FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Systém pro správu chatbota
Student:	Jan Šafařík
Vedoucí:	Ing. David Kreuz
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Navrhněte a implementujte systém pro správu chatbota, který bude sloužit ke komunikaci se zaměstnanci společnosti. Ti díky chatbotovi budou moci pokládat otázky týkající se produktového portfolia.

Systém bude umět:
Vytvářet nové intenty a označovat specifické entity.
Sestavovat vzorové příběhy konverzace pro strojové učení.
Sledovat statistiky úspěšnosti chatbota při analýze otázek.
Procházet historické konverzace a zpětně označovat nerozpoznané otázky.
Komunikovat skrze REST API se službou RASA.

Postupujte v těchto krocích:
Analyzujte požadavky na systém, závislosti domén a komunikaci mezi službami.
Navrhněte backend systému.
Implementujte administrační systém na základě návrhu.
Sestavte testy pro backend.
Připravte deployment aplikace v technologii Docker.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 23. září 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Systém pro správu chatbota

Jan Šafařík

Katedra softwarového inženýrství
Vedoucí práce: Ing. David Kreuz

4. června 2020

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Davidu Kreuzovi za podporu a odbornou pomoc při psaní této práce. Rovněž děkuji společnosti ANECT a.s. za poskytnutou příležitost.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na základě níž se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 4. června 2020 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Jan Šafařík. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Šafařík, Jan. *Systém pro správu chatbota*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato bakalářská práce se věnuje vývoji administračního systému pro správu chatbotů implementovaných pomocí frameworku Rasa Open Source. Práce se zabývá analýzou požadavků, vyhotovením případů užití a na jejich základě pak navržením a implementací systému tak, aby byl testovatelný a do budoucna rozšiřitelný. Za tímto účelem je v práci provedena analýza stávajícího řešení, které spočívá ve vytvoření chatbota pomocí konfiguračních souborů a nástrojů systému Rasa X. Zároveň je zahrnuta analýza funkčních a nefunkčních požadavků společně s případy užití a vymezením uživatelských rolí. Dále je rozebrán návrh systému, který je rozčleněn z hlediska architektury do čtyř vrstev. V backendové části je popsána business vrstva, věnující se business entitám a jejich procesům, servisní, zaměřená na dotazovací jazyk GraphQL pro API rozhraní a datová, jejíž součástí je i struktura PostgreSQL databáze. Frontendová část pokrývá vrstvu prezentační a je v ní představen framework React Admin. Návrh je uzavřen procesem autentizace uživatele pomocí poskytovatele identity Azure Active Directory. Obě části jsou implementovány pomocí jazyka JavaScript, a to s využitím běhového prostředí Node.js v případě backendové části a webového frameworku React v případě části frontendové. V závěru práce jsou popsány jednotkové a integrační testy důležitých komponent backendu.

Klíčová slova webová aplikace, chatbot, Rasa, Node.js, React, GraphQL, Azure Active Directory

Abstract

This bachelor thesis describes the development of an administration system for the management of chatbots implemented using the Rasa Open Source framework. The thesis deals with requirement analysis, defining use cases along with the design and implementation built on their basis with attention to testability and scalability for future purposes. The first part analyzes the existing solution, which consists of creating a chatbot using configuration files and Rasa X system. It also includes analysis of system requirements along with use cases and defines user roles. The presented system design architecture is divided into four layers. The back-end part of the design describes the business layer, which is dedicated to business entities and their processes, the service layer, which focuses on the API interface using the GraphQL query language, and the data layer, which includes PostgreSQL database structure. The front-end covers the presentation layer and introduces the React Admin framework. The design chapter concludes of the user authentication process using Azure Active Directory identity provider. Both parts are implemented using JavaScript language with the Node.js runtime environment in the case of the back-end part and the React web framework in the case of the front-end. Finally, the last part of the thesis describes unit and integration tests of important back-end components.

Keywords web application, chatbot, Rasa, Node.js, React, GraphQL, Azure Active Directory

Obsah

Úvod **1**
Popis problému . . . 1
Startup WaldoBot . . . 1
Cíl práce . . . 2
Motivace . . . 2
Struktura práce . . . 2

1 Teoretická část **3**
1.1 Pojmy a definice . . . 3
1.2 Rasa Open Source . . . 4
1.3 Konfigurační soubory . . . 5
1.3.1 Domény chatbota . . . 6
1.3.2 Trénovací data vzorových zpráv . . . 7
1.3.3 Trénovací data vzorových příběhů . . . 7
1.3.4 Trénování a evaluace . . . 8
1.3.5 Testování a propojení . . . 9
1.3.6 Volání externích akcí . . . 9
1.3.7 Využití paměťových slotů . . . 11
1.3.8 Zhodnocení . . . 11
1.4 Rasa X . . . 11
1.4.1 Funkcionality . . . 12
1.4.2 Zhodnocení . . . 13

2 Analýza **15**
2.1 Analýza požadavků . . . 15
2.1.1 Funkční požadavky . . . 15
2.1.1.1 Uživatelé a přístupy . . . 16
2.1.1.2 Klasifikace zpráv . . . 16
2.1.1.3 Vzorové příběhy . . . 16

2.1.1.4 Proces trénování 17
2.1.2 Nefunkční požadavky 17
2.1.2.1 Použitelnost 18
2.1.2.2 Spolehlivost 18
2.1.2.3 Výkon 18
2.1.2.4 Podporovatelnost 19
2.1.2.5 Zabezpečení 19
2.2 Případy užití 19
2.2.1 Seznam případů užití 20
2.2.2 Popisy případů užití 21
2.2.3 Pokrytí funkčních požadavků 23

3 Návrh **25**
3.1 Hlavní komponenty aplikace 25
3.1.1 Vybrané technologie 25
3.2 Vrstvy aplikace 28
3.2.1 Datová vrstva 29
3.2.1.1 PostgreSQL databáze 30
3.2.1.2 Rasa server 31
3.2.1.3 S3 úložiště 31
3.2.2 Business vrstva 32
3.2.2.1 Business entity 32
3.2.2.2 Business procesy 33
3.2.3 Servisní vrstva 35
3.2.4 Prezentační vrstva 38
3.3 Zabezpečení 40
3.3.1 Autentizace 40
3.3.2 Autorizace 43

4 Implementace **45**
4.1 Nástroje pro vývoj 45
4.2 Backend systému 45
4.3 Frontend systému 53
4.3.1 Ukázky implementace 53
4.3.2 Uživatelské rozhraní 55

5 Testování **59**
5.1 Základní pojmy 59
5.2 Použité technologie 60
5.3 Jednotkové testy 61
5.4 Integrační testy 62

Závěr **65**
Splnění cílů 65

Přínosy práce 66
Navazující plány 66

Literatura **67**

A Seznam použitých zkratek **73**

B Obsah přiložené SD karty **75**

C Případy užití **77**

D Databázový model **79**

Seznam obrázků

1.1 Proces zpracování zprávy frameworkem Rasa Open Source 4
1.2 Náhled na schválení klasifikace zprávy v systému Rasa X [1] 12
1.3 Náhled na modely chatbota v systému Rasa X [2] 13

2.1 Vztah dědění aktérů v případech užití 20
2.2 Diagram aktivit procesu přihlášení 21
2.3 Diagram aktivit schválení vzorové zprávy 22

3.1 Diagram komponent systému 26
3.2 Rozdělení aplikace do vrstev [3] 29
3.3 Třídy databázových modelů týkajících se vzorových zpráv 33
3.4 Třídy databázových modelů týkajících se vzorových příběhů 34
3.5 Třídy kontrolérů obsahujících business logiku 35
3.6 Sekvenční diagram procesu trénování 36
3.7 Sekvenční diagram zpracování GraphQL požadavku 39
3.8 Proces přihlášení uživatele s využitím Implicit Grant 42

4.1 Náhled uživatelského rozhraní pro přihlášení 56
4.2 Náhled uživatelského rozhraní pro opětovné přihlášení 56
4.3 Náhled uživatelského rozhraní pro výpis vzorových zpráv 56
4.4 Náhled uživatelského rozhraní pro editaci vzorové zprávy 57
4.5 Náhled uživatelského rozhraní pro výpis synonym 57
4.6 Náhled uživatelského rozhraní pro výpis částí příběhu 58
4.7 Náhled uživatelského rozhraní pro výpis trénování 58
4.8 Náhled uživatelského rozhraní pro vizualizaci evaluace dat 58

C.1 Diagram případů užití pro přístup uživatelů 77
C.2 Diagram případů užití procesu trénování 77
C.3 Diagram případů užití klasifikace zpráv 78
C.4 Diagram případů užití vzorových příběhů 78

D.1 Relační databázový model uživatelů 79
D.2 Relační databázový model konverzací 79
D.3 Relační databázový model vzorových zpráv 80
D.4 Relační databázový model vzorových příběhů 81

Seznam ukázek kódu

1.1 Konfigurační soubor s doménami chatbota 6
1.2 Trénovací data vzorových zpráv 7
1.3 Trénovací data vzorových příběhů 8
1.4 Implementace externí akce 10
3.1 Jednoduchý Node.js server 27
3.2 Jednoduchá React komponenta 28
3.3 Definice objektu uživatele v knihovně Sequelize 30
3.4 Metody pro operace se záznamy v knihovně Sequelize 31
3.5 Ukázka definice GraphQL schématu 37
3.6 Implementace GraphQL resolverů 37
3.7 GraphQL dotaz pro získání jména uživatele 38
3.8 Odpověd získaná na základě GraphQL dotazu 38
3.9 Jednoduchá aplikace frameworku React Admin 40
3.10 Vykreslení komponenty dle role uživatele 43
4.1 Definice databázového modelu entity trénování 46
4.2 Metoda pro vytvoření záznamu v databázi 47
4.3 Metoda pro spuštění procesu trénování 48
4.4 Třída chyby pro porušení validačních pravidel 48
4.5 Implementace procesu trénování 49
4.6 Sestavení resolveru pro vytváření záznamů 50
4.7 Definice GraphQL schématu externí akce 51
4.8 Implementace třídy ApolloServer 51
4.9 Metoda pro zjištění stavu Rasa serveru 52
4.10 Vytvoření komunikačního kanálu s RabbitMQ 52
4.11 Inicializace strategie pro autentizaci uživatele 52
4.12 React komponenta pro výpis záznamů 53
4.13 React komponenta pro editaci záznamu 54
4.14 React komponenta pro vykreslení aplikace 54
4.15 Konfigurace odchozích požadavků GraphQL klienta 55
5.1 Jednoduchý test pomocí knihovny Jest 60

5.2 Jednotkový test metody pro označování entit 61
5.3 Jednotkový test metody pro vytváření záznamů 62
5.4 Jednotkový test autorizace uživatele 62
5.5 Integrační test třídy pro stavbu resolverů 63
5.6 Integrační test zpracování GraphQL dotazu 63

Seznam tabulek

2.1 Pokrytí funkčních požadavků 1. část 23
2.2 Pokrytí funkčních požadavků 2. část 23
2.3 Pokrytí funkčních požadavků 3. část 23

Úvod

Popis problému

S výrazným rozvojem oboru umělé inteligence v posledních letech se stále více objevuje její aplikace nejen v akademické, ale i komerční sféře. Společnosti se snaží pomocí této technologie částečně či úplně automatizovat některé své obchodní procesy, a tím šetřit nejen finanční, ale i časové prostředky.

Jednou z možností její aplikace je zákaznická podpora, která se zákazníky často opakovaně řeší podobné či úplně stejné otázky. S takovou činností může efektivně pomoci počítačový program schopný rozpoznat, na co se zákazník ptá, a dle kontextu konverzace určit, jak nejlépe na položenou otázku odpovědět. V případě, že se jedná chatovou konverzaci, pak takový program nazýváme chatbotem.

Sestavení komplexního chatbota, jeho provoz a průběžné vylepšování na základě zpětné vazby může být bez správných nástrojů velmi náročné. Z toho důvodu se nabízí řešení ve vytvoření administračního systému, díky kterému by mohli lidé i bez technického vzdělání chatbota snadno spravovat.

Startup WaldoBot

WaldoBot je začínající studentský startup založený autorem práce Janem Šafaříkem a spolužákem Tomášem Stanovčákem, který se zabývá návrhem, vývojem a provozem chatbotů určených především pro zákaznickou podporu e-shopů. Startup se již dočkal několika úspěchů, mezi které patří účast na sérii workshopů s názvem Google For Startups Academy, kam byl pořadateli vybrán z desítek zájemců, rozhovor v novinovém článku pro Lidové noviny s názvem „Česká věda žije! Seznamte se s vycházejícími hvězdami české vědy a techniky“ a postup mezi deset nejlepších startupů v soutěži Nastartujte se od Komerční banky. Největší pozornost však získal nekomerční projekt Ko-

ronavirus 24, v rámci kterého byl vytvořen chatbot odpovídající na otázky ohledně onemocnění COVID-19.

Díky účasti v programu Google For Startups Academy došlo k navázání kontaktu se společností ANECT a.s., která startup oslovila s poptávkou na vytvoření chatbota a administračního systému pro jeho správu. Tento chatbot je vyvíjen v rámci závěrečné bakalářské práce Tomáše Stanovčáka a měl by pomáhat zaměstnancům pracujícím v oddělení prodeje s vyhledáváním informací v produktovém portfoliu. To je v současné době vedeno pomocí excelovské tabulky, v níž je orientace z důvodu velkého množství dat obtížná.

Cíl práce

Cílem práce je vyvinout systém pro správu chatbotů vytvořených pomocí frameworku Rasa Open Source. K tomu bude třeba zmapovat stávající řešení, analyzovat požadavky na systém, sestavit návrh vlastního řešení, to implementovat a na závěr otestovat.

Motivace

Hlavní motivací pro vytvoření této práce je položení základů nového projektu startupu WaldoBot, který v budoucnu zefektivní správu klientských chatbotů, a tím sníží náklady na jejich provoz. Další motivací je možnost vyzkoušet si vývoj produktu od jeho analýzy, přes návrh, až po samotnou implementaci, což je pro realizace dalších aplikací nepostradatelnou zkušeností. Na závěr je to pak příležitost prohloubit své znalosti o frameworku Rasa Open Source a vyzkoušet si práci s dotazovacím jazykem GraphQL.

Struktura práce

Práce je rozdělena na teoretickou a praktickou část. V teoretické části budou postupně rozebrány důležité pojmy a definice z oblasti návrhu chatbotů a zpracování přirozeného textu. Dále bude představen framework Rasa Open Source, s jehož rozhraním musí být systém kompatibilní, a zároveň budou zmapována stávající řešení implementace chatbotů pomocí tohoto frameworku.

Praktická část pak popisuje hlavní fáze vývoje systému. První z nich je analýza, kde budou rozebrány funkční a nefunkční požadavky spolu s případy užití. Druhou je návrh systému, kde bude popsána architektura frontendové a backendové části, včetně popisu zvolených technologií. Třetí fází je samotná implementace, obsahující ukázky některých metod a uživatelského rozhraní, po které bude následovat krátký náhled na provedené testy backendové části aplikace. Na závěr práce dojde ke shrnutí výsledků vývoje a nastínění dalších plánů na rozvoj projektu.

Kapitola **1**

Teoretická část

V této kapitole budou vysvětleny základní pojmy, dále bude představen framework Rasa Open Source a na závěr budou zhodnocena stávající řešení implementace chatbotů pomocí textových konfiguračních souborů a nástrojů systému Rasa X.

1.1 Pojmy a definice

Chatbot je počítačový program, který na vstupu přijme textovou zprávu v přirozeném jazyce a vygeneruje co nejvhodnější odpověď, kterou pak odešle nazpět uživateli. Tento program je nezávislý na chatovací platformě a může být skrze API propojen s aplikacemi, jako je například Messenger, Slack, Skype nebo Microsoft Teams. [4] Chatbot může být využit jako operátor zákaznické podpory nebo osobní asistent, mezi které patří například Siri a Alexa. [5]

NLP (natural language processing) označuje proces zpracování velkého množství dat přirozeného jazyka, která jsou pak pomocí dalších algoritmů převedena do strukturované podoby. Ta obsahuje informace o základních stavebních elementech věty a jejich vztazích. [6]

NLU (natural language understanding) označuje proces, při kterém se program snaží pochopit vstup, který ve formě textové zprávy nebo hlasového vstupu získal. Typicky dokáže rozpoznat význam, i přes to, že se uživatel zeptá různými způsoby. [6]

Intent je označení pro záměr uživatele, který s chatbotem komunikuje. Tato informace by měla být ve zprávě identifikována pomocí procesu NLP a reakce na ní vyhodnocena pomocí NLU. [4] Jako příklad může být věta „V kolik hodin odlétá nejbližší letadlo do Paříže?“. Zde je záměrem uživatele zjistit čas odletu letadla.

Entita označuje důležité klíčové slovo nebo údaj, který chatbotovi napomáhá při pochopení, jakého předmětu se zpráva konkrétně týká. [4] Pro ukázku je použita předchozí zpráva „V kolik hodin odlétá nejbližší letadlo do Paříže?“. Již bylo zjištěno, že je cílem uživatele zjistit čas odletu letadla. Ale nacházejí se zde dvě entity, které specifikují požadavek, a to jsou `nejbližší` a `do Paříže`, díky kterým může chatbot odpověď upřesnit.

1.2 Rasa Open Source

Rasa Open Source je framework využívající umělou inteligenci ke stavbě chatbotů. [7] Skládá se primárně ze čtyř komponent, kterými jsou:

NLU určení záměru uživatele a získání klíčových informací

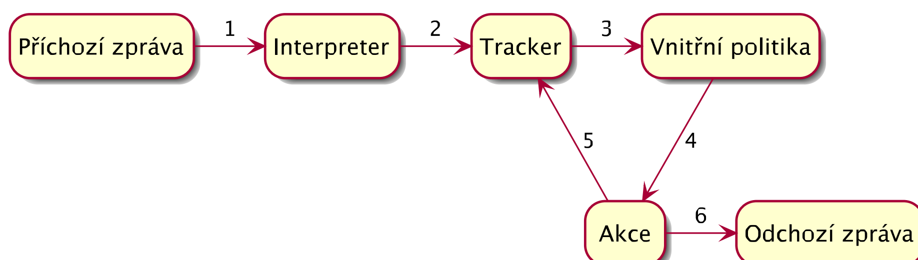
Core výběr odpovědi dle kontextu a historie konverzace

komunikační kanály propojení chatbota s chatovací platformou

integrace služeb rozhraní pro připojení k externímu systému [8]

Na obrázku 1.1 je znázorněn proces přijetí zprávy od uživatele, její vyhodnocení a provedení reakce. Proces probíhá následujícím způsobem:

1. `Interpreter` zpracuje zprávu, určí intent a označí entity.
2. `Tracker` načte aktuální stav konverzace dle identifikátoru uživatele.
3. Načtený stav je odeslán k vyhodnocení dalšího postupu.
4. Dle kontextu je zvolena další akce.
5. `Tracker` poznamená vybranou akci do paměti.
6. Uživateli je odeslána odpověď. [9]



Obrázek 1.1: Proces zpracování zprávy frameworkem Rasa Open Source

Aby bylo možné začít s frameworkem pracovat, je nutné mít dle [10] nainstalované vývojové prostředí jazyka Python a systém pro správu balíků Pip. To lze provést například pro distribuci Ubuntu operačního systému Linux pomocí těchto příkazů:

```
sudo apt update
sudo apt install python3-dev python3-pip
```

Pro zachování konzistence verzí jednotlivých balíčků se podle [10] doporučuje vytvořit oddělené virtuální prostředí:

```
python3 -m venv rasa-env
source rasa-env/bin/activate
```

Nyní lze nainstalovat samotný framework:

```
pip install rasa
```

Nastavení chatbota a správu trénovacích dat lze pak provádět dvěma způsoby, kterými jsou konfigurační soubory a systém Rasa X. Oba budou postupně představeny a zhodnoceny.

