přihlašovací údaje všech jemu dostupných uživatelů.
- `SuperUserAdmin`
  - Jedná se o speciální typ administrátora uživatelů, který má navíc některá oprávnění.
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny `SuperUserAdmin`.
  - Může provádět destruktivní operaci trvalé smazání uživatelského účtu nebo jeho obnovení ze stavu odstraněného do stavu aktivního.
- `GroupAdmin`
  - Administrátor skupin
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny `GroupAdmin`.
  - Jeho privilegia jsou vytváření nových skupin, odstraňování existujících skupin, úprava informací o skupinách.
  - Dále může také přidávat nové členy do libovolné skupiny a odebírat členy z libovolné skupiny.
- `SuperGroupAdmin`
  - Jde o speciální typ administrátora skupin, který má navíc některá oprávnění.
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny `SuperGroupAdmin`.
  - Může provádět destruktivní operaci pro trvalé smazání skupiny nebo obnovení skupiny ze stavu odstraněná do stavu aktivní.
- `Allowed to show members`
  - Povolení vidět ostatní členy skupiny.
  - Role je udělena uživatelům, kteří mají u svého členství ve skupině nastaveno povolení.
  - Povolení je uděleno vždy na konkrétní skupinu, v níž je uživatel členem.
  - Takovémuto uživateli je dovoleno zobrazit další členy, kteří jsou s ním ve skupině.
- `Allowed to add members`

  - Povolení přidávat členy do skupiny.
  - Role je udělena uživatelům, kteří mají u svého členství ve skupině nastaveno povolení.
  - Povolení je uděleno vždy na konkrétní skupinu, v níž je uživatel členem.
  - Uživatel s daným povolením může přidávat do skupiny nové členy.

- `Allowed to remove members`
  - Povolení odebírat členy ze skupiny.
  - Role je udělena uživatelům, kteří mají u svého členství ve skupině nastaveno povolení.
  - Povolení je uděleno vždy na konkrétní skupinu, v níž je uživatel členem.
  - Uživatelovi je dovoleno odebírat členy ze skupiny.



Obrázek 4.6: Diagram případů užití obsahující aktéry, jenž představují role v systému, a interakce nad podnikovými procesy, které mohou v rámci systému vykonávat. (Diagram byl vytvořen v nástroji draw.net.)

Interakce s podnikovými procesy má také vlastní role, které je možné vidět na obrázku 4.6. Obrázek 4.6 obsahující diagramem případů užití pro interakci s podnikovými procesy má 4 role.

- `Candidate Manager`
  - Kandidát na pozici manažera
  - Role je udělena uživatelům, kteří jsou členy skupiny, jejíž název se shoduje s názvem skupiny uvedeným u šablony procesu.
  - Uživatel může upravovat vybrané části informací v šablonách podnikových procesů.

  - V jeho kompetenci je i stáhnutí/zrušení běžící instance procesu, ale jen pokud instance procesu byla vytvořena ze šablony procesu spadající pod jeho kompetenci.

- `TopManager`
  - Nejvyšší manažer
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny `TopManager`.
  - Nejvyšší manažer může vykonávat stejné akce jako kandidát manažera, ale jeho práva se týkají všech procesů v systému.
  - Navíc má privilegia nahrávat nové šablony podnikových procesů a provádět destruktivní operace jakými jsou mazání instancí procesů a mazání šablon instancí procesů.

- `Candidate Assignee`
  - Kandidát na nabyvatele
  - Role je udělena uživatelům, kteří jsou členy skupiny, jejíž název se shoduje s názvem skupiny uvedeným u šablony uzlu.
  - Uživateli je dovoleno spouštět nové instance procesů na konkrétních uzlech, které představují startovací události.
  - Dále uživatel může zabrat/obsadit uzel, který čeká na obsloužení (např. Manuální úloha).

- `Assignee`
  - Nabyvatel
  - Role je udělena uživateli, který si zabral/obsadil instanci uzlu pro obsloužení.
  - Hlavní úlohou nabyvatele je obsloužit instanci uzlu a dodat jí potřebná data.
  - Pokud nabyvatel není schopen instanci uzlu obsloužit, je možné ji uvolnit pro někoho jiného.
  - (Mechanizmus zabírání instance uzlu je nezbytný pro vyloučení kolize současného obsloužení instance.)

Rozšíření specifikace BPMN

V souborech BPMN se vyskytují elementy patřící do různých jmenných prostorů. Jmenné prostory jsou jednoznačně identifikovány pomocí adresy URI. Následující seznam uvádí jmenné prostory a s nimi jejich nejčastěji pojící se aliasy v souborech BPMN.

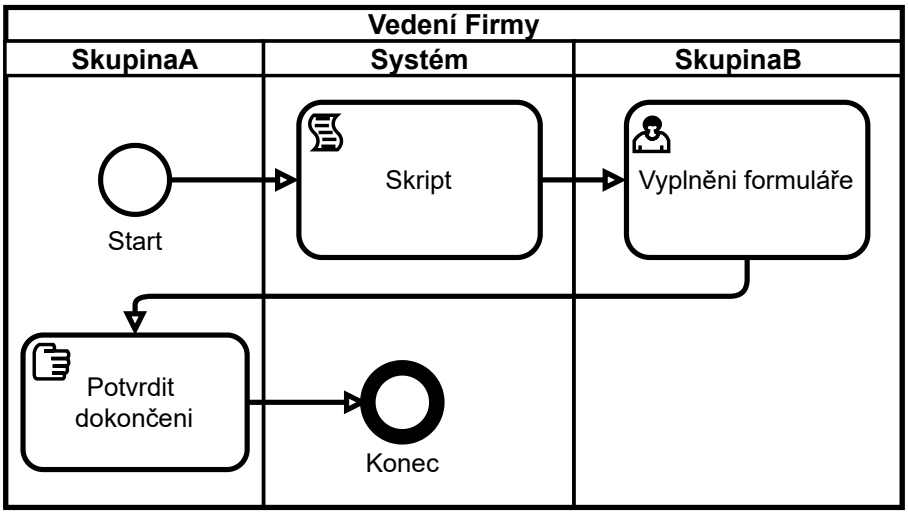
- xsi - `http://www.w3.org/2001/XMLSchema-instance`
- bpmn - `http://www.omg.org/spec/BPMN/20100524/MODEL`
  - Prvky/elementy BPMN 2.0.
  - Definování struktury podnikových procesů, propojení, názvy, výrazy atd.
- bpmndi - `http://www.omg.org/spec/BPMN/20100524/DI`

  - Popis grafické reprezentace elementů/prvků BPMN 2.0.

- dc - `http://www.omg.org/spec/DD/20100524/DC`
  - Určování pozice/polohy a rozměrů objektů v diagramu.
- di - `http://www.omg.org/spec/DD/20100524/DI`
  - Uřčení pozic a směru vektorů.
- camunda - `http://camunda.org/schema/1.0/bpmn`
  - Jmenný prostor obsahující elementy a atributy rozšiřující specifikaci BPMN 2.0.
  - Doplňují potřebné informace pro správné fungování nástrojů Camunda.
- mwe - `http://www.mwarcz.cz/mwe/bpmn/`
  - Prostor pro mnou definované elementy a atributy.

Specifikace BPMN obsahuje řadu elementů, ale některé typy informací neumožňuje v příznivé podobě definovat. Z tohoto důvodu nástroje od společnosti *Camunda* přidávají vlastní jmenný prostor s definovanými elementy či doplňkovými atributy. Podobně tuto absenci řeší i *jBPM*.

Systém MWE taktéž bude vyžadovat některé doplňkové informace pro správné řízení podnikových procesů. Pro elementy doplňující potřebné informace pro systém MWE bude vyhrazen jmenný prostor `http://www.mwarcz.cz/mwe/bpmn/`. Převážně se bude jednat o konstrukce, které obohatí soubory BPMN o možnost definovat výchozí data například pro datové objekty.



Obrázek 4.7: Ukázka jednoduchého podnikového procesu, který je složen z elementů definovaných specifikací BPMN. (Tato ukázka podnikového procesu byla vymodelována v nástroji draw.net.)

V nástrojích Camunda nebo jBPMN je řešeno přidělování uživatelů k úlohám nebo k událostem prostřednictvím speciálních atributů, které přidávají v rámci svého jmenného prostoru. Nicméně specifikace BPMN obsahuje elementy nazývané bazén (*pool*) a pruh

(*lane*), jenž slouží ke grafickému znázornění osoby nebo skupiny, která je přiřazena úlohám a událostem v nich se nalézajících.

Systém MWE využije těchto elementů pro výběr skupin uživatelů zodpovídajících za obsluhu úloh anebo spouštění instancí procesu. Na obrázku 4.7 lze vidět jednoduchý podnikový proces obsahující trojici pruhů. Chování systému MWE pro podnikový proces na obrázku 4.7 bude následující:

- Počáteční událost se nachází v pruhu *SkupinaA*.
  - Jakýkoliv uživatel, který je členem skupina s názvem *SkupinaA*, může spouštět instance procesu.
- Uživatelská úloha Vyplnění formuláře se nalézá v pruhu *SkupinaB*.
  - Jakýkoliv uživatel patřící do skupiny *SkupinaB* může vyplnit formulář.
- Manuální úloha Potvrzení dokončení je obsažená v pruhu *SkupinaA*.
  - Jakýkoliv uživatel, jenž je členem skupiny *SkupinaA*, může provézt potvrzení.

Kapitola 5

Implementace

Obsahem kapitoly je podrobné seznámení s implementací jednotlivých částí projektu, jejich funkcí a způsobem použití. U jednotlivých částí projektu jsou k nalezení informace, jakým způsobem daná část pracuje, jaké jsou její omezení a známé problémy.

Kapitola začíná popisem části, která umožňuje nahrávat do systému podnikové procesy, a následuje ji podrobný rozbor části, jenž tvoří srdce vyvíjeného systému workflow. K čemu by člověku bylo srdce, kdyby neměl krevní oběh. Podobně je na tom i systém workflow, a proto se v další části kapitola věnuje serveru pro běh zpracování uzlů. V kapitole není opomíjena ani komunikace systému s okolním světem prostřednictvím webového serveru nebo komunikace v rámci klíčových částí, které spolupracují a tvoří fungující systém workflow.

5.1 Stavitel šablon procesů

Hlavní úlohou stavitele šablon procesů je zpracovat textový vstup, kterým je obsah souboru BPMN, a z něj vyextrahovat potřebné informace o podnikových procesech pro sestrojení šablon podnikových procesů. Získané informace o vlastnostech podnikových procesů, jejich struktuře a tocích následně převede a uloží do struktur interního datového modelu.

Soubory tvořící stavitele šablon procesů se v repositáři ***MWEServer*** nachází v adresáři `src/bpmnBuilder/` a třída `BpmnBuilder`, která reprezentuje přímo stavitele šablon procesů, se nalézá v souboru `bpmnBuilder.ts`.

Obsah souboru BPMN je textový řetězec ve formátu XML. Tento textový řetězec je vstupem pro stavitele šablon procesů, který pro předzpracování využívá balíček/modul *fast-xml-parser*[1]. Tento balíček se postará o lexikální analýzu textového řetězce ve formátu XML a provede taktéž základní syntaktickou analýzu textového řetězce. Výstupem balíčku, který pokračuje dále ke zpracování, je objekt jazyka JavaScript. Objekt obsahuje normalizovaný abstraktní syntaktický strom, který byl vytvořen na základě vhodně zvolené konfigurace. Konfiguraci pro balíček *fast-xml-parser* obsahuje soubor `fxp.config.ts`.

Aby bylo možné při vývoji využívat všech výhod, které jazyk TypeScript nabízí, jsou v souboru `bpmnFxm.ts` definovány datové typy, které popisují strukturu normalizovaného abstraktního syntaktického stromu složeného z elementů definovaných specifikací BPMN. Definice těchto datových typu je rovněž závislá na zvolené konfiguraci pro balíček *fast-xml-parser* v souboru `bpmnFxm.ts`. Definované datové typy zároveň omezují při vývoji možné

[1] fast-xml-parser https://www.npmjs.com/package/fast-xml-parser

přístupné vlastnosti objektů, které představují elementy a jejich atributy, na pouze nezbytné minimum.

Stavitel šablon procesů je shovívaví k existenci neznámých elementů. Důvody jsou následující:

- Pouze částečná podpora elementů udávaných specifikací BPMN.
  - Dostupné editory pro modelování podnikových procesů generují soubory BPMN včetně některých redundantních dat nebo elementů, které nejsou podporovány systémem MWE.
  - Jedná se nejen o elementy udávané specifikací BPMN ale i o elementy odpovídající doplňujícím specifikací (Camunda BPM má své doplňující elementy stejně jako jBPM.).
- Vlastní elementy
  - Pro správnou funkčnost systému MWE jsou v souborech očekávány i jiné elementy nesoucí potřebné údaje.

Balíček *fast-xml-parser* provádí jen základní syntaktickou analýzu a její výsledek je dále zpracován pomocí syntaktického analyzátoru umístěného v souboru `parser.ts`. Ten vytváří vlastní abstraktní syntaktický strom, který obsahuje i objekty, jenž odpovídají strukturám z interního datového modelu. Definice datových typů tvořící vlastní abstraktní syntaktický strom jsou definovány v souboru `bpmnLevel.ts‘`.

Zpracování probíhá na třech úrovních.

- Úroveň 0
  - Nalezení kořenového elementu.
  - Získání jmenných prostorů a jejich aliasů.
  - Ověření kořenového elementu ‘definitions‘ zda patří do jmenného prostoru BPMN.
- Úroveň 1
  - Získání základních struktur šablon procesů (počty šablon podnikových procesů, jejich základní vlastnosti).
  - Jsou zde zpracovány i kolaborace a bazény, které jsou v systému MWE využívány k identifikaci skupin uživatelů.
- Úroveň 2
  - Zpracování a získání všech podporovaných elementů, ze kterých se skládá šablona procesu.

Syntaktický analyzátor obsahuje pouze synchronní funkce pro vytvoření abstraktního syntaktického stromu obsahující struktury či spíše entity interního datového modelu. Stavitel šablon procesů se na závěr postará o uložení vytvořených entit do databáze. Uložení do databáze probíhá za využití transakce, protože v případě selhání není žádoucí, aby v databázi zůstávali fragmenty špatně zpracovaných šablon procesů.

Problém typové kontroly jazyku TypeScript při práci s jmennými prostory XML

Zde je za vhodno zmínit problém, který se při vývoji objevil. V souborech BPMN se ve velké míře používají jmenné prostory. Zpracování elementů s jejich jmennými prostory zastoupenými libovolnými aliasy přináší do vývoje aplikace v jazyce *TypeScript* problém, který by byl v pouhém jazyce *JavaScrip* přehlédnutelným.

```
<foo:root xmlns:foo="http://namespace.foo"
    xmlns:bar="http://namespace.bar">
        <foo:data>foo</foo:data>
        <bar:text>bar</bar:text>
</foo:root>
```

Obrázek 5.1: Ukázka kódu v jazyce XML ukazující použití jmenných prostorů.

V ukázkovém kódu 5.1 je ukázáno použití jmenných prostorů v jazyce XML. Objekt jazyka JavaScript vygenerovaný balíčkem *fast-xml-parser* obsahuje položky jejichž název odpovídá přesně celému obsahu ve značce XML. Název značky je složen z aliasu jmenného prostoru a názvu elementu, které jsou oddělené od sebe dvojtečkou. Pro přístup k položce je nezbytné složit k ní dynamicky cestu pomocí načteného aliasu a názvu elementu.

V jazyce TypeScript se objeví problém, protože typový systém hlídá datové typy a jejich používání. Velká část volně dostupných zdrojů radí využít přetypování na typ ‘any‘ nebo vytvořit typ pro mapovaní libovolného textu na hodnotu `{ [key:string]: any }`.

Tato řešení, ale zcela zahodí výhodu typové kontroly. Při experimentování s různými konstrukcemi v jazyce TypeScript jsem objevil elegantní způsob, jak docílit potřebné funkcionality.

```
const objectJs: { data: String } = { data: 'foo' }
// Ukazka problemu
objectJs[`${ns}:data`] // Error
// Nalezena reseni
(objectJs as any)[`${ns}:data`] // Success => any
(objectJs as { [key:string]: any })[`${ns}:data`] // Success => any
((objectJs as any)[`${ns}:data`] as string) // Success => String
// Objevene reseni
objectJs[`${ns}:data` as 'data'] // Success => String
```

Obrázek 5.2: Ukázka kódu v jazyce Typescript ukazující případy přístupu k objektům skrze dynamické klíče, se zachováním kontroly datových typů.

V ukázce kódu 5.2 lze spatřit zjednodušenou ukázku daného problému. Na posledním řádku kódu v ukázce je řešení problému, které je použité i v syntaktickém analyzátoru. Není nezbytně nutné využívat přetypování na jiné datové typy, které umožňují přistoupit k položce pomocí libovolného textového klíče. Postačuje pouze přetypovat dynamicky složený textový řetězec na textový řetězec, který odpovídá klíči položky datového typu objektu. Typový systém se k takto získané položce chová stejným způsobem, jako kdyby byla získána běžným způsobem.

5.2 Hlavní běhové jádro procesů

Hlavní běhové jádro procesů je základní komponentou systému MWE a taktéž jednou z hlavních součástí serveru MWE. Při své činnosti nepracuje přímo se souborem BPMN, ale přistupuje k datům uložených ve strukturách interního datového modelu.

Jeho úlohami, za které zodpovídá, jsou:

- Vytváření nových instancí procesu či uzlů ze zvolených šablon.
- Načítání a příprava potřebných dat pro provádění instancí uzlů.
- Zpracování instancí uzlů a hlídání potřeb uzlů.
- Vyhodnocení výsledů a akcí, které nastaly při zpracování instancí uzlů.
- Úprava a uložení dat, které vznikly po vyhodnocených výsledů a akcí.
- Rušení aktivních instancí procesů.

Soubory tvořící hlavní běhové jádro procesů se v repositáři ***MWEServer*** nachází v adresáři `src/bpmnRunner/` a třída `BpmnRunner`, která reprezentuje přímo hlavní běhové jádro procesů, se nalézá v souboru `bpmnRunner.ts`. Pro snadnější a přehlednější vývoj se nenachází veškerá funkcionalita vně třídy `BpmnRunner`, ale je roztříštěna nejen do souborů s pomocnými funkcemi, ale i do zásuvných modulů.

Systém zásuvných modulů byl navržen přímo pro běhové jádro a jeho cílem je delegovat chování různých typů uzlů mimo běhové jádro. Díky větší univerzálnosti běhového jádra je možné rozšiřovat jádro o nové implementace uzlů či služeb pro uzly daleko snadněji. Podrobněji se systému zásuvných modulů seznamují kapitoly Obsluha a řízení běhu procesu 5.2.2, Zásuvné moduly s implementacemi služeb 5.2.5, Zásuvné moduly s implementacemi uzlů 5.2.6. Samotné roztříštění na menší části umožňuje snáze testovat funkčnost za pomocí automatizovaných testů.

5.2.1 Tvorba instancí

Aby hlavní běhové jádro procesů mohlo vůbec provádět zpracování instancí uzlů, tak musí v prvé řadě existovat instance procesů obsahující instance uzlů. Pro vytváření nových instancí procesů musí v systému již existovat šablony procesů. Hlavní běhové jádro procesů se nestará o způsob, jakým se do systému šablony procesů, uzlů, datových objektů a sekvenčních toků dostanou (Systém MWE k vytvoření šablon využívá služeb stavitele šablon procesů, který je popsán v kapitole 5.1).

Při vytváření nové instance procesu nestačí znát pouze šablonu procesu ze kterého instance procesu vychází, ale je nutné znát i šablonu uzlu, který bude vzorem pro vytvoření první instance uzlu ve vytvářené instanci procesu. První instance uzlu v instanci procesu může být nějaká startovací událost, ale není to podmínkou, neboť běhové jádro vidí všechny události, úlohy a brány jako sobě rovnocenné uzly. Každý uzel v sobě nese informaci o implementaci, která má být použita při práci s daným uzlem.

Pomocné funkce určené k vytváření instancí procesů, uzlů, datových objektů a sekvenčních toků se nachází v souboru `initHelpers.ts`. Tyto funkce ovlivňují výchozí hodnoty, které mají instance po vytvoření. Jsou využívány funkcemi běhového jádra, nejen k vytváření nových instancí procesů. Během vyhodnocování výsledků po zpracování instance uzlu může docházet k vytváření nových instancí uzlů a instancí sekvenčních toků, které

mají dle šablony procesů následovat, anebo jsou ukládána data do datových objektů, kdy v případě neexistence instance datového objektu musí být tato instance vytvořena.

