F3 **Fakulta elektrotechnická**
Katedra počítačů

Diplomová práca

Diplomová práca

Backend aplikácie FactCheck

Bc. Rastislav Kopál
Softvérové inžinierstvo

December 2023
`https://github.com/aic-factcheck/fact_check_api`
Vedúci práce: Ing. Jan Drchal, Ph.D.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kopál** Jméno: **Rastislav** Osobní číslo: **508491**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Backend platformy pro sdílené ověřování faktů

Název diplomové práce anglicky:

Crowd-sourcing Platform for Fact-checking (Backend)

Pokyny pro vypracování:

Úlohou je navrhnúť a implementovať backendovú časť platformy zameranú na overovanie faktov pomocou crowdsourcingu. Cieľom systému je najmä zhromažďovanie textových tvrdení (vrátane zdrojov) a ich následné overovanie podložené dôkazmi.
Platforma by mala byť navrhnutá s ohľadom na následné strojové spracovanie zozbieraných dát.
1) Preskúmajte existujúce platformy crowdsourcingu. Zamerajte sa na overovanie faktov a jednoduchosť strojového spracovania zozbieraných údajov.
2) Vyberte si vhodné technológie, navrhnite a implementujte backend systému.
3) Zamerajte sa hlavne na dva prípady použitia: a) nahlásenie tvrdenia pre následný proces overovania, b) proces overovania daného tvrdenia na základe dôkazov.
4) Navrhnite backendovú časť systému na základe spätnej väzby od používateľov (poskytne vedúci práce).
5) Na vyhodnotenie systému použite testerov. Prístup k testerom zabezpečí vedúci práce - testovací tím bude pozostávať buď zo študentov žurnalistiky na FSV alebo členov skupiny NLP na AIC FEL.

Seznam doporučené literatury:

[1] Allen, Jennifer, et al. "Scaling up fact-checking using the wisdom of crowds." Science advances 7.36 (2021): eabf4393
[2] Pinto, Marcos Rodrigues, et al. "Towards fact-checking through crowdsourcing." 2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD). IEEE, 2019.
[3] Sotirakou, Catherine, Theodoros Paraskevas, and Constantinos Mourlas. "Toward the Design of a Gamification Framework for Enhancing Motivation Among Journalists, Experts, and the Public to Combat Disinformation: The Case of CALYPSO Platform." International Conference on Human-Computer Interaction. Springer, Cham, 2022.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Drchal, Ph.D. centrum umělé inteligence FEL

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

Datum zadání diplomové práce: **30.01.2023** Termín odevzdání diplomové práce: **09.01.2024**

Platnost zadání diplomové práce: **22.09.2024**

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

III. PŘEVZETÍ ZADÁNÍ

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

. _____
Datum převzetí zadání

_____
Podpis studenta

Poďakovanie / Prehlásenie

Touto cestou by som chcel poďakovať Ing. Janovi Drchalovi, Ph.D. za ochotu, pripomienky, cenné rady a odborné vedenie, ktoré prispeli k vypracovaniu tejto záverečnej práce.

Vyhlasujem, že som predloženú diplomovú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s s Metodickým pokynom o dodržiavaní etických princípov pri vypracovaní vysokoškolskej práce.

V Prahe, 3. 12. 2023

.....

Abstrakt / Abstract

Táto diplomová práca sa zaoberá tvorbou backendu aplikácie FactCheck, ktorá je zameraná na overovanie faktov na internete a boj proti dezinformáciám. Práca pokrýva celý proces od analýzy požiadaviek, návrhu, implementácie až po testovanie a nasadenie softvéru. V rámci práce je prezentovaný detailný pohľad na softvérovú architektúru, výber vhodných databázových systémov, bezpečnostné aspekty a metodiky vývoja. Aplikácia využíva crowdsourcing a prvky gamifikácie pre zapojenie používateľov a zvyšuje dôveru verejnosti v informácie na internete prostredníctvom transparentného a demokratického procesu overovania faktov. Práca taktiež poskytuje porovnanie existujúcich riešení a navrhuje možnosti budúceho rozvoja platformy.

Kľúčové slová: Backend; Informačný systém; Crowdsourcing; Gamifikácia; Mikroslužby; NodeJS; NestJS; MongoDB; Typescript

This thesis deals with the creation of the backend of the FactCheck application, which is aimed at verifying the veracity of information on the Internet and combating misinformation. The thesis covers the whole process from requirements analysis, design, implementation to testing and deployment of the software. The work presents a detailed view of the software architecture, selection of appropriate database systems, security aspects and development methodologies. The application uses crowdsourcing and gamification elements to engage users and increase public trust in information on the Internet through a transparent and democratic fact-checking process. The thesis also provides a comparison of existing solutions and suggests options for future platform development.

Keywords: Backend; Information system; Crowdsourcing; Gamification; Microservices; NodeJS; NestJS; MongoDB; Typescript

Obsah /

Tabuľky / Obrázky

Kapitola 1
Úvod

Platforma FactCheck je webová aplikácia, ktorej cieľom je zvýšiť dôveru verejnosti v informácie na internete, zabrániť šíreniu dezinformácií a v neposlednom rade rozvíjať kritické myslenie u používateľov internetu. Overovanie faktov na platforme FactCheck nie je delegované iba odborníkom v danej oblasti, ale bežnej verejnosti. Aplikácia využíva prvky gamifikácie pre vytvorenie krátkodobej a dlhodobej motivácie u používateľov, aby vykonávali očakávané aktivity častejšie. Kolaboráciou odborníkov v oblasti žurnalistiky s bežnou verejnosťou chceme prispieť k demokratickému spôsobu overovania faktov na internete. Používatelia aplikácie FactCheck majú následne možnosť vyhľadávať potenciálne klamlivé alebo zavádzajúce tvrdenia a články. K týmto tvrdeniam používatelia nájdu zverejnené analýzy (hodnotenia), s detailnou evidenciou faktov s odkazmi na zdroje. Backend aplikácie FactCheck je vyvinutý ako vysoko škálovateľné riešenie s možnosťou jednoduchého rozširovania. Implementácia využíva najlepšie osvedčené postupy z praxe od analýzy, návrhu, cez implementáciu až po testovanie a nasadenie.

Backend platformy poskytuje REST API koncové body pre kompletný proces overovania faktov. Poskytuje používateľom možnosť sa bezpečne zaregistrovať a opätovne prihlásiť. Vkladať články a tvrdenia, ktoré sú potenciálne klamlivé alebo zavádzajúce. Následne dáva používateľom možnosť analyzovať a hodnotiť existujúce tvrdenia s možnosťou pridania zdrojov a evidencie, ktoré potvrdzujú dané hodnotenie. Umožňuje tiež demokratický spôsob vyhodnocovania hodnotení na základe hlasovania davu. Hodnotenie s najväčším počtom pozitívnych hlasov je označené ako dôveryhodné s najväčšou pravdivosťou. Noví používatelia prichádzajúci na platformu môžu medzi tvrdeniami vyhľadávať, listovať a zobrazovať si hodnotenia iných používateľov. Používatelia majú možnosť nahlasovať iných, podozrivých používateľov. Backend poskytuje rozhranie pre administrátorov, ktorí majú vyššie právomoci ako bežní používatelia. Pre používateľov s rolou admin máme dostupné API na zobrazovanie problémových hlásení a prípadne zablokovanie podozrivých užívateľov.

Pre zvýšenie motivácie boli do systému navrhnuté a implementované prvky gamifikácie, ktorých cieľom je používateľov motivovať sa opätovne vrátiť. Používatelia získavajú za jednotlivé aktivity v systéme určitý počet bodov a tým si zvyšujú reputáciu na platforme. Backend zaznamenáva históriu používateľov a jeho aktivít. Tieto dáta sú následne využité pre tvorbu štatistík a rebríčkov.

V rámci tejto diplomovej práce si porovnáme existujúce riešenie pre overovanie faktov na internete. Ukážeme si aké majú obmedzenia a vysvetlíme si, prečo sme sa rozhodli navrhnúť a vyvinúť vlastný softvérový produkt. Vysvetlíme si potrebnú teóriu pre návrh a tvorbu moderných, škálovateľných a efektívnych backendov. Následne si popíšeme požiadavky na aplikáciu a navrhneme vlastné riešenie pre kolaboratívne overovanie faktov na internete. Popíšeme si technológie, ktoré boli využité pri implementácií a popíšeme si techniky, ktorými zabezpečujeme kvalitu dodaného a nasadeného softvéru. Na záver si uvedieme možnosti budúceho rozvoja platformy FactCheck.

Kapitola 2
Úvod do problematiky

Éra internetu a platformy sociálnych sietí prispievajú k exponenciálne narastajúcemu množstvu dát, ktoré používatelia internetu zverejnia. Internet a sociálne médiá urýchľujú šírenie informácií, čo je požehnaním a neustále silnejúci problémom zároveň. Moderné technológie umožňujú každému s prístupom na internet šíriť svoje názory a zverejňovať informácie v reálnom čase, či sa jedná o textové príspevky, fotografie alebo videá. Cieľom sociálnych sietí je spájanie ľudí aj na veľké vzdialenosti, no aktuálnym trendom dokážu práve naopak, rozdeliť ľudí ktorí sú si blízki, pomocou zdieľania nepravdivých informácií, poloprávd a klamstiev, ktoré vyvolávajú hádky a konflikty.

Šírenie takýchto správ je problémom pre celú spoločnosť, takéto ohýbanie reality využívajú často politici pre zvýšenie popularity, aj na úkor šírenia klamstiev.

Zabrániť šíreniu takýchto správ je obrovský problém, ktorý by viedol k cenzúre názorov. Jedno z riešení, ako zabrániť tomuto trendu je overovanie faktov odborníkmi. Mnohí špecialisti sa venujú overovaniu faktov na internete, či už pracovne alebo ako dobrovoľníci v rôznych organizáciach.

Je nepredstaviteľné, aby akákoľvek organizácia dokázala overovať všetky správy na internete. Proces overenia každého tvrdenia zaberá čas, pretože je nutné vyhľadať zdroje a evidenciu, ktoré zvolené tvrdenie vyvracajú alebo potvrdzujú. Naším cieľom je vytvoriť softvérovú platformu, ktorá by dokázala pomôcť špecialistom overovať fakty na internete pomocou využitia kolektívnej inteligencie a zahrnúť a vzdelať v tomto procese aj bežných ľudí. Využitie kolektívnej sily davu je riešením na problém stále narastajúceho množstva dát a dezinformácií na internete.

V tejto práci sa zameriame na proces overovania tvrdení na internete. Používateľ bude mať možnosť vložiť nové, potencionálne klamlivé tvrdenia, ktoré na internete našiel. Následne proces vyhodnocovania využije silu davu, pre nájdenie evidencie a faktov, ktoré tvrdenie vyvracajú alebo potvrdzujú.

2.1 Falošné správy a fact-checking

So vzostupom platforiem sociálnych sietí a ich vírusovej povahy šírenia bol digitálny vek svedkom nevídaného šírenia falošných správ pomocou internetu. Falošné správy často využívajú predsudky a emócie ľudí na to, aby sa šírili rýchlejšie, ďalej a hlbšie ako pravdivé informácie. Často tiež vyvoláva senzácie alebo cielene klame. Zotrvávanie v šírení nepravdivých informácií so sebou nesie vážne nebezpečenstvo, ako je narušenie dôvery verejnosti, deformovanie demokratických postupov a podnecovanie násilia. Pre príklad, ako sa nepravdivé informácie šíria využijeme dáta z aplikácie Twitter medzi rokmi 2006 až 2017. Približne 126000 falošných správ bolo zdielaných približne troma miliónmi ľudí, pričom falošné správy mali väčší dosah ako tie pravdivé. Vrchné percento falošných správ malo dosah na 1000 až 100000 používateľov, zatiaľ čo pravdivé správy mali len zriedka dosah väčší ako 1000 ľudí. Klamlivé správy sa tiež šírili rýchlejšie ako pravda. Tieto rozdiely dávame za dôsledok miere novosti správ a najmä emociálnym reakciám ľudí [1]. Fenomén falošných správ nabral na popularite najmä po voľbách v

Spojených štátoch amerických z roku 2016 a prezidentských voľbách v Brazílií v roku 2018, po ktorých bola spoločnosť Facebook nútená odstrániť veľké množstvá klamlivého a zavádzajúceho obsahu [2].

Z hľadiska konceptu, môžeme podľa [3] dezinformácie rozdeliť do následujúcich kategórií:

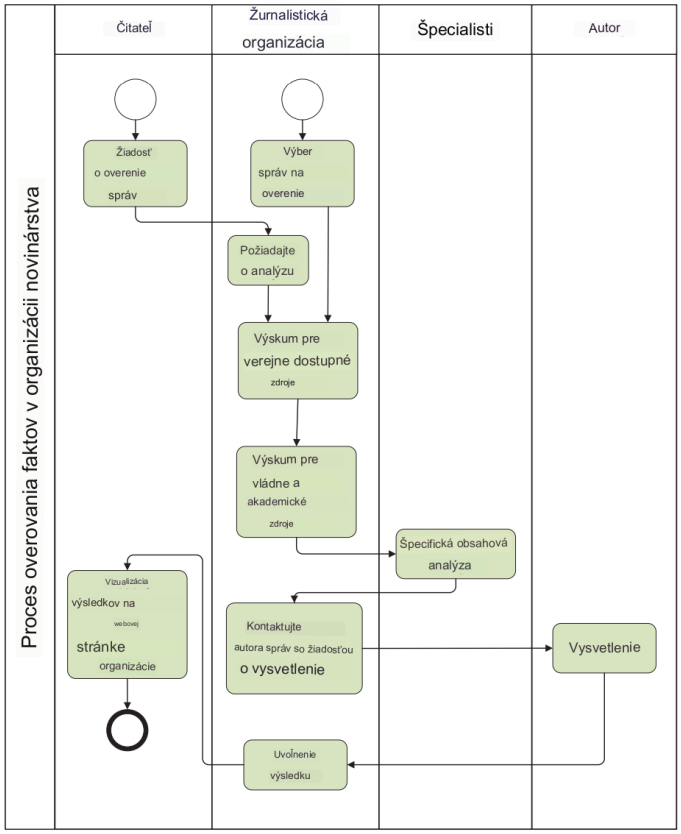
- Satira alebo paródia: Nemá za cieľ nikomu ublížiť, no môže ľudí nechcene oklamať.
- Zavádzajúci obsah: Predkladanie falošných dôkazov (nevinnej osobe), aby sa javili ako vinní a prípadne zdieľali tieto falošné dôkazy ďalej.
- Podvodný obsah: Keď skutočné zdroje sú zamenené podvodom.
- Vymyslený obsah: Nový obsah je kompletne vymyslený, s cieľom oklamať a škodiť.
- Nesprávne spojenie: Keď titulok, vizuály alebo popisy nezodpovedajú obsahu.
- Nesprávny kontext: Keď skutočne správny obsah je zdielaný s nesprávnymi informáciami alebo v nesprávnom kontexte.
- Manipulovaný obsah: Keď skutočne správna informácia alebo vizuál je manipulovaný (upravený) s cieľom oklamať [3].

2.2 Odborné organizácie pre fact-checking

Zastaveniu šíreniu klamstiev a falošných informácií sa venujú odborné organizácie na kontrolu a overovanie faktov, algoritmické techniky detekcie a výučba mediálnej gramotnosti. Napriek tomuto úsiliu je táto úloha stále veľká z dôvodu zložitosti ľudskej psychológie, klamlivým metódam, ktoré používajú tí, ktorí nepravdivé informácie šíria a prepojenej povahe sveta.

Proces overenia tvrdenia alebo faktu odbornou organizáciou je pomerne zdĺhavý a vyžaduje si úsilie špecialistov, ktorí musia overiť či daná informácia je pravdivá pomocou analýzy jej zdrojov. Analýza zahŕňa vyhľadávania evidencie, ktorá potvrdzuje zvolený fakt. Na internete fungujú mnohé organizácie pre overovanie faktov a tvrdení. Niektoré overujú vyhlásenia politikov alebo celebrít. Iné sa zameriavajú na overovanie akýchkoľvek faktov na internete. Tieto organizácie zdieľajú spoločný ciel - udržať kvalitu informácií, ktoré sa zdieľajú internetom pomocou nezávislých analýz a hodnotení rôznymi špecialistami z oblasti žurnalistiky.

Tieto iniciatívy a organizácie zdieľajú podobný proces overovania. Tento proces vždy začína výberom tvrdenia, ktoré sa bude overovať a končia kompletnou analýzou s výsledkom pre zvolené tvrdenie. Nie vždy sa dá každé tvrdenie označiť binárne ako buď pravdivé alebo nepravdivé. Rôzne iniciatívy používajú rôzne množiny výsledkov pre tvrdenie (napríklad: Pravda, čiastočná pravda, polopravda, satira, klamstvo a iné).



Obrázok 2.1. Proces overovania faktov odbornou organizáciou[2].

Na obrázku 2.1 môžeme vidieť bežný proces overovania faktov v odbornej žurnalistickej organizácií, do ktorého sa zapájajú čitatelia, ktorí tvrdenia označia a experti, ktorí pridajú obsahovú analýzu.

2.3 Crowdsourcing

Využitie sily davu (crowdsourcing) v informačných technológiách je technika, ktorá zapája strategicky zvolený dav ľudí alebo skupiny ľudí pre získanie určitého servisu, nápadov alebo obsahu vo forme otvorenej komunikácie. Tento koncept poukazuje, že internet dokáže uľahčiť zhromažďovanie alebo výber užitočných informácií od potenciálne veľkej množiny ľudí. Známy príklad využitia crowdsourcingu je napríklad platforma Wikipedia, ktorá zbiera dáta výlučne decentralizovaným a neznámym davom ľudí [2]. Tento koncept je často využívaný v rôznych iných softvéroch a platformách, ďalším známym príkladom je platforma Stackoverflow, ktorá zbiera riešenia na programátorské problémy od ľudí, pre ľudí. Existuje mnoho ďalších fungujúcich príkladov, kde dav dokáže spolupracovať a následne generovať obsah, ktorý vyžadujeme.

V posledných rokoch sa táto technika rýchlo vyvíja a stáva sa skvelou pomôckou vo vykonávaní určitých úloh na internete. Tieto aktivity a úlohy zahŕňajú distribuovaný spôsob riešenia problémov, otvorenú pozvánku pre verejnosť, kolektívnu kolaboráciu a lacnú a efektívnu výpočtovú schopnosť ľudí [2].

2.3.1 Wikipedia

Platforma Wikipedia je jednou z najúspešnejších projektov, ktoré využívajú crowdsourcing na internete. Ich hlavným princípom je, že ktokoľvek, kdekoľvek a kedykoľvek môže

pridať alebo upraviť existujúci článok. Umožňuje anonymným aj registrovaným používateľom pridávať články, upravovať ich, vymazávať alebo kolektívne kontrolovať pridaný obsah. Umožňuje registrovaným používateľom neustále kontrolovať zmeny a udržiavať tak kvalitu obsahu pomocou overovania faktov, opravovaním nepresností a odstraňovaním vandalizmu. Pri veľkých úpravách sa títo používatelia môžu spojiť a diskutovať o zmenách a môžu dospieť k záveru - ako by daný článok mal byť napísaný. Stránka taktiež poskytuje množstvo príručiek a politík, ktoré sú používatelia žiadaní dodržiavať (objektvitita, overiteľnosť, a iné). Wikipedia má taktiež administratívnu štruktúru - v systéme existujú administrátori, ktorí majú zodpovednosť a práva zablokovať používateľov, ochrániť zvolené články pred úpravou a mnohé iné [4]. Pomocou týchto bodov dokázala táto platforma zabezpečiť kvalitný a najmä rozsiahly obsah informácií, ktorých evidencia faktov je ľahko overiteľná.