1.3 Konfigurační soubory

Nejjednodušší cesta, jak vytvořit nového chatbota, je pomocí konfiguračních souborů ve formátu Yaml a trénovacích dat ve formátu Markdown. Nový projekt lze podle [11] založit příkazem:

```
rasa init --no-prompt
```

Tím dojde k vytvoření všech potřebných souborů a vygenerování prvního modelu. Adresářová struktura projektu je dle [11] následující:

```
/
├── domain.yaml ........................ domény chatbota
├── config.yaml ........................ konfigurace NLU a Core komponent
├── credentials.yaml ................... nastavení komunikačních kanálů
├── endpoints.yaml ..................... nastavení externích služeb
├── models ............................. složka s výchozím modelem
└── data ............................... složka s trénovacími daty
    ├── nlu.md ......................... trénovací data se vzorovými zprávami
    └── stories.md ..................... trénovací data se vzorovými příběhy
```

1.3.1 Domény chatbota

Soubor `domain.yaml` z ukázky 1.1 popisuje domény chatbota používané při sestavování trénovacích dat. Ty jsou označeny následujícími klíči:

intents seznam intentů použitých při klasifikaci zpráv

entities seznam entit označujících klíčová slova

actions seznam externích akcí, jako je například volání API služby

responses šablony textových odpovědí zasílaných uživateli

slots seznam paměťových slotů pro ukládání hodnot během konverzace [12]

```yaml
intents:
- greeting
- request_order_detail

entities:
- order_number

actions:
- action_fetch_order_detail

responses:
  utter_greeting:
    - text: "Dobrý den, co pro vás mohu udělat?"
  utter_order_detail:
    - text: "Vaše objednávka je ve stavu {status}."
  utter_order_not_found:
    - text: "Objednávka číslo {number} nebyla nalezena."
  utter_request_order_number:
    - text: "Zadejte prosím platné číslo objednávky."

slots:
  order_number:
    type: unfeaturized
  is_order_found:
    type: bool
```

Ukázka kódu 1.1: Konfigurační soubor s doménami chatbota

1.3.2 Trénovací data vzorových zpráv

Trénovací data vzorových zpráv obsahují příklady seskupené dle intentu spolu s označením entit, jejichž rozpoznání může být podpořeno regex výrazem. Pomocí znaku `##` je označen začátek příkladu následující slovem `intent` či `regex` a názvem intentu či entity. Na každý nový řádek je pak vložen jeden příklad, ve kterém mohou být pomocí hranatých závorek označeny entity. [13]

V ukázce 1.2 jsou celkem dva intenty představující pozdrav a dotaz na detail objednávky. Samotné číslo objednávky je pak označeno jako entita s klíčem `order_number` a pomocným regex výrazem.

```
## intent:greeting
- dobrý den
- zdravím
- ahoj

## intent:request_order_detail
- jaký je stav objednávky číslo [3894039839](order_number)?
- jak je na tom objednávka číslo [3932671738](order_number)?
- jaký je stav mojí objednávky [0292283983](order_number)?

## regex:order_number
- (^[0-9]*$)
```

Ukázka kódu 1.2: Trénovací data vzorových zpráv

1.3.3 Trénovací data vzorových příběhů

Trénovací data vzorových příběhů jsou složena z příkladů konverzací mezi uživatelem a chatbotem. Pomocí znaku `##` je označen začátek příběhu spolu s jeho krátkým popiskem. Za znakem `*` je zadán název předpokládaného intentu společně s entitami obalenými do složených závorek. V seznamu položek začínajících znakem `-` jsou pak uvedeny reakce provedené ze strany chatbota. Těmi může být odeslání textové odpovědi, spuštění externí akce nebo nastavení hodnoty paměťového slotu. [14]

V ukázce 1.3 se nacházejí dva vzorové příběhy, kde první popisuje situaci úspěšného dotazu na detail objednávky a druhý situaci, kdy bylo nutno zopakovat číslo objednávky, neboť pod původním nebyla žádná nalezena. Pro účely rozhodování je do paměťového slotu s klíčem `is_order_found` uložena hodnota `true` či `false` v závislosti na tom, s jakým výsledkem skončila akce `action_fetch_order_detail`.

```
## Získání detailu objednávky
* greeting
  - utter_greeting
* request_order_detail{"order_number": "0292283983"}
  - slot{"order_number": "0292283983"}
  - action_fetch_order_detail
  - slot{"is_order_found" : "true"}

## Zadání neplatného čísla objednávky a jeho oprava
* greeting
  - utter_greeting
* request_order_detail{"order_number": "3932671738"}
  - slot{"order_number": "3932671738"}
  - action_fetch_order_detail
  - slot{"is_order_found" : "false"}
  - utter_request_order_number
* request_order_detail{"order_number": "2820029383"}
  - slot{"order_number": "2820029383"}
  - action_fetch_order_detail
  - slot{"is_order_found" : "true"}
```

Ukázka kódu 1.3: Trénovací data vzorových příběhů

1.3.4 Trénování a evaluace

Na závěr je nutné z trénovacích dat vygenerovat model, který kombinuje jak klasifikaci vzorových zpráv pomocí NLU, tak vyhodnocování průběhu konverzace na základě vzorových příběhů. Proces je dle [15] možné spustit pomocí následujícího příkazu:

```
rasa train
```

Kvalitu trénovacích dat lze ověřit jejich náhodným rozdělením na trénovací a testovací set v poměru 2:8. Trénovací set je pak použit pro vygenerování NLU modelu za účelem evaluace a testovací pro ověření správnosti klasifikace. Algoritmus zadává příklady zpráv z testovacího setu do NLU modelu a porovnává výsledek s předpokládaným výstupem. [16] Celý proces lze provést následovně:

```
rasa data split nlu
rasa train nlu --data train_test_split/training_data.md
rasa test nlu --nlu train_test_split/test_data.md
```

Po dokončení jsou výsledky uloženy do dvou souborů ve složce `results`. První soubor obsahuje hodnoty `recall`, `precision` a `f1-score` každého z intentů, druhý soubor pak seznam testovacích dat, u nichž došlo k chybné klasifikaci. Hodnoty získané procesem evaluace jsou definovány takto:

recall podíl pravdivě pozitivních měření vůči všem reálně pozitivním

precision podíl pravdivě pozitivních měření vůči pozitivně předpovězeným

f1-score podíl pravdivě pozitivních měření vůči pravdivě pozitivním a aritmetickému průměru falešně negativních a falešně pozitivních [17]

1.3.5 Testování a propojení

S využitím vygenerovaného modelu lze chování chatbota vyzkoušet simulací chatového okna dle [15] přímo v příkazové řádce systému:

```
rasa shell
```

Pro připojení chatbota k chatovacímu kanálu či aktivaci API rozhraní je nutné podle [15] spustit celý Rasa server:

```
rasa run --enable-api
```

Mezi dostupné chatovací kanály patří například Messenger, Slack nebo Skype. Lze je aktivovat jejich přidáním do souboru `credentials.yml`. Jejich přesné nastavení závisí na konkrétní platformě. Konfigurace kanálu Messenger vypadá dle [18] takto:

```yaml
facebook:
  verify: "<verifikační-token>"
  secret: "<ověřovací-klíč>"
  page-access-token: "<přístupový-token>"
```

Tím dojde k vystavení API na adrese `/webhooks/facebook/webhook`, kterou lze později zadat jako adresu pro notifikaci o příchozích zprávách. [18]

1.3.6 Volání externích akcí

Externí akce slouží k přidání složitější business logiky a jsou definovány programátorem nejčastěji v jazyce Python s využitím knihovny Rasa SDK. Každá akce je potomkem rodičovské třídy `Action` a musí implementovat metody `name` a `run`, kde `name` vrací název akce použité v souboru `domain.yaml` a `run` vykonává samotnou logiku. [19]

V ukázce 1.4 je příklad akce `action_fetch_order_detail`. Probíhá tak, že se načte číslo objednávky uložené ve slotu `order_number`, to je předáno

jako argument metodě `fetch_order`, která odešle požadavek na API a vrátí objekt s detailem objednávky. Pokud je návratová hodnota prázdná, odešle se pomocí metody `utter_message` zpráva uživateli, že se objednávku nepodařilo nalézt a do slotu `is_order_found` se uloží hodnota `false`. V kladném případě se uživateli odešle zpráva s informací o stavu objednávky.

```
class ActionFetchOrderDetail(Action):
  def name(self):
    return "action_fetch_order_detail"

  async def run(self, dispatcher, tracker, domain):
    order_number = tracker.get_slot("order_number")
    order = await fetch_order(order_number)
    if order is not None:
      dispatcher.utter_message(
        template="utter_order_detail",
        status=order.status
      )
      return [SlotSet("is_order_found", true)]
    else:
      dispatcher.utter_message(
        template="utter_order_not_found",
        number=order_number
      )
      return [SlotSet("is_order_found", false)]
```

Ukázka kódu 1.4: Implementace externí akce

Soubor `actions.py` s implementací třídy je nutné předat frameworku a spustit dle [19] jako samostatný API server, se kterým bude moct Rasa server komunikovat pomocí HTTP požadavků:

```
rasa run actions
```

Na závěr je potřeba dle [20] chatbota se serverem propojit přidáním jeho URL adresy do souboru `endpoints.yml`:

```
action_endpoint:
  url: "http://localhost:5055/webhook"
```

1.3.7 Využití paměťových slotů

Paměťové sloty slouží k ukládání hodnot během konverzace mezi chatbotem a uživatelem. Tyto hodnoty pak mohou, ale i nemusejí, ovlivňovat predikci dalších událostí, což je určeno typem slotu. [21] Mezi nejdůležitější typy patří:

bool hodnota `true` nebo `false` (ovlivňuje průběh konverzace)

categorical hodnota ze seznamu (ovlivňuje průběh konverzace)

unfeaturized libovolná hodnota (nemá na průběh vliv) [21]

Slot typu `categorical` může být definován například takto:

```
slots:
  delivery_type:
    type: categorical
    values:
    - basic
    - expedited
```

Pokud existuje paměťový slot se stejným názvem jako má entita, pak je při extrakci z textové zprávy automaticky uložena její hodnota. Tomu lze zabránit změnou parametru `auto_fill` v definici slotu v souboru `domain.yml`. [21]

1.3.8 Zhodnocení

Výhody využití textových konfiguračních souborů jsou:

- rychlá stavba prototypu chatbota
- podpora všech funkcionalit frameworku
- jednoduchý zápis ve formátu Yaml a Markdown

Mezi nevýhody jejich využití patří:

- nepřehlednost s narůstajícím množstvím dat
- nemožnost zpětné kontroly proběhlé konverzace
- obtížnost pro člověka bez technického vzdělání

1.4 Rasa X

Rasa X je dalším projektem společnosti Rasa Technologies a jedná se o webovou aplikaci umožňující správu chatbota, která však na rozdíl od frameworku není open-source. [22] Je dostupná ve verzi Community Edition a v rozšířené placené verzi Enterprise Edition. [23]

1.4.1 Funkcionality

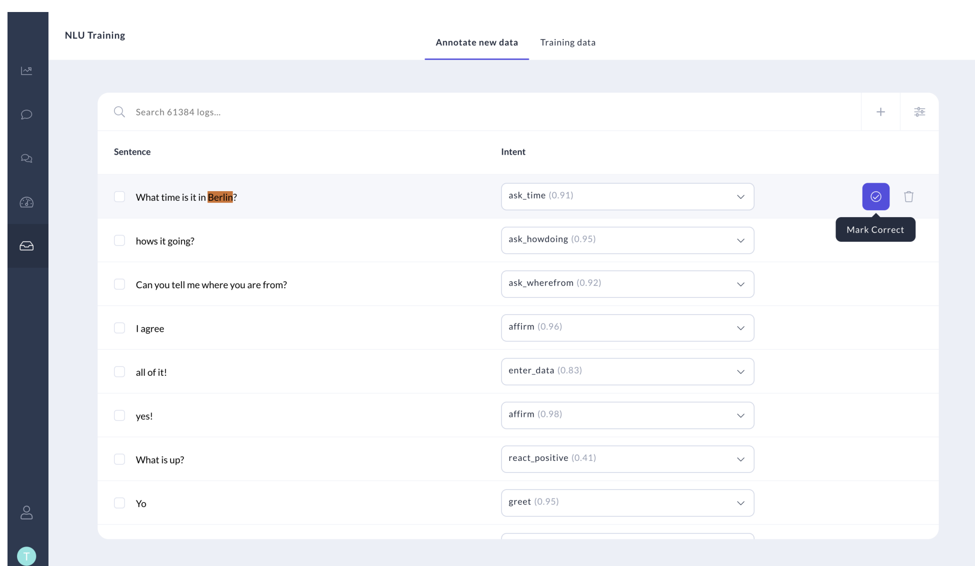
Základními funkcionalitami komunitní verze jsou:

- prohlížení a anotace reálných konverzací
- možnost sdílení chatbota s externími testery
- verzování trénovacích dat a modelů pomocí Git systému [23]

Mezi rozšířené funkce placené verze pak patří:

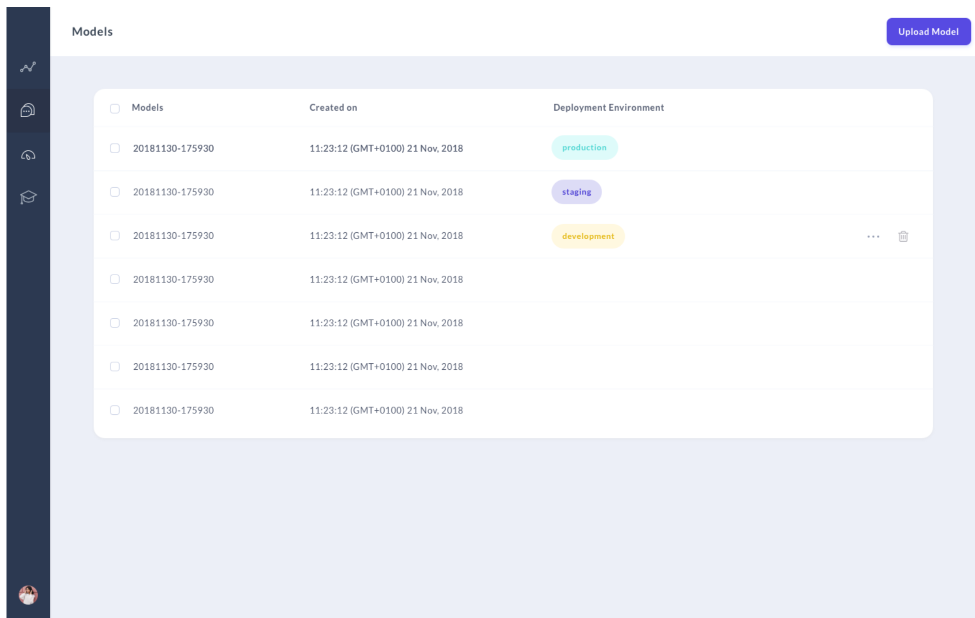
- analytiky chování uživatelů při komunikaci s chatbotem
- odlišný přístup do systému dle úrovně oprávnění
- nasazení modelů do oddělených prostředí
- systém jednotného přihlášení přes poskytovatele identity [23]

Na obrázku 1.2 je náhled na uživatelské rozhraní, kde probíhá schválení klasifikace textové zprávy importované z reálné konverzace mezi uživatelem a chatbotem. Větě „What time is it in Berlin?“ byl s přesností 95 % přiřazen intent s názvem `ask_time` a entita označující město Berlín.



Obrázek 1.2: Náhled na schválení klasifikace zprávy v systému Rasa X [1]

Správa modelů a jejich nasazení do oddělených prostředí je zobrazena na obrázku 1.3. První model je použit v produkčním prostředí, druhý v testovacím a třetí ve vývojovém. Díky tomu lze jednotlivé verze testovat bez negativního vlivu na koncové uživatele.



Obrázek 1.3: Náhled na modely chatbota v systému Rasa X [2]

1.4.2 Zhodnocení

Výhody využití systému Rasa X jsou:

- snadná správa reálných konverzací
- jednoduché a intuitivní uživatelské rozhraní
- zdarma dostupná komunitní verze
- integrované verzování dat a modelů

Nevýhody využití systému Rasa X:

- některé funkcionality existují pouze v placené verzi
- cena za placenou verzi není jasně definovaná
- zdrojové kódy systému nejsou volně dostupné
- navrženo jako podpůrný nástroj konfiguračních souborů

Kapitola 2

Analýza

Tato kapitola se věnuje stanovení funkčních a nefunkčních požadavků a vytvoření případů užití. Během analýzy je zjišťováno, co se bude řešit, spíše než jakým způsobem bude řešení provedeno. Na konci kapitoly by mělo být jasné, kdo bude systém ovládat, co bude systém umět a za jakých okolností ho bude možné používat. K tomu je třeba dle [24] rozložit systém na menší komponenty a porozumět účelu a funkci každé z nich.

2.1 Analýza požadavků

Analýza požadavků je prvním krokem při vývoji systému, neboť jsou při této fázi získány základní vstupy ze strany klienta a v pozdějších fázích slouží jako zadání pro návrh a implementaci. Požadavky typicky dělíme na funkční a nefunkční. [25, 26] Aby bylo možné ověřit kvalitu dodaného systému, je nutné mimo samotné funkcionality definovat i další důležité metriky. Pro tento účel existují různá dělení, avšak zde je použita klasifikace FURPS, reprezentující slova functionality, usability, reliability, performance a supportability. [27]

2.1.1 Funkční požadavky

Funkční požadavky popisují, co by měl systém umět a často odrážejí firemní postupy či pravidla. Tyto požadavky je třeba specifikovat takovým způsobem, aby jim rozuměl běžný uživatel systému. [25] Každý požadavek by měl obsahovat skupinu uživatele, co konkrétně může uživatel provádět, s jakou mírou nutnosti a za jakých podmínek. Dále může být uvedena například priorita požadavku nebo vlastník, který bude požadavek akceptovat. [26]

Požadavky jsou z důvodu přehlednosti rozděleny do čtyřech kategorií. První kategorie se zabývá správou uživatelů, druhá řeší administraci objektů použitých při klasifikaci zpráv, třetí se věnuje správě vzorových příběhů a poslední řeší generování modelů spolu s evaluací trénovacích dat.

2.1.1.1 Uživatelé a přístupy

FR1.1 Autentizace a autorizace uživatele

Uživatel se musí před vstupem do systému autentizovat pomocí služby Azure Active Directory. Přihlásit se může pouze za předpokladu, že existuje jeho záznam v databázi uživatelů, která je spravována nezávisle na poskytovateli identity, a která umožňuje autorizaci uživatele. Po dokončení práce se může uživatel ze systému též odhlásit.

FR1.2 Správa uživatelů

Administrátor může spravovat uživatele systému. Administrátor musí při vytváření zadat e-mail uživatele a nastavit roli anotátora nebo vývojáře.

2.1.1.2 Klasifikace zpráv

FR2.1 Správa intentů

Vývojář může vytvářet nové intenty a editovat existující. Anotátor pak může tyto intenty zobrazit. Vývojář může intent odstranit, čímž dojde k odebrání přidružených vzorových zpráv. Intent nelze odstranit, pokud je obsažen ve vzorovém příběhu.

FR2.2 Správa entit

Vývojář může vytvářet nové entity a editovat existující. Anotátor pak může tyto entity zobrazit. Vývojář může doplnit regex výraz a nastavit, zda se má entita automaticky ukládat do paměťového slotu. Vývojář může entitu odstranit, čímž dojde k jejímu odebrání ze všech vzorových zpráv. Entitu nelze odstranit, pokud je obsažena ve vzorovém příběhu. Vývojář může pro každou entitu spravovat hodnoty a jejich synonyma. Hodnotu nelze odstranit, pokud je součástí vzorového příběhu.

FR2.3 Správa vzorových zpráv

Anotátor může spravovat vzorové zprávy. Ke zprávě musí být přiřazen intent, případně entity nacházející se uvnitř věty. Pro každou označenou entitu může anotátor vybrat její konkrétní hodnotu. Z produkčního chatbota musejí být automaticky importovány zprávy získané během konverzace mezi chatbotem a koncovým uživatelem. Aby mohly být tyto zprávy použity v procesu trénování, musejí být anotátorem označené za schválené. Zprávy mohou být filtrovány dle intentu nebo stavu schválení.

2.1.1.3 Vzorové příběhy

FR3.1 Správa textových odpovědí

Vývojář může spravovat textové odpovědi chatbota. Vývojář může odpověď odstranit pouze za předpokladu, že se nenachází v žádném ze vzorových příběhů.

FR3.2 Správa externích akcí

Vývojář může spravovat externí akce. Akce může být vývojářem odstraněna, pokud se nenachází v žádném ze vzorových příběhů. Systém musí pro každou externí akci vygenerovat unikátní identifikátor, pomocí kterého je později svázána s její implementací.

FR3.3 Správa paměťových slotů

Vývojář může spravovat paměťové sloty typu categorical. Pro každý slot musí vývojář vytvořit alespoň dvě jeho hodnoty. Slot lze odstranit pouze tehdy, pokud se nenachází v žádném ze vzorových příběhů.

FR3.4 Správa vzorových příběhů

Vývojář může spravovat vzorové příběhy. Každý příběh musí být sestaven z navazujících částí, které reprezentují dotaz uživatele (tj. kombinace intentu s entitami) a jednu nebo více reakcí chatbota. Tou může být volání externích akcí, uložení hodnoty do paměťového slotu nebo zaslání textové odpovědi nazpět uživateli.

FR3.5 Prohlížení reálných konverzací

Vývojář může zobrazit reálné konverzace importované z produkčního chatbota. Vývojář může pro každou z nich zobrazit posloupnost zpráv mezi uživatelem a chatbotem. Vývojář může konverzaci odstranit.

2.1.1.4 Proces trénování

FR4.1 Správa modelů

Vývojář může spustit proces trénování, k němuž budou použita všechna trénovací data, a to jak schválené vzorové zprávy, tak vzorové příběhy. Před každým spuštěním musí systém provést validaci všech vstupních dat. Výstupem procesu je model chatbota, který je spolu s trénovacími daty ve formátu Yaml a Markdown nahrán na oddělené úložiště. Vývojář může prohlížet záznamy jednotlivých trénování. Vývojář může model aplikovat na produkční Rasa server, čímž dojde k výměně aktivního modelu za nový.

FR4.2 Evaluace trénovacích dat

Vývojář může pro každý proces trénování zobrazit evaluaci trénovacích dat, která v grafické podobě obsahuje výpis všech intentů a entit spolu s hodnotami parametrů precision, recall a f1-score.

