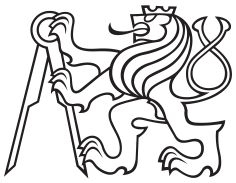
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Systém pro správu uživatelských úkolů v rámci procesní platformy

Bc. Jan Vanke

Vedoucí: Ing. Lukáš Zoubek
Obor: Otevřená informatika
Zaměření: Softwarové inženýrství
Leden 2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vanke** Jméno: **Jan** Osobní číslo: **478080**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Systém pro správu uživatelských úkolu v rámci procesní platformy

Název diplomové práce anglicky:

Human task management system for process automation platform

Pokyny pro vypracování:

1) Familiarize yourself with business process management topics, particularly on the question of process automation.
2) Research on the Camunda platform and a possible approach for user task management. Analyze how the user task management system can be part of the application using the Camunda platform. Examine and compare the possible methods, primarily from the point of their extensibility and the possibilities of their integration into new or existing systems.
3) Analyze the user task management system currently used within the Eprocesses system at FEE CTU.
4) Design the extension of the user task management system for more general usage in other systems. Primarily design how the user task forms will be rendered, how the new feature proposals will be managed within an organization, and how the user tasks will be implemented in Camunda Modeler.
5) Implement the user task management system and validate its usage within a proof-of-concept system based on Camunda platform designed by You.
6) Design and document the approach to how the system can be used in further applications.

Seznam doporučené literatury:

1. FREUND, Jakob a Bernd RÜCKER. Real-Life BPMN (4th edition): Includes an introduction to DMN. 4th edition. Camunda, 2019. ISBN 9781086302097.
2. European Association of Business Process Management (EABPM), Ed., Business Process Management Common Body of Knowledge - BPM CBOK: Leitfaden für das Prozessmanagement, Zweite. Wettenberg: Schmidt, 2009 [Online]. Available: http://www.goetz-schmidt-verlag.de/index.php?id=87
3. RUECKER, Bernd. Practical process automation: orchestration and integration in microservices and cloud native architectures. Beijing: O'Reilly, 2021. ISBN 9781492061458.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Lukáš Zoubek katedra softwarového inženýrství FIT

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

Datum zadání diplomové práce: **30.08.2022** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **19.02.2024**

Ing. Lukáš Zoubek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

. _____
Datum převzetí zadání

_____
Podpis studenta

Poděkování

Rád bych zde především poděkoval vedoucímu práce, Ing. Lukáši Zoubkovi, za jeho ochotu, pomoc a cenné rady.

Děkuji také Centru znalostního managementu za umožnění vzniku této práce a za podporu, které se mi od jeho zaměstnanců a stážistů dostalo.

V neposlední řadě chci poděkovat své rodině a manželce za trpělivost a obrovskou podporu, nejen při psaní této práce, ale i v průběhu celého studia.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a v souladu s Metodickým pokynem o dodržování etických principů pro vypracování závěrečných prací, a že jsem uvedl všechny použité informační zdroje.

V Praze, 10. ledna 2023

Bc. Jan Vanke

Abstrakt

Tématem této diplomové práce je business process management, konkrétně automatizace procesů a správa uživatelských úkolů s využitím procesní platformy Camunda. Teoretická část obsahuje rešerši těchto témat, včetně dalších konceptů a technologií jako je enterprise messaging, návrh rozhraní pomocí GraphQL a nástroj Elasticsearch pro datovou analýzu. V rámci praktické části jsou pak tyto koncepty a technologie aplikovány pro návrh obecného systému pro správu uživatelských úkolů, včetně návrhu souvisejících procesů pro řízení nových požadavků a použití rozšíření. Součástí praktické části je také implementace jednotlivých komponent v rámci proof-of-concept takového systému. Praktická část dále obsahuje analýzu systému pro správu uživatelských úkolů v rámci aplikace Eprocesy, ze kterého rozšíření vychází, a identifikaci hlavních problémů s tímto původním návrhem. Rozšíření je analyzováno z pohledu dalšího možného využití v rámci jiných systémů.

Klíčová slova: BPM, automatizace procesů, BPMN, uživatelské úkoly, formuláře

Vedoucí: Ing. Lukáš Zoubek

Abstract

This diploma thesis deals with Business Process Management; more precisely, process automation and user task management. The theory section describes the topic, including additional concepts and technologies such as Enterprise Messaging, GraphQL service interfaces, and the Elasticsearch data analysis tool. The empirical section applies these concepts to design a general system for user task management, including related processes for new feature proposal management and use of the extension. This section also includes the implementation of the components in a proof of concept of the system extension. The design of the new extension is based on the analysis of the former user task management system used in the Eprocesses application. Analysis of the former approach and its primary design flaws are also included. The new extension is analyzed with regard to deployment in other systems outside the Eprocesses application context.

Keywords: BPM, process automation, BPMN, user tasks, forms

Title translation: Human task management system for process automation platform

Obsah

Obrázky

Kapitola 1
Úvod

Následující část představuje motivaci, hlavní cíle a strukturu této práce.

1.1 Předmluva

Řízení a automatizace procesů mohou přinést společnostem mnoho benefitů, od zvýšení efektivity, přes zvýšení konkurenceschopnosti po snížení nákladů. Tato práce se zaměřuje na jednu z částí řízení a automatizace procesů, a to konkrétně na řízení a správu uživatelských úkolů. Ve snaze omezit množství využívaných systémů, a tím i omezit nároky na dovednosti zaměstnanců, hledají společnosti řešení umožňující jednotným způsobem spravovat a vykonávat uživatelské úkoly. Takové řešení zároveň musí umožňovat audit a sledování práce tak, aby ji bylo možné dále zefektivnit. Aktuálně na trhu open-source řešení chybí způsob, jakým efektivně spojit automatizaci procesů a jednotnou správu uživatelských úkolů z více systémů, bez nutnosti vyvíjet vlastní řešení.

1.2 Motivace a cíle

Centrum znalostního managementu (CZM) na Fakultě elektrotechnické ČVUT dlouhodobě usiluje o digitalizaci a automatizaci procesů v rámci fakulty. Příkladem může být projekt Eprocesy, který za využití platformy Camunda a využití principů business process managementu usiluje o digitalizaci agend studijního oddělení. CZM vyvíjí několik dalších systémů, se kterými primárně pracují zaměstnanci fakulty. Hlavním cílem této práce je využít existující část systému Eprocesy pro vykreslování formulářů a navrhnout obecný systém pro správu uživatelských úkolů, který bude možné využít nejen v prostředí ČVUT FEL.

1.3 Struktura

Práce je strukturována do teoretické a praktické části a závěru. Níže je popsán obsah jednotlivých sekcí. Součástí práce jsou také přílohy obsahující zdrojové kódy.

1.3.1 Teoretická část

Součástí teoretické části této práce je rešerše zabývající se principy business process managementu. V rámci teoretické části je také popsána platforma Camunda a její komponenty. Obě části kladou důraz, vzhledem k tématu práce, především na správu uživatelských úkolů. Závěrem teoretické části práce jsou popsány další technologie a principy důležité pro praktickou část, a to především Enterprise Messaging, GraphQL a systém Elasticsearch.

1.3.2 Praktická část

Praktická část sestává z analýzy současného stavu správy uživatelských úkolů v rámci systému Eprocesy. Jsou zde mimo jiné identifikovány hlavní problémy s aktuálním návrhem. Dále obsahuje analýzu a design rozšíření systému pro správu uživatelských úkolů pro širší použití, mimo aplikaci Eprocesy. Součástí návrhu jsou také procesy pro vývoj formulářů v rámci procesů nebo postup pro řízení nových požadavků na systém. Dle návrhu systému je implementován proof-of-concept tohoto systému. Jsou zde i popis jednotlivých komponent, včetně postupu implementace a jejich použití. Závěrem praktické části je popsáno, jakým způsobem je možné navržené rozšíření využít v rámci dalších systémů.

1.3.3 Závěr a přílohy

V závěru jsou shrnuty výsledky práce a je hodnoceno naplnění jejích cílů. Součástí závěru práce jsou také přílohy obsahující dodatečné ukázky, diagramy a modely.

Část I

Teoretická část

Kapitola 2
Business Process Management

V prostředí stále se rozšiřující digitalizace a automatizace narůstá význam Business Process Management (BPM). BPM umožňuje skrze systematický přístup pro práci s byznys procesy naplnit strategické cíle a optimalizovat procesy v rámci organizací.[1, 2]

Tato kapitola obsahuje úvod do problematiky Business Process Management. V první sekci 2.1 je obsažena definice byznys procesu a jeho praktické příklady. V druhé sekci 2.2 jsou popsány způsoby modelování procesů relevatní pro praktickou část této práce, zvláště pak notace BPMN 2.0. V další sekci 2.3 je poté popsán životní cyklus BPM a jeho vlastnosti. Nakonec, v poslední sekci 2.4 je popsáno, jaké nástroje a metodologie se využívají pro automatizaci procesů, a jak se automatizace v praxi dosahuje.

Business Process Management typicky sestává ze tří hlavních částí:[3]

1. Specifikace a modelování procesů — metodologie a modely používané pro zachycení procesů.
2. Process reengineeering — metologie pro optimalizaci procesů.
3. Automatizace a implementace procesů — použití technologií a metodologií pro koordinaci informačních systémů a lidských účastníků k implementaci, rozvržení, vykonávání a dokončení procesů, tak jak jsou popsány pomocí specifikace nebo modelu.

Z výše popsaných třech částí jsou pro praktickou část práce relevatní především oblasti modelování procesů a automatizace, proto je na ně v rámci teoretické části kladen důraz. Process reengineeering není v rámci této práce dále rozveden, vzhledem k tématu této práce.

2.1 Byznys Proces

Proces nebo také někdy workflow označuje sekvenci kroků — úkolů vykonávaných za očekávaným výsledkem, také je typicky vykonáván opakovaně. V rámci procesů spolu kooperují jak osoby, tak IT systémy nebo dnes i fyzická zařízení za využití konceptů Internet of Things. Procesy mají přímý vliv na to, jak je společnost vnímána a na její úspěšnost, zároveň určují odpovědnosti

a náplň práce jejich zaměstnanců. Současně se stále více zvyšují nároky na digitalizaci, integrace a práci s daty v rámci procesů uvnitř organizací. To jakým způsobem jsou procesy navrženy a vykonávány má vliv, jak na kvalitu služby, tak na efektivitu se kterou jsou vykonávány.[2, 4]

Typickými příklady procesů v rámci organizací mohou být procesy označované jako *order-to-cash*, které jsou zahájeny klientem, který si objednává nějakou službu nebo produkt a končí doručením služby nebo zboží klientovi a zaplacením objednávky. Dalším typem je například *application-to-approval*, který začíná žádostí klienta a končí jejím schválením nebo zamítnutím, konkrétním příkladem tohoto procesu může být například žádost o dovolenou. V rámci organizací se však můžeme setkat i s procesy například z prostředí výrobních linek, kde v rámci procesu spolupracují jak pracovníci, tak různé senzory a fyzická zařízení.[2, 4, 5]

Cílem této sekce bylo nastínit s jakými procesy se lze v rámci organizací setkat. V rozsahu této práce není kompletní rešerše druhů procesů.

2.2 Modelování

Pro zachycení procesů se využívá množství různých způsobů modelování. Tato sekce je výhradně zaměřena na notaci BPMN 2.0 a související notaci DMN 1.2. Notace BPMN 2.0 i DMN 1.2 jsou standardizovány a podporovány širokou škálou nástrojů a současně umožňují přímé spouštění modelů. Jako jeden z předpokladů této práce je využití nástroje, který podporuje BPMN a DMN, právě proto je tato sekce zaměřena pouze na tyto techniky modelování. Současně je pro účely této práce dostatečná pouze základní znalost těchto notací. Kompletní specifikaci standardů je možné nalézt na webových stránkách standardizační organizace Object Management Group[1].

Samotnému modelování procesů typicky předchází analýza, která umožňuje analytikovi danému procesu porozumět. K získání detailnější znalosti procesu se využívají různé metodologie, které nejčastěji zahrnují rozhovory s experty na daný proces.[3]

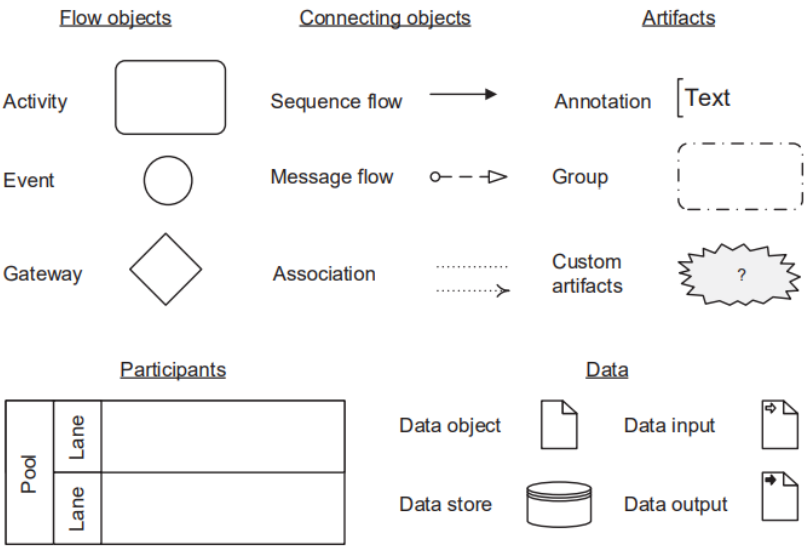
2.2.1 BPMN

Procesní diagramy v BPMN se skládají ze symbolů zobrazených v obrázku 2.1. Jedná se o zjednodušený seznam zobrazující hlavní typy značek, kompletní seznam je možné nalézt ve specifikaci BPMN 2.0 na webových stránkách standardizační organizace OMG. Každý BPMN 2.0 diagram je reprezentován pomocí jazyka XML.

BPMN poskytuje seznam značek a pravidla pomocí nichž je možné symboly skládat — syntax. Dále jednotlivým symbolům a seskupením symbolů přiřazuje význam — sémantiku. Vypovídající hodnota a cíle modelu, například komunikace IT s byznys oddělením, může v určitých momentech převážit nad syntaktickou a sémantickou správností modelu. Právě proto vznikají modely

[1] https://www.omg.org/

procesů typicky na různých úrovních. Záleží co je potřeba na dané úrovni komunikovat a kdo je cílová osoba.[4]

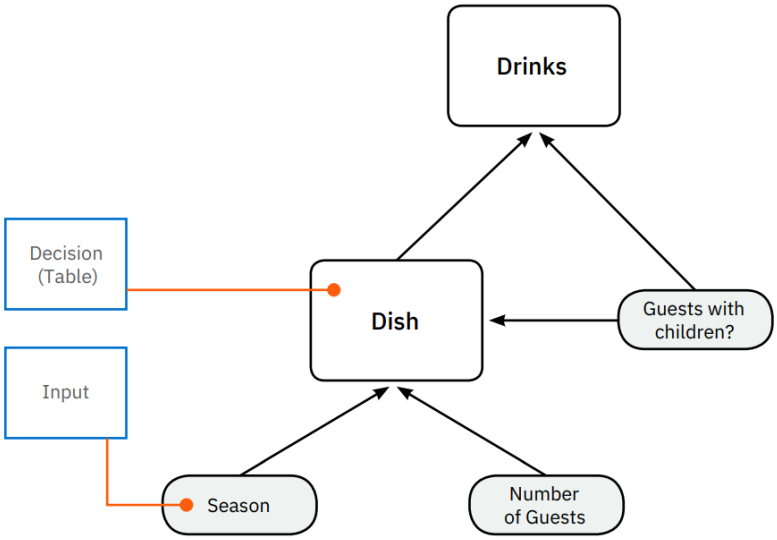


Obrázek 2.1: Základy BPMN notace, zdroj [4]

Základem každého procesu jsou aktivity (activities). Jedná se o atomické jednotky práce vykonané člověkem nebo systémem, příkladem může být vyplnění formuláře, odeslání emailu nebo vygenerování dokumentu. Dalším prvkem v rámci procesů jsou události (events). Ty mohou vzniknout na základě nějakého uživatelského vstupu, například podání žádosti. Mohou také vzniknout po určitém čase, na základě vyhodnocení změny stavu nebo na základě výjimky. Událostí může být například přijetí objednávky, nebo uplynutí lhůty na vyzvednutí zásilky. Třetí důležitou složkou procesů jsou rozhodovací body (gateways). V těchto bodech dochází k vyhodnocení nějaké podmínky, nebo čekání na událost a následnému větvení procesu.[2, 4]

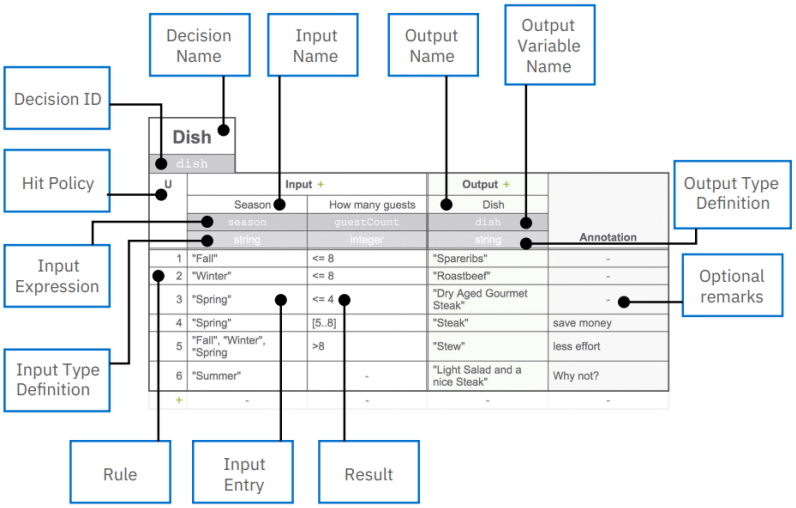
2.2.2 DMN

V této podsekci je pro úplnost zmíněn standard, který je často využíván právě v kombinaci s BPMN a to Decision Model and Notation (DMN). Tento standard je využíván pro modelování komplexní rozhodovací logiky na základě vstupů. Stejně jako u BPMN lze i DMN reprezentovat pomocí jazyka XML. Rozhodovací logika se v DMN modeluje jako Decision Requirements Graph (DRG) zobrazený jako jeden nebo více Decision Requirements Diagrams (DRDs), jak je znázorněno v obrázku 2.2. Stejně jako BPMN je i DMN možné vykonat pomocí specializovaného DMN software — decision execution engine, který může být součástí komplexnějšího řešení pro automatizaci procesů, o těchto systémech dále pojednává sekce 2.4.[6]



Obrázek 2.2: DRD (Decision Requirements Diagrams), zdroj [6]

Každé rozhodnutí je poté modelováno jako tabulka se vstupními a výstupními hodnotami, jak je znázorněno v tabulce 2.3.[4]

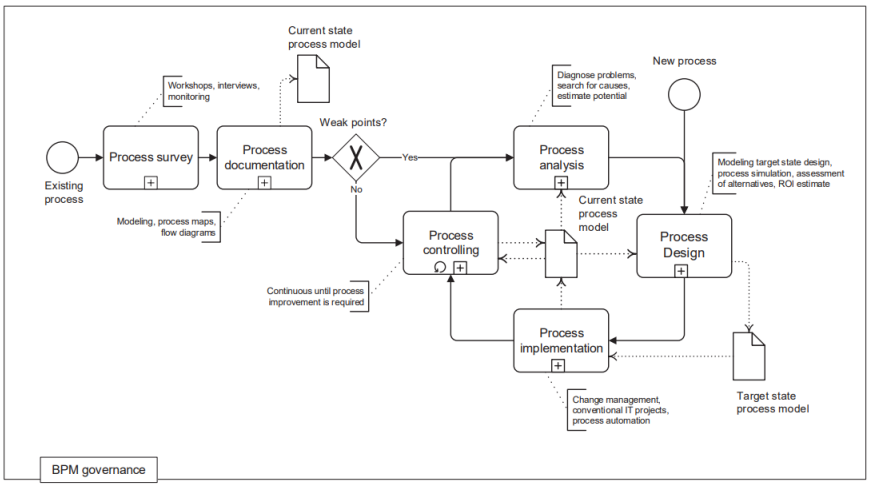


Obrázek 2.3: Decision tabulka, zdroj [6]

Součástí DMN standardu je také výrazový jazyk Friendly Enough Expression Language (FEEL), který se v rozhodovacích tabulkách používá pro vyhodnocení rozhodnutí.[6]

2.3 BPM Životní Cyklus

Na diagramu 2.4 je zobrazen životní cyklus BPM. Začíná potřebou zadokumentovat existující nebo nově vznikající proces. Jak je z diagramu patrné, jedná se o iterativní postup, který má za cíl kontinuálně proces monitorovat, vyhodnocovat a optimalizovat. Koordinací odpovědných stran, aplikovaných metod a nástrojů pro podporu BPM je tímto možné docílit průběžného zlepšování procesů — continuous process improvement.[4, 7]

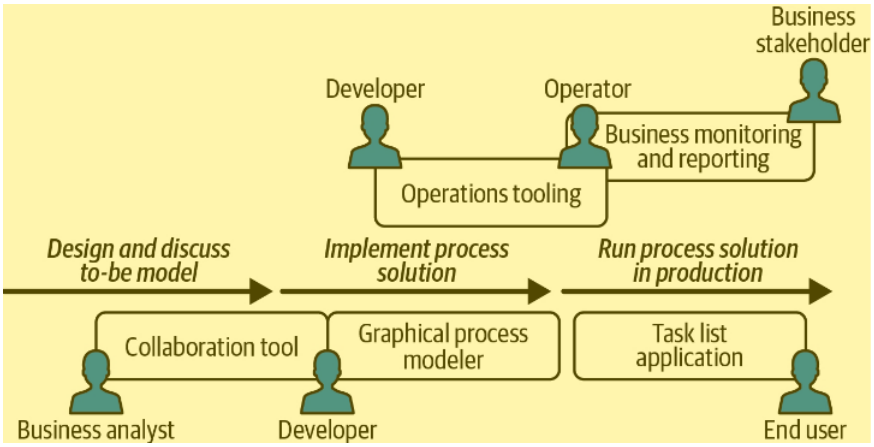


Obrázek 2.4: Camunda BPM životní cyklus, zdroj [4]

V diagramu jsou také zaznamenány artefakty, které v rámci procesu vznikají. Jak již bylo zmíněno v sekci 2.2 někdy diagramy vznikají s různou úrovní detailu.

V každé fázi životního cyklu BPM jsou potřeba různé nástroje, od modelování přes monitorování, administraci až po provoz procesů v produkci. Každý nástroj má různé uživatele a využívá se v jiné fázi BPM. Na obrázku 2.5 je možné vidět jednotlivé fáze a odpovídající druh nástroje.[4, 8]

Softwarové platformy, které slouží pro podporu jednotlivých fází BPM jsou někdy označovány jako Business Process Management Systems (BPMSs).[9]



Obrázek 2.5: BPM nástroje, zdroj [4]

Jednotlivé fáze nejsou vzhledem k rozsahu práce dále popsány. Další sekce 2.4 se především zaměřuje na fázi implementace a tedy i automatizace procesů.

2.4 Automatizace procesů

Tato sekce se zabývá automatizací byznys procesů a rozhodnutí pomocí software. Pro spuštění BPMN a DMN modelů je možné využít takzvaný engine (workflow nebo decision). Engine je software, který nejen umí spouštět BPMN a DMN modely, ale zároveň nám typicky umožňuje:[8]

1. Koordinovat SW komponenty, rozhodování, lidské úkoly, boty pro robotickou automatizaci procesů a fyzická zařízení.
2. Kontrolovat tok dat v procesech.
3. Kontrolovat dodržování modelovacího standardu.
4. Udržovat stav a data procesů.
5. Řídit transakce v procesech.

Jedním z představitelů takového enginu je open-source platforma Camunda, která je dále popsána v kapitole 3.[4, 8]

2.4.1 Software

Procesní engine umožňuje, jak již bylo zmíněno v úvodu k automatizaci procesů, koordinovat SW komponenty. Jedná se o všechny SW komponenty a systémy, které jsou schopné s procesním enginem komunikovat.

Integrace

Integrace softwarových komponent může probíhat různými způsoby, z nichž každý má svá specifika. Jako nejčastější příklady způsobů komunikace mezi systémy je možné úvést následující:[10]

1. Komunikace za využití HTTP protokolu — příklady může být REST[2], webové služby (SOAP)[3]
2. Asynchronní komunikace za využití jiného protokolu než HTTP — Messaging systémy (Apache Kafka, RabbitMQ, Amazon Simple Queue Service), příkladem protokolu může být AMQP[4].
3. gRPC[5] — open-source RPC framework, komunikace probíhá přes binární protokol založený na protokolu TCP.

Integrace pomocí messaging systémů je detailněji popsána v sekci 4.1, synchronní komunikace pomocí HTTP protokolu, za využití grafového dotazovacího jazyka GraphQL poté v sekci 4.2.

Architektura

Architektura systému do kterého je, nebo má být, procesní engine integrován ovlivňuje jakým způsobem budou komponenty komunikovat, a jak budou navrženy samotné procesy. Rozhodování o architektuře ovlivňuje celá řada faktorů, z nichž hlavní jsou uvedeny na následujícím seznamu:[4, 10]

- Nefunkční požadavky na řešení — například požadavky na výkon, škálovatelnost, bezpečenost a dostupnost řešení.
- Funkční požadavky — rozsah a charakter funčkních požadavků.
- Infrastruktura, na které bude software provozován.
- Struktura a zkušenosti týmu — struktura systému podle Conway's law odráží strukturu týmu nebo organizace.
- Omezení a předpoklady — časový rámec projektu, rozpočet a předchozí technologické nebo návrhové rozhodnutí, které musí projekt splňovat.

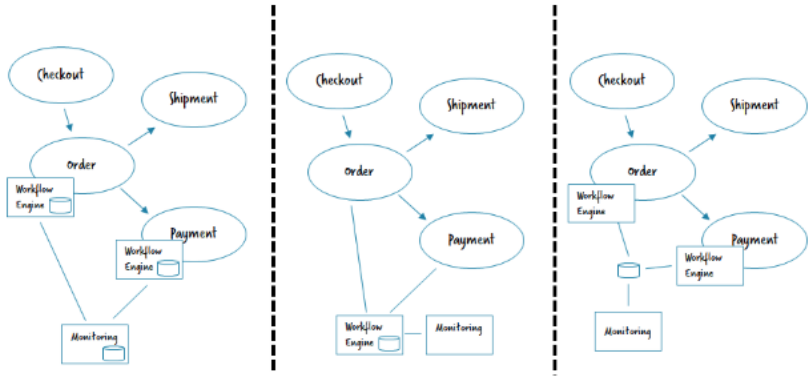
Architekturu řešení ovlivní způsob, jakým lze procesní engine Camunda provozovat, existují tři základní způsoby:[11]

- Samostatný (Remote) Engine — engine je poskytován jako služba po síti.
- Engine sdílený pro kontejner (Aplikační server, Servlet kontejner)

[2] Representational State Transfer
[3] Simple Object Access Protocol
[4] Advanced Message Queuing Protocol
[5] Google Remote Procedure Call

- Embedded Engine — součástí procesní aplikace jako knihovna.

Provozovat procesní engine je pak možné více způsoby, které jsou znázorněny na diagramu 2.6. Pro provoz je možné využít konceptu decentralizovaného procesního enginu, nebo využít centrálně spravovaného enginu, který je sdílený pro celou organizaci nebo část organizace. Oba přístupy je možné do určité míry kombinovat a například sdílet procesní engine jen pro některé týmy a dalším nechat autonomii ve správě vlastního enginu. Další možností jak oba přístupy kombinovat je využít centrální engine, který je využíván jako decentralizovaný. V tomto přístupu se využívá engine v každé aplikaci jako knihovna zvlášť, za využití společné databáze.[4, 12]



Obrázek 2.6: Možnosti provozu procesního engine, zdroj [12]

Mezi základní vlastnosti centralizovaného přístupu se řadí:[12]

- Méně nákladný provoz.
- Monitoring je dostupný out-of-the-box.
- Centrální engine může představovat single-point-of-failure.
- Jednotlivé procesní aplikace jsou vzájemně méně izolované.

Mezi základní vlastnosti distribuovaného přístupu se řadí:[12]

- Aplikace jsou navzájem izolované.
- Autonomie jednotlivých aplikací.
- Větší náklady na provoz — každý tým spravuje svůj procesní engine.
- Distribuovaný monitoring je náročnější.