2.3.2 Stackoverflow

Platforma Stackoverflow je ďalším úspešným príkladom využívajúci crowdsourcing s prvkami gamifikácie ako motiváciou pre používateľov. Táto platforma pracuje na systéme, ktorý sa spolieha na zapojenie komunity akademikov, vedcov a profesionálov v danom odbore. Používateľ, ktorý potrebuje poradiť so špecifickým problémom zverejní svoju otázku a ďalší používatelia môžu začať s písaním odpovedí. Dodatočne upravované môžu byť ako otázky, tak aj odpovede komunitou pre zvýšenie presnosti a pochopiteľnosti. Aplikácia používa hlasovací systém, pomocou ktorého môže zvýšiť popularitu, relevanciu a teda aj dosah otázok a odpovedí. Najlepšie odpovede sú potom vyberané komunitou a to tak, že odpovede s väčším počtom hlasov sa dostávajú vyššie. Používatelia za svoje otázky, odpovede a hlasovania dostávajú body a zvyšujú tým svoju reputáciu. Reputácia funguje ako hlavný element motivácie - za kvalitné odpovede zvýšia svoju reputáciu viac a stávajú sa viac uznávanými členmi komunity [5]. Komunita taktiež môže označovať nevhodné otázky a odpovede, prípadne spam - aby boli skontrolované a prípadne vymazané administrátormi. Používatelia, ktorí dosiahnu určitú reputáciu dostávajú právomoci podobné administrátorským, ktorými dokážu taktiež moderovať diskusie.

2.4 Porovnanie existujúcich riešení

2.4.1 FactCheck.org

FactCheck.org je nezisková a nestranná organizácia, ktorej cieľom je najmä kontrolovať a overovať faktickú presnosť vyhlásení politických osobností USA. Jej cieľom je znížiť dopad klamstiev na verejnosť a teda predchádzať zmätku, ktorý zavádzajúce tvrdenia politikov vyvolávajú. Monitorujú presnosť výrokov najznámejších amerických politikov, ktoré zaznejú či už v televízii, reklame, diskusií alebo akomkoľvek prejave. Cieľom projektu FactCheck.org je aj aplikovať overené praktiky žurnalistiky a zvýšiť informovanosť a porozumenie verejnosti. Projekt bol založený Pensylvánskou univerzitou s cieľom vytvoriť komunitu vedcov, ktorá by sa zaoberala otázkami verejnej politiky na rôznych amerických úrovniach [6].

Obrázok 2.2. Webová stránka platformy FactCheck.org[6].

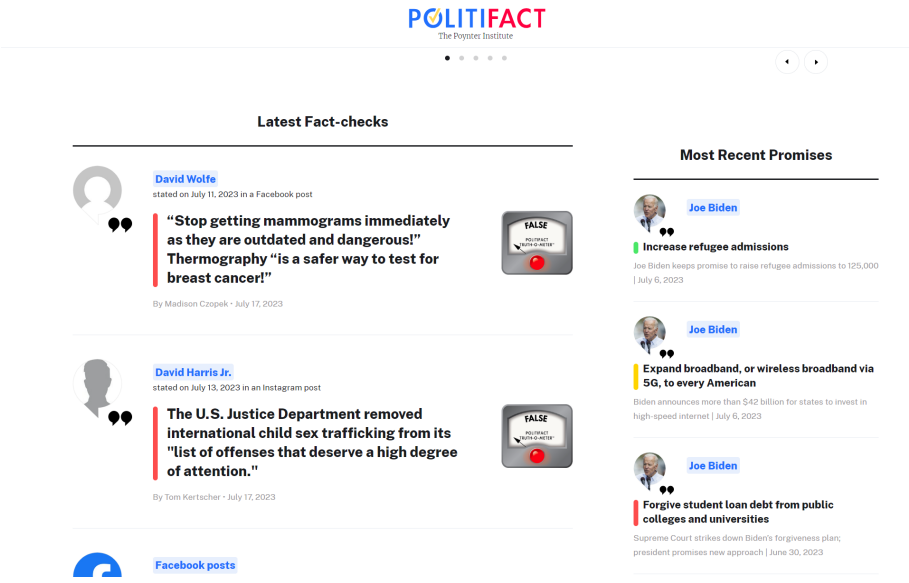
Táto platforma svoje aktivity na overovanie faktov nezabezpečuje využitím crowdsourcingu. Zamestnáva odborníkov na overovanie faktov z oblasti žurnalistiky. Stránka obsahuje články a tvrdenia na rôzne politické témy. Používateľ má možnosť sa zaregistrovať a pridať do systému požiadavku na overenie - či už článku, tvrdenia alebo videa. Používateľ si môže vyhľadať najnovšie, respektíve aktuálne najzaujímavejšie články. Môže si podobne vyhľadať aj aktuálne virálne tvrdenia (V preklade vírusové - ktoré sa v spoločnosti šíria podobne vírusu). Tieto najviac zdieľané tvrdenia sú zobrazené podľa trendov na sociálnych sieťach a aktualizujú ich zamestnanci organizácie alebo iní prispievatelia [6].

2.4.2 Politifact

Politifact funguje podobne ako FactCheck.org ako platforma pre nestranné overovanie a hodnotenie pravdivosti politických tvrdení. Tvrdenia, ktoré sa budú na tejto platforme overovať vyberajú zamestnanci platformy, novinári. Tvrdenia môžu byť vybraté z preslovov, tlačových konferencií, brožúr kampane alebo zaslané emailom od kohokoľvek. Pri rozhodovaní, ktoré tvrdenie sa bude overovať, zamestnanci berú do úvahy následujúce otázky:

- Je to overiteľné ? Niektoré vety, ktoré sú založené napríklad na názoroch nie je možné faktograficky overiť.
- Je tvrdenie významné?
- Je pravdepodobné, že sa zvolené vyhlásenie bude šíriť?
- Rozmýšľal by nad pravdivosťou výroku priemerný človek?

Toto sú otázky, ktoré predchádzajú každému procesu overenia tvrdenia. Sú nevyhnutné, aby profesionáli a špecialisti na overovanie faktov nestrácali čas overovaním bezvýznamných výrokov a tvrdení [7].



Obrázok 2.3. Webová stránka platformy Politifact.com [7].

Táto platforma sa taktiež zameriava práve na vyhlásenia politikov, politických strán a osobností z verejného života, najmä v USA. V procese vyhodnotenia výroku využívajú viac krokový proces. Najskôr tvrdenie preskúma novinár, ktorý overí pôvodné zdroje a následne konzultuje s odborníkmi na danú tému, po ktorom zváži kontext vyhlásenia. Zamestnanec potom rozpíše detailnú analýzu tvrdenia a navrhne verdikt z ich takzvaného `Truth-O-Meter`, ktorý môže nadobúdať hodnoty:

- Pravda – Tvrdenie je pravdivé v danom kontexte a nič podstatné v ňom nechýba.
- Čiastočne pravdivé – Tvrdenie je v podstate pravdivé, no vyžaduje si objasnenie alebo doplniť určité informácie.
- Polopravda – Tvrdenie je čiastočne presné, no vynecháva dôležité detaily alebo vytrháva veci z kontextu, čo prináša priestor pre nejasnosti.
- Čiastočne klamlivé – Výrok obsahuje pravdivé časti, no ignoruje kritické fakty.
- Klamstvo – Tvrdenie je nepresné alebo nepravdivé.
- `Horiace nohavice` – Tvrdenie je nepravdivé a na výsmech.

Takúto analýzu následne posúdi porota troch ďalších redaktorov, ktorí hlasujú o konečnom hodnotení, pre zabezpečenie objektivity procesu. Politifact sa spolieha na profesionálny novinársky proces a odborníkov. Možnosť požiadať emailom o overenie nejakého tvrdenia je veľmi zastaralé, pomalé a najmä neumožňuje škálovateľnosť [7]. Používatelia tu majú možnosť vyhľadávať výroky a tvrdenia podľa amerického štátu, kategórie, konkrétnej politickej osobnosti alebo média.

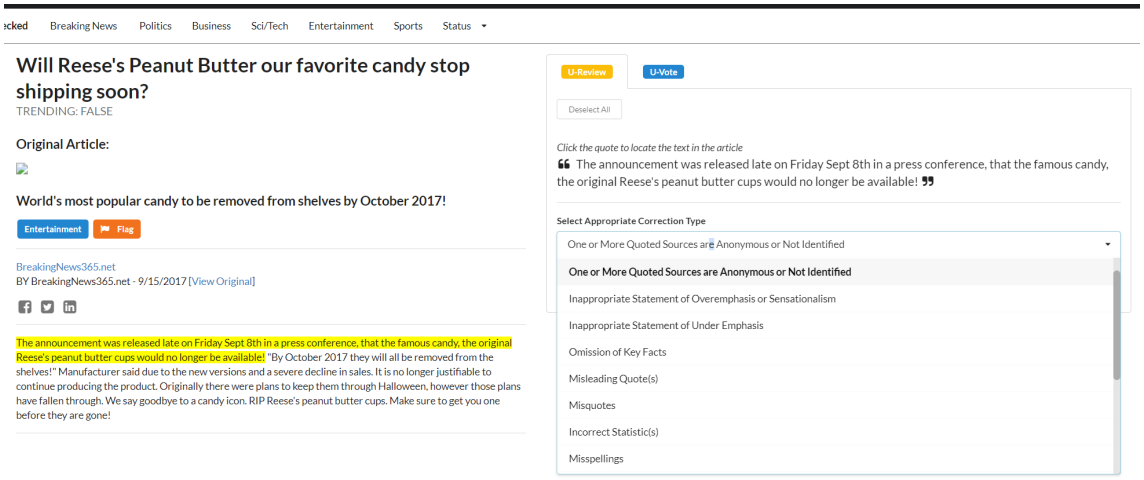
2.4.3 TruthSetter

Platforma TruthSetter na rozdiel od vyššie spomínaných platforiem umožňuje bežným používateľom rozhodovať o pravdivosť tvrdení. Aplikácia nabáda používateľov, aby do systému pridali články a označili tvrdenia, ktoré na internete nájdu, či už sa jedná o blog, sociálnu sieť alebo rôzne iné zdroje infromácií. Prihlásený používateľ zadá URL adresu článku a systém automaticky načíta text alebo ho pridá používateľ manuálne.

Po vložení tvrdenia systém začína demokratický proces analýzy a hlasovania o vytvorených hodnoteniach. Používatelia môžu na platforme hlasovať o tom, či je výrok pravdivý alebo nepravdivý, čím dosiahnu konsenzus davu. Používatelia majú tiež možnosť komentovať tvrdenie, aby dodali kontext, pridali odkazy pre dôkazy, ktoré tvrdenie

potvrdzujú alebo vyvracajú. Platforma TruthSetter používa taktiež aj prvky gamifikácie pre udržanie motivácie u ľudí, ktorí aplikáciu používajú. Získavajú body za rôzne akcie, ktoré v aplikácií vykonajú, napríklad:

- Hlasovanie o článku
- Pridanie textu, ktorý vedie k overeniu faktu
- Pridaním tvrdenia, ktoré viedlo ku overeniu pomocou konsenzu davu



Obrázok 2.4. Pridanie nového tvrdenia na platforme TruthSetter.com [8].

V platforme dokážeme následne nájsť štatistiky a rebríčky najviac aktívnych používateľov, respektíve najlepších overovačov faktov. Toto je veľmi dobrý spôsob vytvárania motivácie u používateľov tejto stránky. Používatelia sa môžu stať súčasťou virtuálnej organizácie na tejto stránke. To môže motivovať odborné organizácie, aby zapojili svojich špecialistov a tým sa zviditeľnili na platforme.

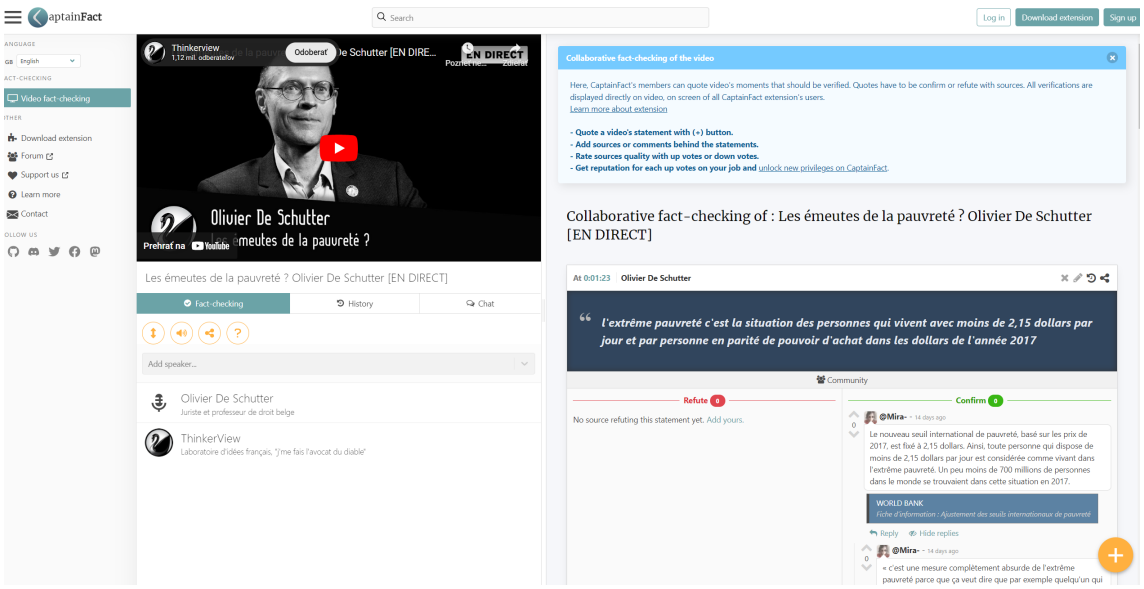
Obrázok 2.5. Proces hlasovania na platforme TruthSetter.com [8].

Vývoj platformy je neaktívny a v procese overovania sa nezapája žiadny automatizovaný proces a žiadny nie je v pláne. Platforma je navrhnutá podobne našej aplikácií, no nie je to softvér s otvoreným kódom. Nie je teda možné využiť nazbierané dáta z tejto platformy ani sa inšpirovať postupmi. Aplikácia poskytuje používateľom aj sadu návodov, ako aplikáciu používať, definíciu bodového systému a návody ako hodnotiť tvrdenia. Platforma obsahuje viaceré chyby a niektoré časti sa javia byť nedokončené.

2.4.4 CaptainFact

Platforma CaptainFact je platforma zameraná na kolektívne overovanie faktov na YouTube videá internete. Ľudia si medzi sebou pomáhajú overovať pravdivosť informácií vo videách a to následovným spôsobom:

- Video je pridané na web stránku
- Tvrdenia su vybraté z daného videa
- Špecialisti na overovanie faktov rozhodnú - potvrdia alebo vyvrátia jednotlivé tvrdenia a podložia ich evidenciou faktických dôkazov.



Obrázok 2.6. Webová stránka platformy CaptainFact [9].

Pomocou rozšírenia v internetovom prehliadači dokáže počas videa upozorniť používateľov na potenciálne zavádzajúce tvrdenia. Platforma nie je automatizovaná. Platforma je zameraná najmä na overovanie tvrdení vo videách [9].

2.4.5 Obmedzenia platforiem

Každá z vyššie spomínaných webových aplikácií má unikátny proces overovania faktov. Niektoré z nich využívajú iba odborníkov v tejto oblasti, iné zapájajú aj neodbornú verejnosť. Niektoré platformy umožňujú používateľom pridávať vlastné tvrdenia a iné si vyhľadávajú samé, čo budú overovať. V každom prípade je proces overovania na týchto platformách zdĺhavý a neumožňuje v reálnom čase vyhľadávať pravdivosti tvrdení, prípadne vkladať nové a hodnotiť ich. Žiadna zo spomínaných platforiem neposkytuje dáta v otvorenej forme, ktoré by boli možné využiť pre automatizované spracovanie. V niektorých prípadoch dokonca pri tvrdeniach chýbajú pôvodné články a celkový kontext daného tvrdenia.

2.4.6 Zhodnotenie novej implementácie

Tento projekt je tvorený v spolupráci s Centrom umelej inteligencie (AIC) na FEL ČVUT. Jedna z priorít tejto softvérovej aplikácie pre overovanie faktov na internete je následný zber dát. Aplikácia musí umožňovať prístup k zozbieraným dátam od používateľov vďaka využitiu sily davu. Dáta musia byť dobre štruktúrované pre následnú analýzu zozbieraných dát pre možnosti tvorby NLP modelov pre overovanie faktov v

AIC. Dáta musia byť jednoducho exportovateľné. Žiadna z existujúcich aplikácií neposkytuje otvorený kód, ktorý by sa dal použiť a ďalej rozširovať pre naše požiadavky. Aj v prípade, že existovala implementácia s otvoreným kódom, nebol to proces overovania faktov pomocou kolektívnej sily davu a demokratického hlasovania. Aj z tohto dôvodu bude potrebné navrhnúť a vyvinúť vlastné riešenie, ktoré bude využívať rôzne stratégie gamifikácie pre udržanie používateľov na platforme a tým zvýši počet zozbieraných dát. Väčším množstvom dát zabezpečíme aj vyššiu mieru transparentnosti v demokratickom hlasovaní a hodnotení tvrdení.

Kapitola 3
Teoretická časť

V následujúcej kapitole sa budeme zaoberať potrebnou teóriou, pre pochopenie návrhu systému z hľadiska architektúry. Následne sa ponoríme do detailov potrebných pre správnu, efektívnu a najmä bezpečnú implementáciu aplikácie s dôrazom na najlepšie techniky a praktiky z praxe.

3.1 Softvérová architektúra

S narastajúcou veľkosťou a komplexitou softvérového systému, dizajn a špecifikácia celkovej štruktúry systému sa stáva výraznejší problém, ako voľba konkrétnych algoritmov a dátových štruktúr pre výpočty. Štruktúra systému zahŕňa hrubú organizáciu systému, globálne kontrolné štruktúry, protokoly pre komunikáciu, synchronizáciu a prístup k údajom. Zahŕňa tiež rozdelenie funkcionality pri návrhu elementov, kompozíciu návrhových elementov, škálovanie systému, výkon a najmä výber a rozhodnutie dizajnových alternatív a vzorov. [10]. Dá sa teda povedať, že architektúra softvérového systému je problematika, ktorá sa zaoberá organizáciou a štruktúrou softvérového systému z toho najvyššieho uhla pohľadu.

3.1.1 Návrhové vzory

Návrhové vzory (v anglickej literatúre Design patterns) pre backend softvérové aplikácie odkazujú na v praxi zavedené princípy návrhu a postupy, ktoré poskytujú štruktúrované pokyny pre tvorbu backend infraštruktúry. Tieto pokyny slúžia ako opakovane použiteľné riešenia bežných problémov s návrhom a umožňujú vývojárom vytvárať škálovateľné, udržiavateľné a efektívne backend aplikácie. Dodržiavaním týchto návrhových vzorov môžu vývojári zabezpečiť logické a dobre štruktúrované rozdelenie komponentov, správnu komunikáciu medzi časťami systému (modulmi) a efektívny tok dát. Vzory zahŕňajú rôzne bežne používané prístupy, z ktorých každý má svoje výhody a nevýhody a ponúka odlišné kompromisy. Príklady návrhových vzorov:

- **Vzory tvorby objektov** - popisujú mechanizmus vytvárania objektov. Zahŕňajú napríklad vzor Singleton - Jeden z najznámejších vzorov, ktorý hovorí, že existuje iba jedna inštancia konkrétnej triedy a poskytuje k nej globálny prístup.
- **Vzory štruktúr** - Popisujú kompozíciu tried a objektov. Do tejto kategórie spadá napríklad vzor Dekorátor, ktorý dynamicky pridáva extra funkcionalitu existujúcim objektom.
- **Vzory správania** - definuje závislosti medzi objektami tak, že keď sa zmení stav jedného objektu - upozornia sa na to všetky objekty na ňom závislé. Známym a často využívaným príkladom je vzor pozorovateľa (Observer). Je to pravdepodobne dokonca jeden z najznámejších návrhových vzorov. Popisuje spoluprácu rôznych objektov bez toho, aby sme do systému zaviedli zlé praktiky (antipattern) tesného prepojenia objektov. Je tiež známy ako vzor `Publish-Subscriber` [11].