2.1.2 Nefunkční požadavky

Nefunkční požadavky popisují obecné nároky na systém jako celek a nejsou přímo spojeny s konkrétními požadavky na funkcionalitu systému z pohledu

uživatele. Obsahují různá omezení a mnohdy jsou důležitější než požadavky funkční, neboť určují celkovou kvalitu a spolehlivost systému. [25] Jejich dělení je odvozeno z klasifikace FURPS a mimo kategorii pro funkčnost, zastupující funkční požadavky, budou postupně popsány kategorie pro použitelnost, spolehlivost, výkon a podporovatelnost. [27, 28] Dále dle [27] patří požadavky na zabezpečení systému mezi funkční požadavky, avšak dle [26] mohou být součástí i nefunkčních požadavků, i přesto, že nemají v klasifikaci FURPS vlastní kategorii.

2.1.2.1 Použitelnost

Popisuje, jak je z hlediska uživatele snadné systém ovládat, jak dobře je zdokumentovaný a jak hodně je svým chováním konzistentní. [27]

NFR1.1 Schválení vzorových zpráv

Systém musí umožnit schválit několik vzorových zpráv najednou.

NFR1.2 Označení entit ve vzorové zprávě

Výběr entity musí být proveden označením slova pomocí myši.

NFR1.3 Jazyková mutace systému

Systém musí být kompletně přeložený do českého jazyka.

2.1.2.2 Spolehlivost

Tato kategorie je zaměřena na náchylnost systému k chybám. Určuje nároky na schopnost obnovení v případě selhání. [27]

NFR2.1 Dostupnost trénovacího serveru

Systém musí před zahájením procesu trénování zkontrolovat dostupnost trénovacího serveru a zabezpečit maximálně jeden aktivní proces.

NFR2.2 Uchovávání modelů

Systém musí vygenerované modely uchovávat v dedikovaném uložišti.

2.1.2.3 Výkon

Udává metriky z pohledu výkonu, jako je rychlost nebo efektivnost systému při odpovědi na jednotlivé požadavky. [27]

NFR3.1 Rychlost načítání obrazovek

Přechod mezi obrazovkami aplikace musí být kratší než 1 sekunda.

NFR3.2 Minimální počet vzorových zpráv

Systém musí umožnit spravovat alespoň 1 000 vzorových zpráv.

NFR3.3 Minimální počet vzorových příběhů

Systém musí umožnit spravovat alespoň 100 vzorových příběhů.

2.1.2.4 Podporovatelnost

Kategorie zaměřena na schopnost systém spolehlivě testovat, rozšiřovat, spravovat a konfigurovat. Mimo to může obsahovat požadavky na instalaci nebo přenosnost systému na různá běhová prostředí. [27]

NFR4.1 Kompatibilita s frameworkem

Systém musí být kompatibilní s verzí 1.7. frameworku Rasa Open Source.

NFR4.2 Minimální verze prohlížeče

Systém musí být přístupný ve formě webové aplikace s podporou internetového prohlížeče Chrome ve verzi 80 a vyšší.

2.1.2.5 Zabezpečení

Kategorie vložená dle [26] mezi nefunkční požadavky obsahuje bezpečnostní pravidla a opatření zamezující pracovat se systémem nepovolaným osobám.

NFR5.1 Komunikace s API rozhraním

Komunikace s API rozhraním backendu musí být zabezpečena pomocí JWT tokenu s klíčem podepsaným algoritmem RS256.

NFR5.2 Komunikace s Rasa serverem

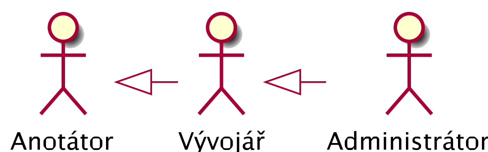
Veškerá komunikace s trénovacím a produkčním Rasa serverem musí být zabezpečena tokenem vloženým jako parametr v URL adrese požadavku.

2.2 Případy užití

Analýza požadavků může obsahovat příběhy a scénáře, které ukazují, jakým způsobem bude uživatel aplikaci používat. Nejběžnějším přístupem jsou takzvané `black-box` případy, popisující, jak se bude systém chovat, ale neřeší, co probíhá na pozadí. [29] Případy užití odráží funkční požadavky, a proto jsou důležitou součástí analýzy, neboť ověřují jejich naplnění. [28]

Každý případ užití je sestaven z aktivit jejichž spuštění ze strany aktéra vede k odpovědi ze strany systému. [29, 24] Aktéři jsou role zastoupené v rámci systému a vykazují určitá chování. Mohou to být nejen lidé, ale i organizace nebo systém samotný. [29] Podle složitosti může být případ užití popsán buď základní charakteristikou s vymezením cílů, nebo detailním popisem. Detailní popis pak obsahuje vstupní a výstupní podmínky, alternativní scénáře či výjimky vyvolané pro nastavená pravidla. [29, 24]

V následujících případech užití se vyskytují celkem tři aktéři, jejichž vztah dědění je zobrazen na obrázku 2.1.



Obrázek 2.1: Vztah dědění aktérů v případech užití

2.2.1 Seznam případů užití

Případy užití jsou opět z důvodu přehlednosti rozděleny do čtyřech kategorií, které korespondují s dělením funkčních požadavků. Všechny případy jsou graficky znázorněny na obrázcích C.1, C.2, C.3 a C.4 umístěných v příloze práce.

- Uživatelé a přístupy

  UC1.1 Přihlásit se

  UC1.2 Odhlásit se

  UC1.3 Spravovat uživatele

- Klasifikace zpráv

  UC2.1 Zobrazit intenty

  UC2.2 Spravovat intenty

  UC2.3 Zobrazit entity a jejich hodnoty

  UC2.4 Spravovat entity a jejich hodnoty

  UC2.5 Spravovat vzorové zprávy

  UC2.6 Schválit vzorovou zprávu

- Vzorové příběhy

  UC3.1 Spravovat textové odpovědi

  UC3.2 Spravovat externí akce

  UC3.3 Spravovat paměťové sloty

  UC3.4 Spravovat vzorové příběhy

  UC3.5 Zobrazit reálné konverzace

  UC3.6 Odstranit reálnou konverzaci

- Proces trénování

  UC4.1 Spustit proces trénování

  UC4.2 Zobrazit záznamy o trénování

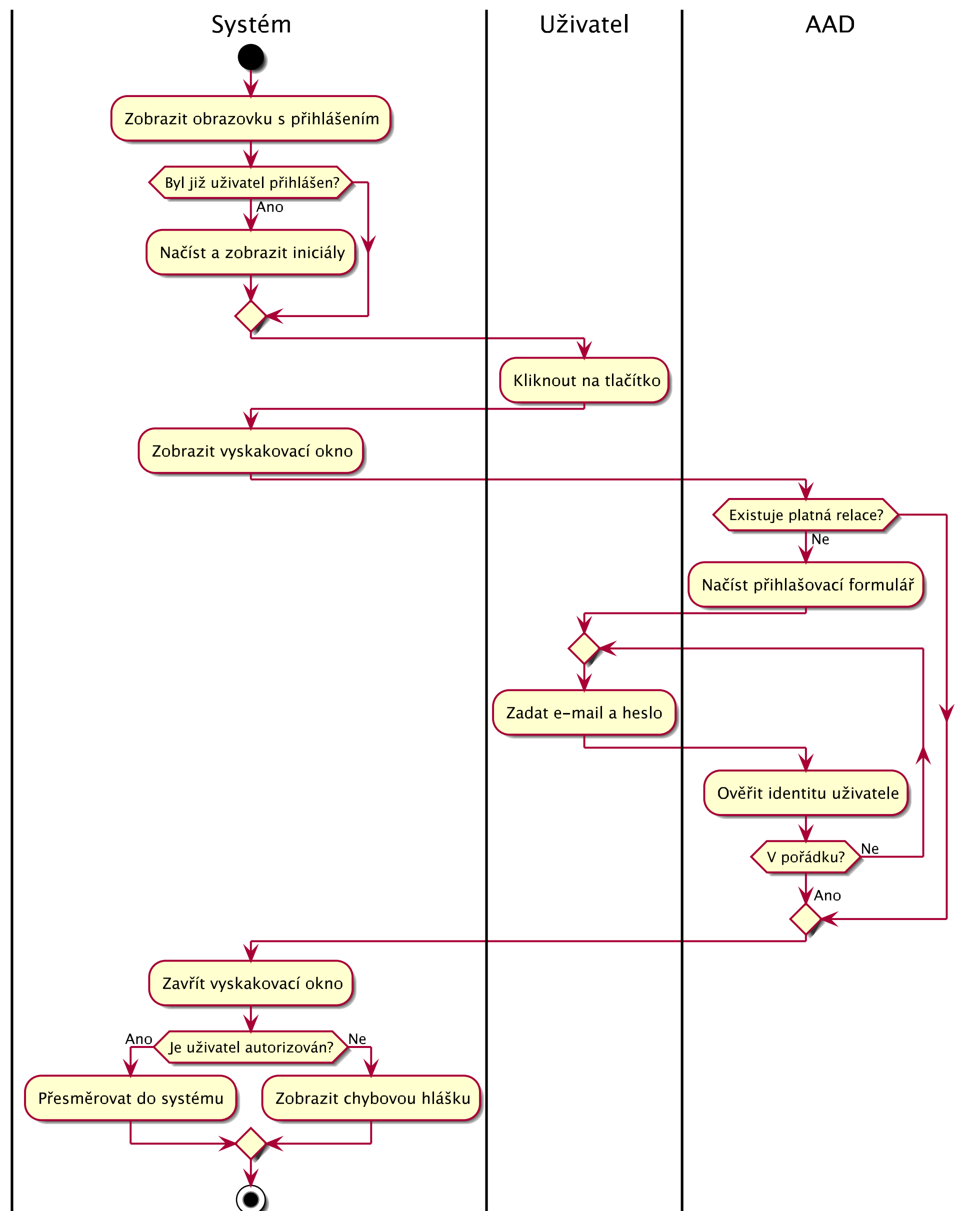
  UC4.3 Aplikovat model na produkci

  UC4.4 Zobrazit evaluaci trénovacích dat

2.2.2 Popisy případů užití

UC1.1 Přihlásit se

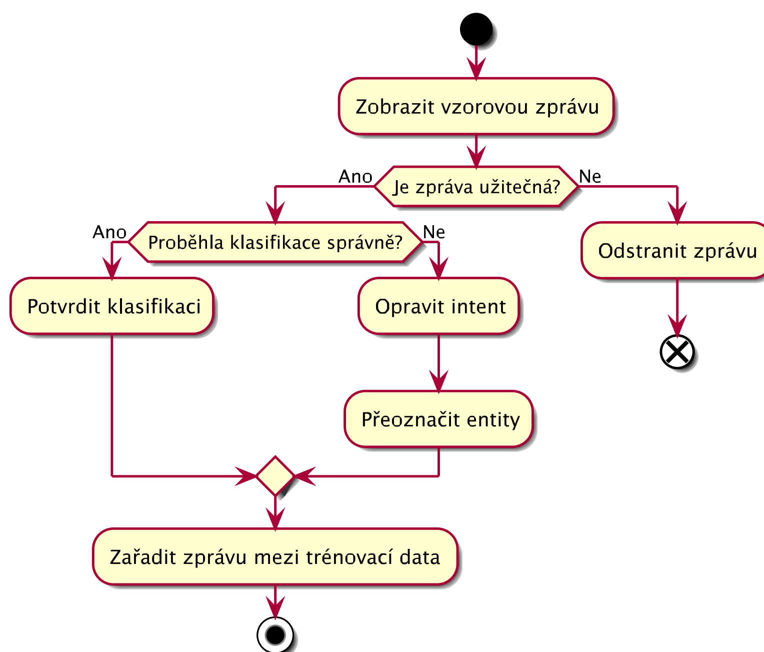
Umožňuje uživateli přihlášení do systému pomocí poskytovatele identity Azure Active Directory. Celý proces je znázorněn na obrázku 2.2.



Obrázek 2.2: Diagram aktivit procesu přihlášení

UC2.6 Schválit vzorovou zprávu

Umožňuje anotátorovi schválit klasifikaci zprávy importované z produkčního chatbota. Systém pro danou zprávu zobrazí odhadnutý intent a označí uvnitř věty všechny rozpoznané entity. Po zkontrolování klasifikace lze zprávu schválit a vytvořit tak novou vzorovou zprávu, nebo odstranit, pokud není pro trénování užitečná. Proces schválení zprávy je znázorněn na obrázku 2.3.



Obrázek 2.3: Diagram aktivit schválení vzorové zprávy

UC3.4 Spravovat vzorové příběhy

Umožňuje vývojářovi vytvářet vzorové příběhy, díky kterým se chatbot naučí reagovat na otázky uživatele dle aktuálního kontextu konverzace.

1. Systém zobrazí formulář pro vytvoření příběhu.
2. Vývojář zadá jeho název a přidá novou část příběhu představující dotaz ze strany uživatele.
3. Vývojář nastaví intent odpovídající dotazu a přidá entity, pokud dotaz obsahuje klíčová slova důležitá pro rozhodování.
4. Vývojář přidá jednu nebo více reakcí, kterou může být volání externí akce, nastavení paměťového slotu či odeslání odpovědi.
5. Vývojář celý proces opakuje, čímž vytvoří jeden vzorový příběh.

UC4.1 Spustit proces trénování

Umožňuje vývojářovi na základě aktualizace trénovacích dat spustit proces trénování, čímž dojde k vygenerování nového modelu. Nejdříve je ověřeno, zda již nějaký proces v aktuální chvíli neprobíhá. Pokud ano, informuje systém uživatele chybovou hláškou. V opačném případě je provedena validace trénovacích dat, jako je například kontrola dostatečného počtu vzorových zpráv a příběhů. Pokud je vše v pořádku, spustí systém nový proces, jehož stav lze sledovat v záznamech trénování.

2.2.3 Pokrytí funkčních požadavků

Aby bylo ověřeno, že systém splňuje všechny funkční požadavky, jsou v tabulkách 2.1, 2.2 a 2.3 pro každý případ užití označeny ty požadavky, které byly během vykonávání scénáře naplněny. [28]

	UC1.1	UC1.2	UC1.3	UC2.1	UC2.2	UC2.3	UC2.4
FR1.1	✓	✓					
FR1.2			✓				
FR2.1				✓	✓		
FR2.2						✓	✓

Tabulka 2.1: Pokrytí funkčních požadavků 1. část

	UC2.5	UC2.6	UC3.1	UC3.2	UC3.3	UC3.4	UC3.5
FR2.3	✓	✓					
FR3.1			✓				
FR3.2				✓			
FR3.3					✓		
FR3.4						✓	
FR3.5							✓

Tabulka 2.2: Pokrytí funkčních požadavků 2. část

	UC3.6	UC4.1	UC4.2	UC4.3	UC4.4
FR3.5	✓				
FR4.1		✓	✓	✓	
FR4.2					✓

Tabulka 2.3: Pokrytí funkčních požadavků 3. část

Kapitola 3

Návrh

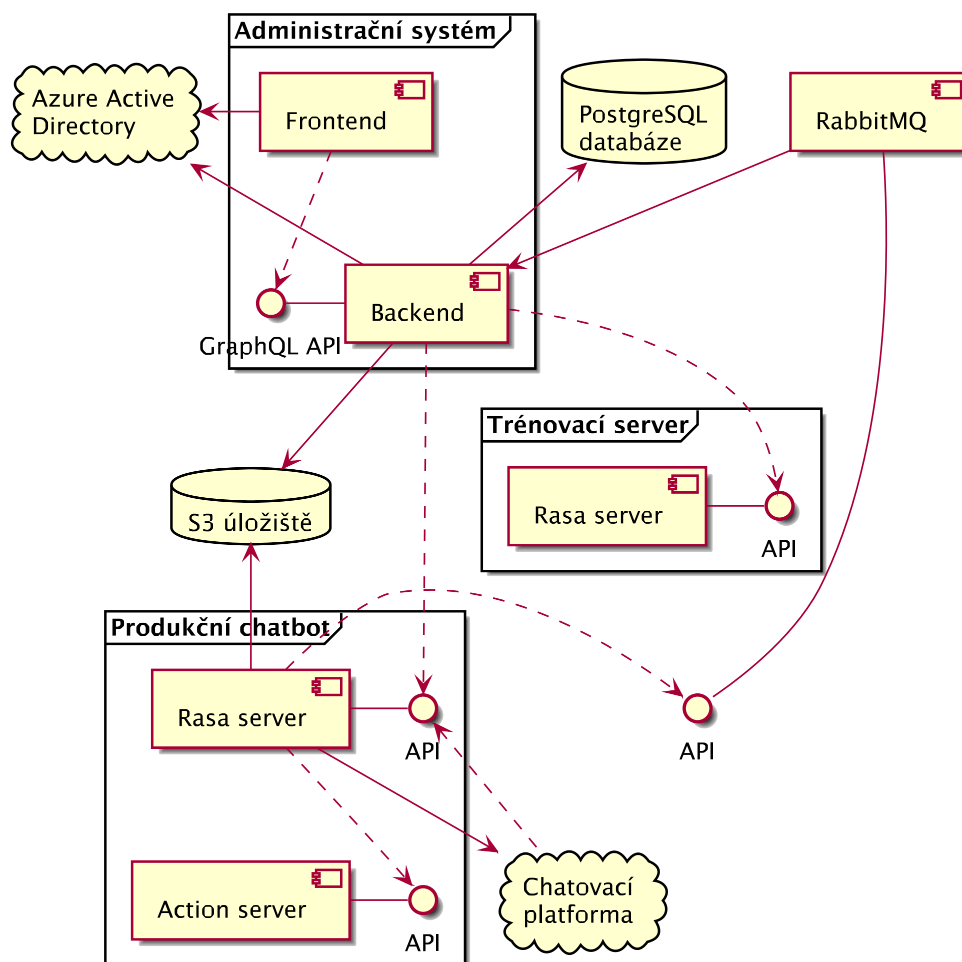
V této kapitole je probrán návrh aplikace včetně použitých technologií a knihoven. Nejprve jsou představeny základní komponenty systému, dále pak architektura backendové a frontendové části. Po definici jednotlivých vrstev aplikace následuje detailnější pohled na každou z nich. V datové vrstvě je zmíněna PostgreSQL databáze, API rozhraní Rasa serveru a možnosti S3 úložiště. V business vrstvě jsou popsány business entity spolu s jejich procesy. Předposlední část, věnovaná servisní vrstvě, obsahuje představení dotazovacího jazyka GraphQL a frameworku Express. V poslední prezentační vrstvě je ukázán framework React Admin a na závěr kapitoly jsou shrnuty přístupy k procesu autentitzace a autorizace uživatele s využitím externího poskytovatele identity Azure Active Directory.

3.1 Hlavní komponenty aplikace

Webová aplikace na frontendu komunikuje s API rozhraním backendu pomocí dotazovacího jazyka GraphQL. Backend tím poskytuje webové aplikaci data, která jsou uložena v PostgreSQL databázi. Dále backend komunikuje s API rozhraním produkčního a trénovacího Rasa serveru, přes který spouští například proces trénování nebo evaluaci trénovacích dat. Vygenerované modely jsou nahrávány na S3 úložiště, odkud je může produkční Rasa server stáhnout a načíst. Uživatelé komunikují s chatbotem prostřednictvím chatovací platformy a všechny jejich konverzace jsou skrze zprostředkovatele zpráv RabbitMQ přeposílány na backend, kde jsou zpracovány a uloženy do databáze. Během autentizace uživatele využívá backend i frontend k ověření totožnosti službu Azure Active Directory.

3.1.1 Vybrané technologie

Backend aplikace by měl splňovat požadavky na výkon, snadnou implementaci a škálovatelnost. Z toho důvodu je vybrána technologie Node.js, což je



Obrázek 3.1: Diagram komponent systému

asynchronní, událostmi řízené runtime prostředí jazyka JavaScript, určené pro tvorbu webových aplikací. Namísto vytváření nových vláken, využívá Node.js smyčku událostí zvanou `Event Loop`, která umožňuje neblokující I/O operace. Díky tomu nemusí programátor řešit uváznutí programu z důvodu zablokování procesů (tzv. deadlock), neboť Node.js nevyužívá systém zámků. [30] Mezi jeho hlavní výhody dle [31] patří:

- Vysoký výkon s ohlednem na škálovatelnost
- Psaní kódu v jazyce JavaScript, který je vhodný rovněž pro frontend
- Nezávilost na platformě a operačním systému

- Využití správce balíků NPM nabízející množství knihoven
- Popularita a široká komunita vývojářů

V ukázce 3.1 je na portu 3000 spuštěn server, který bude na příchozí HTTP požadavky odpovídat textem „Ahoj světe!“ a stavovým kódem 200.

