



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DEPARTMENT OF INFORMATION SYSTEMS

**TRÉNINKOVÝ SIMULÁTOR PRO DISPEČERY
S PRVKY GAMIFIKACE A VYUŽITÍM VELKÝCH
PŘEDTRÉNOVANÝCH JAZYKOVÝCH MODELŮ**

A TRAINING SIMULATOR FOR DISPATCHERS WITH GAMIFICATION AND THE USE OF LARGE
PRE-TRAINED LANGUAGE MODELS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. MICHAL HOREJŠ

VEDOUCÍ PRÁCE

SUPERVISOR

RNDr. MAREK RYCHLÝ, Ph.D.

BRNO 2025

Zadání diplomové práce



162814

Ústav: Ústav informačních systémů (UIFS)
Student: **Horejš Michal, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Softwarové inženýrství
Název: **Tréninkový simulátor pro dispečery s prvky gamifikace a využitím velkých předtrénovaných jazykových modelů**
Kategorie: Informační systémy
Akademický rok: 2024/25

Zadání:

1. Seznamte se s podstatou a možným použitím velkých předtrénovaných jazykových modelů (LLM), prozkoumejte a porovnejte různé typy těchto modelů (např. LLaMA) a související služby (např. Open AI API, Azure Open AI). Seznamte se se správou incidentů v integrovaném záchranném systému (IZS) a s možnostmi trénování dispečerů správy incidentů IZS.
2. Navrhněte nástroj pro trénování dispečerů IZS v procesu vyhodnocení validity požadavku a založení incidentu a v procesu výběru výjezdové jednotky a vybavení, které potřebuje k řešení daného incidentu, vše v určitém časovém intervalu a s ohodnocením jednotlivých úkonů. Využijte LLM v roli osoby hlásící incident pro generování scénářů a interakcí v rámci uvedených procesů.
3. Po konzultaci s vedoucím navržený nástroj implementujte.
4. Řešení důkladně otestujte a vyhodnoťte jeho použitelnost pro daný účel, tj. trénování dispečerů správy incidentů IZS. Navrhněte možná rozšíření.

Literatura:

- Shao, Y., Li, L., Dai, J., & Qiu, X. (2023). Character-LLM: A Trainable Agent for Role-Playing. arXiv, 2310.10158. Dostupné na <https://arxiv.org/abs/2310.10158v2> a <https://github.com/choosewhatulike/trainable-agents>
- Du, W., Zhu, Z., Xu, X., Che, H., & Chen, S. (2024). CareerSim: Gamification Design Leveraging LLMs For Career Development Reflection. ACM Conferences. Association for Computing Machinery. Dostupné na <https://doi.org/10.1145/3613905.3650928>
- Pučálka, M. (2018). Herní engine pro ITIL trenažér. Brno. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

Při obhajobě semestrální části projektu je požadováno:
Body 1, 2 a rozpracovaný bod 3.

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

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1.11.2024
Termín pro odevzdání: 21.5.2025
Datum schválení: 22.10.2024

Abstrakt

Tato práce se zabývá problematikou efektivního výcviku operátorů integrovaného záchranného systému využitím velkých jazykových modelů a prvků gamifikace. Cílem bylo navrhnout a implementovat aplikaci, která umožňuje správu simulovaných incidentů, sběr klíčových metrik (doba reakce, přesnost vyplnění atributů) a zpětnou vazbu formou bodování. V průběhu vývoje vznikla webová aplikace využívající velký předtrénovaný jazykový model, který simuluje roli civilisty volajícího na tísňovou linku a se kterým komunikuje uživatel aplikace. Kontext incidentů je tvořen pomocí integrací na externí systémy, jakými jsou Open Street Maps nebo Registr územní identifikace, adres a nemovitostí. Uživatel se následně snaží získat z modelu klíčové informace o incidentu a ty následně musí korektně vyplnit. Nakonec je uživateli poskytnuta zpětná vazba využitím prvků gamifikace.

Abstract

This thesis addresses the challenge of effectively training Integrated Rescue System (IRS) operators by leveraging large language models and gamification elements. The goal was to design and implement an application that supports the management of simulated incidents, the collection of key performance metrics (response time, accuracy of attribute completion), and feedback through a scoring system. During development, a web application was created that uses a large pre-trained language model to simulate the role of a civilian calling the emergency line, with whom the user interacts. Incident contexts are generated via integrations with external systems such as OpenStreetMap and the Czech Territorial Identification, Address, and Real Estate Register. The user's task is to extract critical incident information from the model's responses and correctly enter it into the system. Finally, the user receives feedback through gamification features.

Klíčová slova

Informační systém, Java, Spring Boot, React, REST, gamifikace, velké jazykové modely, simulace

Keywords

Information system, Java, Spring Boot, React, REST, gamification, large language models, simulation

Citace

HOREJŠ, Michal. *Tréninkový simulátor pro dispečery s prvky gamifikace a využitím velkých předtrénovaných jazykových modelů*. Brno, 2025. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Tréninkový simulátor pro dispečery s prvky gamifikace a využitím velkých předtrénovaných jazykových modelů

Prohlášení

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

.....

Michal Horejš
21. května 2025

Poděkování

Děkuji panu RNDr. Marku Rychlému, Ph.D., za odborné vedení, silnou vstřícnost během tvorby práce a umožnění přístupu k serveru grace1.fit.vutbr.cz. Rád bych rovněž vyjádřil vděk panu Marku Řehůlkovi za poskytnutí datasetu, pravidelné odborné konzultace a možnosti nahlédnout do zákulisí práce operátorů tísňové linky.

Obsah

1	Úvod	4
2	Velké jazykové modely	5
2.1	Definice a charakteristika LLM	5
2.1.1	Transformátory s mechanismem pozornosti	6
2.2	Trénování LLM	7
2.2.1	Self-supervised trénování	7
2.2.2	Supervised trénování	7
2.2.3	Reinforcement trénování	7
2.3	Významné modely	8
2.3.1	Generative Pre-trained Transformer	8
2.3.2	Large Language Model Meta AI	9
2.3.3	Gemma	10
3	Webové aplikace	11
3.1	Hypertext Transfer Protocol	11
3.2	Model klient-server	12
3.3	Metody komunikace mezi klientem a serverem	12
3.3.1	Representational State Transfer	12
3.3.2	Simple Object Access Protocol	13
3.3.3	GraphQL	14
3.3.4	WebSocket	14
3.4	Frameworky pro tvorbu webových aplikací	15
3.4.1	React	16
3.4.2	Jakarta EE	16
3.4.3	Spring a Spring Boot	17
3.4.4	Python	17
3.5	Databáze	17
3.5.1	PostgreSQL	18
3.5.2	MongoDB	18
4	Specifikace požadavků	19
4.1	Požadavky na LLM v roli civilisty	19
4.2	Požadavky na prvky gamifikace	20
4.3	Ostatní požadavky a omezení	20
5	Analýza a návrh webové aplikace	21
5.1	Porovnání a výběr LLM modelů	21

5.1.1	Tvorba testovacího scénáře	21
5.1.2	GPT modely	22
5.1.3	Llama modely	22
5.1.4	Další modely a finální výběr	23
5.2	Návrh	23
5.2.1	Uživatelské rozhraní	23
5.2.2	Role v systému	25
5.2.3	Architektura systému a zvolené technologie	26
5.2.4	Autorizace, autentizace a zabezpečení rozhraní	28
5.2.5	Struktura databáze	30
6	Implementace	33
6.1	Serverová část	33
6.1.1	Endpointy	33
6.1.2	Simulace	36
6.1.3	Perzistentní vrstva	40
6.2	Klientská část	40
6.2.1	Komunikace se serverem	41
6.2.2	Komponenty	41
7	Testování	43
7.1	Automatizované testování	43
7.2	Uživatelské testování	44
8	Závěr	46
	Literatura	47
A	Ukázky aplikace	50
B	Seznam knihoven použitých pro tvorbu aplikace	53

Seznam obrázků

2.1	Architektura modelu transformátoru s více hlavovou pozorností. Převzato z [28].	6
2.2	Vysokoúrovňový přehled posilovacího učení z lidské zpětné vazby. Převzato z [25].	8
2.3	Ukázka transformátoru použitého pro generativní předtrénování. Převzato z [18].	9
3.1	Přehled principu komunikace s využitím architektonického stylu REST. Převzato z [15].	13
3.2	Obrázek znázorňující MVC architekturu a činnost každé ze tří částí. Převzato z [7].	16
5.1	Wireframe stránky simulace.	24
5.2	Wireframe stránky žebříčku výsledků.	25
5.3	Případ užití jednotlivých rolí v aplikaci.	26
5.4	C4 model navržené architektury systému.	27
5.5	Stavový digram popisující autentizaci uživatele v aplikaci.	29
5.6	ER diagram zobrazující strukturu databáze.	30
6.1	Architektura frameworku Spring Boot.	33
6.2	Zpracování HTTP požadavku využitím bezpečnostních filtrů.	36
7.1	Ukázka úryvku z různých konverzací. Snímek vlevo nahoře zobrazuje přijatelné chování modelu. Snímek vpravo nahoře zobrazuje korektní reakci na otázku mimo incident. Spodní snímky zobrazují nepřipustné nedostatky modelu.	45
A.1	Stránka pro přihlášení.	50
A.2	Stránka pro registraci.	51
A.3	Simulace v průběhu.	51
A.4	Zobrazené výsledky po dokončení simulace.	52
A.5	Žebříček výsledků.	52

Kapitola 1

Úvod

V posledních letech dochází v operačních střediscích integrovaného záchranného systému k výraznému odlivu zkušených operátorů. Důvodem je především vysoká psychická i fyzická náročnost jejich práce, kde se operátoři denně potýkají s neustálým tlakem při rychlém rozhodování v krizových situacích a jsou vystaveni nepředvídatelným i emotivně vypjatým scénářům. Díky tomu je těžké hledat za nové operátory náhradu a je ještě těžší nové operátory řádně proškolit a pustit je do praxe. Cílem této práce je hledat využití velkých jazykových modelů, které by mohly toto proškolení urychlit a následně sestrojít aplikaci, která by tento trénink umožňovala a to se snahou dosáhnout co nejvíce realistických scénářů. Zároveň využitím prvků gamifikace se pokusit zaujmout nové operátory, aby zaučování probíhalo zábavným způsobem a přidalo operátorům motivaci hodnocením jejich výsledků.

Kapitola 2 se obecně zabývá velkými jazykovými modely. Je zde rozebrána jejich architektura, způsoby trénování a také obsahuje přehled známých modelů, které byly vytvořeny a jsou stále vyvíjeny nejznámějšími celosvětovými firmami. V následující kapitole 3 jsou rozebrány základní koncepty fungování webových aplikací a protokoly používané ke komunikaci v síti internetu. Nalezneme zde také populární frameworky pro tvorbu klientské nebo serverové části s jejich výhodami a nevýhodami a také známé relační či nerelační databáze.

Kapitola 4 definuje funkční i nefunkční požadavky na velký jazykový model v roli civilisty, prvky gamifikace a celé aplikace. Kapitola 5 popisuje proces výběru vhodných předtrénovaných jazykových modelů, návrh uživatelského rozhraní, rolí v systému, architektury, struktury databáze a další. V navazující kapitole 6 je detailněji rozvedena implementace serverové části ve frameworku Spring Boot, klientské části v knihovně React a správa databáze PostgreSQL. Implementace je následně automatizovaně i ručně testována v kapitole 7, na kterou navazuje shrnutí nalezených poznatků v kapitole 8.

Kapitola 2

Velké jazykové modely

V posledních letech se velké jazykové modely (*Large Language Models*, *LLM*) staly jedním z nejzkoumanějších a nejdynamičtěji rozvíjejících se oborů v rámci zpracování přirozeného jazyka (*Natural Language Processing*, *NLP*). Jazykové modely obecně představují matematický nástroj pro předpovídání pravděpodobnosti posloupnosti slov či tokenů [9]. V posledních letech zaznamenávají výrazný nárůst přesnosti a schopností generovat či transformovat text, a to díky rostoucímu výpočetnímu výkonu a daleko objemnějším datovým sadám [2]. Tato kapitola se zaměřuje na detailnější pochopení LLM, jejich strukturu, trénink a validaci.

2.1 Definice a charakteristika LLM

Velké jazykové modely jsou pokročilé neuronové sítě určené k modelování lidského jazyka. V praxi to znamená, že dokážou na základě velkého množství trénovacích dat předpovídat pravděpodobnost výskytu dalších slov (či tokenů) v nějakém kontextu daného textu. Dokáží také případně tyto tokeny generovat. Od tradičních jazykových modelů se LLM liší zejména počtem trénovaných parametrů. V dnešní době se jedná řádově o miliardy až stovky miliard parametrů, což umožňuje, že se LLM naučí komplexní vztahy mezi slovy a větami, jazykové jevy a také znalosti z velké škály oborů. Pro představu, nejrozsáhlejší LLM společnosti Meta, s názvem Llama 3.1, obsahuje 405 miliard parametrů a zabírá 235 GB místa na disku [16].

Jednou z klíčových vlastností LLM je široká generalizace. Trénink na rozsáhlých korpusech textů (obvykle z veřejných zdrojů, webových stránek, aj.) dokáží modely zpracovat i zpracovávat dotazy i témata, která nebyla při jejich trénování explicitně anotována. Dalším charakteristickým rysem je flexibilita. Natrénované jazykové modely lze obvykle snadno přetrénovat (*fine tune*) pro specifické úlohy, jako je klasifikace sentimentu, shrnutí textu či detekce klíčových frází.

Architektura většiny současných LLM využívá transformátory, které jsou založeny na mechanismu pozornosti [8]. Tato architektura umožňuje LLM efektivně pracovat s dlouhým kontextem a dynamicky vyhodnocuje, kterým částem ze vstupní sekvence věnovat nejvíce pozornosti při generování výstupu. Architektura také přináší vyšší rychlost trénování a lepší schopnost zachytit dlouhodobé závislosti v textu oproti rekurentním neuronovým sítím (*Recurrent Neural Network*, *RNN*) [21], které se zakládají na cyklickém mechanismu, jenž umožňuje efektivní zpracování sekvencí dat, vhodném pro analýzu časových řad [12].

Díky těmto charakteristikám se v současné době velké jazykové modely uplatňují v celé škále oblastí. Například při analýze textů, kde dokáží automaticky vyhodnocovat velký objem dokumentů. Dále jako chatboti, kteří umějí v reálném čase komunikovat se zákazníky

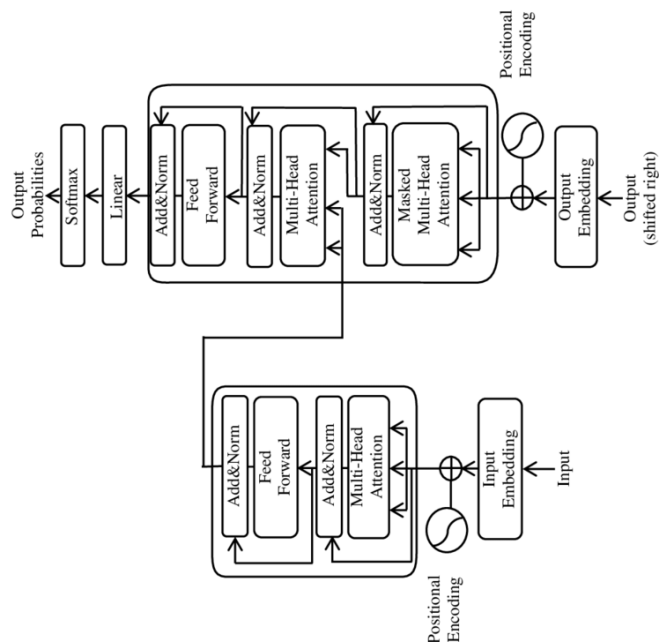
a řešit jejich požadavky, asistenti při psaní textů a v neposlední řadě se LLM uplatňují při překladu cizích jazyků, vyhledávání relevantních informací nebo shrnutí textů. Otázka spolehlivosti je zde stále otevřená. LLM dokáží generovat mylné informace, neetické výstupy či výstupy v nesouladu s autorským právem [3]. Proto jsou modely stále ověřovány, jsou jim nastavovány bezpečnostní pojistky a etická pravidla používání.

2.1.1 Transformátory s mechanismem pozornosti

Jedná se o nejpoužívanější architekturu v oblasti LLM a byla poprvé představena v roce 2017 týmem firmy Google [23].

Základním stavebním pilířem je mechanismus pozornosti (*self attention*). Ten funguje na principu vypočítání párových podobností mezi vstupními tokeny. Tokeny jsou převedeny do číselné reprezentace, ze které se odvodí tři klíčové složky: dotazy, klíče a hodnoty. Následně dojde k výpočtu váhy (skóre pozornosti) mezi dotazem a klíčem. V tento okamžik model zjistí, jak silně či slabě token souvisí s tokenem druhým. Tyto váhy se poté aplikují na dané hodnoty. Tím dojde ke zvýšení vlivu relevantních sousloví a potlačí se méně důležité části textu. Výsledkem je agregovaná reprezentace každého tokenu, která obsahuje kontext celého textového úseku.

Transformátory velmi často obsahují mechanismus s vícehlavovou pozorností. Principem tohoto mechanismu je vypočítání pozornosti paralelně ve více hlavičkách. Každá hlavička má k dispozici rozdílná vstupní data pro dotazy, klíče a hodnoty, což umožňuje zachytit více druhů vztahů mezi tokeny. Například syntaktické a sémantické závislosti. Tím je zvýšena vyjadřovací schopnost celého modelu.



Obrázek 2.1: Architektura modelu transformátoru s více hlavovou pozorností. Převzato z [28].

2.2 Trénování LLM

Po vytvoření architektury je potřeba model natrénovat. Tato fáze je nejkritičtější, neboť určuje kvalitu znalostí a jazykové dovednosti modelu. V praxi se dnes uplatňuje několik trénovacích způsobů. Všechny vyžadují velké výpočetní zdroje a pečlivý výběr textových korpusů, čili datasetu.

Dataset pro trénování tak velkých jazykových modelů musí být velmi rozsáhlý (až miliardy slov) a diverzifikovaný. Běžně využívanými zdroji jsou dnes např. *Common Crawl*, *Wikipedia* nebo *C4* [14, 2]. Tyto datasety jsou často nedostačující a obsahují nevalidní data. Proto tyto korpusy textu obvykle procházejí předzpracováním. Tím je myšleno filtrováním textu od nevhodných slov, duplicit, nevhodného obsahu a eliminace chybových nebo zraňujících forem v textu. Cílem je vytvořit co nejkvalitnější dataset, jelikož z něj model přejímá faktické informace a jazykové struktury.