5.2.2 Obsluha a řízení běhu procesu

Jedna z klíčových vlastností workflow systémů je schopnost kontrolovat a řídit podnikové procesy. O tuto vlastnost se stará právě hlavní běhové jádro procesů. Při kontrole a řízení podnikových procesů se běhové jádro procesů musí zabývat otázkami:

- Jaká instance uzlu se má zpracovat?
- Jak se má instance uzlu zpracovat?
- Jaká data a ze kterých datových objektů mají být instanci uzlu zpřístupněny?
- Kam uložit data po zpracování instance uzlu?
- Jaké další instance uzlů mají být vytvořeny po zpracování uzlu?
- Jakým způsobem zpracovaný uzel ovlivní instanci procesu?

Obrázek 5.3: Řetězec zpracování instance uzlu běhovým jádrem procesu, jenž znázorňuje všechny etapy, kterými při zpracování instance uzlu projde. (Diagram byl vytvořen za využití nástroje draw.net.)

Obrázek 5.3 obsahuje řetězec zpracování instance uzlu, jenž znázorňuje celý postup, který běhové jádro procesů opakuje při zpracovávání každé instance uzlu. Běhové jádro procesů dostává na vstupu informaci, která mu sděluje, jakou instanci uzlu má zpracovat.

Prvním krokem běhového jádra procesů je načtení všech potřebných dat. Mezi potřebná data v první řadě patří samotná instance uzlu a šablona uzlu z níž vychází. Dalšími důležitými daty, která jsou načtena patří instance procesu, do které patří instance uzlu, a šablona procesu dané instance procesu. Načtena je i celá řada dalších návazných dat, která jsou nutná pro další kroky.

Po načtení instance uzlu dojde k nalezení implementace, která má obsloužit danou instanci uzlu. Běhové jádro procesů může obsahovat mnoho odlišných implementací pro obsluhu různých typů uzlů. Dostupné implementace a názvy pod kterým jsou jednotlivé implementace dostupné závisí na knihovně zásuvných modulů s implementacemi uzlů, kterou

obdrželo běhové jádro při svém založení. Do této knihovny nahlíží v okamžiku, kdy potřebuje najít implementaci pro uzel. Podrobně se zásuvným modulům s implementacemi uzlů věnuje podkapitola 5.2.6.

Po nalezení implementace uzlu je následujícím krokem složení kontextu. Kontext obsahuje celou řadu informací, které běhové jádro procesů je ochotné dát k dispozici implementaci uzlu. Kompletní popis obsahu kontextu včetně jeho struktury je rozebrán v podkapitole 5.2.4. Nalezená implementace uzlu neslouží jen k přímému obsloužení instance uzlu, ale také obsahuje jistá nastavení, která ovlivňují výběr dat, jež budou ve složeném kontextu pro implementaci uzlu dostupná. V tomto kroku je pro běhové jádro nejpodstatnější dvojice nastavení, která ovlivňuje výběr vstupních datových objektů a výběr některých šablon uzlů. Vstupní datové objekty běhové jádro procesů rozlišuje do dvou rozsahů:

- Lokální rozsah
- Globální rozsah

Vstupní datové objekty patřící do lokálního rozsahu mají s načteným uzlem přímou vazbu. Tato vazba je dána vstupním datovým tokem (tj. přiřazení datového objektu), který vstupuje do uzlu. Vstupní datové objekty z globálního rozsahu jsou veškeré datové objekty, které instance procesu obsahuje. Lokální rozsah je výchozí volbou běhového jádra, které se snaží následovat datové toky. Globální rozsah je jednou z možností, jak může implementace uzlu obejít datový tok. Implementace uzlu dále může požadovat informace o jiných uzlech a běhové jádro procesů je ochotné některé informace o šablonách uzlů předat implementaci na základě jejího výběru. Podobně jako předchozí nastavení dovolovalo obcházet datový tok, tak i toto chování dovoluje obcházení. Konkrétně může sloužit například pro obcházení sekvenčního toku. V podkapitole 5.2.6 je možné nalézt kompletní popis, jaké náležitosti může a musí splňovat zásuvný modul s implementací uzlu.

Běhové jádro procesů dále mimo kontextu připraví v dalším kroku i soubor funkcí, které poskytují služby, jenž může využívat implementace uzlu. Soubor funkcí služeb je vytvořen z knihovny obsahující zásuvné moduly s implementacemi služeb, které běhové jádro procesů předem obohatí a základní moduly služeb.

Služby, které poskytují základní moduly služeb, jsou:

- Možnost sdělit, z jakých šablon uzlů mají být po zpracování aktuální instance uzlu vytvořeny nové instance uzlů.
- Možnost oznámit ukončení procesu, a to jak možné, tak i vynucené ukončení procesu.
- Možnost ukládat data do lokálního registru dat.
- Možnost ukládat data do globálního registru dat.

Běhovému jádru procesu je možné při vytvoření předat knihovnu obsahující moduly s implementacemi služeb, které mohou dodávat další služby, jež nejsou běžnou součástí běhového jádra. Více o zásuvných modulech s implementacemi služeb lze nalézt v podkapitola 5.2.5.

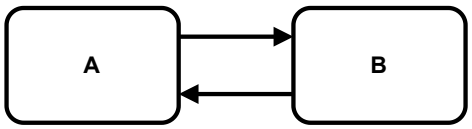
V pátém kroku řetězce zpracování instance uzlu dochází k připravovanému provedení implementace uzlu. Při provádění hraje hlavní roli implementace uzlu a její funkce. Jednotlivé etapy, kterými instance uzlu při provádění projde, odpovídají vždy konkrétní funkci, která může být obsažena v implementaci uzlu. Po provedení implementace (tj. po zpracování instance uzlu) dostává běhové jádro procesu nasbírané výsledky o provádění. Bližší informace o instancích uzlů a jejich provádění je obsahem podkapitoly 5.2.3.

V následujících krocích běhové jádro procesů zpracovává nashromážděné výsledky a vyhodnocuje další postup. Za prvé dojde k vyhodnocení datových výstupů, které vznikly při provádění. Datové výstupy budou uloženy do instancí datových objektů. Zde běhové jádro procesů kontroluje na základě nastavení v implementaci uzlu, které udává rozsah výstupních datových objektů, do kterých datových instancí může instance uzlu vkládat data. Podobně jako u vstupních datových objektů jsou i zde stejným způsobem rozlišeny rozsahy na lokální a globální. Pro lokální rozsah platí, že datové objekty mají k uzlu vztah daný datovým tokem. A pro obejití datového toku opět existuje globální rozsah, který umožňuje ukládat data do libovolného datového objektu, který je součástí daného procesu.

Následuje úprava globálního registru dat, ke které může dojít díky zásuvnému modulu pro sběr žádostí o vložení dat, který poskytlo běhové jádro. Globální registr dat složí k uložení a sdílení dat napříč instancemi uzlů. Globální registr dat je tedy společný pro všechny instance uzlu spadajících pod stejnou instanci procesu.

Stejně tak dojde k úpravě lokálního registru dat. Podobně jako globální registr dat slouží lokální registr dat k uložení dat. Ale data v něm uložená jsou primárně určena jen pro instanci uzlu, v níž se registr nachází. Lokální registr dat je využíván pro předávání dat mezi jednotlivými fázemi, do kterých se může instance uzlu dostat.

Běhové jádro procesů v dalším kroku rozhodne o vytvoření nových instancí uzlů, které mají být dle nasbíraných výsledků vytvořeny. Některé konstrukce podnikových procesů mohou být velmi nebezpečné.



Obrázek 5.4: Ukázka části konstrukce podnikového procesu s nekonečnou smyčkou, která může zablokovat systém workflow. (Diagram byl vytvořen za využití nástroje draw.net.)

Například rekurzivní vytváření instancí uzlů, které je vyobrazeno na obrázku 5.4. Po dokončení uzlu **A** dojde k vytvoření instance uzlu **B**, který vytvoří opět po dokončení novou instanci uzlu **A** a to celé se opět znovu zopakuje. V takovém případě může dojít i k úplnému zhroucení systému. Proto je v jádru zaveden bezpečnostní mechanizmus, který kontroluje počet instancí uzlu, jenž byly vytvořeny z jedné šablony uzlu v rámci jedné instance procesu. Ve výchozím nastavení dovoluje zabezpečení vytvářet z jedné šablony uzlu jen 10 instancí. Počet možných instancí uzlu, které je možné vytvořit, lze ovlivnit pomocí jednoho nastavení zásuvného modulu s implementací uzlu. To dovoluje pro daný typ uzlu snížit či zvýšit počet možných vytvořených instancí. Více o nastavení zásuvného modulu je k nalezení v podkapitole 5.2.6.

Předposledním krokem je vyhodnocen stav instance procesu. V prvé řadě je nezbytné zmínit stavy, do kterých se může dostat instance procesu. Ve specifikaci BPMN jsou všechny procesy vnímány stejně jako úlohy a jsou označovány jednotně jako aktivity (Activity). Ve specifikaci BPMN je uveden životní cyklus aktivit (str428), který ukazuje jednotlivé stavy a přechody mezi nimi.

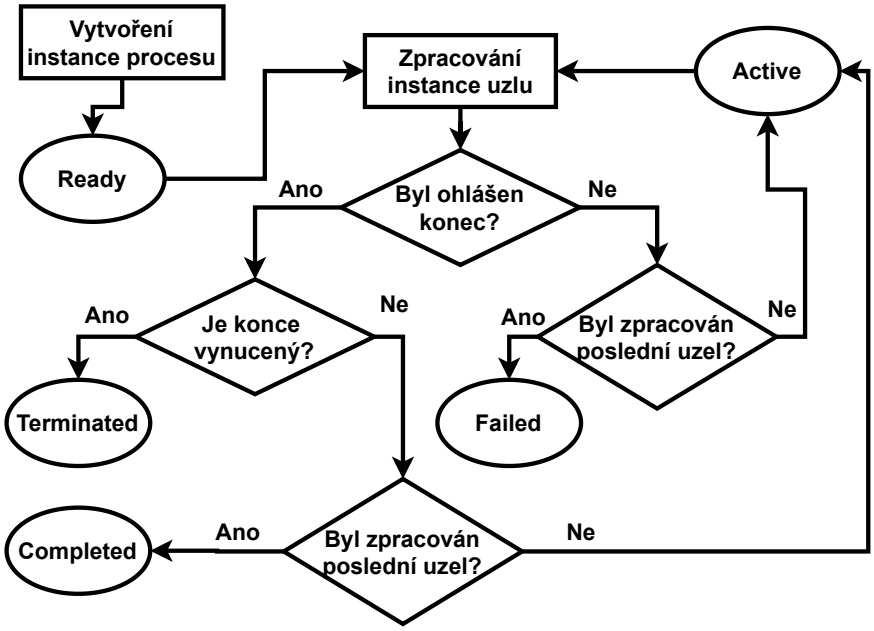
Bohužel systém MWE nepodporuje zaměňování procesů za uzly a striktně je odděluje. Navíc běhové jádro procesů systému MWE je navrženo tak aby nebylo vázáno jen na specifikaci BPMN, ale aby mohlo býti obohaceno do budoucna i o jiné typy uzlů. Proto se v systému MWE odlišují stavy, do kterých se můžou dostat instance procesů a instance uzlů.

Instance procesu se může dostat do stavů:

- Připravená (Ready)
  - Instance procesu neobsahuje žádnou zpracovanou instanci uzlu.
- Aktivní (Active)
  - Instance procesu obsahující zpracované a zpracovávané instance uzlů.
- Dokončená (Completed)
  - Instance procesu, která obsahuje dokončené uzly a zároveň došlo k přirozenému ukončení procesu.
  - Instance procesu, u které došlo k přirozenému ukončení, tj. byly zpracovány všechny vytvořené instance uzlů a zároveň poslední zpracovávaná instance uzlu oznámila, že po jejím zpracování může dojít k ukončení instance procesu.
- Nezdařená (Failled)
  - Instance procesu nebyla řádně ukončena, a přesto neexistuje žádná další instance uzlu, která by mohla být zpracována.
- Ukončená (Terminated)
  - Instance procesu, u níž došlo k vynucenému ukončení neboli poslední zpracovávaná instance uzlu oznámila násilné ukončení instance procesu.
- Stažená (Withdrawn)
  - Jedná se o peciální stav, ke kterému dochází jen při odvolání/stažení instance procesu vlivem externí žádosti na systém.

Na obrázku 5.5 je vyobrazen rozhodovací strom instance procesu, který ukazuje způsob rozhodování o následujícím stavu instance procesu po zpracování instance uzlu. Instance procesu po svém vytvoření je ve stavu připraveno (Ready) a až po zpracování první instance uzlu obsažené v instanci procesu je stav instance procesu změněn. Při změně stavu se rozhoduje na základě dat, která jsou výsledky získanými po zpracování instance uzlu.

Běhové jádro procesů se rozhoduje o následujícím stavu instance procesu z informací nasbíraných službou pro ukončení procesu a počtem nedokončených instancí uzlů v procesu. V prvním kroku běhové jádro procesů zjistí, zda právě zpracovaná instance uzlu oznámila ukončení instance procesu prostřednictvím služby pro ukončení procesu. Běhové jádro procesů rozeznává momentálně dvě možná ukončení, o která mohou žádat instance uzlů. První z nich je vynucené neboli násilné ukončení instance procesu a druhým je běžné neboli přirozené ukončení instance procesu. V případě vynuceného ukončení instance procesu bude nejen změněn stav instance procesu na ukončený, ale i všechny její instance uzlů, které ještě nebyli dokončeny, přejdou do stavu staženo, aby systém rozpoznal, že již instance uzlů nemá dále obsluhovat. Běžné neboli přirozené ukončení instance procesu slouží pro pouhé oznámení o možném konci procesu a běhovému jádru procesů je předána povinnost rozhodnout o ukončení instance procesu. Druhé kritérium na základě, kterého běhové jádro rozhoduje o výsledném stavu instance procesu je počet nedokončených instancí uzlů. Pokud zpracovávaná instance uzlu nebyla poslední instancí uzlu v instanci procesu, tak se instance procesu



Obrázek 5.5: Diagram ukazuje rozhodovací postup, podle kterého je vybírán stav instance procesu po každém zpracování jeho instance uzlu. (Diagram byl vytvořen za využití nástroje draw.net.)

dostává do stavu aktivní. Když ale již neexistuje žádná další instance uzlu, která by mohla býti zpracována, tak je instance procesu ukončena. V případě, že poslední zpracovávaný uzel oznámil možný konec, je instanci procesu přidělen stav dokončená. V opačném případě se instance procesu dostává do stavu nezdařená.

Běhové jádro procesu na úplném konci řetězce zpracování instance uzlu uloží veškerá změněná data a i nově vytvořená data do databáze.

5.2.3 Zpracování uzlů

Běhové jádro procesů kromě připravování dat, vyhodnocování výsledků a ukládání změn musí předně nějakým způsobem získat výsledky k vyhodnocení. Výsledky k vyhodnocení pro následné řízení podnikových procesů jsou kombinací nasbíraných informací pomocí zásuvných modulů s implementacemi služeb a obsahu kontextu po volání funkcí zásuvného modulu s implementací uzlu, který byl vybrán pro obsloužení právě zpracovávané instance uzlu.

Funkce pro zpracování uzlu se nenachází přímo v souboru s běhovým jádrem procesů, ale jsou umístěny ve adresáři `src/bpmnRunner/` v souboru `executeHelpers.ts`.

Během procesu zpracování v běhovém jádře procesů projde instance uzlu několika etapami, během kterých několikrát změní stav, ve kterém se při nich nachází.

Instance uzlu se během své existence může dostat do stavů:

- Čekající (Waiting)
- Připravená (Ready)
- Aktivní (Active)

- Dokončování (Completing)
- Dokončená (Completed)
- Padající (Falling)
- Nezdařená (Failled)
- Stažená (Withdrawn)

Ale během běžného procesu zpracování v běhovém jádře procesů se k některým stavům nedostane. Při běžném procesu zpracování instance uzlu hraje hlavní roli nalezená implementace uzlu, která se má postarat o obsloužení instance uzlu. Funkce, které se mohou vyskytovat v implementaci uzlu, odpovídají etapám, kterými při provádění instance uzlu projde. Ale existuje stav, na který implementace uzlu nemá žádný vliv. Tím stavem je stav stažená. Jedná se o speciální stav, do kterého se může instance uzlu dostat. Ve stavu stažená se vyskytne instance uzlu jen ve chvíli, když běhové jádro procesů rozhodlo o ukončení instance procesu. Všechny ještě nedokončené instance uzlů v instanci procesu poté přechází do stavu stažená.

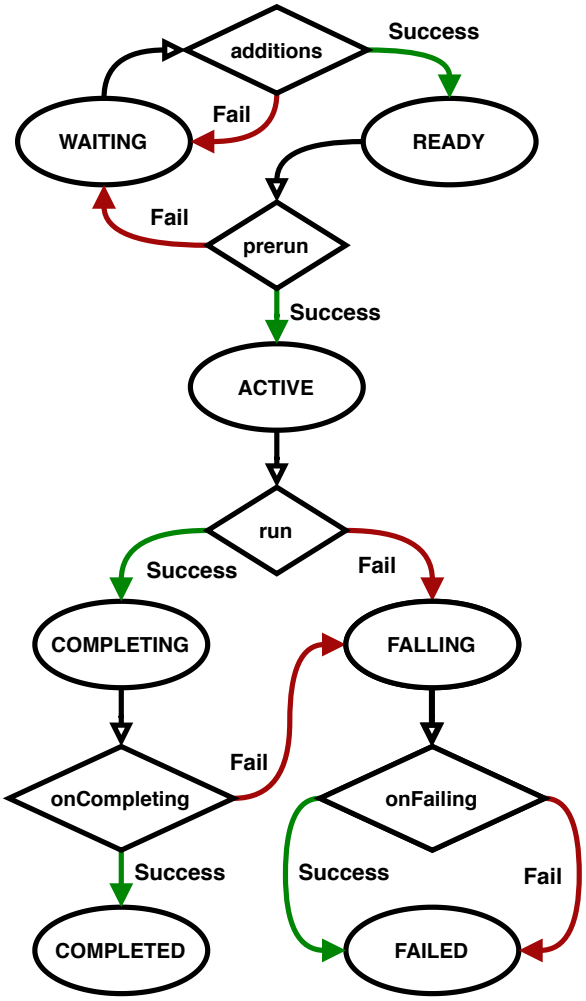
Obrázek 5.6 obsahuje diagram znázorňující závislosti funkcí poskytovaných implementací uzlu na stavy instance uzlu. Při běžném zpracování instance uzlu je instance uzlu na počátku ve stavu připravená a může projít přes etapy:

- Předvyhodnocení
  - Odpovídá funkci implementace uzlu `prerun`.
- Hlavní běh
  - Odpovídá funkci implementace uzlu `run`.
- Dokončování
  - Odpovídá funkci implementace uzlu `onCompleting`.
- Pád
  - Odpovídá funkci implementace uzlu `onFailing`.

Výchozím stavem, ve kterém se instance procesu dostává do systému po jejím vytvoření, je stav připravená. Ve stavu připravená se nachází instance uzlů, které jsou připraveny k pokusu o jejich zpracování.

Při běžném zpracování připravená instance uzlu projde etapou předvyhodnocení. Během předvyhodnocení implementace uzlu má za úkol otestovat, jestli jsou dostupná veškerá potřebná data a zároveň jsou splněny všechny podmínky pro korektní hlavní běh implementace uzlu. Pokud vyhodnocovací funkce selže, tak se instance uzlu dostává do stavu čekající a čeká, než ji probudí jiná instance uzlu či externí akce, která změní podmínky pro zpracování instance uzlu. Po úspěšném dokončení etapy předvyhodnocení přichází etapa hlavní běh. Implementace uzlu při ní provede svojí hlavní činnost, po jejímž provedení lze o instanci uzlu prohlásit za obslouženou.

Některé implementace uzlů mohou chtít rozlišovat svoji činnost na hlavní a vedlejší. Vedlejší činnost je možné vykonat ve fázi dokončování. Příkladem zde může být úloha



Obrázek 5.6: Diagram životního cyklu instance uzlu při jeho zpracování, který ukazuje závislosti mezi stavy instance uzlu a funkcemi zásuvného modulu s implementací uzlu. (Diagram byl vytvořen za využití nástroje draw.net.)