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Ahoj světe!');
});

server.listen(3000, () => {
  console.log(`Server running on port ${port}`);
});
```

Ukázka kódu 3.1: Jednoduchý Node.js server

Z důvodu vyšší přehlednosti a snadnější správy obou částí systému, je pro frontend vybrán rovněž jazyk JavaScript. To přináší možnost sdílet společné části kódu, využívat stejná běhová prostředí a udržovat jednotná pravidla stylu psaní. Pro tvorbu uživatelského rozhraní je použita knihovna React, která nabízí dle [32] tyto výhody:

- Syntaxe podobná jazyku HTML
- Vysoká rychlost vykreslování
- Podpora funkcionálního programování

Tvorba uživatelského rozhraní probíhá na bázi skládání komponent, kde každá z nich je zodpovědná za svůj vnitřní stav. Knihovna se pak stará o to, aby došlo k překreslení pouze těch komponent, u kterých ke změně stavu dojde, což vede k efektivnímu a rychlému vykreslování. Kromě webového prohlížeče podporuje knihovna vykreslování na straně serveru či v mobilní aplikaci, k tomu jsou však zapotřebí další rozšíření. [33] V ukázce 3.2 převzaté z webové stránky knihovny [33] je definována komponenta `HelloMessage`, která pomocí metody `render()` zobrazí pozdrav „Hello Taylor“, přičemž jméno bylo předáno pomocí parametru `name`.

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  document.getElementById('hello-example')
);
```

Ukázka kódu 3.2: Jednoduchá React komponenta

3.2 Vrstvy aplikace

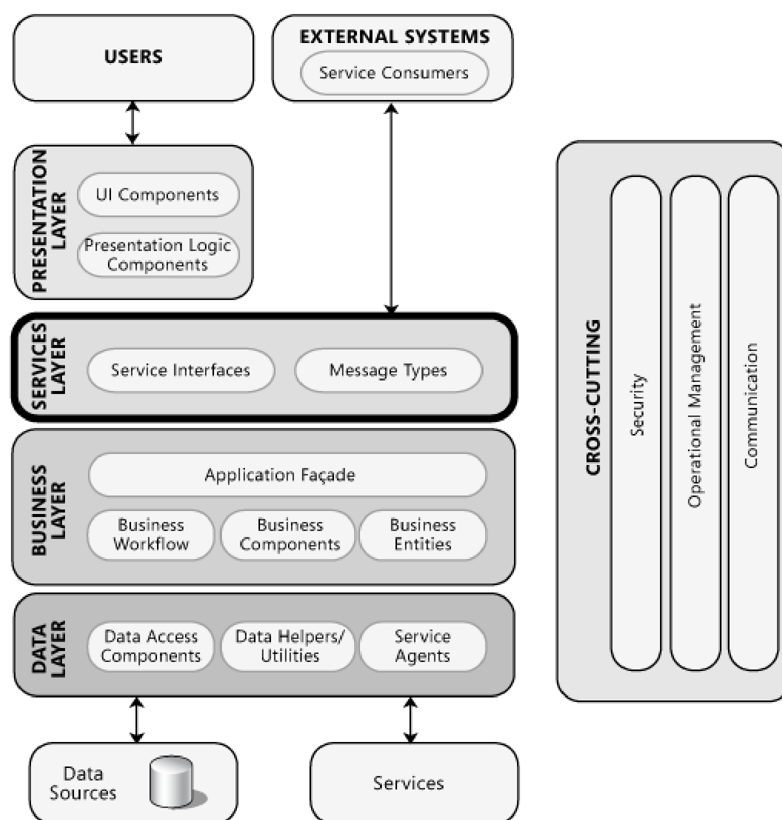
Aplikace je rozdělena do logických celků zvaných vrstvy, kdy každá vrstva má svoji roli a obsahuje komponenty zodpovědné za určité úkoly. To umožňuje snadnou správu, škálovatelnost a znovupoužitelnost jednotlivých částí aplikace. Vrstvy je pak možné oddělit nejen logicky, ale i fyzicky, díky čemuž lze volit rozdílné technologie a měnit dle potřeby návrhová rozhodnutí. [3] Na obrázku 3.2 je zobrazeno rozložení vrstev, jejichž definice je následující:

Prezentační vrstva je zodpovědná za interakci uživatele se systémem. Obsahuje komponenty uživatelského rozhraní a komponenty ovlivňující prezentační logiku.

Servisní vrstva nabízí funkcionality business vrstvy dalším systémům či prezentační vrstvě. V některých případech může chybět a prezentační vrstva pak komunikuje s business vrstvou, nebo přistupuje napřímo k datové vrstvě. Využití servisní vrstvy nabízí podporu různých typů klientů, jako je například webová a mobilní aplikace.

Business vrstva zahrnuje hlavní funkcionality systému a zapouzdřuje business logiku. Obsahuje komponenty zodpovědné za procesy a transakce, pomocné nástroje a komponenty entit umožňující práci s daty.

Datová vrstva poskytuje business vrstvě přístup k datům uloženým v databázi, souborům nahraným v úložišti nebo informacím získaným od externích služeb. [3]



Obrázek 3.2: Rozdělení aplikace do vrstev [3]

Během návrhu je potřeba určit pravidla pro interakce mezi vrstvami tak, aby vzniklo co nejméně závislostí a cyklických referencí. Striktní interakce dovoluje přímou komunikaci pouze se spodní vrstvou, čímž dojde v budoucnu k minimalizaci dopadu na další vrstvy v případě úprav, či rozšíření o nové fukcionality. [3]

Rovněž je nutná definice rozhraní mezi vrstvami. Mezi prezentační a servisní vrtsvou je zvolena komunikace založená na zprávách, která využívá známých standardů k zapouzdření detailů o interakci. [3] V této aplikaci je využit dotazovací jazyk GraphQL nabízející definici schémat, validaci dat a zpracování chybových hlášení. Pro zbylé vrstvy jsou vytvořena rozrhaní, přes která mohou komponenty volat potřebné metody.

3.2.1 Datová vrstva

Datová vrstva obsahuje komponenty pro přístup k rozhraním datových úložišť a externích služeb. Zahrnuje logiku pro operace s daty, jejich mapování na objekty systému a implementuje komunikační protokoly pro interakci s externími službami. [3] V naší aplikaci má tato vrstva zodpovědnost za následující úkoly:

- Přístup k záznamům uloženým v PostgreSQL databázi a operace s nimi
- Volání metod Rasa serveru pomocí API rozhraní
- Práce se soubory nacházejícími se v S3 úložišti
- Odběr událostí od zprostředkovatele zpráv RabbitMQ

3.2.1.1 PostgreSQL databáze

Pro ukládání dat je vybrána open-source objektově orientovaná databáze PostgreSQL. Ta používá a rozšiřuje SQL jazyk v kombinaci s dalšími funkcionalitami. Díky vysokému výkonu, spolehlivosti, integritě dat a kvalitní architektuře patří mezi nejlepší dostupné databáze. Je možné ji spustit na všech známých operačních systémech a má za sebou již třicet let vývoje. [34]

Aby nebylo nutné pracovat se záznamy přímo přes SQL jazyk, je použita knihovna Sequelize s ORM funkcionalitou, která mapuje JavaScript objekty na tabulky v databázi, a vytváří tak rozhraní pro operace s daty. [35] Připojení k databázi lze provést vytvořením objektu třídy `Sequelize`, do jehož konstruktoru se zadá URL adresa databázového serveru. [36] Pomocí této instance pak lze definovat jednotlivé databázové modely. Ty mezi sebou mohou vytvářet relace, ale aby s nimi bylo možné pracovat, je třeba zadefinovat vztahy využitím metod `hasOne()`, `hasMany()` či `belongsTo()`. [37] V ukázce 3.3 má model uživatele tři atributy, patří právě do jedné uživatelské skupiny a každá uživatelská skupina může obsahovat několik uživatelů.

```
const User = db.define('User', {
  userGroupId: Sequelize.INTEGER,
  name: Sequelize.STRING,
  email: Sequelize.STRING
});

User.belongsTo(UserGroup);
UserGroup.hasMany(User);
```

Ukázka kódu 3.3: Definice objektu uživatele v knihovně Sequelize

Pro vytvoření záznamu slouží metoda `create()`, pro úpravu dat `update()` a pro odstranění `delete()`. Pro vyhledávání záznamů lze použít metodu `findAll()` či `findOne()` spolu s parametrem `where`, který omezí výsledky dle zadaných kritérií. [36] Jejich použití je vidět v ukázce 3.4.

Relační databázový model na obrázcích D.1, D.2, D.3 a D.4 dostupných v příloze práce popisuje kolekci tabulek použitých v našem systému spolu s jejich atributy a vzájemnými vazbami.

```
const newUser = await User.create({
  userGroupId: 2,
  name: 'John Snow',
  email: 'john.snow@example.com'
});

const user = await User.findOne({
  where: {
    email: 'john.snow@example.com'
  }
});
```

Ukázka kódu 3.4: Metody pro operace se záznamy v knihovně Sequelize

3.2.1.2 Rasa server

Komunikace s trénovacím a produkčním Rasa serverem je zajištěna pomocí API rozhraní. Díky tomu je možné spouštět proces trénování, měnit aktuálně načtený model za jiný nebo provádět evaluaci trénovacích dat. Mezi důležité endpointy uvedené v dokumentaci [38] patří:

GET /status informace o stavu serveru a podrobnosti o aktuálně načteném modelu chatbota

POST /model/train spuštění procesu trénování s využitím trénovacích dat zaslaných v těle požadavku

PUT /model nahrazení aktuálně načteného model za jiný, jehož umístění je specifikováno v těle požadavku

POST /model/test/intents provedení evaluace trénovacích dat zaslaných v těle požadavku proti modelu specifikovaném v parametru URL adresy

Dle [39] se nedoporučuje veřejně vystavovat API rozhraní Rasa serveru, ale přistupovat k němu pouze přímo z backendu aplikace přes privátní připojení. I přes to lze však komunikaci zabezpečit pomocí autorizačního tokenu, nebo JWT tokenu poskytnutého v hlavičce požadavku.

3.2.1.3 S3 úložiště

Natrénované modely mohou dosahovat velikosti až jednotek gigabajtů. Je třeba zvolit vhodný typ dedikovaného úložště s ohledem na cenu, dostupnost a škálovatelnost. Proto je použito úložiště typu S3, což je jedna ze služeb Amazon Web Services. [40] Pro účely vývoje je využita alternativní služba

s podporou S3 API MinIO, která nabízí webové rozhraní pro snadnou správu dat. Tuto službu lze spustit jako serverovou aplikaci například pomocí virtualizačního nástroje Docker. [41] Důležité pojmy při práci s S3 úložištěm jsou:

object základní jednotka informace skládající se z dat a metadat

bucket místo, kam jsou objekty ukládány

key jedinečná cesta k objektu uloženém v určitém bucketu [40]

3.2.2 Business vrstva

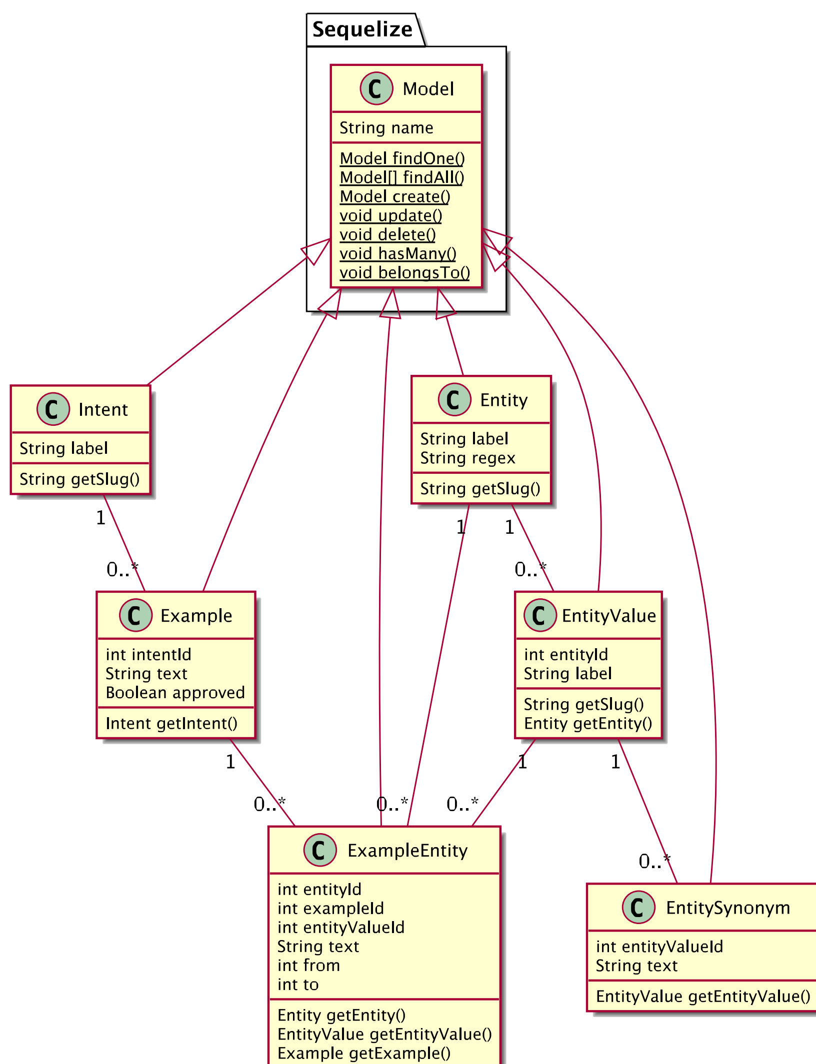
Business vrstva obsahuje aplikační logiku, kam patří získávání dat, jejich validace, následné operace či transformace a to vše při dodržení předem nastavených pravidel. Obsahuje komponenty business entit a komponenty určené k provádění business procesů. [42]

3.2.2.1 Business entity

Business entity představují objekty reálného světa [42] a v systému jsou reprezentovány databázovými modely. Třídy těchto modelů dědí od rodičovské třídy `Model`, která je zprostředkována ORM knihovnou Sequelize a která na tyto modely mapuje tabulky databáze. Ta poskytuje mimo jiné statické metody pro vytváření relací, čtení dat a jejich následnou modifikaci.

V systému je možné spravovat vzorové zprávy, které se používají v procesu trénování. Každé z nich je nutné přiřadit téma (třída `Intent`) a klíčová slova (třída `Entity`) vyskytující se uvnitř věty. Umístění klíčového slova je řešeno třídou `ExampleEntity` držící informaci o začátku a konci výskytu spolu s odkazem na entitu či její konkrétní hodnotu reprezentovanou třídou `EntityValue`. Každá hodnota může obsahovat několik synonym označující tutéž věc, k čemuž slouží třída `EntitySynonym`. Tyto vztahy jsou znázorněny na obrázku 3.3.

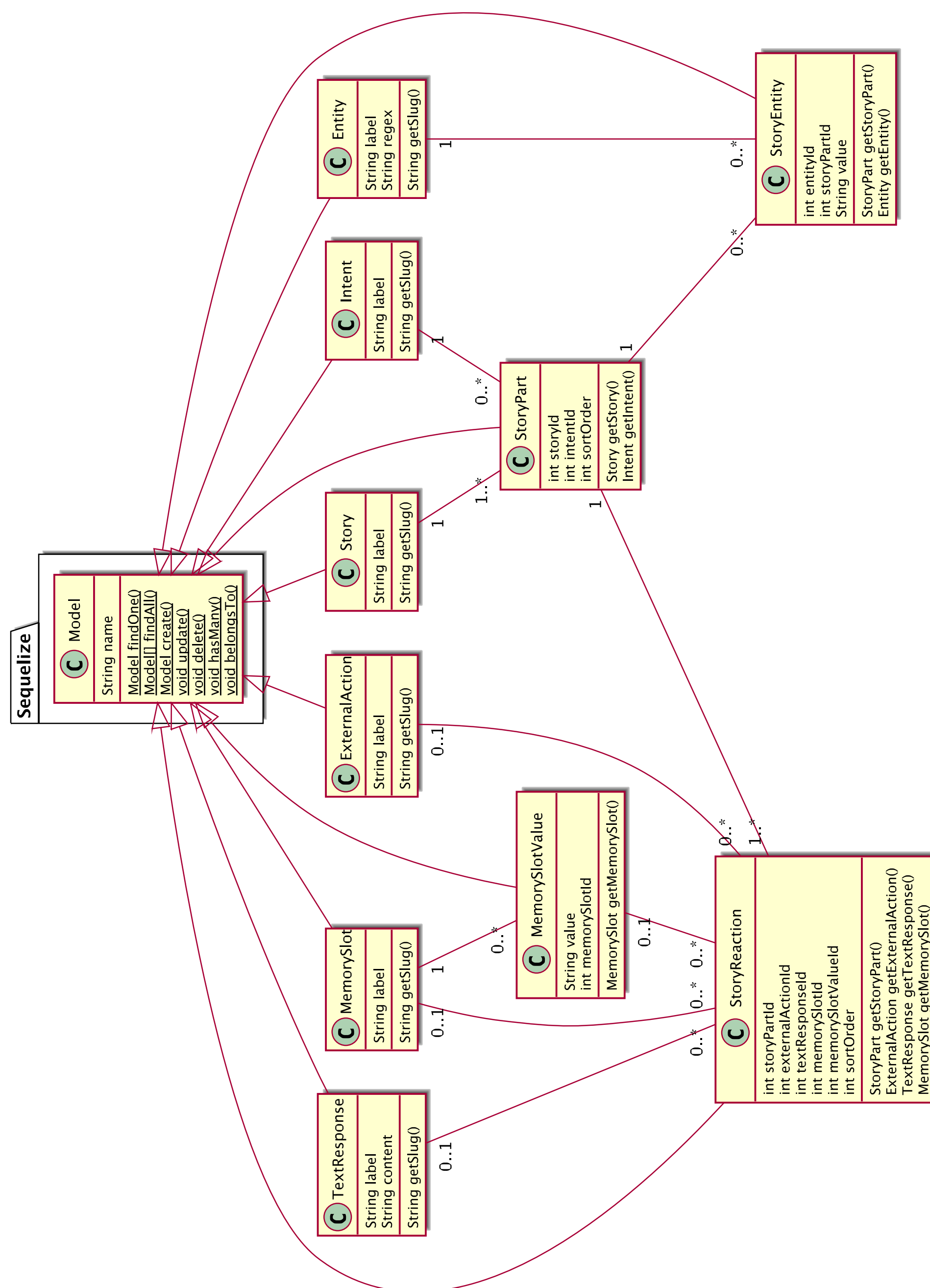
Dále je možné spravovat vzorové příběhy, na základě kterých je chatbot schopen rozhodnout ve výběru odpovědi na otázku uživatele. Vzorový příběh je reprezentován třídou `Story`, jehož základním stavebním prvkem je `StoryPart`. Tato třída představuje právě jednu fázi dialogu sestávající ze zprávy a reakce na ni. Zpráva, její téma a případná klíčová slova jsou řešena vazbou na třídu `Intent` a `StoryEntity`. Reakcí na zprávu uživatele může být i několik a vykonávájí se v určitém pořadí. To zajišťuje atribut `sortOrder` třídy `StoryReaction`. Samotnou reakcí je pak buď odpověď v textové podobě (`TextReaction`), uložení hodnoty do paměťového slotu (`MemorySlot`) nebo zavolání externí akce (`ExternalAction`). Konkrétní hodnota paměťového slotu je reprezentována třídou `MemorySlotValue`. Tyto vztahy jsou znázorněny na obrázku 3.4.



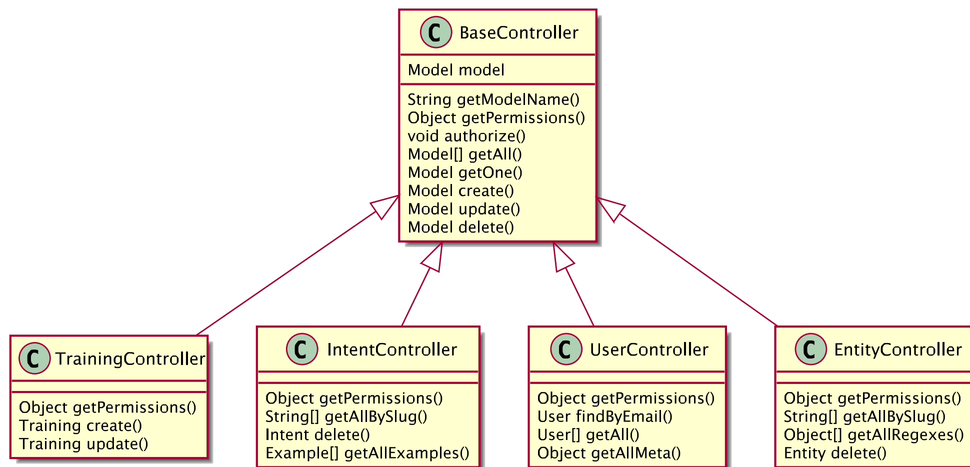
Obrázek 3.3: Třídy databázových modelů týkajících se vzorových zpráv

3.2.2.2 Business procesy

Po přijetí požadavku ze strany klienta a jeho předání business vrstvě je potřeba provést sérii kroků, které povedou k vykonání celého business procesu. [42] Mezi tyto kroky patří například autentizace a autorizace uživatele, validace vstupních dat, dotaz na databázi či spuštění dalších podprocesů. Každá business entita má proto svoji speciální třídu označovanou kontrolér, která obsahuje business logiku a pravidla omezující uživatele provádět určité operace na základě jejich role. Všechny kontroléry mají společnou rodičovskou třídu `BaseController`, od které dědí metody pro CRUD operace.