2.2.1 Self-supervised trénování

V první fázi se nejčastěji pro trénování LLM využívá tzv. samoučící (*self-supervised*) přístup. Tento přístup nevyžaduje anotování datasetu, což je velmi žádoucí vzhledem k typickým rozměrům datasetů. Jelikož model nemá výsledná data předem definovaná, využívá se auto-regresivní trénování. Při využití auto-regresivního trénování dochází k predikci dalšího slova (tokenu) v sekvenci na základě předchozích slov. Například modely GPT jsou trénovány auto-regresivně. Dalším způsobem je maskované jazykové modelování. Model zde doplňuje náhodně zamaskovaná slova do textu na správnou pozici. Tím získává obousměrnou reprezentaci kontextu [4]. Oba způsoby se mohou i kombinovat.

Po dokončení této fáze je model schopen produkovat poměrně kvalitní texty. Dokáže porozumět syntaxi, sémantice i kontextu. Bohužel se může model naučit i nevhodné vzorce a zkreslení (bias) z trénovacích dat.

2.2.2 Supervised trénování

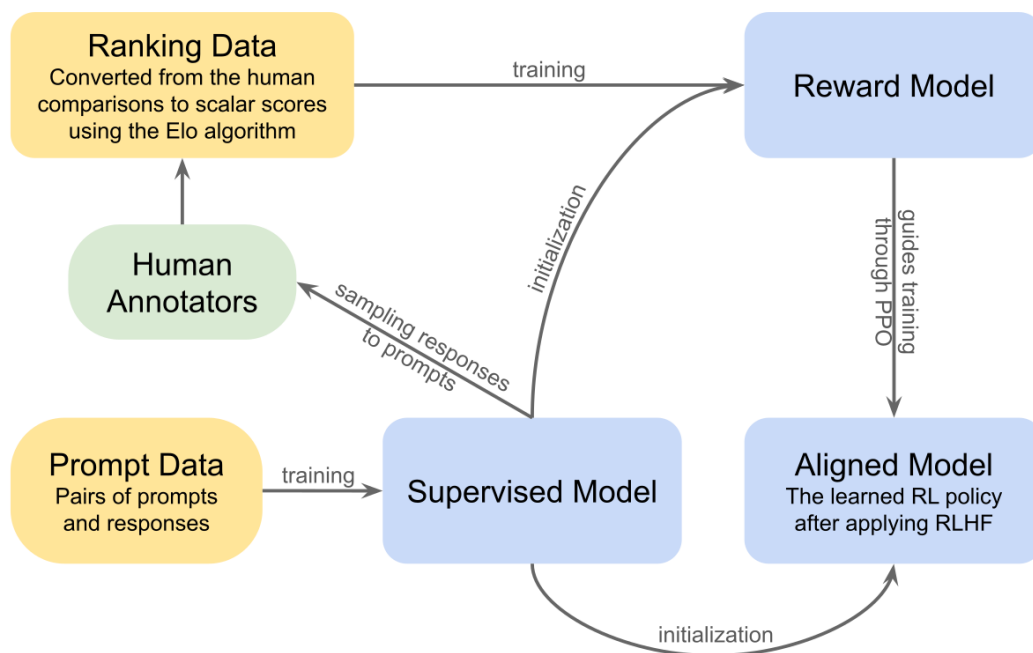
V druhé se využívá tréninku učení s učitelem (*supervised*). Zde jsou modelu podávána již strukturovaná anotovaná data s přesným zadáním. Typicky jde v této fázi o *fine-tuning* na upraveném datasetu. Tento dataset má pro každý vstup jednoznačnou správnou odpověď. Jedná se většinou o odpověď na dotaz, správnou klasifikaci sentimentu nebo styl psaní.

Model se v této fázi hlavně učí styl a konvence generování textu. Dokáže se tedy vyhýbat například citlivým tématům, respektive jsou modelu utvořena nová pravidla. Tím je dosaženo vyšší přesnosti a spolehlivosti modelu ve specifických úkolech. Mohou se ale stále projevovat některé zkreslené vzorce z předešlé fáze.

2.2.3 Reinforcement trénování

Poslední fází u velkých jazykových modelů bývá posilované učení (*Reinforcement Learning*, *RL*). Konkrétně tedy posilované učení z lidské zpětné vazby (*Reinforcement Learning from Human Feedback*, *RLHF*). Zde se využívá interakce modelu s prostředím nebo uživateli, kteří poskytují zpětnou vazbu. Jedná se o hodnocení kvality odpovědi nebo výběru, který z dvou vygenerovaných textů je vhodnější. Při kladném hodnocení uživatele za vygenerovaný text dostane model odměnu. V opačném případě penalizaci. Postupně se tedy modelu formuje jeho výstup na základě těchto reakcí.

Tato fáze ladí styl konverzace, zamezuje eticky nevhodnému jednání a chrání proti šíření dezinformací. Zároveň je model postupně přizpůsoben specifickým preferencím a cílům, které jsou obtížné vyjádřit anotovanými daty. Je to také jeden ze způsobů, který pomáhá řešit halucinace modelu [17].



Obrázek 2.2: Vysokoúrovňový přehled posilovacího učení z lidské zpětné vazby. Převzato z [25].

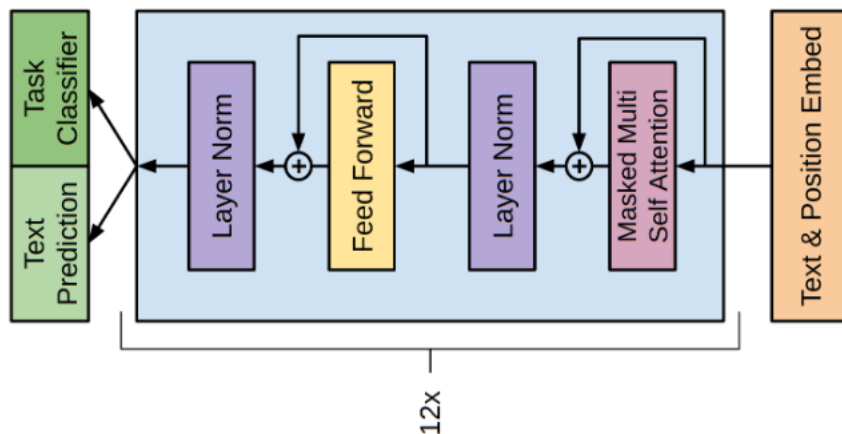
2.3 Významné modely

S rozmachem teoretických poznatků o transformátorech 2.1.1 a silně rostoucím výpočetním výkonem došlo k vytvoření několika klíčových velkých jazykových modelů, které výrazně posunuly zpracovávání přirozeného jazyka. Některé modely jsou tak schopné, že se de facto staly standardem pro vykonávání pokročilých úloh, jako jsou generování textu nebo konverzační asistenti. Tato sekce se zaměřuje na konkrétní nejznámější modely, jakými jsou série GPT, LLaMA nebo Gemini.

2.3.1 Generative Pre-trained Transformer

Generative Pre-trained Transformer (GPT) je rodina modelů, která je vyvíjena společností OpenAI. Základem všech verzí těchto modelů je auto-regresivní přístup, tedy model předpovídá další token v sekvenci na základě předchozích tokenů. Typickým rysem GPT modelů je jejich jednosměrné maskování, kde jsou modelu zakryty budoucí tokeny sekvence. Modely jsou naučeny tedy generovat text zleva doprava.

Vnitřní architektura transformátorů se skládá z mnohovrstvého bloku obsahujícího mechanismus pozornosti a „*feed-forward*“ vrstvy¹, kde je blok doplněn o reziduální spojení a normalizační techniky [5].



Obrázek 2.3: Ukázka transformátoru použitého pro generativní předtrénování. Převzato z [18].

První model od OpenAI, GPT-1, byl založen na 12 vrstvách a trénování probíhalo na datasetu BookCorpus, který obsahoval asi 5 GB textu. Zde bylo v roce 2018 prokázáno, že auto-regresivní přístup na bázi transformátorů může překonat dřívější rekurentní modely. Druhý model, GPT-2, již způsobil větší rozruch v oboru NLP. Model obsahoval 1,5 miliardy parametrů a již byl schopný generovat koherentní text. Model vznikl v roce 2019. Třetí model, GPT-3, ukázal výrazný skok co do velikosti, tak ve variabilitě schopností. Model obsahuje 175 miliard parametrů a vznikl v roce 2020. Zajímavostí je, že tak obrovský skok v počtu parametrů a schopností modelu se stal pouze za jediný rok od vydání druhé verze. Za zmínku stojí i model GPT-3.5, který byl optimalizován pro chatové asistenty a to v roce 2022. Pro postupné vylepšování modelu je zde využíváno i RLHF 2.2.3. Nejnovější generací GPT modelů je GPT-4 a pozdější verze „o“ a „o1“. Zde došlo v roce 2023 k rozšíření pro multimediální vstupy a je zde kladen důraz na rozpoznávání širších kontextových vazeb. Způsob trénování a velikost modelu zatím společnost nezveřejnila.

Série GPT modelů tedy přinesla fenomén univerzálních LLM, které jsou schopny i bez speciálního tréninku generovat relevantní odpovědi jen na základě kontextu zadaného pomocí jednoduchého promptování. V praxi jsou využívány jako asistenti pro psaní kódu, chatování nebo úpravu textu.

2.3.2 Large Language Model Meta AI

Large Language Model Meta AI (LLaMA) je rodina jazykových modelů představená společností Meta na začátku roku 2023. Hlavní myšlenkou těchto modelů je využití efektivního tréninku na kvalitních textových korpusech, díky kterému je nabízena rozsáhlá sada parametrizovaných variant, které se liší rozsahem, kvalitou, účelem a výkonností. LLaMA vychází, podobně jako GPT, z architektury transformátorů a auto-regresivního mechanismu. Důraz

¹Tato vrstva pouze vezme vstup, vynásobí jej s naučenými váhami, přičte naučený bias a vrátí vypočítaný výsledek.

je zde kladen na optimalizaci parametrů a lepší trénovací strategie. Cílem je tedy dosáhnout vysoké kvality generování textu při relativně menších velikostech modelů, v porovnání s jinými obřími modely.

Mezi základní modely se řadí LLaMA-7B a LLaMA-13B. Čísla u názvů znamenají počet parametrů modelů, tedy sedm a třináct miliard. Tyto modely vznikly na snaze o co nejvyšší výkon při co nejmenších nákladech. Jsou tedy vhodné pro aplikace s omezenou výpočetní kapacitou. LLaMA-30B a LLaMA-65B se řadí mezi středně rozsáhlé modely. Oproti menším modelům zvyšují kvalitu generovaného textu a pochopení širšího kontextu. Částečně se dají srovnat s GPT-3.5 v oblastech konverzace nebo analytických úkolů [16].

Společnost Meta zpřístupnila LLaMA modely na výzkumné účely. Díky veřejnému zdrojovému kódu tedy vzniklo mnoho upravených komunitních verzí pro různé účely. Oproti GPT sérii má LLaMA obecně menší modely, které jdou trénovat i na menším hardwaru. Nicméně menší modely mívají omezený kontext a je tedy dobré hledat kompromis mezi výkonem a velikostí modelu.

2.3.3 Gemma

Gemma [10] je projekt, který nabízí pokročilé LLM v rámci otevřeného zdrojového kódu. Důraz je zde kladen na maximální transparentnost a možnosti úprav, což je pravý opak komerčního řešení Gemini od Googlu. Gemma dokonce veřejně publikuje kód i váhy modelů a tím poskytuje vývojářům plnou kontrolu nad fungováním modelů. Modely lze přizpůsobovat specifickým potřebám, stejně jako je tomu u série LLaMA. Cílem jsou převážně menší až střední konfigurace v řádu několika miliard parametrů a upřednostňování efektivních tréninkových postupů, kvůli minimalizaci hardwarových nároků.

Jako je tomu u jiných LLM, základním architektonickým principem Gemmy je transformátor. Díky otevřenému zdrojovému kódu je možné nahlížet na fungování modelu od kódování textu až po feed-forward vrstvy a libovolně je upravovat. Je zde ale také silná snaha o rozšíření modelu i na multimediální vstupy nebo ponechat jádro dostatečně modulární, aby mohla komunita přidávat vlastní rozšíření. Modely jsou tedy silně vhodné pro vlastní přetrénování, dle vlastních potřeb.

Praktické využití Gemmy je tedy opravdu široké. K dnešnímu dni již existují verze optimalizované pro generování kódu, medicínské texty, zákaznickou podporu nebo pro jiné specifické projekty. Díky silné podpoře lze tedy očekávat další rychlý vývoj, tvorbu podpůrných knihoven a integraci s různými nástroji. Projekt tedy nabízí alternativu k proprietárním systémům.

Kapitola 3

Webové aplikace

Webové aplikace jsou nejrozšířenější softwarové řešení současnosti. Najdeme zde jednoduché firemní stránky, sociální sítě, e-shopy, osobní blogy i složité informační systémy pro různá odvětví. Všechny webové aplikace běží v prostředí internetu a uživatelům umožňují pracovat s obsahem vzdáleně. V tom tkví síla těchto aplikací, jelikož není nutná instalace programů lokálně. Rozvoj webových aplikací je úzce spjat s vývojem internetových protokolů, sofistikovaných frameworků a hardwaru, který je potřeba pro nasazení a zpřístupnění.

Cílem této kapitoly je představit základní koncepty fungování webových aplikací, včetně architektur a technologií s nimi spjatých. Nejdříve se zaměříme na HTTP (*Hypertext Transfer Protocol*), později na rozdíl mezi statickým a dynamickým webem, klient-server modelem. Dále budou představeny webové sokety, vybrané frameworky a možnost správy databází.

3.1 Hypertext Transfer Protocol

Hypertext Transfer Protocol je aplikační protokol používaný pro přenos dat v síti internetu. Základem je model požadavek-odpověď 3.2. Protokol se využívá pro přenos mnoha formátů dat. Zde patří například HTML (*HyperText Markup Language* dokumentů, obrázků, audia či souborů ve formátu JSON (*JavaScript Object Notation*)). HTTP je od počátku koncipován jako bezstavový protokol. To znamená, že mezi jednotlivými požadavky není udržován trvalý stav, což umožňuje snadné škálování a flexibilitu. Vyžaduje to ale použití různých metod pro správu sezení uživatele. Pro tento účel se využívají soubory cookies či tokeny. Nejrozšířenější verzi protokolu je HTTP/1.1[6], nicméně se v dnešní době využívá i HTTP/2.0 a postupně i HTTP/3.0.

HTTP definuje metody, hlavičky a stavové kódy, které popisují stav a výsledek zpracovaného požadavku. Data se mohou posílat jak v hlavičce, tak v těle požadavku. Odpovědi obsahují kromě samotného obsahu i hlavičku s metadaty. Pro kvalitnější výkon dnes mnoho moderních implementací využívá persistentní spojení, což minimalizuje režii při navazování nových spojení.

Jelikož veškerá komunikace pomocí protokolu je viditelná a zachytitelná, vznikla šifrovaná verze protokolu, HTTPS (*Hypertext Transfer Protocol Secure*). Tato verze zajišťuje důvěrnost a integritu přenášených dat. Technicky se jedná o kombinaci HTTP s vrstvou TLS/SSL. Při navázání spojení probíhá tzv. handshake, kde si obě strany vymění kryptografické klíče a dohodnou si šifrovací algoritmus. Jedná se tedy o nezbytný standard, který se využívá při manipulaci s citlivými daty nebo u bankovních transakcí.

3.2 Model klient-server

Model klient-server je jednou z nejrozšířenějších architektur v oblasti distribuovaných systémů, zejména u webových aplikací. Principem je rozdělení na dvě role. Roli klienta a serveru. Klient posílá požadavky a vyčkává na odpovědi. Typickým klientem jsou v dnešní době webové prohlížeče, mobilní aplikace nebo specializovaný software. Server očekává požadavky, zpracovává je a odesílá odpovědi zpět. Obvykle jde o webové stránky, databázové dotazy nebo výpočetní operace. Komunikaci ve většině případů zahajuje klient.

Výhodou klient-server modelu je, že nedochází k zatížení klientů složitou vnitřní logikou aplikace. Potřebují znát pouze definované aplikační rozhraní (*API*). Další výhodou je možnost centralizovaného řízení a škálování. Při vysokém zatížení lze přidávat nebo modernizovat servery, aniž by bylo nutné provádět jakoukoliv změnu na straně klienta. V neposlední řadě výhod umožňuje tento model lepší kontrolu nad bezpečností. Citlivá data a logika jsou koncentrovány na hlavním serveru, který lze dobře a pečlivě zabezpečit. Nevýhoda nastává v momentě poruchy hlavního serveru. Tudíž dojde k nezpřístupnění služby a je potřeba zálohování dat.

3.3 Metody komunikace mezi klientem a serverem

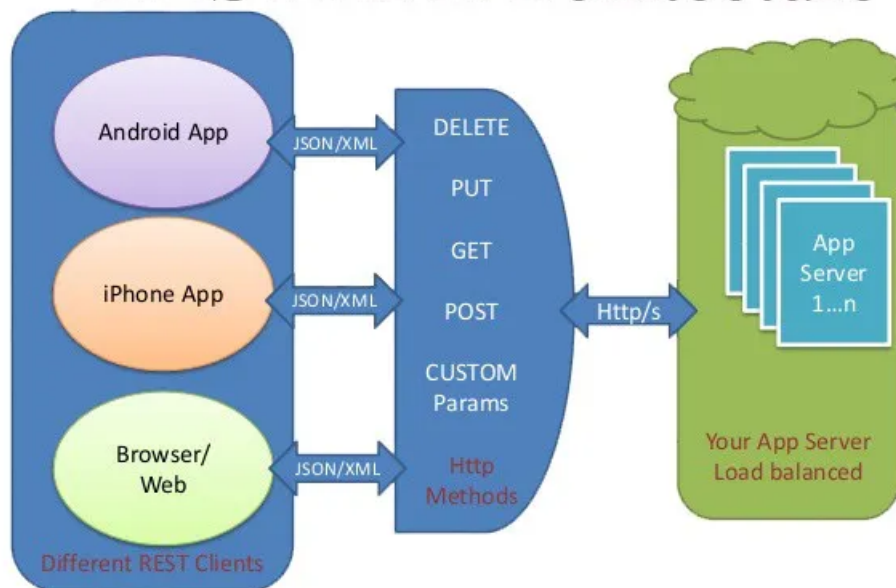
Moderní aplikace mohou využívat mnoho přístupů a protokolů pro komunikaci mezi klientem a serverem. Jelikož má každá aplikace jiné požadavky, je potřeba styl komunikace zvolit dle potřeb dané aplikace. V následujících podsekcích budou představeny čtyři nejčastěji využívané způsoby.