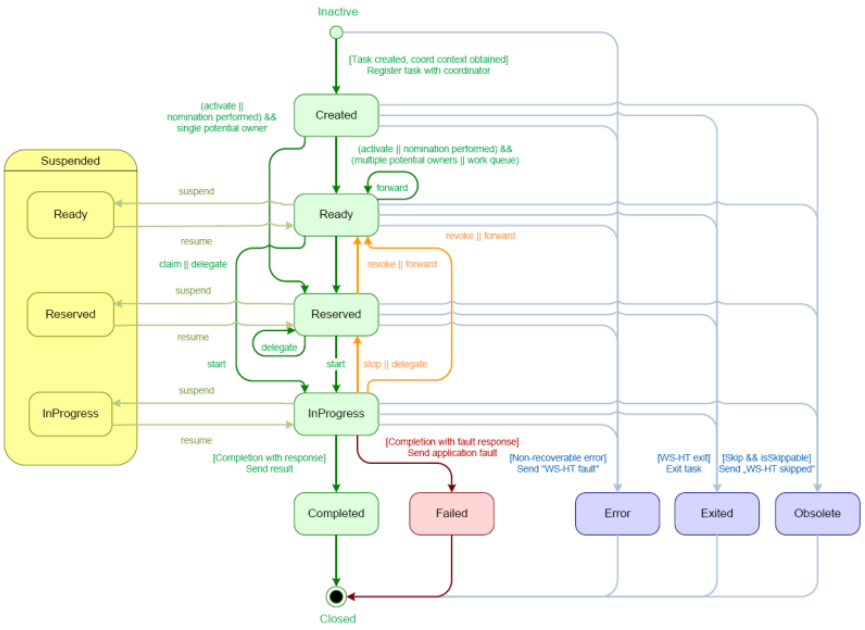
2.4.2 Uživatelské úkoly

Procesní automatizace nutně neznamená, že jsou všechny aktivity automatizované. Další úlohou procesního enginu je takzvaný — human workflow

management. Workflow lze definovat jako kolekci úkolů, které jsou organizovány k dokončení nějakého byznys procesu. Procesní engine má tedy odpovědnost za informování odpovědných osob a úkolech, které mají v rámci procesu přiřazené, zpracovává výsledek, když uživatel úkol splní a umožňuje spravovat jeho životní cyklus.[3]

Životní cyklus úkolů

Většina procesních enginů umožňuje spravovat životní cyklus uživatelských úkolů. Příklad takového životního cyklu je možné vidět na diagramu 2.7. Další příklad lze nalézt v kapitole o platformě Camunda 3.



Obrázek 2.7: Příklad životního cyklu user tasku v rámci JBoss BPM řešení, zdroj [13]

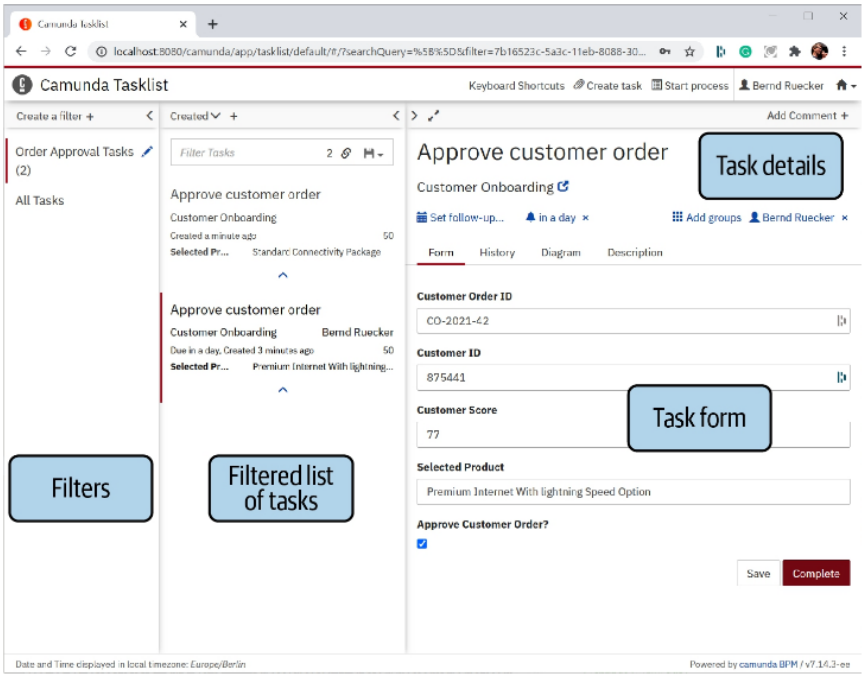
Příklady operací, které by měl procesní engine umožňovat vykonávat nad úkolem jsou následující:

- Vytvoření a odstranění úkolu.
- Přiřazení — přiřazení řešení úkolu na osobu (řešitele).
- Vyřešení úkolu.
- Delegování — zpracování části úkolu jinou osobou.
- Přiřazení kandidátských skupin nebo uživatelů — přiřazení potencionálních řešitelů úkolu.
- Prioritizace úkolů — možnost přiřadit prioritu k úkolům pro přednostní dokončení úkolů.

Nástroje mohou poskytovat i další podporu jako jsou například notifikace, eskalace nebo řešení dovolených. Zatímco je samozřejmě možné tyto scénáře namodelovat přímo v procesu, zanesly by do něj zbytečnou komplexitu a je lepší se jim vyhnout a řešit je na úrovni BPMS.[4]

Uživatelské rozhraní

Součástí odpovědností BPMS je i umožnit práci s úkolem pomocí uživatelského rozhraní. Nástroje poskytují různé možnosti, jednou z nich je vlastní uživatelské rozhraní poskytované přímo dodavatelem procesního engine. Příkladem takového uživatelského rozhraní je Camunda Tasklist znázorněný na obrázku 2.8. Další možností je, že procesní engine poskytuje API pomocí kterého je možné si takové rozhraní vyvinout podle vlastních potřeb nebo ho integrovat do existujícího systému.



Obrázek 2.8: Ukázka Camunda Tasklist, zdroj [4]

Na závěr je potřeba zmínit, že většina BPMS má v dnešní době do určité míry i podporu pro koordinaci a automatické zpracování uživatelských úkolů pomocí Robotic Process Automation (RPA) botů. Boti jsou softwarové komponenty schopné vykonávat část manuální práce napříč SW systémy, aniž by bylo potřeba existující systémy měnit. Tento fakt může v určitých případech ušetřit zdroje na automatizaci.

2.4.3 Shrnutí kapitoly

V rámci této kapitoly byl představen pojem byznys procesu a byl zde uveden úvod do tématiky Business Process Management. Vzhledem k tématu

praktické části této práce byl kladen důraz na téma automatizace procesů, zvláště pak práce s uživatelskými úkoly. Pro uvedení do kontextu práce zde byly také představeny základy modelování procesů pomocí notace BPMN 2.0 a základní životní cyklus BPM.

Kapitola 3
Camunda

Tato kapitola představuje komplexní open-source platformu Camunda 7. Platforma poskytuje nástroje využitelné ve všech fázích Business Process Management, které byly blíže popsané v přechozí kapitole 2. Kapitola se zaměřuje především na popis jednotlivých komponent platformy Camunda a způsoby, jakými je možné platformu provozovat. Kapitola se také zaměřuje na to, jak platforma umožňuje pracovat s uživatelskými úkoly. Závěrem kapitoly jsou pro úplnost uvedeny dvě krátké sekce, jedna zabývající se použitím procesního engine jako multi-tenantního systému a druhá popisující, jak je možné u procesního engine zaručit vysokou dostupnost.

V době vzniku této práce je již dostupná novější verze procesního engine Camunda 8 (Zeebe), která však nemá podporu pro větší část funkcí, které poskytuje Camunda ve verzi 7. Camunda 8 se také výrazně liší v architektuře procesního engine a v licenčních podmínkách. Vzhledem k těmto faktům je v rámci této práce dále jako Camunda označována verze 7 a popis nové verze není součástí této práce.

3.1 Přehled

Součástí platformy jsou komponenty podporující všechny fáze životního cyklu BPMN popsaného blíže v 2.3. Jedná se následující komponenty:[11]

- Process Engine Core — knihovna napsaná v jazyce Java, zodpovědná za spouštění modelů v BPMN 2.0 a DMN 1.3 standardech. Využívá relační databázi pro ukládání stavu. Procesní engine poskytuje Java API pomocí kterého je ho možné ovládat[1].
- Modeler — modelovací nástroj pro standardy BPMN 2.0 a DMN 1.3.
- Cawemo — modelovací nástroj pro standardy BPMN 2.0 a DMN 1.3, který umožňuje souběžnou spolupráci nad modely.
- Tasklist — webová aplikace pro správu uživatelských úkolů, která umožňuje lidským účastníkům procesů vykonávat jejich práci.

[1] Dokumentace Java API — `https://javadoc.io/doc/org.camunda.bpm/camunda-engine/latest/index.html`

- Cockpit — webová aplikace pro monitoring a provoz procesů. Cockpit umožňuje sledovat a ovlivňovat stav procesů a řešit případné incidenty.
- Admin — webová aplikace umožňující spravovat skupiny, uživatele a oprávnění.
- REST API — umožňuje ovládat procesní engine pomocí rozhraní. Tasklist, Cockpit, Admin komunikují s procesním enginem právě pomocí tohoto REST API.

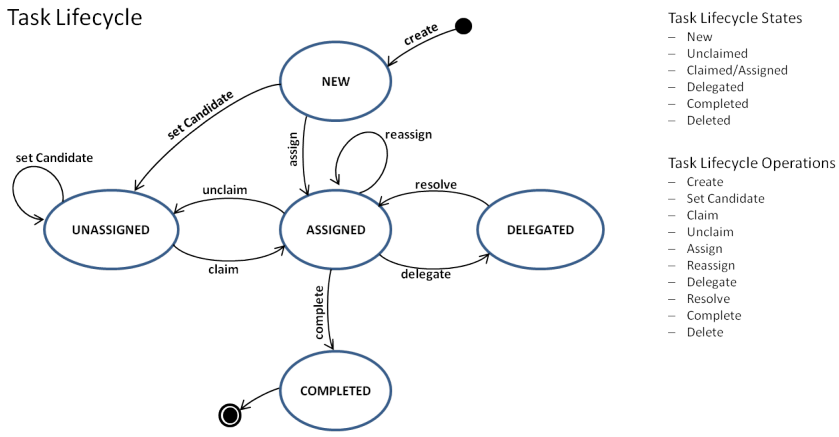
Procesní engine poskytuje integrace pro vývoj Java Enterprise aplikací, například pro populární open-source Spring Framework nebo CDI (Context and Dependency Injection). CDI je Java EE 6 standard pro Dependency Injection.[11]

Pro dodatečnou funkcionalitu nad rámec BPMN notace poskytuje Camunda řadu rozšíření BPMN standardu[2]. Příkladem rozšíření může být element `taskListener`, který umožňuje navázat vlastní logiku na události týkající se uživatelských úkolů v procesu, například vytvoření nového úkolu.[11]

3.2 Správa uživatelských úkolů

V rámci podsekce 2.4.2 bylo popsáno jak souvisí procesní automatizace a správa uživatelských úkolů, včetně ukázky životního cyklu uživatelského úkolu. V této sekci je popsáno jak se pro správu uživatelských úkolů dá využít procesní platforma Camunda.

Životní cyklus uživatelského úkolu vzniklého v rámci platformy Camunda je znázorněn na diagramu 3.1.



Obrázek 3.1: Životní cyklus user tasku v rámci Camunda platformy, zdroj [11]

V rámci životního cyklu je potřeba ujasnit základní pojmy spojené s životním cyklem úkolů v rámci platformy Camunda:

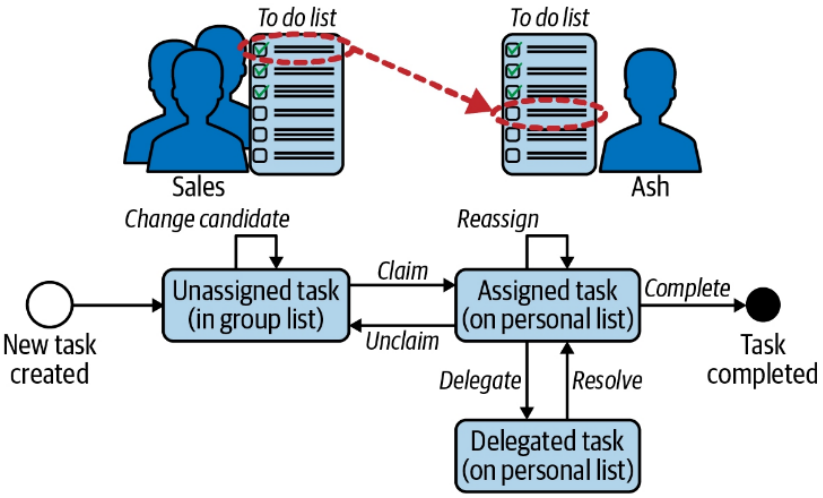
[2]Dokumentace rozšíření — `https://docs.camunda.org/manual/latest/reference/bpmn20/custom-extensions/`

- Řešitel (angl. assignee) úkolu — osoba odpovědná za vykonání a splnění daného úkolu.

- Operace delegace (angl. delegate) — operace kdy řešitel úkolu deleguje vykonání části práce na konkrétní osobu. Po vykonání části práce je úkol předělen zpět na řešitele, který je odpovědný za jeho dokončení.

- Kandidátská skupina a kandidátský uživatel (angl. candidate group and user) — osoba nebo člen skupiny může převzít daný úkol (angl. claim task) a stát se tak řešitelem.

Na diagramu 3.2 je znázorněn rozdíl mezi kandidátskou skupinou, řešitelem úkolu a operací delegace.

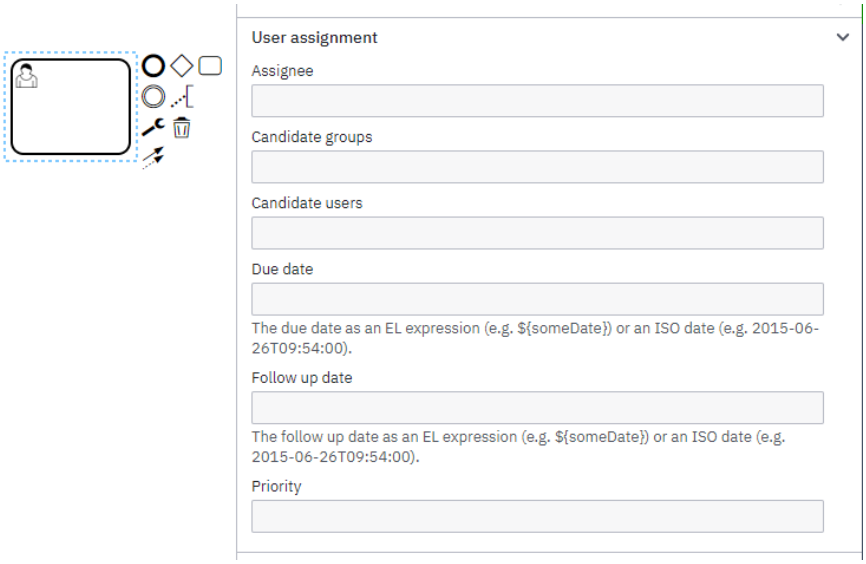


Obrázek 3.2: Rozdíl mezi kandidátskou skupinou a řešitelem úkolu, zdroj [4]

Platforma Camunda za využití rozšíření BPMN notace a webové aplikace Camunda Tasklist poskytuje dodatečnou podporu pro následující funkcionalitu spojenou s uživatelskými úkoly:

- Notifikace o úkolech
- Eskalace a termíny úkolů
- Nastavení priority úkolů.

Tato dodatečná funkcionalita se konfiguruje jako atributy uživatelských úkolů v rámci BPMN modelu. Ukázku konfigurace pomocí vývojového nástroje Camunda Modeler je možné vidět na obrázku 3.3.



Obrázek 3.3: Camunda Modeler — konfigurace atributů uživatelského úkolu, zdroj: autor

3.2.1 Struktura uživatelského úkolu

V této podsekci je uvedena základní struktura uživatelského úkolu v rámci procesního enginu Camunda. Následující seznam obsahuje význam jednotlivých atributů:[11]

- id — Identifikátor konkrétní instance úkolu.
- assignee — Řešitel úkolu.
- name — Název úkolu.
- created — Datum a čas vytvoření úkolu.
- delegationState — Stav delegace úkolu (čekající nebo vyřešený).
- description — Popis úkolu.
- due — Termín do kdy má být úkol dokončen (datum a čas).
- executionId — Identifikátor odpovídající dané nadřazené procesní instanci.
- followUp — Termín připomenutí úkolu (datum a čas).
- owner — Vlastník úkolu, využívaný hlavně v případě delegace.
- priority — Priorita úkolu.
- processDefinitionId — Identifikátor procesní definice.
- processInstanceId — Identifikátor konkrétní procesní instance.

- taskDefinitionKey — Identifikátor definice úkolu (unikátní v rámci jednoho .bpmn souboru)
- suspended — Indikátor, zda-li je úkol pozastaven.
- tenantId — Identifikátor tenanta v případě že jsou využívány.

Dále je uveden příklad takového uživatelského úkolu.

```
{
    "id":"anId",
    "name":"Validate client request",
    "assignee":"anAssignee",
    "created":"2013-01-23T13:42:42.000+0200",
    "due":"2013-01-23T13:49:42.576+0200",
    "followUp":"2013-01-23T13:44:42.437+0200",
    "delegationState":"RESOLVED",
    "description":"Check user proposal.",
    "executionId":"anExecution",
    "owner":"anOwner",
    "priority":42,
    "processDefinitionId":"aProcDefId",
    "processInstanceId":"aProcInstId",
    "taskDefinitionKey":"aTaskDefinitionKey",
    "suspended": false,
    "tenantId":"aTenantId"
}
```

3.2.2 Vykonávání úkolů a formuláře

Do správy uživatelských úkolů spadá také jejich vykonávání pomocí grafického rozhraní. Jak již bylo v předchozích sekcích zmíněno, Camunda 7 poskytuje výchozí uživatelské rozhraní ve formě webové aplikace Camunda Tasklist. Zároveň platforma umožňuje vytvořit vlastní grafické rozhraní, například pomocí zmíněného Camunda REST API nebo vlastního rozhraní postaveného nad Java API poskytované procesním enginem. Vlastní grafické rozhraní může být případně integrováno do existujícího systému.

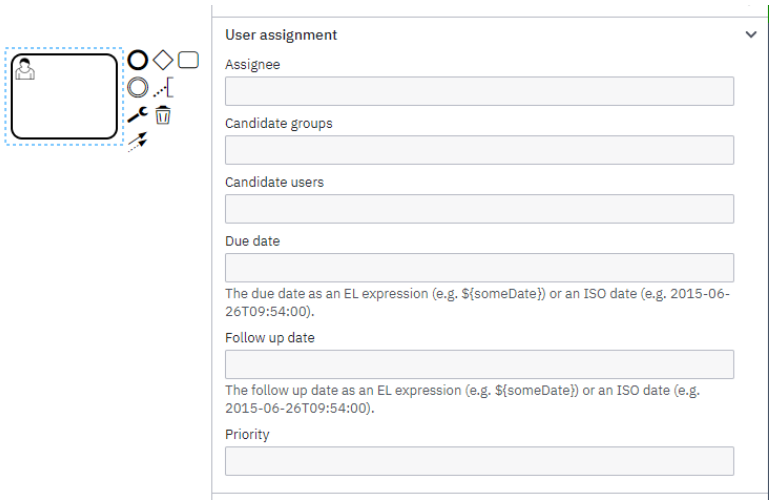
Camunda Tasklist

Výchozí aplikace poskytovaná přímo platformou Camunda 7. Mezi základní funkcionalitu poskytovanou webovou aplikací Tasklist se řadí:

- Filtrování úkolů
- Vytváření nových persistentních filtrů úkolů
- Spouštění procesů
- Vykonávání úkolů — zobrazení formulářů

Formuláře úkolů lze zobrazovat několika způsoby:

- Přesměrování do externí aplikace a vykreslení formuláře pomocí definice atributu `formKey` u user task elementu.
- Generované formuláře — HTML formulář vykreslený z definice přímo v BPMN souboru. Definice je uvedena přímo jako rozšíření standardu BPMN. Camunda Tasklist poté umí daný formulář vyrenderovat. Ukázku konfigurace v Camunda Modeler je možné vidět na obrázku 3.4.
- Camunda Forms za použití knihovny form-js — společně s BPMN souborem se dodá ještě JSON definice formuláře, kterou je možné vygenerovat automaticky z WYSIWYG editoru formulářů.



Obrázek 3.4: Camunda Modeler — konfigurace generovaného formuláře, zdroj: autor

V době vzniku této práce jsou oba přístupy pomocí generovaných formulářů velice limitované a stačí pouze na splnění základních požadavků na formuláře. Pro komplexnější scénáře nelze generované formuláře použít. Příklady složitějších požadavků jsou uvedeny v následujícím seznamu:

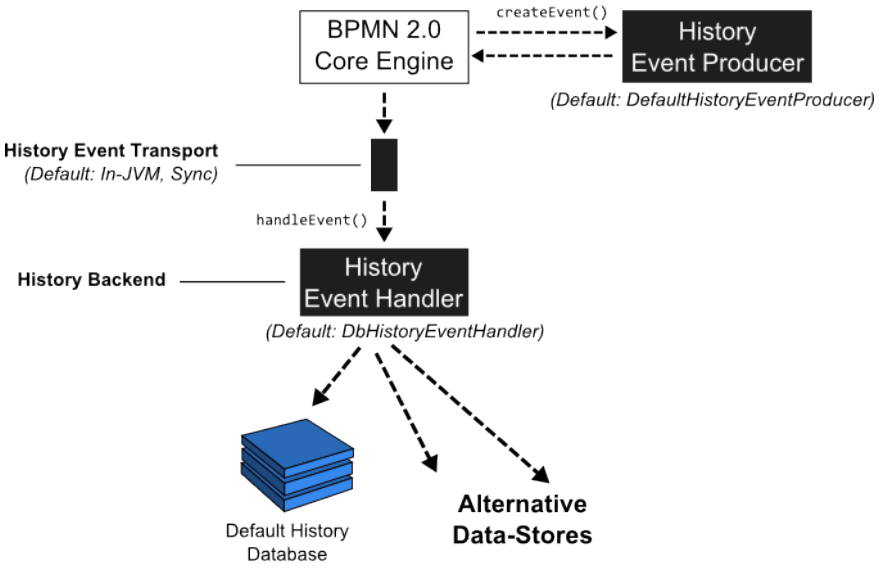
- Podmíněné formátování na základě hodnot jiných prvků.
- Možnost ovlivnit rozložení formulářových prvků.
- Přidání složitějších validačních pravidel na hodnoty vstupu.
- Jiné než základní vstupní prvky (například tabulky).

3.2.3 Stav a historie úkolů

Procesní engine udržuje stav úkolu v relační databázi. Zároveň ukládá historii stavu a provedené operace nad uživatelským úkolem do databáze historie.

Způsoby jakými lze ukládat auditní data o úkolech jsou popsány v této podsekci.

Architektura historie v platformě Camunda je znázorněna na diagramu 3.5.



Obrázek 3.5: Camunda History architektura, zdroj: [11]

Zpracování události v rámci procesního engine proběhne v následujících krocích:

1. V rámci procesního engine vznikne událost[3], například vznik nové procesní instance, dokončení uživatelského úkolu nebo změna procesních proměnných. Jádro procesního engine volá History Event Producer.
2. History Event Producer vytváří a vrací objekt události, která vznikla v rámci procesního engine (History Event).
3. Vytvořený History Event předává procesní engine History Event Handler, metodě `handle(HistoryEvent)`.
4. Ve výchozím nastavení History Event Handler uloží History Event do History databáze, která je oddělená od databáze, kterou Camunda používá pro běžný provoz procesů.
5. Je možné poskytnout vlastní zpracování History Eventu, například provést transformaci dat a uložit je do vlastní databáze, nebo je odeslat do messaging systému.

Pro zpracování historických událostí jsou tedy tři základní možnosti:[11]

[3]Celý výčet událostí, na které Camunda umí reagovat je uveden v rámci dokumentace - `https://docs.camunda.org/javadoc/camunda-bpm-platform/7.18/org/camunda/bpm/engine/impl/history/event/HistoryEventTypes.html`

- Výchozí zpracování historických událostí pomocí uložení do relační databáze.
- Vlastní zpracování historických událostí společně s uložením do relační databáze. Příkladem vlastního řešení může být odesílání do messaging systému nebo ukládání do vlastní databáze nad kterou je možné provádět rychlé vyhledávání.
- Kombinace výchozího a vlastního zpracování.

Camunda poskytuje více způsobů jakým přistupovat k datům z výchozí historické databáze, lze využít jednu ze služeb poskytovaných pomocí Java API[4], nebo služby v rámci výchozího Camunda REST rozhraní[5].

3.2.4 Události

V předchozí podsekci 3.2.3 je uveden první druh událostí a to historické události, nad kterými nám umožňuje Camunda naslouchat a zpracovávat je. Dále je v procesním enginu možné definovat ještě dva základní druhy zpracování událostí:[11]

- Execution Listeners — umožňuje reagovat na start a konec libovolného elementu v procesu a procesní instance samotné.
- Task Listeners — umožňuje zpracovávat základní události u uživatelských úkolů a to create, assignment, update, timeout, complete, a delete.

Pomocí Camunda rozšíření je možné definovat zpracování událostí na globální úrovní, tedy společné zpracování pro všechny procesy v rámci daného procesního enginu.[4]

V rámci distribuce Camunda pro projekt Spring Boot je možné využít takzvaný Spring Eventing Bridge, který umožňuje snadno pomocí anotace `@EventListener` naslouchat na událostech v rámci procesu.[11]

3.3 Architektura

V rámci této sekce je představena architektura samotného procesního engine Camunda a jakým způsobem lze procesní engine provozovat.

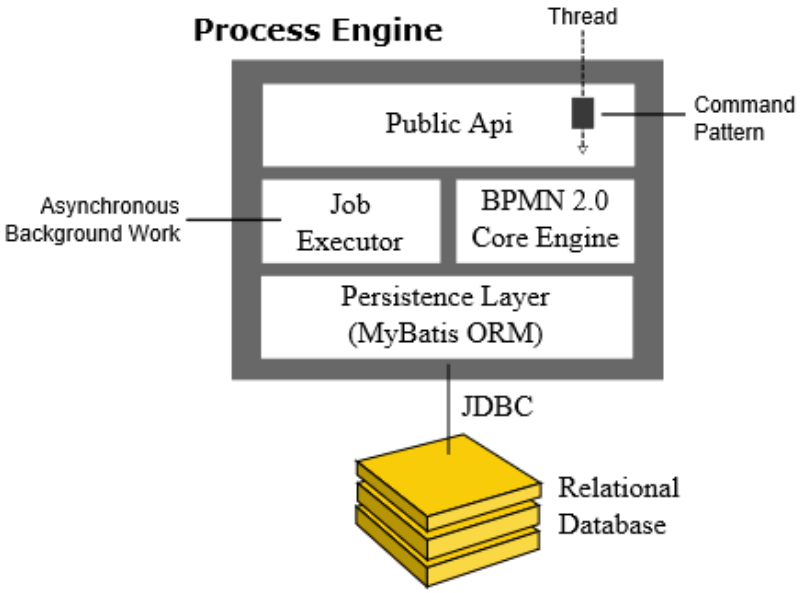
3.3.1 Procesní engine

Na diagramu 3.6 lze vidět architekturu procesního engine Camunda. Sestává z následujících základních komponent:[11]

[4] Camunda HistoryService — `https://docs.camunda.org/javadoc/camunda-bpm-platform/7.18/org/camunda/bpm/engine/HistoryService.html`

[5] Camunda REST API History — `https://docs.camunda.org/manual/latest/reference/rest/history/`

- Perzistentní vrstva a relační databáze.
- Job Executor odpovědný za asynchronní operace v procesech, jako jsou například časovače.
- BPMN 2.0 Core Engine umožňující parsovat a spouštět modely v notaci BPMN 2.0.
- Public API — servisně orientované API umožňující pracovat s procesním enginem. Jednotlivé odpovědnosti procesního engine jsou rozděleny do služeb. Každá ze služeb je thread-safe.



Obrázek 3.6: Architektura procesního engine Camunda, zdroj: [11]

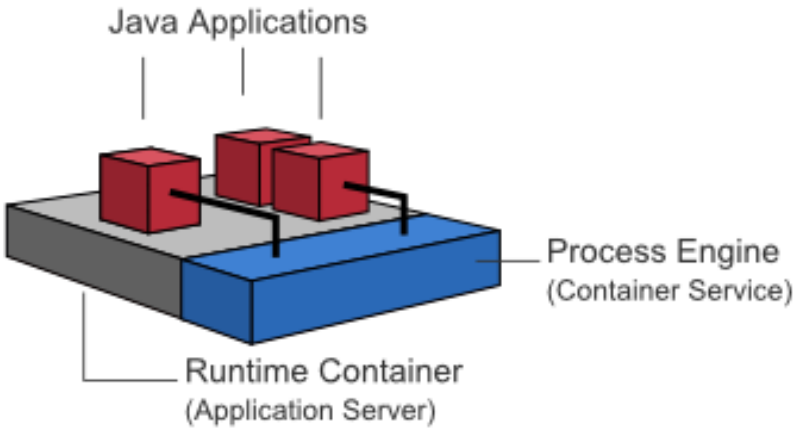
3.3.2 Způsoby provozování

Tato sekce popisuje tři základní způsoby, jak provozovat procesní engine Camunda a detailněji rozbírá vlastnosti jednotlivých architektur. Jednotlivé způsoby se liší způsobem nasazení i tím, jak mohou jednotlivé procesní aplikace komunikovat s procesním engine. Každý ze způsobů je vhodný pro jiné prostředí a případ použití, proto je součástí této sekce také popis, kdy je vhodné danou architekturu použít.

Container managed process engine

Jak je znázorněno na diagramu 3.7, tak je v případě Container Managed modelu procesní engine sdílený pro daný kontejner, který typicky představuje aplikační server (například JBoss, Tomcat, Payara). Procesní engine je poté

dostupný všem procesním aplikacím které jsou v rámci tohoto kontejneru nasazeny.[11]

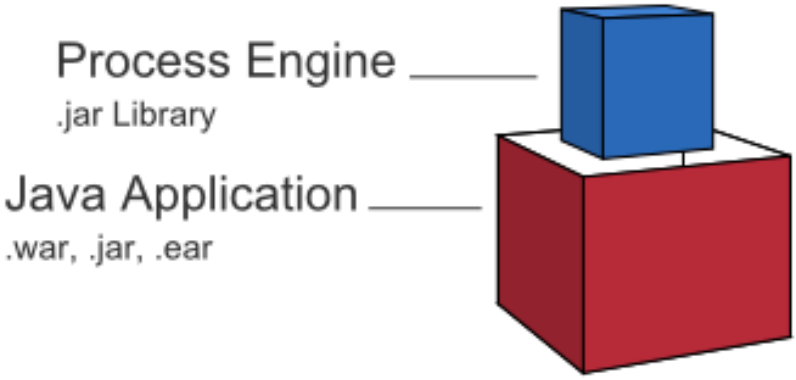


Obrázek 3.7: Container managed process engine Camunda, zdroj: [11]

Dnes je spíše odklon od toho modelu a využívají se další dva a to Remote a Embedded process engines. Tuto změna výrazně ovlivnilo rozšíření servisně orientovaných architektur, pro které jsou vhodnější právě další dvě architektury.[11]

Embedded process engine

V případě Embedded modelu je procesní engine využíván jako Java knihovna v rámci procesní aplikace, jak je znázorněno na diagramu 3.8.