Obrázek 3.4: Třídy databázových modelů týkajících se vzorových příběhů



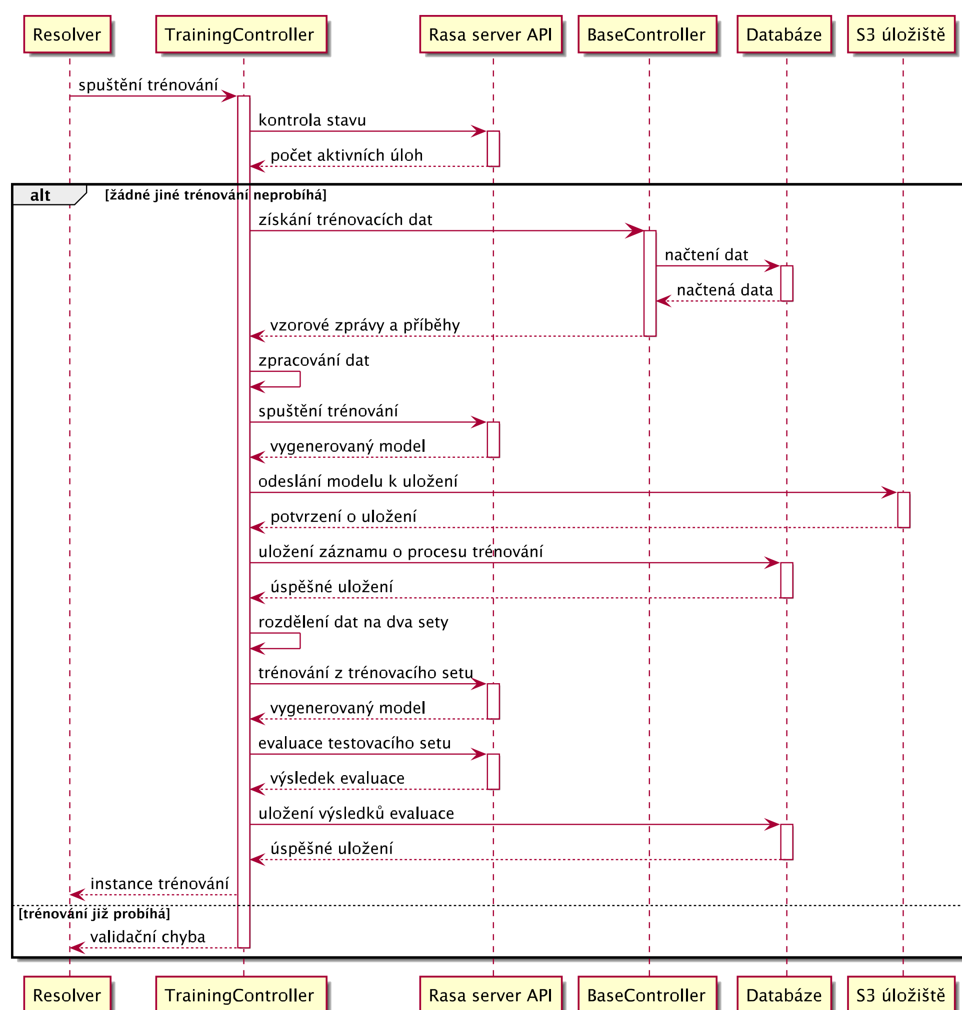
Obrázek 3.5: Třídy kontrolérů obsahujících business logiku

Některé z nich, jako `TrainingController`, přepisují metody rodičovské třídy a přidávají k nim vlastní business logiku. Jiné, jako `UserController`, zase poskytují pomocné metody, které usnadňují vyhledávání a práci s daty. Na závěr pak všechny třídy implementují metodu `getPermissions()`, která na základě instance aktuálního uživatele určuje, zda má pro danou CRUD operaci oprávnění. Rozhraní důležitých tříd je znázorněno na obrázku 3.5.

Nejdůležitějším procesem je vygenerování modelu chatbota s využitím trénovacích dat. Na začátku vytvoří uživatel nový záznam entity trénování, čímž dojde k zavolání metody `create()` třídy `TrainingController` a ke kontrole, zda již v současné chvíli nějaké trénování neprobíhá. Pokud ano, je akce ukončena s chybou, jinak jsou postupně získána všechna trénovací data a převedena do formátu Markdown a Yaml, aby mohla být odeslána na trénovací Rasa server. Po dokončení procesu je vrácen vygenerovaný model, který je následně spolu s trénovacími daty odeslán do S3 úložiště. Na závěr je provedeno rozdělení dat na trénovací a testovací set v poměru 8:2, kdy první je použit pro vygenerování nového modelu a druhý k evaluaci intentů a entit vůči tomuto modelu. Průběh je znázorněn na obrázku 3.6.

3.2.3 Servisní vrstva

Servisní vrstva poskytuje prezentační vrstvě přístup k funkcionalitám business vrstvy skrze svoje rozhraní, což umožňuje přístup různých typů klientů najednou a vyšší úroveň kompozice systému. [3] Jádrem servisní vrstvy je framework Express, který se stará o poskytnutí základního API rozhraní. Ačkoliv je framework Express poměrně malá knihovna nabízející pouze základní funkcionality, existuje velké množství rozšíření ve formě `middlewares`, díky kterým



Obrázek 3.6: Sekvenční diagram procesu trénování

lze mimo jiné řešit například deserializaci nebo kešování dat. [31] Mezi hlavní funkcionality frameworku patří:

- Zpracování požadavků na základě HTTP metody a URL adresy
- Využití integrovaných nástrojů pro vykreslování webových stránek
- Volání přídavných funkcí ovlivňujících proces zpracování požadavku [31]

Každý příchozí požadavek je v našem systému postupně předáván sekvencí zmíněných `middlewares`, jimiž jsou `bodyParser.json()` pro převedení těla požadavku do formátu JSON, `passport.initialize()` pro inicializaci autentizační strategie nebo `helmet()` pro aplikaci dvanácti důležitých pravidel,

která nastavují HTTP hlavičky odpovědi z důvodu předejití známým útokům na serverovou aplikaci. [43]

Pro komunikaci mezi klientem a API je jako alternativa klasického REST API vybrán dotazovací jazyk GraphQL. Ten na místo několika URL cest a HTTP metod využívá pouze jednu, kam jsou zasílány buď dotazy pro získání dat, nebo mutace pro jejich úpravu. Výhodou této technologie je jasně definovaná podoba objektů, o které lze žádat, což umožňuje získat pouze ta data, která jsou skutečně potřeba, a šetřit tak provoz sítě. Další výhodou je uniformní podoba odpovědí a to nejen samotných dat, ale i chybových zpráv. Na závěr je to snadné verzování API, kdy změna ve struktuře objektu neovlivní funkcionalitu stávajících dotazů. [44] V ukázce 3.5 převzaté z dokumentace jazyka [45] je příklad definice objektu uživatele spolu s definicí dotazu pro získání aktuálně přihlášeného uživatele.

```
type Query {
  me: User
}

type User {
  id: ID
  name: String
}
```

Ukázka kódu 3.5: Ukázka definice GraphQL schématu

Dále je třeba ke každému dotazu vytvořit příslušný resolver neboli logiku, kterou je nutné pro získání dat vykonat. V ukázce 3.6 vrací první metoda objekt přihlášeného uživatele a druhá uživatelovo jméno. Odesláním HTTP požadavku na API s obsahem z ukázky 3.7 v těle vede k získání jména přihlášeného uživatele a vrácení odpovědi z ukázky 3.8. [45]

```
function Query_me(request) {
  return request.auth.user;
}

function User_name(user) {
  return user.getName();
}
```

Ukázka kódu 3.6: Implementace GraphQL resolverů

```
{
  me {
    name
  }
}
```

Ukázka kódu 3.7: GraphQL dotaz pro získání jména uživatele

```
{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

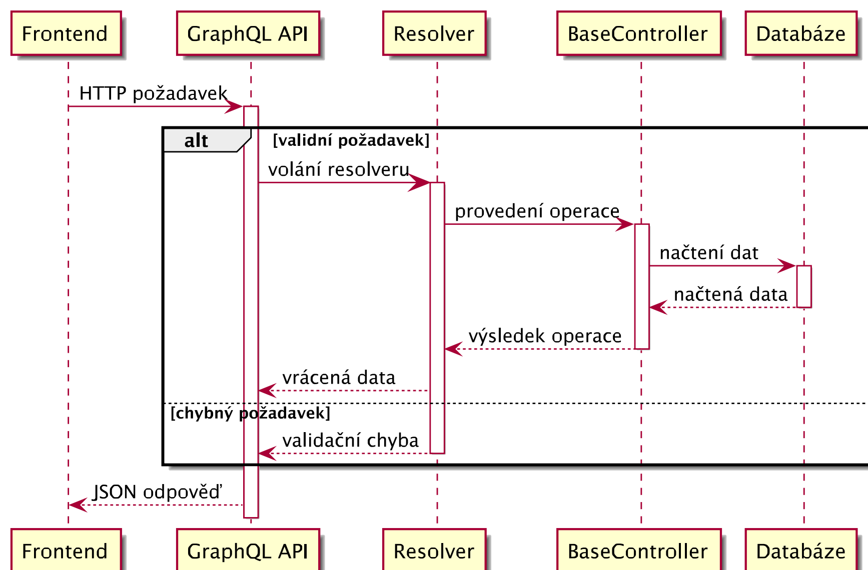
Ukázka kódu 3.8: Odpověd získaná na základě GraphQL dotazu

Pro přehlednější správu resolverů a snadnou integraci s Express frameworkem je využita knihovna Apollo Server. Ta nabízí produkční řešení pro implementaci jazyka GraphQL a mimo jiné poskytuje nástroje pro kešování dat, autentizaci, monitoring a testování. [46] Instance třídy ApolloServer se vkládá jako middleware, čímž dojde k zaregistrování HTTP metod `POST` a `GET` na adrese `/graphql`, kam je následně možné požadavky zasílat.

V našem systému jsou pro všechny business entity vydefinovány příslušné typy GraphQL objektů a resolvery poskytující CRUD operace. Například pro typ `Intent` jsou vytvořeny dotazy s názvem `allIntents` (vrátí všechny intenty) a `Intent` (vrátí intent dle ID) či mutace s názvem `createIntent` (vytvoří intent), `updateIntent` (aktualizuje inetent) a `deleteIntent` (odstraní intent). Průběh zpracování požadavku je znázorněn na obrázku 3.7.

3.2.4 Prezentační vrstva

Prezentační vrstva je zodpovědná za interakci s uživatelem. Obsahuje komponenty pro vykreslení uživatelského rozhraní, kam uživatel zadává vstup do systému, nebo kde zobrazuje výsledná data, a komponenty prezentační logiky, jejichž úlohou je řídit chování a strukturu aplikace. [47] V našem systému je prezentační vrstva implementačně oddělena od ostatních vrstev a je vyvíjena v samostatném repozitáři. Oddělením vrstvy je umožněna podpora různých typů klientů, kterými mohou být současně webová, mobilní i desktopová aplikace. Prezentační vrstva komunikuje s business vrstvou přes API rozhraní servisní vrstvy, konkrétně s využitím dotazovacího jazyka GraphQL.



Obrázek 3.7: Sekvenční diagram zpracování GraphQL požadavku

Protože je náš systém určený především pro CRUD operace nad velkým množstvím různých typů objektů, je pro rychlejší tvorbu uživatelského rozhraní použit framework React Admin, který staví na webovém frameworku React a nabízí hotové jádro aplikace s velkým množstvím komponent určených pro zobrazování a správu dat. Jeho součástí je i připravené rozhraní pro komunikaci se servisní vrstvou a prezentační logika podporující vykreslování na základě práv uživatele či různých jazykových mutací. [48] Všechny komponenty využívají pro svůj vzhled Material Design, což je open-source systém doporučení a nástrojů určených pro tvorbu moderního uživatelského rozhraní, za kterým stojí společnost Google. [49]

Základním prvkem frameworku je komponenta `Admin`, která ve svém parametru `dataProvider` přijímá implementaci rozhraní pro komunikaci se servisní vrstvou a ve svém těle pole komponent typu `Resource` označujících business entity, jejichž data budou v aplikaci spravována. [50] Kdykoliv je potřeba získat nebo upravit nějaký záznam, použije framework Data Provider, který má funkci adaptéru, a který převádí dotazy do podoby HTTP požadavků. Každý Data Provider musí implementovat skupinu metod zodpovědných za provádění CRUD operací, mezi které patří například `getList()`, `getOne()`, `create()`, `update()` nebo `delete()`. [51] K využití dotazovacího jazyka GraphQL pro komunikaci s API je vybrán existující Data Provider `ra-data-graphql-simple`, který staví na knihovně Apollo Client a který je kompatibilní s knihovnou Apollo Server používanou v servisní vrstvě. [52]

V ukázce 3.9 převzaté z dokumentace frameworku [50] je příklad jednoduché aplikace, která pomocí komponenty `ListGuesser` zobrazí v podobě ta-

bulky výpis uživatelů, kde každý sloupec odpovídá jednomu atributu objektu uživatele. Po načtení webové stránky zavolá framework metodu `getList()`, na základě které vyšle Data Provider na API dotaz s názvem `allUsers`, načež se vrátí odpověď ve formátu JSON obsahující pole všech uživatelů.

```
import { Admin, Resource, ListGuesser } from 'react-admin';

const App = () => (
  <Admin dataProvider={dataProvider}>
    <Resource name="User" list={ListGuesser} />
  </Admin>
);
```

Ukázka kódu 3.9: Jednoduchá aplikace frameworku React Admin

3.3 Zabezpečení

Zabezpečení je dle [53] řazeno mezi průřezové téma týkající se všech vrstev aplikace. Mezi důležité pojmy v oblasti zabezpečení webové aplikace patří:

autentizace proces ověření totožnosti na základě přihlašovacích údajů, který umožňuje zabezpečenou kontrolu identity a přístupu

autorizace proces udělení oprávnění pro vykonání určité činnosti

autorizační server služba zodpovědná za přidělení přístupového tokenu klientovi na základě úspěšné autentizace

přístupový token typ bezpečnostního tokenu udělovány autorizačním serverem a využitý pro získání chráněného obsahu [54]

JWT otevřený standard popisující zabezpečený způsob přenosu informace mezi stranami ve formě JSON objektu [55]

3.3.1 Autentizace

Pro autentizaci uživatele je na základě požadavků použit centralizovaný poskytovatel identity Azure Active Directory. Díky němu může uživatel pomocí jediného přihlášení získat přístup do všech aplikací, které sdílejí stejný centralizovaný adresář. Tato vlastnost se též označuje jako SSO. Pro usnadnění implementace přihlašovacího procesu do různých typů aplikací byla vytvořena platforma Microsoft Identity, která nabízí podporu otevřených protokolů jako OAuth 2.0 nebo OpenID Connect. [56]

Aby bylo možné platformu Microsoft Identity využívat, je třeba svoji aplikaci zaregistrovat na portálu Microsoft Azure. Důležitými parametry při registraci jsou `Application ID` (též `Client ID`), pomocí kterého je aplikace identifikována a `Redirect URI` označující adresu, kam je nutné uživatele po úspěšném přihlášení přesměrovat. [57].

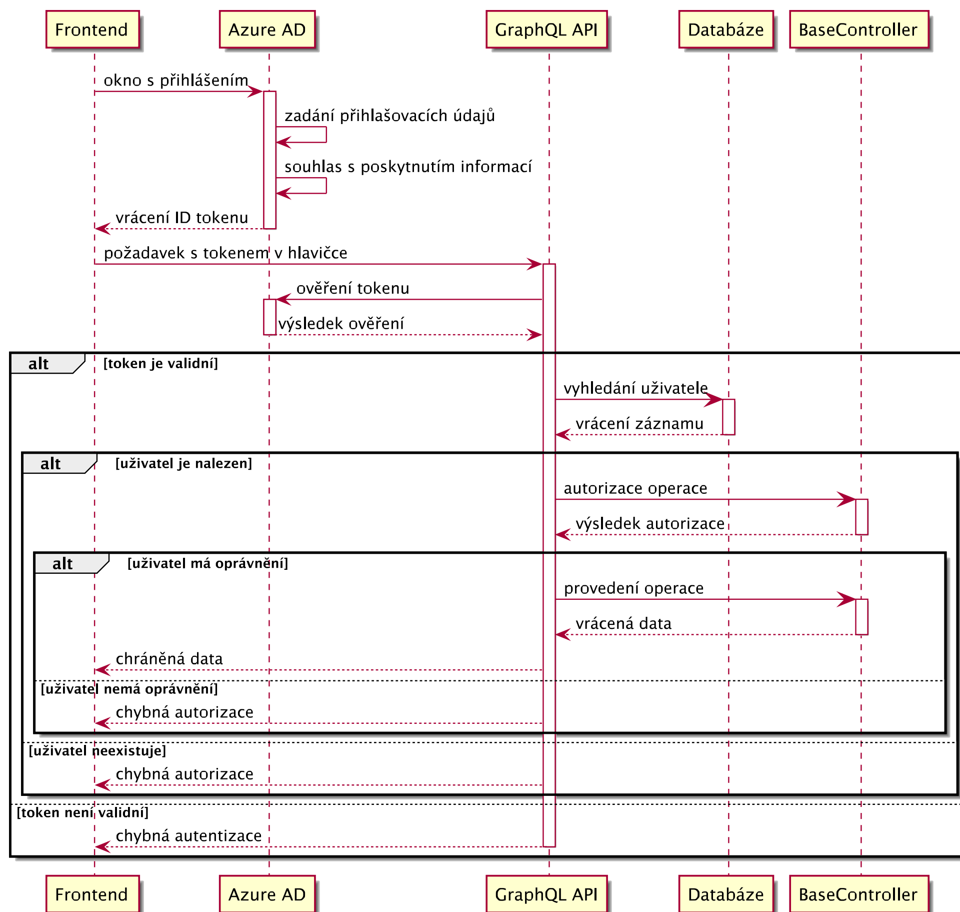
Systém využívá optimalizaci protokolu OAuth 2.0 zvanou Implicit Grant, která je výhodná pro veřejné webové klienty běžící přímo v prohlížeči. Tato cesta umožňuje získat přístupový ID token z platformy Microsoft Identity bez nutnosti komunikace s API serverem, neboť je celý proces přihlášení a správy relace řízen přímo z webového klienta. Důvodem, proč je tento postup vybrán je, že autorizační servery a poskytovatelé identity nepodporují CORS požadavky. Navíc není nutné během přihlášení uživatele přesměrovávat na jinou adresu, což zlepšuje dojem z používání aplikace. [58] Implicit Grant však přináší i řadu omezení, proto by neměl být používán, pokud je webová stránka vykreslována na straně serveru. V takovém případě je lepší postupovat ve dvou krocích, kdy je nejdříve získán autorizační kód, pomocí kterého je následně odeslán požadavek na přístupový token. [57]

Proces přihlášení probíhá tak, že aplikace nejdříve ověří, zda je uživatel autentizován. Pokud není, otevře se nové okno s adresou Azure AD, kde je požádán o zadání přístupových údajů a poskytnutí souhlasu ke zpřístupnění informací, které aplikace pro fungování potřebuje. Po úspěšném přihlášení je uživatel přesměrován zpět spolu s JWT tokenem vloženým v URL adrese. Ten obsahuje ve svém těle základní informace o uživateli, jako je jeho jméno a příjmení nebo preferované přihlašovací jméno. Tento token je pak přikládán v hlavičce `Authorization` každého požadavku zaslaného na API server, který si před vrácením chráněných dat ověří jeho platnost a identitu uživatele. Tento proces je zachycen na obrázku 3.8.

Pro implementaci celého procesu na straně webového klienta je použita knihovna Microsoft Authentication Library for JavaScript (MSAL.js). Nejprve je nutné vytvořit instanci třídy `UserAgentApplication`, která ve svém kostruktoru vyžaduje `Client ID` a adresu Azure AD dostupnou z administračního rozhraní aplikace na portálu Microsoft Azure. Pomocí této instace pak lze volat metody pro spuštění procesu přihlášení `loginPopup()`, získání informací o aktuálním uživateli `getAccount()` nebo `logout()` pro následné odhlášení. Důležitá data a tokeny ukládá knihovna v úložišti prohlížeče. [59]

Framework React Admin umožňuje řídit autentizaci pomocí skupiny metod zabalených v objektu zvaném Auth Provider, kterými jsou `login()` pro přihlášení uživatele, `logout()` pro jeho odhlášení, `checkError()` pro kontrolu chyby vyvolané požadavkem na API (typicky kód 401), `checkAuth()` pro vrácení stavu, zda je uživatel přihlášen a `getPermissions()` pro získání úrovně oprávnění uživatele, což je využito v procesu autorizace. [60]

Na straně backendu je autentizace prováděna pomocí knihovny Passport, která je aplikována jako jeden z `middlewares` frameworku Express. Tato knihovna umožňuje výběr z více než 500 autentizačních strategií, jako je



Obrázek 3.8: Proces přihlášení uživatele s využítím Implicit Grant

přihlášení pomocí Google, Facebook či LinkedIn účtu. [61] Pro naše účely je však použita knihovna `passport-azure-ad` nabízející Bearer Token strategii, která dokáže získat přístupový token z hlavičky či těla HTTP požadavku, ověřit jeho platnost a předat informace uložené v tokenu k dalšímu zpracování. Strategii lze použít vytvořením instance třídy `BearerStrategy`, jenž ve svém konstruktoru přijímá `Client ID`, URL adresu manifestu s podrobnostmi o konfiguraci aplikace a metodu obsahující logiku pro dokončení autentizace uživatele. [62]

V praxi proběhne pokus o dohledání uživatele v databázi na základě e-mailu získaného z těla tokenu. Pokud není uživatel nalezen nebo je token neplatný, je vyhozena výjimka a vrácena odpověď se stavovým kódem 401. V opačném případě jsou informace o uživateli zapsány mezi ostatní parametry objektu požadavku a předány dále. [61]

3.3.2 Autorizace

Celkem jsou rozlišovány tři role uživatelů, kterými jsou dle stupně oprávnění administrátor, vývojář a anotátor. Administátor má přístup ke všem funkcionalitám systému včetně správy uživatelů. Vývojář má na starosti návrh a vývoj chatbota, proto má k jeho správě kompletní přístup. Nejnižší úrovní je anotátor, jehož úkolem je zlepšovat schopnost chatbota klasifikovat vstupní data vytvářením nových vzorových zpráv a schvalováním zpráv importovaných z reálných konverzací.

V prezentační vrstvě je potřeba, na základě konkrétní role, omezit přístup k zobrazení a úpravě některých objektů. To je zajištěno dynamickým vykreslováním komponenty `Resource` uvnitř komponenty `Admin` frameworku React Admin. O získání úrovně oprávnění se stará metoda `getPermissions()`, o níž byla zmínka v rámci objektu zvaném Auth Provider. [63] Ta provede po načtění aplikace dotaz na API a vrátí informace o aktuálně přihlášeném uživateli. Příklad omezení přístupu je nastíněn v ukázce 3.10.