ScriptTask ze specifikace BPMN. Její hlavní činností je provést zadaný skript a její vedlejší úlohou je vybrat, jaké uzly mají pokračovat po jejím dokončení.

Při havním běhu nebo při dokončování může dojít k chybě a dojde k pádu implementace. V takovémto případě se instance uzlu dostává při svém zpracování do etapy pád. Etapa pád má sloužit pro reakci na selhání implementace uzlu. Původně byla zamýšlena pro využití v implementacích, které umožňují alternativní sekvenční tok v případě chyby. Bohužel v současné verzi systému MWE není plně způsobilá pro tuto činnost a jedná se spíše o experimentální záležitost.

Mimo běžné zpracování instance uzlu může dojít ještě ke dvěma různým průběhům při práci běhového jádra procesů s instancí uzlu. Jedním již zmíněným je manipulace s instancí uzlu v případě ukončování instance procesu. Druhý průběh slouží pro doplnění potřebných dat k provedení implementace uzlu.

Tento průběh se skládá jen z jedné etapy:

- Doplnění dodatků

  - Odpovídá funkci implementace uzlu `additions`.

Během etapy doplnění dodatků jsou implementaci uzlu přidána doplňková data, která mohla vzniknou vlivem změny systému nebo externí akcí. Od implementace se očekává, že vyhodnotí obdržená data a uloží je do lokálního registru dat. Instance uzlu, která úspěšně projde doplněním dodatků, se dostává do stavu připravená.

5.2.4 Složení kontextu

S kontextem pracují zásuvné moduly obsahující implementací uzlů, jejichž popis je k nalezení v podkapitole 5.2.6. Při zpracování instancí uzlů nemají zásuvné moduly s implementacemi k instanci uzlu přímí přístup. Přímí přístup k instanci uzlu anebo i k jiným objektům, kterými můžou být instance procesu, vstupní a výstupní datové objekty nebo šablony uzlů, je nebezpečný hned ze dvou důvodů.

- Bezpečnost
  - Není žádoucí, aby zásuvný modul mohl manipulovat se všemi daty.
  - Některá data nenesou žádnou informační hodnotu na základě, které by zásuvný modul měl manipulovat se systémem MWE.
  - Je vhodné zvolit bezpečné hranice mezi běhovým jádrem a zásuvnými moduly.
- Jednotný formát dat
  - Interní reprezentace objektů v systému MWE se mohou časem měnit a odlišovat se od původního návrhu.
  - Aby nebylo nutné v takovém případě všechny zásuvné moduly upravovat s každou větší změnou, je přívětivější zvolit jednotný prostředek nesoucí chtěná data.
  - Jednotný formát dat sebou nese i příznivější názvy pro použití při vývoji či při propagaci dále (Například v podmíněných sekvenčních tocích může být umožněno využívat ve výrazu přímo data z kontextu.).
- Odvolání změn
  - V zásuvných modulech nebo i přímo v systému MWE může dojít k selhání.
  - Jak selhání úmyslné, tak i neúmyslné selhání zásuvného modulu nesmí ohrozit již existující data.
  - Poškození dat v sestrojeném kontextu není problémem, neboť kontext je sestaven individuálně pro instanci uzlu a běhové jádro postupuje ve zpracování dat z kontextu až po vyhodnocení stavu zpracování.

V adresáři `src/bpmnRunner/` v souboru `runContext.ts` lze nalézt datové typy, které definují strukturu kontextu a dat v něm obsažených. Zároveň se zde nachází funkce pro jeho složení.

Obsah kontextu a význam jednotlivých položek:

- `$GLOBAL`
  - Data uložená v registru dat v rámci instance procesu.

  - K těmto datům mají přístup všechny instance uzlů patřící pod stejnou instanci procesu.
  - Data jsou určená jen ke čtení.
  - Zápis dat do globálního registru dat je možný jen skrze funkce poskytující tuto službu.

- `$LOCAL`
  - Data uložená v registru dat v rámci instance uzlu.
  - K těmto datům má přístup pouze instance uzlu, které daný lokální registr dat patří.
  - Data jsou určená jen ke čtení.
  - Zápis dat do lokálního registru dat je možný jen skrze funkce poskytující tuto službu.

- `$INCOMING`
  - Obsahuje informace o příchozích sekvenčních tocích.
  - Obsah je určen jen ke čtení a změny v nich se neprojeví v systému.
  - Jedná se o pole objektů.
  - `id`
    * Identifikátor šablony sekvenčního toku
  - `came`
    * Příznak, zda přes tento sekvenční tok došlo k přechodu na instanci uzlu.
  - `flag`
    * Pomocné označení pro rozpoznaní speciálních sekvenčních toků.
    * Např. pro rozpoznání výchozích sekvenčních toků u elementů typu brána.

- `$OUTGOING`
  - Obsahuje informace o odchozích sekvenčních tocích.
  - Obsah je určen jen ke čtení a změny v nich se neprojeví v systému.
  - Jedná se o pole objektů.
  - `id`
    * Identifikátor šablony uzlu, ke kterému vede sekvenční tok.
  - `expression`
    * Jedná se o logický výraz.
    * Může být použit při rozhodování o budoucím sekvenčním toku.
  - `flag`
    * Pomocné označení pro rozpoznaní speciálních sekvenčních toků.
    * Např. pro rozpoznání výchozích sekvenčních toků u elementů typu brána.

- `$INPUT`
  - Obsahem jsou objekty představující datové objekty nesoucí vstupní data.

  - K objektům je možné přistoupit prostřednictvím názvu datového objektu.
  - Např. Datový objekt s názvem 'Vaše Pošta' je dostupný v `$INPUT['Vaše Pošta']`.
  - Datové objekty a jejich data jsou určena jen ke čtení a změna v nich se neprojeví v systému.

- `$OUTPUT`
  - Obsahem jsou objekty představující datové objekty nesoucí výstupní data.
  - K objektům je možné přistupovat prostřednictvím názvu datového objektu.
  - Pozor! Změny provedené ve zde přítomných datových objektech se projeví v systému.
  - Po dokončení instance uzlu jsou data obsažená v datových objektech uložena.

- `$SELF`
  - Část informací o aktuální instanci uzlu a k ní patřící šabloně uzlu.
  - `id`
    - Identifikátor instance uzlu.
  - `startDateTime`
    - Datum a čas vytvoření instance uzlu.
  - `endDateTime`
    - Datum a čas ukončení instance uzlu.
  - `name`
    - Název šablony uzlu.
  - `bpmnId`
    - Identifikátor šablony uzlu, který byl načten ze souboru BPMN.
  - `implementation`
    - Název implementace, která obstarává zpracování uzlu.

- `$NODES`
  - Obsahuje seznam šablon uzlů, o který si požádala implementace zpracovávající instanci uzlu.
  - `id`
    - Identifikátor šablony uzlu.
  - `bpmnId`
    - Identifikátor šablony uzlu, který byl načten ze souboru BPMN.
  - `name`
    - Název šablony uzlu.
  - `implementation`
    - Název implementace, která bude obstarávat zpracování uzlu.
  - `data`
    - Výchozí data, která budou obsahem lokálního registru instancí vycházejících ze šablony uzlu.

5.2.5 Zásuvné moduly s implementacemi služeb

Systém zásuvných modulů je rozdělen do dvou částí, z níž jedou z nich jsou zásuvné moduly obsahující implementace služeb a druhou částí jsou zásuvné moduly obsahující implementace uzlů. Druhá část je rozebírána v podkapitole 5.2.6. Ve složce `src/bpmnRunner/` v souboru `pluginsImplementation.ts` se nachází rozhraní `ServiceImplementation`, které pevně definuje minimální požadavky běhového jádra na zásuvný modul poskytující službu.

Zásuvné moduly s implementací služeb dodávají do běhového jádra procesu prostředky a chování, které mohou následně využívat moduly s implementacemi uzlů. Tyto moduly jsou v základu velmi jednoduché. Jedná se o objekty či třídy, které musí obsahovat funkci, která vyková nějakou službu, a název pod kterým bude daná funkce dostupná v zásuvných modulech s implementacemi uzlů.

5.2.6 Zásuvné moduly s implementacemi uzlů

Systém zásuvných modulů je rozdělen do dvou částí. První částí se zabývá podkapitola 5.2.5. Druhou část systému zásuvných modulů tvoří zásuvné moduly obsahující implementaci uzlů. Ve složce `src/bpmnRunner/` v souboru `pluginsImplementation.ts` se nachází rozhraní `NodeImplementation`.

Oproti velice jednoduché stavbě zásuvného modulu s implementací služby, je podoba zásuvného modulu s implementací uzlu, kterou definuje rozhraní mnohem více komplexnější.

První volitelnou položkou, kterou může zásuvný modul obsahovat, jsou nastavení (`options`). Nastaveni ovlivňuje chování běhového jádra procesu při načítání a přípravě dat či při zpracovávání výsledků po zpracování instance uzlu.

V nastaveních je možné změnit následující vlastnosti:

- Rozsah požadovaných vstupů `scope_inputs`
  - Ovlivňuje, jaké datové objekty budou připraveny pro provedení implementaci uzlu.
  - Lokální rozsah `local`
    - Je výchozím rozsahem.
    - Implementaci uzlu jsou připraveny data jen z vstupních datových objektů, které jsou přiřazeny k uzlu pomocí vstupních datových toků.
  - Globální rozsah `global`
    - Implementaci uzlu jsou připraveny data ze všech datových objektů, které jsou v procesu aktuálně dostupné.
- Rozsah požadovaných výstupů `scope_outputs`
  - Ovlivňuje výběr datových objektů, do kterých je možné uložit data upravené implementací uzlu.
  - Lokální rozsah `local`
    - Je výchozím rozsahem.
    - Umožňuje data uložit jen do výstupních datových objektů, které jsou přiřazeny k uzlu pomocí výstupních datových toků.
  - Globální rozsah `global`

    * Umožňuje data uložit do libovolného datového objektu, který se v procesu nachází.

- Maximální počet opakovaní uzlu `max_count_recurrence_node`
  - Z bezpečnostních důvodů systém omezuje počet instancí uzlu, které jsou vytvořeny z jedné šablony uzlu v rámci jedné instance procesu.
  - Nastavením jiné číselné hodnoty je ovlivněn maximální počet opakovaného vytvoření instance uzlu.
  - Ve výchozím stavu běhové jádro toleruje 10krát opakované vytvoření instance z jedné šablony uzlu.
  - Není doporučené používat příliš vysoké hodnoty a obcházet tak zabezpečovací systém.
- Poskytnutí informací o uzlech `provideNodes`
  - Jedná se o funkci, která vrací pravdivostní hodnotu.
  - Slouží k získání základních informací o jiných uzlech v šabloně procesu.
  - Funkce je určena k vyfiltrování jen chtěných uzlů.

Dalšími položkami jsou funkce, které obstarávají různé zpracování instance uzlu při různých stavech, ve kterých se může instance uzlu nacházet při zpracování.

Na obrázku 5.7, který ukazuje životní cyklus instance uzlu, jsou přechody mezi jednotlivými stavy ovlivněny právě těmito funkcemi. Jejich úspěšné či neúspěšné dokončení rozhoduje o zvoleném přechodu z jednoho stavu instance uzlu na jiný stav. Životní cyklus instance uzlu je popsán v podkapitole 5.2.3.

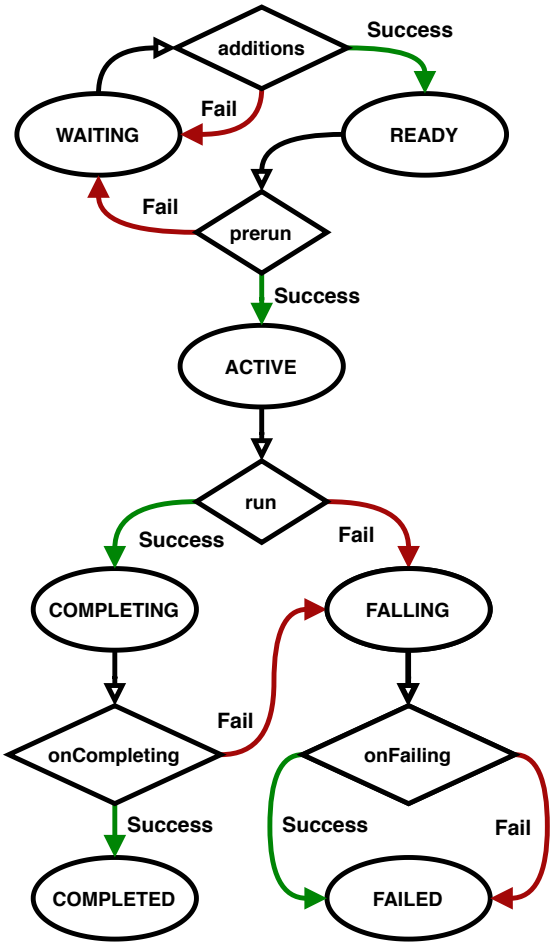
Aby zásuvný modul s implementací uzlu byl schopný vykonávat nějaké rozumné úkoly, tak při volání jeho funkcí obsahující implementaci pro obsluhu mu jsou běhovým jádrem předány pro něj připravené prostředky a informace. Každá funkce dostane data označovaná jako kontext (`context`) a balíček funkcí (`fn`). Popisu kontextu a o datech, která jsou kontext obsahuje či může obsahovat, se věnuje kapitola 5.2.4.

Balíček funkcí obsahuje funkce určené pro poskytnutí služeb. Obsah funkcí v balíčku je závislý na zásuvných modulech s implementací služeb, které používá běhové jádro procesů. Název funkce v balíčku odpovídá názvu, který definuje zásuvný modul s implementací služby. Při použití funkcí z balíčku je potřeba chápat rozdíl mezi běžnou funkcí a funkcí se službou.

U funkcí se službou nedochází k okamžitému volání funkce ze zásuvných modulů s implementacemi služeb, ale běhové jádro si pouze poznamená, že má být služba zavolána a s jakými parametry. K samotnému provedení funkcí se službami dojde až během vyhodnocování výsledků zpracované instance uzlu. Více informací o práci běhového jádra je k nalezení v podkapitole 5.2.2.

Jednou z funkcí je volitelná funkce pro doplňování dodatečných dat do uzlu (*additions*). V některých případech nemusí instance uzlu obsahovat všechna potřebná data pro dokončení instance uzlu a čeká na jejich doplnění. Doplnění dodatečných dat neboli dodatků do instance uzlu je prováděno běhovým jádrem prostřednictvím této funkce.

Druhou volitelnou funkcí, která také hraje svoji roli při práci s dodatky, je funkce pro získání informací, jaká data mají být doplněna do instance uzlu (*additionsFormat*). Tato funkce se odlišuje od ostatních, neboť neovlivňuje stav instance uzlu, ale vývojář zásuvného modulu za pomocí funkce může informovat o potřebách, které pro úspěšné zpracování



Obrázek 5.7: Diagram životního cyklu instance uzlu při jeho zpracování, který ukazuje závislosti mezi stavy instance uzlu a funkcemi zásuvného modulu s implementací uzlu. (Diagram byl vytvořen za využití nástroje draw.net.)

je nutné splnit. Návratová hodnotou funkce je objekt, jehož názvy položek jsou zároveň názvy požadovaných dodatků. Hodnotou položek jsou objekty s přesně danou strukturou pro popis formulářového prvku.

Struktura obsahuje následující vlastnosti:

- Typ formulářového prvku `type`
- Výchozí hodnotu formulářového prvku `default`
- Pole obsahující možné hodnoty `possibilities`
- Textovou nápovědu, radu či popis významu dodatku `hints`

Jedinou povinnou vlastností je typ formulářového prvku. Ostatní vlastnosti jsou volitelné. Tato struktura umožňuje generování formulářů pro vyplnění dodatků, které zjednoduší doplňování dodatků potřebných pro zpracování instance uzlu.

Další volitelnou funkcí je funkce pro přípravu před samotným hlavním zpracováním instance uzlu (*prerun*). Její úspěšné zpracování značí, že instance uzlu má všechny potřebné

údaje potřebné pro dokončení. V opačném případě funkce vyhodí chybu a běhové jádro rozhodne, že uzel bude čekat, dokud nedojde k doplnění potřebných údajů. Hlavní účel této funkce, pro který byla zamýšlena je právě kontrola dat a údajů, bez kterých by nebylo možné označit instanci uzlu za dokončenou.

Jedinou povinnou funkci je funkce pro hlavní zpracování instance procesu (*run*). V těle této funkce by se měla vyskytovat hlavní část implementace zásuvného modulu. Po dokončení této funkce lze zpracovávanou instanci uzlu prohlásit za zpracovanou. Úspěšné dokončení této funkce rozhoduje o úspěšném či neúspěšném zpracování instance uzlu. Po úspěšném provedení funkce přechází instance uzlu do stavu dokončování, ale při neúspěšném provedení funkce dochází u instance uzlu k pádu.

Implementace zásuvného modulu může obsloužit výsledné stavy po úspěšném i neúspěšném zpracování instance uzlu. Pro obsluhu úspěšného zpracování lze využít funkci pro dokončení (*onCompleting*). Účelem, pro který byla funkce zamýšlena, je vykonat nějaké dokončovací akce, které přímo nesouvisí s hlavní náplní práce zásuvného modulu, ale stále je nezbytné je vykonat *(Například výběr dalších uzlů pro aktivaci, uložení informací o provedení aj.)*. Funkce nemusí být vykonána úspěšné. V takovém případě dojde k pádu a instance uzlu je prohlášena za neúspěšně zpracovanou.

Pro obsluhu neúspěšně zpracované instance je možné zásuvný modul obohatit o funkci určenou právě pro tento případ (*onFailing*). Použití této funkce je spíše okrajovou záležitostí. Funkce byla zamýšlena k možnému použití v souvislosti s elementem události chyby (*ErrorEventDefinition*) ze specifikace BPMN, kdy nemusí pád obsluhy instance znamenat pád celého procesu.

5.3 Server pro běh zpracování uzlů

Jedna ze dvou klíčových částí, která tvoří server MWE a zároveň je i nedílnou součástí systému MWE, je server pro běh zpracování uzlů. Druhou klíčovou částí serveru MWE tvoří webový server s GraphQL, jehož popisem se zaobírá kapitola 5.4.

Hlavní běhové jádro procesů slouží jen pro jednorázové úkony, jakým je například zpracovaní instance uzlu, ale o opakované zpracovávání a řízení výběru instancí uzlů se stará server pro běh zpracování uzlů. Server během své činnosti využívá služeb, které poskytuje hlavní běhové jádro procesů. Server zodpovídá za plánování zpracování instancí uzlů, výběr připravených instancí uzlů ke zpracování a filtrování fronty, jenž obsahuje připravené intence uzlů ke zpracování.

Zdrojové kódy tvořící server pro běh zpracování uzlů se nachází v adresáři `src/runnerServer/` a hlavní třída `RunnerServer` představující server se nalézá v souboru `runnerServer.ts`. Funkce pro jeho samotné vytvoření, konfiguraci a spuštění jsou obsaženy v souboru `server.ts` v adresáři `src/`.

Klíčovými částmi serveru je fronta instancí uzlů a hlavní vykonávací smyčka zpracování uzlů. V současném stavu je využívána v serveru fronta typu FIFO. Fronta je určena pro instance uzlů, které jsou připraveny k provádění. Při vytváření serveru je očekáváno vložení fronty s výchozím seznamem instancí uzlů připravených ke zpracování, neboť server osobně nepřistupuje přímou cestou k datům v databázi. Připojení k databázi server pouze předává do hlavního běhového jádra procesů, které si skrze něj načítá a ukládá data. Více o fungování běhového jádra procesů je k nalezení v kapitole 5.2.

Do fronty server zasahuje při výběru další instance uzlu pro zpracování anebo při plánování instancí uzlů. K plánování instancí uzlů je využíván primitivní algoritmus, na jehož vstupu je seznam instanci uzlů, u kterých došlo k nějakým změnám. Algoritmus rozdělí

instance uzlů dle jejich stavů na připravené ke zpracovaní a nepřipravené. Za nepřipravené instance uzlů jsou považovány všechny instance uzlů, jejichž stav nemá hodnotu připraveno. Na základě rozdělení instancí uzlů na skupiny jsou následně z fronty odstraněny naplánované instance uzlů, které spadají do skupiny nepřipravených instancí uzlů, a do fronty jsou přidány instance uzlů ze skupiny připravených instancí uzlů.