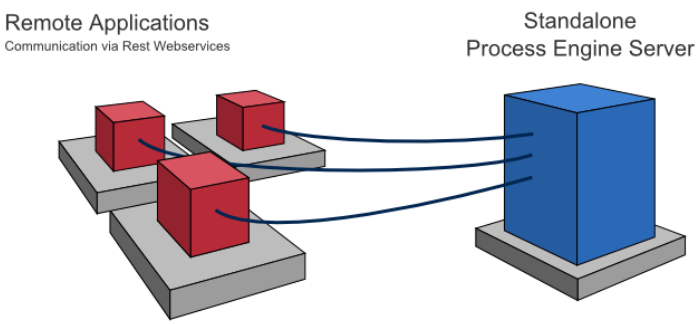
Obrázek 3.8: Embedded process engine Camunda, zdroj: [11]

Výhodou tohoto modelu je jeho snadné použití pro Java vývojáře a podpora pro J2EE frameworky, jako je Spring Boot Framework, s tím je také spojená větší zátěž na vývojáře, kteří často procesní engine musí konfigurovat a

spravovat. Tento způsob je vhodný, pokud je požadované mít jen jednu komponentu, která se bude nasazovat společně s procesy. Pro některé scénáře může být náročnější zprovoznit webové aplikace (Tasklist, Cockpit, Admin) a výchozí REST rozhraní poskytované Camundou.[11]

Remote process engine

V případě Remote modelu je procesní engine poskytován jako služba po síti. Pro ovládání procesního engine se dá využív více způsobů komunikace, typicky to bude pomocí HTTP protokolu, kdy procesní engine vystavuje REST rozhraní a procesní aplikace obsahují REST klienty. REST není jediný způsob jak procesní engine ovládat, ale existují i komunitní rozšíření například pro Google RPC protokol nebo pro komunikaci přes messaging system Apache Kafka. Na diagramu 3.9 je možné vidět centrální sdílený procesní engine a několik procesních aplikací, které s ním komunikují po síti.[11]



Obrázek 3.9: Remote process engine Camunda, zdroj: [11]

V době vzniku této práce je spíše preferovaný přístup kdy je procesní engine poskytován jako služba na síti, tedy model takzvaného remote engine. Hlavní důvody jsou především následující:[11]

- Oddělenost aplikací a procesní engine. Je možné snadno lokalizovat vzniklé chyby a zranitelnosti se nepřenáší na další komponenty.
- Lepší škálovatelnost. Procesní engine lze škálovat nezávisle na procesní aplikaci.
- Umožňuje Software-as-a-Service model, kdy je procesní engine poskytován jako služba po síti. S modelem SaaS jsou spojeny především nižší náklady na údržbu.
- Nezávislost na technologii. V případě embedded enginu jsou vývojáři závislí na jazyku Java, naopak při komunikaci po síti je možné využít v podstatě libovolnou alternativu.

Na druhou stranu využívání procesního engine jako služby po síti představuje určité technické výzvy, které jsou především následující:[11]

- Komplikovanější programovací model. Používání komunikace po síti přes REST rozhraní je pro vývojáře méně přívětivé než používání knihovny pro daný jazyk a framework.
- Transkace v případě komunikace po síti. Pokud využívám procesní engine jako službu po síti, nemohu zaručit transakce v kódu procesní aplikace a zároveň v procesním engine.
- Testovatelnost je náročnější. Pro jednotkové testy potřebuji zvláštní prostředí a dodatečné nástroje.

3.3.3 Multi-tenantní systémy

V případě že stejný procesní engine chce využívat více nezávislých stran, je možné využít dvou způsobů, jak je v rámci jednoho procesního engine rozdělit. Prvním je pomocí odděleného schéma v rámci databáze. Druhý způsob je pomocí takzvaného markeru na úrovni jednoho záznamu v rámci databáze, jedná se o atribut určující, kterému tenantovi daný záznam (například úkolu nebo procesní instance) náleží.[11]

3.3.4 Vysoká dostupnost

Camunda umožňuje zaručit vysokou dostupnost a rozložení zátěže systému pomocí replikace stejné procesní aplikace do více instancí, které využívají stejné databázové schéma. Díky tomu, že procesní engine ukládá stav do sdílené databáze při každé transakci, je možné bezpečně rozložit práci i na stejné procesní instanci mezi více instatní, které sdílí stejnou databázi.[11]

3.4 Shrnutí

V rámci této kapitoly byly popsány jednotlivé komponenty platformy Camunda, která poskytuje nástroje pro podporu všech fází řízení procesů. Vzhledem k tématu praktické části práce byl kladen důraz na správu uživatelských úkolů, jejich strukturu a možnosti naslouchání na události nad těmito úkoly. Na závěr kapitoly byly představeny způsoby provozování procesního enginu Camunda a jednotlivé způsoby byly porovnány. Na závěr bylo uvedeno, jak lze využít platformu Camunda pro multi-tenantní systémy a jak zajistit u procesního engine vysokou dostupnost.

Kapitola 4
Technologie

Součástí této kapitoly je popis konceptů a technologií využívaných v rámci praktické části. Jedná se především o koncepty a technologie použité pro integraci systémů, konkrétně se jedná o Enterprise Messaging, popsaný v sekci 4.1 a dotazovací jazyk GraphQL a související technologie, popsané v sekci 4.2.

4.1 Enterprise Messaging

Enterprise messaging je jedním ze způsobů integrací systémů. Jedná se o sadu konceptů a technologií umožňující rychlou, asynchronní a spolehlivou komunikaci po síti mezi službami. Výhodou je, že příjemce nemusí být v době odeslání zprávy dostupný, ale zprávu si může z messaging systému vyzvednout zpětně.[14]

Tato sekce obsahuje vysvětlení základních principů integrace pomocí messaging systémů. Zároveň obsahuje srovnání různých software označovaných jako messaging system nebo message-oriented middleware (MOM), což jsou softwarové systémy, které umožňují realizovat service-to-service integraci pomocí messaging.

4.1.1 Základní principy

Základní termíny použité v rámci této sekce jsou následující:[15]

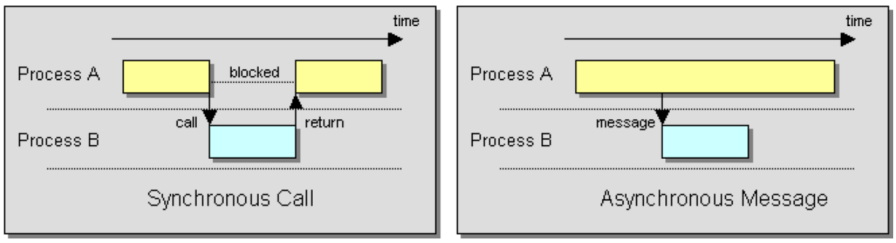
- `channel (kanál)` — někdy označovaný jako queue, je virtuální cesta pomocí které si služby předávají zprávy. Kanál se chová jako kolekce zpráv ke které může mít současně přístup více programů.
- `message (zpráva)` — pole bytů přenášené pomocí kanálu, může využívat různé serializační formáty, jako je například JSON, Avro, protobuf.
- `sender / producer` — program který vytváří a odesílá zprávu pomocí kanálu.
- `receiver / consumer` — program který přijímá zprávu z kanálu.

4.1.2 Vlastnosti messaging systémů

Jedním z hlavních důvodů pro použití messaging systému je využití asynchronní komunikace. Asynchronní komunikace přináší mnoho výhod, například využití fire-and-forget stylu komunikace, kdy odesílatel zprávy nemusí čekat na odpověď cílového systému, je ale také zdrojem dodatečné komplexity. Jedná se především o složitější vývoj a debugování systému. Rozdíl mezi blokující — synchronní komunikací a asynchronní fire-and-forget komunikací je možné vidět na diagramu 4.1.[14, 15]

Výhodou messaging systémů je jejich robustnost a spolehlivost. Většina řešení umožňuje nastavit dodatečné mechanismy, které zvyšují spolehlivost celkového řešení jako je například ukládání zpráv na disk místo do operační paměti, dále pak potvrzení o přijetí zpráv a případné znovuodeslání. Z pohledu záruk je důležité zmínit, že messaging systémy často zaručují doručení zprávy, ale nezaručují pořadí ve kterém se zprávy do cíle dostanou.[14, 15]

Messaging systémy také mohou řešit situaci, kdy je cílový systém krátkodobě nedostupný a díky tomu že si messaging systém zprávy ukládá, může si je cílový systém vyzvednou až bude znovu aktivní.[14, 15]



Obrázek 4.1: Asynchronní versus synchronní komunikace, zdroj [14]

Při výběru messaging systému je nutné brát zřetel na takzvaný vendor lock-in, tedy případ kdy messaging systém využívá nějakou proprietární část, například komunikační protokol a změna dodavatele daného messaging systému by nebyla možná, nebo by se jednalo o nákladný zásah do celkového řešení.[14]

V rámci rozsáhlejších systémů se typicky budou vyskytovat různé jazyky a knihovny, výhodou a zároveň důležitou vlastností messaging systémů je jejich podpora pro danou technologii, programovací jazyk a případně i knihovnu.[14]

Z pohledu výkonu může být velkou výhodou asynchronní komunikace to že nedochází k blokování vláken na straně odesílatele a zároveň mohu na přijímající straně kontrolovat kolik zpráv consumer může současně zpracovávat a tím přecházet zahlcení.[14, 15]

Při implementaci integrace pomocí messaging systému je také nutné zvážit, že na úrovni infrastruktury vzniká nová komponenta, která bude vyžadovat zdroje na správu a provoz, stejně jako je tomu například u nového databázového systému.

4.1.3 Messaging API

Jako Application Programming Interface (API) je v kontextu enterprise messaging systémů označováno rozhraní pro využívání a ovládání messaging systému z různých programovacích jazyků, knihoven nebo frameworků. Messaging API specifikuje, jakým způsobem probíhá interakce s jednotlivými komponentami messaging systémů, nespecifikuje ale jakým způsobem jsou zprávy přenášeny, to je odpovědností messaging protokolů.[16]

Většina messaging systémů poskytuje proprietární API a některé podporují standardy jako je například otevřený standard Java Message Service API (JMS).[16]

4.1.4 Messaging protokoly

Protokolem v kontextu enterprise messaging systémů je myšlena specifikace, jakým způsobem budou přenášeny zprávy, kombinace zpráv nebo packetů pro danou operaci při komunikaci v rámci messaging systému.[16]

Messaging systémy mohou podporovat vlastní proprietární protokol nebo některý z otevřených standardizovaných protokolů, jako je například:[16, 17, 18, 19]

- MQTT — Je OASIS[1] standard, který je navržen tak, aby měl co nejmenší nároky na šířku pásma a potřebný software na straně odesílatele a příjemce. Vzhledem k jeho vlastnostem je často využíván pro integraci v rámci Internetu věcí, kde se typicky vyskutuje větší množství zařízení s malým výkonem.
- AMQP — Advanced Message Queuing Protocol je otevřený OASIS standard, který byl navržen jako náhrada pro dosavadní proprietarní protokoly. AMQL poskytuje širokou řadu vlastností, jako je spolehlivé rozdělení zpráv do front, publish-subscribe messaging a velice flexibilní routing zpráv. Díky bohatým vlastnostem, bezpečnosti a rozšiřitelnosti protokolu je široce využíván v prostředí enterprise aplikací.
- STOMP — Simple Text Orientated Messaging Protocol je jednoduchý protokol vycházející silně z protokolu HTTP. Vzhledem k tomu, že messaging systémy musí převádět textovou, člověkem čitelnou, reprezentaci protokolu na vnitřní reprezentaci, má tento protokol většinou dopady na výkon. Vzhledem k této negativní vlastnosti není široce využíván v průmyslu jako je tomu u předchozích dvou protokolů.

4.1.5 Srovnání - Messaging systems

V rámci této podsekce práce jsou popsány a porovnány tři implementace messaging systémů, a to Apache Kafka, RabbitMQ a ActiveMQ. Všechny

[1] Organization for the Advancement of Structured Information Standards (OASIS) je nezisková organizace zajišťující standardizaci protokolů pro oblasti jako je Internet věcí, kyberbezpečenost a další.

tři messaging systémy jsou open-source. Jejich komerčním alternativy, jako je například Red Hat AMQ, IBM MQ, Google Cloud Pub/Sub a Amazon Kinesis Data Streams byly ze srovnání vyřazeny, protože nesplňují licenční požadavky specifikované pro praktickou část této práce.[20, 21, 22]

ActiveMQ

ActiveMQ je open-source multi-protocol message broker založený na jazyku Java. ActiveMQ v době vzniku této práce existuje ve dvou variantách — Classic a Artemis. Artemis je novou verzí systému, která zatím poskytuje jen část potřebné funkcionality. V momentě kdy se verze Artemis vyrovná verzi Classic v poskytované funkcionalitě, tak se stane novou hlavní verzí. V této práci je popsána verze Classic, vzhledem k tomu že je v době vzniku této práce stále více stabilní. Základní vlastnosti ActiveMQ jsou následující:[20]

- Nativní messaging protokol ActiveMQ je binární protokol OpenWire.
- ActiveMQ podporuje protokoly Stomp, AMQP, MQTT.
- ActiveMQ podporuje JMS, REST a WebSocket rozhraní a většinu populárních jazyků, knihoven a frameworků, včetně frameworku Spring Boot.[2]
- Podporuje většinu návrhových vzorů se skupiny Enterprise integration patterns. Tyto návrhové vzory se využívají pro tvorbu nových distribuovaných entreprise aplikací, nebo pro integraci stávajících systémů.

RabbitMQ

RabbitMQ je open-source message broker. Základní vlastnosti RabbitMQ jsou následující:[21, 23]

- Nativní messaging protokol RabbitMQ je otevřený protokol AMQP.
- RabbitMQ podporuje protokoly Stomp, MQTT, HTTP, WebSocket a RabbitMQ Streams protokol.
- RabbitMQ podporuje JMS a další rozhraní pro většinu populárních jazyků, knihoven a frameworků, včetně frameworku Spring Boot.[3]
- Podporuje většinu z EIP.[4]
- RMQ umožňuje konfigurovat složitější routing stategie pro distribuci zpráv. RMQ například umožní vytváření hierachie kanálů neboli topiců, tak že příjemce zpráv dostává opravdu jen zprávy které ho zajímají.

[2] Seznam podporovaných klientů: `https://activemq.apache.org/cross-language-clients`

[3] Seznam podporovaných klientů: `https://www.rabbitmq.com/devtools.html`

[4] Ukázka EIP v RMQ: `https://github.com/videlalvaro/rmq_patterns`

Apache Kafka

Apache Kafka je distribuovaný open-source streaming systém pro stream processing, real-time data pipelines a datovou integraci. Základní vlastnosti Apache Kafka jsou následující:[24, 22]

- Kafka využívá proprietární binární protokol přenášející data přes TCP protokol.
- Kafka se oproti ostatním messaging systémům vyznačuje vysokou propustností a nízkou odezvou. Díky snadnému horizontálnímu škálování a využití sekvenčního zápisu na disk a Zero-copy principu.
- Kafka umožňuje perzistentní ukládání zpráv. Kafka zároveň umožňuje nastavit retenci zpráv, tak aby ke smazání došlo jen v případě nakonfigurovaných podmínek nebo dosažení limitů. Kafku lze proto využít pro získání historie zpráv.
- Kafka umožňuje snadno přidávat nové příjemce zpráv s minimálním dopadem na výkon.
- Kafka poskytuje klienty, jak pro příjem, tak odesílání zpráv pro velkou řadu nejpopulárnější programovacích jazyků a frameworků (Java, C++, Python, Go a další). Dále je možné využít i některého z komunitních open-source klientů[5].
- Kafka poskytuje pokročilejší klienty jako je například Kafka Connect API pro data systémy a Kafka Streams pro stream processing.
- Kafka neumožňuje oproti předchozím dvěma messaging systémům konfigurovat složitější routing strategie. Vzhledem k tomu může být složitější filtrování zpráv na straně která přijímá zprávy. Kafka umožňuje specifikovat takzvaný topic, který je ekvivalentem kanálu, popsaného v terminologii. Každý topic lze dále rozdělit do jednotlivých partitions do kterých se zprávy přiřazují na základě jejich klíče. Kafka zaručuje pořadí zpráv pouze na úrovni partition, nikoliv na úrovni topic a je proto důležité dbát správného návrhu Kafka topiců i jejich partitions.
- Kafka umožňuje nastavit replikaci na topic, může tedy existovat více kopií dat na úrovni topicu, které mohou být i v rámci jiných lokací.

Pro systém Apache Kafka jsou zprávy jednoduché pole bajtů, proto je často nutné aplikovat na zprávy vnitřní strukturu dat nazvanou schéma. Pro serializaci zpráv lze použít některý ze serializačních formátů, například některý z jednoduších, lidsky čitelných formátů jako je Javascript Object Notation (JSON). Nebo některý z formátů, který nabízí dodatečné funkce jako zavedení datových typů a verzování schéma, příkladem takového formátu je Apache Avro.[22]

[5]Apache Kafka, seznam komunitních klientů: `https://cwiki.apache.org/confluence/display/KAFKA/Clients`

Ve většině případů je systém dále rozšířen o takzvané schema registry, které umožňuje definovat strukturu přenášených zpráv a sdílet ji napříč integrovanými systémy.[22]

4.1.6 Shrnutí

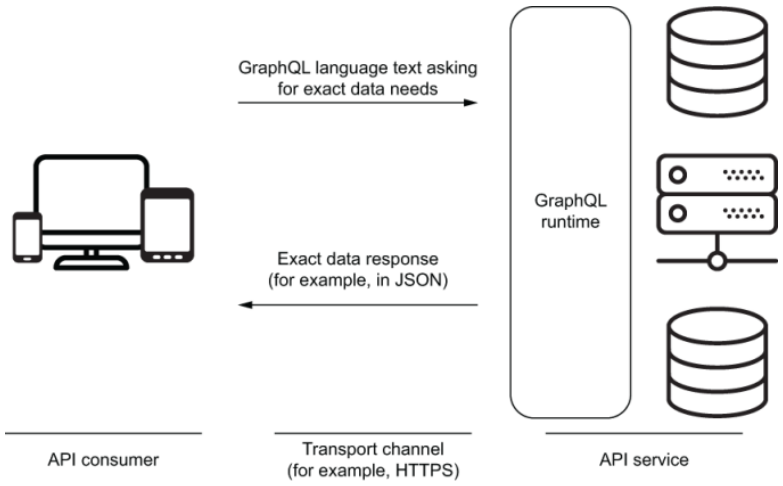
V rámci této sekci byla představena základní terminologie a koncepty integrace systémů pomocí messaging systémů. Dále byly v rámci této sekci srovnány hlavní open-source zástupci messaging systémů.

4.2 GraphQL

Součástí této sekce je popis technologie GraphQL (GQL), která sestává jednak ze specifikace pro dotazovací jazyk, tak i engine na serveru, který tyto dotazy umí vykonávat. Detailní popis, včetně příkladů je uveden v podsekci 4.2.1. V podsekci 4.2.2 jsou dále popsány specifika GQL, a jakým způsobem se liší od alternativních přístupů pro poskytování API. Další podsekce 4.2.3 se zabývá bezpečností GQL API. Závěrem, v podsekci 4.2.4, jsou uvedeny doporučení pro použití GQL.

4.2.1 Popis

Pojmem GraphQL se označuje deklarativní jazyk pro API nad daty, ale zároveň GraphQL Runtime, jak je možné vidět na obrázku 4.2. GraphQL runtime obsahuje popis struktury dat, které poskytuje ve formě takzvaného GraphQL schéma. Zároveň je runtime na serveru schopný interpretovat a zpracovat požadavky klienta, ať už se jedná o operace čtení (queries) nebo zápisu (mutations).[25, 26]



Obrázek 4.2: Graphql, zdroj [25]

GraphQL není závislá na konkrétním druhu zdroje dat, typickým příkladem může být grafová nebo relační databáze, případně REST rozhraní. Další

důležitou vlastností GraphQL je, že je silně typový. Dokument specifikace GraphQL je volně dostupný[6] a dle této specifikace vznikly implementace pro různé jazyky a frameworky, včetně jazyka Java[7].[25, 26]

4.2.2 Srovnání

Jak již bylo zmíněno v úvodu o GraphQL, API vystavené pomocí tohoto standardu vynucuje, aby bylo popsáno pomocí GraphQL schéma, což je velká výhoda oproti REST rozhraní, kde se musí využít dalších rozšíření, jako je například Open API standard. GraphQL schéma nám tak poskytuje aktuální dokumentaci daného systému, která vždy odpovídá implementaci. Na obrázku 4.3 je možné vidět příklad takového GQL schématu.[25, 26]

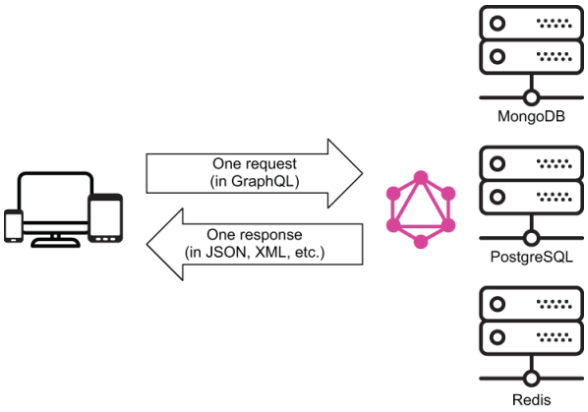
```
type Tweet {
    id: ID!
    body: String
    Author: User
    Stats: Stat
}

type User {
    id: ID!
    username: String
    full_name: String
    name: String @deprecated
}

type Stat {
    views: Int
}

type Query {
    Tweet(id: ID!): Tweet
    Tweets(limit: Int, skip: Int, sort_field: String, sort_order: String): [Tweet]
}

type Mutation {
    createTweet (
        body: String
    ): Tweet
    deleteTweet(id: ID!): Tweet
}
```

Obrázek 4.3: Graphql - ukázka schéma, zdroj autor

Grafová povaha GQL také umožňuje optimalizovat dotazy klienta, tak aby se omezilo množství přenášených dat, tím že si klient vybere jen atributy daného typu, které opravdu potřebuje. Runtime vrstva nám také umožňuje spojit více různých zdrojů a poskytovat je pod jedním rozhraním, klient tak nemusí provádět více separátních dotazů, ale provede jeden a o složení

[6] Specifikace GQL - `https://spec.graphql.org/`
[7] GQL Java knihovna - `https://www.graphql-java.com/`

odpovědi se postará GQL runtime. Na diagramu 4.4 je znázorněno složení více dotazů z různých zdrojů.[25, 26]



Obrázek 4.4: Graphql - multirequest, zdroj [25]

Možnost provést dotaz spojující více zdrojů spolu přináší možné problémy spojené s výkonem daných dotazů, které zasahují do velkého množství zdrojů. Tato vlastnost GQL může být také potencionálně zneužita v případě denial-of-service útoků, kdy se útočník pokusí vyčerpat zdroje serveru příliš komplexním dotazem. Způsoby, jakými lze předcházet těmto rizikům jsou detailněji popsány v podsekci 4.2.3 zabývající se bezpečeností GraphQL API.[25]

V případě service-to-service komunikace, tedy komunikace mezi dvěma backend systémy, převažují spíše výhody alternativních přístupů, jako je Google Remote Procedure Call (gRPC), které umožňuje lépe optimalizovat konkrétní volání a kde výhody GQL zmíněné dříve v této sekci nejsou tak významné.[27]

4.2.3 Bezpečnost

Tato podsekce se zabývá zabezpečením GraphQL API. Konkrétně pak specifiky, které vychází z vlastností GQL, a to především možnosti vykonávat komplexní dotazy přes více různých zdrojů, což představuje určité specifické výzvy z pohledu bezpečnosti.

Limity

Jsou různé způsoby, jak zabránit klientům, aby přetěžovali GraphQL server a došlo tak k vyčerpání zdrojů. Limitování požadavků daného klienta je jedním z nich. Pro klienty využívající vystavované rozhraní je důležité, aby věděli, zda-li ještě splňují limity a kdy dojde k jejich vynulování. Jedním z možných způsobů, jak o limitech klienta informovat jsou hlavičky v odpovědi ze serveru, příklad takových hlaviček může být následující:[27]

```
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4999
X-RateLimit-Reset: 1372700873
```

Limit počtu dotazů je omezen hlavičkou `X-RateLimit-Limit`. Zbývající počet dotazů je omezen hlavičkou `X-RateLimit-Remaining`. Unixový čas, určující kdy dojde k resetu limitů pro daného klienta je zase určen hlavičkou `X-RateLimit-Reset`.[27]

Limity pro počty dotazů je dobré kombinovat s limity na velikost jednotlivých dotazů. GraphQL umožňuje vytvářet komplexní, zanořené dotazy a tak by vyčerpání zdrojů na serveru mohlo dojít i v případě že klient bude v kratších intervalech pouštět rozsáhlé dotazy, které na serveru zaberou delší výpočetní čas. Oba limity je dobré kombinovat s Timeout limitem, tedy s časem po kterém bude dotaz ukončen s chybou.[25, 27]

Autorizace a autentizace

Obecně je preferovaným způsobem vynechat autentizaci, tedy ověření identity klienta, přímo z GraphQL rozhraní a využít pro ni jiné mechanismy, jako je například autentizační midlleware. Hlavním důvodem je možnost vynechat konkrétní autentizační mechanismy bez změn v GQL schématech.[25, 27]

Autorizace, konkrétně její část obsahující byznys logiku, by měla být spíše součástí logiky aplikace a neměla by se objevovat přímo v GraphQL rozhraní. Příkladem může být ověření, zda-li daný uživatel může uzavřít konkrétní smlouvu. Tyto byznysová pravidla se typicky často mění, a neměla by zasahovat do API. Dalším případem jsou takzvané API scopes, neboli oblasti ke kterým daný klient může přistupovat. Příkladem může být ověření, že je klient administrátor a může vidět všechna nastavení aplikace. V tomto případě je dobrým zvykem nastavovat tyto omezení přímo podle daných objektů, než na konkrétní dotazy. V případě rozsáhlejších schémat je tak snazší kontrolovat, že někde designér API nezapomněl dotaz, který umožní k danému zdroji přistoupit.[25, 27]

4.2.4 Doporučení

V této podsekci jsou uvedena některá doporučení pro tvorbu rozhraní pomocí GraphQL, mezi tato doporučení patří:[25, 26, 27]

- Přístup design-first, kdy nejprve vzniká specifikace rozhraní a až poté implementace, umožní lepší komunikaci se zainteresovanými stranami, například s jiným týmem využívajícím navrhované rozhraní nebo se znalci dané domény. Je proto preferovaný oproti přístupu code-first, kdy vzniká první implementace a až dle ní vzniká, například generováním z kódu, specifikace.
- Návrh by se měl provádět vzhledem ke klientským požadavkům na API a ne konkrétním typům poskytovaných dat.
- Schéma by mělo mít co největší vypovídací hodnotu a mělo by jasně stanovovat kontrakt. Schéma by také mělo obsahovat co nejméně skrytého chování pro klienty, tomu se dá zabránit například rozdělením dotazů nebo přidáním výchozích hodnot.

4.2.5 Alternativy

Vzhledem k omezením stanoveným v rámci praktické části této práce budou rozhraní využívat GraphQL. Pro úplnost jsou zde uvedeny, alespoň v krátkosti nejpoužívanější, alternativní přístupy pro integraci systémů:

- Representational State Transfer (REST) a Open API specifikace (OAS). Službu lze označit jako RESTful pokud splňuje sadu architektonických omezení, jako je například splnění architektonického stylu client-server, bezstavovosti, jednotného rozhraní. REST sám o sobě nespecifikuje jakým způsobem má být popsána poskytovaná služba, proto vznikla Open API specifikace umožňující standardizovaným způsobem definovat poskytovanou REST službu.[28, 29]
- Simple Object Access Protocol (SOAP) a popis služeb pomocí Web Service Description Language (WSDL). SOAP je protokol využívající hlavně HTTP pro komunikaci pomocí XML zpráv po síti. WSDL umožňuje popsat pomocí XML poskytované služby.
- Google Remote Procedure Call (gRPC). Open-source framework pro propojení služeb po síti. Pro specifikaci rozhraní se používá serailizační mechanismus Protocol Buffers[8].[30]

4.2.6 API Gateway

V případě distribuovaných systémů je velice často potřeba rozšířit architekturu těchto systémů o komponentu zvanou API Gateway, která slouží jako vstupní bod do systému, umožňující přístup k funkcionalitám jednotlivých služeb. Klient využívající systém je poté odstíněn od implementačních detailů, například počtu služeb nebo způsobu komunikace. API Gateway zároveň může být, a v praxi často je, rozšířena o dodatečnou funkcionalitu, kterou by každá ze služeb musela řešit samostatně. Příkladem takové funkcionality může být zabezpečení, auditování dotazů nebo nastavení limitů. API Gateway tedy poskytuje dodatečnou abstraktní vrstvu, která by ale v ideálním případě měla obsahovat minimum byznys logiky.[27]

Je více různých způsobů, jak využít GraphQL v rámci API Gateway. V rámci této práce jsou uvedeny tři základní možné způsoby:[27]

- Jednoduché proxy, kdy API Gateway obsahuje spojená schémata jednotlivých služeb. V rámci tohoto přístupu nedochází k transformaci dotazů ani spojení společných atributů u typů. Jedná se o snadno aplikovatelný přístup, při kterém lze zaručit, že API Gateway bude jen primitïvní vrstvou bez složitého skládání dotazů a tranformací. Nevýhodou tohoto přístupu je, že nelze naplno využít sílu deklarativního jazyka GraphQL.
- Schema Stitching je přístup, kdy jako v předchozím případě dochází ke spojení schémat jednotlivých služeb na úrovni API Gateway. V rámci

[8] `https://developers.google.com/protocol-buffers`

tohoto přístupu jsou popsány závislosti mezi typy jednotlivých služeb a je zde popsána i logika, jak tyto závilosti v rámci dotazů vyřešit. Hlavní nevýhodou je, že API Gateway obsahuje logiku pro vyřešení závislostí mezi službami a mohou vznikat problémy při změnách na úrovni API jednotlivých služeb.