```
<Admin authProvider={authProvider}>
  {permissions => [
    <Resource name="customers" />,
    permissions === ADMIN && <Resource name="categories" />
  ]}
</Admin>
```

Ukázka kódu 3.10: Vykreslení komponenty dle role uživatele

Oprávnění na úrovni prezentační vrstvy musí reflektovat pravidla nastavená na úrovni business vrstvy, protože nestačí pouze omezit, jaké komponenty se mají vykreslit, ale je třeba chránit provádění operací skrze API rozhraní. Z toho důvodu každý kontrolér implementuje metodu `getPermissions()`, která pro konkrétní business entitu vrací seznam povolených CRUD operací, které může aktuálně přihlášený uživatel provádět. Tuto metodu pak využívá metoda `authorize()` třídy `BaseController`, která aplikuje získané hodnoty na výchozí nastavení oprávnění a v případě, že se uživatel dožaduje operace, kterou provádět nesmí, je vyhozena výjimka.

Kapitola 4

Implementace

Tato kapitola se věnuje implementaci backendové a frontendové části systému. Na začátku je představen správce balíčků NPM a pomocné nástroje použité při vývoji, dále je popsána struktura obou částí aplikace spolu s ukázkami zdrojového kódu. V sekci o frontendu jsou navíc přiloženy ukázky obrazovek a komponent z uživatelského rozhraní.

4.1 Nástroje pro vývoj

Jak již bylo v předešlé kapitole popsáno, backend i frontend systému je programován v jazyce JavaScript. Pro udržení konzistence a přehlednosti kódu napříč repozitáři je nutné zachovat stejný styl a pravidla. K tomu slouží nástroje ESlint pro vynucení předdefinovaného stylu jazyka a Prettier pro jednotné formátování kódu. Konfigurace obou nástrojů se nachází v kořenovém adresáři projektu v souborech `.eslintrc` a `.prettierrc`. [64, 65]

Dalším důležitým nástrojem je správce balíčků NPM, díky kterému mohou obě vývojová prostředí aplikace instalovat a využívat již existující knihovny. Soubor `package.json` obsahuje názvy a verze požadovaných knihoven spolu se základními informacemi o projektu samotném. Dále může obsahovat definice skriptů, jako je například spuštění aplikace, spuštění testů či kontrola kvality kódu. Zavoláním příkazu `npm install` jsou staženy všechny požadované závislosti a následně jsou umístěny do složky `node_modules`. [66]

4.2 Backend systému

Repozitář s backendovou částí systému obsahuje kromě zdrojových kódů také inicializační skripty pro sestavení struktury databáze, ukázkový projekt chatbota se souborem `docker-compose.yaml` pro spuštění vyžadovaných služeb infrastruktury, jednotkové či integrační testy a konfigurační soubory vývojového prostředí. Adresářová struktura složky se zdrojovými kódy je následující:

```
/src
├── auth ........................ komponenty pro autentizaci uživatele
├── config .................................... konfigurační soubory
├── controllers ............................. třídy business procesů
├── errors .................................................. třídy chyb
├── graphql ......................... schéma a resolvery GraphQL API
├── jobs .......................................... asynchronní úlohy
├── models .......................... databázové modely business entit
├── services ......................... služby datové a servisní vrstvy
├── app.js ............................. aplikace frameworku Express
└── server.js ..................... server běhového prostředí Node.js
```

Databázové modely reprezentující business entity jsou umístěny ve složce `models`. Pro každou entitu je uveden název odpovídající názvu tabulky v databázi a atributy odpovídající jejím sloupcům. Atributy jako ID nebo časové razítko nejsou potřeba zadávat, neboť jsou doplněny migračním scriptem při inicializaci struktury databáze. V ukázce 4.1 je definice databázového modelu entity trénování, která udržuje informaci o aktuálním stavu procesu, název souboru s modelem chatbota, poznámky k vydání a označení, zda je model aplikován na produkční Rasa server.

```
const Training = db.define('Training', {
  status: sequelize.INTEGER,
  modelFilename: sequelize.STRING,
  releaseNotes: sequelize.STRING,
  modelDeployed: sequelize.BOOLEAN
});
```

Ukázka kódu 4.1: Definice databázového modelu entity trénování

Business procesy jsou reprezentovány třídami označovanými jako kontroléry a nacházejícími se ve složce `controllers`. Díky společné rodičovské třídě `BaseControler` nabízejí metody pro všechny základní CRUD operace. Do konstruktoru je při vytváření potomka potřeba zadat databázový model business entity, přes který kontrolér provádí změny v databázi. Ukázka 4.2 obsahuje metodu pro vytvoření záznamů `create()` s podporou manuálního řazení, kdy jsou nové prvky vkládány na poslední pozici.

Každá business entita má vlastní kontrolér, který implementuje specifické metody nebo přepisuje zděděné, jako je tomu v ukázce 4.3, kde se metoda `create()` kromě vytvoření záznamu stará o zajištění dostupnosti trénovacího Rasa serveru, získání trénovacích dat z ostatních kontrolérů a spuštění asynchronního procesu trénování.

```
class BaseController {
  constructor(model, options = {}) {
    this.model = model;
    this.options = options;
  }

  async create(data) {
    if (this.options.sortable) {
      const { count } = await this.getAllMeta();
      data.sortOrder = count;
    }

    return this.model.create(data);
  }
}
```

Ukázka kódu 4.2: Metoda pro vytvoření záznamu v databázi

Důležitou skupinou tříd jsou chyby nacházející se ve složce `errors`. Ty jsou vyhazovány v určitých situacích na základě porušení validačních pravidel (třída `UserInputError`), neautorizovaných pokusů (třída `ForbiddenError` či `AuthenticationError`) nebo při jakýchkoliv neočekávaných problémech (třída `ServerError`). V ukázce 4.4 dědí `UserInputError` z třídy poskytnuté knihovnou Apollo Server, která je při zachycení převedena na unifikovaný formát GraphQL rozhraní a odeslána v odpovědi zpět klientovi.

Některé procesy je nutné spouštět asynchronně, bez čekání na jejich dokončení. Často se skládají ze série navazujících úloh, kde výsledek předešlé akce je vstupem akce následující. Tyto úlohy se nacházejí ve složce `jobs` a jsou spouštěny buď z kontrolérů nebo na základě vnější události. Jedním z nejdůležitějších úkolů je vygenerování modelu chatbota, při kterém jsou nejprve trénovací data převedena do formátu Markdown, a pak odeslána na trénovací Rasa server ke zpracování. Model chatbota je spolu s trénovacími daty nahrán na S3 úložiště, což je nastíněno v ukázce 4.5.

Business procesy jsou zpřístupněny skrze GraphQL API nacházející se v servisní vrstvě. Definice dotazů, mutací a datových typů pro každou z business entit jsou umístěny ve složce `graphql`. Z důvodu vysokého počtu entit je vytvořena pomocná třída `ResolverBuilder`, která sestavuje jednotný formát GraphQL rozhraní a v rámci jednotlivých resolverů mapuje CRUD operace na rozhraní tříd kontrolérů. V ukázce 4.6 metoda `buildCreate()` sestaví mutaci a resolver pro vytvoření nového záznamu, kdy nejdříve provede autorizaci uživatele a následně zavolá metodu `create()` daného kontroléru.

```
class TrainingController extends BaseController {
  constructor() {
    super(Training);
  }

  async create(data) {
    const { activeTrainingJobs } = await getServerStatus();
    if (activeTrainingJobs > 0) {
      throw new UserInputError(TRAINING_ALREADY_RUNNING);
    }

    const trainingData = {
      domain: {
        intents: await intentController.getAllBySlug(),
        entities: await entityController.getAllBySlug()
      },
      nlu: {
        examples: await intentController.getAllExamples()
      },
      stories: await storyController.getFullStories()
    };

    const trainingInstance = await super.create(data);
    startModelTraining(trainingInstance, trainingData);
    return trainingInstance;
  }
}
```

Ukázka kódu 4.3: Metoda pro spuštění procesu trénování

```
class UserInputError extends ApolloUserInputError {
  constructor(errorCode = 'default') {
    super(`Input validation error: ${errorCode}`);
    this.errorCode = errorCode;
  }
}
```

Ukázka kódu 4.4: Třída chyby pro porušení validačních pravidel

```
try {
  const nlu = nluToMarkdown(inputData.nlu);
  const domain = toYaml(inputData.domain);
  const stories = storiesToMarkdown(inputData.stories);

  const model = await trainModel(domain, nlu, stories);
  await uploadModel(model.filename, model.data);
  await uploadTrainingData(domain, nlu, stories);

  training.modelFilename = model.filename;
  training.status = FINISHED;
} catch (err) {
  training.status = FAILED;
} finally {
  await training.save();
}
```

Ukázka kódu 4.5: Implementace procesu trénování

Každá business entita musí mít kromě definice databázového modelu také definici GraphQL typu, který má stejně jako DTO funkci objektu pro přenos dat. Při vytváření instance třídy `ResolverBuilder` je nutné spolu s kontrolérem zadat i atributy DTO objektu. Sestavené rozhraní GraphQL dotazů a mutací je vidět v ukázce 4.7 na příkladě schématu entity externí akce. Pokud nějaká entita jednu z CRUD operací nepodporuje, stačí změnit parametry výsledného objektu a zavolat metody pro sestavení pouze některých operací.

Všechna schémata jsou nakonec spojena do objektu a předána v konstruktoru třídy `ApolloServer`, která je pomocí metody `applyMiddleware` aplikuje jako middleware frameworku Express. Důležitým parametrem je funkce `context`, která umožňuje přečíst informace nacházející se v objektu HTTP požadavku a zpřístupnit je resolverům v další fázi zpracování. V našem případě je v rámci kontextu prováděna autentizace uživatele a autorizace pro přístup do systému. V případě úspěšného ověření je objekt uživatele předán dál, což je vidět v ukázce 4.8.

Rozhraní služeb datové vrstvy se nachází ve složce `services` a obsahuje například metody pro komunikaci s API rozhraním trénovacího či produkčního Rasa serveru. V ukázce 4.9 je znázorněna implementace metody, která zašle požadavek typu GET na endpoint `/status`, čímž získá aktuální verzi načteného modelu a počet aktivně prováděných úloh.

Další příkladem z datové vrstvy je odběr událostí vkládaných produkčním Rasa serverem do fronty služby RabbitMQ. Ta funguje jako zprostředkovatel

```
class ResolverBuilder {
  constructor(fields, controller) {
    this.fields = fields;
    this.controller = controller;
    this.type = new GraphQLObjectType({
      name: this.controller.getModelName(),
      fields: () => fields
    });
  }

  buildCreate() {
    const { id, createdAt, ...args } = this.fields;
    return {
      [`create${this.controller.getModelName()}`]: {
        type: this.type,
        args,
        resolve: (parent, data, { user }) => {
          this.controller.authorize('create', user);
          return this.controller.create(data);
        }
      }
    };
  }
}
```

Ukázka kódu 4.6: Sestavení resolveru pro vytváření záznamů

zpráv mezi producentem a konzumentem. V ukázce 4.10 je provedeno vytvoření komunikačního kanálu pomocí metody `createChannel()`, zajištění existence fronty pomocí `assertQueue()` a přihlášení k odběru událostí z fronty metodou `consume()`. Přijaté události obsahují zprávy vyměněné mezi koncovým uživatelem a chatbotem spolu s informacemi o jejich klasifikaci. Každá z nich je nejprve převedena do formátu JSON a následně odeslána k dalšímu zpracování. Po dokončení je službě RabbitMQ odesláno potvrzení, na základě kterého je možné odebrat z fronty další událost.

Proces autentizace je prováděn strategií implementovanou pomocí třídy `BearerStrategy`, která je rovněž s využitím knihovny Passport aplikována jako middleware frameworku Express. Ta v ukázce 4.11 využívá API rozhraní služby Azure Active Directory pro validaci tokenu obsaženého v hlavičce požadavku. Z tokenu je následně získán e-mail uživatele, pomocí kterého je spárován se svým účtem.

```
const fields = {
  id: {
    type: new GraphQLNonNull(GraphQLID)
  },
  label: {
    type: new GraphQLNonNull(GraphQLString)
  }
};

const builder = new ResolverBuilder(fields, actionController);

const resolvers = {
  queries: builder.buildQueries(),
  mutations: builder.buildMutations()
};
```

Ukázka kódu 4.7: Definice GraphQL schématu externí akce

```
const app = express();
app.set('port', config.appPort);

const apolloServer = new ApolloServer({
  schema,
  context: async ({ req, res }) => {
    const user = await authenticateUser(req, res);
    if (!user) {
      throw new AuthenticationError();
    }
    return { user };
  }
});

apolloServer.applyMiddleware({
  app,
  path: '/graphql'
});
```

Ukázka kódu 4.8: Implementace třídy ApolloServer

```
async function getServerStatus() {
  const { data } = await send(SERVER_STATUS_ENDPOINT, 'GET');
  return {
    version: data.fingerprint.version,
    activeTrainingJobs: data.num_active_training_jobs
  };
}
```

Ukázka kódu 4.9: Metoda pro zjištění stavu Rasa serveru

```
const channel = await connection.createChannel();

await channel.assertQueue(config.queue);

channel.consume(config.queue, (message) => {
  const str = message.content.toString();
  handleMessage(JSON.parse(str)).then(() => {
    channel.ack(message);
  });
});
```

Ukázka kódu 4.10: Vytvoření komunikačního kanálu s RabbitMQ

```
const strategy = new BearerStrategy(
  {
    identityMetadata: config.identityMetadata,
    clientID: config.clientId
  },
  async (token, done) => {
    const email = token.email;
    const user = await userController.findByEmail(email);
    return done(null, user);
  }
);
```

Ukázka kódu 4.11: Inicializace strategie pro autentizaci uživatele

4.3 Frontend systému

Repozitář s frontendovou částí systému obsahuje veškeré zdrojové kódy klientské aplikace, ze které lze sestavit statickou webovou stránku. Adresářová struktura složky se zdrojovými kódy je následující:

/src
- components .... opakovaně používané komponenty
- config .... konfigurační soubory
- locales .... jazykové mutace systému
- providers .... autorizační a datové rozhraní
- resources .... komponenty pro editaci business entit
- utils .... pomocné nástroje a konstanty
- App.jsx .... hlavní komponenta aplikace
- index.js .... vykreslovací script frameworku React

4.3.1 Ukázky implementace

Každá business entita má ve složce `resources` komponenty pro vylistování záznamů, přidání nového a editaci stávajícího. Ty jsou tvořeny z podkomponent poskytnutých frameworkem React Admin, a pokud není jejich základní rozhraní dostačující, lze je na míru upravit, případně přidat vlastní. V ukázce 4.12 je pomocí komponent `List` a `Datagrid` sestavena tabulka pro výpis textových odpovědí chatbota s podporou stránkování.

```
const TextResponseList = props => (
  <List exporter={false} {...props}>
    <Datagrid rowClick="edit">
      <TextField source="label" />
      <TextField source="content" />
      <EditButton />
    </Datagrid>
  </List>
);
```

Ukázka kódu 4.12: React komponenta pro výpis záznamů

K editaci business entit jsou použity komponenty `Edit` a `SimpleForm`. Pro každý atribut jsou dle datového typu přidány formulářové prvky společně s validačními pravidly. To je znázorněno v ukázce 4.13, kde je navíc vložena komponenta `SlugPreviewField` pro zobrazení vygenerovaného klíče odpovědi s prefixem `utter`. Podobným způsobem je pak implementován i formulář pro vytváření záznamů.

```
const TextResponseEdit = props => (
  <Edit {...props} undoable={false}>
    <SimpleForm>
      <TextInput source="label" validate={[required()]} />
      <SlugPreviewField prefix="utter" />
      <TextInput source="content" validate={[required()]} />
    </SimpleForm>
  </Edit>
);
```

Ukázka kódu 4.13: React komponenta pro editaci záznamu

Všechny business entity je potřeba zaregistrovat v hlavní rodičovské komponentě `Admin`, čímž dojde k jejich přidání do navigace aplikace a propojení s API rozhraním servisní vrstvy. Konkrétní implementace způsobu komunikace s API, zajištění autentizace a autorizace či lokalizace uživatelského rozhraní je poskytnuta pomocí atributů `dataProvider`, `authProvider` a `i18nProvider`. Některé entity jsou viditelné pouze pro uživatele s určitou rolí, a proto jsou přidávány dynamicky. Použití komponenty `Admin` je znázorněno v ukázce 4.14.

```
const App = () => (
  <Admin
    dataProvider={dataProvider}
    authProvider={authProvider}
    i18nProvider={i18nProvider}
  >
    {permissions => [
      <Resource
        name="examples"
        create={ExampleCreate}
        edit={ExampleEdit}
        create={ExampleCreate}
      />,
      ...(permissions <= DEVELOPER ? getDevResources() : []),
      ...(permissions <= ADMIN ? getAdminResources() : [])
    ]}
  </Admin>
);
```

Ukázka kódu 4.14: React komponenta pro vykreslení aplikace

Pro komunikaci se servisní vrstvou je použit Data Provider založený na knihovně Apollo Client, která poskytuje třídy a metody sloužící pro navázání spojení s GraphQL API rozhraním. Všechny odchozí požadavky je možné modifikovat pomocí tříd `HttpLink` a `ApolloLink`, které jsou následně propojeny a použity v konstruktoru třídy `ApolloClient`. Díky nim je nastavena adresa API serveru a hlavička HTTP požadavku, kam je vložen JWT token z paměti prohlížeče. To je ukázáno na příkladě 4.15.

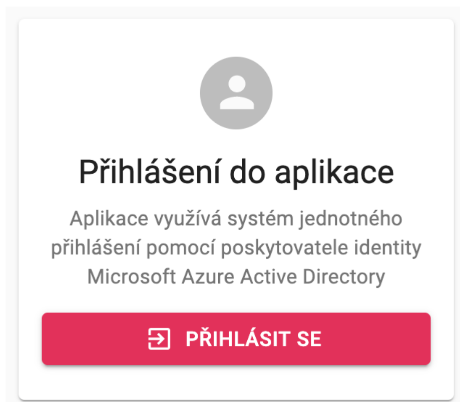
```
const httpLink = new HttpLink({
  uri: `${config.apiUrl}/graphql`
});

const authLink = new ApolloLink((operation, forward) => {
  const token = localStorage.getItem('msal.idtoken');
  operation.setContext({
    headers: {
      Authorization: `Bearer ${token}`
    }
  });
  return forward(operation);
});