Vykonávací smyčka zpracování uzlů plni hlavni úlohu, kterou je opakované zpracování instancí uzlů pomocí běhového jádra procesů. Ve skutečnosti se v aktuální verzi serveru nejedná o jednu smyčku, ale o dvojici vnořených cyklů (vnější a vnitřní cyklus). Vnější cyklus probíhá, dokud je povoleno provádění neboli dokud je serveru povolen běh. Průběh vnějšího cyklu se skládá ze dvou etap. První etapou je zpracovávání instancí uzlů naplánovaných ve frontě. Druhá etapa, která následuje, je čekání.

První etapa je tvořena *vnitřním cyklem*. Vnitřní cyklus se opakuje, dokud je serveru dovolen běh a zároveň dokud ve frontě existují instance uzlů připravených ke zpracování. Během provádění těla vnitřního cyklu dochází k vyjmutí naplánované instance uzlu a k jejímu zpracování prostřednictvím běhového jádra procesů. Na základě výsledků po zpracování instance uzlu dojde v prvé řadě na přeplánování instancí uzlů ve frontě a poté také k volání funkcí pro zpětné volání. Funkce pro zpětné volání je možné serveru předat při vytvoření a jsou určeny k vnějším reakcím na změny týkající se instancí uzlů a instancí procesů po zpracování naplánované instance uzlu.

Druhá etapa vnějšího cyklu, jenž nese označení *čekání*, slouží pro odstranění aktivního čekaní. Aktivní čekání je situace, kdy je opakovaně prováděno vyhodnocení podmínky cyklu a mezi jednotlivými vyhodnoceními podmínky neprobíhají žádné účelné operace, dokud nedojde k jejímu splnění. Ve většině existujících interpretů jazyka JavaScript výskyt aktivního čekání znamená zablokování smyčky událostí. Stručné vysvětlení smyčky událostí v Node.js a odkazy na vysvětlující články lze nalézt zkraje kapitoly 5.5. K přerušení aktivního čekání je využito mechanismu slibů (*Promise*) v jazyce JavaScript. Při zahájení etapy čekání dojde k vytvoření slibu, který po uplynutí nastaveného času přejde ze stavu čekající (*pending*) do stavu vyřešeno (*resolved*) či odmítnuto (*rejected*). Během čekání na vyřešení slibu nedochází k blokování smyčky událostí v interpretu.

```
function wait(ms) {
       return new Promise((resolve) => setTimeout(() => resolve(), ms))
}
async function foo() {
       await wait(3000)
}
```

Obrázek 5.8: Ukázka kódu v jazyce JavaScript obsahuje funkci pro obalení zpětného volání časovače a vytvoření slibu, který slouží k čekání po určenou dobu.

Zde bych rád upozornil na moje řešení přerušitelného slibu, které je v serveru využíváno. V komunitě vývojářů je známý a používaný vzor pro čekání v asynchronních funkcí. Vzor využívá slibu, který čeká x milisekund, jenž používá funkci `setTimeout` k oznámení vyřešení slibu (viz. následující ukázka kódu 5.8).

Bohužel jsem nebyl schopen nalézt žádné rozumné řešení přerušitelných slibů, které by bylo jednoduché a zároveň nemělo problém s možnými úniky paměti (*memory leaks*). V případě serveru pro běh zpracování uzlů není nezbytné čekat vždy konstantní dobu mezi kontrolami neprázdnosti fronty, ale hodí se mechanizmus pro přerušení čekání. Přerušení čekání

se hodí v případě, že dojde k vložení/naplánování instancí uzlů do fronty a fronta tím pádem již není prázdná.

```
function wait(ms) {
	let interruptionSignal = () => { }
	const promise = new Promise((resolve, reject) => {
		let waitTimeout = setTimeout(() => resolve(), ms)
		interruptionSignal = () => {
			clearTimeout(waitTimeout)
			reject('Interrupted')
		}
	})
	return { promise, interruptionSignal }
}
async function foo() {
	let { promise, interruptionSignal } = wait(3000)
	// ... Zde muze dojit k~predani funkce pro~preruseni slibu.
	try {
		await promise
		// Probuzeni vlivem vyprseni casu.
	} catch {
		// Probuzeni vlivem preruseni.
	}
}
```

Obrázek 5.9: Ukázka kódu v jazyce JavaScript názorně ukazující funkci pro vytvoření přerušitelného slibu bez úniků paměti.

Ukázka kódu 5.9 obsahuje funkci, jejímž výsledkem je slib, k jehož vyřešení dojde po uplynutí zadané doby a současně vrací funkci pro přerušení slibu. Při volání funkce pro přerušení funkce ukončí naplánované spuštění zpětného volání časovače v slibu a provede jeho odmítnutí. Tímto způsobem vytvořený přerušitelný slib nezanechává ve smyčce událostí v interpretu jazyka JavaScript žádný nedokončený blok kódu, který by zůstával v paměti. Navíc je i možné rozlišit probuzení způsobené přerušením od probuzení způsobené vlivem uplynutí času čekání.

Při popisu serveru pro běh zpracování uzlů v této kapitole se vyskytují narážky, že se jedná o popis současné či aktuální verze serveru. Předchozí verze serveru výrazným způsobem trpěla na úniky paměti (*memory leaks*) a to v řádu stovek MB za den. To činilo předchozí verzi pro dlouhodobý provoz nepoužitelnou.

Na vině byla špatná implementace bloku kódu, který tvoří vykonávací smyčku zpracování uzlů. Implementace obsahovala rekurzi asynchronní funkce. V následující ukázce kódu 5.10 je znázorněna příčina. Asynchronní funkce v jazyce JavaScript je jen jiná forma slibů. V okamžiku rekurze asynchronní funkce dochází k řetězení slibů[25] (Jedná se o kombinaci A a B.). Problémem velmi rychle rostoucího úniku paměti jsou právě data, se kterými se ve funkci pracuje[25]. Funkce uvolňuje své alokované zdroje až po svém skončení, ale v případě zřetězených slibů nedochází k ukončení funkce, dokud neskončí celá rekurze[25].

```
async function foo(x) {
	// ... telo funkce foo
	await foo(x+1)
}
// Je interne prevadeno na~stale bobtnajici objekt slibu:
// A
Promise.resolve()
	// ... dalsi zanoreni
	.then(/* funkce foo s~x = N + 2 */)
	.then(/* funkce foo s~x = N + 1 */)
	.then(/* funkce foo s~x = N + 0 */)
// B
new Promise(resolve => {
	// ... telo funkce foo
	return new Promise(resolve => {
		// ... telo funkce foo
		return new Promise(resolve => {
			// ...
		})
	})
})
```

Obrázek 5.10: Ukázka kódu v jazyce JavaScript ukazující ...

O tomto jevu a jeho opravě jsem se dozvěděl z článku[2][25], jenž je uveřejněn na webu alibabacloud.com.

5.4 Webový server s GraphQL

Server MWE je tvořen dvěma klíčovými částmi. První z nich je server pro běh zpracování uzlů, kterému se věnuje kapitola 5.3, a druhá klíčová část, která tvoří server MWE, je webový server.

Webový server slouží ke komunikace serveru MWE s vnějším světem. Pro vybudování rozhraní pro komunikaci webového serveru je možné v současné době volit mezi dvěma populárními způsoby. První z nich je vytvoření aplikace s architekturou REST, která jako své rozhraní využívá protokolu HTTP v plném jeho rozsahu. Druhou možností je použít relativně mladou technologii z dílny Facebooku zvanou GraphQL. Webový server vytvořený pro server MWE využívá ke komunikaci s klienty právě GraphQL.

Zdrojové kódy webového serveru jsou rozděleny do několika míst v repositáři MWEServer. Soubor obsahující funkce pro jeho samotné vytvoření a spuštění jsou obsaženy v adresáři `src/` v souboru `server.ts`‘. Dále ve stejném adresáři se také nachází soubor `config.ts`, jehož obsahem jsou základní nastavení webového serveru s GraphQL.

Tato nastavení byla oddělena do samostatného souboru z důvodu ulehčit a zpřehlednit konfiguraci serveru při jeho nasazení do provozu na cílovém zařízení *(Cílovým zařízením je myšlen fyzický či virtuální server, na kterém bude serverová aplikace provozována.)*.

[2]Článek o úniku paměti v Node.js https://www.alibabacloud.com/blog/solving-memory-leaks-caused-by-co-and-recursive-calls_595121

Soubory, které přímo tvoří rozhraní GraphQL se nacházejí v adresáři `src/graphql/`. Struktura obsahu tohoto adresáře je rozdělena do několika logických celků:

- Soubory obsahující definice schématu GraphQL (tj. `src/graphql/typeDefs/*.graphql`)
- Soubory obsahující řešitele GraphQL (tj. `src/graphql/resolvers/*.ts`)
- Soubor, který definuje kontext a vytváří funkci pro jeho získání (tj. `src/graphql/context.ts`)
- Soubory s vygenerovanými datovými typy jazyka TypeScipt
  - Složka `src/graphql/generated/` spolu se souborem `types.ts` je vygenerována až před sestavením aplikace nebo je možné je nechat vygenerovat za pomocí jednoho ze skriptů v souboru `package.json`.
  - Není potřebné, aby soubory byly součástí repositáře, jelikož se jedná o generované soubory, které jsou vytvořené nástrojem *graphql-codegen* na základě obsahu souborů, jenž obsahují definice schématu GraphQL (tj. soubory `src/graphql/typeDefs/*.graphq`
- Soubor s konstantami pro jednoznačnou identifikaci kanálů pro odbebírání dat ze serveru (tj. `src/graphql/subscriptionChanel.ts`).
- Soubor, jehož obsahem je funkce pro nastavení komunikace mezi hlavním a vedlejším procesem.
  - Jedná se o soubor `workerSetup.ts`.
  - Jeho význam bude vysvětlen v kapitole 5.5, která se zabývá spolupráci webového serveru a serveru pro běh zpracování uzlů.

Pro vybudování webového serveru s GraphQL je použit balíček *graphql-yoga*[3], který sdružuje sadu balíčků potřebných pro vytvoření serveru GraphQL v prostředí Node.js a jeho vývoj. Balíček poskytuje třídu `GraphQLServer` pro vytvoření vlastního webového serveru s GraphQL. Při vytváření serveru GraphQL je nutné ještě dodat nezbytné části k jeho provozu.

První z nich je kontext nebo funkci pro získání kontextu. Dále je potřebné dodat sadu funkcí nazývanou souhrnně jako řešitelé. A na závěr poslední část, kterou je schéma GraphQL, jenž obsahuje definice serverem poskytovaných datových typů. Těmto třem částem se věnují blíže následující trojice podkapitol.

Schéma GraphQL

Schéma GraphQL definuje všechny datové typy, které server MWE prostřednictvím rozhraní GraphQL může poskytovat. Všechny soubory, z nichž je schéma tvořeno, se nachází v adresáři `src/graphql/typeDefs/`. Schéma je rozděleno do několika menších souborů, které jsou složeny do jednoho schématu pomocí balíčku *graphql-import*[4]. Balíček *graphql-import* dodává do jazyka pro definování schémata GraphQL konstrukce, jenž právě umožňují rozdělit schéma do více souborů.

Jelikož schéma GraphQL a definice v něm obsažené tvoří předpis pro strukturu dotazů, který může klient vytvořit, tak je vhodné zde vložit k jednotlivým definicím i dokumentační komentáře. Tyto dokumentační komentáře jsou následně k dispozici vývojářům klientských

[3] graphql-yoga https://www.npmjs.com/package/graphql-yoga
[4] graphql-import https://www.npmjs.com/package/graphql-import

aplikací v nástrojích pro tvorbu dotazů GraphQL a usnadňují pochopení a práci s rozhraním.

Schéma GraphQL webového serveru systému MWE je tvořeno následujícími soubory:

- `user.graphql`
  - Obsahuje definice datových typů uživatel, dotazy pro získání uživatelů a mutace pro manipulaci s uživateli.
- `group.graphql`
  - Jeho obsahem jsou datové typy skupina, dotazy pro získání skupin, mutace pro manipulaci se skupinami.
- `member.graphql`
  - Je tvořen definicemi datových typů člena/členství a mutacemi pro manipulaci členstvím uživatelů ve skupinách.
- `auth.graphql`
  - Uvnitř souboru jsou k nalezení datové typy pro získání autorizačního tokenu.
- `bpmn.graphql`
  - Jeho součástí jsou datové typy reprezentující šablony procesů, šablony uzlů, instance procesů, instance uzlů nebo datové objekty, které tvoří podnikové procesy v systému MWE.
  - Dále se zde nachází definovaná řada dotazů, kterými je možné data získat, mutace, jenž umožňují provádět širokou škálu akcí, a taktéž jsou zde definovány odběry, k nimž se může klient přihlásit.
- `schema.graphql`
  - Jde o kořenový soubor schématu GraphQL, který je vstupním souborem pro nástroj z balíčku *graphql-import*.

Webový server systému MWE je vyvíjen v jazyce TypeScript. S datovými typy, které jsou definovány v rámci schématu GraphQL, není možné přímo používat v kódech napsaných v jazyce TypeScript. V komunitě utvořené kolem jazyku TypeScript vzniklo několik různých způsobů pro práci s GraphQL.

Při vývoji webového serveru systému MWE byl zvolen způsob, při kterém jsou ze souboru obsahující schéma GraphQL napsaném v jazyce schématu GraphQL vygenerovány odpovídající datové typy pro použití v jazyce TypeScript. Pro generování datových typů jazyka TypeScript je používán nástroj obsažený v balíčku *graphql-codegen*[5].

Tento nástroj se řídí nastaveními, jež jsou obsažená v souboru `.graphqlconfig.yml`. Při spuštění jednoho ze skriptů pro sestavení projektu, který využívá tento nástroj, dojde k vygenerování datových typu jazyka TypeScript do souboru `types.ts` v adresáři `src/graphql/generated/`. Vygenerované datové typy jsou následně využívány pří vývoji řešitelů, kteří tvoří nepostradatelnou součást serveru GraphQL.

Toto zvolené řešení přineslo při vývoji řešitelů jisté problémy, které jsou rozebrány v podkapitole 5.4.

[5] graphql-codegen https://www.npmjs.com/package/graphql-codegen

Kontext v serveru GraphQL

Server vytvořený za pomocí balíčku *graphql-yoga* potřebuje při jeho vytváření znát kontext nebo funkci pro získání kontextu. Při zpracovávání dotazu GraphQL od klienta jsou data k odeslání získána prostřednictvím funkcí, které bývají nazývány jako řešitelé. Aby se řešitelé nemusely složitým způsobem zabývat vytvářením či nalezením objektů potřebných pro získání a filtrování dat *(např. připojení k databázi, objekty poskytující služby aj.)*, tak je možné využít pro tyto účely kontext. Kontext během zpracování dotazu GraphQL je předáván postupně všem řešitelům, kteří dotaz řeší.

Webový server systému MWE nepoužívá statický kontext, ale používá funkci pro získání kontextu, která je dynamicky vygenerovaná za pomoci funkce `generateContextFunction` obsažené v souboru `src/graphql/context.ts`. Funkce pro získání kontextu je prováděna pro každý serverem přijatý dotaz GraphQL od klienta. Aby v této funkci nebylo nutné opakovaně vytvářet objekty, které mohou být sdíleny napříč všemi dotazy GraphQL, tak jsou tyto objekty vytvořeny jen jednou a následně jsou vloženy do vygenerované funkce pro získání kontextu.

Ve funkci pro získání kontextu pouze probíhá autentizace uživatele čili klienta, který dotaz zaslal, a získání dat o něm z databáze. Autentizace uživatele z dotazu na server je realizována funkcí poskytující možnost autentizace pomocí tokenu neboli anglicky *Bearer authentication*. Více o autentizaci uživatele je k nalezení v kapitole 5.6.

Kontext ve webovém serveru systému MWE obsahuje mimo objektů, které poskytují služby pro získání dat nebo vykonání akcí, také objekty obsahující informace o uživateli, jenž dotaz zaslal.

Kontext webového serveru systému MWE se skládá z následujících pěti položek:

- Připojení k databázi (`db`)
- Identifikovaný klient (`client`)
- Hlavní běhové jádro procesů (`runner`)
- Implementace návrhového vzoru *Publisher/Subscriber* pro odběry v GraphQL (`pubsub`)
- Pomocný objekt pro komunikaci mezi pracovními vlákny (`worker`)

Připojení k databázi je realizováno pomocí nástroje pro objektově relační mapování. Konkrétně je využíván balíček *typeorm*[6], se kterým se velice dobře pracuje v rámci projektů psaných v jazyce TypeScript. Skrze připojení k databázi spravovaném balíčkem *typeorm* mohou řešitelé pracovat s entitami odpovídajícím internímu modelu dat aplikace a díky tomu mohou číst a upravovat data v databázi.

Identifikovaný klient je nepovinnou částí kontextu. Nese informace o uživateli, který zaslal dotaz GraphQL, a o jeho členstvích ve skupinách. Na základě dat o identifikovaném klientovi mohou následně řešitelé upravit své chování pro získávání požadovaných dat při zpracovávání dotazu GraphqQL.

Hlavní běhové jádro procesů v kontextu je objektem poskytující své služby. Slouží pro volání externích akcí pro práci se systémem MWE. Umožňuje, aby řešitelé mohly vykonávat akce pro vytváření nových instancí procesů, rušení instancí procesů nebo doplňování dodatků do instancí uzlů. V kapitole 5.2 lze nalézt podrobný popis běhového jádra procesů.

Implementace návrhového vzoru *Publisher/Subscriber* pro odběry v GraphQL je speciálním objektem, jehož účelem je poskytnout služby pro zasílání zpráv s daty klientům,

[6] typeorm https://www.npmjs.com/package/typeorm

kteří se přihlásili k jejich odebírání. Server MWE využívá výchozí implementaci tohoto návrhového vzoru, který využívá *WebSocket* a umožňuje tak vytvářet spojení mezi serverem a klientem pro zasílání zpráv s daty.

Pomocný objekt pro komunikaci mezi pracovními vlákny slouží taktéž pro zasílání zpráv s daty, ale nejedná se o zprávy mezi klientem a serverem nýbrž o zasílání zpráv mezi hlavním a vedlejším pracovním vláknem. Pomocný objekt zprostředkovává komunikaci mezi webovým serverem s GraphQL a serverem pro běh zpracování uzlů za pomocí unifikovaného rozhraní. Podrobnosti o pracovních vláknech se objevují v kapitole 5.5, která se věnuje jejich využívání v systému MWE.

Řešitelé GraphQ

Řešitelé je souhrnné označení pro funkce, které obstarávají mapování a získávání dat při zpracování dotazu GraphQL. V serveru GraphQL vytvořeného prostřednictvím balíčku *graphql-yoga* dostávají funkce/řešitelé při jejich volání čtveřici parametrů.

- Rodič
  - Jde o objekt, jenž byl výsledkem řešení předchozího řešitele, který je přímím předchůdcem aktuálního řešitele, při zpracování dotazu GraphQL.
- Argumenty
  - Parametry předávané klientem v dotazu GraphQL.
  - Jedná se o data, s nimiž řešitel pracuje a na jejichž základě může upravovat své chování.
- Kontext
  - Jedná se o objekt obsahující sdílené objekty, jež jsou přístupné všem řešitelům při zpracování dotazu.
  - Podkapitola 5.4 *Kontext v serveru GraphQL* se zaobírá obsahem a vytvářením kontextu ve webovém serveru systému MWE.
- Informace
  - Obsahuje informace před připravené knihovnou.
  - Tyto informace nehrají žádnou roli v serveru MWE.

Server MWE je vyvíjen v jazyce TypeScript a stejně tak i řešitelé. V závěru podkapitoly 5.4 *Schéma GraphQL* je zmíněno, že při vývoji byl zvolen postup generování datových typů jazyka TypeScript ze souborů obsahující schéma GraphQL. Díky obsahu vygenerovaného souboru s datovými typy jazyka TypeScript může vývojové prostředí v kombinaci s kompilátorem jazyka TypeScript poskytnou vývojáři oporu a kontrolu při vytváření řešitelů.

Řešitelé jsou ke zpracování dotazu GraphQL puštěni až po lexikální, syntaktické a sémantické kontrole, kterou provádí server GraphQL. Proto vně řešitelů není nezbytně nutná existence mechanizmu pro testování správnosti obdržených dat v argumentech. Jinak řečeno vygenerovaný datový typ pro argumenty řešitelů, budou vždy nabývat daného typu při provádění konkrétního řešitele.