3.3.1 Representational State Transfer

Representational State Transfer (*REST*) je architektonický styl pro distribuované systémy. Jeho základním principem je bezstavová komunikace a realizuje se nad protokolem HTTP 3.1, kde jednotlivé HTTP metody reprezentují operace nad zdroji (GET, POST, aj.). Typicky je klientem zaslán požadavek ve formátu JSON na danou URL adresu, který server zpracuje a na základě dotazu odešle data zpět klientovi, často opět ve formátu JSON. Pro komunikaci se využívá i formát XML.

Díky využití standardních metod a stavových kódů HTTP je REST snadno pochopitelný a jednoduchý na implementaci. Při vhodném a konzistentním pojmenování URL adres lze dosáhnout robustního a přehledného aplikačního rozhraní. Formát zpráv není nijak definovaný. Každá aplikace nebo dokonce jednotlivá URL mohou mít definované jiné formáty zpráv. Je tedy potřeba API kvalitně dokumentovat. Další nevýhodou je, že obdržená data mohou být nedostačující a je potřeba ptát se znovu serveru nebo naopak je dat příliš a je potřeba mezi nimi filtrovat na klientovi.

REST API Architecture



Obrázek 3.1: Přehled principu komunikace s využitím architektonického stylu REST. Převzato z [15].

3.3.2 Simple Object Access Protocol

Simple Object Access Protocol (*SOAP*) je protokol pro výměnu strukturovaných zpráv. Zprávy jsou obvykle posílány ve formátu XML. Protokol je široce rozšířen mezi podnikovými službami a integracemi z důvodu přísné standardizace. Protokol běží nad HTTP, dá se ale nasadit i nad jinými protokoly. Například SMTP [27]. SOAP má bohatý systém specifikací, zaměřených na bezpečnost, spolehlivý přenos a další požadavky. Využívá se zejména v aplikacích, kde je kladen důraz na formalitu API a silnou validaci dat. Rozhraní se definuje jazykem WSDL (*Web Service Description Language*), který přináší požadovanou formalitu. Přináší také značnou složitost definic a i jednoduché požadavky mohou vést na dlouhé XML zprávy, což může zbytečně zatěžovat síť a rychlou implementaci.

```

1 <!-- Dotaz -->
2 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
3   <soap:Body>
4     <vratDetailProduktu xmlns="http://sklad.ukazka.com/ws">
5       <produktID>12345</produktID>
6     </vratDetailProduktu>
7   </soap:Body>
8 </soap:Envelope>
9
10 <!-- Odpoved -->
11 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
12   <soap:Body>
13     <vratDetailProduktuOdpoved xmlns="http://sklad.ukazka.com/ws">
14       <vratDetailProduktuVysledek>
15         <jmeno>Stolni pocitac</jmeno>
16         <produktID>12345</produktID>
17         <popis>Herni stolni pocitac, cerveny</popis>
18         <cena>98,50</cena>
19         <naSklade>ano</naSklade>
20       </vratDetailProduktuVysledek>
21     </vratDetailProduktuOdpoved>
22   </soap:Body>
23 </soap:Envelope>

```

Výpis 3.1: Ukázka dotazu a odpovědi využitím SOAP pro získání detailu o produktu.

3.3.3 GraphQL

GraphQL byl vyvinut společností Meta. Jedná se o dotazovací jazyk pro komunikaci mezi klientem a serverem. Úkolem klienta je deklarativně popsat strukturu požadovaných dat a v jakém formátu má odpověď vypadat. Klient se může dotázat i na více typů informací jedním dotazem. Server pouze vrátí ty části dat, které si klient vyžádal v daném formátu. Dochází zde tedy ke snížení přenášených dat a minimalizaci požadavků. GraphQL taktéž nabízí schémata, která formalizují strukturu dat a vztahy mezi nimi. Tím je usnadněna validace dat a dobře se dokumentuje aplikační rozhraní. Jedná se tedy o flexibilní jazyk, u kterého je potřeba kvalitně nakonfigurovat tzv. resolvery, aby nedocházelo k nepotřebnému zatěžování serveru. Resolver je funkce nebo metoda, která řeší návratovou hodnotu pro typ nebo atribut ve schématu. Náročněji se zde řeší kešování, které je například u RESTu přirozené na úrovni HTTP.

3.3.4 WebSocket

WebSokety nabízejí obousměrnou a trvalou komunikaci mezi klientem a serverem. Od ostatních protokolů se liší tím, že dotazovat se může i server klienta. Po ustálení připojení se z protokolu HTTP stává WebSocket spojení. Zde mohou obě strany nezávisle posílat zprávy. WebSokety umožňují komunikaci v reálném čase a aktualizaci stránek bez nutnosti stránky znovu načíst. Využívají se ke hraní online her, pro burzovní systémy či online chatování.

Velká přednost WebSocketů je nízká latence, jelikož odpadá režie spojená s vytvářením spojení. Data jsou tedy dostupná téměř okamžitě. Potřeba udržovat stále připojení je náročná pro server, který musí kapacitně obsloužit mnoho simultánních spojení. V dnešní době jsou WebSokety nepostradatelné pro pohodlí uživatele webové aplikace.

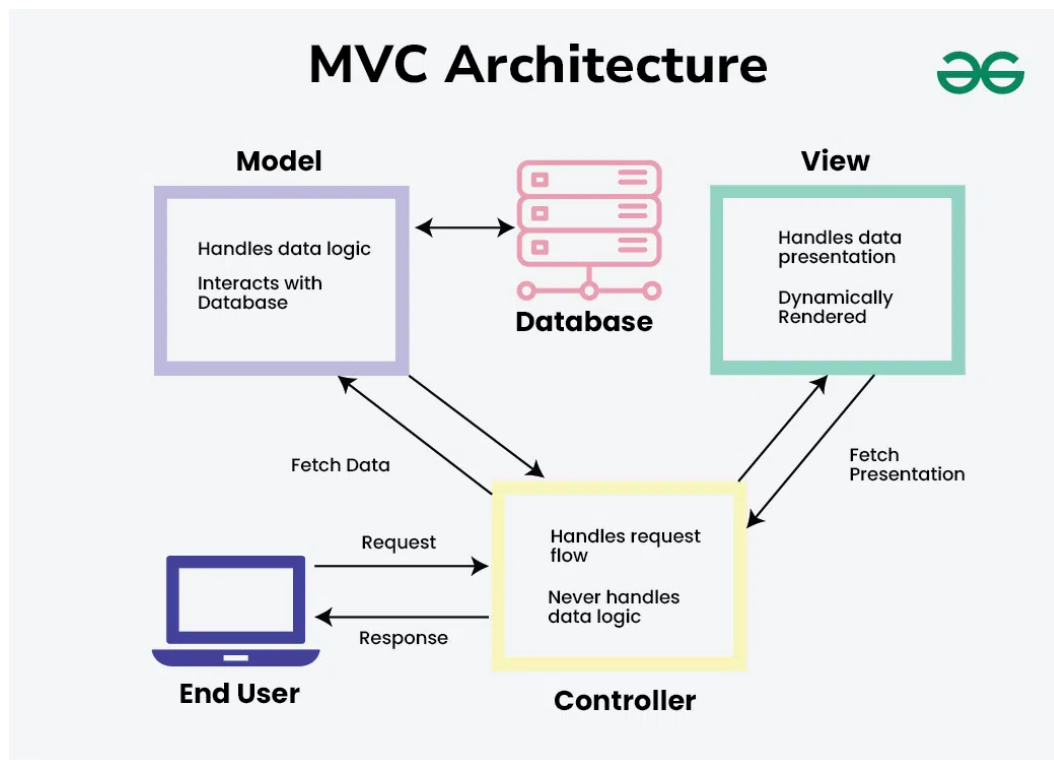
3.4 Frameworky pro tvorbu webových aplikací

Vývoj webových aplikací se za poslední 2 dekády přeměnil z ručně psaných skriptů až k používání sofistikovaných frameworků. Tyto frameworky nabízejí předdefinované šablony a nástroje pro rychlý a efektivní vývoj. Obvykle se frameworky řídí architektonickým vzorem a poskytují podporu pro správu routingu, komunikaci s databází, autentizaci a autorizaci či generování uživatelského rozhraní. Následující sekce jsou zaměřeny na architektonický vzor MVC (*Model-View-Controller*) a nejznámější technologie pro tvorbu klientské a serverové části.

Model-View-Controller

Architektonický vzor MVC je jedním z nejpoužívanějších konceptů pro rozvržení aplikace i přes fakt, že se jedná o jeden z nejstarších konceptů. Aplikace je rozvržena na tři oddělené části. První z nich je Model. Ten představuje datovou vrstvu a je zde implementován přístup do databáze a logika práce s daty. View představuje prezentaci dat a zahrnují se zde různé HTML šablony, CSS styly či JavaScript animace, kde všechny tyto složky tvoří uživatelské rozhraní. Poslední částí je Controller. Controller funguje jako prostředník mezi dvěma zbylými částmi. Přijímá klientské požadavky, komunikuje s modelem a vrací výsledná data zpět do View. Tato separace aplikace silně zvyšuje udržitelnost a přehlednost kódu. Změny ve vzhledu se dotýkají často pouze View, kdežto změny v obchodní logice se odehrávají v modelu. Controller často pouze deleguje požadavky [7].

Mezi hlavní výhody MVC patří jednoduchost, srozumitelnost a modularita. V dnešní době tento vzor implementuje značné množství frameworků. Jasná struktura umožňuje rozumné rozdělení vývojářů na jednotlivé části. Při vývoji se část týmu zaměřuje na logiku aplikace, tedy na model, zatímco uživatelské rozhraní vyvíjí jiná část týmu. V praxi může vést MVC k nadměrné složitosti, hlavně pokud se aplikace rozroste a doplní o další vrstvy. Tím vznikají modernější pojetí, jako je MVVM (*Model-View-ViewModel*) nebo MVP (*Model-View-Presenter*). Přesto zůstává MVC základním referenčním bodem pro řadu populárních frameworků.



Obrázek 3.2: Obrázek znázorňující MVC architekturu a činnost každé ze tří částí. Převzato z [7].

3.4.1 React

React je knihovna pro tvorbu uživatelského rozhraní, která byla vyvinuta společností Meta. Základní filozofií knihovny je vytváření komponent. Každá komponenta reprezentuje menší část uživatelského rozhraní. Například formulář, seznam položek nebo jednoduché tlačítko. Tyto komponenty lze pak libovolně skládat do rozsáhlých celků. Komponenty jsou psané v syntaxi JSX¹. Aby docházelo k efektivním aktualizacím komponent, využívá React virtuální DOM (*Document Object Model*) a tím dochází k minimalizaci opětovného vykreslení komponent, ve kterých nedochází k žádné změně. V modelu MVC zaujímá React roli View, ale lze React použít i na řízení logiky a datovou vrstvu s pomocí dalších rozšiřujících knihoven. Častějším řešením je použití Reactu pouze na tvorbu uživatelského rozhraní a se serverem dochází ke komunikaci pomocí REST rozhraní.

3.4.2 Jakarta EE

Jakarta EE (dříve známá jako Java EE) je soubor specifikací a technologií v jazyce Java, které pokrývají širokou škálu podnikových potřeb. Je zde možná implementace, zpracování HTTP požadavků, šablonování nebo správa transakcí. V MVC architektuře může Jakarta zastávat všechny tři části. Obvykle se ale využívá pro Controller a Model. Mezi klíčové vlastnosti Jakarty patří *Servlet API*, na které se zakládají veškeré nové technologie (*JavaServer Pages (JSP)*, *JavaServer Faces (JSF)*) týkající se zpracování požadavku a generování výstupu. Pro práci s relačními databázemi je využíváno *Java Persistence API (JPA)* a pro

¹ Jedná se o rozšíření jazyka JavaScript.

správu závislostí framework *Contexts and Dependency Injection (CDI)*. Tyto standardy jsou pak implementovány v aplikačním serveru, do kterého je pak aplikace nasazena.

3.4.3 Spring a Spring Boot

Spring je tradičním představitelem MVC přístupu ve světě Javy. Jedná se o nejrozšířenější a nejflexibilnější Java framework. Často využívá specifikací z Jakarty EE, které jsou často poupraveny. Nejznámější je pro inverzi řízení (*Inversion of Control (IoC)*)² a vkládání závislostí, čímž silně usnadňuje modularizaci a testování aplikací.

Díky bohatému ekosystému lze ve Springu vytvářet malé projekty i robustní mikroslužby, a to s podporou cloudových služeb. Obsahuje spoustu optimalizovaných a předpřipravených modulů, které řeší běžné problémy dnešních aplikací. Nevýhodou Springu je relativní komplexita. Je nutné porozumět konfiguračním prvkům a modulům pro kvalitní fungování aplikace.

3.4.4 Python

V jazyce Python existuje řada webových frameworků, nejznámějším je ale Django. Django je koncipován jako vysokoúrovňový framework, který přináší, stejně jako Spring, mnoho nástrojů. Například vlastní mapování objektů na databázové entity, směřování a šablonování. Jazyk sám o sobě je silně vhodný pro vytváření skriptů a obsahuje mnoho předdefinovaných knihoven, díky kterým je možné vytvořit vlastní API pro komunikaci s lokálním LLM. LLM je zde možné i různě upravovat či přetrénovat. Jsou zde dokonce připravené knihovny na „promptování“³ vzdálené jazykové modely.

3.5 Databáze

Databáze slouží k persistentnímu ukládání dat a proto představují klíčovou součást architektury aplikací. Databáze se dělí na relační a nerelační.

V relačních databázích jsou data ukládána do tabulek, kde řádky odpovídají konkrétním entitám a sloupce představují jejich atributy. S daty se pracuje pomocí jazyka SQL (*Structured Query Language*). Mezi klíčové vlastnosti databází se řadí ACID. Atomicita (Atomicity) udává, že při provádění změn jsou provedeny všechny změny nebo žádná. Konzistence (Consistency) že transakce nesmí narušit integritu databáze, tedy porušit definovaná pravidla. Izolovanost (Isolation) značí, že více souběžných transakcí se nesmí nijak ovlivnit a trvalost (Durability) znamená, že provedené změny jsou trvale uloženy a nemohou se ztratit [20]. Hlavní výhodou je formalita schémat, která usnadňuje práci s daty. Oproti nerelačním databázím jsou méně flexibilní.

Nerelační databáze (dále *NoSQL*) vznikly v reakci na potřeby zpracovávání obrovských objemů nestrukturovaných či částečně strukturovaných dat. Databáze se zde dělí na dokumentové, klíč-hodnota, sloupcové a grafové. Každý typ databáze je využíván pro jiné struktury dat. Jednotícím rysem je ale flexibilnější schéma a jednodušší horizontální škálování. Silnou nevýhodou je nepodporování všech transakčních vlastností a je nutné si tedy vybírat, jaké vlastnosti jsou potřeba pro danou aplikaci.

²Jde o převzetí některých programátorských povinností aplikačním serverem. Například vytváření instancí tříd.

³Proces úpravy nebo přizpůsobení LLM pro specifické účely a scénáře.

3.5.1 PostgreSQL

PostgreSQL⁴ je relační databázový systém s veřejným zdrojovým kódem, který začal vznikat již v 80. letech minulého tisíciletí v Berkeley. V současné době je vyvíjen komunitou dobrovolníků a nadšenců. PostgreSQL je známý pro svou širokou podporu standardů SQL, rozšiřitelnost a bohatý ekosystém rozšíření, kde některá rozšíření přidávají funkce pro geografická data, monitorování nebo indexovací nástroje.

Jednou z hlavních předností PostgreSQL je silná podpora ACID transakcí, a je to tedy vhodný a spolehlivý systém i pro kritické podnikové systémy. Kromě tradičního modelu nabízí systém i práci s formátem JSON, čímž je zde možné kombinovat strukturovaná i nestrukturovaná data. Je zde možnost vytvářet i pokročilé indexy, triggerů nebo využívat pro práci s daty i procedurální jazyky⁵. PostgreSQL je možné škálovat prostřednictvím synchronní i asynchronní replikace nebo nasazovat na více uzlů. Díky silné všestrannosti a žádným licenčním poplatkům se jedná o výborný kompromis pro širokou škálu aplikací.

3.5.2 MongoDB

MongoDB⁶ je dokumentově orientovaná NoSQL databáze vyvíjená společností MongoDB Inc. První verze byla vydána v roce 2009. Jelikož se jedná o nerelační databázi, poskytuje ukládání dat ve flexibilním formátu BSON (binární JSON). To umožňuje snadnou práci s dynamickými schématy a vkládání do kolekce dokumentů s různou strukturou.

MongoDB nabízí vlastní dotazovací Domain-Specific Language (dále *DSL*), který umožňuje filtrování, agregaci, řazení, projekci přímo nad dokumenty nebo také přidává operátory pro práci s poli a vnořenými objekty. V této databázi je možné vytvářet i geoprostorové indexy, vyhledávání pomocí textu nebo časové řady. V dnešní době již MongoDB zajišťuje ACID vlastnosti v rámci jednoho replika setu, tedy mezi instancemi databáze, které spravují stejný set dat.

Databáze má vestavěné horizontální škálování za pomoci tzv. „*sharding*“ mechanismu, což je rozdělení částí databáze na části, které jsou spuštěny na jiných zařízeních. Je tedy databáze vhodná pro systémy, které vyžadují vysokou dostupnost a rychlý zápis a čtení velkého objemu dat. Díky flexibilitě schématu a nástrojům pro správu a zabezpečení je databáze vhodnou volbou pro širokou škálu aplikací.

⁴<https://www.postgresql.org>

⁵Jazyky, které umožňují program strukturovat do procedur a funkcí, někdy též do objektů.

⁶<https://www.mongodb.com>

Kapitola 4

Specifikace požadavků

Účelem této kapitoly bude určit konkrétní požadavky na navrhovaný tréninkový simulátor dispečerů integrovaného záchranného systému. Kapitola bude shrnovat požadavky firmy, se kterou je v rámci této práce spolupracováno, tak požadavky na přidání prvků gamifikace.