const clientOptions = {
  link: authLink.concat(httpLink),
  cache: new InMemoryCache()
};
```

Ukázka kódu 4.15: Konfigurace odchozích požadavků GraphQL klienta

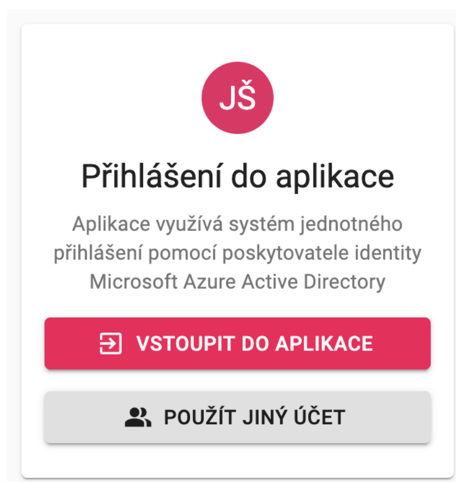
4.3.2 Uživatelské rozhraní

Před vstupem do systému se musí uživatel autentizovat. V případě, že se jedná o nové přihlášení, zobrazí aplikace komponentu s tlačítkem pro otevření vyskakovacího okna s přihlašovacím formulářem služby Azure Active Directory. Pokud však došlo z odhlášení z důvodu vypršení platnosti relace, zobrazí aplikace iniciály posledního uživatele spolu s tlačítky pro návrat do aplikace nebo změnu aktivního účtu. Oba popsané stavy jsou vidět na obrázcích 4.1 a 4.2.

V horní části uživatelského rozhraní se nachází panel s názvem stránky, tlačítkem pro aktualizaci dat a tlačítkem pro odhlášení uživatele. Po levé straně obrazovky je umístěna navigace se seznamem zaregistrovaných business entit a uprostřed se nachází komponenta s aktuální obsahem stránky. Na obrázku 4.3 je náhled na seznam vzorových zpráv, kde v prvním sloupci je

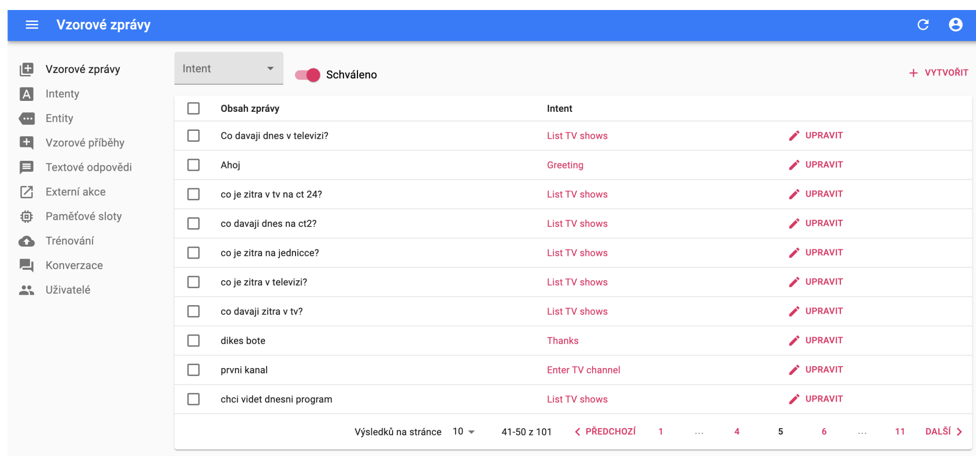


Obrázek 4.1: Náhled uživatelského rozhraní pro přihlášení



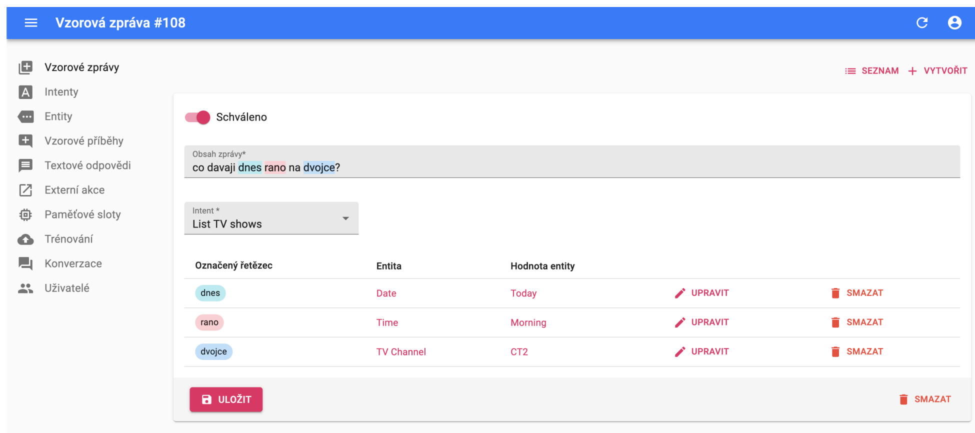
Obrázek 4.2: Náhled uživatelského rozhraní pro opětovné přihlášení

vypsán jejich obsah a v druhém název propojeného intentu. Formulář pro editaci vzorových zpráv je pak na obrázku 4.4. Zde jsou uvnitř textového vstupu pro obsah zprávy barevně vyznačeny všechny nalezené entity, jejichž typ a hodnotu je možné editovat kliknutím na záznam ve výpisu níže.

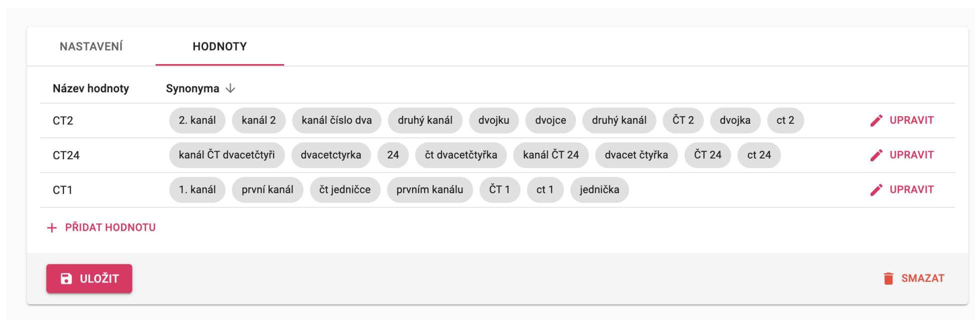


Obrázek 4.3: Náhled uživatelského rozhraní pro výpis vzorových zpráv

Komponenta pro výpis záznamů podporuje zároveň zobrazení dat, která jsou s danou business entitou v relaci. Tak tomu je na obrázku 4.5, kde jsou pro každou hodnotu entity vykreslena všechna její synonyma.



Obrázek 4.4: Náhled uživatelského rozhraní pro editaci vzorové zprávy

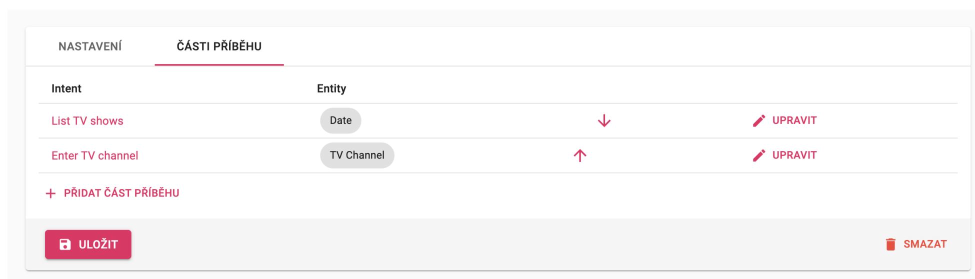


Obrázek 4.5: Náhled uživatelského rozhraní pro výpis synonym

U některých business entit záleží na pořadí, v jakém se objeví při sestavování trénovacích dat. K tomu slouží podpora manuálního řazení pomocí tlačítek nachazejících se v tabulce záznamů. Příkladem takové business entity jsou části vzorového příběhu, jako je tomu na obrázku 4.6.

V systému jsou evidovány všechny proběhlé procesy trénování spolu s jejich stavem, poznámkou, co bylo v dané verzi změněno a možností vygenerovaný model aplikovat na produkční Rasa server pomocí tlačítka `publikovat`. To je znázorněno na obrázku 4.7.

Po dokončení procesu trénování je provedena evaluace trénovacích dat, ze které je možné zjistit, jak kvalitní data jsou, jak často dochází k omylům při klasifikaci vzorových zpráv a zda je těchto dat dostatečné množství. K vizualizaci parametrů `precision`, `recall` a `f1-score` je použita knihovna Recharts, jejíž náhled je na obrázku 4.8.

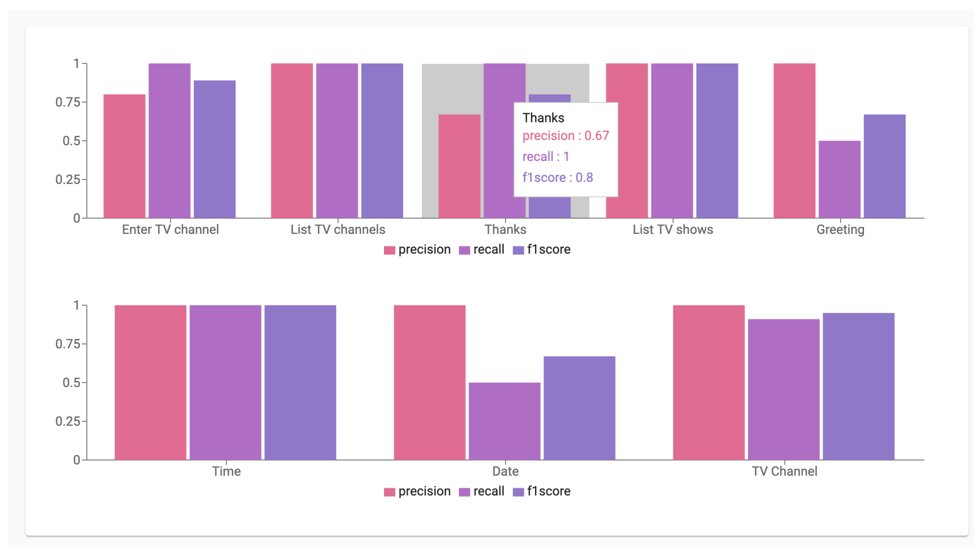


Obrázek 4.6: Náhled uživatelského rozhraní pro výpis částí příběhu

Stav	Poznámky k vydání		
✓	Přidány vzorové příběhy	STATISTIKY	PUBLIKOVAT
✓	Přidána entita pro čas vysílání	STATISTIKY	PUBLIKOVAT
✓	Přidány nové vzorové zprávy	STATISTIKY	PUBLIKOVAT
✓	Vylepšeny vzorové příběhy	STATISTIKY	PUBLIKOVAT
✓	První verze	STATISTIKY	PUBLIKOVAT

Výsledků na stránce 10 1-5 z 5

Obrázek 4.7: Náhled uživatelského rozhraní pro výpis trénování



Obrázek 4.8: Náhled uživatelského rozhraní pro vizualizaci evaluace dat

Kapitola **5**

Testování

Tato kapitola se věnuje testování backendové části aplikace s cílem ověřit funkcionalitu všech důležitých komponent, nalezení případných nedostatků a jejich následné odstranění. Testování umožňuje simulovat různé podmínky, které budou v provozu na systém kladeny. Cílem je sestavit testy takovým způsobem, aby pokryly co největší množství kódu, aby nekladly příliš velké nároky na jejich budoucí údržbu a aby byly snadno opakovatelné, nejlépe automatizovatelné. [67] Nejprve jsou popsány základní pojmy a rozdělení testů, dále jsou představeny použité technologie, po kterých následují jednotkové a integrační testy spolu s ukázkami implementace.

5.1 Základní pojmy

Dle znalosti vnitřní struktury aplikace rozdělujeme testy na tyto kategorie:

White Box předpokládají znalost zdrojového kódu a umožňují odhalit zdroj problému. Jsou však náchylnější na zásahy do implementace.

Black Box neřeší implementaci aplikace a testují pouze celkové chování proti vystavenému rozhraní. [68, 67]

Z pohledu rozsahu prováděných testů pak rozlišujeme následující kategorie:

Jednotkové testy se zaměřují testování tříd a metod v izolovaném prostředí. Většinou probíhají velice rychle, jsou snadno automatizovatelné a patří do kategorie White Box testů. Problémem je častá provázanost s okolím, což lze řešit mockováním objektů, které simulují závislé komponenty.

Integrační testy cílí na testování integrace a spolupráce komponent. Podle toho, zda se jedná o komponentu vlastní, patří do skupiny White Box, nebo Black Box, pokud jde o komponenty třetích stran.

Systémové testy patří do kategorie Black Box a testují aplikaci jako celek. Typicky jsou prováděny dle testovacího scénáře, který odráží případy užití definované v analýze projektu. [68]

5.2 Použité technologie

K provádění jednotkových a integračních testů JavaScript kódu je vybrána knihovna Jest z důvodu snadné konfigurace, možnosti mockování objektů a podpory testování asynchronního kódu. Knihovna obsahuje sadu pomocných funkcí označovaných `matchers`, které slouží k ověřování správnosti hodnot různých datových typů. [69] V ukázce 5.1 převzaté z dokumentace knihovny [70] je znázorněno testování jednoduché funkce sčítající dvě čísla. Správnost výpočtu je ověřena pomocí metod `expect()` a `toBe()`.

```
function sum(a, b) {
  return a + b;
}

test('Součet 1 + 2 se rovná 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Ukázka kódu 5.1: Jednoduchý test pomocí knihovny Jest

Mezi používané `matchers` patří tyto metody:

toBe() porovná výsledek s hodnotou

toThrow() očekává vyhozenou výjimku

toBeDefined() kontroluje, zda je hodnota definována [71]

Po zavolání příkazu `jest` v adresáři projektu dojde ke spuštění všech nalezených testů a vypsání výsledků do konzole:

```
Test Suites:  7 passed, 7 total
Tests:        54 passed, 54 total
Time:         6.433s, estimated 8s
```

Aby bylo možné testovat práci s databázovými modely, je rovněž použita knihovna Sequelize Mock, která vytváří falešné rozhraní knihovny Sequelize. [72] Na závěr je pro integrační testování GraphQL API rozhraní použita knihovna `apollo-server-testing` umožňující vytvořit testovacího klienta pro simulování komunikace s API pomocí metody `createTestClient()`. [73]

5.3 Jednotkové testy

Mezi důležité komponenty systému patří metody pro práci s trénovacími daty, jako je například transformace vzorových zpráv a příběhů do jazyka Markdown. V ukázce 5.2 je test pomocné metody `markEntities()`, která na vstupu přijme textovou zprávu a pole entit, ze kterých následně sestaví řetězec obsahující označená klíčová slova.

```
test('Vrátí zprávu se dvěmi označenými entitami', () => {
  const text = 'Jaké je dnes v Praze počasí?';
  const entities = [
    {
      from: 16,
      to: 20,
      entity: 'location',
      value: 'prague'
    },
    {
      from: 9,
      to: 12,
      entity: 'date',
      value: 'today'
    }
  ];
  const output = markEntities(text, entities);
  const expectedOutput = 'Jaké je [dnes](date:today) v
  ↪  [Praze](city:prague) počasí?';
  expect(output).toBe(expectedOutput);
});
```

Ukázka kódu 5.2: Jednotkový test metody pro označování entit

Mezi další komponenty, pro které je nutné jednotkový test vytvořit, patří metody třídy `BaseController`, která je rodičovskou třídou všech tříd obsahujících business procesy. Test v ukázce 5.3 vytvoří mock objektu `Intent` a otestuje správnost metody `create()` určené pro vytváření nových záznamů.

Důležitým aspektem systému je bezpečnost, a proto jsou vytvořeny testy ověřující proces autorizace uživatele při provádění operací nad business entitami. K tomu slouží metoda `authorize()` přijímající název operace a objekt uživatele. V našem systému může nové intenty vytvářet pouze role vývojáře nebo administrátora, proto při pokusu anotátora o provedení akce je vyhozena výjimka `ForbiddenError`. To je znázorněno v ukázce 5.4.

```
const db = new SequelizeMock();
intentController.model = db.define(
  'Intent',
  { label: 'Pozdrav' }
);

test('Vytvoří intent a porovná label', async () => {
  const intent = await intentController.create({
    label: 'Pozdrav'
  });
  expect(intent.label).toBe('Pozdrav');
});
```

Ukázka kódu 5.3: Jednotkový test metody pro vytváření záznamů

```
test('Při pokusu o vytvoření vyhodí výjimku', async () => {
  const loggedUser = UserMock.build({
    role: userRole.ANNOTATOR
  });
  expect(() => {
    intentController.authorize('create', loggedUser);
  }).toThrow(ForbiddenError);
});
```

Ukázka kódu 5.4: Jednotkový test autorizace uživatele

5.4 Integrační testy

Na vyšší úrovni, z pohledu zpracování požadavku uživatele, se nacházejí resolvery, které jsou podle názvu business entity mapovány na GraphQL dotazy či mutace. Je třeba ověřit jejich spolupráci s komponentami obalujícími business procesy a třídou pro stavbu GraphQL rozhraní `ResolverBuilder`. Test v ukázce 5.5 kontroluje sestavení mutace s názvem `createIntent`, pomocí které dojde po zavolání metody `resolve()` nejprve k autorizaci operace a následně k vytvoření nového intentu.

Na nejvyšší úrovni se pak nachází Apollo Server přijímající samotné HTTP požadavky obsahující GraphQL dotazy a mutace. V poslední ukázce 5.6 je proveden test odeslání mutace s názvem `createIntent` spolu s proměnnou `label` a následné porovnání obsahu nově vytvořené instance.

```
test('Zavolá resolver pro vytvoření intentu', async () => {
  const loggedUser = UserMock.build();
  const builder = new ResolverBuilder(args, intentController);
  const resolver = builder.buildCreate();
  const intent = await resolver.createIntent.resolve(
    null,
    { label: 'Poděkování' },
    { user: loggedUser }
  );
  expect(resolver.createIntent).toBeDefined();
  expect(intent.label).toBe('Poděkování');
});
```

Ukázka kódu 5.5: Integrační test třídy pro stavbu resolverů

```
const CREATE_INTENT = gql`
mutation createIntent($label: String!) {
  intent: createIntent(label: $label) {
    id
    label
  }
}
`;

test('Zavolá mutaci pro vytvoření intentu', async () => {
  const loggedUser = UserMock.build();
  const server = createTestServer(loggedUser);
  const { query } = createTestClient(server);
  const res = await query({
    query: CREATE_INTENT,
    variables: {
      label: 'Detail objednávky'
    }
  });
  expect(res.data.intent.id).toBeDefined();
  expect(res.data.intent.label).toBe('Detail objednávky');
});
```

Ukázka kódu 5.6: Integrační test zpracování GraphQL dotazu

Závěr

V práci byla provedena analýza, návrh, implementace a otestování administračního systému pro správu chatbotů. K tomu bylo v první řadě potřeba definovat základní pojmy z oblasti zpracování přirozeného jazyka, seznámit se s frameworkem Rasa Open Source, a následně prozkoumat jeho možnosti při tvorbě a administraci chatbotů s využitím textových konfiguračních souborů či systému Rasa X. Během analýzy byly zpracovány funkční a nefunkční požadavky, vytvořeny případy užití, které byly doplněny o scénáře užití a diagramy aktivit. Na závěr kapitoly byla provedena kontrola splnění funkčních požadavků. V kapitole věnované návrhu byla popsána architektura aplikace a to jak backendové, tak frontendové části. Pro implementaci obou částí byl vybrán jazyk JavaScript, kdy na backendu bylo použito běhové prostředí Node.js a na frontendu webový framework React. Ke komunikaci webové aplikace s API rozhraním backendu byl použit dotazovací jazyk GraphQL. Kapitola byla doplněna o sekvenční diagramy důležitých úloh a diagramy tříd business entit včetně jejich procesů. K ukládání dat byla vybrána databáze PostgreSQL, pro kterou byl v rámci kapitoly vytvořen relační databázový model. Návrh byl uzavřen částí věnovanou autentizaci a autorizaci uživatele pomocí poskytovatele identity Azure Active Directory. Kapitola zabývající se implementací řešila náhled na důležité třídy a metody, jak u backendové, tak frontendové části včetně ukázek uživatelského rozhraní. V závěrečné kapitole byly rozebrány přístupy k testování backendové části systému, mezi které patřily jednotkové a integrační testy. Kapitola byla doplněna o ukázky provedených testů s využitím knihovny Jest určené pro testování jazyka JavaScript.

Splnění cílů

V práci byly splněny všechny cíle, které spočívaly ve vývoji systému pro správu chatbotů postavených pomocí frameworku Rasa Open Source. Úspěšně byly zmapovány přístupy, sepsána analýza požadavků, navržena architektura, provedena implementace a otestování systému.

Přínosy práce

Díky práci se čtenář seznámí s postupnými kroky vývoje administračního systému od samotné analýzy problému, přes implementaci řešení až po jeho otestování. Dále práce umožňuje poznat framework Rasa Open Source, který nabízí nástroje pro snadnou tvorbu komplexních chatbotů. V neposlední řadě se čtenář naučí pracovat s dotazovacím jazykem GraphQL a využívat poskytovatele identity Azure Active Directory pro autentizaci uživatele.

Navazující plány

Administrační systém je před nasazením ve firmě ANECT a.s. nutné rozšířit o možnost správy formulářových komponent, které byly použity v chatbotovi pro produktové portfólio tvořeném kolegou Tomášem Stanovčákem a které nebyly z důvodu rozsahu součástí této práce. Rovněž je v plánu přidat po zkušebním provozu systému nové funkcionality, jako je interaktivní testování klasifikace, rozšíření odpovědí chatbota o navigační tlačítka nebo možnost označování importovaných konverzací pomocí štítků. V budoucnu je cílem vytvořit kompletní platformu, která umožní registrovaným uživatelům navrhnout vlastního chatbota, který bude za měsíční poplatek provozován na naší infrastruktuře.

Literatura

[1] Rasa Technologies Inc.: Rasa X: Preview of assistant improving. [online], [cit. 2019-12-15]. Dostupné z: https://rasa.com/docs/rasa-x/_images/rasa-x-nlu-training-mark-correct.png

[2] Rasa Technologies Inc.: Rasa X: Preview of deployment environments. [online], [cit. 2019-12-15]. Dostupné z: https://rasa.com/docs/rasa-x/_images/rasa-x-models.png

[3] Microsoft Corp.: Microsoft Docs: Layered Application Guidelines. [online], [cit. 2020-03-04]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658109(v=pandp.10)

[4] Khan, R.; Das, A.: *Build better chatbots.* Apress Media, první vydání, 2018, ISBN 978-1-4842-3110-4.

[5] Vogel, J.: *Chatbots: Development and Applications.* Diplomová práce, University of Applied Sciences, International Media and Computing, 2017.

[6] Rasa Technologies Inc.: NLP vs. NLU: What's the Difference and Why Does it Matter? 2019. Dostupné z: https://blog.rasa.com/nlp-vs-nlu-whats-the-difference/

[7] Rasa Technologies Inc.: Rasa: Open source conversational AI. [online], [cit. 2020-03-05]. Dostupné z: https://rasa.com/

[8] Rasa Technologies Inc.: Rasa: Getting Started with Rasa. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/getting-started/

[9] Rasa Technologies Inc.: Rasa: Architecture. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/architecture/

[10] Rasa Technologies Inc.: Rasa: Installation. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/installation/

[11] Rasa Technologies Inc.: Rasa: Tutorial Rasa Basics. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/rasa-tutorial/

[12] Rasa Technologies Inc.: Rasa: Domains. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/core/domains/

[13] Rasa Technologies Inc.: Rasa: Training Data Format. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/nlu/training-data-format/

[14] Rasa Technologies Inc.: Rasa: Stories. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/core/stories/

[15] Rasa Technologies Inc.: Rasa: Command Line Interface. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/command-line-interface/

[16] Rasa Technologies Inc.: Rasa: Testing Your Assistant. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/testing-your-assistant/

[17] Powers, D. M. W.: *Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation.* 01 2011, s. 2229–3981.