Podobně jako je schéma GraphQL rozděleno do několika souborů, tak jsou podobným způsobem rozděleni i řešitelé do více souborů. V projektu jejich rozdělení na soubory odpovídá rozdělení schématu a každý soubor s řešiteli obsahuje všechny potřebné řešitele pro mapování a získávání dat odpovídajících datových typů, které jsou definovány stejnojmenném souboru se schématem GraphQL.

- `user.ts`
  - Obsahuje řešitele pro zpracování částí dotazů, které odpovídají datovým typům definovaných v souboru `user.graphql`.
- `group.ts`
  - Obsahem souboru jsou řešitelé sloužící k provádění částí dotazů, jenž jsou složeny z datových typů obsažených v souboru `group.graphql`.
- `member.ts`
  - Řešitelé uvnitř soboru zpracovávají části dotazů využívající datové typy definované v souboru `member.graphql`.
- `auth.ts`
  - Práci při zpracování dotazu, jenž obsahují datové typy definované v souboru `auth.graphql`, obstarávají řešitelé z tohoto souboru.
- `bpmn.ts`
  - Obsahem souboru jsou řešitelé, kteří zpracovávají části dotazů, jenž jsou složeny z datových typů definovaných v souboru `bpmn.graphql`.
- `index.ts`
  - Dá se hovořit, že soubor odpovídá z části definicím obsaženým v souboru `schema.graphql`.
  - V souboru dochází ke spojení všech řešitelů pro dotazy, mutace, odběry a definované datové typy do jediného objektu, který je možné použít při vytváření serveru GraphQL.

Při bližším pohledu do řešitelů v souborech lze na první pohled spatřit, nejen že struktura všech řešitelů je navzájem dosti podobná, ale navíc i žádný z řešitelů neobsahuje přímo rozhodovací logiku pro získávání a filtrování dat. Převážná většina funkcí řešitelů získává data nepřímou cestou skrze prostředníka. Prostředníkem zde jsou funkce, které tvoří interní rozhraní serveru MWE a udržují veškerou logiku pro získávání a filtrování dat na jediném místě. Díky tomuto aspektu nemusí být systém MWE do budoucna závislí jen na serveru GraphQL, ale může být snáze rozšířen i o jiné formy komunikaci s okolním světem.

Zmíněným funkcím, jenž tvoří interní rozhraní serveru MWE, se věnuje kapitola 5.6.

Zároveň při pohledu na funkce tvořící řešitele je možné si všimnout výskytu specifických komentářů `// @ts-ignore`. Tyto komentáře s daným obsahem slouží k umístění značky, která oznamuje kompilátoru jazyka TypeScript, aby ignoroval neshodu datových typů na následujícím řádku. Vyskytuje se zde totiž problém se zdvojenými datovými typy, které figurují v serveru MWE.

Interně server MWE pracuje s entitami, které jsou definované v souborech umístěnými v adresáři `src/entity/`. S těmito entitami pracuje i interní rozhraní serveru MWE, ale

pro funkce tvořící řešitele jsou využívány vygenerované datové typy jejichž původ se odvíjí od definic datových typů v souborech obsahující schéma GraphQL. I když datové typy do jisté míry představují stejné objekty, tak jsou zde odlišnosti, které typová kontrola kompilátoru jazyka TypeScript netoleruje. Prvé řadě není chtěné, aby server GraphQL poskytoval všechny údaje, které tvoří entity.

Ukázkovým příkladem je zde uživatel, jehož entita v systému nese údaje o jeho přístupovém heslu. Data tvořící heslo sice nejsou uchovávána v prostém textovém tvaru, který by obsahoval přímo heslo, ale i přes tento fakt není vhodné umožnit zveřejnění této položky. Z podobných důvodů je tomu i u dalších položek, které tvoří entity.

Ale nejedná se o jediný důvod, proč je nezbytně nutné obcházet datovou kontrolu kompilátoru jazyka TypeScript. Mimo to že datové objekty vycházející ze schématu GraphQL neobsahují všechny položky, které je možné nalézt v interně používaných entitách, tak mohou obsahovat i některé položky navíc. Zde v systému existuje taktéž vhodný příklad.

Šablona uzlu obsahuje textový výraz v položce `candidateAssignee`, který obsahuje text, jímž je v systému MWE určována skupina, jejíž členové mohou obsloužit či vykonat instance uzlů vycházející z konkrétní šablony uzlu. Server GraphQL mimo pouhého získání výrazu umožňuje i jeho přiřazení ke skupině a klient může v dotazu přistoupit skrze položku `candidateGroup` přímo k odpovídající skupině.

Toto byly dva hlavní problémy stojícími za problémy typové kontroly kompilátoru jazyka TypeScript.

Problém pramení již ze zvoleného způsobu generování datových typů ze schématu GraphQL. Pro budoucí projekty využívající kombinaci jazyka TypeScript a serveru GraphQL je vhodné zauvažovat nad použitím jiného způsobu, který bude spíše cílit primárně na TypeScript a návrh univerzálně použitelných entit v něm, ze kterých bude generováno schéma GraphQL *(Takže opačný postup, než který byl zvolen při vývoji serveru MWE.)*. O prostředky pro tento způsob se snaží projekt *TypeGraphQL*(https://typegraphql.com/).

5.5 Paralelní aplikace v Node.js aneb pracovní vlákna

Název podkapitoly může býti lehce zavádějící, ale vše bude vysvětleno. Při vývoji aplikací pro Node.js se téměř nikdy neuvažuje nad paralelní aplikací, protože jazyk JavaScript ve svém základnu nedisponuje žádnými prostředky k tomu určenými. Jazyk JavaScript a jeho interpret je určen ke tvoření jedno vláknových aplikací. Při zpracovávání kódu aplikace se v jeden okamžik provádí vždy jen jeden úkon. Provádění kódu v interpretu v Node.js je řízeno smyčkou událostí (anglicky *Event Loop*).

Velmi zjednodušeně a ve stručnosti řečeno během hledání připraveného kódu pro provedení interpret nahlíží v cyklu do několika front obsahující funkce zpětného volání. Smyčka událostí je rozdělena do několika fází a každá fáze má právě jednu frontu pro zpětná volání. Pokud interpret nalezne kód připravený ke zpracování, tak jej začne zpracovávat. Až po jeho zpracování hledá další kód, který bude dále proveden. [23][10]

Knihovny, které poskytuje Node.js, umožňují využívat speciální moduly/knihovny, díky kterým je možné například využít časovače pro naplánování zavolání funkce (př. `setTimeout`, `setinterval`) nebo různé neblokující I/O operace (př. `fs`). Celý tento cyklus ještě komplikují tzv. microtasks. [23]

Pro získání podrobnější a mnohem přesnější informaci o fungování smyčky událostí je vhodné se podívat na velmi dobře zpracovaný článek uveřejněný webu IBM[7] anebo stručnější článek publikovaný na webu dev.to[8], který graficky vysvětluje hlavní princip. Předchozí vysvětlení je založeno na‘obsahu právě těchto dvou článků[23][10].

Server MWE potřebuje, aby v jeden okamžik bylo možné obsluhovat požadavky od klientů pomocí webového serveru, který zprostředkovává komunikaci klientů se serverem a plní jejich požadavky, a zároveň musí i probíhat zpracování připravených instancí uzlů, o které se stará server pro běh zpracování uzlů. Pokud by všechny úkony, jenž provádí webový server a server pro běh zpracování uzlů, měli probíhat jen na jediném vlákně, tak by to sebou neslo řadu možných problémů.

Mnoho těchto problémů má přímou souvislost se smyčkou událostí v interpretu v Node.js. Pokud zpracovávaný kód interpretem je výpočetně náročný nebo obsahuje nekonečný cyklus, tak dojde k zablokování smyčky událostí v interpretu. Blokování smyčky událostí v interpretu je překážkou pro plynulý běh programu a má za následek velmi nepříjemný jev.

Dochází při něm k hromadění nevyřízených bloků kódu ve frontách zpětného volání, které jsou vykonávány se zpožděním, anebo v nejhorším případě může dojít k pádu interpretu jazyka JavaScript. Na serveru MWE by to mohlo vést například k následujícímu případu. V momentě, kdyby probíhalo zpracování instance uzlu s náročnější implementací, tak by nemohly být obslouženy požadavky zaslané na server od klientů.

Server MWE pro komunikaci s klientem používá GraphQL, které má již v základu potenciál pro pomalejší odezvu způsobenou nutností lexikální, syntaktické a sémantické analýzy následovanou mapováním dat na abstraktní strom dotazu GraphQL. Klienti by museli čekat příliš dlouho na obsloužení a v horších případech by mohlo docházet i k vypršení času pro zaslání odpovědi na dotaz od klienta.

Velmi podobně by tomu mohlo býti i naopak. Tedy v případě, kdy obsluha požadavků od klientů by blokovala provádění činností pro zpracování instancí uzlů.

Jazyk JavaScript a jeho interpret sice nejsou uzpůsobeny pro vytváření a provoz paralelních aplikací, ale to neznamená, že by se s tím komunita vývojářů smířila. Pro náročné I/O operace, kterými jsou například čtení a zápis do databáze či souborů nebo pro komunikaci přes síť s jinými aplikacemi, jsou v knihovnách Node.js a v současném interpretu nástroje pro jejich neblokující obsloužení (tj. obsloužení neblokující smyčku událostí). Pro obsluhu funkcí, které jsou náročné na výpočetní výkon, lze taktéž najít prostředky v knihovnách Node.js nebo v knihovnách třetích stran, které jsou veřejně dostupné přes správce balíčků (např. NPM, Yarn).

V knihovnách Node.js je možné najít tři balíčky/moduly, které je možné využít pro účely vývoje paralelní aplikace [23].

- child_process[9]
- cluster[10]
- worker_threads[11]

Při vývoji serveru MWE proběhlo několik drobných experimentů, jejichž výsledkem bylo zvolení modulu *worker_threads* pro finální nasazení v serveru MWE. Modul se po-

[7] Článek o smyčce událostí v Node.js https://developer.ibm.com/technologies/node-js/tutorials/learn-nodejs-the-event-loop/

[8] Článek o smyčce událostí https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif

[9] child_process https://nodejs.org/api/child_process.html

[10] cluster https://nodejs.org/api/cluster.html

[11] worker_threads https://nodejs.org/api/worker_threads.html

prvé objevil v knihovnách Node.js 10 jako experimentální modul[15], ale od již od vydání Node.js 12 se jedná o stabilní součást knihovny[16]. Modul *worker_threads* poskytuje prostředky pro práci s více vlákny v rámci aplikace v Node.js. Nejedná se přitom o přímou práci s vlákny.

Před pracovními vlákny bylo v Node.js možné řešit paralelizmus pomocí vytváření nových podřízených procesů. Toto řešení přináší vyšší režijní náklady a zároveň i problém se vzájemnou komunikací mezi procesy.

Pracovní vlákna umožňuji využít více vláken v rámci jednoho procesu (systémového procesu) a zároveň poskytují prostředky pro zasílání zpráv mezi pracovními vlákny. Následující stručná ukázka, kterou tvoří dvojice seznamů, je vypůjčena z článku[12][22], který se zabývá pochopením pracovních vláken v Node.js.

Po spuštění procesu Node.js se běžně vytvoří[22]:

- Jeden proces
- Jedno vlákno
- Jedna smyčka událostí
- Jedna instance motoru jazyka JavaScript (V8)
- Jedna instance Node.js

Po spuštění procesu Node.js, který pracuje s pracovními vlákny, se vytvoří[22]:

- Jeden proces
- Více vláken
- Jedna smyčka událostí na vlákno
- Jedna instance motoru jazyka JavaScript na vlákno (V8)
- Jedna instance Node.js na vlákno

Zapojení pracovních vláken do serveru MWE

Server MWE se skládá ze dvou částí, kterými je dvojice serverů. Pro jejich současný provoz a jejich vzájemnou komunikaci jsou využívány pracovní vlákna a prostředky pro práci s nimi obsažené v modulu `worker_threads`.

Při práci s pracovními vlákny je jedno z vláken hlavním vláknem a v něm jsou vytvářeni pracovníci (tj. instance třídy `Worker`). Nový pracovník při vytváření potřebuje znát buďto kód jazyka JavaScript, který bude vykonávat, anebo musí znát cestu k souboru, ve kterém je kód jazyka JavaScript uložen. [16]

Webový server systému MWE a server pro běh zpracování uzlů jsou na sobě navzájem nezávislé. Oba dva by se dali provozovat jako samostatné aplikace. Nicméně by tím přišli o možnost se vzájemně informovat o změnách, které provádějí při práci s podnikovými procesy. Ale bez vzájemné komunikace by systém jako celek nemohl správně fungovat.

Pracovní vlákna mezi sebou mohou komunikovat prostřednictvím zasílání zpráv přes porty. Práce s portem v hlavním vlákně se liší od práce s portem ve vedlejším vlákně čili v pracovníkovi. Protože oba dva servery, z nichž se skládá server MWE, si jsou svou důležitostí

[12]

rovny, tak byla v projektu vytvořena pomocná třída `WorkerHelper`. Třída se nachází v adresáři `src/utils/` v souboru `workerHelpers.ts` spolu s pomocnými funkcemi pro zasílání zpráv o změnách podnikových procesů. Pomocná třída `WorkerHelper` zapouzdřuje částečně práci s pracovními vlákny. Při vytváření instance třídy dochází k rozlišování mezi hlavním vláknem a vedlejším vláknem.

Pokud dochází k vytváření nové instance v rámci hlavního procesu, tak dojde k vytvoření nového pracovníka, kterému bude předána cesta k souboru se skriptem pro spuštění. Nově vytvořený pracovník je uchováván v instanci pomocné třídy a je využíván jako port, přes který jsou posílány zprávy. Jestliže je instance třídy vytvářena v pracovníkovi čili ve vedlejším vlákně, pak nedochází k vytvoření dalšího pracovníka, ale místo pracovníka je uchováván port odkazující na rodiče, který stojí za pracovníkovým vytvořením. Při následném posílání zprávy prostřednictvím instance pomocné třídy bude vybrán vhodný port pro zaslání zprávy. Díky tomu se zmenší náklady na režii spojenou s rozlišováním hlavního a vedlejšího vlákna při zasílání zpráv mezi servery.

Servery, z nichž se skládá server MWE, si mezi sebou sdělují dva typy informací. Prvním typem informací jsou zprávy o změnách, které se týkají instancí procesů (tj. instancí podnikových procesů, které vychází ze šablon podnikových procesů v systému.). Druhým typem informací jsou zprávy, které informují o změnách spojených s instancemi uzlů.

Obsah posílaných zpráv je tvořen z kódu neboli označení, dle kterého je možné ihned rozlišit o jaký typ zprávy se jedná. Dále jsou součástí obsahu posílané zprávy i data, jejichž obsah závisí na typu zprávy. Obsahují buďto instanci procesu nebo seznam instancí uzlů.

Při posílání zpráv mezi vlákny je dobré minimalizovat obsah posílaných dat ve zprávě. Data, která pošle ve zprávě jedno vlákno druhému vláknu, nejsou sdílená a nedochází zde podobně jako u volání funkcí k předání objektu prostřednictvím reference, ale pro druhé vlákno je vytvořená přesná kopie posílaných dat. S velikostí posílaných dat rostou tady i náklady na tvorbu jejich kopie.

Aby nedocházelo při komunikaci mezi webových serverem systému MWE a serverem pro běh zpracování uzlů ke zbytečnému posílání dat, tak soubor `workerHelpers.ts` obsahuje i pomocné funkce. Tyto pomocné funkce před odesláním zprávy zařídí, že z posílaných objektů budou odstraněny některé položky. Všechny položky, jejichž data nemají přímou souvislost se základními změnami, které mohou nastat při zpracování instancí uzlů či instancí procesů, jsou z objektů odstraněny. (*Např. při zasílání instance procesu není nutné zasílat i její šablonu, datové objekty, sekvenční toky a uzly.*)

Nyní na řadu přichází část, která vysvětlí použití pomocné třídy v serverech a objasnění obsahu souborů s názvem `workerSetup.ts`, jenž byly doteď v předchozích kapitolách přehlíženy.

Při vytváření serverů prostřednictvím funkcí v souboru `server.ts` v adresáři `src/` dochází k pokusu o vytvoření a nastavení pracovníka. Projekt je vyvíjen v jazyce TypeScript, ale pracovníci mohou pracovat pouze se skripty, jež jsou napsány v jazyce JavaScript. Není tedy možné při vývoji plně využívat nástrojů *ts-node*, *tsjest* a musí být ošetřeny při vytváření serverů případy, při kterých je server spuštěn prostřednictvím takovéhoto nástroje a informovat o této situaci na textovém výstupu.

Pro nastavení vytvořeného pracovníka slouží funkce `workerSetup`. Tato funkce se nachází v souboru `workerSetup.ts`. Úlohou funkce je propojit server s kanálem zpráv. Očekává se zde nastavení pro naslouchání a zpracování příchozích zpráv s daty, a zároveň i nastavení způsobů, jakým server bude odesílat zprávy.

Soubor `workerSetup.ts` pro nastavení serveru pro běh zpracování uzlů se nachází v adresáři `src/runnerServer/`. Je zde nastaveno naslouchání na příchozí zprávy obsahující

informace o změnách instancí uzlů. Data obsahující seznam instancí uzlů je předán do serveru prostřednictvím funkce, která slouží k reakci na změnu instancí uzlů. Přijatá data jsou tedy zpracována a na jejich základě je upravena fronta s naplánovanými instancemi uzlů pro zpracování. Dále jsou zde nastavena zpětná volání serveru pro běh zpracování uzlů, které jsou volány při změnách instancích uzlů či instancích procesů, s nimiž server pracuje. Při změně instancí uzlů dojde k vytvoření zprávy a jejímu zaslání prostřednictvím pomocné třídy `WorkerHelper` a podobným způsobem je obsloužena i reakce na změnu instance procesu.

Webový server systému MWE má soubor `workerSetup.ts` s funkcí pro nastavení jeho komunikace s druhým vláknem umístěný v adresáři `src/graphql/`. Ve funkci pro nastavení se vyskytuje jen nastavování pro příjem zpráv a samotné odesílání zpráv je realizováno ve funkcích řešitelů. Je nastaveno naslouchání na zprávy o změnách instancích uzlů, a současně i o změnách instancích procesů. Při obdržení zprávy jsou obdržená data přeposlána klientům za pomocí služeb objektu `PubSub`, který spravuje přihlášení odběrů a zasílání správ odběratelům (Viz. kapitola 5.4 *Webový server s GraphQL*). V nastavovací funkci v souboru `workerSetup.ts` není potřebná existence nastavení pro zasílání správ druhému vláknu, neboť je instance pomocné třídy pro práci s pracovními vlákny předávána v kontextu GraphQL všem řešitelům. Řešitelé následně v případě potřeby zasílají předanou instanci pomocné třídy zprávy s daty.

5.6 Interní rozhraní serveru

Interní rozhraní serveru MWE je tvořeno řadou funkcí, jejichž prostřednictvím lze relativně bezpečným způsobem interagovat se systémem MWE. Soubory obsahující funkce, které společně tvoří interní rozhraní serveru MWE, se nachází v adresáři `src/api/` a jeho podadresáři `src/api/bpmn`.

Funkce tvořící interní rozhraní serveru MWE obsahují logiku pro získávání dat z databáze, jejich filtrování, úpravu a vyřizování služeb jakými je autorizace uživatelů nebo manipulace se systémem prostřednictvím běhového jádra. Zároveň při všech těchto aktivitách dochází k autorizaci neboli ke kontrole práv, na jejíž základě je uživateli uděleno právo provádět jednotlivé akce a manipulace skrze rozhraní.

Server MWE využívá ke zprostředkování komunikace mezi serverem a klientem služeb serveru GraphQL. V současné podobě neobsahuje server MWE žádné jiné přístupové rozhraní, skrze nějž by mohli klienti interagovat se serverem. Z tohoto důvodu by v projektu nemuselo interní rozhraní serveru MWE existovat samostatně, ale mohlo by býti součástí řešitelů (Funkcí určených k řešení dotazu GraphQL v serveru GraphQL. Viz. kapitola 5.4).

Ale existují důvody pro jeho osamostatnění od řešitelů serveru GraphQL. Mimo zpřehlednění kódu ve funkcích řešitelů je myšleno i na možné budoucí rozšíření serveru MWE. Server MWE je možné při dalším vývoji snáze rozšířit nová komunikační rozhraní. Například může být vybudován server s architekturou REST pro komunikaci s klienty, ve kterém bude možné využít již existujících funkcí z interního rozhraní serveru MWE.