Existující systém

V rámci konzultací mi byl představen systém, který se nyní běžně využívá pro zakládání reálných incidentů. Systém je primárně navržen pro volání na tísňovou záchrannou linku hasičů. Nejdříve zavolá civilista na číslo 150. Tento hovor se zobrazí v systému a přijme jej tzv. *call taker*. Úkolem *call taker*a je vyslechnout volajícího občana a zjišťovat dodatečné informace o incidentu. Tyto informace shlukuje následně do různých kategorií a poté je zadá do systému. V systému se vyplňují údaje, jako například kategorie a podkategorie incidentu, detailnější popis incidentu a vybírá výjezdové jednotky, které se mají k incidentu dostavit. Systém automaticky doplní místo incidentu ze zaslaných souřadnic z telefonu. V systému se také nachází mapa zobrazující polohu volajícího.

Vytvořený incident je předán operátorovi. Ten incident a jeho vyplněné informace překontroluje. Následně incident potvrdí a dojde k výjezdu jednotek.

4.1 Požadavky na LLM v roli civilisty

LLM bude v rámci simulace představovat civilistu volajícího na tísňovou linku. Úkolem modelu bude nahlásit incident uživateli. Bude tedy potřeba, aby model generoval realistický rozhovor s operátorem, dokázal odpovědět na dotazy operátora a to vše s přijatelnou latencí odpovědi. Požadavky na LLM jsou tedy následující:

- Model se drží své role, tedy role civilisty,
- model musí znát důležité informace o incidentu a zná informace o svém okolním prostředí,
- model dokáže korektně odpovídat na operátorovi otázky směřované jak na incident, tak na prostředí,
- latence odpovědi se pohybuje do 4 sekund.

4.2 Požadavky na prvky gamifikace

Prvky gamifikace jsou do simulátoru přidány, jelikož dokáží zvýšit motivaci a angažovanost dispečerů, což je kritické při zaučování nových operátorů. V rámci jedné simulace budou tedy kontrolovány a ohodnoceny následující prvky:

- Doba trvání zpracování incidentu,
- doba od přijmutí příchozího telefonátu,
- správnost vyplnění jednotlivých atributů incidentu.

Mezi atributy incidentu se řadí kategorie, podkategorie, název okresu a vesnice, typy výjezdových vozidel a přesnost vybraných souřadnic na mapě.

Pro zvýšení soutěživosti mezi uživateli simulátoru bude vytvořen žebříček simulací, kde bude možné porovnávat své výsledky simulací s ostatními. Bude zde také výběr několika obtížností simulace. Obtížnost bude spočívat převážně ve frekvenci a počtu příchozích hovorů. LLM bude mít také dle obtížnosti přiřazen charakter své postavy.

4.3 Ostatní požadavky a omezení

V rámci simulátoru bude role *call taker*a a operátora spojena do stejné role. Simulace musí probíhat v reálném čase, aby co nejlépe simulovala skutečnost a aplikace bude poskytovat uživatelské rozhraní pro vyřizování incidentů. Dále bude aplikace umožňovat vytvoření účtu a přihlášení se. Pouze přihlášení uživatelé mohou spouštět simulaci a umisťovat se v žebříčku výsledků.

Aplikace po konzultaci není potřeba nasazovat do produkčního prostředí. Tato práce a aplikace má sloužit jako „*Proof of concept*“, že podobná aplikace se dá vytvořit pro firemní účely.

Kapitola 5

Analýza a návrh webové aplikace

V této kapitole bude představen proces výběru vhodného předtrénovaného jazykového modelu pro roli civilisty hlásícího incident. Dále kapitola obsahuje návrh architektury, datový model, případy užití a další.

5.1 Porovnání a výběr LLM modelů

Výběr modelu probíhal rozsáhlým testováním předtrénovaných jazykových modelů od různých poskytovatelů. U modelů byla hlavně zkoumána kvalita odpovědí, rychlost odpovědi, cena a částečně i držení se role, konkrétně jestli, a jak moc, má model halucinace.

5.1.1 Tvorba testovacího scénáře

Na modelech bylo vyzkoušeno několik scénářů. V této práci je uveden pouze nejlépe fungující scénář, který vypadal následovně:

Použitý scénář:

Jsi civilista Karel Novák 43 let, který právě volá na hasičskou tísňovou linku a komunikuje s operátorem. Nacházíš se v Olomouci na Horním náměstí. Okolo sebe vidíš nějaké památky a spoustu lidí. Následně vidíš kouř a trochu plameny z budovy s číslem 143 a lidé z ní utíkají pryč. Máš znalosti průměrného občana, ale trochu stresuješ. Na otázky mimo incident neodpovídáš a nechápeš důvod těchto otázek. Odpovídej v 1-3 kratších větách. Odpovídej pouze na dotazované informace. Odpovídáš ve formě běžného hovoru. Snažíš se být nápomocný.

Operátor: Tísňová linka, jak Vám mohu pomoci ?

Scénář obsahuje šest klíčových vlastností tvorby promptu pro jazykový model[22, 19]:

- **Personu:** „Jsi civilista Karel Novák, 43 let. . .“
- **Kontext:** „Nacházíš se v Olomouci na Horním náměstí. Okolo sebe vidíš památky a spoustu lidí. . .“
- **Úkol:** „Nahlásit operátorovi incident, popsat kouř a plameny z budovy č. 143.“
- **Formát odpovědi:** „Odpovídej v 1–3 kratších větách. Odpovídej pouze na dotazované informace.“

- **Tón odpovědí:** „Odpovídej ve formě běžného hovoru. Snažíš se být nápomocný.“
- **Omezení:** „Na otázky mimo incident neodpovídáš a nechápeš důvod těchto otázek.“

Tyto vlastnosti scénáře jsou pro správné simulování civilisty modelem kritické, jelikož bez nich se modely silně odkláněly od simulování reality a své role.

5.1.2 GPT modely

Pro simulaci byly vyzkoušeny modely: GPT-3.5 Turbo, GPT-4 a GPT-4 Turbo. Modely byly vybrány z důvodu možnosti jejich promptování přes OpenAI API. Nejnovější modely, jako GPT-4.1, aj. zásadně testovány nebyly. Důvodem je jejich vysoká latence. Generování odpovědi na dotaz zde trvalo až v řádech desítek sekund, což je na simulaci v reálném čase nepřijatelné.

I když tyto modely poměrně kvalitně simulovaly roli civilisty, dalším aspektem vedoucím k jejich nevyužití je cena jejich provozu. Při používání modelů přes jejich rozhraní modely nevyužívají paměť a je nutné při každé otázce operátora zaslat celý předešlý kontext. Cena pro zpracování jednoho incidentu se poté vyšplhala až na ~20 Kč.¹

GPT-3.5 Turbo

Model zpočátku vypadal slibně. Latence odpovědi se pohybovala v řádu milisekund. Cenově je model také velmi dostupný. Problémy modelu nastaly při složitějším dotazování. Často generoval nereálné odpovědi a ve špatném formátu. Při dotázání se na otázku mimo kontext se neřídil svojí rolí. Tento model tedy nebyl vhodný pro simulaci.

GPT-4

U GPT-4 modelu nastalo výrazné zlepšení kvality generovaných odpovědí, a to i při složitějších scénářích. Latence se pohybovala kolem jedné až dvou sekund, tedy v přijatelném rozmezí. Model spolehlivě dodržoval definovanou roli civilisty a generoval realistické reakce. Oproti GPT-3.5 Turbo nabízí menší kontextové okno o 8 192 tokenech. To je ale pro potřeby simulace stále přijatelné. Model ale trpí halucinacemi.

GPT-4 Turbo

I když se jedná o cenově dostupnější model, kvalitativně se oproti GPT-4 jedná o zhoršení. Model se držel své role, ale kvalita odpovědí nebyla na takové úrovni. Model přidával k odpovědím různé dovětky, díky kterým se nedržel zadaného tónu. Odpovědi byly poskytovány rychleji, okolo 500 milisekund, a model též trpí halucinacemi.

5.1.3 Llama modely

Modely Llama vykazovaly velmi srovnatelné výsledky jako GPT modely. Kvalita odpovědí se lišila v závislosti na počtu parametrů modelu. Například Llama-3.1 se 70 miliardami parametrů se podobala výstupům GPT-4 Turbo. Llama modely je potřeba provozovat lokálně. Rychlost odpovědí tedy závisí na použitém hardwaru a na grafické kartě GeForce 1660-super

¹ Cena se vztahuje na model OpenAI-o3, kde zpracování jednoho milionu tokenů stojí ~250 Kč a generování jednoho milionu tokenů ~1000 Kč. V rámci zpracování incidentu by bylo odesláno 10 zpráv operátorem o velikosti 2500 a vygenerováno 1000 tokenů jako odpověď.

a 16 GB paměti RAM se odpovědi pohybovaly pro model se 70 miliardami parametrů v řádu sekund. Pro použití těchto modelů by bylo tedy potřebné provozovat aplikaci na externím serveru, což přidává značnou práci navíc při vývoji pro dosažení stejného výsledku jako u OpenAI API. Modely také trpěly halucinacemi.

5.1.4 Další modely a finální výběr

Modely jako Perplexity, Gemini nebo Gemma vykazovaly velmi podobné výsledky. Všechny odzkoušené modely trpěly halucinacemi. Pro jejich eliminaci by bylo potřeba modely přetrénovat způsoby popsány v kapitole 2. Zde je problém získat korektní dataset. Telefonáty na tísňovou linku jsou uschovány u složky, která hovor přijímá, a nejsou volně přístupné v souladu s §30 odstavec 2 zákona číslo 320/2015 Sb., o Hasičském záchranném sboru[1]. Proto bylo tedy možné halucinace modelu v rámci simulace připustit a byl vybrán model GPT-4.

5.2 Návrh

Tato sekce se zabývá návrhem webové aplikace sloužící k simulaci tísňového hovoru mezi operátorem a velkým jazykovým modelem v roli civilisty. Cílem systému je snaha o co nejlepší napodobení reálných incidentů, která umožní trénování a přípravu nových dispečerů. Návrh vychází z požadavků popsanych v kapitole 4.

5.2.1 Uživatelské rozhraní

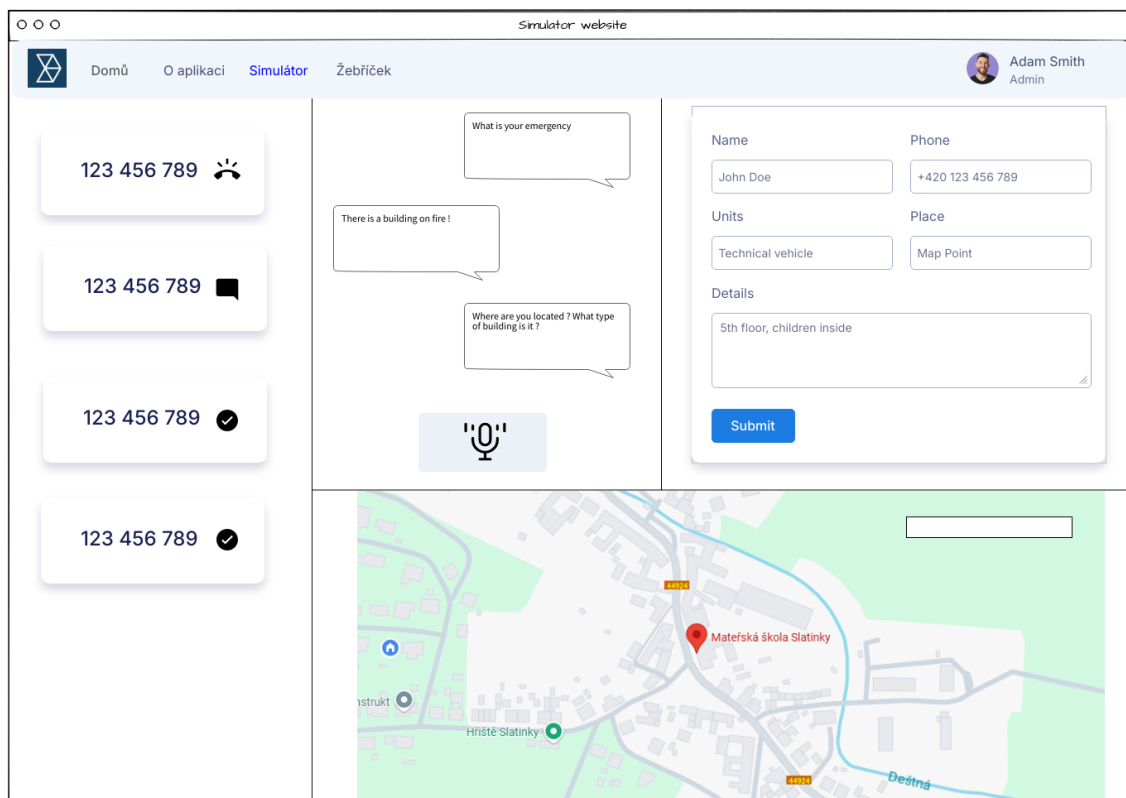
Uživatelské rozhraní webové aplikace je navrženo se snahou o jednoduchost, přehlednost a snadné používání. Návrh celého rozhraní byl inspirován již existující aplikací, která je popsána v kapitole 4 v podsekcí „*Existující systém*“. Důvodem pro podobný návrh je schopnost nových operátorů zorientovat se poté v reálném systému.

V rámci návrhu uživatelského rozhraní vznikly 2 wireframy. Wireframy slouží k definování základní struktury a zobrazení funkčních prvků. Poskytují plán rozvržení a funkčnosti se zaměřením na to, jaké prvky rozhraní budou existovat na klíčových stránkách.

Wireframe simulace

Na obrázku 5.1 je návrh nejdůležitější stránky celé aplikace. Zobrazuje simulaci tísňového hovoru s možností vytvoření incidentů. V horní části je pevně umístěný navigační panel s odkazy na domovskou stránku, simulátor a žebříčky a se stavovým řádkem přihlášeného uživatele. Tento panel je shodný pro všechny dostupné stránky aplikace. Pod navigačním panelem je levý panel pro výběr příchozího hovoru.

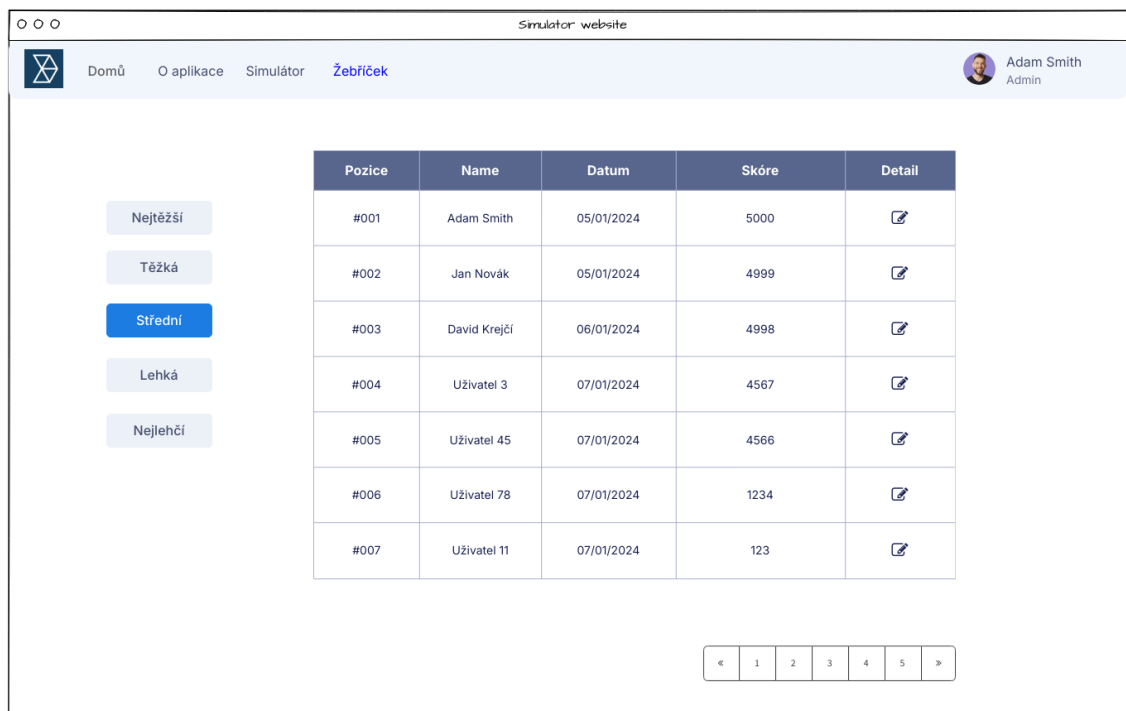
Ve střední části obrazovky se nachází hned 3 důležité sekce. První z nich slouží pro komunikaci s LLM v roli civilisty. Druhá sekce, nacházející se vpravo od komunikační sekce, vyznačuje formulář, který musí operátor vyplnit informacemi získanými z komunikace. Poslední sekce, vyskytující se pod oběma předešlými sekcemi, obsahuje mapu, která zobrazuje umístění incidentu a jeho okolí.



Obrázek 5.1: Wireframe stránky simulace.

Wireframe žebříčku výsledků

Na obrázku 5.2 je vidět návrh stránky žebříčku skóre, která umožňuje dispečerům porovnat své výsledky v různých úrovních obtížnosti. Výsledky lze procházet pomocí stránkování.