[18] Rasa Technologies Inc.: Rasa: Facebook Messenger. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/connectors/facebook-messenger/

[19] Rasa Technologies Inc.: Rasa: Rasa SDK. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/api/rasa-sdk/

[20] Rasa Technologies Inc.: Rasa: Actions. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/core/actions/

[21] Rasa Technologies Inc.: Rasa: Slots. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/core/slots/

[22] Rasa Technologies Inc.: Rasa X: Improve your contextual assistant with Rasa X. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa-x/

[23] Rasa Technologies Inc.: Rasa: Pricing. [online], [cit. 2020-04-1]. Dostupné z: https://rasa.com/product/pricing/

[24] Dennis, A.; Wixom, B. H.; Roth, R. M.: *Systems analysis and design.* John Wiley, páté vydání, ISBN 978-1-118-05762-9.

[25] Sommerville, I.: *Software engineering.* Pearson, 9 vydání, ISBN 0-13-703515-2.

[26] Requirements Engineering: Sběr a analýza požadavků. Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiložené SD kartě. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/171216/course/section/28533/3_RequirementsEngineering.pdf

[27] Chung, L.; Sampaio, J. C.: *On Non-Functional Requirements in Software Engineering.* 01 2009, s. 363–379.

[28] Analýza a sběr požadavků. Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiložené SD kartě. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/225503/mod_resource/content/4/03.prednaska.pdf

[29] Craig, L.: *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development.* Prentice Hall PTR, třetí vydání, 2004, ISBN 01-314-8906-2.

[30] OpenJS Foundation: Node.js: About. [online], [cit. 2020-02-18]. Dostupné z: https://nodejs.org/en/about/

[31] Mozilla and individual contributors: Express/Node introduction. [online], [cit. 2020-02-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

[32] TechMagic: React vs Angular vs Vue.js — What to choose in 2020? 2020. Dostupné z: https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d

[33] Facebook Inc.: React: A JavaScript library for building user interfaces. [online], [cit. 2020-04-19]. Dostupné z: https://reactjs.org/

[34] The PostgreSQL Global Development Group: PostgreSQL: About. [online], [cit. 2020-03-05]. Dostupné z: https://www.postgresql.org/about/

[35] Mozilla and individual contributors: Using a Database (with Mongoose). [online], [cit. 2020-03-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

[36] Depold, S.: Sequelize: Getting started. [online], [cit. 2020-03-05]. Dostupné z: https://sequelize.org/v5/manual/getting-started.html

[37] Depold, S.: Sequelize: Associations. [online], [cit. 2020-03-05]. Dostupné z: https://sequelize.org/v5/manual/associations.html

[38] Rasa Technologies Inc.: Rasa: HTTP API. [online], [cit. 2020-03-05]. Dostupné z: https://rasa.com/docs/rasa/api/http-api/

[39] Rasa Technologies Inc.: Rasa: Configuring the HTTP API. [online], [cit. 2020-06-02]. Dostupné z: https://rasa.com/docs/rasa/user-guide/configuring-http-api/

[40] Malý, M.: Používáme datové úložiště Amazon S3. 2009. Dostupné z: https://www.zdrojak.cz/clanky/pouzivame-datove-uloziste-amazon-s3/

[41] MinIO, Inc.: MinIO Quickstart Guide. [online], [cit. 2020-03-05]. Dostupné z: https://docs.min.io/

[42] Microsoft Corp.: Microsoft Docs: Business Layer Guidelines. [online], [cit. 2020-03-04]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658103(v=pandp.10)

[43] Baldwin, A.; Hahn, E.: Helmet: Express.js security with HTTP headers. [online], [cit. 2020-05-25]. Dostupné z: https://helmetjs.github.io/

[44] The GraphQL Foundation: GraphQL: A query language for your API. [online], [cit. 2020-04-18]. Dostupné z: https://graphql.org/

[45] The GraphQL Foundation: GraphQL: Introduction to GraphQL. [online], [cit. 2020-04-18]. Dostupné z: https://graphql.org/learn/

[46] Meteor Development Group Inc.: Apollo Server: Introduction to Apollo Server. [online], [cit. 2020-05-25]. Dostupné z: https://www.apollographql.com/docs/apollo-server/

[47] Microsoft Corp.: Microsoft Docs: Presentation Layer Guidelines. [online], [cit. 2020-04-19]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658081(v=pandp.10)

[48] Marmelab: React-Admin. [online], [cit. 2020-04-19]. Dostupné z: https://marmelab.com/react-admin/

[49] Google Inc.: Material Design: Get started. [online], [cit. 2020-04-19]. Dostupné z: https://material.io/collections/get-started/

[50] Marmelab: React-Admin: My First Project. [online], [cit. 2020-04-19]. Dostupné z: https://marmelab.com/react-admin/Tutorial.html

[51] Marmelab: React-Admin: Data Providers. [online], [cit. 2020-05-25]. Dostupné z: https://marmelab.com/react-admin/DataProviders.html

[52] Marmelab: React-Admin: ra-data-graphql-simple. [online], [cit. 2020-05-25]. Dostupné z: https://github.com/marmelab/react-admin/tree/master/packages/ra-data-graphql-simple

[53] Microsoft Corp.: Microsoft Docs: Crosscutting Concerns. [online], [cit. 2020-04-20]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658105(v=pandp.10)

[54] Microsoft Corp.: Microsoft Docs: Microsoft identity platform developer glossary. [online], [cit. 2020-04-20]. Dostupné z: https://docs.microsoft.com/en-us/azure/active-directory/develop/developer-glossary

[55] Auth0: Introduction to JSON Web Tokens. [online], [cit. 2020-04-20]. Dostupné z: https://jwt.io/introduction/

[56] Microsoft Corp.: Microsoft Docs: Authentication basics. [online], [cit. 2020-04-20]. Dostupné z: https://docs.microsoft.com/en-us/azure/active-directory/develop/authentication-scenarios

[57] Microsoft Corp.: Microsoft Docs: Application types for Microsoft identity platform. [online], [cit. 2020-04-20]. Dostupné z: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-app-types

[58] Microsoft Corp.: Microsoft Docs: Microsoft identity platform and Implicit grant flow. [online], [cit. 2020-04-20]. Dostupné z: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-implicit-grant-flow

[59] Microsoft Corp.: Microsoft Authentication Library for JavaScript (MSAL.js). [online], [cit. 2020-04-20]. Dostupné z: https://github.com/AzureAD/microsoft-authentication-library-for-js

[60] Marmelab: React-Admin: Authentication. [online], [cit. 2020-04-20]. Dostupné z: https://marmelab.com/react-admin/Authentication.html

[61] Hanson, J.: Passport: Documentation. [online], [cit. 2020-04-20]. Dostupné z: http://www.passportjs.org/docs/

[62] Microsoft Corp.: Microsoft Azure Active Directory Passport.js Plug-In. [online], [cit. 2020-04-20]. Dostupné z: https://github.com/AzureAD/passport-azure-ad

[63] Marmelab: React-Admin: Authorization. [online], [cit. 2020-04-20]. Dostupné z: https://marmelab.com/react-admin/Authorization.html

[64] OpenJS Foundation and other contributors: ESLint: Pluggable JavaScript linter. [online], [cit. 2020-05-25]. Dostupné z: https://eslint.org/

[65] Long J., contributors: Prettier: Opinionated Code Formatter. [online], [cit. 2020-05-25]. Dostupné z: https://prettier.io/

[66] npm, Inc.: npm Documentation: npm-package.json. [online], [cit. 2020-05-25]. Dostupné z: https://docs.npmjs.com/files/package.json

[67] Testování. Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiložené SD kartě. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/171216/course/section/28533/6_Testing.pdf

[68] Testování aplikací. Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiložené SD kartě. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/225520/mod_resource/content/2/09.prednaska.pdf

[69] Facebook Inc.: Jest: Delightful JavaScript Testing. [online], [cit. 2020-05-09]. Dostupné z: https://jestjs.io/en

[70] Facebook Inc.: Jest: Getting Started. [online], [cit. 2020-05-09]. Dostupné z: https://jestjs.io/docs/en/getting-started

[71] Facebook Inc.: Jest: Using Matchers. [online], [cit. 2020-05-09]. Dostupné z: https://jestjs.io/docs/en/using-matchers

[72] Blink UX: Sequelize Mock: Getting Started. [online], [cit. 2020-05-25]. Dostupné z: https://sequelize-mock.readthedocs.io/en/stable/docs/getting-started/

[73] Meteor Development Group Inc.: Apollo Server: Integration testing. [online], [cit. 2020-05-25]. Dostupné z: https://www.apollographql.com/docs/apollo-server/testing/testing/

Příloha A

Seznam použitých zkratek

FR Functional Requirement

NFR Non-functional Requirement

UC Use Case

CRUD Create, Read, Update, Delete

CORS Cross-Origin Resource Sharing

JSON JavaScript Object Notation

JWT JSON Web Token

HTTP Hypertext Transfer Protocol

API Application Programming Interface

URL Uniform Resource Locator

HTML Hypertext Markup Language

UI User Interface

ORM Object-Relational Mapping

AAD Azure Active Directory

S3 Simple Storage Service

SQL Structured Query Language

DTO Data Transfer Object

SSO Single Sign On

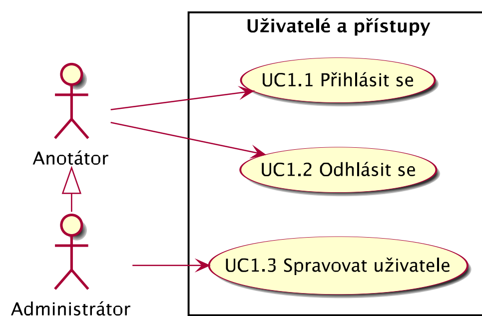
Příloha B

Obsah přiložené SD karty

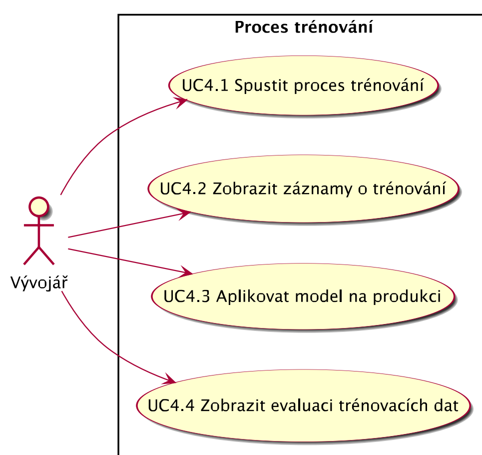
```
│ readme.txt ............................ stručný popis obsahu SD karty
├── thesis ......................... zdrojová forma práce ve formátu LaTeX
├── BP_Safarik_Jan_2020.pdf .................. text práce ve formátu PDF
└── literatura .............................. kopie citovaných přednášek
    ├── BI_SI1_Analyza_a_sber_pozadavku.pdf
    ├── BI_SI1_Testovani_aplikaci.pdf
    ├── BI_SI2_Requirements_Engineering.pdf
    └── BI_SI2_Testovani.pdf
```

Příloha C

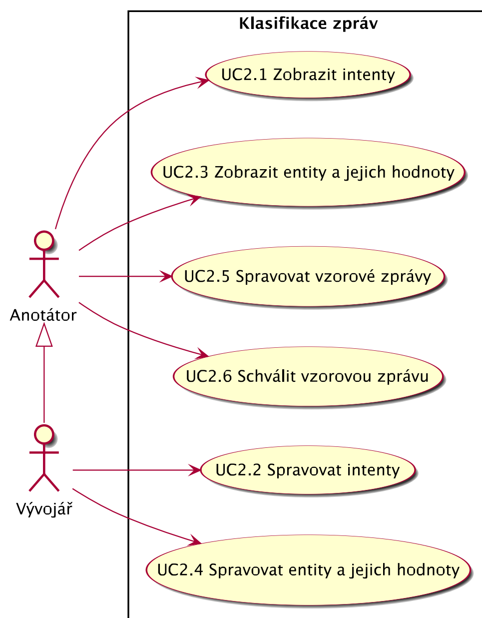
Případy užití



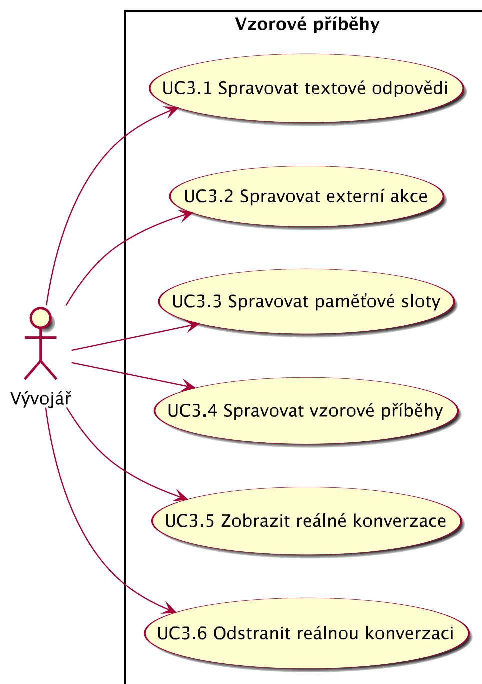
Obrázek C.1: Diagram případů užití pro přístup uživatelů



Obrázek C.2: Diagram případů užití procesu trénování



Obrázek C.3: Diagram případů užití klasifikace zpráv



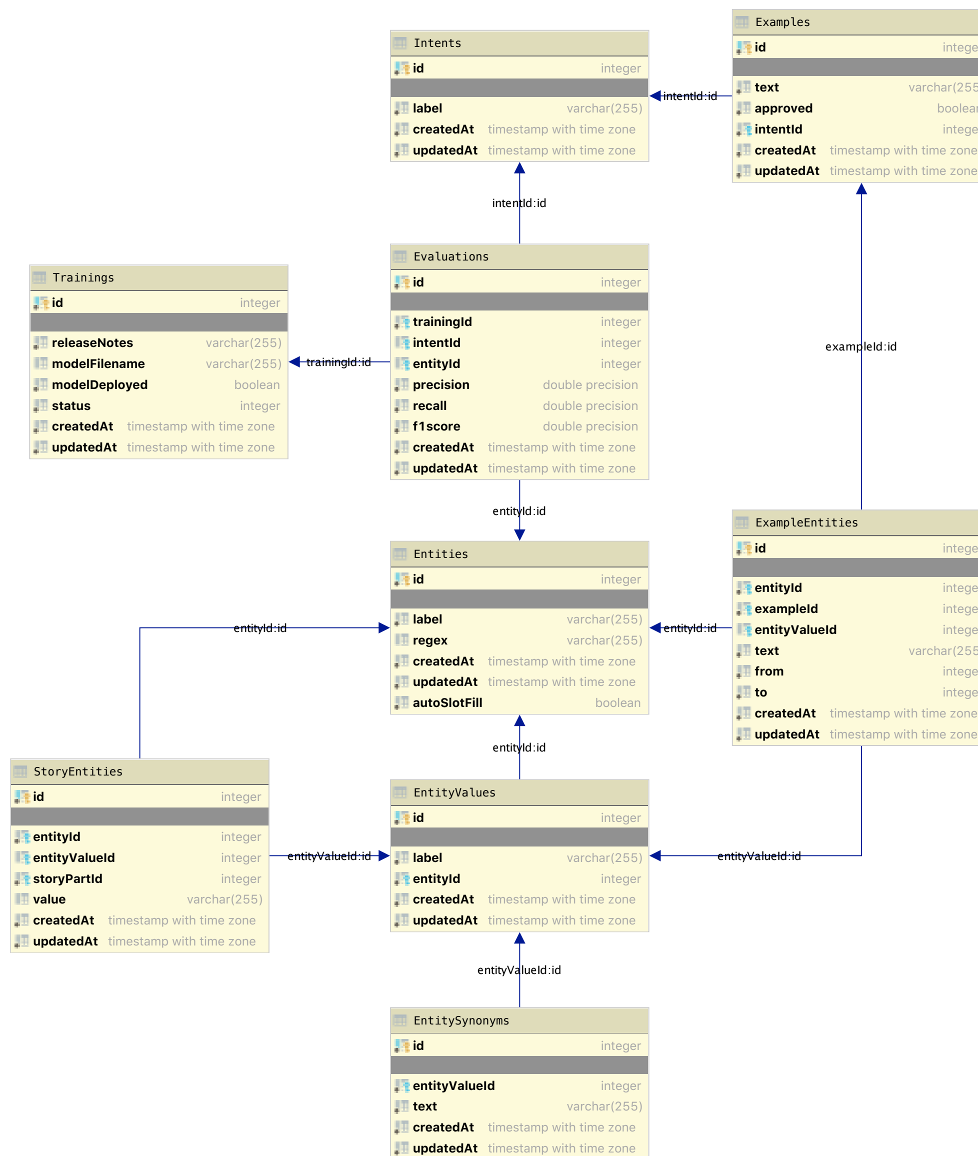
Obrázek C.4: Diagram případů užití vzorových příběhů

Příloha D

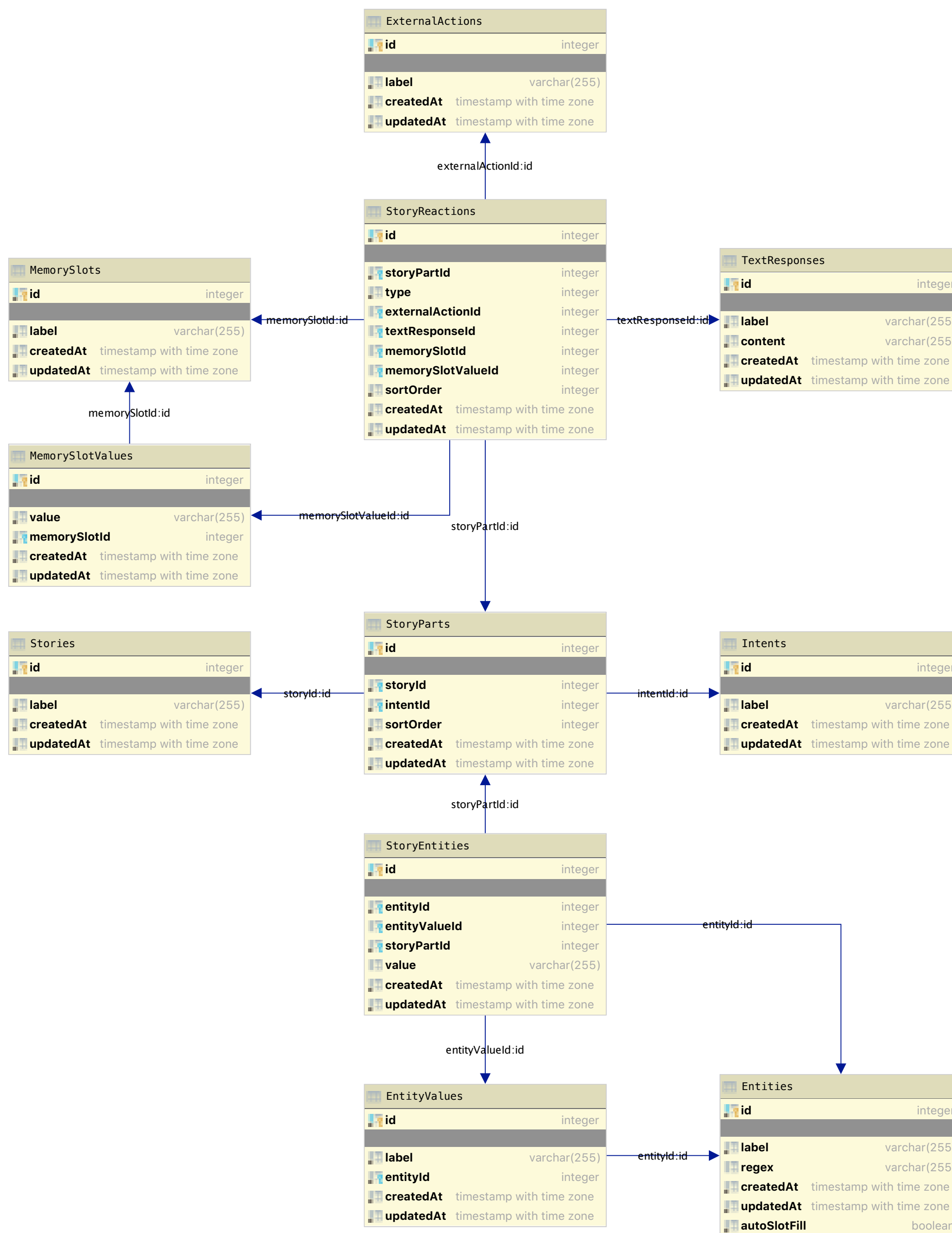
Databázový model

Obrázek D.1: Relační databázový model uživatelů

Obrázek D.2: Relační databázový model konverzací



Obrázek D.3: Relační databázový model vzorových zpráv



Obrázek D.4: Relační databázový model vzorových příběhů