Pokud by neexistovalo interní rozhraní serveru MWE, tak by přidávání dalšího rozhraní mohlo vést k dvojímu kódu, který by poskytoval stejné či podobné služby a hrozilo by nejednotné nastavení přístupových práv pro jejich provádění.

Podobná situace s dvojím kódem může nastat i při rozšiřování rozhraní GraphQL o nové datové typy či akce. Dobrým příkladem z projektu můžou být akce pro změnu hesla uživatele a akce pro restart hesla uživatele.

Změnu hesla uživatele můžou provádět administrátoři uživatelů anebo si uživatel může změnit své vlastní heslo, ale musí zároveň zadat své staré heslo. Restart hesla je ve skutečnosti stejná akce jako změna hesla. Jediným rozdílem oproti změně hesla je v tom, odkud je nové heslo získáno. V případě změny hesla nové heslo zadává uživatel, který jej chce změnit, ale v případě restartu hesla nové heslo vygeneruje server.

Bez oddělení funkce pro změnu hesla by musely v serveru GraphQL existovat dva řešitelé, kteří by oba prováděli téměř stejné úkony. Nejen že by mohlo docházet k problémům (Např. dojde k opravě chyby v jednom z řešitelů, ale stejná chyba zůstala v druhém z nich.), ale zároveň se nejedná ani o dobrou programátorskou praktiku, neboť cílem programátora by mělo být psát znovupoužitelný kód.

Autentizace uživatelů

Jednou ze zajímavějších částí, která je součástí interního rozhraní serveru MWE, jsou funkce sloužící k autentizaci uživatele a generování ověřovacího tokenu. Pro účely autentizace uživatelů je v komunitě vývojářů aplikací pro Node.js velmi populární balíček *passport*[13]. Tento balíček je určen k nasazení v serverových aplikací s architekturou REST. Je navržen pro použití v kombinaci se serverem z balíčku *express*. Způsoby autentizace uživatele, kterými je možné autentizaci provádět, je možné rozšířit zásuvnými moduly vytvořenými pro balíček *passport*. Je dostupné široké spektrum zásuvných modulů, které zprovozňují autentizaci uživatele prostřednictvím různých služeb nebo postupů.

Ale je zde jeden problém. Server MWE pracuje se serverem GraphQL a nikoliv se serverem REST. Tento fakt způsobuje komplikaci při volbě a použití zásuvných modulů, které jsou použitelné i pro autentizaci v GraphQL. Aby bylo možné využít zásuvný modul, tak musí zásuvný modul poskytovat možnost reagovat na výsledek autentizace pomocí prostředků nezávislých na architektuře REST. Například obsluha výsledků autorizace funkcí zpětného volání.

Pro účely autorizace v serveru MWE jsou využity dva způsoby autorizace uživatele. První z nich je autorizace na základě lokálních dat (Zásuvný modul z balíčku *passport-local*). Jedná se o notoricky známou autorizaci pomocí přihlašovacího jména a hesla. Druhá metoda je autorizace uživatele pomocí autorizačního tokenu. Konkrétně je využita metoda nazývaná *Bearer Authentication* (Zásuvný modul z balíčku *passport-http-bearer*).

Při tomto způsobu autorizace uživatele hraje důležitou roli tokenu a data, která jsou v něm zašifrovaná. Token je kryptickým textovým řetězcem, ve kterém jsou skryté údaje o platnosti tokenu a uživateli. Server MWE používá pro generování a dekódování tokenu prostředky poskytované balíčkem `jsonwebtoken`[14]. Balíček umožňuje pracovat s tokeny JWT a provádět jejich generování, verifikaci a dekódování jejich obsahu. Samotné ověřování platnosti dekódovaných dat má plně na starost server MWE.

Veškeré funkce tvořící autorizační rozhraní se nachází v souboru `auth.ts` v adresáři `src/api/` včetně funkcí pro generování a validaci tokenů JWT.

[13] passport https://www.npmjs.com/package/passport
[14] jsonwebtoken https://www.npmjs.com/package/jsonwebtoken

Kapitola 6

Testování

Kapitola se zabývá popisem způsobu testovaní systému při jeho vývoji. Zaobírá se jednotkovými testy a integračními testy, které společně tvoří automatizované testy, jež pomáhaly při vývoji systému a ověřují funkčnost jeho částí. Zároveň se zde nachází informace o demonstrační klientské aplikaci, jejímž účelem je poskytnout prostředky pro ověření správného fungování systému workflow.

6.1 Automatizované testy

Vývoj systému workflow je již celkem komplexnější záležitostí. Při jeho vývoji může docházet k řadě vedlejším efektům, které mohou narušit očekávané chování vyvíjeného systému. Aby při vývoji nedocházelo k nepříjemným překvapením, tak je dobré využívat alespoň částečně automatizované testy. Mezi nepříjemná překvapení lze řadit například objevení nefungující části systému, která ještě před týdnem fungovala správně. Zároveň také testovaní pomocí automatických testů je rychlejší, něž manuální testování a není laxní při kontrole očekávaných výstupů.

Proto při vývoji serveru MWE jsou využity automatizované testy. Pro vytváření automatizovaných testů jsou využívány převážně prostředky a nástroje poskytované balíčkem *jest*[1] a balíčkem *jest-extended*[2]. Zdrojové kódy testů a zdrojů dat se nacházejí v adresáři `tests/` a jeho podadresářích. Rozdělení testů a struktura adresáře `tests/` odpovídá komponentám systému MWE (tj. jednotlivým částem serveru) anebo jejich účelu.

Testy pokrývají následující části serveru:

- Testování základních entit
  - Nachází se v adresáři ‘tests/db/‘.
  - Jedná se o otestování základních entit, které odpovídají internímu datovému modelu.
  - Jejich existence byla nezbytná převážně na počátku vývoje.
- Testování stavitele šablon procesů
  - Nachází se v adresáři ‘tests/bpmnBuilder/‘.

[1] jest https://www.npmjs.com/package/jest
[2] jest-extended https://www.npmjs.com/package/jest-extended

  - Jednotkové testy pro testování částí, z nichž se stavitel šablon procesů skládá, se nachází v souboru ‘parser.test.ts‘.
  - V souboru ‘bpmnBuilder.test.ts‘ se nalézají integrační testy pro ověření korektního chování stavitele šablon procesů.

- Testování hlavního běhového jádra procesů
  - Nachází se v adresáři ‘tests/bpmnRunner/‘.
  - Jednotkové testy pro testování částí, ze kterých je složeno běhové jádro procesů, se nachází v souborech ‘initHelpers.test.ts‘, ‘runContext.test.ts‘, ‘executeHelpers.test.ts‘.
  - Integrační testy, které testují celkové chování běhového jádra, jsou obsaženy v souboru ‘bpmnRunner.test.ts‘.

- Testování zásuvných modulů s implementací uzlů
  - Nachází se v adresáři ‘tests/bpmnRunnerPlugins/‘.
  - Zde jsou umístěny jednotkové testy pro testovaní zásuvných modulů s implementací uzlů.
  - Ale ukázalo se, že testování těchto modulů, je výhodnější až v rámci integračních testů nad běhovým jádrem.

- Testování rozhraní GraphQL
  - Nachází se v adresáři ‘tests/gql/‘.
  - Zde se nalézající testy pro testování rozhraní GraphQL, byly vytvořeny jako experiment.
  - Záměrem pro jejich vytvoření bylo automatizovat ověření funkčnosti rozhraní (Kontrola vrácených dat a chybových stavů).
  - Tento typ integračních testů se ukázal jako časově náročný na vývoj a vzhledem k současnému postupu při vývoji klientské aplikace bylo rozhodnuto o přerušení vytváření těchto testů pro další části rozhraní GraphQL.

6.2 Klientská aplikace

Jednoduchá klientská aplikace je vyvíjena za účelem snadnější demonstrace funkčnosti vyvíjeného workflow systému. Jelikož se jedná o klientskou aplikace, jejímž účelem je nabídnout grafické rozhraní pro práci se serverem MWE, tak byla aplikace pojmenována **MWE Client** (Klient MWE).

Projekt klient MWE se nachází v samostatném repositáři Git a je tak oddělen od projektu serveru MWE. Jedná se o webovou aplikaci, která je postavena na frameworku *Vue.js* a zároveň využívá knihovnu komponent *Vuetify*. Při rozhodování o výběru knihovny komponent, která by pomohla při vývoji klientské aplikace, byly mezi kandidáty mimo *Vuetify* taktéž knihovny *BootstrapVue* a *Buefy*. S knihovnou *BootstrapVue* jsem se již v minulosti setkal a s fungováním některých komponent jsem nebyl spokojen. Pro výběr knihovny *Vuetify* hrálo velkou roli její dobře zpracovaná dokumentace. Jelikož server MWE umožňuje komunikaci s ním jen prostřednictvím rozhraní GraphQL, tak je v klientské aplikaci za účelem komunikace využíváno služeb knihovny *vue-apollo*.

Vyvíjená klientská aplikace umožňuje provádět se systémem workflow většinu interakcí pro práci s podnikovými procesy a administraci uživatelů. Klientská aplikace plně pokrývá všechny interakce či úkony, jenž může uživatel po serveru MWE požadovat. Informace o interakcích se serverem MWE na základě různých rolích uživatelů lze nalézt v kapitole 4 *Analýza a návrh systému*.

Jednou z funkcí, kterou aplikace neumožňuje, je interaktivní vytváření podnikových procesů (tj. neobsahuje editor podnikových procesů s exportem do souborů BPMN). Pro účely vytvoření souboru BPMN s vymodelovanými podnikovými procesy je nutné využít externích nástrojů/editorů. V klientské aplikace ovšem existuje možnost vybrat soubor BPMN a před odesláním na server je možné manuálně upravit textový obsah souboru.

Klient MWE sloužil k manuálnímu testování funkčnosti serveru MWE při jeho vývoji. Sloužil k testování následujících případů:

- Testování dostupnosti komunikačního rozhraní GraphQL na serveru MWE
- Ověřování správného určování rolí uživatelů při různých interakcích se serverem
- Testování trvalého spojení klienta a serveru prostřednictvím odběrů
  - Server GraphQL umožňuje přihlásit se k odebírání některých dat, která mu jsou zaslána, pokud dojde k jejich změně.

Na základě průběžného testování skrze klientskou aplikace byly opravovány chyby nejen serverové aplikace ale i samotné klientské aplikace.

6.3 Zhodnocení

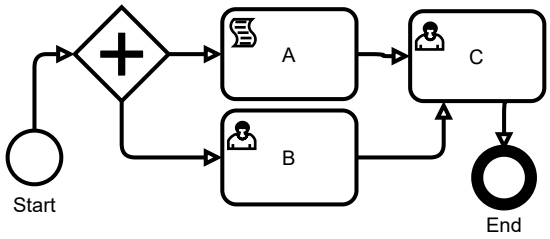
Výsledný jednoduchý workflow systém poskytuje všechny interakce a role definované při návrhu systému, který je obsahem kapitoly 4. Systém je schopný zpracovávat a řídit interní podnikové procesy, které se skládají ze základních prvků pro modelovaní podnikových procesů.

Mezi tyto prvky, které jsou ve výsledném workflow systému podporovány, patří prvky:

- **Úlohy:** Task, Script Task, Manual Task, User Task
- **Brány:** Parallel Gateway, Inclusive Gateway, Exclusive Gateway
- **Události:** Start Event, End Event, Terminate End Event, Link Intermediate Throw Event, Link Intermediate Catch Event
- **Data:** Data Object, Data Object Reference
- **Sekvence:** Sequence Flow, Condition Sequence Flow
- **Uskupení:** Pool, Lane, Colaboration

Podrobný popis podporovaných prvků a jejich chování je možné nalézt v manuálu k systému MWE.

Při testování validních scénářů tedy podnikových procesů obsahujících na první pohled jasné konstrukce prvků, které jsou udávané ve specifikaci BPMN, systém fungoval korektně. Problém nastává v okamžiku, kdy podnikový proces obsahuje konstrukce prvků, které je možné vidět na obrázku 6.1.

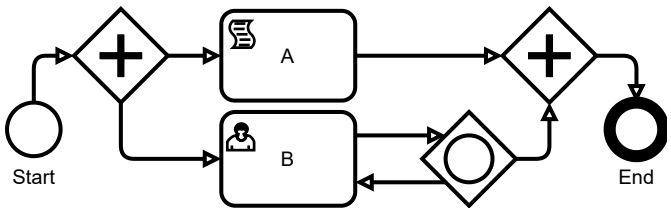


Obrázek 6.1: Ukázkový případ konstrukce podnikového procesu s vícenásobným vstupním sekvenčním tokem do úlohy, který způsobuje problémy ve vyvíjeném workflow systému.

Prvním typem problém je používání prvků (mimo Parallel Gateway) ke slučování sekvenčních toků. Dané prvky nemají pevně definované chování pro tyto případy, a proto se ve výsledném workflow systému jejich chování při takovémto případě může lišit na základě průběhu individuálních instancí procesu.

Chování systému může býti následující:

- Cíl sekvenčního toku ještě neexistuje a je vytvořena instance uzlu.
- Cíl sekvenčního toku je nalezen mezi nedokončenými instancemi uzlů a dojde k sloučení sekvenčních toků.
- Cíl sekvenčního toku již existoval a byl dokončen, proto systém založí novou instanci uzlu, neboť se domnívá, že se jedná o cyklus v procesu.



Obrázek 6.2: Ukázkový případ konstrukce podnikového procesu s cyklem, který způsobuje za určitých podmínek problémy ve vyvíjeném workflow systému.

Další problém je způsobován cykly v podnikovém procesu. Ve druhé ukázce na obrázku 6.2 lze vidět podnikový proces, který tuto chybu způsobuje. První vytvořená instance slučovací paralelní brány (prvek Parallel Gateway) po obdržení dvou sekvenčních příchozích toků bude systémem prohlášena za dokončenou. Díky smyčce na inkluzivní bráně dojde k opakovanému vytvoření instance slučovací paralelní brány, ale ta bohužel nikdy neobdrží druhý očekávaný příchozí sekvenční tok.

Pro server využívané rozhraní GraphQL neobsahuje samo o sobě žádný bezpečnostní či šifrovací mechanismus pro zabezpečení komunikace mezi serverem a klientem. Proto při jeho nasazení by mělo být zabezpečení realizováno externími prostředky. Pro většinu typů aplikací využívajících GraphQL postačuje zvolit namísto *protokolu HTTP* jeho šifrovanou variantu *protokol HTTPS*. A podobně zvolit i bezpečnou variantu protokolu, pokud aplikace s GraphQL využívá odběry (Subscriptions) postavené na protokolu WebSocket (Namísto `ws://` spíše `wss://`).

Pokud již je řeč o odběrech v GraphQL, tak demonstrační klientská aplikace a vyvíjený server mají problém s nadměrnou komunikací skrze tento mechanismus. Pokud je server

dlouhodobě plně vytížen a systém zpracovává velké množství instancí uzlů, tak jsou klientským aplikacím zasílány zprávy o změně okamžitě po každé provedené obsluze instance uzlu.

Zároveň zde nastává problém při mapovaní dat pro zaslání klientovi. Klient při žádosti k odebírání specifikuje v dotazu i strukturu dat, která chce dostávat. Ale při samotném mapování dat není dostupná informace o identitě klienta a klient se tak nedostane k datům, ke kterým by měl přístup v běžném dotazu GraphQL.

Na vině obou problémů může být mé nevhodné použití výchozí knihovny, anebo výchozí knihovna neumožňuje řešit tyto problémy. Proto by bylo dobré se pro budoucí vývoj aplikací s GraphQL zaměřit na jiné knihovny poskytující služby pro realizaci odběrů.

V kapitole 5 a jejích podkapitolách lze nalézt zmíněny další problémy v souvislosti s danou částí implementace. Tyto další problémy, které nastaly při vývoji, byly nalezeny a v mnohých případech opraveny.

Kapitola 7

Závěr

Tato bakalářská práci si kladla za cíl navrhnou a implementovat jednoduchý workflow systém, jenž je určený k provozu na platformě Node.js. Navržený a implementovaný systém pro zpracování a řízení podnikových procesů vychází ze specifikace BPMN a jeho aplikační rozhraní jest realizováno s využitím GraphQL.

Po prostudování specifikace BPMN a existujících systému byl vytvořen návrh jednoduchého workflow systému, který byl následně implementován v jazyce TypeScript. Výsledný systém je schopný zpracovávat a řídit privátní podnikové procesy složené ze základních prvků pro modelování podnikových procesů, které jsou definovány specifikací BPMN. Při ověřování funkcionality výsledného workflow systému prostřednictvím klientské aplikace, která byla k tomuto účelu vyvíjena současně se systémem, byly nalezeny některé nestandardní konstrukce podnikových procesů, na které systém neumí vhodným způsobem reagovat.

Při vývoji jsem objevil některé programové konstrukce, které s jistotou využiji v budoucích projektech. Jmenovitě se jedná například o přerušitelné sliby (Interruptible Promises) nebo automatizované testování rozhraní GraphQL.

Na rozvoji systému je možné dále pokračovat, a to například vývojem zásuvných modulů pro rozšíření systému o další podporované prvky, jež jsou užívány při modelování podnikových procesů. Ale spíše než rozšiřovat stávající systém, doporučuji roztříštit projekt na menší nezávislé moduly, které by umožňovali složení workflow systému na míru dle požadavků firem. Mám v plánu osamostatnit hlavní běhové jádro procesů a přepracovat koncept zásuvných modulů, aby lépe korespondoval s potřebami zásuvných modulů pro ovlivňování řízení procesů a usnadnil rozšiřitelnost běhového jádra o další služby.

(Demonstrační klientská aplikace je dostupná na webové adrese `https://mwe.mwarcz.cz/`*. Rozhraní vyvíjeného workflow systému je dostupné skrze webovou adresu* `https://mwe.mwarcz.cz/graphql` *a pro účely přímého procházení rozhraní je dostupný i nástroj Playground na adrese* `https://mwe.mwarcz.cz/play`*.)*

Literatura

[1] Bonitasoft. *Bonita 7.10*. 7.10. 2019 [cit. 2020-01-05]. Dostupné z: https://documentation.bonitasoft.com/bonita/7.10/.

[2] Byron, L. *GraphQL: A data query language* [online]. Facebook, září 2015 [cit. 2019-12-5]. Dostupné z: https://engineering.fb.com/core-data/graphql-a-data-query-language/.

[3] Camunda. *The Camunda BPM Manual*. 7.12. 2019 [cit. 2020-01-05]. Dostupné z: https://docs.camunda.org/manual/7.12/.

[4] Cflowapps. *Workflow Engine* [online]. Cflowapps, 2020 [cit. 2019-11-30]. Dostupné z: https://www.cflowapps.com/workflow/workflow-engine/.

[5] Chatterjee, D. *The world's best serverless database, now with native GraphQL* [online]. The Linux Foundation, červen 2019 [cit. 2019-12-20]. Dostupné z: https://fauna.com/blog/the-worlds-best-serverless-database-now-with-native-graphql.

[6] Facebook. *GraphQL - Working Draft - July 2015* [online]. Facebook, 2015 [cit. 2019-12-5]. Dostupné z: http://spec.graphql.org/July2015/.

[7] Facebook. *GraphQL - June 2018 Edition* [online]. Facebook, 2018 [cit. 2019-12-10]. Dostupné z: http://spec.graphql.org/June2018/.

[8] Foundation, G. *Introduction to GraphQL* [online]. The GraphQL Foundation, 2018 [cit. 2019-12-10]. Dostupné z: https://graphql.org/learn/.

[9] Foundation, G. *Getting Started With GraphQL.js* [online]. The GraphQL Foundation, 2019 [cit. 2019-12-10]. Dostupné z: https://graphql.org/graphql-js/.

[10] Hallie, L. *JavaScript Visualized: Event Loop* [online]. DEV.to, prosinec 2019 [cit. 2020-04-16]. Dostupné z: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif.

[11] Hasura. *Instant realtime GraphQL engine* [online]. Hasura, 2019 [cit. 2019-12-20]. Dostupné z: https://hasura.io/.

[12] Kissflow. *Is a Workflow Engine the Same as a Business Rule Engine?* [online]. Kissflow, březen 2019 [cit. 2019-11-30]. Dostupné z: https://kissflow.com/workflow/workflow-engine-business-rule-engine-difference/.