- Schema Federation je přístup, který se zaměřuje na problémy se Schema Stitching. Jedná se o decentralizovaný přístup, kdy jednotlivé služby definují svá schémata, takzvaných podgrafů, a popisují, jak rozšiřují schéma ostatních služeb. Jednotlivé služby mají v tomto případě kontrolu nad vývojem svého schéma.

4.2.7 Shrnutí

V rámci této sekce byla popsána technologie GraphQL, včetně příkladů schémat a dotazů. Na závěr byly uvedeny doporučení týkající se bezpečnosti a návrhu Graphql API a pro úplnost také alternativní přístupy pro integraci služeb.

4.3 Elasticsearch

V rámci této sekce je popsán open-source analytický engine Elasticsearch. Existují další alternativy, jako například Apache Solr, jejich popis není součástí této práce, vzhledem k omezením stanoveným v rámci praktické části této práce v podsekci 6.1.1.

Jak již bylo zmíněno v předchozím odstavci, Elasticsearch je open-source distribuovaný, škálovatelný real-time vyhledávací a analytický engine. Elasticsearch je primárně využíván pro full-text vyhledávání a datovou analýzu v reálném čase, případně pro jejich kombinaci. V některých případech může také sloužit jako hlavní úložiště dat. Elasticsearch je implementován pomocí jazyka Java a interně využívá komplexní knihovnu Apache Lucene, která umožňuje vykonávat analýzu dat, včetně full-text vyhledávání. Elasticsearch je možné provozovat jako samostatný server, který si ukládá a indexuje data serializované jako JSON dokumenty a poskytuje REST rozhraní pomocí kterého je možné s daty pracovat a systém ovládat.[31, 32, 33]

Každá instance Elasticsearch se nazývá node, pokud dojde ke spojení více instancí, tak se takové uskupení nazývá cluster. Každý node v rámci clusteru ví o ostatních nodech a umí přesměrovat dotaz klienta na odpovídající node. Jednotlivé nody v rámci clusteru mohou mít různé role, například master role značí že node spravuje daný cluster, data node se stará o uchovávání dat a vykonává CRUD operace. V případě lokálního vývoje nebo u menších systémů může být dostatečný cluster s jedním nodem, u větších systémů je pak typické, že se rozdělí odpovědnosti mezi jednotlivé nody a podle potřeby daného systému je možné tyto instance libovolně škálovat.[31, 33]

Elasticsearch poskytuje klientskou knihovnu pro jazyk Java a zároveň framework Spring Boot poskytuje za pomoci projektu Spring Data Elasticsearch dodatečnou abstraktní vrstvu pro práci s nástrojem Elasticsearch.[33]

4.4 Shrnutí

V rámcí této kapitoly byly popsány a srovnány hlavní technologie použité v rámci praktické části aplikace, a to především technologie GraphQL v rámci sekce 4.2, následně technologie a koncepty používané v rámci Enterprise Messagingu v sekci 4.1 a analytický engine Elasticsearch v sekci 4.3.

Část II

Praktická část

Kapitola 5

Stávající stav

Součástí této kapitoly, v sekci 5.1, je popis systému Eprocesy, který je vyvíjen v rámci Centra Znalostního Managementu ČVUT fakulty elektrotechnické (FEL). Součástí sekce o systému Eprocesy je také popis jeho architektury a dalších souvisejících integrovaných systémů. Dále tato kapitola obsahuje popis stávajícího stavu správy uživatelských úkolů v rámci systému Eprocesy v sekci 5.2 a s ním i problémy v aktuálním návrhu.

5.1 Popis systému

Systém Eprocesy vznikající v rámci prostředí fakulty elektrotechnické ČVUT má za cíl pomocí BPM a automatizace procesů, popsaných blíže v rámci kapitoly 2, podpořit dlouhodobé záměry fakulty, a to konkrétně pokračovat v elektronizaci procesů a zajištění efektivní IT podpory, rekonstruovat studijní oddělení a dokončit elektronizaci jeho agendy. Aplikace Eprocesy byla uvedena do pilotního provozu v roce 2020. Nová verze vznikající v rámci Centra Znalostního Managementu má za cíl plně elektronizovat žádosti studijního oddělení pro studenty fakulty elektrotechnické. Seznam žádostí, které bude možné pomocí systému elektronicky zpracovat je následující:

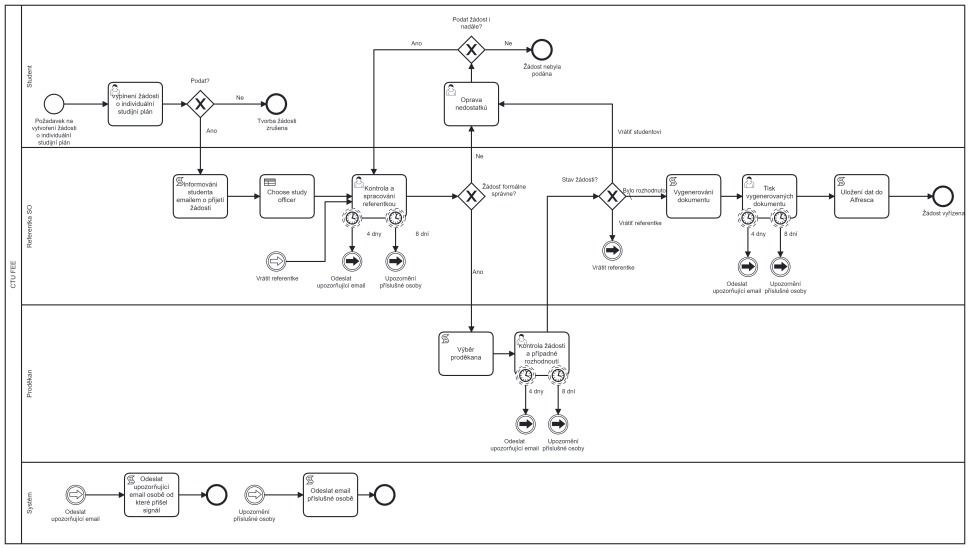
1. Žádost o změnu zadání závěřečné práce
2. Žádost o náhradní termín odevzdání závěrečné práce
3. Žádost o uznání zápočtů a zkoušek
4. Žádost o sociální stipendium
5. Žádost o přestup
6. Žádost o přerušení studia
7. Žádost o prospěchové stipendium
8. Žádost o individuální studijní plán
9. Žádost o prodloužení platnosti zadání ZP

10. Univerzální žádost

V rámci systému Eprocesy figurují následující aktéři:

- Studentka nebo student
- Referentka nebo referent studijního oddělení
- Vedoucí katedry
- Vedoucí práce
- Proděkanka nebo proděkan
- Administrátor
- Zaměstnanec fakulty

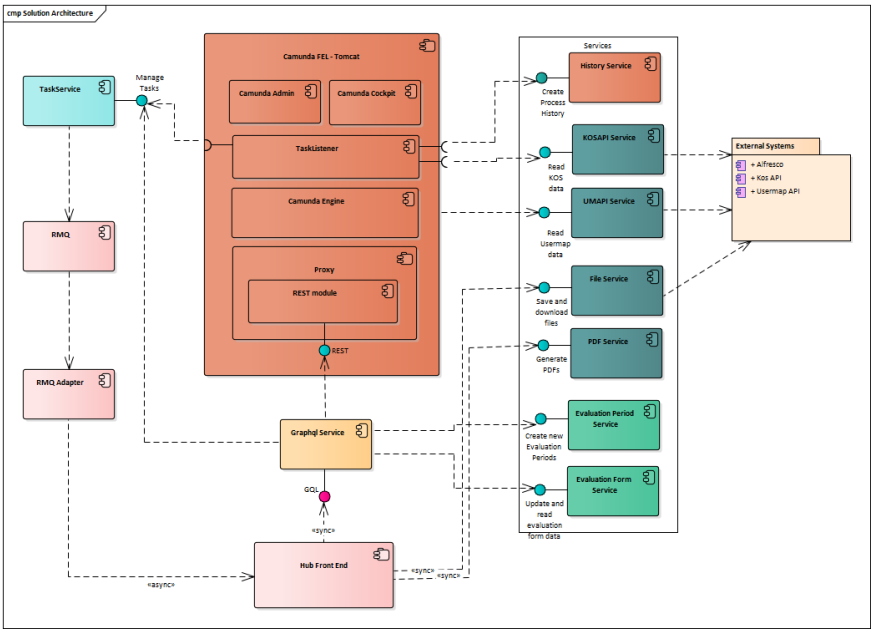
Na obrázku 5.1 je možné vidět ukázku BPMN modelu pro proces žádosti o individuální studijní plán. BPMN modely jsou blíže popsány v teoretické částí této práce v rámci sekce 2.2 o modelování procesů.



Obrázek 5.1: Žádost o individuální studijní plán, zdroj CZM ČVUT FEL

5.1.1 Architektura řešení

Na UML diagramu komponent 5.2 je znázorněna stávající architektura systému Eprocesy. Jádrem systému je centrálně spravovaný procesní engine Camunda, jak je možné vidět na diagramu, tento systém a možnosti jeho využití jsou blíže popsány v kapitole 3.



Obrázek 5.2: Diagram komponent ve stávajícím stavu, zdroj autor

V následujícím seznamu je uveden popis jednotlivých komponent ze kterých je složen systém Eprocesy:

- TaskListener — komponenta zpracovávající základní události nad uživatelskými úkoly v rámci procesního engine Camunda. Obecný task listener byl popsán v podsekci 3.2.4.
- TaskService — klíčová komponenta pro správu uživatelských úkolů, která je detailněji popsána v podsekci 5.2.3.
- RMQ — open-source systém typu message broker, jeho využití a možné alternativy jsou dále popsány v podsekci 4.1 týkající se messaging systémů.
- RMQ Adapter — komponenta umožňující asynchronní přijímání úkolů a notifikací v grafickém rozhraní systému.
- Hub front-end — grafické rozhraní systému Eprocesy integrované do systému Hub, který integruje více systémů v rámci FEL. Systém Hub je detailněji popsán v podsekci 5.1.2.
- Graphql service — služba zajišťující vystavení GQL služeb pro využití z frontend aplikace. Služba mapuje GQL rozhraní na jednotlivé REST služby.
- Services — služby poskytující dodatečnou byznys funkcionalitu pro implementované procesy nebo pro uživatelské rozhraní. Příkladem může být KOSAPI service, služba umožňující číst v JSON formátu data o

studentech a předmětech ze systému KOS[1], nebo PDF service umožňující generovat PDF dokumenty.

- External Systems — externí služby využívané v rámci aplikace, příkladem může být document management system Alfresco určený pro správu a ukládání dokumentů. Komunikace s těmito systémy probíhá synchronně pomocí protokolu HTTP. Externí systémy poskytují REST API, které s nimi umožňují dále pracovat.

Komunikace mezi komponentami je převážně synchronní pomocí HTTP protokolu. Výjimkou je odesílání uživatelských úkolů a notifikací do grafického rozhraní, které využívá systému RabbitMQ a protokolu Websocket, pro asynchronní komunikaci mezi TaskService a Hub front-end.

5.1.2 Systém Hub

Část komponent z předešlé podsekce 5.1.1 o architektuře systému Eprocesy spadá pod systém jednotného frontendu Hub FEL, který vzniká souběžně se systémem Eprocesy a se kterým je systém Eprocesy integrován.

Záměrem systému Hub je v budoucnu poskytnout jednotné uživatelské rozhraní pro aplikace vznikající v rámci CZM pro FEL ČVUT. Další systémy integrované do systému Hub jsou následující:

Aplikace pro hodnocení zaměstnanců ČVUT FEL — využívá automatizace pomocí platformy Camunda. Využívá shodnou Camunda instanci jako systém Eprocesy.

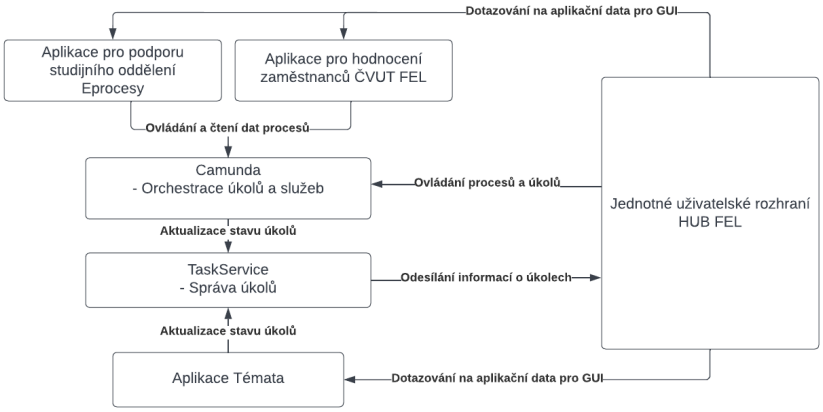
Aplikace Témata — systém pro nabídku a schvalování témat semestrálních projektů a dizertačních prací. Jde o samostatný systém, který vystavuje pro Hub FEL REST rozhraní pomocí kterého je do systému integrován. Zároveň pomocí asynchronního volání TaskService odesílá do Hub FEL aktualizace o uživatelských úkolech. Aplikace Témata nevyužívá procesní engine Camunda.

Komponenty využívané napříc systémy jsou následující:

- TaskService
- RMQ
- RMQ Adapter
- Graphql Service
- Hub front-end
- Některé ze společných služeb, například File Service

[1] Studijní informační systém v prostředí ČVUT

Způsob komunikace jednotlivých systémů je znázorněna na diagramu 5.3.



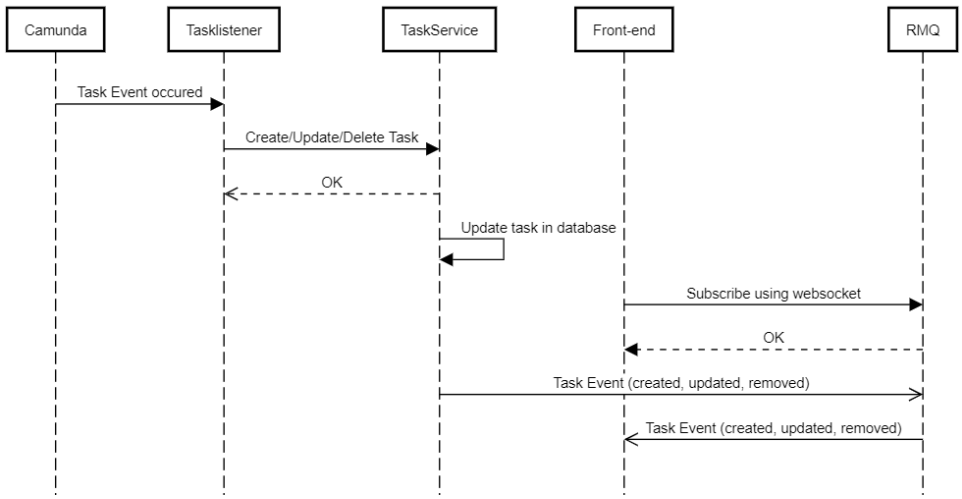
Obrázek 5.3: Komunikace systémů, zdroj autor

5.2 Stávající stav správy uživatelských úkolů

Uživatelské úkoly vznikající v systému Eprocesy a jejich správa je plně odpovědností procesního enginu Camunda. Životní cyklus těchto úkolů tedy odpovídá životnímu cyklu úkolů popsaných v podsekci 3.2.

V této sekci je popsán způsob, jakým systém Eprocesy integruje své úkoly do systému Hub, dále je zde uvedena struktura těchto úkolů a popis komponenty TaskService, která je důležitá pro správu úkolů v rámci systému Hub.

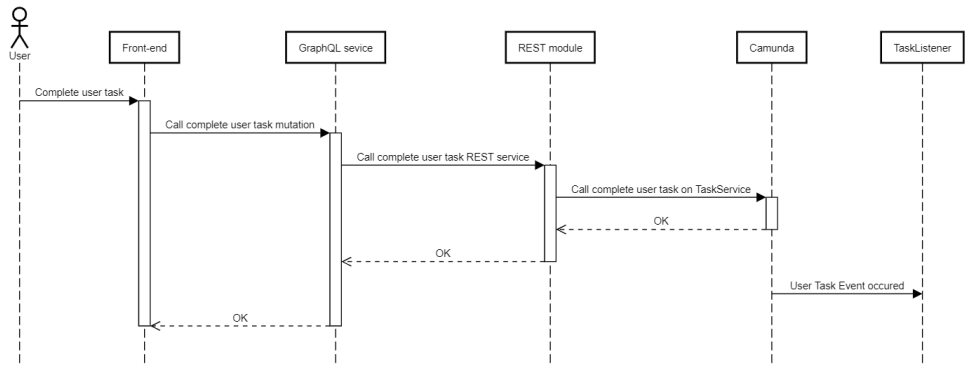
Ukázku komunikace změny na uživatelském úkolu je možné vidět na následujících sekvenčních diagramech. Na diagramu 5.4 je možné vidět, jakým způsobem je uživatelské rozhraní informováno o změně uživatelského úkolu.



Obrázek 5.4: Komunikace při změně uživatelského úkolu, zdroj autor

Na diagramu 5.5 je možné vidět změnu ze strany uživatele, přes grafické

rozhraní, kde uživatel splní daný úkol. To jakým způsobem se uživatelské úkoly dají zpracovat v GUI systému Hub je detailněji popsáno v podsekci 5.2.4.



Obrázek 5.5: Změna uživatelského úkolu z GUI, zdroj autor

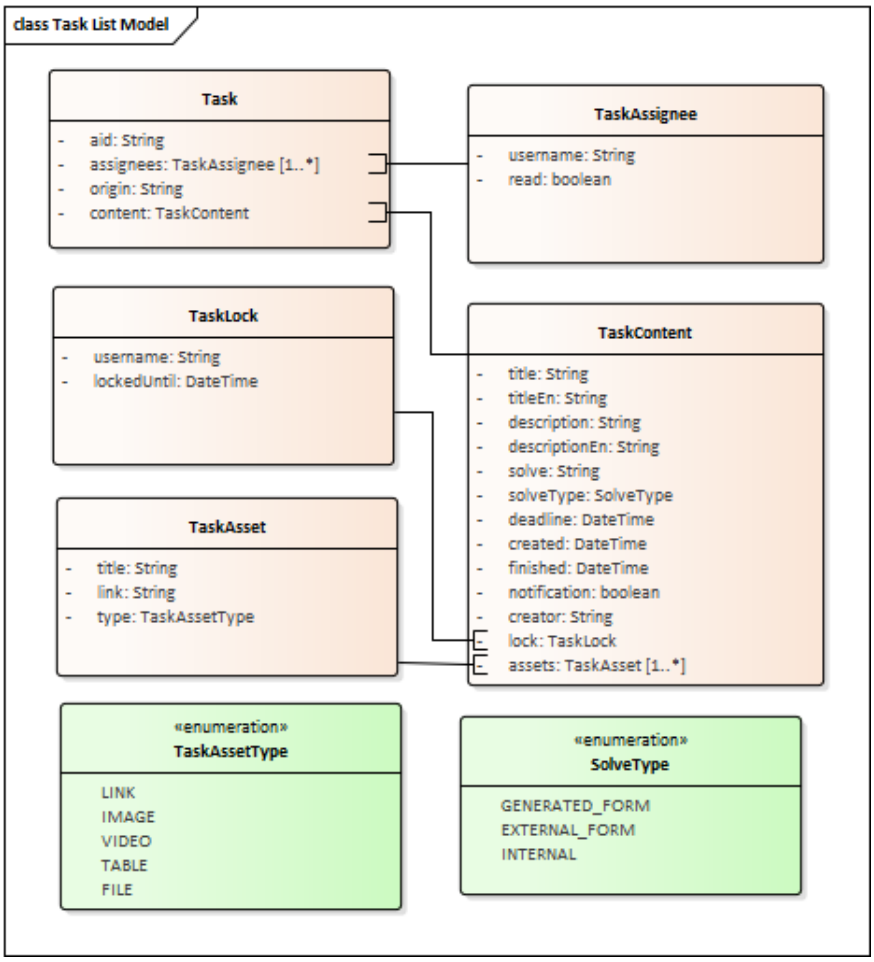
Způsob jakým lze naslouchat na události nad úkoly v rámci procesního engine Camunda jsou popsány v rámci podsekce 3.2.4 v teoretické části této práce. Pro dokončení úkolu se využívá služba TaskService procesního engine Camunda.[2]

5.2.1 Úkoly v systému

V rámci této podsekce je popsána struktura uživatelských úkolů v systému Eprocesy, respektive v systému Hub s nímž je systém Eprocesy silně integrován, jak bylo popsáno v předchozí podsekci 5.1.2.

Struktura úkolů vznikajících v platformě Camunda je popsána v podsekci 3.2.1. Na diagramu 5.6 je možné vidět strukturu uživatelského úkolu, tak jak je definována ve službě systému Hub — TaskService, která je blíže popsána v podsekci 5.2.3. Datovou transformaci úkolu ze systému Camunda na strukturu předepsanou TaskService provádí komponenta Task Listener popsaná v podsekci 5.2.2.

[2]Camunda Taskservice umožňuje pracovat s úkoly v rámci procesního engine pomocí Java API — `https://docs.camunda.org/javadoc/camunda-bpm-platform/7.18/org/camunda/bpm/engine/TaskService.html`



Obrázek 5.6: TaskList — struktura uživatelského úkolu, zdroj autor

Rozdíly vůči struktuře Camunda úkolu jsou především následující:

- Úkol v TaskService může mít více řešitelů. Pro uzamčení úkolu proti více souběžným úpravám se využívá TaskLock, který obsahuje uživatelské jméno a aktuálního řešitele a platnost zámku.
- V TaskService neexistuje koncept kandidátských ani řešitelských skupin.
- V TaskService neexistuje koncept delegace úkolu.
- TaskService má podporu pro dva jazyky v atributech title a description.
- V TaskService může úkol obsahovat přílohy ve formě souborů, odkazů, obrázků, videa nebo tabulek.
- V TaskService se ukládají i dokončené úkoly. Camunda sice ukládá auditní data o vyřešených úkolech, ale tyto data jsou součástí pouze auditních — historických — dat a nejsou součástí hlavní aplikační databáze engine Camunda.

- TaskService poskytuje tři způsoby jak vykonat a dokončit úkol pomocí GUI. Tyto tři způsoby jsou blíže popsány v další části této podsekce.

Solve a SolveType

V rámci systému Hub je možné vyřešit uživatelský úkol třemi způsoby, které jsou určeny výčtovým typem SolveType:

- INTERNAL — atribut solve v tomto případě obsahuje relativní referenci na interní modul v GUI.
- EXTERNAL_FORM — atribut solve v tomto případě obsahuje odkaz pro přesměrování do externího GUI, kde je úkol možné vyřešit.
- GENERATED_FORM — atribute solve v tomto případě obsahuje identifikátor úkolu v externím systému. GUI si poté podle hodnoty atributu origin a solve „vyžádá" JSON definici formuláře od systému Camunda.

Interní formuláře

V případě využití interního formuláře v rámci GUI obsahuje atribute solve u uživatelského úkolu referenci na interní formulář v následujícím formátu:

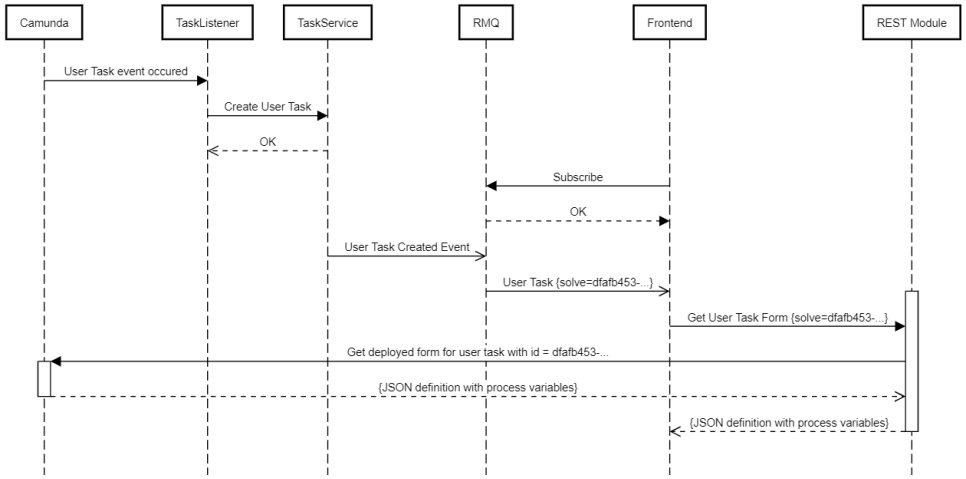
```
/<název modulu>?<query parametry>
```

Příklad hodnoty kterou může políčko solve pro interní formuláře nabývat je následující:

```
/evaluation?evaluationId=12323&showBig=true
```

Generované formuláře

Na diagramu 5.7 je možné vidět, jakým způsobem probíhá vykreslování generovaného formuláře z pohledu GUI. Jakmile uživatelský úkol doputuje standardní cestou popsanou již na obrázku 5.4 na Front-end, tak si Front-end pomocí synchronního API získá specifikaci formuláře v JSON formátu, včetně hodnot procesních proměnných pomocí REST modulu, který je součástí centrálně spravované Camunda instance. Dle této JSON definice a hodnot proměnných je Front-end schopný vykreslit formulář pro uživatele.



Obrázek 5.7: Generované formuláře, zdroj autor

Tento způsob vykreslení formulářů je blíže popsán v podsekci 5.2.4 týkající se dynamických (generovaných) formulářů.

5.2.2 TaskListener

Komponenta TaskListener má odpovědnost za odesílání aktualizace stavů uživatelských úkolů ze systému Camunda / Eprocesy. Task Listener je rozšíření procesního engine Camunda.

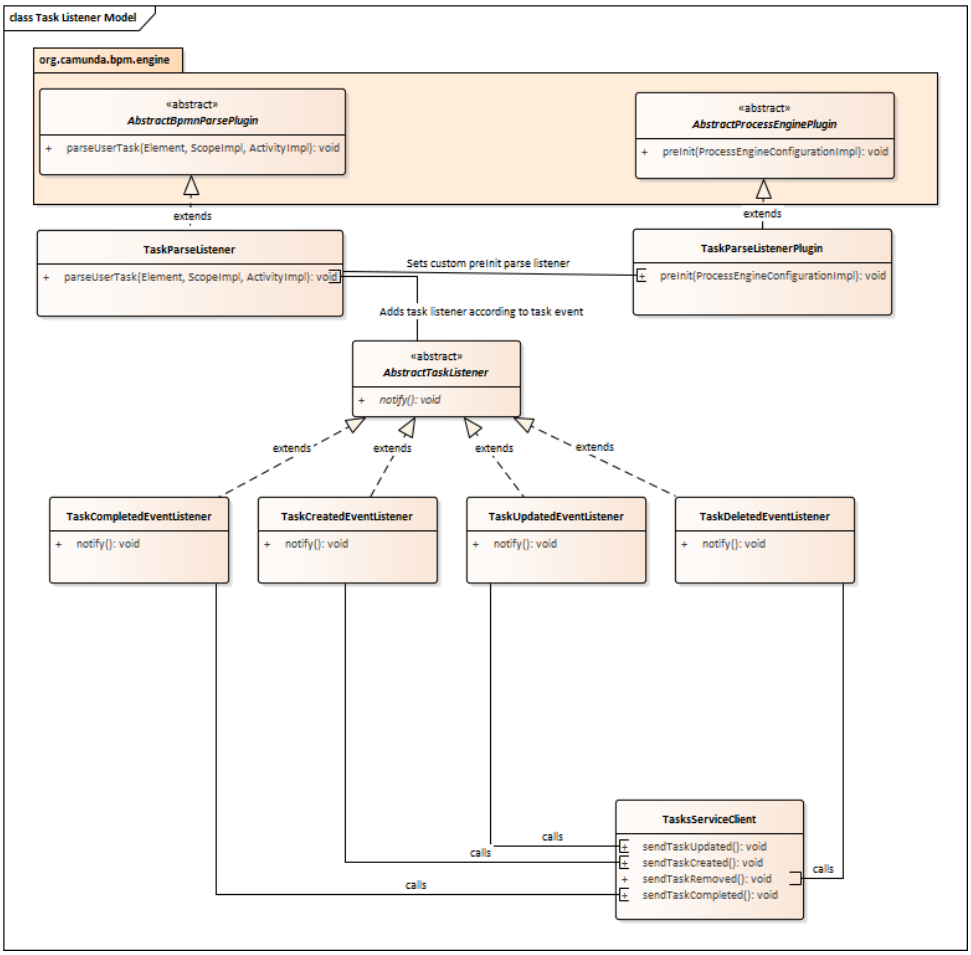
Komponenta má následující strukturu:

- parse-plugin — část rozšíření odpovědná za nastavení společných tříd, jako listeners pro události na všechny uživatelské úkoly. Při nasazení nového procesu se při parsování BPMN modelu vezmou všechny uživatelské úkoly v rámci dané definice a navážou se na ně listenery.