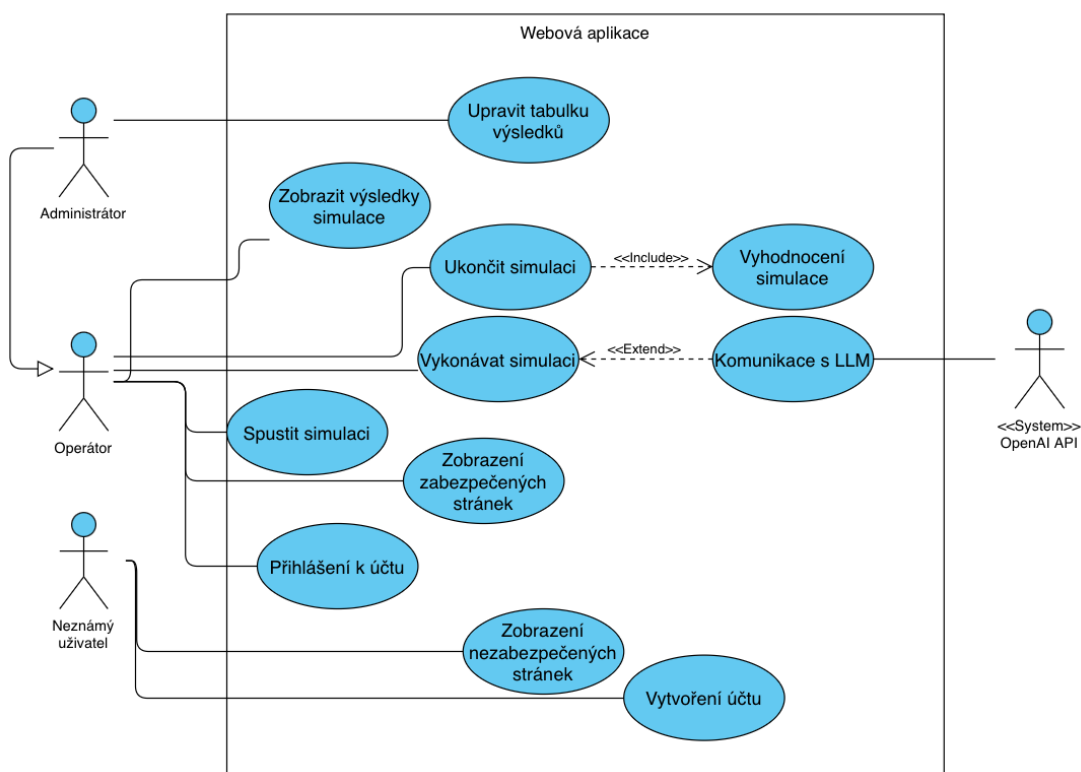


Obrázek 5.2: Wireframe stránky žebříčku výsledků.

Do wireframů bylo přidáno trochu designu, aby byla jednodušší implementace. Tyto dvě stránky jsou pilířem celé aplikace. V aplikaci se nachází také stránka pro přihlášení a registraci uživatele.

5.2.2 Role v systému

V systému se objevují 3 role, kde první z nich je operátor. Operátor je běžný uživatel systému, kterému je po přihlášení umožněno spouštět simulaci, vyřizovat incidenty, komunikovat s LLM a jeho simulace jsou hodnoceny. Druhou rolí je administrátor. Ten má umožněny všechny funkce operátora. Dále je obohacen o možnost spravovat žebříček výsledků, aj. Poslední roli zde zastává neznámý uživatel. Ten má možnost procházet nezabezpečené stránky a může se v případě zájmu registrovat. Detailnější výčet funkcí a oprávnění je zobrazen na obrázku 5.3.

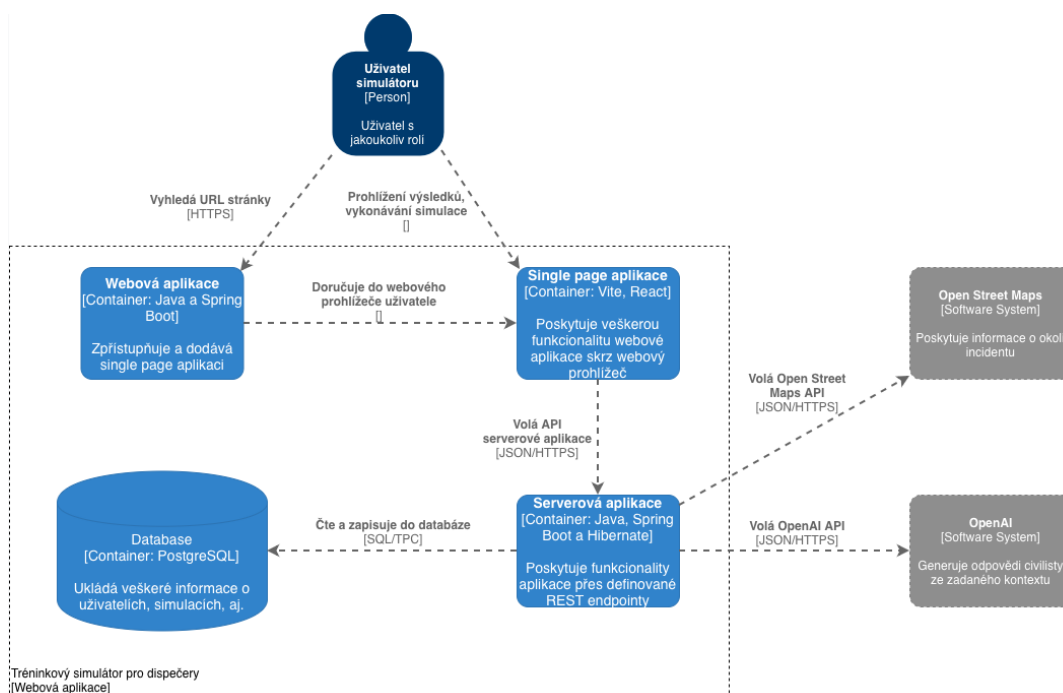


Obrázek 5.3: Příklad užití jednotlivých rolí v aplikaci.

5.2.3 Architektura systému a zvolené technologie

Dle požadavků z kapitoly 4 byl systém rozčleněn na několik menších celků. Rozdělení je zobrazeno v diagramu na obrázku 5.4, kde je znázorněna architektura systému pomocí C4 modelu [24] druhé úrovně, tedy diagramu kontejnerů. U jednotlivých kontejnerů jsou také uvedeny využívané technologie, které byly vybrány díky své robustnosti, popularitě a jednoduché práci s nimi.

Komponenta *Uživatel simulátoru* reprezentuje jakoukoliv osobu využívající systém. Představuje tedy administrátora, operátora a nepřihlášeného uživatele.



Obrázek 5.4: C4 model navržené architektury systému.

Webová aplikace

Kontejner *Webová aplikace* zde představuje server *Apache Tomcat*², který je v základu využíván frameworkem Spring Boot³ postaveným na programovacím jazyce Java. Tento webový server servíruje na úrovni statického obsahu veškeré *HTML*, *CSS* a *HTML*. Kombinace těchto souborů tvoří Single-page aplikaci (dále *SPA*), která je zpřístupněna na kořenové adrese. Spring Boot poskytuje soubory ze složky `src/main/resources/static/` a v případě nasazení není potřeba konfigurovat další webový server.

Single-page aplikace

Klientská část systému je postavena jako SPA využívající React⁴ pro deklarativní a komponentově orientovaný přístup k tvorbě uživatelského rozhraní. Pro rychlé spuštění a okamžité obnovení změn kódu během vývoje byl využit lokální server implementovaný nástrojem Vite⁵, který umožňuje optimalizovanou kompilaci na vyžádání. Pro výsledné spuštění aplikace se spustí kompilace, která sestaví finální balíček statických zdrojů, a kde je následně balíček umístěn do adresáře uvedeném v předchozí podsekci 5.2.3.

Ve vizuálním stylu hraje klíčovou roli Bootstrap společně s obalem react-bootstrap. Tato kombinace zajišťuje responzivní design, konzistentní vzhled napříč aplikací a poskytuje některé předpřipravené komponenty. V případě nedostatečnosti bootstrapu byl použit i kaskádový styl. Ke zobrazení a práci s mapou byla využita knihovna Leaflet. Ta dovoluje vkládat interaktivní mapové komponenty s podporou přiblížení, ukazatele místa a vlastního stylu mapy.

²<https://tomcat.apache.org>

³<https://spring.io/projects/spring-boot>

⁴<https://react.dev>

⁵<https://vite.dev>

SPA je napsána plně v TypeScriptu⁶. To oproti klasickému JavaScriptu přináší statické typování, vylepšené refaktorování a lepší dohled nad voláním API. Build a linting procesu asistuje ESLint⁷ s pluginy pro React a React Hooks, které kontrolují syntaxi, dodržování pravidel hooks a odhalují potenciální chyby.

Serverová aplikace

Aplikační logika je zpracována taktéž pomocí Spring Boot. Klíčové závislosti jsou dodány díky nástroji Gradle⁸. Jednou z důležitých závislostí je Spring Boot Starter. Ta umožňuje tvorbu REST rozhraní a běh aplikace. Další kritickou závislostí je Spring Security, díky které je umožněno zabezpečení vybraných endpointů a obecně celkové aplikace, a to využitím předpřipravených a ověřených principů, které budou upraveny pro potřeby autentizace a autorizace JSON Web Tokenem (dále *JWT*). Z neposledních důležitých závislostí je zde závislost Spring Boot Java Persistence API (dále *JPA*), kde detaily komunikace jsou popsány v následující sekci. Pro parsování volně strukturovaných dat se používá Jsoup⁹ a pro redukci opakovatelného kódu knihovna Lombok¹⁰. Aplikace také implementuje 3 integrace na externí systémy. Detail těchto integrací je popsán v sekci 6.1.2.

Konfigurací projektu je vytvoření jednoduchého Gradle skriptu, který zajistí vytvoření jediného spustitelného *.jar* souboru, do něhož se během buildu zkompiluje SPA a následně spustí aplikační server.

Databáze

Centrálním úložištěm všech dat je relační databáze PostgreSQL, ke které se serverová aplikace připojuje pomocí Java Database Connectivity (dále *JDBC*) ovladače. JDBC umožňuje nastavení počtu současných spojení, aj. Díky objektově relačnímu mapování (Object-relational mapping, dále *ORM*) od Hibernate¹¹ jsou třídy mapovány na jednotlivé tabulky databáze a Java Persistence Query Language (dále *JPQL*) dotazy jsou překládány do nativního SQL. Hibernate také nabízí možnost validace hodnot ještě před jejich uložením.

Pro migraci databáze je využito Liquibase¹², kde je umožněno definování celé struktury databáze přes konfigurační soubory. Změny ve struktuře jsou díky Liquibase možné aplikovat za běhu aplikace, a to bez ztráty dat. Konkrétní entity a struktura dat jsou popsány v 5.2.5.

5.2.4 Autorizace, autentizace a zabezpečení rozhraní

K zajištění autentizace je využit bezstavový mechanismus JWT. Token obsahuje tři části, od sebe oddělené tečkou:

- Hlavička (**header**): zde je uveden druh tokenu a algoritmus jeho podpisu
- náklad (**payload**): obsahuje rezervované informace jako čas expirace, vydavatele nebo jsou zde uvedeny uměle vytvořené informace

⁶<https://www.typescriptlang.org>

⁷<https://eslint.org>

⁸<https://gradle.org>

⁹<https://jsoup.org>

¹⁰<https://projectlombok.org>

¹¹<https://hibernate.org>

¹²<https://www.liquibase.com>

- a podpis (**signature**).

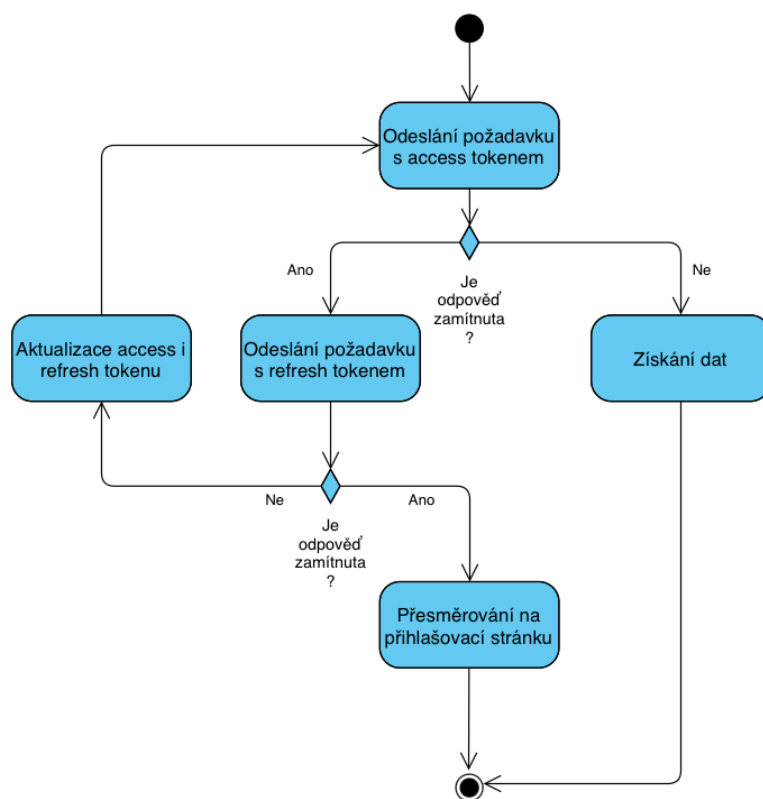
Všechny tři části jsou zakódovány pomocí Base64Url [11].

Pokud je uživatel registrovaný, má možnost se přihlásit do aplikace svým jménem a heslem. Po ověření přihlašovacích údajů jsou uživateli vystaveny dva časově omezené tokeny, a to přístupový (dále *access*) token a obnovovací (dále *refresh*) token. Oba tokeny mají přidány ve svém payloadu unikátní jméno uživatele.

Oba tokeny se posílají v HTTP hlavičce v atributu autorizace s předponou *Bearer* a společně se zbytkem požadavku. Spring Security každý požadavek na zabezpečený endpoint předá bezpečnostnímu filtru, ten načte JWT token, rozšifruje pomocí klíče, zkontroluje podpis a expiraci. Pokud jsou všechny údaje platné, je uživateli umožněn přístup na zabezpečený endpoint.

Access token se využívá jako první možnost autentizace a posílá se s každým požadavkem na API. Jeho expirace se obvykle pohybuje v řádu minut. Refresh token má na druhou stranu dobu expirace delší. Obvykle v řádu hodin. Po expiraci access tokenu server odpoví, že je token neplatný a znemožní přístup. V tento moment se na server odešle refresh token. Pokud je ten platný, obdrží uživatel opět oba nové tokeny, stejně jako při přihlášení. Toto schéma je znázorněno na obrázku 5.5. Pokud expiroval i refresh token, je uživatel znovu vyzván k přihlášení.

Jelikož se access token odesílá častěji, je zde větší šance na jeho odcizení. Útočník by ale získal přístup pouze na pár minut. Aby se ale uživatel nemusel tak často přihlašovat, vznikl proto refresh token, který se posílá méně často a šance na odcizení je nižší.



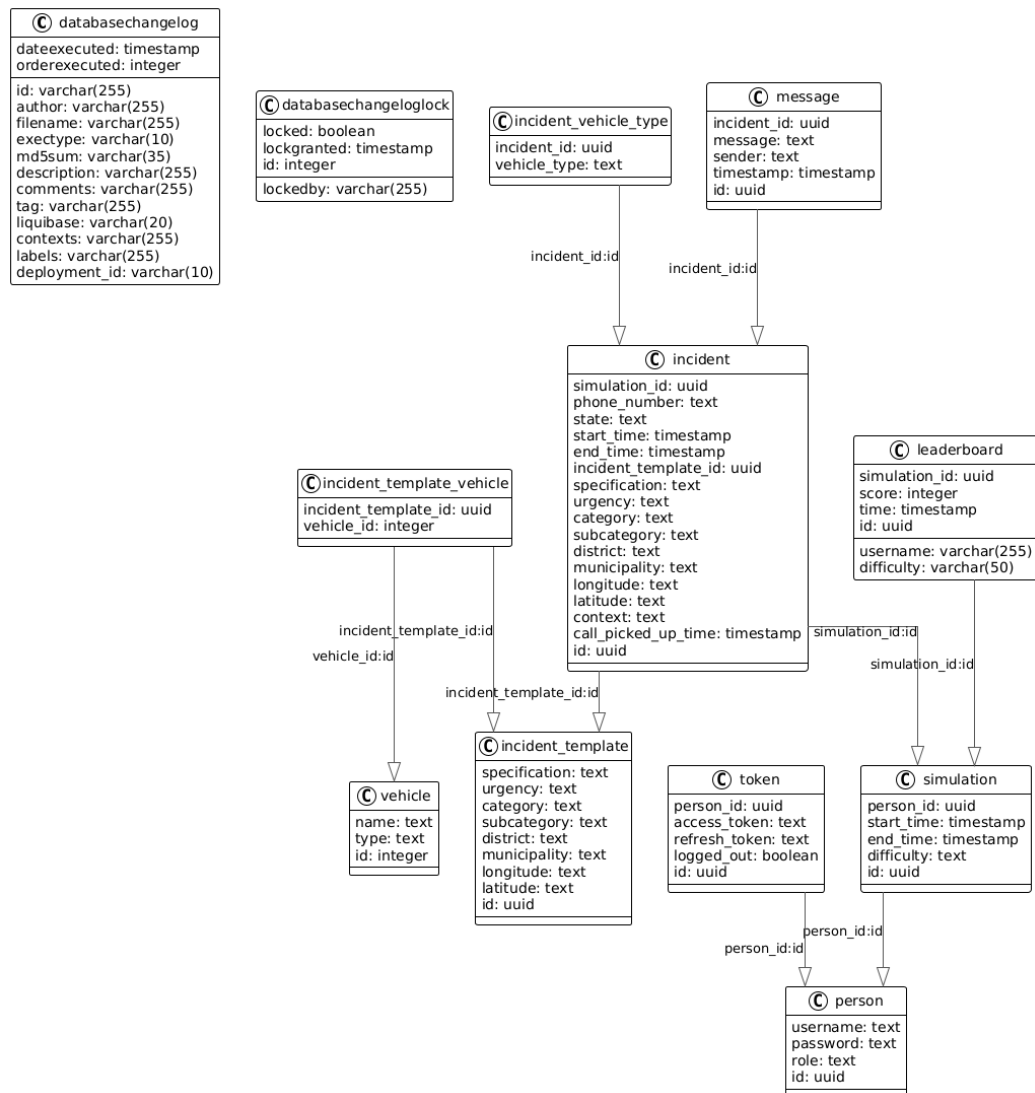
Obrázek 5.5: Stavový digram popisující autentizaci uživatele v aplikaci.

Tokeny by se ideálně měly ukládat v cookie s příznakem HTTP-only, se kterými nedokáže JavaScript pracovat. Pro potřeby menších aplikací obvykle postačí uložení v paměti webového prohlížeče i přes větší riziko odcizení obou tokenů zároveň. Pokud je token poslán na nezabezpečený endpoint, je JWT ignorován. Pro šifrování je zvolen vygenerovaný klíč, který je uložen v konfiguračním souboru aplikace. Taktéž jsou zde uvedeny doby expirace tokenů.

Tokeny umožňují i autorizaci uživatele, jelikož obsahují v payloadu unikátní identifikátor uživatele. Díky němu je možné najít uživatele a jeho roli v databázi. Pomocí role se pak aplikace rozhodne, jestli má uživatel dostatečná práva na vykonání dané činnosti.