3.1.2 Architektonické vzory

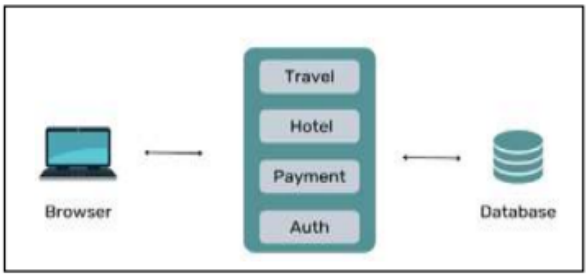
Architektonické vzory sú všeobecné znovupoužiteľné riešenia pre opakovane sa vyskytujúce problémy v návrhu softvérových architektúr. Reprezentujú abstrakciu softvéru z toho najvyššieho uhla pohľadu a poskytujú riešenia, ktoré majú vplyv na komponenty, globálne vlastnosti a mechanizmy celého softvérového systému Tieto vzory popisujú základné komponenty systému a ako medzi sebou budú komunikovať. Známe vzory z pohľadu komunikácie medzi komponentami:

- **Vrstvená architektúra** (Layered) - Vykonávanie komponentov na vysokej úrovni je závislé od komponentov na nižšej úrovni. Funkčnosť komponentov na nižšej úrovni je opäť závislá na komponentoch na ešte nižšej úrovni. Týchto vrstiev môže byť väčší počet, kde každá vrstva spracováva iba svoju oddelenú úlohu.
- **Klient-server** - Dva komponenty, ktoré medzi sebou musia komunikovať. Sú od seba nezávislé (často rozdielne procesy - na rôznych zariadeniach).
- **Udalosťami riadené** (Event-driven) - Architektonický model, ktorý spája distribuované softvérové systémy, a umožňuje efektívnu komunikáciu, ktorá je realizovaná najmä vyššie spomínaným `Publish-Subscribe` návrhovým vzorom.
- **Mikroslužby** (Microservis) - Je architektonický vzor, kde softvérový systém je štruktúrovaný do menších oddelených celkov, ktoré nazývame služby. Každá služba je navrhnutá a vyvíjaná okolo konkrétnej časti biznis logiky. Jednotlivé služby môžu byť následne nezávisle vyvíjané, testované, nasadzované a škálovateľné. Mikroslužby sú teda oddelené procesy a môžu medzi sebou komunikovať pomocou mechanizmov ako napríklad HTTP/REST alebo front správ.

Porozumením rôznych, vyššie spomenutých softvérových architektúr a vzorov a ich následná aplikácia v praxi pomáha vývojárom robiť informované rozhodnutia o návrhu. To vedie k vývoju spoľahlivejších, efektívnejších a kvalitnejších softvérov [12]

3.1.3 Monolitická architektúra

Jedna z najstarších softvérových architektúr je práve monolitická architektúra. Cieľom je vytvoriť softvérovú aplikáciu ako samostatnú jednotku. Funkcionalita systému a komponenty sú navzájom závislé, úzko prepojené a často ich nie je možné skompilovať samostatne. S narastajúcim množstvom požiadaviek na softvér v monolite neustále narastá počet prerekvizít, knižníc a závislostí. Monolit obsahuje definované triedy, funkcie a menné priestory pre celú aplikáciu.



Obrázok 3.1. Príklad monolitickej architektúry backend aplikácie [13].

Benefity monolitu:

- Všetka logika, ktorá spracováva požiadavky sa spúšťa ako jeden proces
- Testovanie funkcionality a písanie testov je triviálne
- Monitorovanie je jednoduché

- Vývojári nemusia brať do úvahy zabezpečenie komunikačných kanálov medzi službami
- Lacnejšie a jednoduchšie nasadenie aplikácie
- Monolitické aplikácie môžu mať často lepší výkon

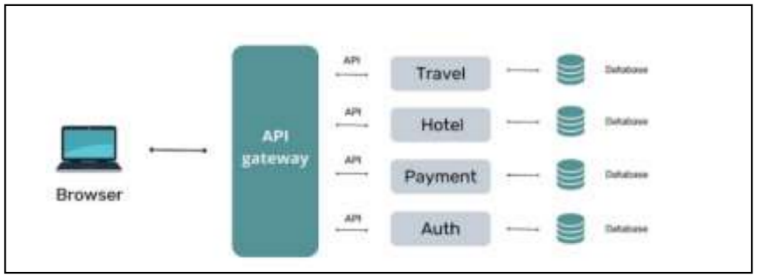
Negatíva monolitu:

- Pochopenie - monolit neustále rastie a tým sa zväčšuje komplexita a čas pochopenia
- Zavádzanie zmien je náročnejšie
- Škálovateľnosť
- Bariéry pri zavádzaní nových technológií

Tento prístup zjednodušuje počiatočný vývoj a nasadzovanie, keďže máme iba jeden zdrojový kód a databázu. Problém nastáva pri neustálom rozširovaní a zväčšovaní systému a následne zavádzať čo i len malé zmeny, je výzva. Tento prístup robí udržovanie a škálovanie softvérového systému v dlhodobom hľadisku náročné, často až neudržateľné [13].

3.1.4 Mikroslužbová architektúra

Slovo mikroslužba dostáva v poslednej dobe veľkú pozornosť - začínajúc spoločnosťami ako Netflix, Amazon, Spotify a podobne. Mikroslužbová architektúra je metóda vývoja distribuovaných systémov, ktoré majú adresovať a riešiť nedostatky monolitickej architektúry. Biznis logika je rozdelená do malých, jednoúčelových nezávislých celkov - servisov, ktoré medzi sebou komunikujú pomocou presne zadefinovaných API rozhraní. Každá mikroslužba zabezpečuje určitú špecifickú časť celkovej biznis logiky. Môžu byť vyvíjané, nasadzované a škálované samostatne. Každá služba by si mala zachovať svoju autonómiu, aby bola zachovaná hlavná myšlienka tohto architektonického vzoru [13].



Obrázok 3.2. Príklad mikroslužbovej architektúry backend aplikácie [13].

Benefity mikroslužieb:

- Nezávislé komponenty (vývoj, zmeny, inštalácia)
- Jednoduchšie pochopenie kódu
- Lepšia škálovateľnosť
- Možnosť pracovať agílnejšie

Negatíva mikroslužieb:

- Extra komplexita
- Distribúcia systému
- Pridaná zložitosť pri konfigurácií, logovaní, monitoringu
- Testovanie - veľké množstvo nezávisle nasadzovaných komponentov

Prístup mikroslužieb ponúka výhody, ako škálovateľnosť, izolácia chýb a bugov alebo flexibilita pri pridávaní rôznych technológií. Pridáva však extra zložitosť súvisiacu s koordináciou služieb, sieťovou komunikáciou a konzistentnosťou údajov [13].

3.1.5 Ktorú architektúru zvoliť?

Odpovedať na túto otázku je možné len po analýze biznisových požiadaviek na daný projekt. Je nutné poznať finančný rozpočet, odhady, pracnosť a taktiež vedomosti a schopnosti vývojárov, ktorí budú na projekte pracovať. Malé spoločnosti a startupy často môžu využiť výhody monolitickej architektúry - najmä z jednoduchosti a rýchlosti počiatočného vývoja a nasadenia. Na druhú stranu, výhody mikroslužieb využijeme pri zavádzaní zmien do štruktúry aplikácie.

Kedy zvoliť monolitickú architektúru:

- Produkt musíme nasadiť čo najrýchlejšie
- Plán je vytvoriť jednoduchú aplikáciu a neplánujeme do budúcna žiadne veľké zmeny
- Chýbajú nám špecialisti, ktorí dokážu rozdeliť systém do mikroslužieb

Kedy zvoliť mikroslužbovú architektúru:

- Cieľom je vyvinúť rozsiahly, sofistikovaný systém
- Máme špecialistov, ktorí majú skúsenosti a technické vedomosti s vývojom mikroslužieb
- Systém musí byť škáľovateľný
- Je potrebné zabezpečiť udržiavatelnosť
- Schopnosť, respektíve jednoduchosť systému integrovať s inými systémami

Niektorí odborníci z praxe tvrdia, že aplikácia by sa mala zo začiatku vyvíjať ako monolit, z dôvodov jednoduchšej počiatočnej implementácie a časom zmeniť architektúru na mikroslužbovú. Hlavnou podstatou vyššie spomínaných architektonických vzorov ale je, že už nebude nutné meniť architektúru aplikácie. Výber optimálnej architektúry teda vždy závisí na požiadavkách každého projektu a je nutné zvážiť rôzne faktory [13].

3.2 Databázové systémy

Databázové systémy, najmä pri webových aplikáciach sú základným prvkom softvérového systému. Umožňujú efektívne ukladanie, vyhľadávanie a manažment veľkého množstva dát. Z tohto dôvodu je výber vhodného databázového systému mimoriadne dôležitý - ovplyvňuje výkon, bezpečnosť, spoľahlivosť aj škálovatelosť. V následujúcej kapitole si porovnáme výhody a neývhody rôznych databázových systémov, aby sme následne mohli robiť informované rozhodnutia pri výbere.

3.2.1 SQL Databázové systémy

SQL databázy sú založené na relačnom modely. Základným prvkom SQL databáz sú tabuľky s definovanými reláciami medzi nimi. Tieto relácie zabezpečujú jednoduchosť dopytovania - vyhľadávanie a celkovú manipuláciu a analýzu dát pomocou jazyka SQL (Structured Query Language). Výhody SQL vidíme najmä pri práci so štruktúrovanými dátami s preddefinovanou schémou - preto sú vhodné v aplikáciach, kde potrebujeme zabezpečiť konzistenciu a integritu uložených dát.

Silné vlastnosti SQL (ACID):

- Atomicita - Využíva tzv. **transakcie** - postupný proces vykonávania, ktorý sa skladá z logických jednotiek. Buď sa vykonajú všetky kroky, alebo žiadna z nich. Atomicita zaručuje, že databáza bude fungovať v každej situácií, vrátane výpadkov napájania, chýb a zlyhaní systému - v opačnom prípade nastane takzvaný `rollback` - návrat do pôvodného stavu.

- Konzistencia - Transkacia vytvorí platný stav údajov. Pri zlyhaní by sa mali dostať všetky údaje do pôvodného stavu.
- Izolácia - Táto vlastnosť zabraňuje konfliktom medzi súbežnými transkaciami.
- Vytrvalosť (Durability) - Údaje by mali byť dostupné v správnom stave aj po zlyhaní alebo reštarte systému [14].

SQL respektíve DBMS (Database Managment Systems) ponúkajú výkonné optimalizačné techniky, mechanizmy indexovania, normalizácie dát pre zabezpečenie efektívneho dopytovania a manipulácie s dátami. SQL databázy sú využívané v mnohých odvetviach, v enterprise systémoch, webových aplikáciach, dátovej analytike a mnohých ďalších. Flexibilita SQL umožňuje architektom a vývojárom modelovať zložité relačné vzťahy, zabezpečiť pravidlá a vykonávať komplexné dotazy [15].

Výhody SQL:

- Dáta sú štruktúrované.
- Dodržovanie `ACID` vlastností.
- SQL je medzi vývojármi všeobecne rozšírenejšie.
- Množstvo SQL databáz s otvoreným kódom (`Open-source`). Veľká podpora komunity.

Nevýhody SQL:

- Dodržiavaním `ACID` princípov sa znižuje škálovateľnosť. Typické škálovanie SQL databáz je vertikálne - zvyšovaním výkonu hardvéru.
- Dáta musia byť dopredu definované a normalizované - chýba flexibilita.
- Komplexné dotazy, ak máme príliš veľa relácií medzi dátami.

3.2.2 NoSQL Databázové systémy

NoSQL (**Not Only SQL**) sa stáva viac a viac rozšírené pri neustálom zväčšovaní dát, ktoré sa na internete zbierajú. Dáta sú typicky príliš komplexné, neštruktúrované a veľmi náročné reprezentovať klasickými SQL tabuľkami a reláciami medzi nimi. NoSQL majú rôzne formy reprezentácie dát. Existujú databázy typu kľúč-hodnota, dokumentové, grafové prípadne NoSQL databázy založené na určitom type stĺpcov. NoSQL databázy všeobecne dokážu spracovať dáta rýchlejšie ako tradičné SQL databázy, keďže nie vždy sú zachované vyššie spomínané `ACID` vlastnosti. Dátové modely NoSQL databáz sú väčšinou jednoduchšie [16].

Vlastnosti NoSQL (BASE):

- V podstate dostupné (Basically Available) - Preferujeme dostupnosť nad konzistentnosťou - v prípade výpadkov siete.
- Mäkký stav (Soft state) - Akceptujeme, aby systém mohol byť na čas nekonzistentný.
- Nakoniec konzistentné (Eventually consistent) - Systém sa môže dostať do nekonzistentného stavu - no v čase sa opäť musí dostať do konzistentného stavu [17]

Výhody NoSQL:

- Horizontálne škálovanie - pridávaním počtu serverov.
- Flexibilita pri počiatočnom vývoji - nevyžaduje si striktné schémy a dáta môžu byť dokonca neštruktúrované.
- Lepší výkon pri určitých typoch operácií v niektorých databázach.

Nevýhody NoSQL:

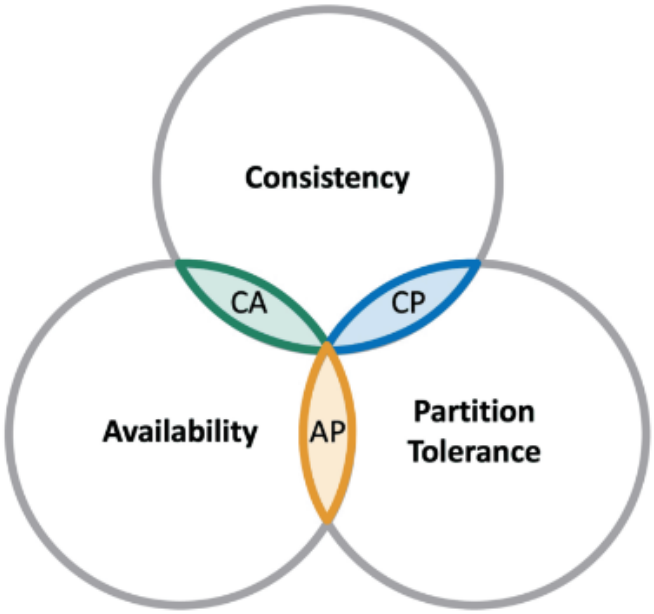
- Nedodržuje `ACID` princípy
- Množstvo dopytovacích jazykov pre rôzne NoSQL systémy (dokumentové, grafové, databázy typu kľúč-hodnota a iné).
- NoSQL databázy sú novšie a zatiaľ nemajú tak veľkú podporu komunity.

3.2.3 CAP teorém

CAP teorém je teorém, ktorý tvrdí, že distribuovaný systém môže spĺňať maximálne 2 z 3 vlastnosti z následujúceho listu:

- Konzistentnosť (Consistency) - Dáta sú na všetkých partíciach distribuovaného výpočtového systému naraz.
- Dostupnosť (Availability) - Dáta sú neustále dostupné pre čítanie a zmeny.
- Tolerancia rozdelenia (Partition tolerance) - Distribuovaný výpočtový systém musí pokračovať v správnom fungovaní aj napriek zlyhaní v sieti, pri ktorom niektoré partície môžu byť na čas nedostupné.

Pri systémoch s vlastnosťami ACID je náročné dosiahnuť vysokú škálovateľnosť, pretože začnú vznikať konflikty medzi rôznymi aspektami vysokej dostupnosti, ktoré sú často neriešiteľné [18].



Obrázok 3.3. CAP teorém, najviac 2 z 3 vlastností môžu byť splnené [18].

3.2.4 MongoDB

MongoDB je dokumentová NoSQL databáza. Reprezentuje a ukladá štruktúrované dáta ako JSON dokumenty s dynamickými schémami (interne sa tento formát nazýva BSON - Binary JSON). V porovnaní s tradičnými SQL databázami, MongoDB má rýchlejší beh pre požiadavky na vkladanie, zmeny a jednoduché `query` (dopytovanie sa na dáta). MongoDB je dobrá voľba pre aplikácie, ktoré potrebujú spracovať väčšie dáta, ktorých schéma sa môže často meniť [19].

Nalepšie praktiky - ako navrhovať MongoDB schémy:

Aj napriek tomu, že MongoDB dokáže pracovať s neštruktúrovanými dátami, ukážeme si najlepšie techniky návrhu schém pre tento systém. Správny návrh je kritický pre rýchly a škálovateľný systém. V opačnom prípade nebudeme schopný využiť plný potenciál tohto databázového systému naplno. Pri návrhu schémy sa musíme zamerať najmä na to, čo bude vyhovovať našej aplikácií - dve aplikácie, ktoré používajú rovnaké dáta môžu mať úplne odlišné schémy, ak sú používané iným spôsobom [20].

Pri návrhu schémy pre MongoDB sa musíme často rozhodnúť, či v danej situácií budeme preferovať vkladanie alebo odkazovanie. Rozoberieme si najčastejšie prípady použitia a preferovaný prístup:

- 1:1 (One-to-One) - Preferovať vkladanie kľúč-hodnota v rámci daného dokumentu.
- 1:Niekoľkým (One-to-Few) - Taktiež uprednostňovať vkladanie do dokumentu, pokiaľ nie je žiadny presvedčivý dôvod proti tomu.
- 1:Veľa (One-to-Many) - Ak potrebujeme prístup k týmto objektom, je to dobrý dôvod, aby sme tento objekt nevkladali, ale preferovali vytvorenie vlastnej kolekcie. Je dobrým zvykom snažiť sa vyhnúť operáciám spájania a vyhľadávania (`joins`, `lookups`). No ak by to znamenalo zlepšenie celkového návrhu schémy, nie je to žiadny problém.
- 1:Miliónom (One-to-Squillions) - Vnorené polia by nemali neobmedzene rásť (každý BSON objekt je obmedzený na 16MB). Ak je na strane `veľa` viac ako stovky dokumentov - nevkladajte ich. Ak je na strane `veľa` viac ako tisícky dokumentov, nepoužívajte pole referencií (ObjectId). Polia s veľkým počtom objektov alebo referencií sú dobrým dôvodom prečo ich nevkladať.
- N:N (Many-to-Many) - Obe kolekcie na strane `veľa` musia mať uložené referencie v poli [20].

Vždy, keď modelujeme systém pomocou databázy MongoDB, tento proces by mal nasledovať spôsoby a vzory prístupov k dátam zo strany konkrétnej aplikácie. Vždy chceme štruktúrovať svoje dáta tak, aby sa zhodovali so spôsobom, akým ich naša aplikácia získava (dopytuje sa na ne) a aktualizuje [20].

3.2.5 Ako zvoliť databázový systém

Na otázku, ktorý z vyššie spomínaných databázových modelov je lepší neexistuje jednoznačná odpoveď. Pri návrhu softvérového systému je nutné dospieť k záveru na základe biznisových požiadaviek daného projektu a následne, softvérový architekt musí zvážiť možnosti a vykonať informované rozhodnutie. SQL databázy, respektíve databázové systémy, ktoré dodržiavajú `ACID` princípy sú vhodnejšie v prípade, že musíme zabezpečiť neustálu konzistenciu dát, predvídavosť a spoľahlivosť. Naopak, vlastnosti `BASE` sú vhodné pre aplikácie, kde sa očakáva neustály a veľký nárast dát (pre takzvané systémy veľkých dát). Poskytujú väčšiu flexibilitu, lepšie škálujú a v neposlednom rade sú vhodné pri agílnom type vývoja softvéru. Na druhú stranu, pri používaní databázových systémov s BASE vlastnosťami musíme dbať a myslieť na tieto obmedzenia aj pri implementácií funkcionality. Každý databázový systém má svoje obmedzenia, silné a slabé vlastnosti. Výber toho správneho má často zásadný vplyv na celkovú efektivitu aplikácie.