- task-listeners — třídy reagující na události nad uživatelským úkolem, mají odpovědnost za odeslání aktualizace do TaskService, která je blíže popsána v další podsekci 5.2.3.

- rest-client — klienti pro externí služby, v případě systému Eprocesy se jedná především pro REST klienta služby TaskService.

Na následujícím UML class diagramu 5.8 jsou znázorněny jednotlivé třídy v rámci Task Listener rozšíření.



Obrázek 5.8: Task Listener class diagram, zdroj autor

5.2.3 TaskService

TaskService je komponenta systému Hub, se kterým je systém Eprocesy silně integrován. Tato komponenta má odpovědnost za udržování stavu a správu uživatelských úkolů z více integrovaných systémů. Struktura této komponenty a její hlavní části jsou následující:

- REST rozhraní — Rozhraní je rozděleno na dvě části. Interní část poskytuje rozhraní pro interní systémy, například pro vytváření nových ukolů. Druhá část slouží pro potřeby Front-end aplikace, například pro získání aktuálních úkolů pro uživatele.
- Servisní vrstva — Tato vrstva obsahuje veškerou komplexnější logiku, například pro filtrování nebo zakládání úkolů.
- Databázová vrstva — TaskService využívá NoSQL databázi MongoDB pro ukládání stavu úkolů.
- RMQ Publisher — Část systému odpovědná za odesílání zpráv do message broker RabbitMQ.

Technologie

TaskService je aplikace vytvořená za pomocí jazyka Java 17 a využívá frameworku Spring Boot ve verzi 2.7.5. Zároveň využívá build systém Maven 3. Pro svůj běh v testovacím a produkčním prostředí využívá technologii Docker Swarm.

5.2.4 Dynamické formuláře

V rámci systému Eprocesy existuje první verze dynamických (generovaných) formulářů na základě JSON definice. Součástí této podsekce je popis současného stavu tohoto systému.

Systém dynamických formulářů sestává ze dvou hlavních částí a to GUI a backend části. GUI část je odpovědná za správné vykreslení formuláře podle JSON definice, kterou dostane podle identifikátoru aktivity v rámci procesu (uživatelský úkol nebo start event). Backend část je odpovědná za správné poskytnutí JSON definice pro daný úkol a následné správné zpracování operací nad tímto úkolem, jako je například splnění nebo uložení úkolu. Příklad komunikace těchto částí je možné nalézt v diagramu 5.5 na konci sekce 5.2.1.

JSON struktura

Definice formuláře má předepsanou JSON strukturu, kterou musí každý soubor s definicí formuláře splňovat, tak aby bylo možné formulář v GUI správně vykreslit.

Struktura formuláře vypadá následovně:

- description — metadata formuláře, jeho popis a dodatečné informace.
  - info — informativní box vykreslený v GUI.
  - paragraphs — popis formuláře ve formě jednotlivých kroků procesu.
    - title — titulek kroku.
    - text — text kroku.
    - active — informace zdali je krok aktivní.
    - icon — klíč pro ikonu.
- formSections — jednotlivé sekce s funkčními prvky formuláře.
  - sectionId — unikátní identifikátor sekce.
  - label — popis dané sekce.
  - formFields — vstupní formulářové prvky.
    - variable — název proměnné daného vstupního prvku.
    - value — hodnota vstupního prvku.
    - formType — určuje typ vstupního prvku, může nabývat jedné z osmi hodnot uvedených dále v této podsekci.

        - label — informativní popis vstupního prvku.
        - properties — vlastnosti vstupního prvku, jsou závislé na typu vstupního prvku a jsou popsány dále v této podsekci.
            - key — klíč vlastnosti.
            - value — nastavení vlastnosti.
    - properties — vlastnosti lze stejně jako u jednotlivých prvků aplikovat na celou sekci. Vlastnosti jsou popsány dále v této podsekci.
        - key — klíč vlastnosti.
        - value — nastavení vlastnosti.

- formActions — ve výchozím stavu se u formuláře nezobrazují žádná tlačítka. Je čistě na vývojáři aby styl a akci tlačítka definoval pomocí tohoto pole.
    - variable — název proměnné do které se po stisknutí tlačítka odešle hodnota true.
    - text — text tlačítka.
    - styleType — u každého tlačítka jsou definovány dva styly PRIMARY a SECONDARY.
    - actionType — u každého tlačíka jsou definovány dva druhy operace SUBMIT a SAVE, v závislosti na definici je při kliknutí na tlačítko zavolána odpovídající služba.
    - icon — klíč pro ikonu tlačítka.
    - confirmDialog — pokud je definován, je uživateli po kliknutí zobrazen potvrzovací dialog.
        - title — titulek potvrzovacího dialogu.
        - text — text potvrzovacího dialogu.
        - confirmButton — text potvrzovacího tlačítka v dialogu.
        - cancelButton — text tlačítka které zruší akci dialogu a vrátí se zpět na formulář.

- additionalVariables — dodatečné proměnné formuláře. V případě že je využita některá z vlastností závislá na proměnné, která nemá přiřazený žádný formField.
    - key — název proměnné.
    - value — hodnota proměnné.

Ukázka definice reálného formuláře je součástí přílohy C.

Výčet typů vstupních prvků definovaných pomocí atributu formType je následující:

- `STRING` — textový vstup.
- `NUMBER` — číselný vstup.

- `DATE` — vstupní prvek pro datum, jak textový, tak ve stylu datepicker.
- `BOOLEAN` — jednoduchý zaškrtávací vstupní prvek ve stylu checkbox.
- `SELECT` — nabídka ve stylu dropdown. Případně lze vykreslit jako zaškrtávací seznam nebo takzvaný multiselect, který umožňuje výběr více možností.
- `ENUM` — jednoduchý SELECT obsahující jen pevný výčet možností.
- `FILE` — vstupní prvek pro výběr jednoho nebo více souborů.
- `ADDRESS` — Skládá se z polí: street - String, city - String, country - String, postcode - String.

Jak již bylo zmíněno, jednotlivé prvky a sekce je možné ovlivňovat pomocí atributu properties, který může obsahovat páry klíč hodnota s nastavením vlastností. V případě properties u kterých je jako typ uveden Expression je možné zadat textové hodnoty "true"nebo "false"a nebo název proměnné podle které se má daná vlastnost vyhodnotit plus negaci dané proměnné pomocí znaku "!". Příklady hodnot které tedy může Expression výraz nabývat jsou následující:

- `true` — boolean hodnota
- `false` — boolean hodnota
- `showCompleteAddress` — Název proměnné
- `!showCompleteAddress` — Negace proměnné

Seznam vlastností které je možné definovat na úrovni sekcí je součástí přílohy D. Kompletní seznam vlastností, z nichž některé jsou specifické pro daný typ vstupního prvku, které je možné definovat pro jednotlivé vstupní prvky jsou součástí přílohy E.

GUI část

Vykreslování formulářů na základě JSON definice je součástí komponenty frontend Hub, jak komponenta zapadá do celkové architektury systému je možné nalézt v podsekci 5.1.1.

Frontend aplikace využívá silně typový jazyk Typescript a UI framework React. Další technologie použité pro vývoj GUI části jsou následující:

- Zustand — knihovna pro práci se sdíleným stavem v rámci React aplikací.
- styled-components — knihovna umožňující flexibilním způsobem využívat CSS v rámci React aplikace.
- Create React App — build systém s vlastním vývojářským serverem.
- React Router — knihovna pro implementaci dynamického routování v rámci single page aplikace.

Camunda část

V této části je popsána komponenta pro získání definice formuláře z BPMN modelu pro daný uživatelský úkol a následné odeslání do GUI, včetně operací `complete` a `save` nad uživatelským úkolem. Tato komponenta vznikla jako součást API umožňující pracovat s procesním engine Camunda. V podsekci 5.1.1 o architektuře systému Hub a Eprocesy je tato komponenta označena jako Camunda Proxy.

Camunda Proxy je komponenta napsaná v jazyce Java 17, využívá webového frameworku Spring Boot. Proxy je provozována jako součást aplikačního serveru Tomcat, kde se zároveň nachází i procesní engine Camunda, díky tomu může plně využívat Java API procesního enginu. Způsoby využívání procesního engine a různé architektury jsou blíže popsány v teoretické části této práce, v podsekci 3.3.1.

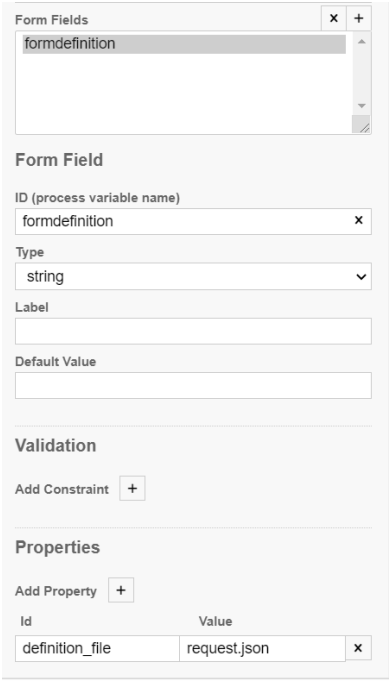
Definice formuláře ve formátu JSON se nahraje do repozitáře s procesem, který je strukturován následovně:

- `/forms` — složka obsahující JSON soubory s formuláři.
- `/definitions` — složka obsahující BPMN a DMN soubory.
- `/scripts` — složka obsahující Javascript soubory se skripty pro procesy.

Pro nasazení procesu se využívá Camunda REST API, kam se nahraje obsah všech složek z repozitáře pomocí jednoduchého bash skriptu. Vývojář může na daný formulář následně odkázat z BPMN modelu pomocí názvu JSON souboru. Definice se provede do camunda:formField prvku se speciálním identifikátorem formdefinition, který musí mít vlastnost s pevně daným identifikátorem `definition_file`.

```
<bpmn:startEvent id="StartEvent_UnRequest" name="Nová žádost">
<bpmn:extensionElements>
<camunda:formData>
  <camunda:formField id="formdefinition" type="string">
    <camunda:properties>
      <camunda:property id="definition_file"
      value="start.json" />
    </camunda:properties>
  </camunda:formField>
</camunda:formData>
</bpmn:extensionElements>
</bpmn:startEvent>
```

Ukázka definice takového formuláře z Camunda Modeler je na obrázku 5.9.



Obrázek 5.9: Definice formuláře v Camunda Modeler, zdroj autor

V případě že chce GUI získat definici formuláře pro daný uživatelský úkol v JSON formátu včetně hodnot proměnných z dané procesní instance, tak může využít REST API v komponentě Camunda Proxy. Základní služby jsou následující:

- `GET /form/definition/formId` — vrátí definici formuláře obohacenou o hodnoty procesních proměnných pro danou procesní instanci.
- `POST /form/submit` — uloží stav proměnných v rámci formuláře do dané procesní instance a splní daný úkol.
- `POST /form/save` — uloží stav proměnných v rámci formuláře do dané procesní instance.

5.2.5 Identifikované problémy s návrhem

V rámci této podsekce jsou popsány hlavní identifikované problémy se správou uživatelských úkolů v rámci systémů Eprocesy a Hub.

Napojení dalších systémů

Napojení dalších systémů lze rozdělit na dva případy:

1. Systém založený na platformě Camunda.
2. Systém bez využití platformy Camunda.

V případě systémů které využívají platformu Camunda aktuálně neexistuje jednoduchý způsob jak začít využívat systém dynamických formulářů a vyžadavalo by to složitou implementaci vlastního REST rozhraní, které by bylo ekvivalentem komponenty Camunda Proxy. Komponenta Camunda Proxy a systém dynamických formulářů je blíže popsán v podsekci 5.2.4.

V případě systémů nevyužívajích platformu Camunda je problém v případě že se jejich životní cyklus nebo struktura uživatelského úkolu liší o definice uživatelského úkolu v TaskService. Struktura i životní cyklus uživatelských úkolů jsou blíže popsány v sekci 3.2 a podsekci 5.2.1 pro úkoly v systému Hub a Eprocesy.

Synchronní aktualizace úkolů

V rámci procesního engine může vznikat značné množství událostí nad úkoly, pro každou z těchto událostí musí Tasklistener provést synchronní volání Taskservice, což způsobuje problémy s nadbytečnou komunikací na síti a s výkonem na straně procesního engine. Styl komunikace fire-and-forget, kdy aktualizaci úkolu asynchronně Camunda odešle v případě vzniku události a nečeká na odpověď by byla v tomto případě vhodnější volbou.

Dynamické formuláře

Problémy spojené se systémem dynamických formulářů jsou popsány zvlášť a to v následujícím seznamu:

- Chybějící proces vytváření a řízení nových komponent. V případě více systémů využívajících systém dynamických formulářů se začnou požadavky na komponenty lišit a je potřeba je formálním způsobem řídit. Takovýto mechanismus v současné době pro systém dynamických formulářů chybí.
- Použitelnost pro vývojáře. Pro propojení formuláře a uživatelského úkolu neexistuje vlastní rozhraní v modelovacím nástroji.
- Chybějící nástroj pro vývoj dynamických formulářů. Vývojáři musí využívat jedno z testovacích prostředí, kam musí složitě nasadit proces, aby mohli otestovat, jak bude vypadat formulář po vykreslení.
- Použitelnost systému pro vykreslení formulářů v rámci jiných systémů. Systém pro vykreslení formuláře podle JSON definice je navázat na Frontend aplikaci Hub a není ji možné snadno použít samostatně, případně ji integrovat do jiného systému.
- Verzování formulářů. Formulář existuje vždy jen v poslední verzi. V případě že by existovala běžící instance procesu ve starší verzi a vyžadovala by starší verzi formuláře, tak by v případě nasazení nové verze procesu, s novým formulářem, bylo nutné provést migraci staré verze procesu, což nemusí být žádoucí.
- Změny ve struktuře JSON souboru s definicí formuláře by znamenaly i změna v backend aplikaci, díky aktuální podobě rozhraní.

Auditní data

V případě událostí nad uživatelskými úkoly se ze systému TaskService lze dozvědět vždy jen aktuální stav a systém neukládá auditní data. V případě že by bylo potřeba auditovat změnu stavu úkolů, byl by za poskytnutí a uložení těchto dat odpovědný pouze externí systém. Příklad auditních dat, které by mohl systém ukládat je například změna řešitele, nebo delegace úkolu.

Analýza nad uživatelskými úkoly

Jedním z důležitých požadavků pro reporting nad procesy je možnost provádět komplexnější vyhledávání a analýzu nad uživatelskými úkoly. Pokud například vedoucí studijního oddělení chce vyhledat konkrétní úkol jeho podřízených nebo chce zobrazit metriky ohledně délky řešení daných úkolů, nemá k tomu aktuálně dostupný žádný nástroj.

5.2.6 Shrnutí

V rámci této sekci byl popsán stávající stav správy uživatelských úkolů v rámci systémů Eprocesy a Hub. Byla zde popsána struktura uživatelských úkolů, a jakým způsobem se liší od strukturu úkolů v rámci platformy Camunda. Dále zde byly detailněji popsány jednotlivé komponenty, které jsou důležité pro správu uživatelských úkolů v rámci systémů. Velký důraz byl kladen na popis systému dynamických formulářů. V neposlední řadě byly v této sekci identifikovány nejdůležitejší nedostatky stávajícího stavu správu uživatelských úkolů, které budou adresovány v návrhové části této práce.

Kapitola 6
Design

V rámci této kapitoly je popsán návrh rozšíření pro správu uživatelských úkolů pro obecné využití v rámci jiných systémů. V sekci 6.1 je představen nový návrh a popsáno, jakým způsobem adresuje problémy popsané v kapitole 5. Pro toto rozšíření byly dále navrženy související procesy popsány v sekci 6.2.

6.1 Návrh rozšíření

V této sekci je popsán návrh rozšíření správy uživatelských úkolů v rámci systémů Eprocesy a Hub. Návrh architektury softwarového systému sestává z následujících částí, kterou jsou popsány v této části práce:[34]

- Softwarové a systémové komponenty, spojení a omezení.
- Požadavky zainteresovaných stran.
- Vysvětlení, jak a proč daný návrh řešení naplňuje požadavky zainteresovaných stran.

Požadavky zainteresovaných stran vychází ze zpětné vazby uživatelů systému a ze zkušeností s provozem systémů Eprocesy a Hub. Požadavky primárně vychází z problémů se stávajícím návrhem, identifikovaných v podsekci 5.2.5.

6.1.1 Požadavky, předpoklady a omezení

Tato podsekce slouží jako podklad pro důležitá rozhodnutí týkající se návrhu systému pro správu uživatelských úkolů, obsahuje klíčové požadavky na systém, včetně omezení a dalších předpokladů, které je nutné vzít v potaz při návrhu systému.

Požadavky

Následující část obsahuje požadavky na návrh rozšíření systému pro správu uživatelských úkolů v rámci projektu Eprocesy a Hub. Požadavky primárně

vychází z identifikovaných problémů popsaných blíže v podsekci 5.2.5. Jsou rozděleny na požadavky koncových uživatelů a požadavky vývojářů.

Požadavky koncových uživatelů:

1. Jako administrátor nebo vedoucí studijního oddělení požaduji, aby bylo možné vyhledávat full-textově nad uživatelskými úkoly.
2. Jako vedoucí studijního oddělení požaduji, aby byly auditovatelné operace nad uživatelskými úkoly.
3. Jako administrátor a vedoucí studijního oddělení požaduji, aby bylo možné reportovat nad uživatelskými úkoly následující metriky:
   - Celková délka řešení úkolu.
   - Délka řešení úkolu daným uživatelem.
   - Počet změn řešitele nad úkolem.
   - Čas po který byl úkol bez řešitele.
4. Jako administrátor požaduji, aby systém umožňoval agregovat úkoly z více systémů založených na procesním engine Camunda.
5. Jako uživatel požaduji, aby systém umožňoval vyřešit uživatelský úkol pomocí formuláře.
6. Jako uživatel požaduji, aby systém umožňoval uložit rozpracovaný formulář k uživatelskému úkolu.

Požadavky analytiků a vývojářů procesních aplikací:

1. Jako analytik požaduji nástroj, který mi umožní snadno vyvíjet dynamické formuláře v lokálním prostředí.
2. Jako analytik požaduji nástroj, který mi umožní vyzkoušet finální zobrazení dynamického formuláře v grafickém rozhraní, přímo z JSON definice dynamického formuláře.
3. Jako analytik požaduji nástroj, který mi umožní vyzkoušet zobrazení formuláře úkolu v rámci nasazeného procesu v lokálním prostředí.
4. Jako vývojář požaduji rozšíření systému dynamických formulářů o možnost vybrat verzi formuláře pro konkrétní nasazený uživatelský úkol v rámci procesu.
5. Jako analytik požaduji rozšíření nástroje Camunda Modeler o snadný způsob jak zadat identifikátor a verzi formuláře pro danou aktivitu.
6. Jako vývojář požaduji ukázku použití dynamických formulářů mimo systém Hub a Eprocesy.

Předpoklady

V rámci této části jsou popsány předpoklady pro vznik rozšíření systému správy uživatelských úkolů, tyto předpoklady ovlivňují zvolený návrh a především popisují, jakým způsobem se plánuje využívání systému do budoucna:[35]

1. Rozšíření pro správu uživatelských úkolů budou využívat další systémy využívající procesní platformu Camunda.
2. Systém dynamických úkolů bude dále rozšiřován o komponenty podle požadavků nových integrovaných systémů.

Omezení

Následující omezení ovlivňují návrh rozšíření, především udávají, jakým způsobem musí být rozšíření systému pro správu uživatelských úkolů navrženo, aby splňovalo požadavky na využití v rámci existujícího systému s minimálním dopadem na existující projekty:

1. Použité technologie jsou ve shodě s aktuálně využívanými technologiemi v rámci projektů Eprocesy a Hub.
    a. Backend aplikace budou napsané v jazyce Java 17.
    b. Backend aplikace budou využívat framework Spring Boot verze 2.7.5.
    c. Backend aplikace budou využívat build systém Maven 3.
    d. Pro ukládání a indexování dat se bude využívat systém Elasticsearch 7.17.3.
2. Navržená architektura odpovídá koncepci systémů Eprocesy a Hub a snaží se dodržovat zásady servisně orientované architektury.
3. Nově vznikající API jsou pro frontend poskytována ve formě GraphQL API.
4. Rozšíření bude možné využívat v rámci stávající infrastruktury projektů Eprocesy a Hub.
5. Rozšíření bude využívat existující funkční celky.
    a. Vykreslování dynamických formulářů bude vycházet z otestované verze části frontend aplikace, tak aby se snížil dopad na existující aplikace.
6. Komponenty budou využívat distribuovaný verzovací systém Gitlab, včetně integrovaného package registry.
7. Pro komponenty bude v rámci Gitlab pravidelně spouštěn build a testy.

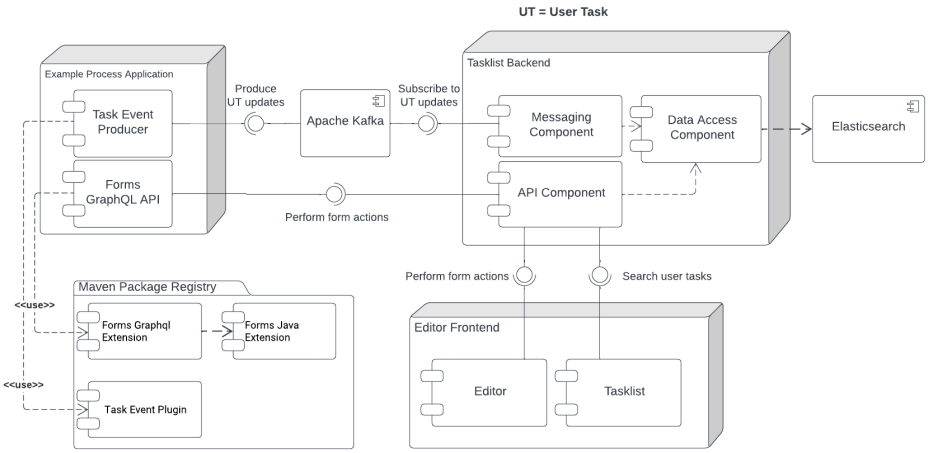
6.1.2 Koncept řešení

Účelem této podsekce je poskytnout přehled navržené architektury a hlavních komponent. Detailní popis jednotlivých komponent je uveden v dalších podsekcích, zahrnují také zvažované alternativy a odůvodnění daného přístupu.

Řešení by mělo sestávat z několika hlavních částí, které jsou pro přehlednost krátce popsány v následujícím seznamu:

- Vytvoření Java backend knihovny poskytující Java API pro správu úkolů s dynamickými formuláři v platformě Camunda.
- Vytvoření Java backend knihovny poskytující GraphQL API pro správu úkolů s dynamickými formuláři v platformě Camunda.
- Vytvoření služby agregující úkoly z více procesních aplikací využívajících platformu Camunda, umožňující full-textové vyhledávání a datovou analýzu nad úkoly. Služba zároveň ukládá auditní informace o uživatelských úkolech, umožňující generovat reporty.
- Vytvoření Java backend rozšíření umožňující odesílat události nad uživatelskými úkoly do messaging systému Apache Kafka a návrh související struktury Kafka topiců.
- Rozšíření modelovacího nástroje Camunda Modeler o specifikaci formulářů a verzování.
- Vytvoření editoru pro dynamické formuláře umožňující snadný vývoj uživatelských úkolů využívajících dynamické formuláře. Editor zároveň slouží jako ukázka vykreslování formulářů mimo systém Hub a práci se službou agregující úkoly z více procesních aplikací.
- Návrh souvisejících procesů pro analýzu a vývoj procesů využívajících uživatelské úkoly s dynamickými formuláři.

Na diagramu 6.1 je možné vidět koncept návrhu řešení a přehled komunikace v rámci daného systému.



Obrázek 6.1: Návrh architektury — koncept, zdroj autor

V rámci návrhu jsou znázorněny následující komponenty:

- `Ukázková procesní aplikace` — aplikace napsaná v jazyce Java, za pomocí frameworku Spring Boot. Obsahují embedded procesní engine Camunda. Způsob provozování embedded procesního engine je detailněji popsán v podsekci 3.3.2. Zároveň aplikace importuje rozšíření pro odesílání úkolů do message broker systému Apache Kafka a rozšíření pro GraphQL API.
- `Package Registry` — ukládá artefakty vytvoření za pomocí nástroje Maven pro Java projekty.
  - `Task Event Plugin` — rozšíření umožňující odesílat události týkající se uživatelských úkolů v rámci dané procesní aplikace využívající procesní engine Camunda.
  - `Forms Extension` — rozšíření poskytující rozhraní, které umožňuje pracovat s uživatelskými úkoly v rámci dané procesní aplikace. Základní operace, které umožňuje provádět nad uživatelským úkolem jsou uložení a dokončení úkolu. Zároveň rozhraní umožňuje získat JSON definici dynamického formuláře pro vykreslení v GUI a hodnoty procesních proměnných.
- `Apache Kafka` — durable message broker, umožňující asynchronní odeslání událostí nad uživatelskými úkoly, je detailněji popsán v podsekci 4.1 o messaging systémech.
- `Elasticsearch` — analytický engine sloužící pro ukládání, analýzu, vyhledávání a filtrování dat, je detailněji popsán v podsekci 4.3.
- `TaskList Backend` — aplikace napsaná v jazyce Java, za pomocí frameworku Spring Boot.

  - `Messaging Component` — komponenta zpracovávající události nad uživatelskými úkoly. Provádí uložení do auditních dat v rámci Elasticsearch.
  - `API Component` — komponenta zprostředkovávající rozhraní pro dokončení, uložení a vyhledávání uživatelských úkolů z různých systémů a zároveň rozhraní pro vyhledávání úkolů.
- `Editor Frontend` — editor dynamických formulářů, využívaný pro snadný lokální vývoj dynamických formulářů, který zároveň slouží jako ukázka použití dynamických formulářů mimo GUI front-end Hub a Eprocesy.

6.1.3 Rozhraní v systému

Důležitým rozhodnutím je, jakým způsobem budou poskytována a specifikována rozhraní backend TaskExtension rozšíření a TaskService. Vzhledem k omezení, které uvádí že nově poskytovaná rozhraní pro frontend aplikace budou ve formě GraphQL API je tento způsob detailněji popsán v rámci sekce 4.2 v teoretické části této práce

6.1.4 Messaging systém

Technologie popsané v rámci 4.1 — ActiveMQ, RabbitMQ a Apache Kafka sdílí sadu společných vlastností a funkcionality. Pro implementaci této práce byl zvolen systém Apache Kafka místo ActiveMQ nebo RabbitMQ, a to především z následujících dvou klíčových důvodů:

- Hlavním důvodem je možnost využít Apache Kafka pro takzvaný replay dat, kdy máme stav nějaké entity uložený jako sekvenci událostí a můžeme tento stav vždy z událostí rekonstruovat.
- Je lépe škálovatelný a umožňuje snadno přidávat nové příjemce zpráv bez dopadu na výkon.
- Umožňuje ukládání historie zpráv, pro další analytické zpracování. Historie zároveň slouží jako spolehlivý auditní log událostí nad danou entitou.
- Je dostatečné jednoduché směrovací schéma pro doručování zpráv.

6.1.5 Java rozšíření

V rámci této podsekce je popsána komponenta umožňující pracovat s dynamickými formuláři v rámci procesní aplikace pomocí Java API. Rozšíření je určeno pro procesní engine Camunda, který je součástí Spring Boot aplikace jako knihovna, tedy v embedded modelu. Jednotlivé možnosti nasazení a provozování procesního engine Camunda jsou detailněji popsány v rámci podsekce 3.3.2.

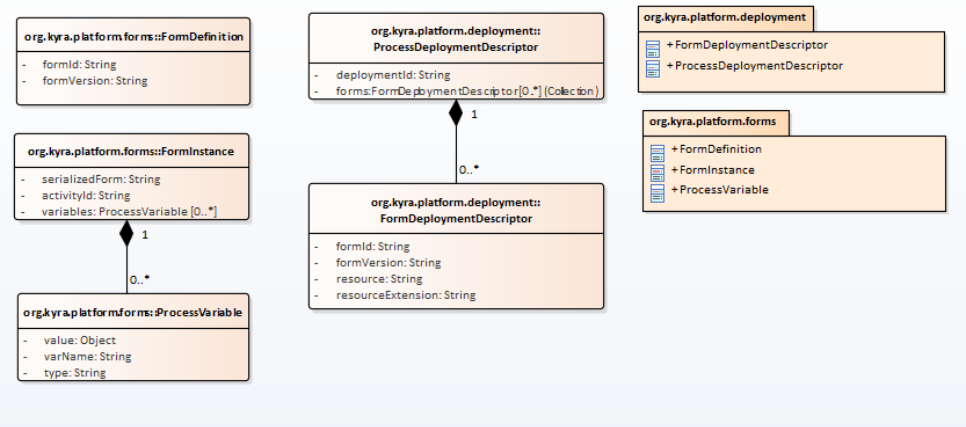
Rozšíření má poskytovat následující funkcionalitu:

- `FR-TE-01` — Umožní nasadit proces, který na elementech Start Event nebo User Task může obsahovat atributy určují identifikátor a verzi formuláře, který se pro daný Start Event nebo User Task mají vykreslit v GUI aplikace. Tyto atributy jsou *formId* a *formVersion*.
- `FR-TE-02` — Umožní získat přes API pro instanci daného User Task s daným identifikátorem serializovaný formulář a hodnoty procesních proměnných v rámci daného kroku.
- `FR-TE-03` — Umožní přes API uložit hodnoty pro daný User Task podle identifikátoru dané instance úkolu do procesních proměnných.
- `FR-TE-04` — Umožní přes API splnit daný User Task podle identifikátoru dané instance úkolu a uložit hodnoty do procesních proměnných.