[13] Kothari, A. *What's a Workflow Engine – All You Need to Know [3+ Uses]* [online]. Tallyfy, 2018 [cit. 2019-11-30]. Dostupné z: https://tallyfy.com/workflow-engine/.

[14] Lyon, W. *Fullstack GraphQL with Neo4j* [online]. Neo4j, Inc., 2019 [cit. 2019-12-20]. Dostupné z: https://neo4j.com/blog/fullstack-graphql-with-neo4j/.

[15] Node.js. *Node.js v10.20.1 Documentation*. 10.x. 2018 [cit. 2020-04-19]. Dostupné z: https://nodejs.org/docs/latest-v10.x/api/worker_threads.html.

[16] Node.js. *Node.js v12.16.3 Documentation*. 12.x. 2019 [cit. 2020-04-19]. Dostupné z: https://nodejs.org/docs/latest-v12.x/api/worker_threads.html.

[17] Olin, E. *The Linux Foundation Announces Intent to Form New Foundation to Support GraphQL* [online]. The Linux Foundation, listopad 2018 [cit. 2019-12-5]. Dostupné z: https://www.linuxfoundation.org/press-release/2018/11/intent_to_form_graphql/.

[18] OMG.org. *Business Process Modeling Notation Specification*. 1.0. Listopad 2006 [cit. 2019-12-16]. Dostupné z: https://www.omg.org/spec/BPMN/1.0/PDF.

[19] OMG.org. *Business Process Model and Notation (BPMN)*. 2.0.2. Leden 2011 [cit. 2019-12-16]. Dostupné z: https://www.omg.org/spec/BPMN/2.0/PDF.

[20] OMG.org. *About the business process model and notation specification version 2.0.2* [online]. OMG, leden 2014 [cit. 2019-12-30]. Dostupné z: https://www.omg.org/spec/BPMN/.

[21] Oracle. *Oracle Workflow API Reference*. 2.6.4. červen 2006 [cit. 2019-11-30]. Dostupné z: https://docs.oracle.com/cd/B19306_01/workflow.102/b15855.pdf.

[22] Parody, L. *Understanding Worker Threads in Node.js* [online]. NodeSource, červen 2019 [cit. 2020-04-16]. Dostupné z: https://nodesource.com/blog/worker-threads-nodejs.

[23] Perry, S. *Introduction to the event loop in Node.js* [online]. IBM, prosinec 2018 [cit. 2020-04-16]. Dostupné z: https://developer.ibm.com/technologies/node-js/tutorials/learn-nodejs-the-event-loop/.

[24] RetHat. *JBPM Documentation*. 7.31.0.Final. 2019 [cit. 2020-01-05]. Dostupné z: https://docs.jboss.org/jbpm/release/latestFinal/jbpm-docs/html_single/.

[25] Yijun. *Solving Memory Leaks Caused by Co and Recursive Calls* [online]. Alibaba Cloud, červenec 2019 [cit. 2020-03-25]. Dostupné z: https://www.alibabacloud.com/blog/solving-memory-leaks-caused-by-co-and-recursive-calls_595121.

Příloha A

Obsah přiloženého paměťového média

Obsah kořenového adresáře:

- `MWEServer/`: Zdrojové texty vyvíjeného workflow systému
- `MWEClient/`: Zdrojové texty demonstrační klientské aplikace
- `zprava/`: Zdrojové texty pro vytvoření zprávy
- `resources/`: Zdrojová forma obrázků a ukázek
- *bc.pdf*: Technická zpráva

Demonstrační klientská aplikace je dostupná na webové adrese https://mwe.mwarcz.cz/. Rozhraní vyvíjeného workflow systému je dostupné skrze webovou adresu https://mwe.mwarcz.cz/graphql a pro účely přímého procházení rozhraní je dostupný i nástroj Playground na adrese https://mwe.mwarcz.cz/play.

Zdrojové texty klientské aplikace a serveru jsou dostupne i na webu Github.

- Server https://github.com/MWarCZ/MWEServer
- Klient https://github.com/MWarCZ/MWEClient

Příloha B

Jak vytvořit BPMN soubor

Úvodní seznámení

Prozatím existuje jediný způsob, jak vytvořit/nahrát šablonu procesu do serverové aplikace *MWEServer*. MWEServer je schopen zpracovat textový soubor obsahující definici procesu dle specifikace BPMN 2.0. Pro vytváření souborů BPMN určených k testování aplikace MWEServer byl využíván jeden z ukázkouvých příkladů, který používající balíček **bpmn-js**[1] a **bpmn-js-properties-panel**[2].

Konkrétně se jedná o jeden z ukázkových příkladů **bpmn-js-examples**[3], který je volně dostupný na serveru GitHub. Tento nástroj byl zprovozněn a to s panelem určeným původně pro vytváření souborů BPMN pro nástroje od společnosti Camunda. V současné době je dostupný na mém osobním serveru na URL bpmnio.mwarcz.cz (`http://bpmnio.mwarcz.cz`).

Většinu potřebných úkonů potřebných k modelování procesu a vygenerování souboru BPMN potřebného pro aplikaci MWEServer lze v tomto nástroji vykonat. Výjimku činí pouze elementy a atributy spadající pod jmenný prostor `http://www.mwarcz.cz/mwe/bpmn/` (Dále označován jen jako *jmenný prostor mwe.*), které je nutné přidat manuálně do souboru BPMN.

Omezení

Při modelování je možné použít veškeré elementy, které jsou vyjmenovány a popsány v manuálu k systému tj. kapitola C. MWEServer v aktuální verzi nepodporuje podprocesy (SubProcess) a je schopen zpracovat jen základní procesy. Pro základní procesy platí dle specifikace BPMN, že v jednom bazénu (Pool) může být jen jeden základní proces. Dále MWEServer nepodporuje zasílání zpráv mezi základními procesy, které je možné vymodelovat v nástroji. Základní proces lze vymodelovat a nahrát na MWEServer i když není umístěn do žádného bazénu.

[1] Balíček **bpmn-js** `https://github.com/bpmn-io/bpmn-js`

[2] Balíček **bpmn-js-properties-panel** `https://github.com/bpmn-io/bpmn-js-properties-panel`

[3] Projekt **bpmn-js-examples** `https://github.com/bpmn-io/bpmn-js-examples/tree/master/properties-panel`

Modelování a manuální úpravy

Modelování v nástroji je dosti intuitivní. Po vložení elementu a jeho označení je možné změnit jeho druh za pomocí ikony klíče. (Např. element *Task* lze změnit na *ScriptTask, ManualTask* aj.) Přidávání a úprava atributů je možné pomocí bočního panelu. Po označení elementu se v tomto panelu ukáže sada vlastností. Podporované vlastnosti/atributy naleznete popsány v manuálu k systému tj. kapitola C u jednotlivých elementů.

Mezi nejdůležitější elementy a jejich atributy patří následující:

- Lane
  - Atribut Name: Určuje název skupiny, která má vliv na některé elementy v něm umístěné.
  - Např. spouštení procesu (StartEvent), obsluha úlohy (UserTask, ManualTask)
- ScriptTask
  - Script: Zde umístěný skript bude spouštěn při provádění uzlu.
- SequenceFlow.ConditionExpression
  - Expression: Podminka zde přítomná ovlivňuje sekvenční tok procesu.
- DataObject, DataObjectReference
  - Atribut Name: Ovlivňuje pod jakým názvem budou data dostupná pro uzly či podmíněné sekvenční toky.

Výsledný vymodelovaný proces/procesy lze stáhnout ve formě souboru BPMN a SVG.

Jak již bylo zmíněno dříve, tak některé důležité elementy a atributy ze *jmenného prostoru mwe* je nutné přidat manuálně do staženého souboru BPMN vygenerovaného nástrojem. Nejdůležitější je úprava elementu *DataObject*. *Jmenný prostor mwe* přidává možnost vložit do datového objektu výchozí data. Pro použití *jmenného prostoru mwe* je nutné jej přidat do jmenných prostorů definovaných v kořenovém elementu `definitions`, tak jako je možné vidět na následujícím příkladu B.1.

```
<bpmn2:definitions ...
       xmlns:bpmn2="http://www.omg.org/spec/BPMN/20100524/MODEL"
       xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
       xmlns:mwe="http://www.mwarcz.cz/mwe/bpmn/"
       ... > ... </bpmn2:definitions>
```

Obrázek B.1: Vzorová ukázka přidání *jmenného prostoru mwe* do kořenového elementu `definitions`.

Další ukázka kódu ukazuje přidání podelementů do elementu datového objektu, který přidává do objektu výchozí data. V datovém objektu je hledán element `json` z *jmenného prostoru mwe*, který obsahuje text. Jak název elementu napovídá, tak obsahem je text ve formátu JSON. (MWEServer v aktuální verzi vyžaduje, aby obsah elementu `json` začínal objektem.) Vzorovou ukázku, jak by měla definice datového objektu vipadat, lze spatřit na ukázce B.2.

```
<bpmn2:dataObject id="DataObject_0xhy3ix">
       <bpmn2:extensionElements>
              <mwe:json>
              { "text": "lorem", "isIt": true }
              </mwe:json>
       </bpmn2:extensionElements>
</bpmn2:dataObject>
```

Obrázek B.2: Vzorová ukázka definice datového objektu s výchozími daty.

Závěr

Před samotným modelováním procesů pro MWEServer je doporučeno nastudovat minimálně kapitolu C.

Příloha C

Manuál k systému

C.1 Výchozí účty a přihlašovací údaje

Server MWE při svém zprovoznění obsahuje výchozí uživatelské účty. Následující seznam obsahuje chráněné uživatelské účty, které mají trvalé členství v chráněných skupinách, a výchozí přihlašovací údaje pro přihlášení se k účtům. Existence chráněných účtů zjišťuje, aby nedošlo k trvalé ztrátě práv provádět některé akce pro všechny existující uživatele na serveru.

Seznam chráněných uživatelských účtů:

- Systémový uživatel
  - Přihlašovací jméno: **system**
  - Není možné se přihlásit jako systémový uživatel.
  - Systémový uživatel je vyhrazen pro označení serveru neboli zjednodušeně systémový uživatel je serverem využívaný účet.
- Administrátor uživatelů
  - Přihlašovací jméno: **useradmin**
  - Výchozí heslo: **UserAdmin**
  - Je trvalým členem skupiny *UserAdmin*.
- Super administrátor uživatelů
  - Přihlašovací jméno: **superuseradmin**
  - Výchozí heslo: **SuperUserAdmin**
  - Je trvalým členem skupiny *SuperUserAdmin*.
- Administrátor skupin
  - Přihlašovací jméno: **groupadmin**
  - Výchozí heslo: **GroupAdmin**
  - Je trvalým členem skupiny *GroupAdmin*.
- Super administrátor skupin

  - Přihlašovací jméno: **supergroupadmin**
  - Výchozí heslo: **SuperGroupAdmin**
  - Je trvalým členem skupiny *SuperGroupAdmin*.

- Nejvyšší manažer
  - Přihlašovací jméno: **topmanager**
  - Výchozí heslo: **TopManager**
  - Je trvalým členem skupiny *TopManager*.

Na serveru se po zprovoznění nacházejí i obyčejní nechránění uživatelé. Přihlašovací jména obyčejných uživatelů mají formát `user<číslo>` (např. *user7*, *user8*, *user9*). Pro výchozí přihlašovací hesla obyčejných uživatelů platí, že je stejné jako jejich uživatelské jméno. Výchozí obyčejní uživatelé slouží jen jako ukázka uživatelských účtů v různých stavech (např. aktivní, uzamknutý, chráněný, odstraněný).

C.2 Chráněné skupiny a role

- **UserAdmin**
  - Administrátor uživatelů
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny *UserAdmin*.
  - Mezi jeho privilegia patří vytváření nových uživatelských účtů, odstraňování existujících uživatelských účtů nebo jejich uzamykání v případě potřeby.
  - Oproti roli *User* je mu umožněno upravovat osobní a přihlašovací údaje všech jemu dostupných uživatelů.

- **SuperUserAdmin**
  - Jedná se o speciální typ administrátora uživatelů, který má navíc některá oprávnění.
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny *SuperUserAdmin*.
  - Může provádět destruktivní operaci trvalé smazání uživatelského účtu nebo jeho obnovení ze stavu odstraněného do stavu aktivního.

- **GroupAdmin**
  - Administrátor skupin
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny *GroupAdmin*.
  - Jeho privilegia jsou vytváření nových skupin, odstraňování existujících skupin, úprava informací o skupinách.
  - Dále může také přidávat nové členy do libovolné skupiny a odebírat členy z libovolné skupiny.

- **SuperGroupAdmin**
  - Jde o speciální typ administrátora skupin, který má navíc některá oprávnění.
  - Roje je udělena uživatelům, kteří jsou členy chráněné skupiny *SuperGroupAdmin*.

  - Může provádět destruktivní operaci pro trvalé smazání skupiny nebo obnovení skupiny ze stavu odstraněná do stavu aktivní.

- **TopManager**
  - Nejvyšší manažer
  - Role je udělena na základe členství uživatele ve šupině *TopManager*.
  - Umožňuje nahrávat nové šablony procesů na server a zároveň manipulace nad všemi šablonami procesů a jejich instance včetně destruktivních akcí jakými je trvalé smazání šablony nebo instance procesu.

C.3 Podporované elementy BPMN

Kapitola obsahuje popis a ukázky použití elementu BPMN, které server MWE podporuje.

C.3.1 Definitions

Element je kořenovým elementem souboru BPMN. Slouží k definování jmenných prostorů použitých pro všechny elementy obsažené v souboru BPMN.

Ukázka XML C.1:

```
<definitions xmlns:name="uri"
       xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
       xmlns:camunda="http://camunda.org/schema/1.0/bpmn"
       xmlns:mwe="http://www.mwarcz.cz/mwe/bpmn/"
       ...
>
       <collaboration>...</collaboration>
       <process>...</procees>
       ...
</definitions>
```

Obrázek C.1: Ukázka definice elementu `definitions` v souboru BPMN.

Atributy:

- `xmlns`
  - Výchozí jmenný prostor pro značky XML jejichž název neobsahuje předponu oddělenou dvojtečkou.

- `xmlns:foo`
  - Definuje alias pro jmenný prostor, který je použit jako předpona u názvů značek XML.
  - V definici za dvojtečkou je specifikován název aliasu.

C.3.2 Process

Element představuje podnikový proces. Je přímím potomkem elementu `definitions`.

Ukázka XML C.2:

```
<process id="id" name="string" isExecutable="bool"
    processType="none | private | public"
    camunda:versionTag="number | semver"
    mwe:versionType="number | semver"
    mwe:version="number | semver">
        ...
</process>
```

Obrázek C.2: Ukázka definice elementu `process` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název procesu.
- `isExecutable`
  - Příznak spustitelnosti procesu.
- `processType`
  - Typ podnikového procesu.
- `mwe:versionType`
  - Formát verze.
  - Platné volby: *number* (Celé číslo), *semver* (*Major.Minor.Path*).
- `mwe:version`
  - Verze procesu.
  - Důležité při stejném *id* pro rozlišení procesu.
- `camunda:versionTag`
  - Verze procesu.
  - Stejný účel jako *mwe:version*.
  - Při existenci obou dvou atributů je použit atribut *mwe:version*

C.3.3 Colaboration

Kolaborace současný systém využívá pro identifikaci skupin uživatelů, kteří se stávají potencionálními manažery. Těmto uživatelům systém přiděluje práva provádět nad podnikovými procesu některé úkony pro správu šablon procesů a instancí procesů. Element kolaborace je přímím potomkem elementu `definitions`.

Ukázka XML C.3:

```
<collaboration>
       <participant name="string" processRef="id_process" />
       ...
</collaboration>
```

Obrázek C.3: Ukázka definice elementu `collaboration` v souboru BPMN.

Atributy:

- `participant`
  - `name`
    - Název účastníka.
    - Pro systém představuje rovněž název skupiny uživatelů, která obsahuje potencionální manažery.
  - `processRef`
    - Reference na proces, ke kterému bude navázán název skupiny.

C.3.4 Pool, Lane

Následující elementy hrají pro systém důležitou roli. Systém následující elementy využívá pro identifikaci skupin uživatelů, kteří se stávají kandidáty na nabyvatele uzlů či aktéry. Elementy `lane` nesou název, který je systémem vnímán jako názvem skupiny uživatelů. Název skupiny je přiřazen všem uzlů, jejichž reference se v `lane` nachází. Identifikovaní uživatelé následně mohou obsadit a obsloužit instance uzlů nebo zahájit instanci procesu přes daný uzel. Element `laneSet` je přímím potomkem elementu `process`.

Ukázka XML C.4:

C.3.5 Sequence Flow

Jedná se o element představující sekvenční tok. Určuje průběh vytváření instancí uzlů během postupu při vykonávání instance procesu. Podpora různých typů sekvenčních toků závisí na konkrétních implementaci pro zpracování uzlů *(viz. Task, Gateway, StartEvent, EndEvet atd.)*. V základu systém rozlišuje dva typy sekvenčních toků.

- Běžný sekvenční tok
  - K provedení sekvenčního toku dojde vždy.

```
<laneSet>
       <lane name="string">
              <flowNodeRef>
                     <!-- id_taks | id_gateway | id_event | id_data -->
              </flowNodeRef>
       </lane>
       <!-- ... -->
</laneSet>
```

Obrázek C.4: Ukázka definice elementu `laneSet` v souboru BPMN.

- Podmíněný sekvenční tok
  - K provedení sekvenčního toku dojde jen za splnění výrazu, který obsahuje.
  - Současná verze systému podporuje výrazy, které jsou tvořeny kódem jazyka JavaScript.
  - Ve výrazech je možné využívat položky kontextu (viz. kontext).

Ukázka XML C.5:

```
<!-- Bezny sekvencni tok -->
<sequenceFlow id="id" name="string"
    sourceRef="id_taks | id_gateway | id_event"
    targetRef="id_taks | id_gateway | id_event" />
<!-- Podmineny sekvencni tok -->
<sequenceFlow id="id" name="string"
    sourceRef="id_taks | id_gateway | id_event"
    targetRef="id_taks | id_gateway | id_event">
        <conditionExpression xsi:type="bpmn2:tFormalExpression">
            1=='1'
        </conditionExpression>
</sequenceFlow>
```

Obrázek C.5: Ukázka definice elementu `sequenceFlow` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název sekvenčního toku.
- `sourceRef`
  - Reference na uzel, ze kterého sekvenční tok vychází.

- Uzlem může být úloha, událost nebo brána.

- `targetRef`
  - Reference na uzel, do kterého sekvenční tok vstupuje.
  - Uzlem může být úloha, událost nebo brána.

C.3.6 Data Object

Datový objekt pro uchovávání dat, které v podnikových procesech jsou součástí datových toků. Systém umožňuje do datového objektu vložit výchozí data za využití rozšiřujícího elementu `json`, v němž je možné uvést data, která budou formátována do podoby textového řetězce ve formátu JSON.

Ukázka XML C.6:

```
<!-- Prazdny datovy objekt -->
<dataObject id="id" name="string"/>
<!-- Datovy objekt obsahujici vychozi data -->
<dataObject id="id" name="string" mwe:strict="bool">
       <extensionElements>
              <mwe:json>
                     { "json": "with", "data": 1 }
              </mwe:json>
       </extensionElements>
</dataObject>
```

Obrázek C.6: Ukázka definice elementu `dataObject` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.

- `name`
  - Název datového objektu.
  - Pod tímto názvem objekt bude dostupný v systému.

- `mwe:strict`
  - Příznak upravující chování uzlů, které ukládají data do datového objektu.
  - Příznak se týká pouze výchozích údajů, které uzly mohou generovat po svém zpracování.

- `mwe:json`
  - Výchozí data poskytovaná datovým objektem.
  - Očekávána je textová hodnota obsahující data ve formátu JSON.

C.3.7 Data Object Reference

Reference ukazující na datový objekt. Zastupuje datový objekt v diagramu a umožňuje jeden datový objekt použit na více místech a zároveň zachovat přehlednost diagramu. V případě, že datový objekt není pojmenován, je datový objekt pojmenován po jménu reference na datový objekt.

Ukázka XML C.7:

```
<dataObjectReference id="id" name="string"
   dataObjectRef="id_dataObject"/>
```

Obrázek C.7: Ukázka definice elementu `dataObjectRef` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název reference.
  - Pokud datový objekt není pojmenován je pro jeho jméno použit název první pojmenované datové reference, která na datový objekt odkazuje.
- `dataObjectRef`
  - Reference/Odkaz na datový objekt s odpovídajícím unikátním identifikátorem.