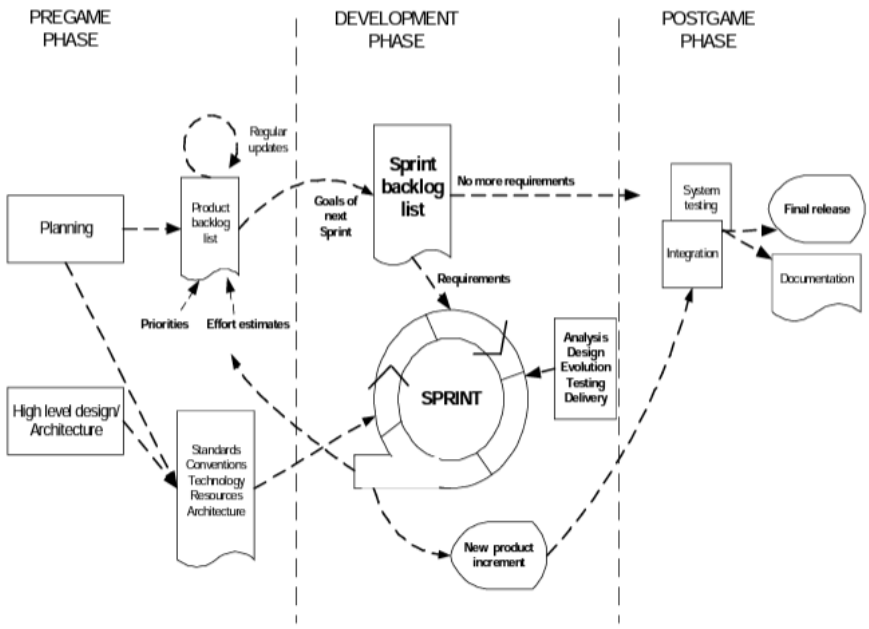
3.3 Metodiky vývoja

Metodiky softvérového vývoja sú štruktúrované postupy, ktoré majú usmernovať celkový proces vývoja softvérových systémov. Ponúkajú princípy, postupy a procesy pre manažment a organizáciu softvérových projektov rôznych veľkostí.

- Vodopádová metodika - Historicky najstaršia metodika vývoja, ktorá dodržiava určitú sekvenciu krokov. Zahŕňa viaceré fázy ako zber požiadaviek, návrh systému, vývoj systému, následne testovanie a nasadenie. Každá fáza musí byť dokončená pred postupom do ďalšej fázy.
- Agílna metodika - Prioritizujú flexibilitu, kolaboráciu a takzvaný iteratívny proces vývoja. Dôležitejšia je adaptácia a pripravenosť na zmeny požiadaviek. Funkcionalita sa vyvíja postupne, po malých prírastkoch, v agílnom slangu známe ako šprinty. Preferovaná je častá komunikácia, samostatná organizácia v rámci tímu a neustály feeback. Poznáme rôzne alternatívy agílneho vývoja, ktoré si rozoberieme v ďalšej kapitole.

3.3.1 Scrum

Scrum je kolekcia postupov a procesov pre agílny projektový manažment, aplikovaný aj pri vývoji softvéru. Kladie dôraz na tímovú prácu a iteratívny proces k dopredu definovanému cieľu. Nedefinuje žiadne špecifické techniky pre vývoj. Popisuje iba fungovanie členov tímu v neustále sa meniacom prostredí. Scrum pomáha zlepšovať postupy v organizácií, pretože poskytuje mnohé manažérske aktivity zamerané na identifikáciu prekážok a nedostatkov v procese vývoja [21].



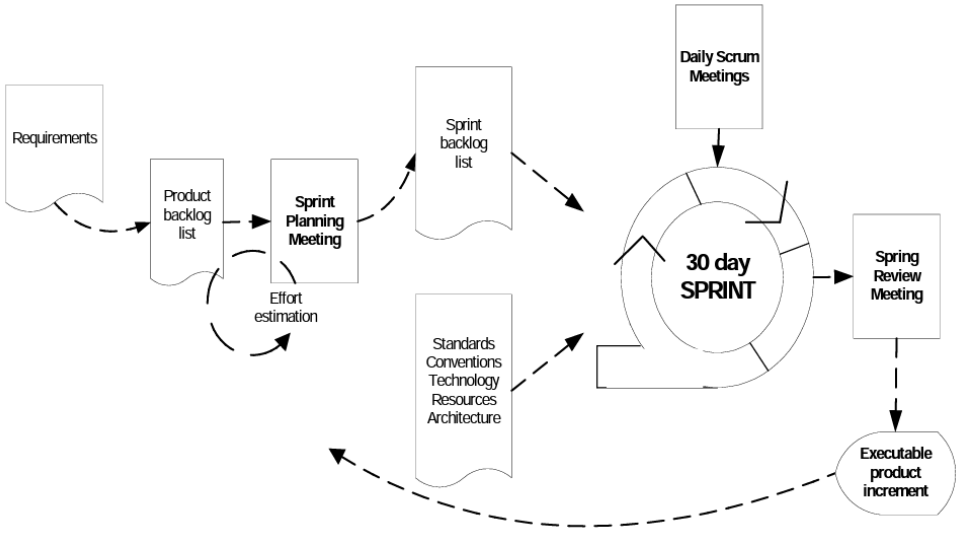
Obrázok 3.4. Fázy SCRUM metodológie [21].

V metodológií Scrum musíme určiť role jednotlivým členom tímu, pre efektívnu spoluprácu a manažment projektu.

- **Vlastník produktu** (Product owner) - je účastník, ktorý zastupuje biznisový alebo používateľský pohľad na vývoj. Definuje úlohy a ciele, ktoré sa musia vykonať. Tieto úlohy a ciele spisuje do takzvaného `projektového backlogu`. Produktový backlog je zoznam úloh a požiadaviek, ktoré musia byť vyriešené a dodané pre konečný produkt.
- **Scrum master** - Je sprostredkovateľ medzi vývojovým tímom a produktovým vlastníkom. Jeho úloha je kontrolovať, či vývoj dodržuje definované pravidlá, procesy a

praktiky. Taktiež dohliada na časový harmonogram procesu vývoja a progres. Je zodpovedný za odstraňovanie prekážok, aby tím mohol pracovať čo najproduktívnejšie

- **Vývojový tím** - Skladá sa z profesionálov, ktorí vykonávajú softvérový návrh, vývoj, testovanie a nasadenie tak, aby uvoľnili produktový prírastok (increment) [21].



Obrázok 3.5. Sprint metodológie Scrum [21].

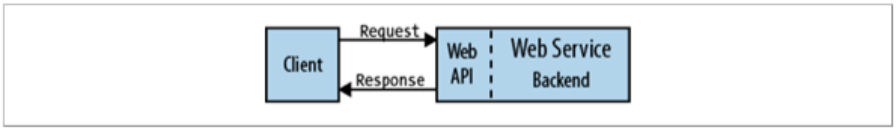
Procesy Scrumu:

- **Sprint** - Je časovo ohraničený rámec. Tím sa sám organizuje, aby v tomto časovom rámci bol schopný vyprodukovať ďalší prírastok. V praxi môže mať rôzne dĺžky trvania, od 10 dní do celého kalendárneho mesiaca.
- **Plánovanie sprintu** - Dvojfázové stretnutie organizované Scrum masterom, kde sa rozhodnú cieľe pre najbližší sprint.
- **Denný scrum** - Je denné stretnutie členov tímu, kde každý člen popíše svoj progres a čo je ďalej nutné dokončiť. Nemalo by trvať dlhšie ako 15 minút.
- **Kontrola sprintu** - Posledný deň sprintu prebieha stretnutie pre kontrolu progresu. Vývojári, scrum master, používatelia, produktový vlastník a manažment sa stretne a predstaví sa nový prírastok, ktorý bol pridaný v tomto sprinte. Účastníci diskutujú o novom prírastku a rozhodnú sa pre nadchádzajúce aktivity, ktoré sa môžu pridať do backlogu.
- **Retrospektíva sprintu** - Je príležitosť pre tím scrumu, aby skontroloval svoj plán a navrhol a vytvoril vylepšenia, ktoré sa majú pridať v ďalších sprintoch [21].

3.4 REST API

Webové servisy typu **REST** sú realizáciou architektonického štýlu REST pre tvorbu webovo založených API rozhraní (Aplikačné programové rozhranie). Architektúra REST sa riadi množinou princípov, vďaka ktorým webové servisy zabezpečia interoperabilitu a škálovatelnosť. Webové služby, ktoré dodržiavajú REST princípy využívajú štandardné HTTP metódy (GET, POST, PUT, PATCH, DELETE) na vykonávanie operácií so zdrojmi. Zdroje sú vždy identifikovateľné podľa jedinečného identifikátora a následne vďaka jedinečným URL adresám, respektíve URI (Uniform Resource Identifier). Služby REST využívajú bezstavovosť protokolu HTTP. Klienti môžu jednoducho integrovať

s webovými servismi typu REST odosielaním HTTP požiadaviek. Jednoduchú integráciu zabezpečuje presne definované API rozhranie. Klientské aplikácie môžu prijímať odpovede v rôznych formátoch, či už JSON, XML alebo iných. Webový servis teda poskytuje API - množinu dát a funkcií, vďaka ktorým je možná komunikácia medzi dvoma procesmi. Ako vidíme na obrázku 3.6, webové API (rozhranie) je tvár webovej služby, ktorá čaká a reaguje na požiadavky klientských aplikácií.



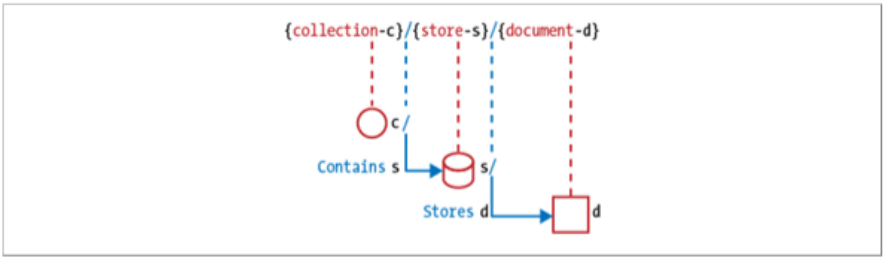
Obrázok 3.6. Webové rozhranie (API) [22].

Architektonický štýl REST je často používaný a aplikovaný pri návhrhoch webových rozhraní pre moderné webové služby . Webové rozhranie, ktoré podlieha architektonickým princípom REST sa nazýva RESTful API, ak pozostáva zo vzájomne prepojených zdrojov. Táto množina zdrojov sa nazýva aj model zdrojov rozhrania REST [22].

3.4.1 Návrh REST API

Návrh REST rozhraní je viac umenie a skúsenosti softvérového architekta, než exaktná veda. Tento architektonický vzor síce ponúka mnohé princípy, no niektoré prípady použitia nemajú konkrétne formáty, ktoré treba dodržiavať. Niektoré osvedčené postupy sú už zabudované v HTTP štandardoch, zatiaľ čo iné pseudo-štandardy a postupy sa neustále vyvíjajú vďaka komunite. Uvedieme si najčastejšie pravidlá pri navrhovaní webových rozhraní REST [22].

- Hierarchický vzťah označujeme oddeľovačom lomka (/). Napríklad https://factcheck.cz/articles/:articleId/claims/
- Na zvýšenie čitateľnosti by sme mali používať pomlčku (-) a naopak sa vyhnúť znaku podčiarknutia (_)
  - https://factcheck.fel.cvut.cz/new-claims
- Vždy preferovať názvy URI s malými znakmi
- Pre názvy dokumentov používať singulár, no pre názvy kolekcií používať plurál
- CRUD operácie by sa nemali zahŕňať do názvu URI
- Query komponent URI by mal byť použitý pri filtrovaní z kolekcií alebo úložiska :
  - GET https://factcheck.fel.cvut.cz/users
  - GET https://factcheck.fel.cvut.cz/users?role=admin
- Query komponent by mal byť použitý pre stránkovanie kolekcií a výsledkov z databáz
  - GET https://factcheck.fel.cvut.cz/users?page=25&perPage=50
- Metóda GET musí byť použitá pre získanie reprezentácie zdroja alebo zoznamu zdrojov v kolekcií
- Metódu PUT používať na vkladanie a zároveň aktualizáciu udájov o zdroji v kolekcií
- Metóda POST musí byť použitá pre vytváranie zdrojov v kolekcií
- Metóda DELETE musí byť použitá pre odstránenie zdrojov v kolekcií [22]



Obrázok 3.7. Diagram URI, model zdrojov a ich asociací [22].

Existuje mnoho ďalších pravidiel, no vyššie sme si uviedli základné a často používane, od ktorých sa architekt systému pri návrhu systému môže odraziť.

3.5 GraphQL API

GraphQL je dotazovací jazyk a API runtime s otvoreným kódom, vytvorený pôvodne interne spoločnosťou Facebook v roku 2012 a následne zverejnený v roku 2015. GraphQL ponúka alternatívny prístup k vyššie spomínanému rzohraniu REST API. Na rozdiel od REST, ktorý zvyčajne vyžaduje viacero požiadaviek na rôzne koncové body pre rôzne údaje, GraphQL umožňuje klientom presne špecifikovať údaje, ktoré potrebujú. A to všetko priamo v jednej požiadavke. Tento spôsob dopytovania znižuje nadmerné načítanie údajov, čo vedie k efektívnejšiemu dopytovaniu dát. GraphQL tiež poskytuje silne typizovanú schému, ktorá definuje dostupné údaje a operácie, čo klientom uľahčuje vyhľadávanie a interakciu s API. GraphQL navyše podporuje aktualizácie v reálnom čase prostredníctvom takzvaného subscription, čo umožňuje klientom prijímať aktualizácie údajov v reálnom čase bez potreby dotazov. GraphQL si získal popularitu ako výkonný nástroj na vytváranie efektívnych a flexibilných API. Pôvodne bolo GraphQL API v spoločnosti Facebook používané ako alternatíva REST API pre mobilné zariadenia, pretože umožňuje znížiť množstvo dát prenášaných po sieti [23].

```
// a GraphQL query
author (id:"1") {
    id
    name
    avatarUrl
    articles (limit: 2) {
        name
        urlSlug
    }
}
```

```
// a GraphQL query result
{
  "data":{
    "author":{
      "id":"1",
      "name":"RobinWieruch",
      "avatarUrl":"https://domain.com/authors/1",
      "articles":[
        {
```

```
            "name":"TheRoadtolearnReact",
            "urlSlug":"the-road-to-learn-react"
          },
          {
            "name":"ReactTestingTutorial",
            "urlSlug":"react-testing-tutorial"
          }
        ]
      }
    }
  }
```

Môžme si všimnúť, že máme iba jednu query pre viaceré zdroje (author, article). Priamo v požiadavke sa nachádza špecifikácia atribútov, ktoré očakávame aby boli zo servera odoslané, aj napriek tomu, že daná entita obsahuje mnoho ďalších atribútov vo svojej GraphQL schéme. REST API architektúra by potrebovala aspoň dve po sebe následujúce API volania pre získanie entity autora a následne jeho článkov. Použitím GraphQL vieme túto požiadavku odoslať v jednom requeste [23].

Výhody GraphQL:

- Deklaratívne načítanie údajov - Klient si volí atribúty entít, ktoré potrebuje
- Žiadne nadmerné načítanie (nepotrebných dát) - mobilné zariadenia zvyčajne načítavajú príliš mnoho dát, ak sa používa rovnaké API s webovým klientom
- Využíva moderné trendy
- Silne typizovaný - GraphQL dotazy sú pisané v takzvanej GraphQL SDL (Schema Definition Language)
- Neustále rastúci ekosystém [23]

Nevýhody GraphQL:

- Zložitosť dotazov - GraphQL je iba dotazovací jazyk, serverová aplikácia musí vykonať prístupy do databáz k rôznum tabuľkám (articles, users, ...). Problémy nastávajú, ak klient žiada o príliš veľa vnorených entít a atribútov. Preto sú nevyhnutné mechanizmy ako maximálna hĺbka dotazu, váhovanie komplexity dotazu, vyhýbanie sa rekurzií a podobne.
- Limitovanie prístupov (rate limiting) - pre GraphQL operáciu nie je jednoduché dopredu určiť, či je daný dotaz jednoduchý alebo príliš komplexný.
- Efektívne využívanie cache pamäte [23]

3.6 12-faktorová aplikácia

12-Faktoroá aplikácia je metodológia vývoja moderných, škálovateľných a najmä udržiavateľných webových aplikácií. Pôvodne definovaná zakladateľom spoločnosti a cloudového poskytovateľa Heroku a stala sa široko rozšírenou množinou princípou pre vývoj cloudových aplikácií.

- Codebase - jedna kódová základňa, ktorá je verzovaná pomocou nástrojov pre správu verzií. Preferovať časté nasadenia.
- Depencencies - explicitne definované a izolované závislosti aplikácie.
- Config - Konfiguračný súbor uložený v behovom prostredí.
- Backing services - Zaobchádzať so záložnými servismi ako s pripojenými zdrojmi.
- Build, release, run - Striktná separácia fáz výstavby a behu aplikácie

- Process - Aplikáciu spúšťať ako jeden bezstavový proces
- Port binding - Služby exportovať pomocou väzby portov
- Concurrency - Škálovať cez procesný model
- Disposability - Maximalizácia robustnosti pomocou rýchleho spustenia a jemného ukončenia.
- Dev/Prod parity - Vývojové prostredia a produkčné prostredia čo najviac podobné.
- Logs - Logovanie ako tok udalostí
- Admin processes - Admin a manažérske úlohy ako jeden z procesov.

Využitím a aplikovaním týchto princípov vývojár zaistí osvedčené postupy a vytvorí aplikáciu, ktorá je ľahko nasaditeľná, škálovateľná a manažovateľná v moderných cloudových prostrediach [24].

3.7 Zaistenie bezpečnosti používateľských dát

Zaistenie bezpečnosti používateľských dát musí byť jedna z najvyšších priorít každej aplikácie. Musíme zabezpečiť ochranu citlivých údajov používateľov a predchádzať neoprávneným prístupom. Na tieto účely existujú rôzne techniky, algoritmy a postupy pre zabezpečenie webových aplikácií. Využívajú sa pri tom kryptografické mechanizmy ako JWT a hashovacie algoritmy, ktoré si rozoberieme v ďalších kapitolách.

3.7.1 Hashovanie hesiel

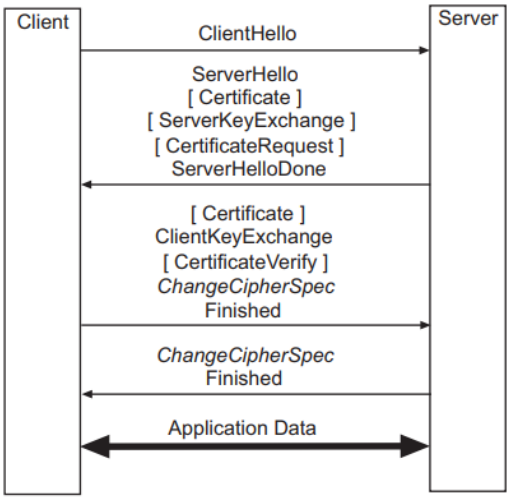
Hashovanie je kryptografický proces, respektíve transformácia dát do reťazca fixnej dĺžky. Využíva sa napríklad pri overeniach integrity údajov a ukladaní hesiel v databáze. Ukladanie hesiel v otvorenom formáte je okrem legálneho hľadiska (GDPR) nebezpečné pre používateľov systému. Pri akomkoľvek neoprávnenom prístupe do databázy alebo úniku údajov by bolo možné vidieť zoznamy hesiel používateľov.