API bude servisně orientované a bude poskytovat dvě služby, první z nich —`FormDefinitionService`, která umožňuje získat definice formuláře z BPMN modelu a druhou — `FormInstanceService`, která umožňuje pracovat s konkrétní instancí procesu a formuláře. Detailní rozhraní je součástí přílohy se zdrojovými kódy obsahující Javadoc. Rozšíření zároveň bude implementováno jako knihovna, kterou je možné importovat jako Maven dependency do libovolného Spring Boot projektu využívajícího procení engine Camunda v embedded variantě.

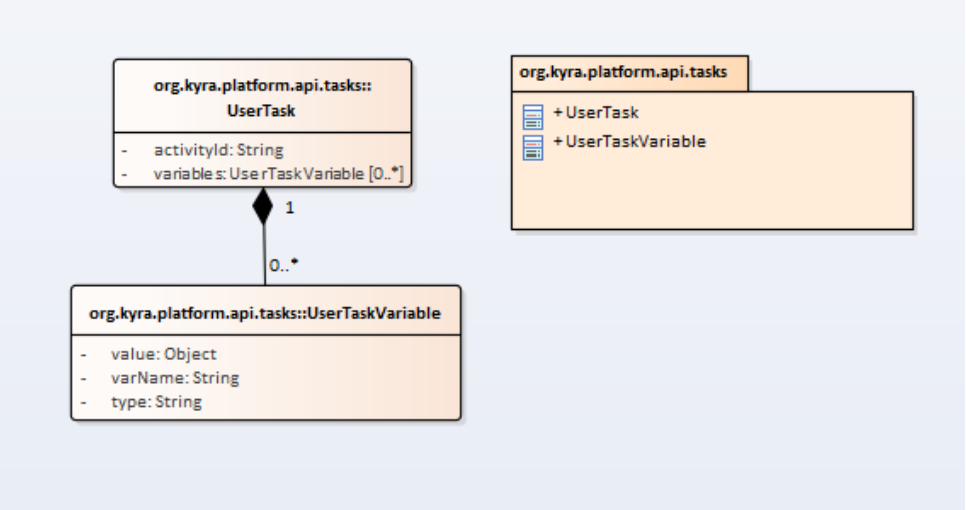
Návrh datového modelu

Na diagramu 6.2 je možné vidět návrh datového modelu backend rozšíření pro práci s dynamickými formuláři, tato část se týká funkčních požadavků `FR-TE-01` a `FR-TE-02`.



Obrázek 6.2: Datový model — Formuláře, zdroj autor

Diagram 6.3 obsahuje návrh datového modelu zahrnující `FR-TE-03` a `FR-TE-04`.



Obrázek 6.3: Datový model — Uživatelské úkoly, zdroj autor

Návrh logiky služeb

Logiku služeb detailněji popisují aktivity diagramy, které jsou součástí přílohy F. Jednotlivé služby implementují funkční požadavky na backend rozšíření pro dynamické formuláře, specifikované na začátku podsekce.

Diagram F.1 popisuje službu, která umožní získat definici formuláře v JSON formátu podle ID aktivity, jde tedy o implementaci logiky pro službu popsanou ve `FR-TE-01`. Diagram F.2 obsahuje postup získání definice formuláře z BPMN modelu dané procesní instace, jedná se tedy o popis požadavku `FR02`. Diagram F.3 obsahuje získání objektu s definicí formuláře z modelu procesní instance. Uložení úkolu popsané v rámci `FR-TE-03` je znázorněno v rámci diagramu F.4. Dokončení uživatelského úkolu podle funkčního požadavku `FR-TE-04` je zobrazeno na diagramu F.5.

Zvolené technologie

Navrhované technologie vychází z omezení specifikovaných v podsekci 6.1.1.

Seznam technologií s odpovídajících verzí:

- Java 17
- Maven 3
- Spring Boot 2.7.5
- Camunda 7.17.0

Součástí použitých technologií jsou také podpůrné knihovny, jako je například Mapstruct nebo Lombok.

6.1.6 GraphQL rozšíření

V předchozí podsekci 6.1.5 bylo popsáno Java rozhraní, pomocí kterého je možné ovládat dynamické formuláře v rámci procesní aplikace. Aby bylo možné využít rozhraní pro práci s dynamickými formuláři po síti, je součástí návrhu ještě rozšíření, které za využití knihovny s Java API pro práci s dynamickými formuláři vystavuje GraphQL rozhraní. Důvodem proč nejsou rozšíření spojena v jedno je vytvoření dodatečné vrstvy abstrakce, tak aby mohlo podle potřeby vzniknout další rozšíření umožňující pracovat s dynamickými formuláři například pomocí REST, gRPC nebo dalšího libovolného rozhraní, za využití existující Java knihovny pro dynamické formuláře, popsané v podsekci 6.1.5.

Popis rozhraní

Rozhraní bude popsáno pomocí GraphQL schématu. GraphQL je blíže popsáno v podsekci 6.1.3. Na snímku 6.4 je možné vidět schéma pro rozšíření umožňující získat formulář pro danou aktivitu, pomocí query `getFormInstace`. Dále umožňuje provést dvě mutace, `submitForm` pro odeslání formuláře a `saveUserTask` pro uložení uživatelského úkolu.

```
type Query {
    getFormInstance(
        activityId: String!
    ): FormInstance
}

type Mutation {
    submitForm(
        form: FormInstanceInput!
    ): String

    saveUserTask(
        form: FormInstanceInput!
    ): String
}
```

```

# INPUTS
input FormInstanceInput {
    activityId: String!
    variables: [ProcessInputVariable!]!
}

input ProcessInputVariable {
    varName: String!
    type: String!
    value: String
}
# TYPES
type FormInstance {
    activityId: String!
    serializedForm: String!
    variables: [ProcessVariable!]
}

type ProcessVariable {
    varName: String!
    type: String!
    value: String
}
```

Obrázek 6.4: Schéma rozšíření pro práci s uživatelskými úkoly a formuláři, zdroj autor

Schéma je také součástí přílohy obsahující zdrojové kódy GraphQL rozšíření.

Technologie

Navrhované technologie vychází z omezení specifikovaných v podsekci 6.1.1.

Seznam technologií s odpovídajících verzí:

- Java 17
- Maven 3
- Spring Boot 2.7.5
  - Spring for GraphQL
- Camunda 7.17.0

6.1.7 Rozšíření Camunda Modeler

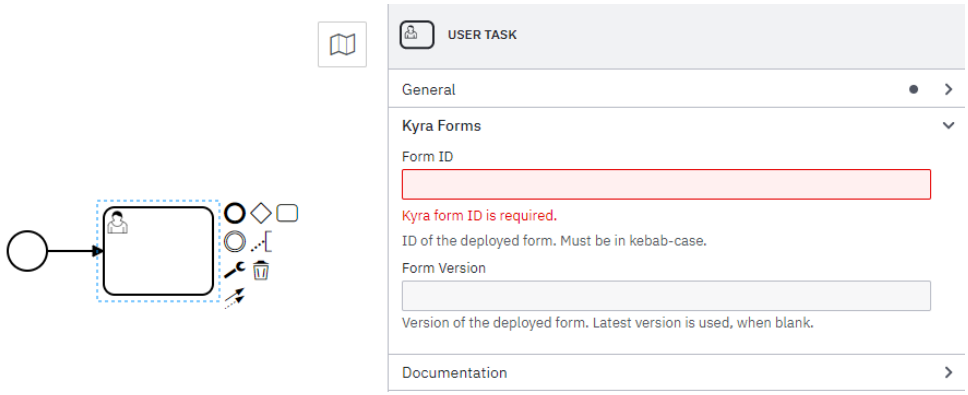
Nástroj Camunda Modeler umožňuje modelování a implementaci spustitelných BPMN a DMN modelů, jak bylo zmíněno v sekci 3.1 o komponentách a nástrojích posyktovaných platformou Camunda. Aby vývojáři mohli snadno definovat, jaký dynamický formulář a jaká konkrétní verze se má pro danou aktivitu použít, je potřeba rozšířit stávající grafické rozhraní nástroje Camunda Modeler o dva vstupní prvky pro identifkátor a verzi formuláře.

Funkcionalita, kterou by mělo rozšíření splňovat je následující:

- `FR-MOD-01` — V nástroji Camunda Modeler je možné pro element User Task definovat identifikátor a verzi formuláře.
- `FR-MOD-02` — V nástroji Camunda Modeler je možné pro element Start Event definovat identifikátor a verzi formuláře.
- `FR-MOD-03` Rozšíření musí validovat formát identifikátoru vůči regulárnímu výrazu `^([a-z][a-z0-9]*)(-[a-z0-9]+)*$` a pokud dojde k porušení, tak na něj upozornit uživatele.
- `FR-MOD-04` Rozšíření musí validovat formát verze vůči regulárnímu výrazu `^([0-9]+)\.([0-9]+)\.([0-9]+)` a pokud dojde k porušení, tak na něj upozornit uživatele.

Pro nástroj Camunda Modeler existuje množství rozšíření, které je možné do nástroje přidat.[1]. Grafický návrh rozšíření je možné vidět na obrázku 6.5.

[1] Camunda Modeler Rozšíření - `https://github.com/camunda/camunda-modeler-plugins`



Obrázek 6.5: Rozšíření Camunda Modeler, zdroj autor

Camunda Modeler využívá knihovny React.js. a pro jeho implementaci je možné využít šablonu poskytnutou Camunda vývojáři[2].

6.1.8 Kafka Rozšíření

Rozšíření umožňuje odesílat události vzniklé nad uživatelskými úkoly v systému Camunda do systému Apache Kafka, tento systém je blíže popsán v teoretické části 4.1 zabývající se messaging systémy.

Kafka rozšíření využívá Camunda Spring Eventing Bridge, pro snadné naslouchání nad událostmi nad uživatelskými úkoly v rámci procesního engine Camunda, popsaný blíže v podsekci 3.2.4 teoretické části práce, která popisovala událsoti vznikající v rámci procesního engine. Toto rozšíření dále odesílá vzniklé události nad úkoly do Kafka Topic s konfigurovatelným identifikátorem, počtem replik a partitions.

Návrh Apache Kafka topics

Vzhledem k povaze dat a požadavkům na zpracování byla navržena následující výchozí struktura Kafka Topics:

- Bude existovat jeden Kafka topic: `camunda.event.task`
- Kafka topic bude využívat jednu repliku.
- Kafka topic bude využívat partitioning podle klíče zprávy, kterou bude `definitionKey`[3] procesu v rámci kterého vznikla událost nad daným úkolem. Předpokládá se, že většina procesních aplikací v prostředí CZM bude obsahovat mezi jedním až deseti procesy. Zároveň tímto nastavením bude zaručeno, že události nad stejným úkolem budou uloženy ve správném pořadí a události budou rovnoměrně distribuovány mezi jednotlivé partitions.

[2] Šablona pro Camunda Modeler rozšíření - `https://github.com/camunda/camunda-modeler-plugin-example`

[3] Dokumentace třídy pro definici procesu — `https://docs.camunda.org/javadoc/camunda-bpm-platform/7.18/org/camunda/bpm/engine/repository/ResourceDefinition.html#getKey()`

Strukturu, včetně nastavení replik a počtu partitions je možné konfigurovat pomocí Spring application properties. V hlavičce zprávy bude pro snazší filtrování na straně, která zprávy přijímá uveden v atributu `eventName` jeden z následujících názvů událostí nad uživatelským úkolem: `create`, `assignment`, `complete`, `update`, `delete`, `timeout`.

Struktura zprávy vychází ze struktury uživatelského úkoly v rámci procesního engine Camunda. Tato struktura je detailněji popsána v teoretické částí této práce v podsekci 3.2.1. Avro schéma je součástí přílohy se zdrojovými kódy Kafka rozšíření.

Technologie

Navrhované technologie vychází z omezení specifikovaných v podsekci 6.1.1.

Seznam použitých technologií s odpovídajících verzí:

- Java 17
- Maven 3
- Spring Boot 2.7.5
- Apache Avro 1.11.0
  - Spring Kafka 2.8.3
  - Kafka Avro Serializer 7.1.1
- Camunda 7.17.0

6.1.9 Tasklist Backend aplikace

Aplikace Tasklist umožňuje především agregovat úkoly z více procesních aplikacích postavených nad procesní platformou Camunda. Díky napojení na systém Apache Kafka asynchronně zpracovává události nad uživatelskými úkoly a tyto úkoly a události ukládá do analytického nástroje ElasticSearch.

Tasklist poskytuje dvě rozhraní. Jedno pro čtení a full-textové vyhledávání uživatelských úkolů. Druhé rozhraní umožňující práci s dynamickými formuláři. Aplikace směřuje požadavky týkající se dynamických formulářů přímo na cílovou procesní aplikaci podle identifikátoru tenanta uvedeného jako atribut daného uživatelského úkolu.

Struktura aplikace

Aplikace je strukturována do následujících Maven modulů, každý s odpovídající odpovědností:

- app — základní modul, obsahuje základní konfiguraci a vstupní bod aplikace.
- api — poskytuje rozhraní pro front-end.

- data — obsahuje persistentní vrstvu a konfiguraci napojení na databázi.
- external — zajišťuje komunikaci s externími systémy, v tomto případě operace směřované na konkrétní procesní aplikace.
- messaging — obsahuje napojení na Apache Kafka a asynchronní zpracování událostí nad uživatelskými úkoly.
- service — byznys logika aplikace.

Popis rozhraní

Navrhované GraphQL rozhraní je součástí přílohy se zdrojovými kódy pro aplikaci Tasklist. Rozhraní je možné rozdělit na dvě logické části, první je část umožňující vyhledávat uživatelské úkoly a druhou částí je API pro práci s dynamickými formuláři.

Technologie

Navrhované technologie vychází z omezení specifikovaných v podsekci 6.1.1.
Seznam technologií s odpovídajících verzí:

- Java 17
- Maven 3
- Spring Boot 2.7.5
- Apache Avro 1.11.0
  - Spring Kafka 2.8.3
  - Kafka Avro Serializer 7.1.1
- Elasticsearch 7.17.3

6.1.10 Editor formulářů

Aplikace Editor umožňuje vývojářů a analytikům snadno implementovat a otestovat zobrazování dynamických formulář. Aplikace zároveň slouží jako demonstrace použití systému dynamických formulářů mimo systém Hub a Eprocesy. Další část aplikace slouží jako jednoduchá ukázka implementace grafického rozhraní vůči připravované backend aplikaci Tasklist. Aplikace bude naplňovat požadavky specifikované v rámci podsekce 6.1.1.

Aplikace bude rozdělena na dvě části, jedna sloužící pro vývoj dynamických formulářů a jako ukázka použití dynamických formulářů. Druhá sekce jako ukázka práce s Tasklist API.

Editor

V rámci této části aplikace budou dvě sekce. Jedna bude umožňovat zobrazit vykreslený formulář podle zadaných proměnných a JSON definice formuláře. Zároveň bude umožňovat měnit vlastnosti jednotlivých prvků a sekcí v rámci formuláře.

Sekce bude určena primárně pro vývojáře a analytiky implementující dynamicky vykreslované formuláře, kteří si budou chtít ověřit, zda-li se jejich formulář zobrazuje správně, ještě před nasazením samotného procesu. Tento nástroj zefektivní proces vývoje a omezí chyby vzniklé při vývoji formulářů v rámci procesů.

Druhá sekce bude umožňovat zobrazit dynamický formulář podle identifikátoru uživatelského úkolu v rámci běžícího procesu, včetně hodnot procesních proměnných. Aplikace bude také umožňovat provádět operace pro uložení uživatelského úkolu a odeslání formuláře. Vývojář nebo analytik implementující proces si tak bude moci snadno vyzkoušet průchod nasazeným procesem za využití dynamických formulářů ve svém lokálním prostředí.

Tasklist

Část aplikace označená jako Tasklist především demonstruje použití dynamických formulářů mimo aplikaci Hub a zároveň slouží jako ukázka rozhraní pro full-textové vyhledávání nad uživatelskými úkoly. Aplikace tedy obsahuje dva základní scénáře:

- Vyhledávání úkolů dle textového vstupu.
- Vykreslení dynamického formuláře po kliknutí na tlačítko „Solve".

Technologie

Navrhované technologie vychází z omezení specifikovaných v podsekci 6.1.1.

Seznam technologií s odpovídajících verzí:

- NodeJS 14.18.1
- npm 6.14.15
- React 18
- Typescript 4.2.4
- Styled Components 5.1.3
- Zustand 3.5.5
- Apollo Client 3.5.0

6.1.11 Shrnutí

Součástí této sekce byl návrh celkové architektury řešení a jednotlivých komponent, tak aby splňovaly požadavky, předpoklady a omezení stanovené na začátku této sekce. Implementace dle tohoto návrhu je součástí kapitoly 7 o implementaci celého řešení.

6.2 Související procesy

Součástí této sekce je návrh procesů pro práci s dynamickými formuláři. Pro popis procesů byla zvolena notace BPMN popsaná v rámci teoretické části v sekci 2.2 týkající se modelování procesů.

6.2.1 Řízení požadavků na nové komponenty

Pro řízení požadavků na nové komponenty pro systém dynamických formulářů se bude využívat systém Gitlab Issues, který slouží pro plánování práce na projektu a vytváření zadání. Pro zadání nového požadavku na komponentu bude vytvořena nová šablona sestávající z následujících částí:

- Souhrn
  - Identifikátor
  - Krátký popis
  - Typ
- UI/UX návrh
- Zmapovaný use case
- Popis chování
- Struktura
  - Atributy (label, componentId)
  - Podporované vlastnosti
- Ukázka v JSON formátu

Šablona pro zadání nového požadavku na komponentu dynamických formulářů je součástí přílohy se zdrojovými kódy pro front-end aplikaci Editor formulářů v podsložce `.gitlab/issue_templates`.

Celý proces zapracování nového požadavku na dynamické formuláře od samotného zadání požadavku uživatele je možné vidět na diagramu procesu G.1, který je součástí přílohy.

6.2.2 Implementace procesů

Na diagramu G.2 je možné vidět, jakým způsobem je integrován vývoj dynamických formulářů do vývoje procesů.

Pro usnadnění vývoje dynamických formulářů se využívá aplikace popsaná v podsekci 6.1.10 a rozšíření aplikace Camunda Modeler popsané v podsekci 6.1.7. Nástroje umožňující otestovat vykreslování dynamického formuláře před nasazením procesu mají za cíl snížit chybovost při vývoji dynamických formulářů a zároveň usnadnit práci vývojáři nebo analytikovi, který daný proces implementuje. Nástroj Editor formulářů zároveň umožňuje otestovat nasazený proces.

6.2.3 Shrnutí

Součástí této podsekce byl popis procesů souvisejících s vývojem a správou systému dynamických formulářů.

6.3 Shrnutí kapitoly

Tato kapitola představuje návrhovou část pro celkovou architekturu rozšíření systému dynamických formulářů a detailní návrh jednotlivých komponent. V další kapitole 7 je uveden postup implementace jednotlivých komponent a proof-of-concept systému rozšíření pro správu uživatelských úkolů, dle návrhu popsaného v rámci této kapitoly. Návrh především adresuje problémy popsané v kapitole 5 o stávajícím stavu. Řešení je zároveň navrženo tak, aby ho bylo možné využít i mimo kontext aplikace Eprocesy.

Kapitola 7
Implementace

V rámci této kapitoly jsou popsány jednotlivé implementované komponenty, postup implementace a způsob použití jednotlivých komponent.

7.1 Rozšíření Camunda Modeler

Rozšíření Camunda Modeler pro usnadnění vývoje procesů využívajících dynamické formuláře, jehož návrh je popsán v rámci podsekce 6.1.7, vychází ze starter projektu[1] poskytovaného pro snazší implementaci komunitních rozšíření pro nástroj Camunda Modeler.

Rozšíření implementuje novou skupinu vstupů pro Start Event a User Task elementy v rámci BPMN. Součástí skupiny vstupů jsou dva textové vstupní prvky, prvním z nich je identifikátor formuláře a druhým je verze formuláře.

Pro validaci identifikátoru se používá následující regulární výraz:

```
^([a-z][a-z0-9]*)(-[a-z0-9]+)*$
```

Příklad validní hodnoty identifikátoru je následující:

```
form-identifier
```

Pro validaci verze se používá následující regulární výraz:

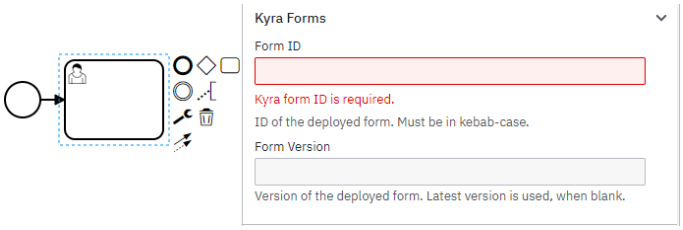
```
^([0-9]+)\.([0-9]+)\.([0-9]+)
(?:-([0-9A-Za-z-]+(?:\.[0-9A-Za-z-]+)*))
?(?:\+[0-9A-Za-z-]+)?$
```

Příklad validní hodnoty verze je následující:

```
0.0.1
```

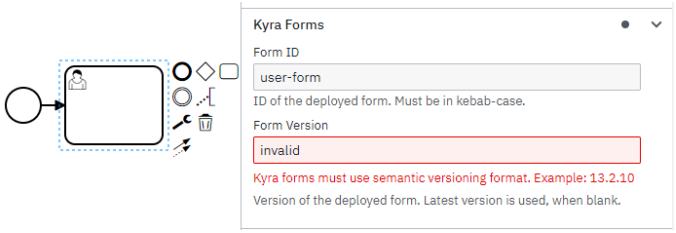
Na obrázku 7.1 je možné vidět, jakým způsobem se vykresluje rozšíření v rámci Camunda Modeler.

[1] Camunda Modeler Plugin Starter — `https://github.com/camunda/camunda-modeler-plugin-example`



Obrázek 7.1: Ukázka validní konfigurace formuláře v rámci Camunda Modeler rozšíření, zdroj autor

Na obrázku 7.2 je ukázka validace vstupu v rámci rozšíření.



Obrázek 7.2: Ukázka validace v rámci Camunda Modeler rozšíření, zdroj autor

7.1.1 Použití

Pro použití rozšíření je potřeba mít nainstalovaný nástroj npm (správce Javascript balíčků), následně je nutné získat zdrojový kód rozšíření a náhrát ho do adresáře s nainstalovaným Camunda Modeler, do podsložky:

```
modeler/resources/plugins
```

Pro instalaci potřebných závislostí a knihoven je potřeba ve složce s rozšířením spustit následující příkaz:

```
npm install
```

Pro sestavení rozšíření je následně nutné spustit příkaz:

```
npm run bundle
```

Pro aktivaci rozšíření je poté potřeba restartovat Camunda Modeler a zkontrolovat, že se rozšíření správně načetlo v levém horním menu, pod položkou Plugins.

7.1.2 Struktura rozšíření

Rozšíření pro Camunda Modeler se skládá z následujících částí:

- `index.js` — export Node.js modulu s rozšířením.
- `package.json` — obsahuje závislosti nutné pro sestavení rozšíření.

- `moddle/activity.json` — soubor určený pro knihovnu bpmn-moddle, který rozšiřuje metamodel BPMN o nové atributy s identifikátorem a verzí formuláře.
- `client.js` — obsahuje registraci jednotlivých rozšíření (BPMN modelu a panelu se vstupními prvky)
- `properties-panel/` — soubory obsahující funkce pro vykreslení nových vstupních prvků v rámci vstupního panelu.

7.1.3 Shrnutí

Součástí této sekce byl popis struktury a použití komponenty rozšíření Camunda Modeler aplikace umožňující snazší implementaci dynamických formulářů pro uživatelské úkoly v rámci procesů automatizovaných pomocí procesní platformy Camunda.

7.2 Editor formulářů

Tato sekce obsahuje popis implementace a použití aplikace Editor formulářů, jejíž návrh je blíže popsán v rámci podsekce 6.1.10.

Ukázka aplikace je ve formě snímků obrazovky součástí přílohy H.

Cílem implementace bylo vytvořit aplikaci nezávislou na kontextu aplikací Eprocesy a Hub pro obecné použití. V rámci implementace byly provedeny následující kroky:

- Vytvoření nové aplikace Editor formulářů.
- Import částí pro vykreslení dynamických formulářů z aplikace Hub, včetně grafických komponent.
- Napojení aplikace na nové rozhraní dynamických formulářů, které je součástí GraphQL rozšíření popsaného dále v rámci podsekce 7.3.2.
- Vytvoření nové sekce Tasklist a napojení na rozhraní aplikace Tasklist Backend popsané v rámci sekce 7.5.

Aplikace využívá existující část aplikace Hub Frontend pro vykreslení dynamických formulářů. Tato část zdrojových kódů je dílem Centra znalostního managementu ČVUT FEL a s jeho souhlasem byla uveřejněna a je součástí zdrojových kódů této práce, tak aby mohla být aplikace Editor formulářů spuštěna jako funkční celek. Zdrojové kódy, které nejsou součástí této práce, ale jsou dílem CZM jsou v rámci zdrojových kódů označeny v souboru `LICENCE.txt`.

7.2.1 Použití

Jsou dva základní způsoby jak aplikaci Editor Formulářů spustit. Prvním z nich je za použití nástroje Node Package Manager. Druhý způsob je pomocí technologie Docker.

NPM

Pro použití aplikace je nutné získat zdrojové kódy a následně provést instalaci potřebných závislostí a knihoven spuštěním následujícího příkazu ve složce s aplikací:

```
npm install
```

Do souboru `.env` je potřeba zanést konfiguraci aplikace, především pak do proměnné `REACT_APP_SERVER_URL` adresu GraphQL rozhraní aplikace Tasklist Backend.

Pro spuštění aplikace je poté potřeba použít příkaz:

```
npm start
```

Aplikace se poté spustí a je přístupná na adrese: `http://localhost:3000/`.

Docker

Aplikaci Editor formulářů je možné spustit za pomocí technologie Docker. V rámci praktické části vznikla konfigurace pro Docker a Gitlab CI/CD, tak aby bylo možné z každé nové verze aplikace vytvořit docker image a nahrát ho automaticky do Gitlab Container Registry.

Ze systému Gitlab je poté možné daný image získat pomocí jednoduchého příkazu. Konfigurace pro Gitlab CI/CD je součástí přílohy se zdrojovými kódy v souboru `.gitlab-ci.yml`. Konfigurace pro Docker je součástí souboru `Dockerfile`. Ve složce je také soubor `docker-compose.yml`, který slouží pro snadné spuštění aplikace pomocí nástroje docker-compose. Detailní popis spuštění, včetně konkrétních příkladů je součástí souboru `README.md` v příloze se zdrojovými kódy, ve složce s aplikací Editor Formulářů.

7.2.2 Shrnutí

V rámci této sekce byl popsán postup implementace aplikace Editor formulářů a způsob jeho použití při lokálním vývoji dynamických formulářů.

7.3 Java Knihovny

Součástí této sekce je popis knihoven implementovaných pro použití v rámci procesních aplikací v platformě Camunda. Všechny knihovny jsou napsány pomocí jazyka Java a využívají build systém Maven. Knihovny jsou zároveň nahrávány do package registry, které je součástí systému Gitlab. Ukázku použití těchto knihoven v rámci procesní aplikace je možné vidět v rámci sekce 7.4.

Postup implementace byl následující:

- Vytvořit modulární procesní aplikaci pomocí build systému Maven a knihovny Spring Boot.

- Implementovat tři moduly s minimem závislostí, kde každý modul odpovídal jednomu z rozšíření.
- Provést testování v rámci jedné procesní aplikace.
- Rozdělení jednotlivých modulů do vlastních repozitářů a postupný import do jedné aplikace z Gitlab Package Registry a iterativní testování.

7.3.1 Java rozšíření

Návrh rozšíření umožňujícího pracovat v rámci procesní aplikace s dynamickými formuláři pomocí Java API je možné nalézt v rámci podsekce 6.1.5.

Použití

Pro použití rozšíření v rámci procesní aplikace využívající build systém Maven a framework Spring Boot je možné využít Maven dependency z Gitlab Package Registry přidáním následující závislosti do `pom.xml` souboru:

```
<dependency>
  <groupId>org.kyra.platform</groupId>
  <artifactId>forms-java</artifactId>
  <version>${forms-java.version}</version>
</dependency>
```

Aby bylo možné využít dynamických formulářů v rámci definice procesů je potřeba, aby BPMN obsahovaly následující XML namespace:

```
xmlns:kyra="kyra-platform.io"
```

Namespace se konfiguruje jako atribut prvku `bpmn:definitions`, tedy například následovně:

```
<bpmn:definitions ... xmlns:kyra="kyra-platform.io
```

Rozšíření pro Camunda Modeler popsané v rámci 7.1 přídává tento namespace automaticky.