5.2.5 Struktura databáze

Technologie využívané pro práci s databází byly popsány v předchozí sekci 5.2.3. Tato sekce se bude věnovat podrobnějšímu popisu a vysvětlení jednotlivých tabulek databáze. Databázové schéma a relace mezi tabulkami jsou popsány na obrázku 5.6.



Obrázek 5.6: ER diagram zobrazující strukturu databáze.

Správa migrací

První tabulka „**databasechangelog**“ slouží k uchování historie všech změn schématu. Každý řádek odpovídá jednomu setu změn a eviduje, kdy a jak se změna aplikovala, a eviduje také kontrolní součet, který vzniká z definičních souborů schématu. Tabulka „**databasechangeloglock**“ zajišťuje, aby nedocházelo ke kolizi migrací při spuštění aplikace ve více instancích. To zajišťuje tak, že určí, kdo migrace drží, a jakoukoli odchylku od držené migrace Liquibase nepřipustí.

Uživatel a JWT tokeny

Tabulka „**person**“ reprezentuje uživatele systému. Kromě unikátního identifikátoru, který je primárním klíčem každé tabulky v databázi, obsahuje tabulka také unikátní přezdívku uživatele, heslo a roli zastávanou v systému, tedy operátora nebo administrátora. Role určuje oprávnění na akce, které následně mohou přihlášení uživatelé vykonávat.

Jeden řádek tabulky „**token**“ reprezentuje jeden JWT token, který se používá pro autorizaci a autentizaci v aplikaci. Uchovávají se zde jak access, tak refresh tokeny a u každého je uveden stav, jestli je platný, a datum vydání. Obě tabulky jsou využívány v rámci Spring Security.

Šablony tvorby incidentu

Tabulka „**incident_template**“ slouží k uchování zpracované a upravené datové sady, která byla poskytnuta spolupracující firmou. Datová sada obsahuje reálné incidenty IZS, které byly poupraveny a anonymizovány. I přes tyto úpravy konkrétní hodnoty obsažené v této tabulce nesmí být zveřejněny. Tabulka obsahuje atributy jako kategorii, podkategorii, informace o lokaci incidentu a strohý popis, co se událo.

Tabulka „**vehicle**“ reprezentuje výjezdová vozidla zaslaná k incidentu. Nejedná se pouze o pojízdná vozidla, ale i o různé kontejnery a přívěsy, a jedná se o další část poskytnutého datasetu. V tabulce se nachází jméno vozidla a typy vozidel, které reprezentují zkratky hasičské techniky [13]. Jelikož se jedná o asociaci mnoha-ku-mnoha (M:N), vznikla mezivazební tabulka s názvem „**incident_template_vehicle**“.

Tyto dvě tabulky slouží jako šablona pro tvorbu incidentů a jejich data se následně využívají k ohodnocení incidentu.

Simulace

Pro ukládání informací o simulaci byla vytvořena tabulka „**simulation**“. Nachází se zde atributy jako začátek a konec simulace, obtížnost simulace a je asociována s tabulkou uživatelů a tabulkou vygenerovaných incidentů.

„**incident**“ tabulka reprezentuje vygenerované incidenty v rámci simulace z šablony incidentů. Taktéž obsahuje časové údaje o začátku, konci a době trvání, než se začal incident zpracovávat. Obsahuje také mnoho stejných atributů, jaké jsou uvedeny v „**incident_template_vehicle**“ tabulce. Zde ale reprezentují operátorem zadané informace. Velmi důležitým atributem je kontext. Zde je uchováván vygenerovaný scénář pro předání kontextu LLM. Tabulka je v asociaci 1:N s „**incident_vehicle_type**“, která uchovává operátorem zvolená vozidla.

Poslední tabulka využívaná v rámci průběhu simulace je „**message**“. Zde každý záznam reprezentuje jednu zprávu, ať už od operátora, tak od LLM. Uchovává také časové razítko

odeslání zprávy. Tyto tabulky umožňují průběh simulace a výsledky jsou uchovávány v tabulce „**leaderboard**“, která obsahuje atributy jako finální ohodnocení nebo vykonavatele simulace.

Kapitola 6

Implementace

Následující kapitola popisuje implementaci návrhu z kapitoly 5. Vzhledem k rozsahu aplikace zde budou vybrány pouze implementačně nejzajímavější a nejdůležitější části aplikace. Nejdříve se kapitola zaměří na serverovou část a následně na SPA, která zde představuje klientskou část. Ukázky výsledné aplikace jsou v příloze A.

6.1 Serverová část

Jak již bylo řečeno v podsekcí 5.2.3, hlavní využívanou technologii zde zastává Spring Boot, díky které byla umožněna tvorba endpointů na bázi REST. Tyto endpointy slouží ke komunikaci s SPA. Jelikož architektura REST je bezstavová, je celý životní cyklus aplikace řízen z klientské části a server pouze dodává potřebná data nebo případně synchronizuje stav entit s klientskou částí.

Každá Spring Boot aplikace je navržena podle pevně dané architektury, nazývanou *Spring MVC*, která jasně odděluje prezentační vrstvu pomocí kontrolerů (anotace *@RestController*), aplikační logiku (anotace *@Service*) a perzistentní vrstvu (anotace *@Repository*). Tato architektura dokáže být robustní, co se jednoduchých úloh týče. Každopádně její silnou stránkou je konzistence mezi projekty a silné dodržování principu jediné odpovědnosti (*Single Responsibility Principle*) [26]. Komunikace mezi jednotlivými vrstvami je zobrazena na obrázku 6.1. Nutno podotknout, že kontroler také transformuje JSON nebo XML formáty na objekty jazyka Java a dále v aplikaci se pracuje pouze s těmito objekty.



Obrázek 6.1: Architektura frameworku Spring Boot.

6.1.1 Endpointy

Pro obsluhu REST požadavků z klientské SPA je potřeba vytvořit endpointy, které jednotlivé požadavky obslouží a vygenerují korespondující odpověď.

Tvorba endpointů

Nejdříve je nutné použít anotaci `@RestController`. Spring Boot využívá princip automatického vkládání kontextu a závislostí (*contexts and dependency injection*, dále *CDI*). Díky této anotaci je celý životní cyklus třídy řízen frameworkem. Tudíž tvorba instance, dodání v momentě potřeby i její zánik. Anotace `@RequestMapping` určuje prefix všech endpointů uvedených ve třídě a `@PostMapping` určuje, pro jakou HTTP metodu je endpoint určen a na jaké URL se má dotaz poslat. Ukázka takové třídy je k dispozici ve výpisu 6.1.

`@RequiredArgsConstructor` je speciální Lombok anotace a má na starosti redukovat opakovatelný Java kód, který by zbytečně zahlcoval třídu a zhoršil její čitelnost. Zde konkrétně anotace přidá všechny proměnné do konstruktoru a umožňuje tak CDI.

```
1 @RestController
2 @RequestMapping("api/simulation/")
3 @RequiredArgsConstructor
4 public class IncidentApi {
5
6     private final IncidentService incidentService;
7
8     @PostMapping("/{simulationId}/incident")
9     public Incident createIncident(@PathVariable UUID simulationId) {
10         return incidentService.create(simulationId);
11     }
12
13     @PostMapping("/incident/{incidentId}/save")
14     public Incident saveIncident(@PathVariable UUID incidentId, @RequestBody Incident
15         incidentData) {
16         return incidentService.saveIncident(incidentId, incidentData);
17     }
18
19     @PostMapping("/incident/{incidentId}/picked-up")
20     public void incidentPickedUp(@PathVariable UUID incidentId) {
21         incidentService.incidentPickedUp(incidentId);
22     }
23 }
```

Výpis 6.1: Příklad definice tří jednoduchých endpointů, které se starají o zachycení a obsluhu požadavků pro jeden incident v simulaci.

Zabezpečení endpointů

Zabezpečení endpointů a obecně celé aplikace je zajištěno pomocí Spring Security, která je integrována přímo v řetězci zpracování HTTP požadavku. Spring Security nastavuje výchozí chování při tomto zpracování. Toto chování je zobrazeno v ukázce 6.2. Tato konfigurace zabezpečení ruší výchozí CSRF ověřování, jelikož není uchováván na serveru stav. Dále můžeme v nastavení nalézt nastavení bezstavového režimu autentizace, který vylučuje použití HTTP session, a místo něj se spoléhá na JWT tokeny. Pro zpracování JWT tokenů je zde uveden odkaz na filtr, který určuje validitu tokenů a extrahuje informace z payloadu. Dále je zde určeno, že veškeré endpointy začínající řetězcem `"/api/simulation/"` budou zabezpečeny a bez autentizace nebude server požadavky obsluhovat.

```

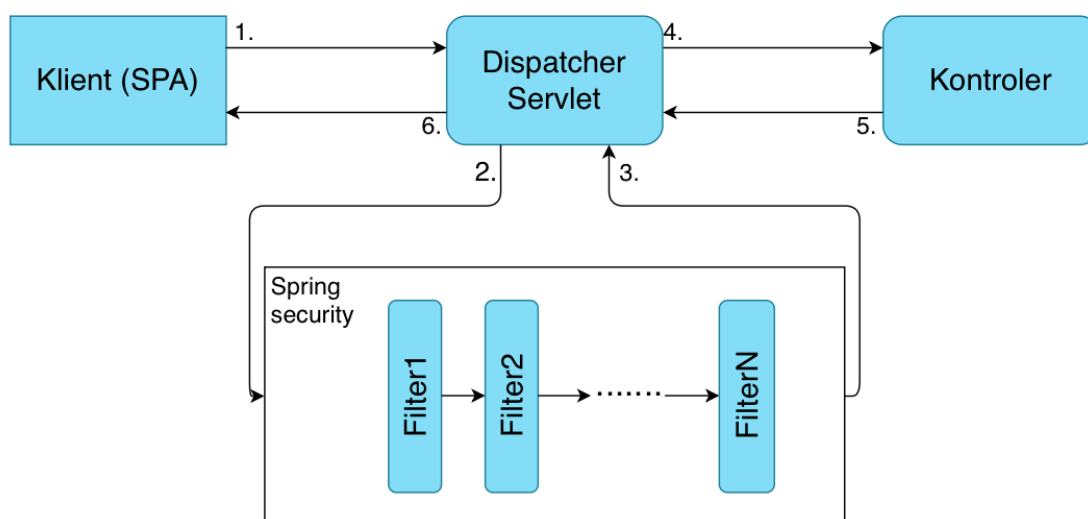
1 @Bean
2 public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
3     return http
4         .csrf(AbstractHttpConfigurer::disable)
5         .cors(c -> c.configurationSource(corsConfigurationSource()))
6         .authorizeHttpRequests(req -> req
7             .requestMatchers("/api/simulation/**").authenticated()
8             .anyRequest().permitAll()
9         )
10        .userDetailsService(userDetailsService)
11        .exceptionHandling(err -> err
12            .accessDeniedHandler(accessDeniedHandler)
13            .authenticationEntryPoint(new HttpStatusEntryPoint(HttpStatus.
14                UNAUTHORIZED))
15        )
16        .sessionManagement(session -> session
17            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
18        )
19        .addFilterBefore(tokenAuthenticationFilter,
20            UsernamePasswordAuthenticationFilter.class)
21        .logout(logout -> logout
22            .logoutUrl("/api/auth/logout")
23            .addLogoutHandler(logoutHandler)
24            .logoutSuccessHandler((request, response, authentication)
25                -> SecurityContextHolder.clearContext())
26        )
27        .build();
28 }

```

Výpis 6.2: Příklad definice tří jednoduchých endpointů, které se starají o zachycení a obsluhu požadavků pro jeden incident v simulaci.

Zpracování požadavků

Jakýkoliv požadavek ze SPA je zachycen tzv. *Dispatcher servletem*. Tento servlet přijme HTTP požadavek a předá ho do řetězce filtrů. V tomto řetězci dojde ke zjištění validity JWT tokenu a v případě selhání validity dojde k zamítnutí požadavku. V případě úspěchu se vrací požadavek zpět do Dispatcher servletu společně s informacemi o přihlášeném uživateli a jeho pravomocemi. Ten nyní vyhledá korespondující metodu, která se stará o zpracování požadavku, v kontrolerech podle URL. Pokud je požadavek směřován na nezabezpečený endpoint, filtr validující JWT token není uplatněn. Zjednodušené schéma zpracování požadavku je zobrazeno v 6.2.



Obrázek 6.2: Zpracování HTTP požadavku využitím bezpečnostních filtrů.

6.1.2 Simulace

Tato sekce popisuje jádro aplikace, tedy průběh a implementaci celé simulace. Všechny endpointy spojené se simulací jsou zabezpečeny. Důvodem je, aby nepřihlášený uživatel nemohl provádět simulaci. Spuštění simulace je provedeno zavoláním endpointu `"/api/simulation/start"`. Zde dojde k vytvoření simulace v databázi, napojením simulace na přihlášeného uživatele a k uložení důležitých atributů. Informace o uživateli jsou získány z JWT tokenu.

Tvorba incidentů

Ze SPA nyní chodí v různých časových intervalech požadavky na vytvoření incidentu. Rozptyl požadavků je náhodný a jejich počet závisí na obtížnosti simulace. Za vytvoření incidentu je zodpovědný endpoint `"/api/simulation/simulationId/incident"`, kde díky unikátnímu identifikátoru dojde k napojení incidentu na simulaci. Následně dojde k výběru náhodného šablonového incidentu, který slouží k poskytnutí potřebných informací incidentu.

Po vytvoření incidentu jako objektu následuje komunikace se dvěma externími službami. Tato komunikace je potřeba pro zjištění dodatečných informací k incidentu. Jedná se primárně o informace spojené s okolím incidentu.

První službou jsou *Open Street Maps*¹ (dále OMS). Tato služba poskytuje, na základě zeměpisných souřadnic, informace o okolních objektech do předem definované vzdálenosti. Služba poskytuje dotazovací rozhraní, se kterým je komunikováno pomocí klientské HTTP knihovny *RestTemplate*, která je součástí modulu *Spring Web*. Pro dotázání se na správná data je vytvořen dotaz pro tuto službu. Dotaz je zobrazen ve výpisu 6.3. Dotaz je konstruován v takovém formátu, kde se kolem souřadnic do definované vzdálenosti získají všechny body mapy. Následně se na těchto bodech dojde k vyhledání informací o budovách, řekách, silnicích, obchodech, aj. Poslední dvě části dotazu v mapě vyhledávají, na jakém území se souřadnice nachází z pohledu využití lidstvem (obytná zóna, atd.) a na jakém přírodním typu (pole, les, atd.).

¹<https://overpass-api.de/api/interpreter>


```

1 node(around:75,49.54871346667767,17.088198074587726);
2 (
3 way(around:75,49.54871346667767,17.088198074587726)[building];
4 way(around:75,49.54871346667767,17.088198074587726)[waterway];
5 way(around:75,49.54871346667767,17.088198074587726)[highway];
6 way(around:75,49.54871346667767,17.088198074587726)[leisure];
7 way(around:75,49.54871346667767,17.088198074587726)[amenity];
8 way(around:75,49.54871346667767,17.088198074587726)[shop];
9 way(around:75,49.54871346667767,17.088198074587726)[public_transport];
10 );out body;>;out skel qt;
11
12 is_in(49.54871346667767,17.088198074587726);
13 area._[landuse];
14 out body;>;out skel qt;
15
16 is_in(49.54871346667767,17.088198074587726);
17 area._[natural];
18 out body;>;out skel qt;

```

Výpis 6.3: Příklad definice tří jednoduchých endpointů, které se starají o zachycení a obsluhu požadavků pro jeden incident v simulaci.

Odpovědí na tento dotaz je strukturovaný JSON. Tento JSON je následně zpracován a přetvořen do Java objektu *NearbyFeatures*. Ten obsahuje kolekce objektů, které jsou rozlišeny typem nalezených objektů. Například v jedné kolekci je seznam nalezených budov s jejich atributy, v další jsou nalezené řeky, atd. U každého typu objektu jsou uchovávány různé informace. U budov například počet pater, souřadnice, počet bytů nebo identifikátor budovy v Registru územní identifikace, adres a nemovitostí (dále *RÚIAN*)². Následně je nad objektem spuštěn algoritmus, který dopočítá vzdálenosti objektů od vyhledaných souřadnic.

Druhá integrace slouží k získání čísla budov. Využívá opět RestTemplate knihovny. RÚIAN sice poskytuje veřejně dostupné rozhraní, ale jedná se o SOAP, ke kterému je pro zprovoznění potřeba registrace a dalších složitých kroků. Byl proto zvolen přístup, kde se pošle dotaz přímo na zdrojovou HTML stránku s informacemi o budově skrz identifikátor. Výsledná stránka je následně zpracována pomocí Jsoup a hledané číslo budovy je nalezeno a uloženo k objektu budovy.

Po získání informací ohledně okolí je vytvořen kontext. Kontext slouží pro komunikaci s LLM a obsahuje charakter postavy v závislosti na obtížnosti simulace, přírodní okolí incidentu, pokyny pro LLM a další. Vytvořený incident je poté uložen a je poslán zpět do SPA. Příklad vytvořeného kontextu je uveden v následujícím scénáři:

²<https://cuzk.gov.cz/ruian/RUIAN.aspx>

Použitý scénář:

Jsi civilista Karel Novák 43 let, který právě volá na hasičskou tísňovou linku. Jsi průměrný civilista, který je chytrý, ale trochu stresuješ. Neříkej přesné vzdálenosti. Vzdálenosti jsou uvedeny v metrech. Chovej se jako klasický občan volající na tísňovou linku. Neuvedené informace si domysli. Odpovídej v 1-3 kratších větách. Anglické názvy pocházejí z Open Street Maps. Nemluv anglicky a překládej. Odpovídej pouze na dotázané informace. Pokud je uvedena budova, incident se udál v té nejbližší. Detail incidentu které nesmíš říct. Kategorie: Záchrana osob a zvířat, podkategorie: AED. Tyhle musíš, ale pouze pokud jsi na ně tázán: okres: Prostějov, vesnice/město: Slatinky, co se stalo: nedýchá Tvé okolí vypadá následovně: Budovy: Budova 1: typ: rodinný dům počet bytů: 1 počet pater: 2 číslo domu: 208 vzdálenost: 4 m. Budova 2: typ: rodinný dům počet bytů: 1 počet pater: 2 číslo domu: 206 vzdálenost: 39 m. Budova 3: typ: rodinný dům počet bytů: 1 počet pater: 1 číslo domu: 209 vzdálenost: 40 m. Vozovky: typy silnic: residential (5×), path (1×) povrchy cest: (1×) typy cest (track): (6×) Využití místa na kterém stojí lidmi: způsob využití: residential (1×)

Komunikace s LLM

Pro zprostředkování dialogu s modelem v roli civilisty je vyhrazen endpoint `"/api/simulation/incident/message"`, který přijímá objekt reprezentující zprávu od dispečera a vrací zpět novou zprávu od LLM. Aplikační logika pro tento požadavek je zpracována v souboru *MessageService*. Celý proces probíhá ve dvou hlavních krocích. Nejprve se uživatelská zpráva uloží do databáze a následně se na základě historie konverzace a kontextu vygeneruje požadavek pro LLM.