BCRYPT je často používaná hashovacia funkcia v moderných webových aplikáciach pre bezpečné ukladanie hesiel. Je založená na šifre Blowfish, ktorá je známa pre jeho schopnosť spomaliť útoky hrubou silou. Využíva takzvanú soľ. BCRYPT je adaptívna funkcia, ktorá dokáže zostať odolná voči útokom hrubou silou aj pri neustále narastajúcej výpočtovej sile moderných počítačov. BCRYPT je aktuálne bezpečnostný štandard pre hashovanie hesiel [25]. Toto tvrdenie o aktuálnom štandarde je z roku 2015.

Argon2 - je víťazný hashovací algoritmus pre heslá z roku 2015. Tento algoritmus je navrhnutý ako pamäťovo náročný, no rezistentný voči novým útokom. Pamäťovo náročný znamená, že počas procesu hashovania je vyžadované určité množstvo pamäte. Vďaka tomu je viac odolný voči paralelnému procesovaniu ako napríklad útokmi založených na GPU alebo takzvaných ASIC útokoch (ASIC je špecializovaný hardvér pre lámanie hesiel). Napomáha znižovať výhody útočníkov, ktorí sa snažia získať prístup pomocou špecializovaných hardvérov [26].

3.7.2 SSL - Šifrovaná komunikácia

SSL (Secure Sockets Layer) je štandardný bezpečnostný protokol pre zabezpečenie šifrovanej komunikácie medzi webovým prehliadačom a serverom. Využíva sa najmä na ochranu citlivých informácií prenášaných cez internet ako prihlasovacie údaje (heslá), kreditné karty a podobne.



Obrázok 3.8. Proces SSL podania rúk [27].

SSL proces začína klientom (napríklad používateľ vo webovom prehliadači), ktorý sa chce pripojiť k serveru zabezpečenému SSL. Nasleduje proces podania rúk (`Handshake`), kde klient požiada webový server o identifikáciu. Server odošle kópiu svojho SSL certifikátu spolu s verejným kľúčom servera. Klient overí certifikát u certifikačnej autority, ktorá certifikát vydala. Ak je certifikát dôveryhodný, klient vygeneruje symetrický kľúč relácie a zašifruje ho použitím verejného kľúča servera a potom ho odošle späť serveru. Server následne dešifruje symetrický kľúč relácie pomocou svojho súkromného kľúča a odošle späť potvrdenie zašifrované kľúčom relácie. Šifrovaná relácia začína. Od tohto momentu klient a server šifrujú všetky prenášané dáta pomocou kľúča relácie [27].

3.7.3 JWT tokeny

JWT (Json Web Token) umožňuje digitálne bezpečnú reprezentáciu a výmenu nárokov (anglicky claims) medzi dvoma alebo viacerými skupinami na internete. Tieto nároky sú reprezentované v JSON formáte (The JavaScript Object Notation). Následne môžu byť tieto nároky zašifrované (JWE - Encrypted) alebo digitálne podpísané (JWS - Signed) [28].

Každé JWS sa skladá z 3 častí:

- **Hlavička** - Opisuje základné parametre, ako je dané JWT podpísané. Typ algoritmu, a typ tokenu.
- **Náklad** (Payload) - Telo tokenu, ktoré sa podpisuje. Popisuje nároky v JSON formáte.
- **Podpis** - Podpis správy (tokenu), použitím algoritmu definovaného v hlavičke, zakódovaný do base64.

Spojením bodkami „ . “ vyššie spomenutých troch vlastností JWT tokenov nám vznikne plnohodnotný JWS token. Tento token obsahuje množinu hodnôt typu kľúč-hodnota a je možné overiť, či daný token nebol zmenený a použiť ho na autentifikáciu a autorizáciu používateľov do aplikácie.

Obrázok 3.9. Príklad JWT podpísaných nárokov vo forme JWS.

Na predchádzajúcom obrázku vidíme príklad podpísaných nárokov. Nami implementovaná autorizácia a autentifikácia pracuje práve s JWS tokenami. Pri každej HTTP požiadavke, pre ktorú je potrebné overiť používateľa a prípadne jeho rolu je potrebné odoslať tento nami vydaný JWS token v `Authorization` hlavičke požiadavky.

Kapitola 4
Analýza

Správna a skorá identifikácia funkčných a nefunkčných požiadavkov je kľúčová a často determinuje úspech a efektivitu vyvíjanej softvérovej aplikácie. Pri nesprávne určených požiadavkách sa môže stať, že dodaný softvér nespĺňa čo i len základnú potrebu zainteresovaných osôb (používatelia, operátori, programátori, softvéroví architekti, investori a podobne). Zber a analýza požiadavkov by mala byť prvá fáza každého softvérového vývoja. Pomocou zozbieraných požiadavkov softvéroví architekti namodelujú a navrhnú systém tak, aby spĺňal všetky požiadavky, ktoré sú od aplikácie vyžadované. Funkčné a nefunkčné požiadavky poskytujú vývojárom lepšie pochopenie celkového systému a tým pádom dokážu dodať systém, respektíve funkcionalitu systému, ktorú zainteresované osoby očakávajú.

V tejto kapitole budeme nadväzovať na kapitolu o analýze existujúcich riešení pre overovanie faktov pomocou sily davu. Existujúce riešenia pre náš problém nie sú dostačujúce a ani vhodné. Našim cieľom je teda navrhnúť a vyvinúť novú webovú aplikáciu. Funkčné a nefunkčné požiadavky aplikácie boli navrhnuté po hĺbkovej analýze existujúcich riešení a odborných článkov popisujúcich možnosti využitia crowdsourcingu pre overovanie faktov na internete. Pre zaistenie objektívneho názoru na funkcionalitu systému boli do prvej testovacej fázy zahrnutí študenti žurnalistiky, ktorí vyjadrili svoj názor voči budúcnosti a možným rozšíreniam aplikácie. Okrem toho bolo nutné prečítať diskusie a fóra k existujúcim riešeniam - čo o nich šíria používatelia alebo špecialisti. Následný návrh funkčných požiadavkov sme diskutovali s produktovým vlastníkom aplikácie FactCheck. Po tomto procese sme dospeli k zoznamu funkčných a nefunkčných požiadavkov, ktoré si predstavíme v nasledujúcej kapitole. Ukážeme navrhnutý spôsob využitia crowdsourcingu a gamifikácie pre udržanie používateľov na platforme. V poslednom kroku sa zameriame na proces vyhodnocovania faktov na našej platforme.

Požiadavky sú často popisované pomocou modelov MoSCow a FURPS. Tieto klasifikačné modely budeme využívať aj v tejto práci, preto je nutné ich najskôr zadefinovať.

4.1 FURPS Model

Označenie FURPS je skratka prvých písmen anglických slov, ktoré presne popisujú tento model. Model bol prvýkrát vyvinutý spoločnosťou Hewlett-Packard v roku 1987 a následne sa stal štandardom pri klasifikácií funkčných a nefunkčných požiadavkov softvérových systémov [29]. Význam jednotlivých slov je následovný:

- **F** - Functionality (Funkcionalita) - Popisuje hlavnú funkcionalitu, požiadavky systému alebo `release` (uvoľnenie softvéru). Čo to urobí pre používateľa?
- **U** - Usability (Použiteľnosť) - Definuje dostupnosť, ako človek alebo počítač môže využiť rozhrania pre nich navrhnuté. Ako môžu iné systémy, procesy alebo človek interagovať a používať tento systém?
- **R** - Reliability (Spoľahlivosť) - Definuje spoľahlivosť systému, robustnosť a toleranciu chýb. Tieto faktory majú veľký vplyv na náklady a zdroje potrebné pre vývoj systému. Ako veľmi je kritická dostupnosť systému?

- **P** - Performance (Výkonnosť) - Definuje výkon a efektívnosť, čo zahŕňa rýchlosť, spotrebu energie, škálovateľnosť. Tieto faktory je nutné zvážiť už pri vývoji a odzrkadlia sa aj pri nasadzovaní a prevádzkových nákladoch. Ako rýchlo musí systém odpovedať na požiadavku?
- **S** - Supportability (Udržiavateľnosť) - Popisuje udržiavateľnosť systému, jeho modulárnosť a jednoduchosť integrácie s inými systémami. Popisuje tiež jednoduchosť aktualizácií, vylepšení a opráv. Aká frekvencia aktualizácií bude nutná? Aká je tolerancia v prípade, že systém bude na nejaký čas nefunkčný pri opravách alebo aktualizáciách?

V praxi sa ďalšia analýza FURPS faktorov aplikuje na škálu priorít a z tohto je možné vytvoriť počiatočný `product backlog` [29].

4.2 MoSCoW Model

Tento model vznikol ako spôsob prioritizácie požiadavkov pôvodne inej metodológie - Dynamic System Development Method. Využíva pohľad z veľmi jednoduchej perspektívy. Model MoSCoW sa stal často používaný odbornou verejnosťou pre prioritizáciu požiadavkov systémov [29]. Jeho skratka je taktiež odvodená z anglických slov, ktoré ho popisujú:

- **Mo - Must Have** (Musí mať) – Popisuje požiadavky, ktoré softvér musí spĺňať pri vydaní produktu.
- **S – Should have** (Mal by mať) – Definuje požiadavky, ktoré majú vysokú prioritu a ak je možné, mali by byť zahrnuté pri vydaní.
- **Co – Could have** (Mohol by mať) – Kategória požiadaviek, ktoré by bolo dobré mať, ak je to možné. Nemali by ale výrazne zvýšiť náklady.
- **W - Won't have (this time)** (Zatiaľ nebude mať) – Skupina požiadaviek, ktoré sú vyžadované (od zainteresovaných osôb), ale nebudú zahrnuté vo vydaní produktu [29].

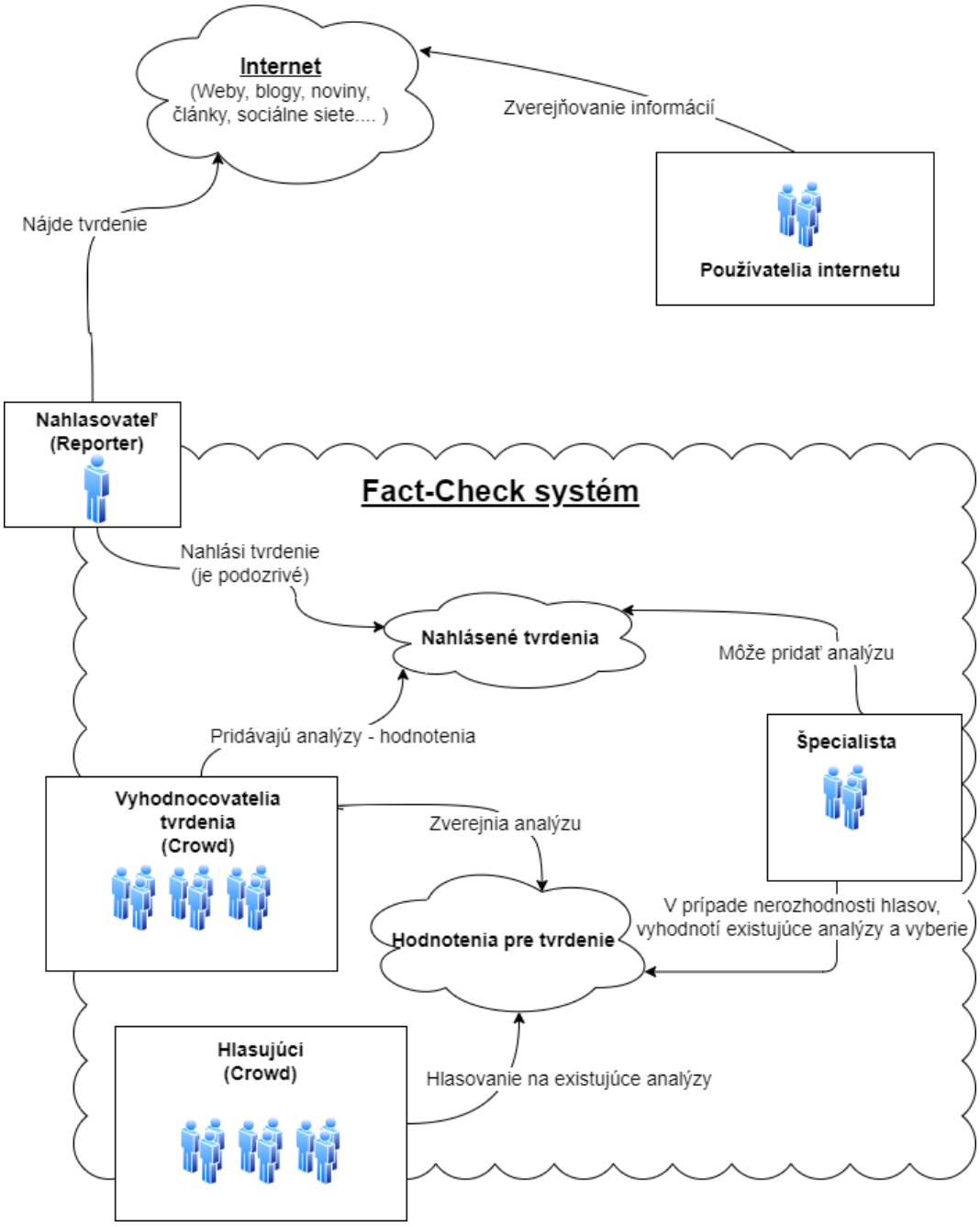
4.3 Analýza systému FactCheck

Po analýze existujúcich riešení a diskusiách s produktovým vlastníkom (rola v metodológií scrum a v našom prípade vedúcim práce Ing. Jan Drchal, Ph.D.) sme spoločne iteratívne dospeli k návrhu, ktorý vyhovuje všetkým požiadavkám na platformu. Návrh systému je inšpirovaný štúdiou o procese overovania faktov odbornými (žurnalistickými) organizáciami z roku 2019 [2]. V tejto štúdií sa opisuje systém, ktorý by dokázal uchopiť aj silu davu pre overovanie faktov na internete. Inšpiráciu sme našli aj v iných častiach existujúcich systémov, ktoré sme analyzovali v úvodných kapitolách. Našli sme silné a slabé stránky jednotlivých aplikácií a zakomponovali sme do plánovania aj požiadavky produktového vlastníka.

4.3.1 Základná funkcionalita

Cieľom našej aplikácie je využiť silu davu, kolektívnych vedomostí a schopností ľudí na internete v spojení s odborníkmi z praxe dosiahnuť čo najlepšie výsledky. Je jasné, že odbornosť ľudí, ktorí majú za sebou štúdium a/alebo roky praxe sa nedajú nahradiť anonymnými, potenciálne nevzdelanými ľudmi na internete. S neustále rastúcim množstvom nepravdivých informácií na sociálnych sieťach, kde odborné organizácie nemajú dostatočnú kapacitu pre riešenie všetkých dezinformácií je takýto systém nevyhnutné

riešenie. V spojení s automatizovanými metódami je to podľa nášho názoru optimálne riešenie. Nami navrhovaný proces overovania faktov je založený na tom, že zapojíme do vyhodnocovania faktov akýchkoľvek používateľov internetu. Ako špecialistov, tak aj neodbornú verejnosť. Systém nedokáže fungovať, ak nedokážeme zabezpečiť dostatok používateľov. Na následujúcom obrázku 4.1 môžeme vidieť biznis logiku procesu overovania faktu v našom systéme FactCheck. Anonymní používatelia internetu neustále zverejňujú textové informácie na webe. Pre našu aplikáciu je nepodstatné, či sa jedná o text na sociálnej sieti, blogu alebo web stránke.



Obrázok 4.1. Biznis logika procesu overovania faktov.

V ďalšom kroku je potrebné, aby v našom systéme pracovali používatelia, ktorí budú vkladať články a tvrdenia z internetu. Týchto používateľov nazývame oznamovatelia

(Reporter). Reporter môže pridať tvrdenie do systému FactCheck s motiváciou, že potrebuje naozaj overiť, či daný text je pravdivý alebo nie. Môže taktiež len začať diskusiu, pretože je tvrdenie nové a chce získať body za aktivitu. V každom prípade, tvrdenie bolo pridané do systému. V tomto bode sa v databáze nachádzajú tvrdenia, ktoré sú hodné overovania. Špecialisti a bežní používatelia si môžu listovať a filtrovať existujúce tvrdenia. Ak ich zaujme akékoľvek tvrdenie, majú možnosť pridať hĺbkovú analýzu s cieľom overiť pravdivosť tvrdenia. Dôležitá je evidencia faktov s odkazmi, na ktorých je hodnotenie založené. Po nájdení podrobnej evidencie si používateľ zvolí verdikt pre tvrdenie. To môže nadobúdať jedno z hodnôt: `Pravdivé`, `Čiastočne pravdivé`, `Bez výsledku`, `Neoveriteľné` alebo `Nepravdivé`. Po vložení evidencie má možnosť používateľ napísať krátky text a odôvodniť svoje rozhodnutie. Tvrdenie už môže obsahovať viaceré hodnotenia od rôznych používateľov - tie môžu byť rôznej kvality. Používatelia následne majú možnosť hlasovať o existujúcich hodnoteniach, kde majú možnosť udeliť hlas: `Súhlasím`, `Nedostatok informácií` alebo `Nesúhlasím`. Finálne hodnotenia sú zoradené podľa počtu hlasov a teda hodnotenie s najväčším počtom pozitívnych hlasov môžme klasifikovať ako najviac dôveryhodnú analýzu. Používatelia majú možnosť nahlasovať podozrivých používateľov a hodnotenia.

Najdôležitejšie časti celého systému sú tvrdenia (`claims`) a hodnotenia/analýzy tvrdení (`reviews`).

- **Tvrdenie** (`Claim`) - Tvrdenie je akékoľvek slovné spojenie, veta, úryvok alebo časť článku, o ktorej si nie ste istí či je pravdivá alebo nepravdivá. Pridanie nového tvrdenia do databázy umožní následne zahájiť proces overovania.
- **Článok** (`Article`) - Každé tvrdenie sa musí vzťahovať k článku respektíve textu z ktorého pochádza, aby sme ho vedeli spätne dohľadať a prípadne v budúcnosti dokázali vyhodnocovať webové stránky, z ktorých článok, respektíve tvrdenie pochádza. V skratke, článok poskytuje tvrdeniu kontext.
- **Hodnotenie** (`Review`) - Hodnotenie je spojené s konkrétnym výrokom. Každý používateľ môže k zvolenému tvrdeniu pridať iba jedno hodnotenie. Hodnotenie by mala byť hĺbková analýza tvrdenia a musí byť podložená faktami s odkazmi na webové zdroje. Ku hodnoteniu je možné pridať text napísaný používateľom, ktorý by nemal byť založený na názore, ale objektívne podľa nájdených faktov. Pri hodnotení si používateľ musí zvoliť jednu z piatich kategórií, do ktorej dané tvrdenie zaradí.

Nie vždy je jednoduché alebo vôbec možné priamo určiť, či dané tvrdenie je pravdivé alebo nepravdivé. Tvrdenie môže byť ovplyvnené názorom každého človeka. Človek môže byť zaujatý ideologickými, náboženskými alebo inými názormi. Môže sa jednať aj o tvrdenie, ktoré nie je možné podložiť vedecky alebo pomocou jednoduchého zberu verejných dát. Preto využijeme množinu hodnôt, ktorú využívajú aj odborné žurnalistické organizácie [2]. Niektoré majú túto množinu rozšírenejšiu, iné používajú hodnôt menej. Tvrdenie v systéme FactCheck môže nadobudnúť hodnoty:

- **Pravdivé** - Tvrdenie je pravdivé a je možné vytvoriť hodnotenie podložené faktami a odkazmi na webové zdroje.
- **Čiastočne pravdivé** - Informácia v tvrdení je čiastočne pravdivá, čo dokazujú priložené fakty. Niektoré fakty môžu byť nepresné alebo informácia je vytrhnutá z kontextu.
- **Bez výsledku** - K tvrdeniu sú iba nepresvedčivé dôkazy.
- **Neoveriteľné** - Možnosť, kedy označujeme tvrdenie ako nemožné vyhodnotiť či je pravdivé alebo nepravdivé na základov faktov a evidencie.