Konfigurace

Rozšíření umožňuje identifikovat a verzovat dynamické formuláře pomocí deployment descriptor JSON souboru, který zahrnuje následující položky:

- identifikátor nasazení
- pole objektů obsahujících konfiguraci pro jednotlivé formuláře
  - identifikátor formuláře, používaný v Camunda Modeler rozšíření popsaného v rámci sekce 7.1 pro identifikaci formuláře

- verze formuláře k jednoznačné identifikaci při nasazení procesu
- zdrojový soubor s formulářem
- přípona souboru s formulářem

Příklad deployment descriptor:

```
{
  "deploymentId": "abcd",
  "forms": [
    {
      "formId": "first",
      "formVersion": "1.0.0",
      "resource": "first.json",
      "resourceExtension": ".json"
    },
    {
      "formId": "second",
      "formVersion": "1.0.0",
      "resource": "second.json",
      "resourceExtension": ".json"
    }
  ]
}
```

Výchozí umístění souboru je `/resources/deployment-descriptor.json` v resources složce Spring Boot procesní aplikace. Výchozí umístění je možné ovlivnit pomocí následujícího nastavení Spring Boot aplikace:

```
kyra:
  forms:
    descriptor: "deployment-descriptor.json"
```

7.3.2 GraphQL rozšíření

Knihovna umožňující pracovat s dynamickými formuláři pomocí GraphQL rozhraní závisí na rozšíření uvedeném v předchozí podsekci 7.3.1. Návrh tohoto rozšíření je možné nalézt v rámci návrhové části v podsekci 6.1.6.

Použití

Pro přidání rozšíření do procesní aplikace využívající build systém Maven a framework Spring Boot je potřeba importovat Maven dependency z Gitlab Package Registry přidáním následující závislosti do `pom.xml` souboru:

```
<dependency>
  <groupId>org.kyra.platform</groupId>
  <artifactId>forms-graphql</artifactId>
  <version>${forms-graphql.version}</version>
</dependency>
```

Graphql rozšíření importuje knihovnu Spring GraphQL pro vytváření GraphQL API. Pro otestování, zda-li importovaná knihovna správně funguje, je možné pomocí následujícího nastavení Spring Boot aplikace povolit zobrazení grafického rozhraní GraphiQL, které umožňuje procházet nasazené GraphQL schéma a vykonávat jednotlivé dotazy:

```
spring:
  graphql:
    graphiql:
      enabled: true
      path: "/graphiql"
```

Následně je možné na adrese běžící aplikace s cestou `/graphiql` spustit například následující GraphQL dotaz pro získání definice formuláře, včetně hodnot procesních proměnných:

```
query {
    getFormInstance(activityId: "task-id") {
        activityId,
        serializedForm,
        variables {
            value
            varName
            type
        }
    }
}
```

7.3.3 Kafka rozšíření

Rozšíření umožňuje odesílat události do systému Apache Kafka a jeho návrh je součástí podsekce 6.1.8.

Použití

Pro přidání rozšíření do procesní aplikace využívající build systém Maven a framework Spring Boot je potřeba přidat následující Maven dependency z Gitlab Package Registry do `pom.xml` souboru:

```
<dependency>
  <groupId>org.kyra.platform</groupId>
  <artifactId>task-extension</artifactId>
  <version>{task-extension.version}</version>
</dependency>
```

Pro otestování Kafka rozšíření je možné lokálně využít postupy popsané v rámci sekce 7.6, která obsahuje způsoby, jakými je možné použít lokálně systém Apache Kafka.

Konfigurace

Rozšíření umožňuje provést konfiguraci Kafka topics do kterých se mají odesílat události nad uživatelskými úkoly. Konfiguraci je možné nastavit společnou pro všechny typy událostí, nebo rozdělit podle jednotlivých typů (například jen události, jako jsou vytvoření nového úkolu nebo přiřazení řešitele). Příklad takové konfigurace, v rámci procesní Spring Boot aplikace využívající rozšíření pro odesílání událostí do Apache Kafka, je následující:

```
camunda:
  messaging:
    kafka:
      bootstrap-servers: "localhost:9092"
      topic:
        task-event:
          all:
            name: "camunda.event.task"
            replicas: 1
            partitions: 1
```

V tomto případě dojde k připojení na Apache Kafka server běžící na adrese `localhost:9092` a k vytvoření topicu `camunda.event.task` kam budou odeslány události všech typů. Kafka Topic zároveň využívá jednu repliku a jeden partition. Systém Apache Kafka je detailnější popsán v teoretické části této práci v sekci 4.1 o Messaging systémech.

7.3.4 Shrnutí

Součástí této sekce je popis Java rozšíření, které umožňují pracovat s dynamickými formuláři a uživatelskými úkoly pomocí různých druhů rozhraní. Pro každé rozšíření byl uveden způsob použití a jeho konfigurace. Zároveň zde byl uveden postup implementace jednotlivých rozšíření.

7.4 Ukázka procesní aplikace

Součástí praktické části této práce bylo také vytvořit aplikaci, která demonstruje použití Java knihoven uvedených v rámci sekce 7.3.

Procesní aplikace využívá build systém Maven a framework Spring Boot, zároveň obsahuje embedded procesní engine Camunda a nakonfigurovaná Java rozšíření pro práci s dynamickými formuláři a odesílání událostí nad uživatelskými úkoly do systému Apache Kafka.

Pro spuštění aplikace je potřeba mít připravené prostředí, především nainstalovaný Java Development Kit (JDK) ve verzi 17. Následně je možné aplikaci spustit pomocí embedded Tomcat, který poskytuje knihovna Spring Boot. Ve výchozím nastavení aplikace běží na portu 8080, zároveň jsou ve výchozím nastavení povoleny Camunda Web aplikace (Tasklist, Cockpit a

Admin). Po přístupu na adresu `/camunda/app/tasklist/` je možné ověřit, že webové aplikace fungují.

V rámci podsekce 7.3.2 je popsáno jakým způsobem ověřit, že funguje GraphQL rozšíření. Pro ověření, že funguje Apache Kafka rozšíření je potřeba nejprve připravit infrastrukturu, tento postup je popsán v sekci 7.6.

Ukázková procesní aplikace obsahuje jednoduchou implementaci procesu s konfigurací dynamických formulářů, aby bylo možné dále v rámci 7.2 ověřit, že dojde k nasazení formulářů, správnému odeslání úkolu pomocí Apache Kafka, uložení v rámci Tasklist Backend aplikace a následnému vykreslení v rámci aplikace Editor formulářů.

7.5 Tasklist Backend aplikace

Aplikace Tasklist Backend slouží pro agregaci úkolů z více procesních aplikací využívající procesní engine Camunda. Detailní návrh backend aplikace Tasklist je popsán v rámci podsekce 6.1.9.

Pro spuštění aplikace je potřeba mít připravené prostředí, především nainstalovaný Java Development Kit (JDK) ve verzi 17, případně pak technologii Docker. Aplikace Tasklist pro správné fungování potřebuje mít funkční infrastrukturu a to především systém Apache Kafka a systém Elasticsearch, způsob jakým je možné je v lokálním prostředí spustit je popsán v následující sekci 7.6.

7.5.1 Konfigurace

Součástí konfigurace aplikace Tasklist jsou následující položky:

- nastavení připojení k Elasticsearch
- nastavení připojení k Apache Kafka
- pole objektů obsahujících nastavení klientů pro jednotlivé procesní aplikace
  - appId — identifikátor aplikace, odpovídá identifikátoru Camunda tenantů.
  - url — adresa procesní aplikace.
  - authentication — objekt pro HTTP Basic autentizaci (uživatelské jméno a heslo).

Ukázka konfigurace pro Tasklist aplikaci je následující:

```
kyra:
  tasklist:
    clients:
      - appId: "example"
        url: "http://localhost:8002/graphql"
```

```
        authentication:
          username: "user"
          password: "password"
    elastic:
      connectedTo: "localhost:9200"
      connectTimeout: 5000
      socketTimeout: 5000
      authentication:
        username: "elastic"
        password: "changeme"
camunda:
  messaging:
    kafka:
      bootstrap-servers: "localhost:9092"
```

7.5.2 Použití

Aplikaci Tasklist Backend je možné spustit jako standardní Spring Boot aplikaci pomocí JDK 17 a embedded aplikačního serveru Tomcat. Dalším způsobem jak aplikaci spustit je za pomocí technologie Docker. V rámci praktické části byla připravena konfigurace pro Docker a Gitlab CI/CD, tak aby bylo možné z každé nové verze aplikace vytvořit docker image a nahrát ho automaticky do Gitlab Container Registry. Ze systému Gitlab je poté možné daný image získat pomocí jednoduchého příkazu. Konfigurace pro Gitlab CI/CD je součástí přílohy se zdrojovými kódy v souboru `.gitlab-ci.yml`. Konfigurace pro Docker je součástí souboru `Dockerfile`.

Spuštění aplikace pomocí technologie Docker sestává z následujících kroků:

1. Autentizace vůči Gitlab Container Registry pomocí příkazu docker login.
2. Spuštění aplikace pomocí příkazu docker run.

Přesné příklady příkazů jsou součástí souboru README.md v příloze se zdrojovými kódy aplikace Tasklist Backend.

7.6 Infrastruktura

V rámci infrastruktury potřebné pro správné fungování všech komponent v rámci praktické části této práce je potřeba zprovoznit následující služby s odpovídající verzí:

- Elasticsearch 7.17.3
- Apache Kafka 2.3.0

Pro spuštění infrastruktury v rámci lokálního vývojového prostředí je součástí přílohy se zdrojovými kódy i soubor `docker-compose.yml`, který popisuje a konfiguruje celou infrastrukturu. Pomocí nástroje docker-compose

a technologie Docker je možné celou infrastrukturu spustit pomocí následující příkazu ve složce se souborem `docker-compose.yml`:

```
docker-compose up
```

Služby jsou následně vystaveny na následujících portech a připravené pro použití:

- Elasticsearch — 9200 (uživatelské jméno: `elastic`, heslo: `changeme`)
- Kibana — 5601
- Apache Kafka Broker — 9092
- Grafické rozhraní pro práci s Apache Kafka — 3030

Pro spuštění Apache Kafka se používá připravený docker image od LensesIO[2] a je možné i samostatně spustit pouze Apache Kafka pomocí následujícího příkazu:

```
docker run --rm -e ADV_HOST=localhost
-p 3030:3030 -p 9092:9092 -p 8081:8081
-p 8083:8083 -p 8082:8082 -p 2181:2181
landoop/fast-data-dev
```

7.7 Shrnutí kapitoly

V rámci této kapitoly je popsán postup při implementaci jednotlivých komponent navržených v rámci sekce 6.1. Zároveň jsou zde uvedeny i postupy pro spuštění a konfiguraci jednotlivých komponent a způsob jakým ověřit, že dané komponenty v lokálním vývojovém prostředí fungují správně.

[2]Připravený docker image pro práci s Apache Kafka, Schema registry a ZooKeeper: `https://github.com/lensesio/fast-data-dev`

Kapitola 8
Další využití

Tato kapitola popisuje, jakým způsobem je možné využít navržené rozšíření správy uživatelských úkolů a implementované komponenty v rámci dalších systémů mimo aplikaci Eprocesy. Dále popisuje, jakým způsobem je možné na tuto práci navázat.

8.1 Využití v rámci dalších systémů

V případě návrhu jednotlivých komponent i architektury systému byl kladen důraz na jejich rozšiřitelnost a modularitu. Jednotlivé funkční celky, jako jsou například rozšíření popsaná v rámci sekce 7.3, lze využít nezávisle na sobě a importovat je do libovolných procesních aplikací využívajících procesní embedded engine Camunda. Díky konfiguračním možnostem jednotlivých rozšíření je možné je použít s již existující infrastrukturou v rámci daného prostředí. Je tedy možné využít již existující instanci Apache Kafka, kterou může organizace například spravovat centrálně a poskytovat ji jednotlivým aplikacím jako službu.

Frontend část pro správu uživatelských úkolů je možné využít jako nástroj pro editaci a vývoj formulářů přímo v lokálním prostředí, a tedy nezávisle na omezeních například testovacích prostředí v rámci dané organizace. Editor popsaný v sekci 7.2 je možné snadnou konfigurací využít, jak pro jednu instanci procesní aplikace, která jen musí importovat GraphQL rozšíření, tak i přímo s aplikací Tasklist, která umožňuje agregovat úkoly z více systémů. Možnost spojit více úkolů z více systémů je výhodou oproti existujícím řešením, která vždy pracují jen s jedním procesním enginem. Vzhledem ke stále se rozšiřujícímu trendu servisně orientovaných architektur je možnost agregovat úkoly z více systémů stále důležitější. Pro organizace je zároveň důležité mít možnost vykonávat nad agregovanými úkoly datovou analýzu.

Rozšíření tedy umožňuje snadno za využití kombinace backend rozšíření a aplikace Tasklist vytvořit vlastní grafické rozhraní. Toto rozhraní umožní uživatelům pracovat na úkolech agregovaných z více systémů. Systém dynamických formulářů přesouvá náročnost na vytváření grafických formulářů z frontend vývojářů přímo na analytiky nebo vývojáře implementující dané procesy, kteří jsou nyní méně závislí na zdrojích frontend týmu a mohou sami do určité míry rozhodnout o podobě formulářů. Rozšíření zároveň analyti-

kům a vývojářům poskytuje nástroje pro usnadnění této práce při vývoji dynamických formulářů. Proces vývoje formulářů je pak zadokumentován v rámci sekce 6.2. Při integraci vykreslování dynamických formulářů do nově vznikajících, případně existujících frontend aplikací lze využít ukázky použití v rámci aplikace Editor formulářů. Aplikace je vytvořena modulárně, aby případná integrace do jiné React aplikace byla snadná.

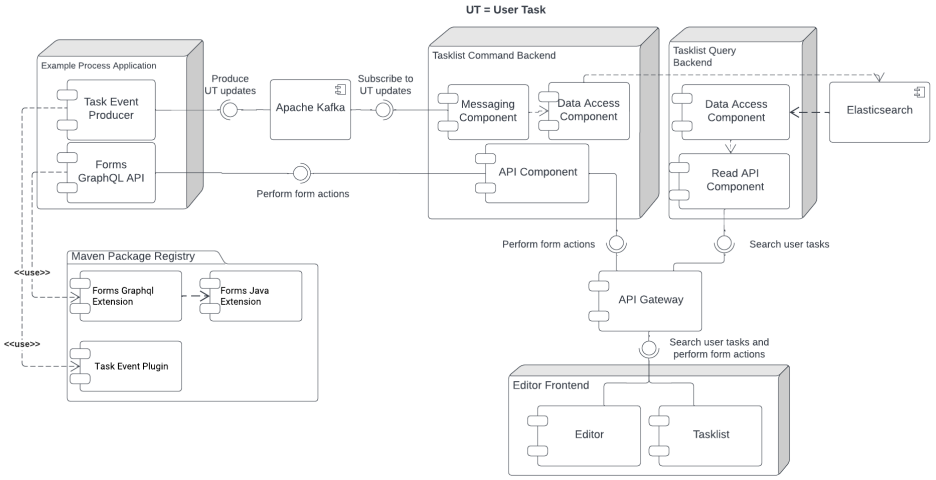
Návrh rozšíření zároveň uvažuje i změnové požadavky, zvláště pak na systém dynamických formulářů, kdy různé systémy budou mít různé potřeby pro frontend komponenty v rámci grafického rozhraní. Z důvodu změnových požadavků byl v rámci sekce 6.2 popsán proces pro řízení požadavků na systém dynamických formulářů a backend rozhraní dynamických formulářů bylo upraveno takovým způsobem, aby v případě změnových požadavků docházelo k zásahům jen na straně frontend a omezily se tak potřebné zásahy do backend části systému a snížily se potřebné kapacity vývojového týmu.

8.2 Rozšiřitelnost

V této sekci jsou popsány způsoby, jakými by bylo možné navázat na tuto práci.

8.2.1 Tasklist

Aplikace Tasklist, vyvinutá v rámci v praktické části, slouží jako proof-of-concept. Jedním ze způsobů, jak navázat na tuto práci, by bylo vytvoření plnohodnotné Tasklist aplikace, včetně samostatné frontend aplikace. Součástí by mohla také být změna architektury. V závislosti na míře, v níž čtecích operace převažují nad modifikujícími operacemi, by z pohledu škálovatelnosti bylo dobré zvážit použití návrhového vzoru Command Query Responsibility Segregation (CQRS), který by rozdělil Tasklist aplikaci na dva celky, které je možné škálovat samostatně. První celek by sloužil pro čtecí operace a druhý celek pro modifikující operace nad úkoly a formuláři a pro zápis změn do datového zdroje. Rozdělení těchto dvou celků by bylo pro frontend možné skrýt přidáním nové komponenty — API Gateway. Možné způsoby implementace pro GraphQL aplikace jsou popsány v rámci podsekce 4.2.6. Návrh tohoto rozšíření a změny architektury je možné vidět na obrázku 8.1. Vzhledem k modulárnímu návrhu PoC aplikace by ji bylo možné rozdělit na dva celky bez nákladných zásahů.



Obrázek 8.1: Návrh možné architektury, zdroj autor

Tento návrh by do architektury systému vnesl dodatečnou komplexitu rozdělením služeb a implementací API gateway, zároveň by však umožnil nezávisle škálovat služby pro čtecí a modifikující (create, update, delete) operace.

8.2.2 Služba agregující procesy

Dalším způsobem, jak navázat na tuto práci, by bylo vytvoření nové služby, která by agregovala procesy z jednotlivých systémů, stejně jako Tasklist agreguje úkoly. Vzhledem k tomu, že ve většině organizací se procesy mohou skládat z více souvisejících procesů, případně podprocesů, bylo by možné pomocí podobné architektury vytvořit službu, která by z messaging systému přijímala aktualizace procesních instancí a umožňovala nad nimi provádět datovou analýzu. Služba by zároveň mohla umožnit zapouzdření souvisejících procesů do celků, aby v rámci organizace bylo možné monitorovat a auditovat end-to-end procesy, které se skládají z více podprocesů nebo souvisejících procesů.

8.3 Shrnutí

Součástí této kapitoly je popis použití navrženého systému v rámci dalších aplikací a systémů. Zároveň je součástí návrh dalšího rozšíření tohoto systému pro správu uživatelských úkolů.

Kapitola 9

Závěr

V rámci teoretické části této práce byly představeny hlavní principy procesního řízení, včetně modelování pomocí BPMN notace a životního cyklu BPM. Hlavní důraz byl kladen na automatizaci procesů a správu uživatelských úkolů. Dále byla v rámci teoretické části práce představena platforma Camunda, její komponenty a způsoby, jakými je možné platformu využít v jednotlivých fázích BPM cyklu. Vzhledem k tématu praktické části byl kladen hlavní důraz na práci s uživatelskými úkoly. V poslední části byly detailněji představeny další principy a technologie použité v praktické části, konkrétně Enterprise Messaging, GraphQL a Elasticsearch.

Praktická část zahrnovala analýzu současného stavu správy uživatelských úkolů v rámci systému Eprocesy v prostředí ČVUT FEL. Analýza sestávala z popisu jednotlivých komponent a principů, zároveň identifikovala hlavní problémy s aktuálním návrhem. Mezi hlavní problémy se řadila použitelnost pro vývojáře, náročná rozšiřitelnost o další komponenty a chybějící způsob integrace do dalších systémů. Na problémy popsané v analýze stávajícího stavu se zaměřovala další část, v rámci které byl představen návrh rozšíření správy uživatelských úkolů pro obecnější použití i mimo prostředí aplikace Eprocesy. Pro návrh byly vymezeny předpoklady, omezení a funkční i nefunkční požadavky. Návrh sestával z celkové architektury systému, detailního popisu jednotlivých komponent, definice jednotlivých rozhraní a specifikace použitých technologií. Závěrem návrhu byly popsány procesy pro řízení nových požadavků na systém dynamických formulářů a postup pro implementaci dynamických formulářů v rámci procesů využívajících platformu Camunda. Pro popis procesů byla zvolena notace BPMN, popsaná v teoretické části.

Součástí praktické části práce byla také implementace jednotlivých navržených komponent a proof-of-concept systému využívajícího rozšíření systému pro správu uživatelských úkolů. V rámci implementační fáze také byly aplikovány principy pro Continuous Integration a Delivery, tak aby se jednotlivé projekty průběžně sestavovali a spouštěli se implementované testy v rámci Gitlab CI/CD. Pro každou z komponent vznikla i dokumentace možných způsobů využití. Závěrem praktické části bylo popsáno, jak je rozšíření možné využít v rámci dalších systémů mimo prostředí aplikace Eprocesy, a zároveň zde bylo navrženo, jakým způsobem by se na práci dalo navázat.

Přílohy

Příloha A

Literatura a zdroje

1. SCARSIG; BENEDICT, Tony; KIRCHMER, Mathias; FRANTZ, Pater; SAXENA, Raju; MORRIS, Dan; HILTY, Jack. *BPM CBOK Version 4.0: Guide to the Business Process Management Common Body Of Knowledge*. 4.0. vyd. Independently published, 2019. ISBN 978-1704809342.
2. DUMAS, M.; ROSA, M.L.; MENDLING, J.; REIJERS, H.A. *Fundamentals of Business Process Management*. Berlin: Springer Berlin Heidelberg, 2013. ISBN 9783642331435. Dostupné také z: `https://books.google.cz/books?id=USVEAAAAQBAJ`.
3. GEORGAKOPOULOS, Diimitrios; HORNICK, Mark; SHETH, Amit. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*. 1995, roč. 1995, č. 3, s. 119–153. ISSN 1573-7578. Dostupné z DOI: `https://doi.org/10.1007/BF01277643`.
4. RUECKER, Bernd. *Practical process automation: orchestration and integration in microservices and cloud native architectures*. Beijing: O'Reilly, 2021. ISBN 978-1492061458.
5. BROCKE, Jan vom; MENDLING, Jan; ROSEMANN, Michael. *Business Process Management Cases Vol. 2: Digital Transformation - Strategy, Processes and Execution*. Vol. 2. Berlin: Springer Berlin, Heidelberg, 2021. ISBN 978-3-662-63049-5.
6. *Camunda Platform: DMN Tutorial* [online] [cit. 2022-09-03]. Dostupné z: `https://camunda.com/dmn/`.
7. BROCKE, Jan vom; ROSEMANN, Michael. *Handbook on Business Process Management 1: Introduction, Methods, and Information Systems*. 2. vyd. Berlin: Springer, Berlin, Heidelberg, 2014. ISBN 978-3-642-45100-3.
8. FREUND, Jakob; RÜCKER, Bernd. *Real-Life BPMN (4th edition): Includes an introduction to DMN*. 4th edition. Camunda, 2019. ISBN 978-1086302097.

9. GRIGORI, Daniela; CASATI, Fabio; DAYAL, Umeshwar; SAYAL, Mehmet; SHAN, Ming-Chien. Business Process Intelligence. *Computers in Industry*. 2004, roč. 53, č. 3, s. 321–343. ISSN 0166-3615. Dostupné z DOI: https://doi.org/10.1016/j.compind.2003.10.007.
10. *Microservices.io/* [online]. 2022 [cit. 2022-09-04]. Dostupné z: https://microservices.io/.
11. *Docs.camunda.io* [online]. Camunda, 2022 [cit. 2022-09-04]. Dostupné z: https://docs.camunda.io/.
12. RUECKER, Bernd. *The Microservices Workflow Automation Cheat Sheet – to Centralize or Decentralize?* [online] [cit. 2022-09-04]. Dostupné z: https://camunda.com/blog/2020/03/the-microservices-workflow-automation-cheat-sheet-to-centralize-or-decentralize/.
13. *Docs.jboss.org* [online] [cit. 2022-09-04]. Dostupné z: https://docs.jboss.org/jbpm/v6.0/userguide/jBPMTaskService.html.
14. HOHPE, Gregor; WOOLF, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 1st edition. Addison-Wesley Professional, 2003. ISBN 978-0321200686.
15. *Enterpriseintegrationpatterns.com* [online]. 2022 [cit. 2022-10-23]. Dostupné z: https://www.enterpriseintegrationpatterns.com/.
16. JACKSON, Callum; COPPEN, Richard J. *Understanding enterprise messaging APIs and protocols: What is a messaging protocol and do I really need one? I think the answer is yes, but why?* [online]. IBM, 2022 [cit. 2022-11-06]. Dostupné z: https://developer.ibm.com/articles/messaging-protocols/.
17. *Stomp.github.io: The Simple Text Oriented Messaging Protocol* [online]. 2022 [cit. 2022-11-06]. Dostupné z: https://stomp.github.io/.
18. *Amqp.org* [online]. 2022 [cit. 2022-11-06]. Dostupné z: https://www.amqp.org/.
19. *Mqtt.org: MQTT: The Standard for IoT Messaging* [online] [cit. 2022-11-06]. Dostupné z: https://mqtt.org/.
20. *Activemq.apache.org* [online]. 2022 [cit. 2022-11-06]. Dostupné z: https://activemq.apache.org/.
21. *Rabbitmq.com* [online]. 2022 [cit. 2022-11-06]. Dostupné z: rabbitmq.com.
22. *Confluent.io: What is Kafka?* [online]. Confluent, 2014-2022 [cit. 2022-11-06]. Dostupné z: https://www.confluent.io/what-is-apache-kafka/.
23. VIDELA, Alvaro; WILLIAMS, Jason J. W. *RabbitMQ in action: distributed messaging for everyone*. 1st edition. Shelter Island: Manning, [2012]. ISBN 978-193-5182-979.

24. NARKHEDE, Neha; SHAPIRA, Gwen; PALINO, Todd. *Kafka: the definitive guide : real-time data and stream processing at scale*. 1st edition. Sebastopol: O'Reilly, 2017. ISBN 978-149-1936-160.
25. BUNA, Samer. *GraphQL in Action*. 1st edition. Manning, 2021. ISBN 978-1617295683.
26. *Graphql.org* [online] [cit. 2022-10-15]. Dostupné z: `https://graphql.org/`.
27. GIROUX, Marc-André. *Production Ready GraphQL: Building well designed, performant, and secure GraphQL APIs at scale*. 2020.
28. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. CALIFORNIA, 2000. Disertace. UNIVERSITY OF CALIFORNIA, IRVINE.
29. *Swagger.io* [online] [cit. 2022-10-15]. Dostupné z: `https://swagger.io/specification/`.
30. *Grpc.io* [online] [cit. 2022-10-15]. Dostupné z: `https://grpc.io/`.
31. GORMLEY, Clinton; TONG, Zachary. *Elasticsearch: the definitive guide*. 1st edition. Sebastopol: O'Reilly, 2015. ISBN 978-1-449-35854-9.
32. CHHAJED, Saurabh. *Learning ELK Stack: Build mesmerizing visualizations, and analytics from your logs and data using Elasticsearch, Logstash, and Kibana*. 1st edition. Birmingham: Packt Publishing Ltd, 2015. ISBN 978-1-78588-715-4.
33. *Elastic.co/* [online]. 2022 [cit. 2022-11-12]. Dostupné z: `https://www.elastic.co/`.
34. AHMED, Abd-Allah; GACEK, Cristina; CLARK, Brad; BOEHM, Barry. On the Definition of Software System Architecture. 1997, s. 1–11.
35. APPLETON, Brad. A Software Design Specification Template [online]. 1994-1997 [cit. 2022-10-08]. Dostupné z: `https://www.bradapp.net/docs/sdd.html`.