C.3.8 Task

Abstraktní úloha slouží převážně pro účely ladění a testování. Ve výchozím stavu je její chování nastaveno na úspěšné dělání ničeho. Abstraktní úloha v systému spadá mezi uzly a všechny uzly mohou systému oznámit, jakou implementací chtějí být zpracovávány.

Ukázka XML C.8:

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN
- `name`
  - Název úlohy.
- `mwe:implementation`
  - Název implementace uzlu v systému MWE, která bude použita pro danou úlohu místo výchozího chování pro tento typ uzlu.

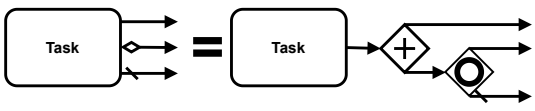
```
<task id="id" name="string" mwe:implementation="implementation_name">
       <dataInputAssociation>
              <sourceRef>
                     <!-- id_dataObject -->
              </sourceRef>
              ...
       </dataInputAssociation>
       <dataOutputAssociation>
              <targetRef>
                     <!-- id_dataObject -->
              </targetRef>
              ...
       </dataOutputAssociation>
</task>
```

Obrázek C.8: Ukázka definice elementu `task` v souboru BPMN.

- `dataInputAssociation`
  - Definuje vstupující datové toky, které nesou data dostupná pro uzel při zpracování.
  - `sourceRef`
    - Odkaz/Reference na datový objekt ze kterého proudí data do uzlu.
- `dataOutputAssociation`
  - Definuje výstupní datové toky, které nesou data vzniklá po zpracování uzlu.
  - `targerRef`
    - Odkaz/Reference na datový objekt do kterého proudí data z uzlu pro uložení.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro abstraktní úlohu se nazývá *Task* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/task.ts`. Implementace uzel vždy zpracuje úspěšně a při dokončování dochází k výběru dalších uzlů pro spuštění na základě sekvenčních toků. Podporuje i vícenásobné a podmíněné odchozí sekvenční toky. Na obrázku C.9 je znázorněno, jak jsou chápány jednotlivé typy sekvenčních toků.



Obrázek C.9: Způsob převodu vícenásobných a podmíněných odchozích sekvenčních toků u úloh.

Při vyhodnocování podmíněných toků je možné využívat globálních objektů, které tvoří kontext. Vstupní a výstupní datové objekty dostupné skrze kontext jsou pouze datové objekty, které odpovídají datovým toků vedoucích z/do uzlu (viz. [Kontext]).

C.3.9 Script Task

Úloha umožnující spouštět/provádět skript v ní obsažený. Struktura je podobná elementu `task` a rozšiřuje ji o několik dalších částí.

Ukázka XML C.10:

```
<scriptTask scriptFormat="js | javascript" id="id" name="string"
   mwe:implementation="implementation_name">
       <script>
              <!-- Skript v~jazyce JavaScript -->
       </script>
       <!-- Viz. Task-->
</scriptTask>
```

Obrázek C.10: Ukázka definice elementu `scriptTask` v souboru BPMN.

Atributy:

- Společné atributy s elementem *Task* jsou k nalezeni u elementu *Task*.
- `scriptFormat`
  - V současném stavu je podporován jen jazyk JavaScript.
  - V budoucnosti je možné uvažovat o rozšířené podpoře skriptovacích jazyků.
- `script`
  - Skript ve zvoleném jazyce, který bude uzlem proveden při zpracování.
  - (Neplatí pokud je výchozí implementace změněna atributem *mwe:implementation*.)

Zásuvný modul s implementací uzlu:

Výchozí implementace pro úlohu skriptu se nazývá *ScriptTask* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/scriptTask.ts`. Implementace během zpracování se pokusí provést skript. Implementace částečně využívá implementace 'Task' a díky tomu při výběru dalších uzlů při svém dokončování je schopná pracovat s vícenásobnými a podmíněnými odchozími sekvenčními toky (viz. *Task*).

C.3.10 Manual Task

Manuální úloha definuje úlohu externího charakteru. Jedná se o typ úlohy, která je vykonána externě mimo systém a v systému dochází jen k potvrzení jejího dokončení. (Například úloha *Výjezd elektrikářů k opravě elektrického vedení* je manuální úlohou a její dokončení potvrdí osoba pracující se systémem třeba po telefonické komunikaci s elektrikáři.)

```
<manualTask id="id" name="string"
    mwe:implementation="implementation_name">
        <!-- Viz. Task-->
</manualTask>
```

Obrázek C.11: Ukázka definice elementu `manualTask` v souboru BPMN.

Ukázka XML C.11:

Atributy:

- Společné atributy s elementem *Task* jsou k nalezeni u elementu *Task*,
- Neobsahuje žádné další odlišné atributy.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro manuální úlohu se nazývá *ManualTask* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/manualTask.ts`. Implementace provede zpracování úlohy až po jejím potvrzení. K potvrzení dojde vlivem uživatele, který zabral obsluhovaný uzel a obsloužil jej. Pro obsluhu uživatelem generuje implementace požadavky na dodatky. Dodatkem je položka s názvem `state`, která představuje typ potvrzení manuální úlohy (completed = splněná, storno = zrušená). Vlastní fáze zpracování proběhne vždy úspěšně a během dokončování dochází k výběru následujících uzlů. Výběr následujících uzlů je vykonáván za pomocí implementace *Task* a je tedy možné využívat vícenásobné a podmíněné výchozí sekvenční toky (viz. Task). Implementace dále vkládá do výchozích datových objektů položku `_state`, která obsahuje stav/typ potvrzení (*completed, storno*).

C.3.11 User Task

Uživatelská úloha slouží k modelování práce, jenž vykonává pracovník firmy, který je v systému. Uživatelská úloha může být zabrána potencionálními pracovníky pro obsluhu a následně jimi může být obsloužena.

Ukázka XML C.12:

```
<userTask id="id" name="string"
    mwe:implementation="implementation_name">
        <!-- Viz. Task-->
</userTask>
```

Obrázek C.12: Ukázka definice elementu `userTask` v souboru BPMN.

Atributy:

- Společné atributy s elementem *Task* jsou k nalezeni u elementu *Task*.
- Neobsahuje žádné další odlišné atributy.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro uživatelskou úlohu se nazývá *UserTask* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/userTask.ts`. Implementace provede zpracování úlohy až po jejím obsloužení uživatelem. Pro obsloužení je nutné doplnit dodatky, které jsou definovány ve vstupním datovém objektu. Ve vstupním datovém objektu je vyhledávána položka `$form` jejíž obsah tvoří požadavky na dodatky. Až po doplnění dodatků probíhá fáze zpracování, při které dojde k uložení získaných dat od uživatele do výstupních datových objektů (Název dodatku v požadavcích se shoduje s názvem, pod kterým bude uložen do datových objektů. Např. dodatek *Jmeno* bude uložen do datového objektu do položky *Jmeno*). Ve fázi dokončení probíhá výběr následujících uzlů, který je vykonáván za pomocí implementace *Task* a je tedy možné využívat vícenásobné a podmíněné výchozí sekvenční toky (viz. Task). Zároveň

Vstupní formulář a požadavky na dodatky: Příklad obsahu datového objektu, který obsahuje formulář, je vidět na následující ukázce kódu. “‘json “‘

```
{
       "$form": {
              "Jmeno": {
                  "type": "text", "hints": "Doplnte cele jmeno."
              },
              "Plat": {
                  "type": "number", "default": 110,
                  "hints": "Plat na~hodinu (Kc)."
              },
              "Vek": {
                  "type": "range", "default": 18,
                  "possibilities": [ 1, 18, 70 ]
              },
              "Cizinec": { "type": "checkbox", "default": false },
              "Pohlavi": {
                  "type": "select",
                  "possibilities": [ "Muz", "Zena" ],
                  "default": "Muz"
              }
       }
}
```

Obrázek C.13: Ukázka dat pro vytvoření formuláře k usertask.

Položky tvořící požadavky na dodatky musí obsahovat typ (type) a dále mohou obsahovat nápovědu (hints), výchozí hodnotu (default) a pole s možnostmi (possibilities). Podporované typy požadavků a obsah jejich položek záleží převážně na schopnostech klientské aplikace.

Zde je seznam možných typů požadavků na dodatky (položek formuláře), které by měli být podporovány:

- `text`
  - Očekává se textový řetězec.

  - ‘default‘: Volitelná výchozí textová hodnota položky formuláře.
  - ‘hints‘: Volitelná textová nápověda pro položku formuláře.

- `password`
  - Očekáván je textový řetězec.
  - ‘default‘: Volitelná výchozí textová hodnota položky formuláře.
  - ‘hints‘: Volitelná textová nápověda pro položku formuláře.

- `number`
  - Je očekávána číselná hodnota.
  - ‘default‘: Volitelná výchozí číselná hodnota položky formuláře.
  - ‘hints‘: Volitelná textová nápověda pro položku formuláře.

- `checkbox`
  - Je očekávána pravdivostní hodnota (true, false).
  - ‘default‘: Volitelná výchozí pravdivostní hodnota položky formuláře.
  - ‘hints‘: Volitelná textová nápověda pro položku formuláře.

- `range`
  - Je očekávána číselná hodnota.
  - ‘default‘: Volitelná výchozí číselná hodnota položky formuláře.
  - ‘hints‘: Volitelná textová nápověda pro položku formuláře.
  - ‘possibilities‘: Volitelné číselné tříprvkové pole ve formátu [velikost kroku, minimální hodnota, maximální hodnota].

- `select`
  - Je očekáváno vybrání jedné z položek z pole ‘possibilities‘.
  - ‘default‘: Volitelná výchozí hodnota položky formuláře.
  - ‘hints‘: Volitelná textová nápověda pro položku formuláře.
  - ‘possibilities‘: Povinné pole obsahující povolené hodnoty pro výběr.

- `hidden`
  - Očekává se, že bude odeslána výchozí hodnota.
  - ‘default‘: Poviná výchozí hodnota položky formuláře.

- `html`
  - Slouží za účelem zobrazení informací uživateli.
  - ‘hints‘: Povinný textový řetězec obsahující HTML.

C.3.12 Start Event

Počáteční událost pro započetí nové instance procesu. V aktuální podobě systému jsou všechny počáteční události brány za běžné události čekající na manuální spuštění procesu.

```
<startEvent id="id" name="string"
    mwe:implementation="implementation_name">
        <dataOutputAssociation>
                <targetRef>
                        <!-- id_dataObject -->
                </targetRef>
        </dataOutputAssociation>
</startEvent>
```

Obrázek C.14: Ukázka definice elementu `startEvent` v souboru BPMN.

Ukázka XML C.14:

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název události.
- `mwe:implementation`
  - Název implementace uzlu v systému MWE, která bude použita pro daný uzel místo výchozího chování pro tento typ uzlu.
- `dataOutputAssociation`
  - Definuje výstupní datové toky, které nesou data vzniklá po zpracování uzlu.
  - `targerRef`
    - Odkaz/Reference na datový objekt do kterého proudí data z uzlu pro uložení.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro počáteční událost se nazývá *StartEvent* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/startEvent.ts`. Implementace je provedena vždy úspěšně a při dokončování dochází k výběru následujících uzlů na základě výstupních sekvenčních toků. Jsou podporovány jen jednoduché sekvenční toky (tj. mimo výchozí a podmíněné). V případě výskytu vícero odchozích sekvenčních toků, jsou následující uzly spuštěny jako paralelně.

C.3.13 End Event

Účelem ukončovací události je předat informaci o ukončení instance procesu. Systém rozlišuje zatím dva typy ukončovacích událostí:

- Běžná ukončovací událost
  - Událost pouze informuje, že proces dospěl do jednoho ze svých konců a je možné proces ukončit, pokud již neexistují žádné uzly čekající na zpracování.

- Přerušující ukončující událost (Obsahuje element `terminateEventDefinition`)
  - Událost vynutí násilně ukončení procesu a přeruší všechny uzly, které ještě čekají na zpracování.

Ukázka XML C.15:

```
<!-- Bezna ukoncovaci udalost -->
<endEvent id="id" name="string"
    mwe:implementation="implementation_name">
        <dataInputAssociation>
                <sourceRef>
                        <!-- id_dataObject -->
                </sourceRef>
                ...
        </dataInputAssociation>
</endEvent>
<!-- Prerusujici ukoncujici udalost -->
<endEvent id="id" name="string"
    mwe:implementation="implementation_name">
        <dataInputAssociation>
                <sourceRef>
                        <!-- id_dataObject -->
                </sourceRef>
                ...
        </dataInputAssociation>
        <terminateEventDefinition />
</endEvent>
```

Obrázek C.15: Ukázka definice elementu `endEvent` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název úlohy.
- `mwe:implementation`
  - Název implementace uzlu v systému MWE, která bude použita pro daný uzel místo výchozího chování pro tento typ uzlu.
- `dataInputAssociation`
  - Definuje vstupující datové toky, které nesou data dostupná pro uzel při zpracování.

- `sourceRef`
  - Odkaz/Reference na datový objekt ze kterého proudí data do uzlu.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro běžnou ukončovací událost se nazývá *EndEvent* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/endEvent.ts`. Implementace je provedena vždy úspěšně a při dokončování nedochází k výběru následujících uzlů. Implementace nevynucuje ukončení, ale pouze oznamuje, že jedna z větví dosáhla konce a je možné v tento okamžik prohlásit proces za dokončený pokud se jedná o poslední větev.

Výchozí implementace pro přerušující ukončovací událost se nazývá *TerminateEndEvent* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/termina` Implementace je provedena vždy úspěšně a při dokončování nedochází k výběru následujících uzlů. Implementace vynucuje ukončení procesu a všechny nedokončené uzly budou přerušeny.

C.3.14 Intermediate Throw Event

Jedná se o přechodnou událost, do které vstupuje sekvenční tok, který je odkloněn dle zvolené definice události. Systém aktuálně podporuje jen jeden typ přechodné události a to konkrétně přechodnou vstupní spojující událost. Jedná se o událost, která přehodí sekvenční tok na přechodnou výstupní spojující událost.

Ukázka XML C.16:

```
<!-- Prechodna vstupni spojujici udalost -->
<intermediateThrowEvent id="id" name="string">
       <linkEventDefinition />
</intermediateThrowEvent>
```

Obrázek C.16: Ukázka definice elementu `intermediateThrowEvent` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název události.
  - Hraje roli při vyhledávání výstupních uzlů k přesměrování sekvenčního toku.
- `linkEventDefinition`
  - Přechodná vstupní spojující událost
  - Sekvenční tok vstupující do události je odkloněn na všechny uzly typu Přechodná výstupní spojující událost, která mají stejný název.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro přechodnou vstupní spojující událost se nazývá *LinkIntermediateThrowEvent* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/linkIntermediateEvent.ts`. Implementace vyhledá v procesu uzly typu Přechodná výstupní spojující událost a vybere je za následující uzly pro provádění procesu.

C.3.15 Intermediate Catch Event

Přechodná událost ze které vystupuje sekvenční tok, který je na ní odkloněn dle zvolené definice události. Systém aktuálně podporuje jen jeden typ přechodné události a to konkrétně přechodnou výstupní spojující událost. Jedná se o událost, která zachytí sekvenční tok, jenž je na ní přehozen přechodnou vstupní spojující událost.

Ukázka XML C.17:

```
<!-- Prechodna vystupni spojujici udalost -->
<intermediateCatchEvent id="id" name="string">
       <linkEventDefinition />
</intermediateCatchEvent>
```

Obrázek C.17: Ukázka definice elementu `intermediateCatchEvent` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název události.
  - Hraje roli při vyhledávání výstupních uzlů k přesměrování sekvenčního toku.
- `linkEventDefinition`
  - Přechodná výstupní spojující událost
  - Tvoří výstupní konec pro uzly typu Přechodná vstupní spojující událost, které na něj přehazují sekvenční tok.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro přechodnou výstupní spojující událost se nazývá *LinkIntermediateCatchEvent* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/linkIntermediateEvent.ts`. Pro výběr následujících uzlů jsou podporovány jen jednoduché sekvenční toky (tj. mimo výchozí a podmíněné).

C.3.16 Parallel Gateway

Paralelní brána je uzlem sloužícím k rozvětvení či sloučení sekvenčního toku. Při rozdělení dochází k paralelnímu vytvoření následujících uzlů. Při slučování dochází k synchronizaci sekvenčních toků před pokračováním sekvenčního toku.

Ukázka XML C.18:

```
<parallelGateway id="id" name="string">
</parallelGateway>
```

Obrázek C.18: Ukázka definice elementu `parallelGateway` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název brány.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro paralelní bránu se nazývá *ParallelGateway* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/gateway.ts`. Během první fáze vykonává slučování sekvenčních toků. Čeká dokud nedojde k jeho aktivaci prostřednictvím všech příchozích sekvenčních toků. Až bude evidován příchod skrze všechny příchozí sekvenční toky, tak započne zpracování uzlu implementací. Implementace během zpracování vybere následující uzly pro vytvoření. Vybírá všechny uzly, ke kterým vedou výstupní sekvenční toky.

C.3.17 Inclusive Gateway

Inkluzivní brána umožňuje rozdělovat sekvenční tok na základně podmínek. Při rozdělování sekvenčního toku jsou vyhodnoceny výrazy tvořící podmínky sekvenčních toků a všechny pozitivně vyhodnocené sekvenční toky budou provedeny. V případě neexistence pozitivně vyhodnocených sekvenčních toků bude proveden výchozí sekvenční tok. Současná verze systému nepodporuje slučování více sekvenčních toků skrze inkluzivní bránu.

Ukázka XML C.19:

```
<inclusiveGateway id="id" name="string" default="id_sequenceFlow">
</inclusiveGateway>
```

Obrázek C.19: Ukázka definice elementu `inclusiveGateway` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název brány.
- `default`
  - Reference na sekvenční tok, který nastane po neúspěšném vyhodnocení všech předchozích podmíněných sekvenčních toků.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro inkluzivní bránu se nazývá *InclusiveGateway* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/gateway.ts`. Implementace vyhodnotí výrazy pro všechny odchozí podmíněné sekvenční toky a následně vybere cílové uzly všech sekvenčních toků s pozitivním výsledkem. Prázdný nebo neexistující výraz je implementací vyhodnocen jako pozitivní. Pokud neexistuje žádný sekvenční tok s pozitivním výsledkem, tak budou vybrány cílové uzly výchozích sekvenčních toků.

C.3.18 Exclusive Gateway

Exkluzivní brána umožňuje řídit sekvenční tok na základně podmínek a vybírá jen jeden sekvenční tok. Při rozhodování o výběru následujícího uzlu dochází k vyhodnocení výrazů obsažených v podmíněných sekvenčních tocích. Odchozí podmíněné sekvenční toky jsou vyhodnocovány v pořadí, v jakém byli definovány při modelování procesu. Je zvolen první pozitivně vyhodnocený sekvenční tok. Současná verze systému nepodporuje slučování více sekvenčních toků skrze exkluzivní bránu.

Ukázka XML C.20:

```
<exclusiveGateway id="id" name="string" default="id_sequenceFlow">
</exclusiveGateway>
```

Obrázek C.20: Ukázka definice elementu `exclusiveGateway` v souboru BPMN.

Atributy:

- `id`
  - Unikátní identifikátor v rámci souboru BPMN.
- `name`
  - Název brány.
- `default`

– Reference na sekvenční tok, který nastane po neúspěšném vyhodnocení všech předchozích podmíněných sekvenčních toků.

Zásuvný modul s implementací uzlu:

Výchozí implementace pro exkluzivní bránu se nazývá *ExclusiveGateway* a obstarává ji zásuvný modul, jehož zdrojové kódy se nalézají v souboru `src/bpmnRunnerPlugins/gateway.ts`. Implementace vyhodnocuje postupně výrazy podmíněných sekvenčních toků v pořadí, ve kterém byly definovány. Vyhodnocování výrazů končí po prvním pozitivně vyhodnoceném výrazu. Cílový uzle patřící sekvenčnímu toku, který obsahuje pozitivně vyhodnocený výraz, je vybrán pro pokračování v provádění procesu. Prázdný nebo neexistující výraz je implementací vyhodnocen jako pozitivní. Pokud neexistuje žádný sekvenční tok s pozitivním výsledkem, tak budou vybrány cílové uzly výchozích sekvenčních toků.