- **Nepravdivé** - Informácia v tvrdení je nepravdivá a hodnotenie je podložené evidenciou faktov.

Základný návrh systému je pomerne jednoduchý. Pozostáva zo štyroch hlavných entít (Používateľ, článok, tvrdenie, hodnotenie). Táto základná funkcionalita by postačovala na to, aby systém dosiahol cieľ - ukladanie článkov a tvrdení z internetu a ich následná analýza. Akú motiváciu majú ale používatelia pridávať čokoľvek do aplikácie FactCheck? Pri takomto návrhu veľmi malú. Preto je nutné pridať gamifikáciu celého systému. Ciel gamifikácie je hlavne udržať už existujúcich používateľov, aby mali opätovnú motiváciu sa vrátiť a zúčastniť sa ďalších hodnotení.

4.3.2 Gamifikácia

Odborníci a špecialisti na overovanie faktov z praxe označujú gamifikáciu v takomto systéme ako nepotrebnú alebo zbytočnú, keďže je to časť ich povinností [30]. Naša gamifikácia ale naopak mieri na používateľov, ktorí nie sú odborníci z praxe. Existuje mnoho spôsobov a návrhov herných dizajnov softvérových systémov, ako udržať používateľov v aplikácií a motivovať ich k vykonávaniu aktivít, ktoré od nich očakávame. Pri návrhu gamifikácie pre overovanie faktov na internete sa inšpirujeme štúdiou systému Calypso z roku 2022, ktorá využíva najaktuálnejšie štúdie a techniky z praxe [30]. Cieľom gamifikácie v systéme FactCheck je teda angažovanosť používateľov a budovanie dôveryhodnosti v komunite pre overovanie faktov. Gamifikácia smeruje na využitie rôznych psychologických techník, ako motivovať používateľov zväčšovať množstvo jeho vstupov vložených do systému pomocou krátkodobej a dlhodobej motivácie. Ako navrhuje systém Calypso a ako využívajú mnohé podobné systémy (napríklad platforma Stackoverflow [31]), pridáme do aplikácie systém bodovania (reputácie) a odmeňovania. Používateľ bude zbierať body vďaka svojím aktivitám v systéme, ako napríklad vkladanie nových článkov, tvrdení, hodnotení a hlasovaní. Hlasovanie na platforme FactCheck má dve rôzne úrovne:

- **Hlasovanie o tvrdení** - Hlasovanie o tvrdení má za cieľ, podobne ako platforma Stackoverflow [31], označovať tvrdenia, ktoré sú zaujímavé (pre budúcu analýzu) alebo nezaujímavé. Zaujímavé tvrdenie znamená, že ho chcem zviditeľniť - aby sa k analýze dostalo čo najviac ľudí. Naopak, hlas pre nezaujímavé tvrdenie znamená, že tvrdenie nie je hodné hĺbkovej analýzy. Tento akt vykonávajú prihlásení používatelia v systéme a je to čiastočne subjektívny názor.
- **Hlasovanie o hodnotení** - Hlasovanie o hodnotení je proces vyhodnocovania pravdivosti tvrdenia. Pre jedno zvolené tvrdenie môžu napísať rôzni používatelia svoje analýzy. Analýza musí byť objektívny proces, podložená evidenciou s odkazmi na faktografické zdroje. Hodnotenie obsahuje zvolený verdikt z vyššie spomínaných kategórií. Proces hlasovania znamená, že používatelia môžu súhlasiť alebo nesúhlasiť s danou analýzou (hodnotením), prípadne označiť, že hodnotenie nemá dostatok informácií.

4.3.3 Reputácia

Platforma musí implementovať systém reputácie, kde používateľ dostáva za každú aktivitu určité, dopredu špecifikované množstvo bodov. Používateľ by mal dostávať body práve za aktivity, ktoré od nich najviac očakávame a teda za:

- Pridanie článkov, tvrdení, hodnotení
- Hlasovanie za tvrdenia
- Hlasovanie za hodnotenia

- Používateľ sa opätovne vracia do aplikácie a je konzistentný
- Zdieľanie tvrdení alebo hodnotení na externé aplikácie (sociálne siete)
- Pozvanie nových ľudí do aplikácie

4.3.4 Filter tvrdení a trendy

Systém musí byť schopný usporiadať tvrdenia v databáze podľa rôznych parametrov. Je nevyhnutné, aby si používateľ mohol vybrať či chce vidieť najstaršie tvrdenia, ktoré ešte nie sú vyhodnotené alebo naopak si zvolí tie najnovšie. Prípadne má možnosť si zvoliť najzaujímavejšie za špecifikované obdobie. Toto obdobie bude pre používateľa voliteľné (napríklad posledných 24 hodín, posledný týždeň, mesiac a podobne). Systém FactCheck poskytne stránkovaný zoznam tvrdení podľa zvoleného filtra, pričom filtrované najzaujímavejšie tvrdenia sa budú zoraďovať podľa počtu pozitívnych hlasov.

4.3.5 Rebríčky, skóre a štatistiky

Systém umožní zobrazovať skóre každého používateľa a jeho históriu - počty entít ktoré pridal (články, tvrdenia, hodnotenia). Zobrazí tiež počty hlasov ktoré boli udelené entitám, ktoré zvolený používateľ vložil. Okrem skóre platforma dokáže usporiadať používateľov do celkového rebríčku podľa zvolenej kategórie - podľa počtu vložených hodnotení alebo podľa reputácie. Systém zobrazí históriu získavania bodov a ich množstvo získané v čase.

4.4 Funkčné požiadavky

Funkčné požiadavky definujú špecifické správanie systému a funkcionalitu, ktoré vyvíjaný softvérový systém musí spĺňať aby boli uspokojené potreby všetkých zainteresovaných osôb. Dá sa povedať, že funkčné požiadavky popisujú, čo systém má robiť, ako sa má správať, prípadne aké majú byť výsledky. Pre grafické znázornenie funkčných požiadavkov použijeme UML diagram prípadov použitia a popíšeme si jednotlivú funkcionalitu.

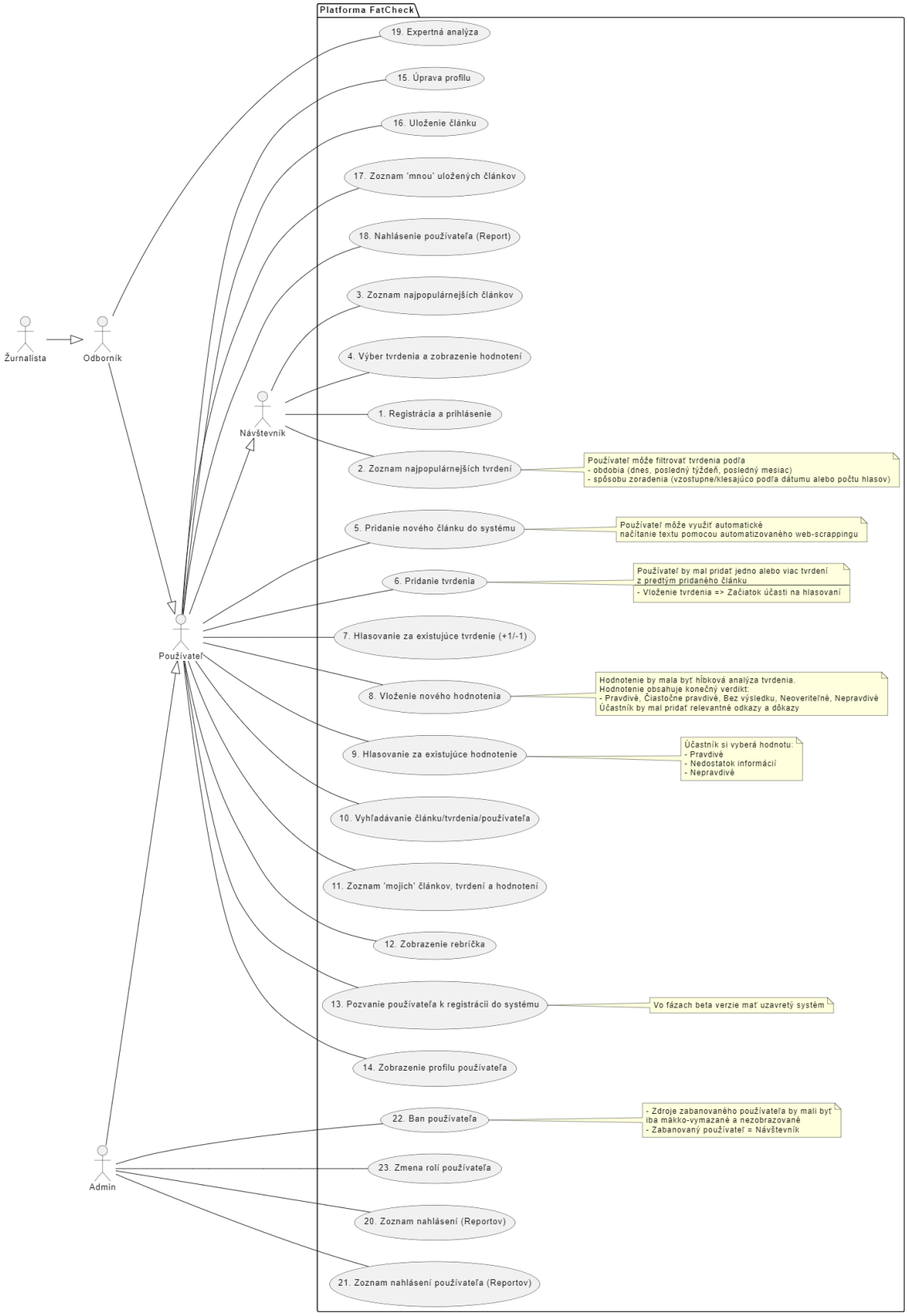
4.4.1 Autorizácia a autentifikácia

Používateľ musí byť schopný sa bezpečne zaregistrovať a opakovane prihlásiť do systému.

- Registrácia používateľa [Mo] - Systém umožní vytvorenie používateľského účtu s uložením hesla v bezpečnej - šifrovanej forme.
- Registrácia používateľa pomocou pozývacieho kódu [Mo] - V beta verzií aplikácie bude systém uzavretý pre verejnosť. Možnosť registrácie je iba pre pozvaných používateľov pomocou náhodne vygenerovaného overovacieho kódu.
- Prihlásenie používateľa [Mo] - Systém umožní prihlásenie používateľa pomocou mena a hesla a vráti klientskej strane podpísaný JWT token.
- Obnovenie autorizačného tokenu bez nutnosti zadávať znova heslo [s] - Takzvaný `Refresh token`.

4.4.2 Správa používateľov

Systém bude umožňovať evidenciu a správu používateľov v systéme a to tak, aby určitá funkcionalita bola určená len pre vybraných používateľov podľa role v systéme. Bežný používateľ má iné právomoci ako admin. Táto funkcionalita zahrňuje:



Obrázok 4.2. UML diagram prípadov použitia aplikácie FactCheck.

- Pridanie používateľa [S]

- Zoznam všetkých používateľov v systéme [Mo]
- Vyhľadanie detailných informácií o zvolenom používateľovi [Mo]
- Aktualizácia údajov o používateľovi [Mo]
- Zmazanie používateľa [Mo]
- Uzavretie/zakázanie používateľského účtu admin používateľom (ban) [C]

4.4.3 Správa nahlásení

Pre udržanie kvalitného obsahu na platforme je nutné pridať možnosť nahlasovať podozrivých používateľov a podozrivý alebo zavádzajúci obsah. V prípade nájdenia takzvaného `trolla`, je potrebné ho nahlásiť a report textovo odôvodniť. Používatelia s rolou admin si môžu zobrazovať existujúce reporty a podľa toho vykonať potrebnú akciu (ignorovať alebo zakázať používateľa).

- Systém umožní používateľom nahlásiť podozrivých používateľov a obsah [S]
- Admin má možnosť vidieť zoznam reportov [S]
- Zoznam reportov pre zvoleného používateľa [S]

4.4.4 Správa článkov

Systém umožní správu článkov, kde článok reprezentuje informácie z akéhokoľvek média vo forme textu, či už to je článok z internetu, príspevok na sociálnej sieti alebo prepísaný text z rádia. Umožní zvoliť jazyk článku, kategóriie článku, titulok, pôvodnú url adresu a časové razítko, kedy bol článok uložený do databázy. Systém umožní:

- Pridať nový článok každému prihlásenému používateľovi [Mo]
- Vypísať zoznam článkov podľa dátumu s využitím stránkovania [Mo]
- Získať detailné informácie o článku podľa špecifikovaného identifikátora článku [Mo]
- Upraviť článok s dodatočným ukladaním histórie zmien. Uskutočniť úpravu článku je povolené iba požívateľom, ktorý daný článok pridali (ďalej len `autor`) alebo používateľom s rolou admin [Mo]
- Zmazať článok (skryť z vyhľadávania, takzvané mäkké zmazanie). Zmazanie článku je umožnené iba autorom a používateľom s rolou admin [S]

4.4.5 Správa tvrdení

Po pridaní článku systém umožní prihláseným používateľom pridať tvrdenia k tomuto článku, ktoré sú priamo skopírovaný text z článku alebo parafráza. Tvrdenie k článku nemusí byť žiadne a môže ich byť neobmedzený počet. Služba poskytne rozhranie pre:

- Pridanie nového tvrdenia k zvolenému článku podľa identifikátora articleId [Mo]
- Zoznam tvrdení pre zvolený článok (podľa articleId) zoradených podľa dátumu s využitím stránkovania [Mo]
- Detailné informácie o tvrdení podľa špecifikovaného identifikátora článku a identifikátora tvrdenia [Mo]
- Upraviť tvrdenie s ukladaním histórie zmien. Upravenie tvrdenia je umožnené iba autorom a používateľom s rolou admin [Mo]
- Zmazať tvrdenie (mäkké zmazanie). Zmazanie tvrdenia je umožnené iba autorom a používateľom s rolou admin [S]

4.4.6 Správa hodnotení

Systém umožní pridať hodnotenie každému prihlásenému používateľovi ku zvolenému tvrdeniu. Hodnotenie (`Review`) je detailná analýza daného tvrdenia a je nutné, aby dané hodnotenie bolo podložené faktami a dôkazmi s url odkazmi na zdroje, na ktorých autor zakladá hodnotenie. Používateľov, ktorí sa zúčastňujú procesu hodnotenia budeme nazývať účastníci procesu overovania. Systém umožní:

- Pridať nové hodnotenie pre zvolené tvrdenie. Toto hodnotenie umožňuje uložiť všetky potrebné informácie v procese overovania faktov (text, zdroje, výsledný verdikt) [Mo]
- Zobraziť zoznam hodnotení pre zvolené tvrdenie (podľa claimId) zoradených podľa dátumu alebo podľa počtu pozitívnych hlasov s využitím stránkovania [Mo]
- Detailné informácie o hodnotení podľa špecifikovaného identifikátora hodnotenia [Mo]
- Upraviť hodnotenie s dodatočným ukladaním histórie zmien. Upravenie hodnotenia je umožnené iba autorom a používateľom s rolou admin [Mo]
- Zmazať hodnotenie (mäkké zmazanie). Zmazanie je umožnené iba autorom a používateľom s rolou admin [S]

4.4.7 Vyhľadávanie

Systém umožní:

- Systém umožní vyhľadať medzi existujúcimi článkami v databáze, podľa zadaného textového reťazca. Vyhľadanie musí brať do úvahy text článku, url zdroja a aj text z titulku [Mo].
- Systém umožní vyhľadať medzi existujúcimi tvrdeniami v databáze, podľa zadaného textového reťazca. Vyhľadanie musí brať do úvahy text tvrdenia [Mo].
- Služba poskytne možnosť vyhľadávať tvrdenia a články podľa zvolených kategórií [S].
- Systém umožní vyhľadať medzi existujúcimi používateľmi v databáze, podľa zadaného textového reťazca. Vyhľadanie musí brať do úvahy meno a priezvisko hľadaného používateľa [Mo].

4.4.8 Gamifikácia

Do aplikácie bude implementovaný systém gamifikácie, ktorého cieľom je pridať herné prvky pre udržanie používateľov na platforme FactCheck (krátkodobá motivácia). Úlohou gamifikácie je aj vytvárať dlhodobú motiváciu u používateľov tým, že ich nabáda neustále zlepšovať svoju reputáciu v komunite. systém umožní:

- **Hlasovanie** - Hlasovanie dvoch úrovní. Jedna časť hlasovania sa vzťahuje k hlasovaniu o tvrdení, aby sa dostali do popredia (medzi zaujímavejšie tvrdenia). Tvrdenie môže byť buď `zaujímavé` alebo `nezaujímavé` a hlas je možné kedykoľvek zmeniť. Druhá úroveň hlasovania je pri procese vyhodnocovania hodnotení. Pri tomto type hlasovania používateľ má možnosť výberu z troch kategórií. Používateľ s hodnotením môže `súhlasiť`, `nesúhlasiť` alebo označiť, že `nemá dostatok informácií` na vyhodnotenie.
- **Štatistiky** - Hlasovanie v aplikácií a pridávanie entít do databáz musí byť realizované tak, aby bolo možné spätne zobraziť históriu aktivít používateľa, za ktoré získaval reputáciu. Táto história bude využívaná pre tvorbu štatistík.
- **Reputácia** - Platforma bude mať presne definované počty bodov, ktoré dostáva za jednotlivé aktivity. Po každej vykonanej aktivite, služba automaticky udelí používateľovi body.

4.5 Nefunkčné požiadavky

Na rozdiel od funkčných požiadavkov, ktoré opisujú `čo má systém robiť`, nefunkčné požiadavky opisujú `ako to má systém robiť`. Sú to kvalitatívne požiadavky na softvér a vlastnosti aplikácie. Pri vývoji je dôležité jasne definovať a zvážiť nefunkčné požiadavky spolu s funkčnými.

4.5.1 N1: Architektúra [S]

Systém bude vyvíjaný v architektonickom štýle mikroslužieb. Táto architektúra nám umožní jednoduchšiu rozširovateľnosť aplikácie. Mikroslužby, ktoré budú implementovať REST API pre prácu s entitami budú ďalej využívať modulárny návrh pre oddelenie závislostí a jednoduchšiu udržiavateľnosť. Každý modul služby bude využívať a dodržiavať architektúru vrstiev, ktorá bude oddeľovať kód do kontrolných štruktúr, servisov a modelov.

4.5.2 N2: Autentifikácia a autorizácia [R/F]

Aplikácia bude zabezpečená tak, aby sa každý používateľ identifikoval pomocou svojho unikátneho emailu a hesla. Aplikácia bude rozlišovať používateľov s rôznymi rolami (bežný používateľ, expert, admin). Niektoré operácie budú povolené iba používateľom so špecifickými rolami. Implementácia použije moderné a najmä bezpečné techniky používané v praxi.

4.5.3 N3: Voľba technológií [S]

Voľba technológií pre implementáciu je ponechaná na tím vývojárov, no musí používať moderné, efektívne a bežne používané technológie.

4.5.4 N4: API rozhranie [U]

Služby backendu budú vystavovať rozhranie REST API, s ktorým budú môcť komunikovať klienti rôzneho typu.

4.5.5 N5: Databáza [S]

Databáza musí byť vysoko škáľovateľná pre podporu veľkých dát a v budúcnosti pre automatizované - strojové spracovanie.

4.5.6 N6: Dokumentácia [U]

Rozhranie REST API, ktoré budú vystavovať jednotlivé služby bude riadne dokumentované využitím OpenAPI štandardu.

4.5.7 N7: Konfigurácia [S]

Služby budú konfigurované cez konfiguračné súbory alebo procesné premenné. Konfiguračné parametre nesmú byť súčasťou kódu.

4.5.8 N8: Continuous Integration a Continuous Deployment [S]

Pre backendové služby bude nakonfigurované automatické spúšťanie testov a automatické nasadzovanie na produkciu.