Příloha B

Seznam zkratek

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BPM	Business Process Management
BPMN	Business Process Management Notation
BPMS	Business Process Management System
CMMN	Case Management Model and Notation
CSS	Cascading Style Sheets
CZM	Centrum znalostního managementu
DMN	Decision Model and Notation
DRD	Decision Requirements Diagram
DRG	Decision Requirements Graph
FEEL	Friendly Enough Expression Language
FEL	Fakulta Elektrotechnická
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IT	Information Technology
JMS	Java Message Service
MOM	Message-Oriented Middleware
OMG	Object Management Group
PoC	Proof-of-concept

PoC	proof-of-concept
POJO	Plain old Java object
REST	Representational State Transfer
RMQ	RabbitMQ
RPA	Robotic Process Automation
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SPA	Single page application
SW	Software
TCP	Transmission Control Protocol
WYSIWYG	What you see is what you get
XML	Extensible Markup Language
ČVUT	České vysoké učení technické v Praze

Příloha C

Ukázka specifikace formuláře

```
  {
"formSections": [
  {
    "sectionId": "personalInformationSection",
    "label": "Osobní údaje",
    "formFields": [
      {
        "variable": "student",
        "formType": "STRING",
        "label": "Student (jméno a přijmení)"
      },
      {
        "variable": "formOfStudy",
        "formType": "STRING",
        "label": "Forma studia"
      },
      {
        "variable": "studyProgram",
        "formType": "STRING",
        "label": "Studijní program"
      },
      {
        "variable": "studyBranch",
        "formType": "STRING",
        "label": "Obor"
      },
      {
        "variable": "email",
        "formType": "STRING",
        "label": "Email"
      },
      {
        "variable": "grade",
        "formType": "NUMBER",
```

```
        "label": "Ročník"
      },
      {
        "variable": "startOfStudy",
        "formType": "DATE",
        "label": "Začátek studia"
      },
      {
        "variable": "personalNumber",
        "formType": "NUMBER",
        "label": "Osobní číslo"
      }
    ],
    "properties": [
      {"key": "readOnly", "value": "true"}
    ]
  },
  {
    "sectionId": "contactInformationSection",
    "label": "Kontaktní informace",
    "formFields": [
      {
        "variable": "address",
        "formType": "ADDRESS",
        "label": "Adresa"
      },
      {
        "variable": "phone",
        "formType": "STRING",
        "label": "Telefon"
      }
    ],
    "properties": [
      {"key": "readOnly", "value": "true"}
    ]
  },
  {
    "sectionId": "requestSection",
    "label": "Náležitosti žádosti",
    "formFields": [
      {
        "variable": "viceDeanDecision",
        "formType": "BOOLEAN",
        "label": "Rozhodnutí proděkana",
        "properties": [
          {"key": "readOnly","value": "true"},
```

```
            {"key": "trueLabel","value": "Žádost schválena"},
            {"key": "falseLabel","value": "Žádost zamítnuta"}
          ]
        },
        {
          "variable": "viceDeanComment",
          "formType": "STRING",
          "label": "Zdůvodnění rozhodnutí proděkana",
          "properties": [
            {"key": "textArea","value": "true"},
            {"key": "required","value": "false"}
          ]
        },
        {
          "variable": "officialJustification",
          "formType": "STRING",
          "label": "Oficiální zdůvodnění rozhodnutí",
          "properties": [
            {"key": "required","value": "true"},
            {"key": "textArea","value": "true"}
          ]
        },
        {
          "variable": "request",
          "formType": "FILE",
          "label": "Žádost",
          "properties": [
            {"key": "readOnly","value": "true"}
          ]
        },
        {
          "variable": "decision",
          "formType": "FILE",
          "label": "Rozhodnutí",
          "properties": [
            {"key": "readOnly","value": "true"}
          ]
        }
      ],
      "properties": [
        {"key": "inlineEdit","value": "true"}
      ]
    }
  ],
  "formActions": [
    {
```

```
      "variable": "endForm",
      "text": "Dokončit",
      "styleType": "PRIMARY",
      "actionType": "SUBMIT"
    },
    {
      "variable": "saveFormPrint",
      "text": "Uložit a odložit",
      "styleType": "SECONDARY",
      "actionType": "SAVE"
    }
  ],
  "description": {
    "paragraphs": [
      {
        "title": "Finální Kontrola",
        "text": "",
        "active": true,
        "icon": "fa-check-circle"
      },
      {
        "title": "Tisk dokumentu, zajištění podpisů a odeslání",
        "text": "",
        "active": true,
        "icon": "fa-print"
      },
      {
        "title": "Dokončení žádosti",
        "text": "Po dokončení kroku stiskněte <b>Dokončit</b>",
        "active": true,
        "icon": "fa-envelope"
      }
    ]
  }
}
```

Příloha D

Vlastnosti sekcí dynamických formulářů

- `readOnly`
  - typ: `Expression`
  - popis: Nastavení sekce jen pro čtení (text)
  - výchozí: `false`
- `disabled`
  - typ: `Expression`
  - popis: Nastavení sekce jako disabled
  - výchozí: `false`
- `visible`
  - typ: `Expression`
  - popis: Nastavení viditelnosti sekce
  - výchozí: `true`
- `inlineEdit`
  - typ: `Expression`
  - popis: Nastavení sekce jako inline-edit (forma zobrazení editovatelného prvku/sekce, neovlivňuje vlastnosti `readOnly`, `disabled` nebo `visible`), v případě typu `file` je ignorováno, zároveň **nemůže** být nastaveno na True jak u sekce, tak u konkrétního prvku, pokud je potomkem takové sekce.
  - výchozí: `false`

Příloha E

Vlastnosti prvků dynamických formulářů

obecné:

- `required`
  - typ: `Expression`
  - popis: Nastavení povinného prvku
  - výchozí: `false`
- `visible`
  - typ: `Expression`
  - popis: Nastavení viditelnosti prvku. Má prioritu před `readOnly`.
  - výchozí: `true`
- `readOnly`
  - typ: `Expression`
  - popis: Nastavení prvku jen pro čtení (text). Má prioritu před `disabled`.
  - výchozí: `false`
- `disabled`
  - typ: `Expression`
  - popis: Nastavení prvku jako disabled
  - výchozí: `false`
- `inlineEdit`
  - typ: `Expression`
  - popis: Nastavení sekce jako inline-edit (forma zobrazení editovatelného prvku/sekce, neovlivňuje vlastnosti `readOnly`, `disabled` nebo `visible`), v případě typu `file` je ignorováno, zároveň **nemůže** být nastaveno na True jak u sekce, tak u konkrétního prvku, pokud je potomkem takové sekce.

  - výchozí: `false`

- `helpText`
  - typ: `String`
  - popis: Pomocný text u prvku, formou “i” ikonky s tooltipem

- `doNotInheritFromSection`
  - typ: `Expression`
  - popis: Umožňuje ovlivnit dědičnost vlastností od sekce do které prvek spadá („nepřebírej vlastnosti rodičovské sekce’ ’)
  - výchozí: `false`

date:

- `fromDate`
  - typ: `String`
  - popis: Název proměnné nebo hodnota označující **začátek** intervalu (vždy inclusive tedy se včetně data)

- `toDate`
  - typ: `String`
  - popis: Název proměnné nebo hodnota označující **konec** intervalu (vždy exclusive tedy bez daného data)

text:

- `placeholder`
  - typ: `String`
  - popis: Text který se vykreslí jako placeholder v inputu (podle label dictionary)

- `autocomplete`
  - typ: `Boolean`
  - popis: Povoluje autocomplete u inputu
  - výchozí: `false`

- `textArea`
  - typ: `Boolean`
  - popis: Pokud je true, jedná se o rozsáhlejší textový vstup - vykreslí se jako text area

- výchozí: `false`

select:

- `values`
  - typ: `String` (JSON Object)
  - popis: Obsahuje JSON, více popsáno u typu Enum a Select.
- `radio`
  - typ: `Boolean`
  - popis: Pokud je true, jednotlivé možnosti enumu se vykreslí jako radio buttony - uživatel může vybrat právě jednu možnost
  - upozornění: Při kombinaci `multiselect=True` a `radio=True` se prvek vykreslí jako multiselect checkbox.
  - výchozí: `false`
- `multiselect`
  - typ: `Boolean`
  - popis: Umožňuje výběr více možností, ukládá hodnoty do pole
  - upozornění: Nekombinovat `multiselect=True` s propojením více form fieldů
  - výchozí: `false`

enum:

- `values`
  - typ: `String` (JSON Object)
  - popis: Obsahuje JSON, více popsáno u typu Enum a Select.
- `radio`
  - typ: `Boolean`
  - popis: Pokud je true, jednotlivé možnosti enumu se vykreslí jako radio buttony - uživatel může vybrat právě jednu možnost
  - výchozí: `false`
- `multiselect`
  - typ: `Boolean`
  - popis: Pokud je true, umožňuje výběr více možností, ukládá hodnoty do pole

- výchozí: `false`

boolean:

- `toggleSwitch`
  - typ: `Boolean`
  - popis: Pokud je nastaveno na true, tak se vykreslí jako toggle switch
  - výchozí: `false`
- `trueLabel`
  - typ: `String`
  - popis: Label pro true hodnotu (při `readOnly=true` se zobrazuje jen tento text)
- `falseLabel`
  - typ: `String`
  - popis: Label pro false hodnotu (při `readOnly=true` se zobrazuje jen tento text)

file:

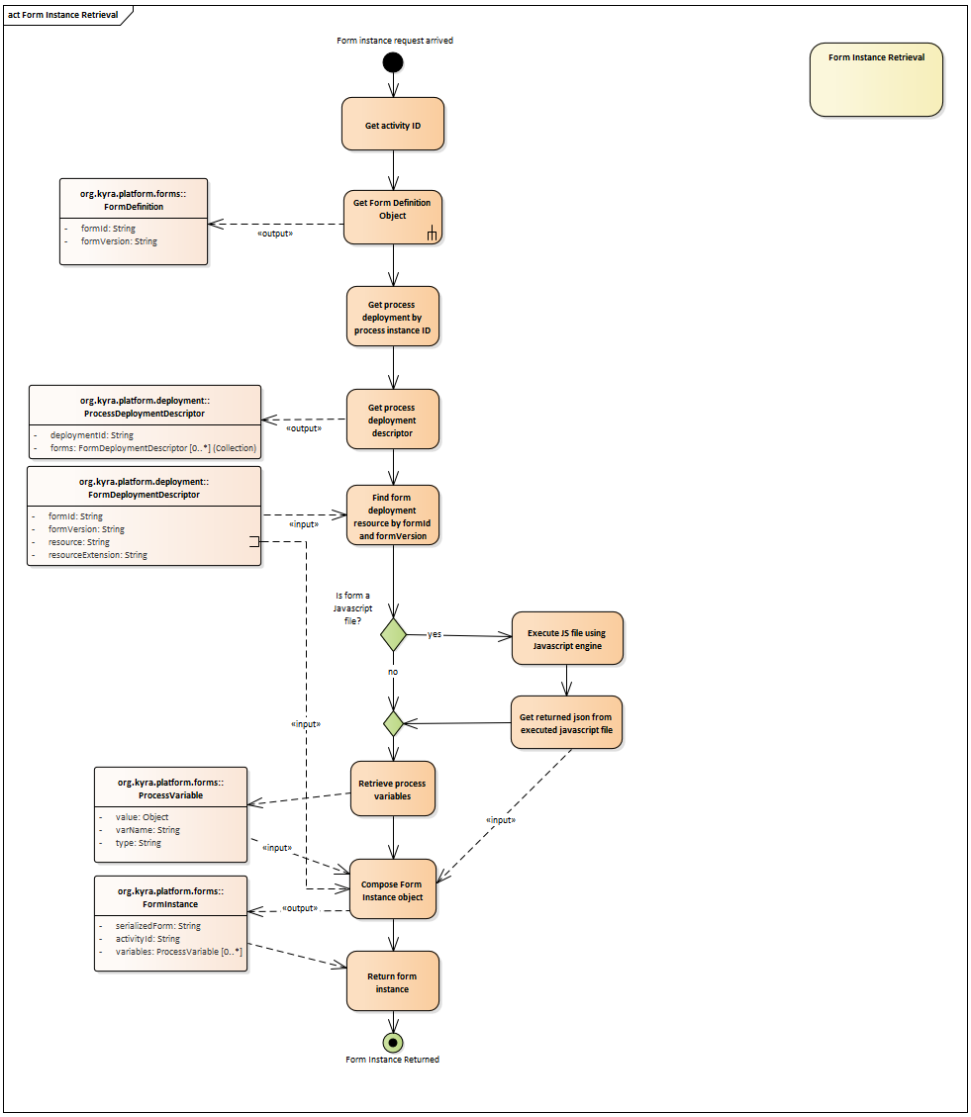
- `multiselect`
  - typ: `Boolean`
  - popis: Pokud je true, je možné nahrát více souborů, ty jsou odesílány jako pole objektů.
  - výchozí: `false`

number:

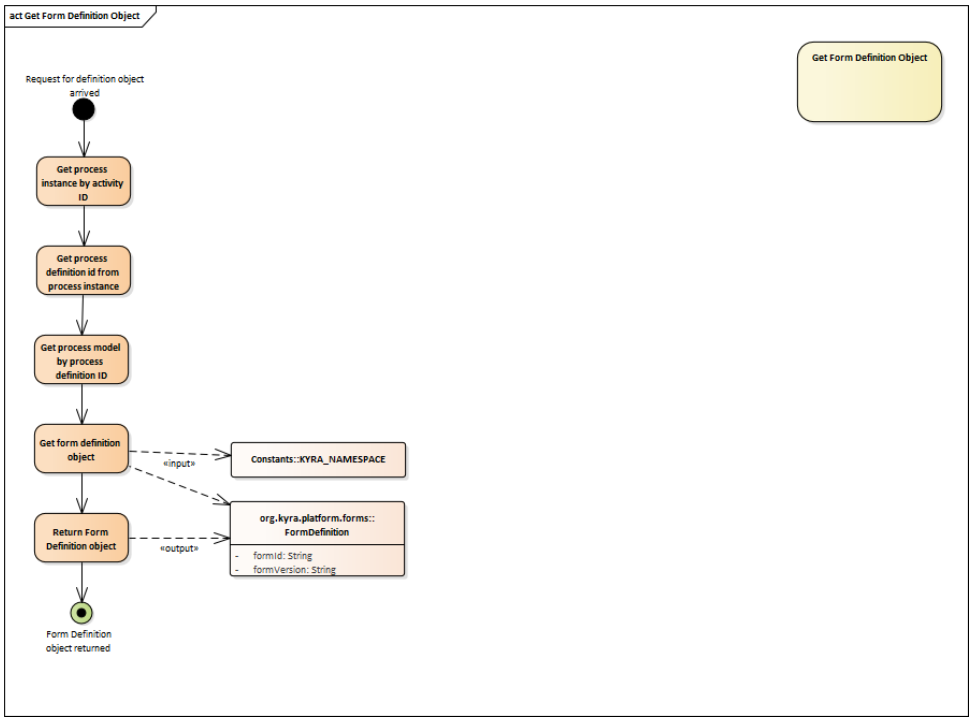
- `decimalPlaces`
  - typ: `Integer`
  - popis: Určuje množství desetinných pozic, které je možné k číslu zadat/které se vykreslí.
  - výchozí: `0`

Příloha F

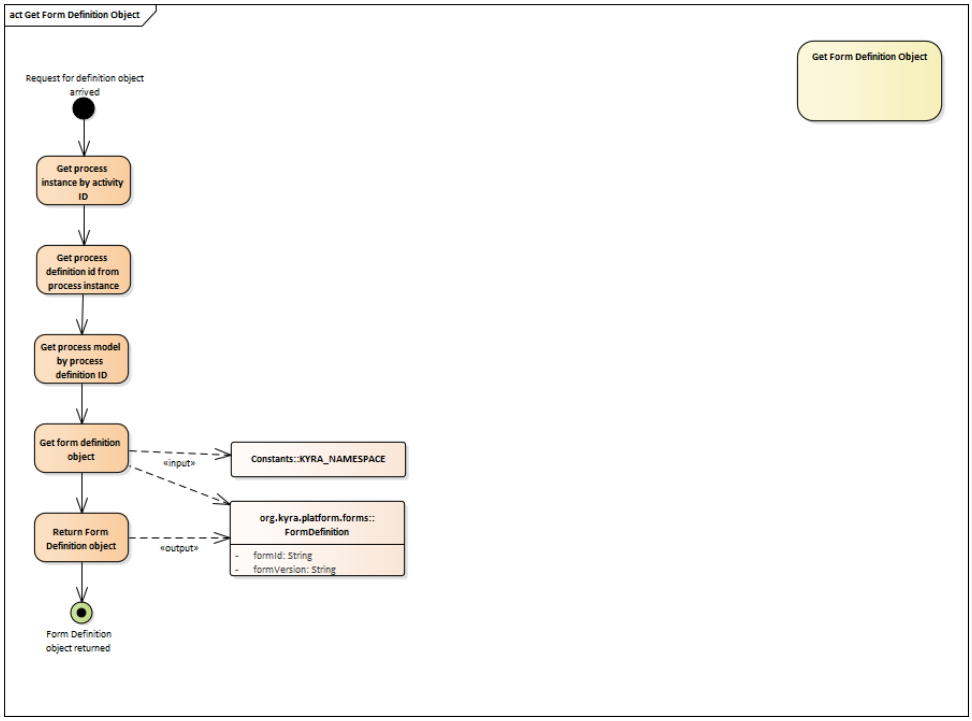
Diagramy aktivit



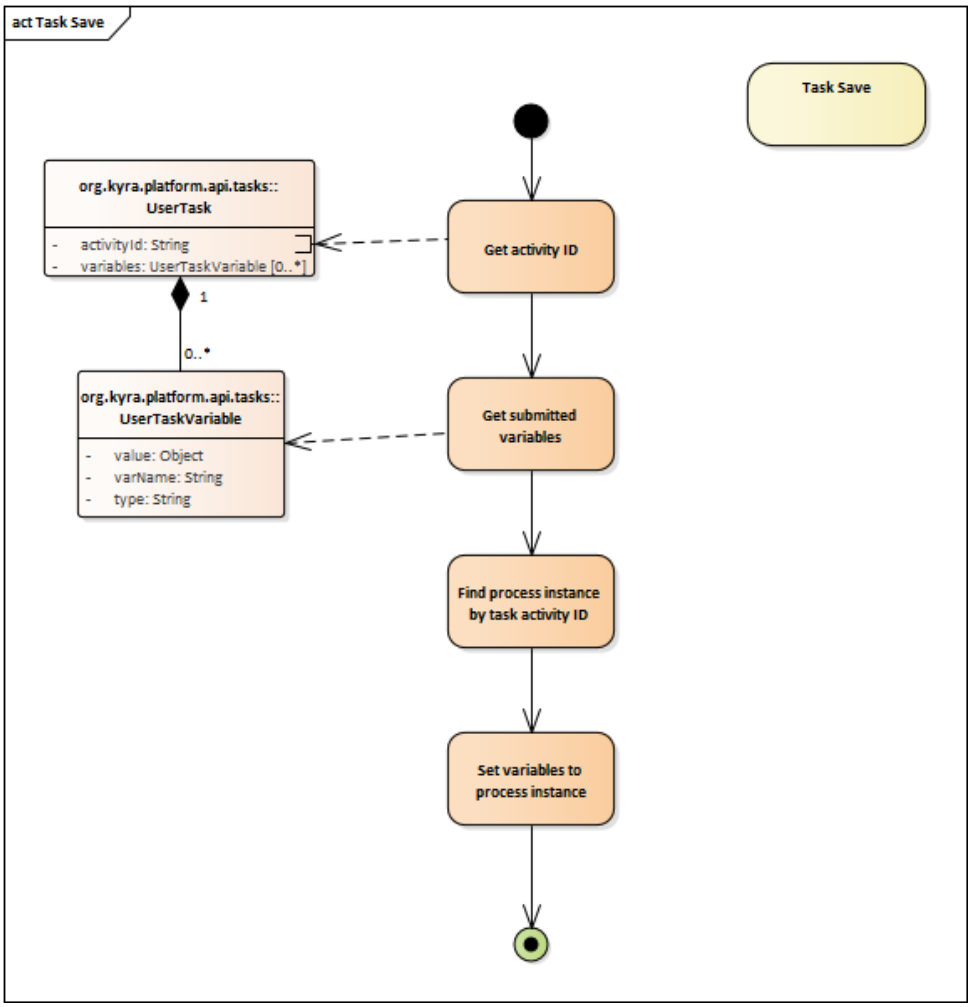
Obrázek F.1: Získání formuláře podle ID aktivity, zdroj autor



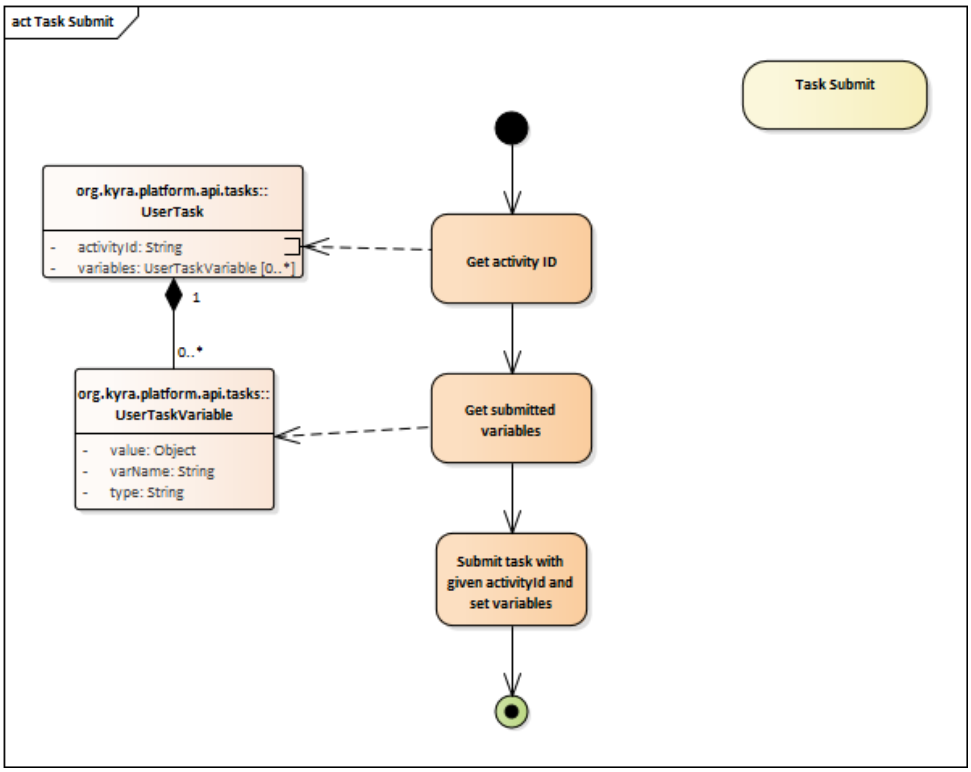
Obrázek F.2: Získání definice formuláře, zdroj autor



Obrázek F.3: Získání identifikátoru a verze formuláře z modelu, zdroj autor



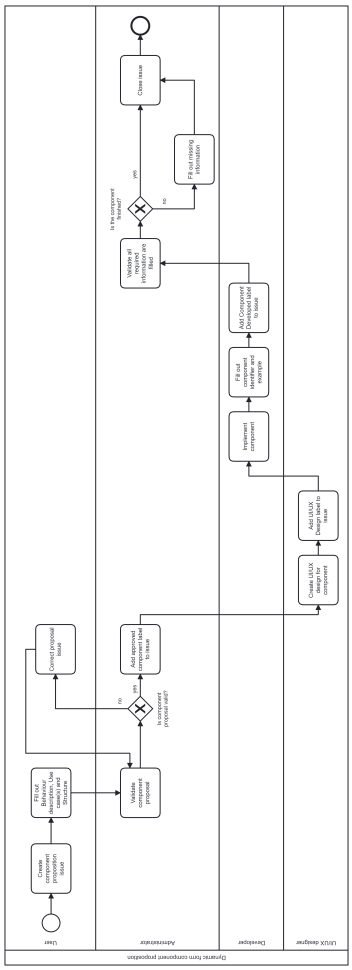
Obrázek F.4: Uložení úkolu, zdroj autor



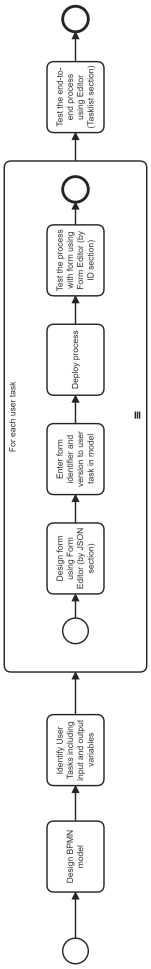
Obrázek F.5: Dokončení úkolu, zdroj autor

Příloha G

Související procesy



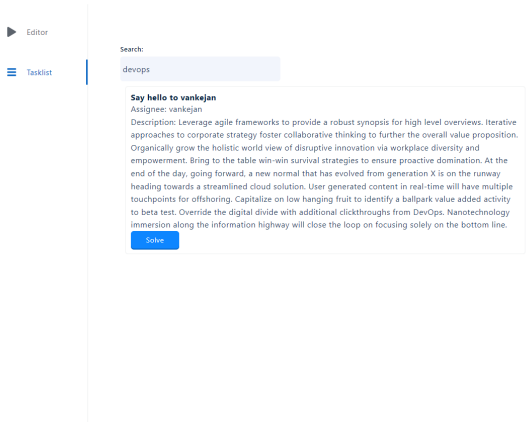
Obrázek G.1: Vytvoření nového požadavku na komponentu dynamických formulářů, zdroj autor



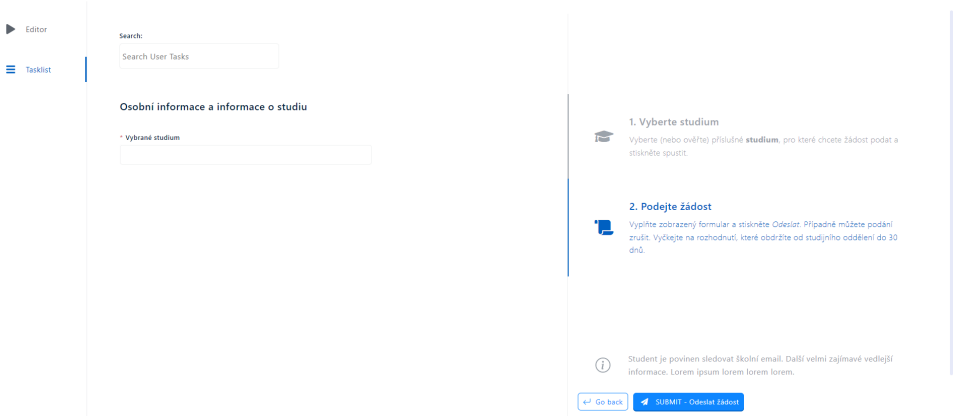
Obrázek G.2: Vývoj formulářů pro proces, zdroj autor

Příloha H

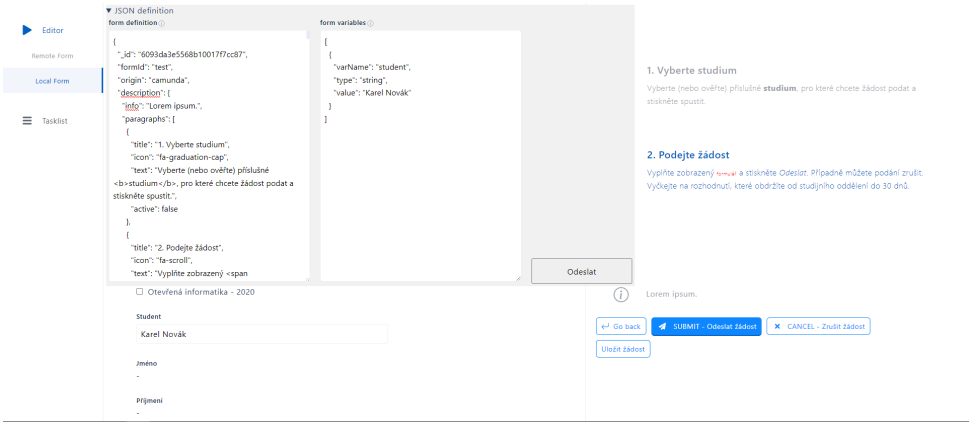
Snímky obrazovek



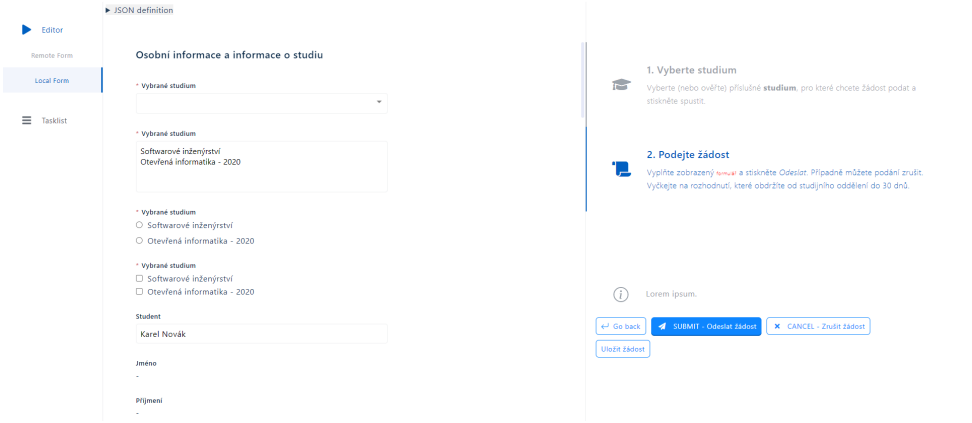
Obrázek H.1: Aplikace Tasklist s full-text vyhledáváním úkolů, zdroj autor



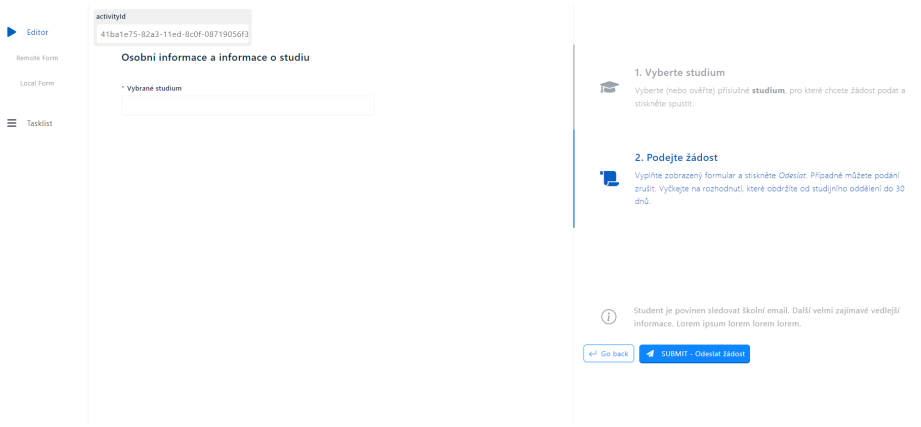
Obrázek H.2: Aplikace Tasklist po kliknutí na tlačítko Solve, zdroj autor



Obrázek H.3: Editor s formulářem pro zadání hodnot proměnných a definice formuláře, zdroj autor



Obrázek H.4: Editor po zadání JSON definice formuláře, zdroj autor



Obrázek H.5: Editor pro zadání ID úkolu a vykreslení formuláře pro běžící procesní instanci, zdroj autor

Příloha I

Zdrojové kódy

Následující struktura popisuje obsah přílohy se zdrojovými kódy:

```
/
├── example ... Ukázková procesní aplikace, která importuje
│               GraphQL a Kafka rozšíření.
├── graphql-extension ... Zdrojové kódy pro GraphQL rozšíření.
├── infrastructure ... Složka s konfigurací a návodem pro přípravu
│   │                  infrastruktury pro lokální použití.
│   └── README.md ... Soubor obsahující návod na
│                     spuštění infrastruktury.
├── java-extension ... Zdrojové kódy pro Java rozšíření, pro práci s
│                      formuláři.
├── kafka-extension ... Zdrojové kódy pro Kafka rozšíření.
├── modeler-plugin ... Zdrojové kódy pro rozšíření nástroje Camunda
│                      Modeler.
├── tasklist-backend ... Aplikace Tasklist Backend.
└── editor ... Zdrojové kódy frontend aplikace.
```