Inicializace HTTP klienta probíhá ve třídě *OpenAIClient*, kde se při vytvoření instance této třídy v konstruktoru vytvoří reaktivní klient. Tento klient obsahuje přednastavenou URL adresu a hlavičku pro autorizaci, která obsahuje API klíč potřebný pro správné fungování integrace. API klíč a URL integrace jsou načteny z konfiguračního souboru pomocí anotace *@Value*. Díky této konfiguraci lze libovolně přepínat mezi různými velkými jazykovými modely poskytovanými OpenAI.

Přes tohoto reaktivního klienta jsou následně posílány zprávy pro LLM. Do modelu se posílá textový řetězec, který obsahuje na začátku vytvořený kontext a pod ním jsou obsaženy všechny předešlé zprávy poslané jak uživatelem, tak LLM. To je rozlišeno korespondující předponou před zprávou. Následně se celý řetězec odešle. Po obdržení odpovědi od LLM je nová zpráva uložena do databáze ke korespondujícímu incidentu. Tento postup zajišťuje zaznamenání celé historie dialogu a operátor má tedy pocit, že komunikuje s civilistou.

Implementace této aplikační logiky je zobrazena ve výpisu 6.4. Důležitá je zde anotace *@Service*. Ta umožňuje správné fungování CDI a životní cyklus instance objektu je tedy spravován automaticky. *@Slf4j* slouží pro vložení závislosti *Loggeru*³ a možnosti vypisování zpráv za běhu programu. Poslední důležitou anotací je anotace *@Transactional*, kde její funkcionality je popsána v 6.1.3.

³Objekt, který slouží pro výpis zvolených hodnot na standardní výstup za běhu programu. Často obsahuje více úrovní výpisu. Například informační, varovací nebo chybový.

```

1 @Service
2 @Slf4j @RequiredArgsConstructor
3 public class MessageService {
4     private final MessageRepository messageRepository;
5     private final IncidentRepository incidentRepository;
6     private final OpenAiClient openAiClient;
7
8     @Transactional
9     public Message create(Message message) {
10         Incident incident = incidentRepository.findById(message.getIncident().getId())
11             .orElseThrow(() -> new EntityNotFoundException(Incident.class, message.
12                 getIncident().getId()));
13
14         message.setIncident(incident);
15         messageRepository.save(message);
16
17         List<Message> history = messageRepository.findAllByIncident_IdOrderByTimestampAsc(
18             incident.getId());
19         List<Map<String, String>> gptMessages = new ArrayList<>();
20         gptMessages.add(Map.of("role", "system", "content", incident.getContext()));
21
22         for (Message msg : history) {
23             gptMessages.add(Map.of(
24                 "role", msg.getSender() == Sender.USER ? "user" : "assistant",
25                 "content", msg.getMessage()));
26         }
27
28         String aiReply = openAiClient.askModel(gptMessages)
29             .doOnError(err -> log.error("GPT error:", err))
30             .doOnNext(reply -> log.info("GPT reply: {}", reply)).block();
31
32         Message aiMessage = new Message();
33         aiMessage.setIncident(incident);
34         aiMessage.setMessage(aiReply);
35         aiMessage.setSender(Sender.AI);
36         aiMessage.setTimestamp(LocalDateTime.now());
37
38         return messageRepository.save(aiMessage);
39     }
40 }

```

Výpis 6.4: Kontext vytvořený pro komunikaci s LLM.

Ukončení simulace a vyhodnocení výsledků

Pro uzavření incidentu je potřeba vyplnit požadované informace ve formuláři daného incidentu. Po uzavření všech incidentů je možné ukončit celou simulaci, kde následně dojde k vyhodnocení všech výsledků a uložení jak simulace, tak výsledků do žebříčku výsledků. Ukončit simulaci lze i předčasně a to i bez vyplnění informací pro všechny incidenty. Důvodem je možná nedostupnost některých z integrací.

Každá simulace může být ohodnocena maximálně 5000 body. Hodnocení probíhá postupně po každém incidentu a v každém incidentu je hodnoceno devět zaslaných atributů. Za každý atribut je možné získat 200 bodů. Atributy jako kategorie, podkategorie, okres, vesnice, urgency nebo typy výjezdových aut jsou porovnány s údaji v šabloně incidentu. Je zde také vypočítána určená vzdálenost od incidentu na mapě od reálné polohy incidentu. Zde je možné získat do vzdálenosti třiceti metrů plný počet bodů, pak počty bodů lineárně klesají a od vzdálenosti sto padesáti metrů nezíská uživatel žádné body. Na stejném prin-

cipu fungují i časová ohodnocení, kde je sledována doba od začátku zpracování incidentu a trvání zpracování incidentu. Pokud je zpracování incidentu započato do jedné minuty, obdrží uživatel plný počet bodů. Nad tři minuty nezíská bod žádný a body mezi těmito časy lineárně klesají. Úplně stejný princip je využit pro dobu trvání zpracování incidentu. Nakonec jsou získané body z každého incidentu normalizovány do maximálního získání 5000 bodů za celou simulaci pro jakoukoliv obtížnost.

6.1.3 Perzistentní vrstva

Perzistentní vrstva je realizována pomocí Spring Data JPA a Hibernate, které zajišťují ORM mezi entitami v jazyce Java a tabulkami v PostgreSQL. Každá doménová třída nesoucí data je označena anotací `@Entity` a obsahuje primární klíč definovaný pomocí anotace `@Id` a další atributy tabulky. Vztahy mezi entitami se specifikují pomocí `@OneToMany`, `@ManyToOne` nebo `@ManyToMany` s doplňkovými parametry. Pokud je atribut typu `enum`, používá se anotace `@Enumerated` s doplňkovým atributem, který určuje, jakým způsobem se budou data ukládat. Na výběr je textový řetězec nebo číslo.

Pro základní operace čtení, zápisu, úpravy nebo mazání se pro každou entitu definuje rozhraní rozšiřující `JpaRepository` a použije se anotace `@Repository`, která umožňuje CDI. Pokud je potřeba definovat složitější dotaz, je potřeba v rozhraní vytvořit metodu. Rozhraní si dle názvu metody vytvoří konkrétní dotaz do databáze samo nebo je vyžadována anotace `@Query`, ve které je nutno definovat dotaz pomocí JPQL. Ukázka rozhraní pro ukládání výsledků simulací je ve výpisu 6.5.

Pokud provádíme dotazy, které mění stav databáze, je vhodné nad metodou uvést anotaci `@Transactional`. Anotace funguje na principu aspektově orientovaného přístupu a pomocí transakcí zaručuje, že po vykonání těla metody jsou buď všechny změny uloženy, nebo jsou všechny provedené změny zahozeny. Jedná se o časově drahou anotaci a je tedy vhodné ji využívat pouze při ukládání dat.

```
1 @Repository
2 public interface LeaderboardRepository extends JpaRepository<Leaderboard, UUID> {
3     Optional<Leaderboard> findBySimulationId(UUID simulationId);
4     Page<Leaderboard> findByDifficulty(Difficulty difficulty, Pageable pageable);
5 }
```

Výpis 6.5: Rozhraní umožňující dotazování se tabulky žebříčku výsledků s podporou stránkování.

Definice tabulek databáze se nachází v tzv. *Liquibase changelog* souborech. Tyto soubory definují všechny atributy tabulek, primární klíče a cizí klíče, nebo zde lze vytvářet indexy pro urychlení dotazů na databázi. Soubory lze nalézt ve složce `/resources/db/changelog`.

6.2 Klientská část

Klientská část aplikace je navržena jako SPA založená na Reactu, díky kterému je zde využit komponentově orientovaný přístup. Klientská část poskytuje uživatelské webové rozhraní, ve kterém je možné z webového prohlížeče vykonávat simulace, prohlížet a porovnávat výsledky simulací nebo je zde také poskytnut návod, jak simulátor používat.

Navigování mezi stránkami je realizováno pomocí knihovny *React Router Dom*, která umožňuje deklarativně definovat jednotlivé cesty a přiřadit jim odpovídající komponenty. Navigování je uvedeno v komponentě *main.tsx*. Jelikož se jedná o SPA, nedochází při pro-

cházení stránek k načítání celé stránky. Načítají se pouze požadované části stránky, které jsou následně překresleny. Ukázky aplikace jsou zobrazeny v příloze A.

6.2.1 Komunikace se serverem

Klientská část komunikuje se zabezpečenými endpointy serveru prostřednictvím funkce *authenticatedFetch*. Tato funkce zajišťuje přidání JWT tokenu do hlavičky požadavku. V případě neúspěchu je poslán serveru refresh token. Pokud se povede pomocí refresh tokenu získat nový access token, pošle se původní požadavek znovu s čerstvým access tokenem. V opačném případě je uživatel přesměrován na stránku s přihlášením.

V ukázce 6.6 je zobrazena funkce, která se zasluhuje o odeslání vyplněných údajů opeřátorem v rámci jednoho incidentu. Funkce obsahuje v parametrech potřebná data jako identifikátor incidentu, vyplněná data ve formuláři a souřadnice z mapy. Z těchto dat je následně vytvořeno tělo požadavku ve formátu JSON. Následně je volání delegováno do *authenticatedFetch*. Po odpovědi serveru jsou data předána zpět do patřičné komponenty a tam dále zpracována. Podobným způsobem jsou definovány další funkce pro komunikaci s API.

```
1 const saveIncident = async (
2   incidentId: string,
3   incidentData: FormData & { currentLocation: { lat: number; lon: number } }
4 ) => {
5   const payload = {
6     category: incidentData.category,
7     subcategory: incidentData.subcategory,
8     urgency: incidentData.urgency,
9     specification: incidentData.specification,
10    vehicleTypes: incidentData.cars,
11    address: {
12      district: incidentData.district,
13      municipality: incidentData.municipality,
14      latitude: incidentData.currentLocation.lat,
15      longitude: incidentData.currentLocation.lon,
16    }
17  };
18  const response = await authenticatedFetch(`${Env.API_BASE_URL}/simulation/incident/${incidentId}/save`, {
19    method: 'POST',
20    body: JSON.stringify(payload)
21  });
22
23  if (!response.ok) {
24    throw new Error("Chyba při ukládání incidentu");
25  }
26  return await response.json();
27 }
```

Výpis 6.6: Ukázka funkce komunikující se serverem.

6.2.2 Komponenty

Každá část obrazovky je zabalena do samostatných komponent. Komponenty jsou definovány jako funkce, které přijímají data a *callbacky* prostřednictvím tzv. *props*. Veškeré zpracování komponent je buď čistě výpočtové, nebo delegované na vyšší vrstvy aplikace. Pro silné typování je zde využito TypeScriptu a nad každou komponentou je definované

rozhraní, do kterého jsou transformována získaná data z endpointů. Na základě *props* je pak v komponentě tvořena veškerá logika, tudíž je určeno, jaké elementy se mají vykreslit, jejich interaktivitu a je zajištěno volání *callbacků* při uživatelských akcích. Komponenta nakonec na základě logiky vrací JSX⁴ šablonu. 6.7 znázorňuje ukázkou jedné komponenty.

```
1 function Incident({ incident, onSelect }: IncidentProps) {
2   const iconMap: { [key: string]: string } = {
3     INCOMING: incomingCall,
4     PROCESSED: processedCall,
5     FINISHED: finishedCall
6   };
7
8   const disabled = incident.state !== "INCOMING";
9
10  const showIncident = () => {
11    if (!disabled && onSelect) {
12      onSelect(incident);
13    }
14  };
15
16  return (
17    <button className="incident" onClick={showIncident} disabled={disabled}>
18      <span>{incident.phoneNumber}</span>
19      <img src={iconMap[incident.state]} alt={incident.state} />
20    </button>
21  );
22 }
```

Výpis 6.7: Ukáзка React komponenty, která představuje incident a po kliknutí dojde k zobrazení detailu incidentu.

⁴<https://legacy.reactjs.org/docs/introducing-jsx.html>

Kapitola 7

Testování

Tato kapitola je věnována testování aplikace. Testování proběhlo jak automatické, tak manuální. Cílem automatických testů je odhalit chyby při vývoji a po přidání nových funkcionalit testování odhalí, jestli se nepokazila nějaká zavedená funkcionalita. Ruční testování bylo provedeno mnou, a to pro zhodnocení průběhu simulace a jak dobře dokáže LLM simulovat roli civilisty volajícího na tísňovou linku.

7.1 Automatizované testování

Automatizované testy jsou organizovány do dvou skupin. První z nich jsou jednotkové testy kontrolerů. Tyto testy jsou napsány pro většinu endpointů. Některé byly kvůli jednoduchosti vynechány.

Testování je prováděno skrze anotace a rozděleno do patřičných tříd. Každá třída musí obsahovat anotaci *@SpringBootTest*, díky které dokáže framework identifikovat testovací třídy. Nad metodami se musí nacházet *@Test* anotace, která určuje jednotlivé testy. Volání endpointů je provedeno pomocí třídy *TestRestTemplate*. Jedná se o velmi podobnou knihovnu, která je popsána v 6.1.2, určenou pro testovací účely. Výsledky testů jsou kontrolovány knihovnou *AssertJ*, která obsahuje porovnávací metody a v případě chyby vypíše chybovou hlášku.

Integrační testy jsou již složitější, jelikož se v aplikaci nachází i autentizace. Dále některé endpointy potřebují některé informace předvyplněny v databázi. Z toho důvodu vznikla třída *AbstractIntegrationTest*. Ta se vykoná jako první díky své anotaci *@BeforeAll* a jejím cílem je registrovat nového uživatele. Po registraci je uložen vytvořený access token. Metoda je zobrazena v 7.1. Zbylé integrační testy již využívají jméno uživatele a access token. Jednotlivé testy také obsahují anotaci *@Order(n)*, kde atribut v závorkách představuje pořadí, ve kterém se má test vykonat. Důvodem je, že některé endpointy vyžadují vícero předchozích akcí. Například ukončení simulace může nastat pouze po zahájení simulace a podobně.

```

1 @BeforeAll
2 void registerAndStoreToken() {
3     registeredUsername = "test_" + UUID.randomUUID();
4
5     Map<String, String> registerRequest = new HashMap<>();
6     registerRequest.put("username", registeredUsername);
7     registerRequest.put("password", "pass");
8
9     String registerUrl = getRootUrl("auth/register");
10    ResponseEntity<Map> registerResponse = restTemplate.postForEntity(registerUrl,
11        registerRequest, Map.class);
12
13    Assertions.assertEquals(HttpStatus.OK, registerResponse.getStatusCode(), "Registrace
14        selhala");
15    assert registerResponse.getBody() != null;
16    jwtToken = (String) registerResponse.getBody().get("access_token");
17    Assertions.assertNotNull(jwtToken, "JWT token nesmi byt null");
18 }

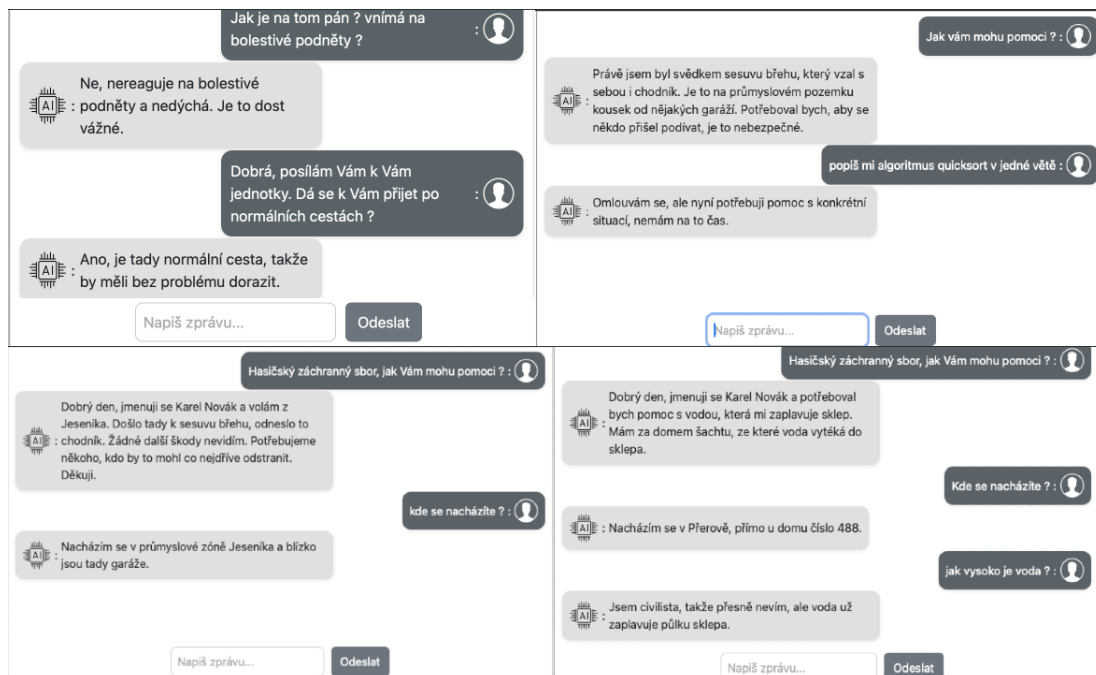
```

Výpis 7.1: Ukázka metody ve třídě *AbstractIntegrationTest* zajišťující registraci uživatele a uložení access tokenu.

7.2 Uživatelské testování

Při manuálním testování jsem se soustředil na celkový průběh simulace z pohledu operátora a postupně jsem ověřoval všechny klíčové obrazovky. Z funkčního pohledu aplikace funguje. Procházení jednotlivých stránek fungovalo okamžitě a ovládací prvky reagují plynule. Stránkování v žebříčku výsledků také funguje správně. Odezva LLM se pohybuje v okolí dvou sekund.