4.5.9 N9: Nasadenie [S]

Súčasťou projektu bude aj nasadenie aplikácie. Aplikácia bude nasadená na OS Debian pomocou dockerizovaných služieb. Nasadený backend musí komunikovať s klientami cez šifrovaný kanál (HTTPS) a využije reverznú proxy pre presmerovanie požiadaviek na rôzne služby.

Kapitola 5
Návrh a dizajn

V tejto kapitole sa budeme venovať návrhu nového systému pre overovanie faktov na internete využitím crowdsourcingu so zapojením prvkov gamifikácie. Návrh musí vychádzať zo špecifikácie funkčných a nefunkčných požiadavkov, ktoré sme si uviedli v predchádzajúcej kapitole. Niektoré z nefunkčných požiadavkov súviseli s architektúrou. V následujúcej kapitole sa teda budeme venovať architektúre aplikácie, aby spĺňala všetky nefunkčné požiadavky. Ukážeme si tiež návrh služieb, aby spĺňali vyžadovanú funkcionalitu, respektíve funkčné požiadavky.

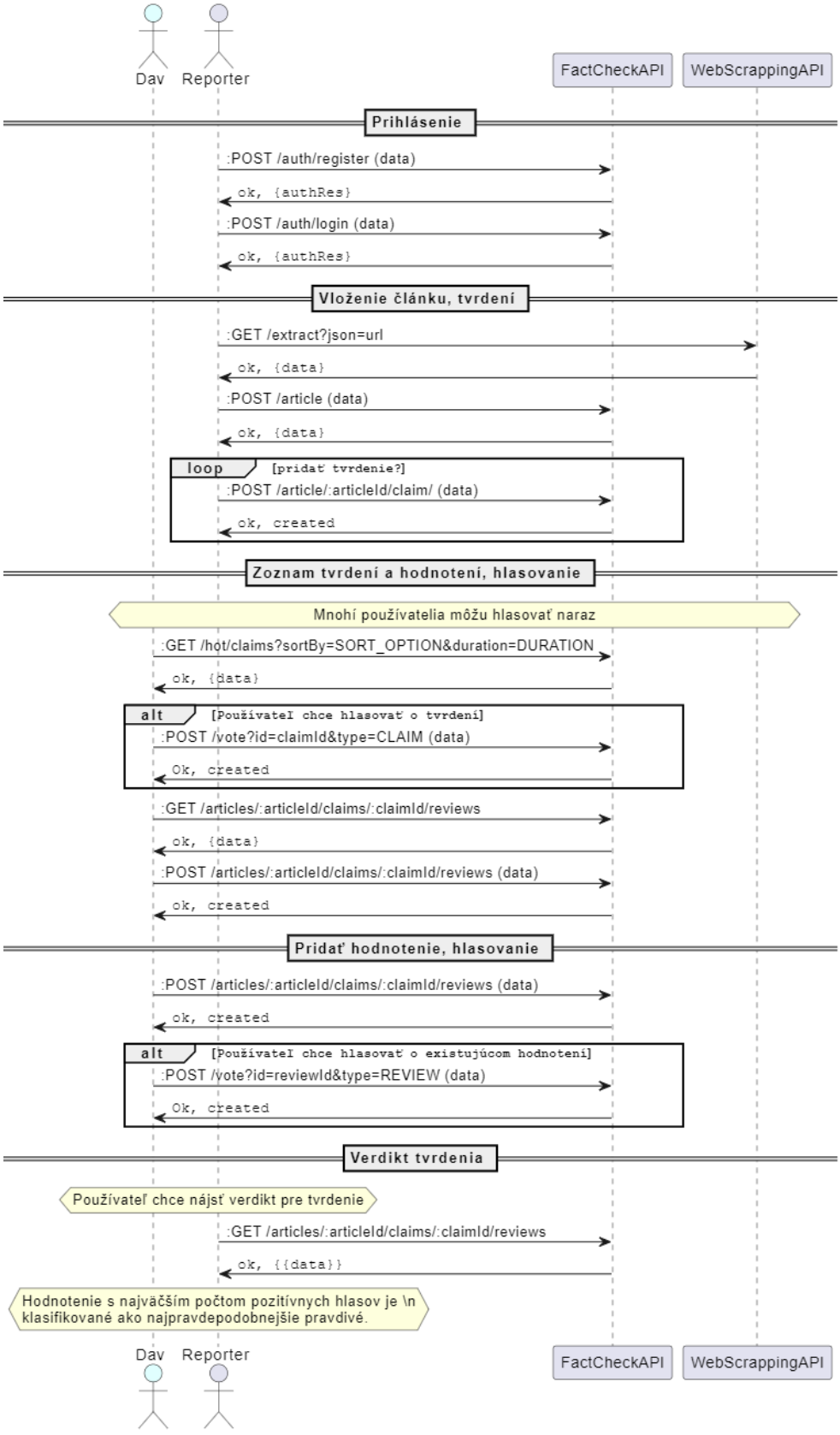
5.0.1 Postup vyhodnocovania faktov

Na obrázku 5.1 môžme vidieť UML sekvenčný diagram, na ktorom je znázornený celý proces vyhodnotenia tvrdenia na základe hlasovania komunity. Proces začína oznamovateľom (`Reporter`), ktorý do systému vloží článok a s ním spojené tvrdenie alebo viaceré tvrdenia. Tvrdenie je uložené v databáze tvrdení, pripravené na hodnotenie. Špecialisti a bežní používatelia (`Dav`) môžu vyhľadávať tvrdenia v systéme pomocou rôznych filtrov. Môžu si zvoliť obdobie, za ktoré sa zobrazia tvrdenia a taktiež podľa čoho bude zoznam zoradený. Služba FactCheck API poskytne stránkovaný zoznam tvrdení podľa zvoleného filtru. Dav si môže prehliadať tvrdenia a hlasovať za jednotlivé tvrdenia ako za `zaujímavé` alebo `nezaujímavé`. Pozitívnym hlasovaním o tvrdeniach sa stávajú zobrazované väčšiemu počtu používateľov.

V ďalšom kroku prichádza analýza tvrdení. Dav má možnosť zobraziť si zvolené tvrdenie a všetky existujúce hodnotenia k nemu. Ak sa používateľ rozhodne, môže prispieť vlastnou analýzou. Okrem toho môže hlasovať o existujúcich tvrdeniach a hodnotiť ich. Používateľ sa rozhodne, či na základe verdiktu a podloženej evidencie s hodnotením `súhlasí`, `nesúhlasí` alebo ho označí, že `nemá dostatok informácií` - v prípade, že hodnotenie nemá dostatočné faktografické dôkazy. Hodnotenie funguje demokratickým princípom. Ako potenciálne najsprávnejšie hodnotenie označujeme to, ktoré dostane najväčší počet pozitívnych hlasov. Hodnotenia s najväčším počtom hlasov sú automaticky zobrazované ako prvé - v prípade, že niekto hľadá pravdivosť daného tvrdenia.

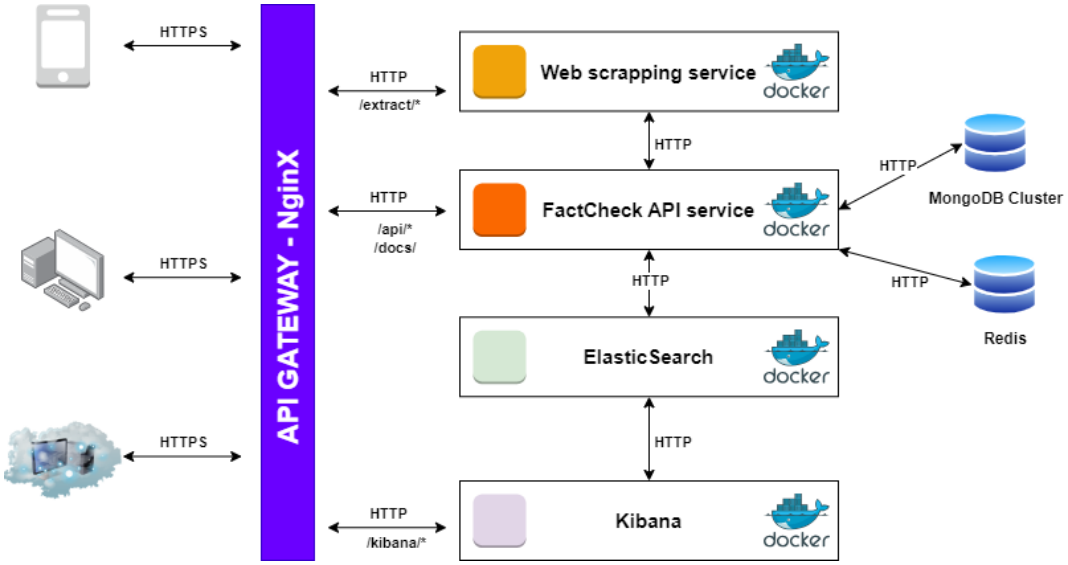
5.1 Architektúra

Architektúra systému bude navrhnutá tak, aby spĺňala všetky požiadavky ako škálovateľnosť (v prípade narastajúceho počtu používateľov), jednoduchá rozširovateľnosť a udržiavateľnosť pre budúci rozvoj aplikácie. Využijeme preto mikroslužbovú architektúru, ktorá umožňuje modularizáciu aplikácie do menších, logicky oddelených celkov - služieb. Takáto modulárna štruktúra umožňuje nezávisle vyvíjať a nasadzovať služby, čo uľahčuje škálovanie konkrétnych komponentov bez ovplyvňovania ostatných. V prípade budúceho rozvoja aplikácie, vývojár alebo vývojársky tím si môžu zvoliť akékoľvek iné technológie, než sa doteraz používali a jednoducho pokračovať vo vývoji vlastnej služby,



Obrázok 5.1. Návrh procesu vyhodnocovania faktov od registrácie používateľa, cez pridanie tvrdenia až po verdikt vytvorený komunitou.

pretože mikroslužbová architektúra podporuje technologickú heterogenitu. Mikroslužbová architektúra taktiež zjednodušujú údržbu a vývoj, keďže každá služba má dobre definovaný rozsah a hranice.



Obrázok 5.2. Architektúra aplikácie FactCheck.

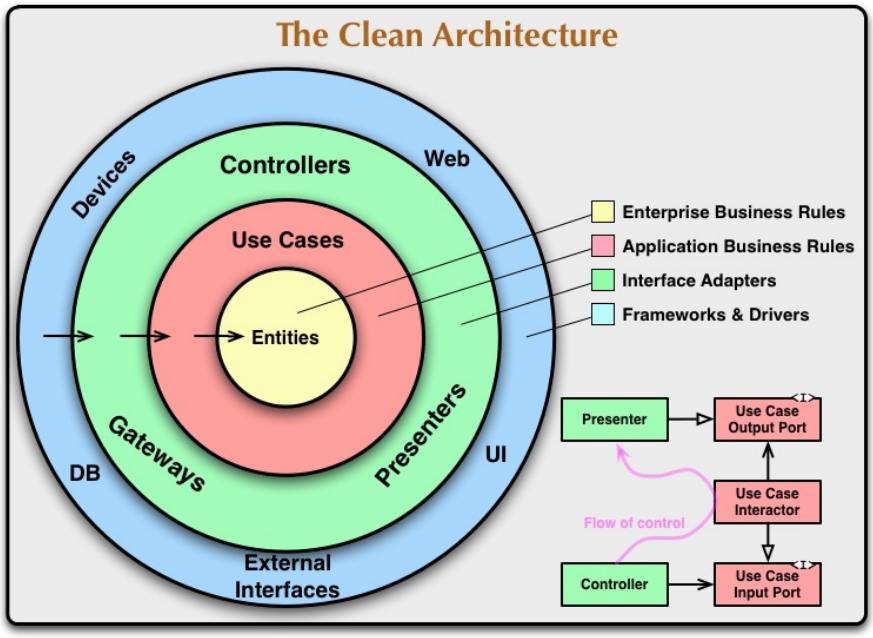
5.2 Dizajn, kontajnerizácia a mikroslužby

V základnej verzií systému budeme mať dve vlastné kontajnerizované API služby, kde každá používa vlastné technológie, podľa účelu služby. Okrem týchto dvoch sme zaviedli do systému aj služby ElasticSearch a Kibana.

- **FactCheck API servis** - Hlavná služba nášeho backendu, ktorá ponúka API rozhranie pre prácu s potrebnými entitami v databáze. Pre túto službu sme vybrali programovací jazyk JavaScript s behovým prostredím Node.js a frameworkom Nest.js.
- **Web scrapping service** - Pomocná služba systému, ktorej účel je automatické dolovanie dát z webových stránok, podľa dodanej url adresy. Táto služba využíva programovací jazyk Python a knižnicu Trafilatura pre extrakciu dát[32]. Táto služba je veľmi jednoduchá a poskytuje iba niekoľko API endpointov pre extrakciu dát z url adresy a vráti extrahovaný text.
- **ElasticSearch** - Kontajnerizovanú službu ElasticSearch [33] využijeme v základnej verzií pre ukladanie logov a ich následnú analýzu. V budúcich verziách má služba ElasticSearch aj ďalšie využitie. Službu môžeme využiť pre implementáciu full-text vyhľadávania namiesto vyhľadávania, ktoré poskytuje MongoDB. Keďže ElasticSearch je nástroj pre vyhľadávanie v texte, poskytuje omnoho výkonnejšie a presnejšie vyhľadávanie.
- **Kibana** - Mikroslužbu Kibana využijeme pre vizualizáciu dát zo služby ElasticSearch, monitorovanie aplikácie FactCheck, interakciu a prácu s dátami a logmi.

Zameriame sa na službu **Fact-Checking API** , kde bude realizovaná hlavná funkcionalita a práca s entitami. Táto služba bude navrhnutá pomocou takzvanej čistej architektúry, ktorá spája najlepšie osvedčené postupy z praxe (Hexagonálna architektúra, cibuľová architektúra a podobne) [34].

Cieľom takejto architektúry je pravidlo, že čím ďalej od stredu kruhu ideme, tým vyšší pohľad na softvér máme (viď obrázok 5.3). Dodržiavaním týchto jednoduchých



Obrázok 5.3. Čistá architektúra [34].

pravidiel (rozdelením softvéru do vrstiev) v súlade s pravidlom závislosti vytvoríme systém, ktorý je jednoducho udržiavateľný a testovateľný. Je teda nutné službu rozdeliť do celkov - kontrolerov, servisov, modelov a entít. Každý celok zabezpečuje iba tú časť, ktorú má naozaj na starosti.

5.2.1 Kontajnerizácia a docker

Pre jednoduchosť zaobalenia jednotlivých služieb budeme využívať kontajnerizáciu s Dockerom. Docker umožňuje jednoduchú kontajnerizáciu každej služby. Kontajnery poskytujú prenosnosť a konzistentnosť. Tým zaisťujú, že aplikácia môže bezproblémovo bežať v rôznych prostrediach, od vývoja až po produkciu.

Kombinácia mikroslužieb a Dockeru poskytuje silnú škálovateľnosť a efektívne využitie zdrojov. S Dockerom je horizontálne škálovanie jednoduché. Ponúka lepšiu udržiavateľnosť a flexibilitu. Každá mikroslužba môže byť aktualizovaná alebo opravovaná nezávisle bez ovplyvnenia iných častí systému. Kontajnerizácia pomocou Dockeru taktiež umožňuje jednoduché vytváranie a správu verzií.

Docker poskytuje mnohé výhody ako je modularita, škálovateľnosť, efektívne využívanie zdrojov, zjednodušené nasadenie a zlepšená udržiavateľnosť softvéru [35].

5.2.2 Návrh používateľských rolí

Po vykonaní analýzy sme dospeli k následujúcim rolám, ktoré sú v systéme nevyhnutné. Každá rola určuje práva a povolenia, ktoré daný používateľ bude mať. Rola používateľa musí byť kontrolovaná pred spustením každej funkcionality. Využijeme takzvaný RBAC (Role-Based Access Control) model. Pridaním princípov RBAC umožníme používateľom, respektíve skupinám priradiť špecifické role, ktoré rozhodujú o prístupe k jednotlivým koncovým bodom.

- **Visitor** (návštevník) - Túto rolu v systéme neukladáme, pretože ju má každý používateľ. Či už je prihlásený alebo nie.

- **User** (Používateľ) - Rolu `user` má každý používateľ po registrácií a povoľuje funkcionalitu pre nich určenú.
- **Admin** (Administrátor) - Rola `admin` v systéme rozširuje kompentencie. Umožňuje operácie s existujúcimi používateľmi, prácu s reportami a podobne.

V prvej fáze aplikácie sa rola `Expert` nebude využívať. No systém bude schopný s ňou pracovať a v budúcich prírastkoch bude pridaná funkcionalita, ktorá bude povolená práve používateľom s touto rolou.

Na základe predchádzajúcej analýzy používateľských rolí sme sa rozhodli pre jednoduchý návrh zabezpečenia prístupu pomocou zoznamu rolí. Každý používateľ bude mať definovaný konečný zoznam rolí, ktoré ho budú oprávňovať vykonávať rôzne akcie. V našom systéme budeme rozlišovať iba štyri role používateľa, kde každá oprávňuje prístup k špecifickým koncovým bodom. Jeden používateľ môže mať priradenú žiadnu alebo viacero rolí. Je to jednoduchý návrh zabezpečenia, ktorý je flexibilný, efektívny a v našom prípade dostatočný a nezavádza do systému extra komplexitu. Backend bude mať dopredu definovaný zoznam existujúcich rolí a nebudeme pre tento účel využívať samostatnú databázovú tabuľku (kolekciu). Uložíme zoznam rolí používateľa priamo do entity používateľa (viď. najlepšie praktiky návrhu schémy pre Mongo).

5.3 Databáza

Po zvážení všetkých prípadov použitia sme vybrali databázu MongoDB. V prvom rade je zvolený programovací jazyk JavaScript, ktorý priamo podporuje prácu s dokumentmi typu JSON. Je to často používaná kombinácia. Výhody flexibilných schém sme využili v prvých inkrementoch vývoja, pri ktorých sa schéma často menila. MongoDB poskytuje dobrú horizontálnu škálovateľnosť a vďaka tejto vlastnosti aj v prípade veľkého nárastu používateľov nebude mať databáza problém. Zabezpečíme tým splnenie požiadavky škálovateľnosti. Databáza MongoDB poskytuje `full text search` index, ktorý využijeme pri implementácií vyhľadávania. Mongo taktiež poskytuje aj sekundárne indexy, ktoré nám umožnia rýchlejší prístup k dátam. Tieto indexy si zvolíme sami, na základe atribútov, podľa ktorých často vyhľadávame v kolekciách.

5.4 Modelovanie dátového úložiska

Ako sme už spomínali v časti analýzy, hlavné entity nášho systému sú používatelia, články, tvrdenia a hodnotenia. Popíšeme si teraz vzťahy medzi týmito entitami a navrhneme ostatné potrebné entity, ktoré budú nutné pre ostatné funkčné požiadavky aplikácie.

5.4.1 Kolekcie v databáze

S dôrazom na efektivitu aplikácie sme navrhli následovný databázový model. Dátový model berie do úvahy silné a slabé vlastnosti MongoDB databázy a aj z tohto dôvody existuje čiastočná redundancia údajov, ktorú ale využívame v náš prospech.