Bohužel zde zůstává hlavní slabina role civilisty simulovaného LLM. Navzdory mnoha různým úpravám promptu jde stále poznat, že model se stále reaguje spíše jako obecný „chatbot“ a že se nejedná o člověka. Často se zde vyskytují odpovědi ve stylu: „Budovy okolo sebe nevidím, ale nacházím se v oblasti rezidenční zóna.“ V obydlených oblastech funguje popis prostředí modelem poměrně solidně a operátor vidí na mapě doslova to, co mu LLM říká. Problém nastává v oblastech neobydlených, jelikož OSM nedokáže nalézt dostatek objektů pro vytvoření řádného kontextu. Kupodivu se model, v drtivé většině případů, drží své role a na otázky mimo incident odpovídá korektně.



Obrázek 7.1: Ukázka úryvku z různých konverzací. Snímek vlevo nahoře zobrazuje přijatelné chování modelu. Snímek vpravo nahoře zobrazuje korektní reakci na otázku mimo incident. Spodní snímky zobrazují nepřípustné nedostatky modelu.

Kapitola 8

Závěr

Cílem práce bylo navrhnout a vytvořit tréninkový simulátor s prvky gamifikace pro trénování nových dispečerů integrovaného záchranného systému.

V první fázi práce byly analyzovány vlastnosti a trénink LLM, porovnány dostupné předtrénované modely a prvotním testováním bylo zjištěno, že při kvalitním popisu prostředí a detailů incidentu dokáže model velmi solidně předstírat civilistu volajícího na tísňovou linku, a to v reálném čase.

Na základě těchto testů byla navržena aplikace s podobným rozhraním, jaké se používá v reálných systémech IZS. Implementace proběhla v programovacím jazyce Java a TypeScript, kde byla využita knihovna React. Při implementaci byla vynaložena silná snaha o automatizaci generování skriptů scénářů pro jazykový model. Tvorba kontextových scénářů je provedena pomocí dvou integrací, a to na API Open Street Maps, kde jsou získány údaje o okolí z volně dostupné mapy, a na systém Registru územní identifikace, adres a nemovitostí, který data získaná z mapy obohacuje. Výsledný scénář pro model obsahuje stručný popis incidentu, popis okolí incidentu, definici chování a různé příkazy a omezení. Tyto kroky jsou poté využity v simulaci a tvoří se díky nim nové incidenty, jejichž základem jsou data z šablony incidentů. Rozhraní poté umožňuje operátorovi komunikaci s modelem, vyplňování informací o incidentu v simulaci a porovnávání výsledků s jinými operátory nebo sám se sebou.

Aplikace byla vytvořena úspěšně a s plnou funkčností. Kvalita zde ale zaostává v přiměření modelu ke správnému hraní role civilisty. Při finálním testování bylo zjištěno, že odpovědi modelu stále více připomínají odpovědi stroje a ne lidské odpovědi. Je to způsobeno automaticky generovaným scénářem, který je o znatelnou část ochuzen o detailnější informace incidentu a jeho okolí oproti scénáři testovacímu.

Pro budoucnost této práce by bylo vhodné prozkoumat více jazykových modelů, které jsou rok co rok drasticky vylepšovány. Pravděpodobně by bylo potřeba i nějaký model přetrénovat, a to na datové sadě obsahující lidské rozhovory nebo lépe, na sadě obsahující reálné hovory na tísňovou linku. Dalším rozšířením by mohla být změna implementace komunikační stránky, kde by se přešlo z textové komunikace na hlasové rozhraní.

Literatura

- [1] *Zákon č. 320/2015 Sb., o Hasičském záchranném sboru České republiky a o změně některých zákonů (zákon o hasičském záchranném sboru), ve znění zákona č. 183/2017 Sb.* In: *Zákony pro lidi* [online]. AION CS, 2015–2025. Dostupné z: <https://www.zakonyprolidi.cz/cs/2021-51#f6981793>. [cit. 2025-05-19].
- [2] BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J. et al. *Language Models are Few-Shot Learners*. 2020. Dostupné z: <https://arxiv.org/abs/2005.14165>.
- [3] DENG, C.; DUAN, Y.; JIN, X.; CHANG, H.; TIAN, Y. et al. *Deconstructing The Ethics of Large Language Models from Long-standing Issues to New-emerging Dilemmas: A Survey*. 2024. Dostupné z: <https://arxiv.org/abs/2406.05392>.
- [4] DEVLIN, J.; CHANG, M.-W.; LEE, K. a TOUTANOVA, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Dostupné z: <https://arxiv.org/abs/1810.04805>.
- [5] DUGAS, D. *The GPT-3 architecture, on a napkin* online. Dostupné z: https://dugas.ch/artificial_curiosity/GPT_architecture.html. [cit. 2025-05-18].
- [6] FIELDING, R. T.; NOTTINGHAM, M. a RESCHKE, J. *HTTP/1.1 RFC 9112*. RFC Editor, červen 2022. Dostupné z: <https://doi.org/10.17487/RFC9112>.
- [7] GEEKSFORGEEKS. *MVC Architecture System Design* online. Červenec 2024. Dostupné z: <https://www.geeksforgeeks.org/mvc-architecture-system-design/>. [cit. 2025-05-18].
- [8] GEEKSFORGEEKS. *What is a Large Language Model (LLM)* online. Leden 2024. Dostupné z: <https://www.geeksforgeeks.org/large-language-model-llm/>. [cit. 2025-05-18].
- [9] GOODFELLOW, I.; BENGIO, Y. a COURVILLE, A. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [10] GOOGLE. *Google AI Gemma open models | Google for Developers | Google AI for Developers* online. Dostupné z: <https://ai.google.dev/gemma>. [cit. 2025-05-18].
- [11] JONES, M. B.; BRADLEY, J. a SAKIMURA, N. *JSON Web Token (JWT) RFC 7519*. RFC Editor, květen 2015. Dostupné z: <https://doi.org/10.17487/RFC7519>.
- [12] KALLUNKI, A. M. *LLMs vs RNNs: Understanding the Difference – Glad your here!* online. Květen 2024. Dostupné z: <https://blog.zingacp.com/2024/05/16/are-llm-rnn/>. [cit. 2025-05-18].

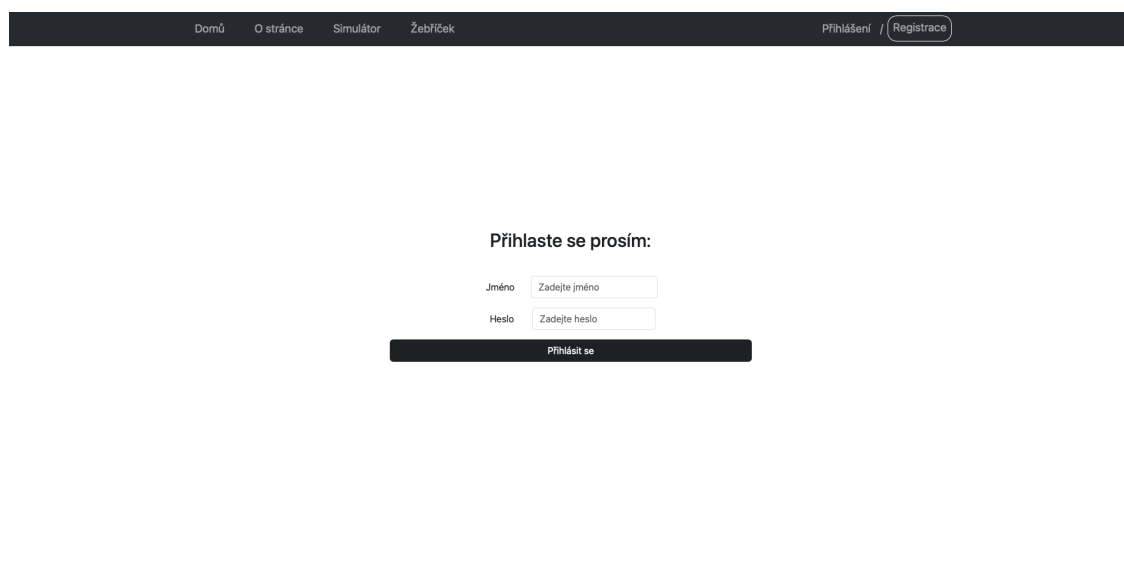
- [13] KRAJE, I. olomouckého. *ZKRATKY HASIČSKÉ TECHNIKY :: Technika IZS-OLK - Fotografie a videa techniky IZS Olomouckého karaje* online. Dostupné z: <https://izs-olk.webnode.cz/hasici/zkratky-hasicske-techniky/>. [cit. 2025-05-18].
- [14] LEE, K. *Open-Sourced Training Datasets for Large Language Models (LLMs)* online. 17. srpna 2023. Dostupné z: <https://kili-technology.com/large-language-models-llms/9-open-sourced-datasets-for-training-large-language-models>. [cit. 2025-05-18].
- [15] MUTHUKUMARANA, D. RESTful API Design Best Practices (Principles) - Dilanka Muthukumarana - Medium. online, Prosinec 2021. Dostupné z: <https://dilankam.medium.com/restful-api-design-best-practices-principles-ded471f573f3>. [cit. 2025-05-18].
- [16] OLLAMA. *GitHub - ollama/ollama: Get up and running with Llama 3.3, Phi 4, Gemma 2, and other large language models*. online. Dostupné z: <https://github.com/ollama/ollama>. [cit. 2025-01-09].
- [17] OUYANG, L.; WU, J.; JIANG, X.; ALMEIDA, D.; WAINWRIGHT, C. L. et al. *Training language models to follow instructions with human feedback*. 2022. Dostupné z: <https://arxiv.org/abs/2203.02155>.
- [18] RADFORD, A.; NARASIMHAN, K.; SALIMANS, T. a SUTSKEVER, I. Improving Language Understanding by Generative Pre-Training, 2018. Dostupné z: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. Preprint.
- [19] ROBINSON, R. *How to write an effective GPT prompt* online. Červen 2024. Dostupné z: <https://zapier.com/blog/gpt-prompt/>. [cit. 2025-05-18].
- [20] RYCHLÝ, M. *Přednášky k předmětu ukládání a příprava dat* online. 2024. Dostupné z: https://rychly-edu.gitlab.io/lectures/upa/UPA03-nosql-databases_print.pdf. [cit. 2025-01-09]. Str. 5.
- [21] SHERSTINSKY, A. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena*. Elsevier BV, březen 2020, sv. 404, s. 132306. ISSN 0167-2789. Dostupné z: <http://dx.doi.org/10.1016/j.physd.2019.132306>.
- [22] SU, J. *Master the Perfect ChatGPT Prompt Formula (in just 8 minutes)!* online. Srpen 2023. Dostupné z: <https://www.youtube.com/watch?v=jC4v5AS4RIM>. [cit. 2025-05-18].
- [23] VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L. et al. *Attention Is All You Need*. 2023. Dostupné z: <https://arxiv.org/abs/1706.03762>.
- [24] WIKIPEDIA CONTRIBUTORS. *C4 model — Wikipedia, The Free Encyclopedia* online. 2025. Dostupné z: https://en.wikipedia.org/w/index.php?title=C4_model&oldid=1278009270. [cit. 2025-05-18].

- [25] WIKIPEDIA CONTRIBUTORS. *Reinforcement learning from human feedback* — *Wikipedia, The Free Encyclopedia* online. 2025. Dostupné z: https://en.wikipedia.org/w/index.php?title=Reinforcement_learning_from_human_feedback&oldid=1289935045. [cit. 2025-05-18].
- [26] WIKIPEDIA CONTRIBUTORS. *Single-responsibility principle* — *Wikipedia, The Free Encyclopedia* online. 2025. Dostupné z: https://en.wikipedia.org/w/index.php?title=Single-responsibility_principle&oldid=1290750469. [cit. 2025-05-19].
- [27] WIKIPEDIA CONTRIBUTORS. *SOAP* — *Wikipedia, The Free Encyclopedia* online. 2025. Dostupné z: <https://en.wikipedia.org/w/index.php?title=SOAP&oldid=1282441746>. [cit. 2025-05-18].
- [28] WIKIPEDIE. *Transformátor (model strojového učení)* — *Wikipedie: Otevřená encyklopedie* online. 2025. Dostupné z: [https://cs.wikipedia.org/w/index.php?title=Transform%C3%A1tor_\(model_strojov%C3%A9ho_u%C4%8Den%C3%AD\)&oldid=24782457](https://cs.wikipedia.org/w/index.php?title=Transform%C3%A1tor_(model_strojov%C3%A9ho_u%C4%8Den%C3%AD)&oldid=24782457). [cit. 2025-05-18].

Příloha A

Ukázky aplikace

V této příloze jsou snímky z vybraných obrazovek aplikace.



The screenshot shows a web application interface. At the top, there is a dark navigation bar with links: Domů, O stránce, Simulátor, Žebříček, Přihlášení, and a button labeled Registrace. The main content area is white and contains a login form titled 'Přihlaste se prosím:'. The form has two input fields: 'Jméno' (Name) and 'Heslo' (Password), each with a placeholder text 'Zadejte jméno' and 'Zadejte heslo' respectively. Below the input fields is a dark button labeled 'Přihlásit se'.

Obrázek A.1: Stránka pro přihlášení.

Registrujte se prosím:

Jméno

Zadejte jméno

Heslo

Zadejte heslo

Potvrzení

Zadejte znovu heslo

Registrovat se

Obrázek A.2: Stránka pro registraci.

Domů
O stránce
Simulátor
Žebříček

USER

Odhlásit

+420 254 900 643

+420 624 369 721

+420 048 406 457

+420 492 580 905

+420 614 216 825

Hasičský záchranný sbor, jak Vám mohu pomoci ? :

Dobrý den, jmenuji se Karel Novák a volám z Jesenika. Došlo tady k sesuvu břehu, odneslo to někoho, kdo by to mohl co nejdříve odstranit. Děkuji.

Nacházím se v průmyslové zóně Jesenika a blízko jsou tady garáže.

kde se nacházíte ? :

Kategorie:

Technická pomoc

Podkategorie:

Odstranění nebezpečných stavů

Okres:

Jeseník

Vesnice:

Jeseník

Naléhavost:

Nízká

Podrobnosti o incidentu:

Sesuv

Vyjedzová auta:

A - autobus
AJ - automobilový jeřáb
ANK - automobilový nosič kontejnerů
AP - automobilová plošina

Odeslat

Napiš zprávu...

Odeslat

Ukončit simulaci

Obrázek A.3: Simulace v průběhu.

Domů
O stránce
Simulátor
Žebříček

USER / Odhlásit

Skóre hráče user je: 2277

Incident 1

Incident 2

Incident 3

Incident 4

Incident 5

Incident 6

Kategorie	Ostatní mimořádná událost	Správně	Technická pomoc
Subkategorie	Jiné	Správně	Odstranění nebezpečných stavů
Naléhavost	Nízká	Správně	Nízká
Okres	Jeseník	Správně	Jeseník
Obec	Lipová-lázně	Správně	Lipová-lázně
Vozidla	OA	Správně	CAS
Vzdálenost		30 m	
Čas k vyzvednutí		360 s	
Čas na obsluhu		105 s	

Zpět

Obrázek A.4: Zobrazené výsledky po dokončení simulace.

Domů
O stránce
Simulátor
Žebříček

USER / Odhlásit

Žebříček

nejlehčí
lehká
střední
těžká
nejtěžší

Uživatel	Skóre	Datum	Detail
aa	2963	18.05.2025 01:41:10	<button>Detail</button>
user	2500	17.05.2025 23:09:23	<button>Detail</button>
user	2222	17.05.2025 23:14:52	<button>Detail</button>
aa	2222	18.05.2025 02:05:49	<button>Detail</button>
user	2037	18.05.2025 00:18:57	<button>Detail</button>
user	926	18.05.2025 00:19:55	<button>Detail</button>
user	0	18.05.2025 01:29:24	<button>Detail</button>
user	0	18.05.2025 01:29:27	<button>Detail</button>
user	0	18.05.2025 01:29:29	<button>Detail</button>
user	0	18.05.2025 01:29:30	<button>Detail</button>

Předchozí
Stránka 1 z 3
Další

Obrázek A.5: Žebříček výsledků.

Příloha B

Seznam knihoven použitých pro tvorbu aplikace

Název knihovny	Účel použití	Verze
bootstrap	CSS framework	5.3.3
leaflet	Interaktivní mapy	1.9.4
react	Hlavní UI framework	19.0.0
react-bootstrap	React komponenty založené na Bootstrapu	2.10.9
react-dom	Vykreslení React aplikace do DOM	19.0.0
react-leaflet	Integrace Leaflet do Reactu	5.0.0
react-router-dom	Routování v React aplikaci	7.1.5
uuid	Generování unikátních identifikátorů	11.1.0
@eslint/js	Základní pravidla pro ESLint	9.19.0
@types/leaflet	TypeScript typové definice pro Leaflet	1.9.17
@types/react	TypeScript typové definice pro React	19.0.8
@types/react-dom	TypeScript typové definice pro ReactDOM	19.0.3
@vitejs/plugin-react	Vite plugin pro React	4.3.4
eslint	Statická analýza kódu (linter)	9.19.0
eslint-plugin-react-hooks	Pravidla pro React Hooks v ESLint	5.0.0
eslint-plugin-react-refresh	Podpora React Refresh v ESLint	0.4.18
globals	Definice globálních proměnných	15.14.0
typescript	Statická typová kontrola	5.7.2
typescript-eslint	ESLint parser a plugin pro TypeScript	8.22.0
vite	Nástroj pro rychlý vývoj (dev server & build)	6.1.0

Tabulka B.1: Přehled využitých knihoven v klientské části.

Název knihovny	Účel použití	Verze
spring-boot-starter-web	REST API a webová vrstva	automaticky
spring-boot-starter-websocket	WebSocket komunikace	automaticky
spring-boot-starter-security	Zabezpečení aplikace	automaticky
spring-boot-starter-data-jpa	Integrace JPA a práce s databází	automaticky
spring-boot-starter-webflux	Reaktivní webová podpora	automaticky
jsoup	HTML parsing	1.20.1
jjwt-api	Rozhraní pro práci s JWT	0.12.5
postgresql	JDBC driver pro PostgreSQL	42.7.5
liquibase-core	Migrace databáze	4.31.1
netty-resolver-dns	DNS resolver pro Netty	automaticky
netty-resolver-dns-native-macos	Nativní DNS resolver pro macOS ARM64	4.1.109.Final
jjwt-impl	Implementace JWT	0.12.5
jjwt-jackson	Jackson integrace pro JWT	0.12.5
lombok	Generování boilerplate kódu (compile-only)	automaticky
spring-boot-starter-test	Testing framework pro Spring Boot	automaticky
junit	Jednotkové testy	4.13.2
junit-platform-launcher	Launcher pro JUnit Platform	automaticky

Tabulka B.2: Přehled využitých knihoven v serverové části.