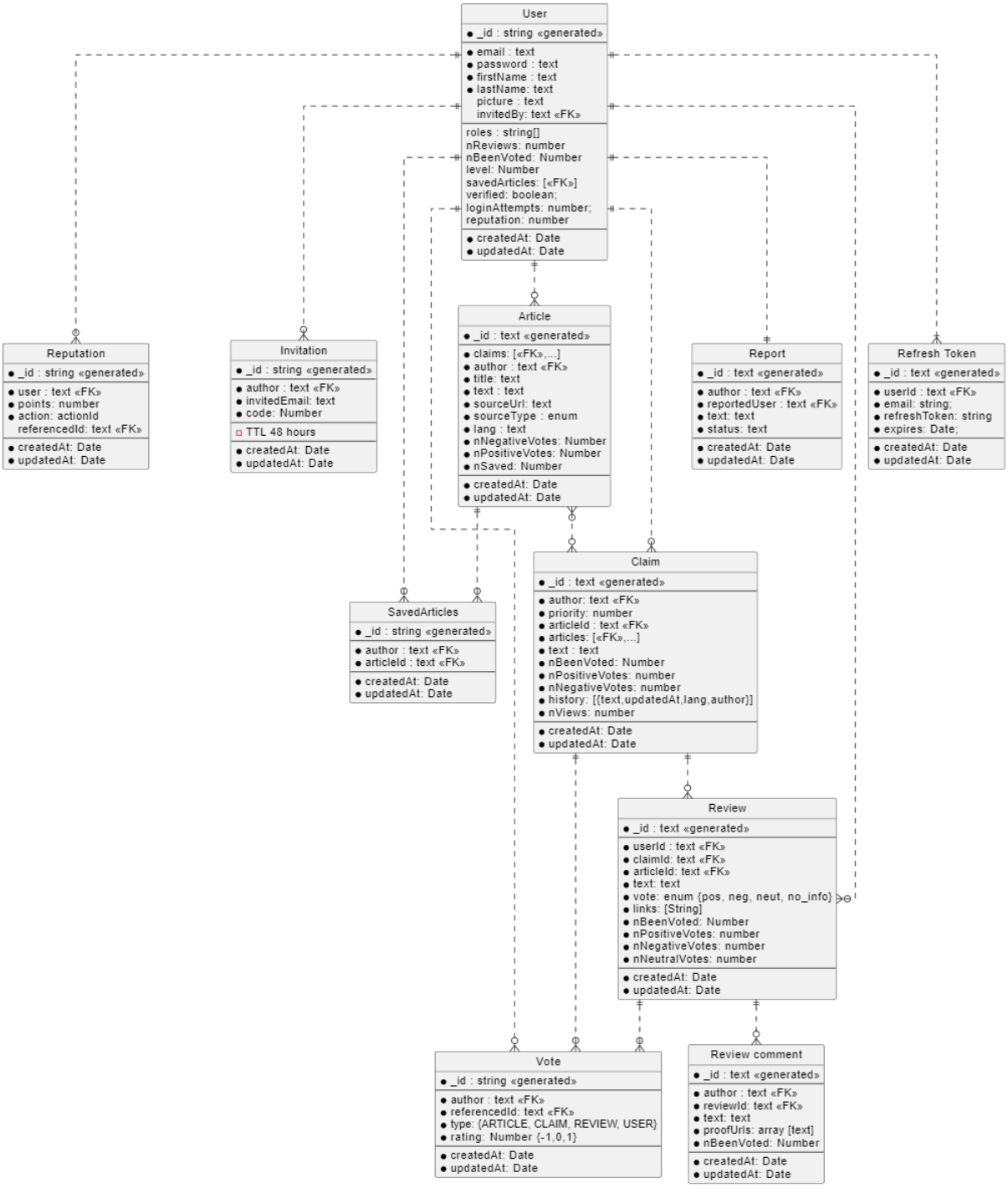
- **User** (Používateľ) - Kolekcia používateľov je základná entita systému a reprezentuje každú osobu používajúcu náš systém. Používateľa identifikuje unikátna emailová adresa. Entita používateľa taktiež má aj ďalšie atribúty ako krstné meno, priezvisko, reputáciu a role používateľa. Do entity používateľa si budeme ukladať aj redundantné údaje ako počet jeho hodnotení, počet hlasovaní, zoznam uložených článkov. Redundantné údaje využijeme pre rýchle čítanie z databázy bez potreby agregácie (článkov,

tvrdení) na úkor väčšieho počtu databázových operácií pri vytváraní (článkov, tvrdení). V systéme sa oveľa častejšie dáta čítajú, než zapisujú. Z tohto dôvodu sme sa rozhodli pre ukladanie aj redundantných údajov o počte iných dokumentov, ktoré daný používateľ vytvoril.

- **Article** (Článok) - Je entita, ktorá reprezentuje ucelený text z média (internet, noviny, prepísaný rozhovor z videa a podobne). Text môže byť napríklad skopírovaný obsah článku, blogu alebo novín. Je vždy spojený s používateľom (autorom článku, ktorý ho pridal). Môže obsahovať aj titulok, internetovú adresu pôvodného textu a typ média (z ktorého pochádza). Táto kolekcia slúži ako kontext pre konkrétne tvrdenie alebo hodnotenie. Cieľom platformy FactCheck je zber dát a kontextu. Z tohto dôvodu budeme využívať kolekciu článkov aj pre automatizované ukladanie textov z odkazov vložených pri vkladaní hodnotení a dolovanie dát z webových odkazov.
- **Claim** (Tvrdenie) - Je veta alebo časť článku, ktorú označujeme ako podozrivú a teda ako potenciálny výrok pre analýzu. Tvrdenie je vždy spojené s konkrétnym článkom, ktorý pridáva tvrdeniu kontext. Je vždy spojené aj s autorom. Používatelia majú možnosť dodatočne zmeniť text tvrdenia, no potrebujeme ukladať históriu zmien každého tvrdenia. Predpokladáme, že tvrdenia budú upravované skôr zriedkavo a preto sme sa rozhodli ukladať históriu priamo v dokumente ako pole histórie zmien, v ktorých si uložíme predchádzajúci text, dátum a identifikátor používateľa, ktorý zmenu vykonal (autor alebo admin). Pri tvrdeniach využívame podobnú redundanciu údajov ako pri kolekcií používateľ o počte hlasov pre rýchlejšie operácie čítania.
- **Review** (Hodnotenie) - Je hĺbková analýza zvoleného tvrdenia. Používateľ, ktorý pridáva hodnotenie do systému musí pridať aj url adresy dôkazov, text analýzy a vyberie verdikt, ako dané tvrdenie hodnotí. Hodnotenie je vždy spojené s jedným konkrétnym tvrdením a jedným autorom. Pridané url adresy sa automaticky pridajú do fronty úloh, kde sa automatizovaným procesom stiahnu texty zo zvolených url adries.
- **Vote** (Hlas) - Je entita, ktorá reprezentuje vyjadrenie názoru používateľa o tvrdení alebo hodnotení. Obsahuje identifikátor referencovanej entity, názov kolekcie (claims, reviews), autora, ktorý hlas pridal a hodnotenie -1, 0, 1. Vďaka explicitnej kolekcií hlasov dokážeme zaznamenávať históriu každého používateľa a následne vytvárať štatistiky.
- **SavedArticle** (Uložený článok) - Je kolekcia, kde každý dokument reprezentuje zvolený článok, ktorý si uložil zvolený používateľ.
- **Invitation** (Pozvánka) - Je kolekcia, ktorá reprezentuje aktívnu pozvánku používateľom systému pre nového používateľa, ktorý sa na základe tejto pozvánky dokáže zaregistrovať do uzavretého systému v beta verzií aplikácie. Pozvánka je aktívna iba 48 hodín a po oneskorenom pokuse o registráciu je entita zmazaná a je vyžadovaná nová pozvánka.
- **Report** (Nahlásenie) - Je kolekcia nahlásení, respektíve upozornení adminov o zvolenom používateľovi, jeho akciách ktoré sú proti pravidlám systému s možnosťou ho následne zakázať (ban). Doležité sú atribúty text, ktorý musí obsahovať dôvod a status. Status popisuje aktuálny stav nahlásenia, ktoré môže nadobúdať hodnoty - `submitted` (odoslaný), `open` (otvorený), `closed` (uzavretý). Tieto stavy môže meniť iba admin po kontrole správy a riešení problému.
- **Reputation** (Reputácia) - Je kolekcia, ktorá reprezentuje získané body každým používateľom. Popisuje za čo získal body, dátum a počet bodov. Vďaka explicitnej kolekcií reputácie dokážeme zaznamenávať históriu získavania bodov každého používateľa a

následne vytvárať štatistiky. V entite používateľa opäť ukladáme redundantnú informáciu o jeho aktuálnej reputácií pre rýchlejšie čítanie.

- **Refresh Token** (Obnovovací token) - Obnovovací token je najmä vhodný pre zvýšenie používateľskej skúsenosti, aby nebolo nutné sa opätovne prihlasovať každý 15 minút. Refresh token je token s dlhodobou životnosťou, ktorý dokáže vymeniť za krátkodobý prihlasovací JWT token. Každý používateľ môže mať aktívny iba jeden obnovovací token, respektíve jedno aktívne zariadenie.



Obrázok 5.4. Návrh dátového modelu.

Každý Dokument v každej kolekcií obsahuje jedinečný identifikátor _id, ktorý je vyžadovaný MongoDB databázou. Každý dokument automaticky pridá atribút `createdAt` - čas vytvorenia dokumentu a `updatedAt`. Pri zavedení redundantných informácií, ako napríklad - v entite tvrdenia ukladáme počet hlasov a zároveň vytvárame entitu Vote, je potrebné zabezpečiť konzistentnosť. Pre tento účel budeme využívať transakcie. V prípade, že niektorá z operácií pri ukladaní do databázy sa nepodarí, je nutné sa navrátiť do pôvodného stavu a nezmeniť hodnoty, prípadne opakovať celú transakciu.

5.4.2 Systém reputácie

Na obrázku 5.4 môžeme vidieť aktuálnu schému našej MongoDB databázy a všetky kolekcie a vzťahy medzi nimi. Kolekcia reputácií ukladá akcie, ktoré používateľ vykonal. Uloží si hodnoty: Kto akciu vykonal, kedy, koľko bodov za ňu dostal, akú akciu vykonal a taktiež identifikátor referencovaného objektu. Uložený JSON objekt v MongoDB databáze bude vyzerať následovne:

```
{
    _id: ObjectId(”64c0e3ef7e99b005a174eec8”),
    user: ObjectId(”64c0e3e57e99b005a174eec0”),
    points: 8,
    referencedId: ObjectId(”64c0e3ef7e99b005a174eec7”),
    action: ’CREATE_ARTICLE’,
    createdAt: ISODate(”2023-07-26T09:14:23.053Z”),
    updatedAt: ISODate(”2023-07-26T09:14:23.053Z”),
  },
```

Ak používateľ bude v čase vykonávať rôzne aktivity, na základe kolekciu `Reputations` dokážeme analyzovať históriu. Môžeme si vyhľadať históriu používateľských aktivít podla akcie (Vytvorenie článku, pridanie tvrdenia a podobne) a následne jednoducho generovať dáta pre tvorbu grafov.

V následujúcej tabuľke môžeme vidieť množstvo reputácie, ktoré používatelia získavajú za jednotlivé aktivity vykonávané na platforme FactCheck.

Aktivita	počet bodov
Pridanie článku	8
Pridanie tvrdenia	20
Pridanie hodnotenia	30
Pozvánka používateľa	30
Zdieľanie FactChecku	15
Hlasovanie (claim)	3
Hlasovanie (Review)	5

Tabuľka 5.1. Reputácia za aktivity v systéme.

5.4.3 Indexy

Každá kolekcia v databáze má automaticky nastavený index na povinnom stĺpci s unikátnym identifikátorom _id. Okrem toho, podľa prístupov k dátam v aplikácií sme navrhli následovné indexy:

Každý riadok tabuľky predstavuje jeden index pre zvolenú kolekciu. Index je vždy definovaný nad konkrétnym atribútom. Pri každom atribúte je uvedený aj typ použitého indexu.

Kolekcia	Atribút	Index
Users	firstName	Text Index
Users	lastName	Text Index
Users	email	Text Index, unique
Articles	title	Text Index
Articles	text	Text Index
Articles	categories	Single Field Index
Articles	sourceUrl	unique
Claims	text	Text Index
Claims	categories	Single Field Index
Reviews	author	Single Field Index
Reviews	claim	Single Field Index
Reviews	text	Text Index
Reputations	User	Single Field Index
Reputations	referencedId	Single Field Index
Invitations	invitedEmail	Single Field Index, unique
Reports	reason	Text Index
Reports	details	Text Index
Reports	status	Single Field Index
Votes	type	Single Field Index

Tabuľka 5.2. Špecifikácia indexov použitých pri návrhu databázy.

5.5 Zabezpečenie aplikácie

JWT tokeny môžu byť použité v spojení s RBAC na vytvorenie bezpečného a škálovateľného autorizačného systému. Implementáciou konceptov RBAC možno používateľom alebo skupinám udeliť špecifické zodpovednosti a povolenia. Server potom môže potvrdiť autorizáciu na základe rolí pre každý chránený zdroj alebo koncový bod pomocou tokenu JWT, ktorý bude obsahovať informácie o rolách používateľa. Táto kombinácia zvyšuje bezpečnosť a zachováva granulárnu kontrolu prístupu tým, že zaisťuje, že k schváleným funkciám systému majú prístup iba oprávnení používatelia so správnymi zodpovednosťami. Podpísaný JWT token bude teda obsahovať zoznam rolí, ktoré používateľ má a podľa nich sa server rozhodne, či požiadavku vykoná alebo zamietne. Telo podpísaného JWT tokenu bude následovné:

```
{
  ”sub”: ”64c0e3e57e99b005a174eec0”,
  ”roles”: [”user”, ”admin”],
  ”iat”: 1690366792,
  ”exp”: 1690546792
}
```

5.6 Špecifikácia formátu odpovedí

Pre komunikáciu medzi klientskými aplikáciami a serverom bude využité rozhranie REST API, ktoré umožní komunikáciu s klientmi rôzneho typu. Komunikácia bude používať formát JSON. Stavové kódy vrátené zo služieb REST API určujú, či sa požiadavka podarila splniť, prípadne ak nastala chyba - špecifikujú dôvod. Tabuľka 5.3

Stavový kód	Popis
1XX	Informačná správa
2XX	Úspech pri spracovaní požiadavky
3XX	Presmerovanie klienta
4XX	Chyba na strane klienta
5XX	Chyba na strane servera

Tabuľka 5.3. Stavové kódy protokolu HTTP.

obsahuje základné rozdelenie stavových kódov, ktoré bude dodržiavať aj backend aplikácie FactCheck.

Ako sme spomenuli vyššie, telo HTTP odpovedí má formát JSON. Telá odpovedí serveru rozdelíme na dve skupiny: `validné odpovede` a `chybové odpovede`, kde každá ma trochu inú štruktúru. Formát validnej odpovede vráti JSON objekt alebo pole objektov priamo v tele odpovede. Bude dodržiavať formát:

```
HTTP/1.1 2XX OK
Content-Type: application/json
{
<RETURNED_DATA>
}
```

Chybová hláška bude dodržiavať formát:

```
HTTP/1.1 <STATUS> ERROR
{
"statusCode": <STATUS_CODE>,
"message": <ERROR_MESSAGE>
}
```

5.7 Špecifikácia REST API

V následujúcich tabuľkách si zadefinujeme potrebné koncové body REST API, ktoré vzišli z analýzy funkčných požiadavkov. Ku každému koncovému bodu zadefinujeme URI, HTTP metódu, popis koncového bodu a pridáme prístupové parametre. Parameter `User` a `Admin` admin označujú používateľské role. Povolenie `Owner|Admin` označuje také povolenie, že prístup k danému koncovému bodu má iba autor daného zdroja alebo admin. Prázdna bunka znamená, že používateľ nepotrebuje žiadnu priradenú rolu pre prístup k danej metóde - je otvorená pre verejnosť (`Návštevníkov`).

URI	Metóda	Popis	Prístup
Prefix: **`/auth`**			
/register	POST	Registrácia nového používateľa	
/login	POST	Prihlásenie existujúceho používateľa	
/refresh-token	POST	Obnovenie tokenu	
/profile	GET	Detail prihláseného používateľa	user

Tabuľka 5.4. Špecifikácia API - Autorizácia a autentifikácia.

Pre skrátenie dĺžok URI v tabuľkách sme zaviedli konštantu **prefix**. Tento prefix sa bude pridávať ku všetkým koncovým bodom z tabuľky. To znamená, že napríklad koncový bod pre registráciu používateľa je **/auth/register** .

Ako metodiku pre tvorbu špecifikácie REST API sme zvolili prístup mapovania logického dátového modelu (LDM) na koncové body. Tento prístup poskytuje dobre štruktúrovaný, konzistentný a ľahko pochopiteľný prístup k tvorbe API. Ako príklad si môžme vziať entitu **users** a koncové body v tabuľke 5.5. Koncové body boli navrhnuté tak, aby klientom ponúkali jednoduchú prácu s entitami (vytvorenie, zmena, zmazanie, zoznam). Tejto metodológie sme sa držali pri návrhu všetkých ostatných koncových bodov pre prácu s entitami (articles, claims, reviews a ďalších). Okrem dátovo orientovaných API koncových bodov sme navrhli aj niekoľko koncových bodov, ktoré popisujú určité akcie v systéme, ako napríklad registrácia, prihlásenie, vyhľadávanie. Tieto akciou orientované API koncové body sa návrhom trochu líšia od dátovo orientovaného návrhu. Použitím LDM ako referenčného bodu však zaisťujeme, že aj tieto koncové body riadené akciami budú mať konzistentný vzťah so základnými entitami dátového modelu, s ktorými interagujú.

URI	Metóda	Popis	Prístup
Prefix: **/users**			
/	GET	Zoznam všetkých používateľov	User
/	POST	Vytvorenie nového používateľa	Admin
/:userId	GET	Detailné informácie	User
/:userId	PATCH	Úprava používateľa	Owner|Admin
/:userId	DELETE	Zmazanie používateľa	Owner|Admin
/:userId/articles	GET	Zoznam článkov používateľa	User
/:userId/claims	GET	Zoznam tvrdení používateľa	User
/:userId/reviews	GET	Zoznam hodnotení používateľa	User
/:userId/reports	GET	Zoznam reportov o používateľovi	Admin

Tabuľka 5.5. Špecifikácia API - users.

URI	Metóda	Popis	Prístup
Prefix: **/articles**			
/	GET	Zoznam všetkých článkov	
/	POST	Vytvorenie nového článku	user
/:articleId	GET	Detailné informácie o článku	
/:articleId	PATCH	Úprava zvoleného článku	Owner|Admin
/:articleId	DELETE	Zmazanie článku	Owner|Admin

Tabuľka 5.6. Špecifikácia API - správa článkov (articles).

URI	Metóda	Popis	Prístup
/vote	POST	Hlasovanie o zvolenom tvrdení	user

Tabuľka 5.9. Špecifikácia API - hlasovanie (votes).

URI	Metóda	Popis	Prístup
Prefix: **/articles/:articleId/claims**			
/	GET	Zoznam všetkých tvrdení	
/	POST	Vytvorenie nového tvrdenia	user
/:claimId	GET	Detailné informácie o tvrdení	
/:claimId	PATCH	Úprava zvoleného tvrdenia	Owner|Admin
/:claimId	DELETE	Zmazanie tvrdenia	Owner|Admin

Tabuľka 5.7. Špecifikácia API - správa tvrdení (claims).

URI	Metóda	Popis	Prístup
Prefix: **/articles/:articleId/claims/:claimId/reviews**			
/	GET	Zoznam všetkých tvrdení	
/	POST	Vytvorenie nového tvrdenia	user
/:reviewId	GET	Detailné informácie o tvrdení	
/:reviewId	PATCH	Úprava zvoleného tvrdenia	Owner|Admin
/:reviewId	DELETE	Zmazanie tvrdenia	Owner|Admin

Tabuľka 5.8. Špecifikácia API - správa hodnotení (reviews).

URI	Metóda	Popis	Prístup
Prefix: **/hot**			
/articles	GET	Zoznam trendy článkov (podĺa uložení)	user
/claims	GET	Zoznam trendy tvrdení (podĺa hlasov)	user
/reviews	GET	Zoznam trendy analýz (podĺa hlasov)	user

Tabuľka 5.10. Špecifikácia API - správa trendov (articles, claims, reviews).

URI	Metóda	Popis	Prístup
Prefix: **/search**			
/articles	GET	Zoznam článkov podľa hladaného výrazu	user
/claims	GET	Zoznam tvrdení podľa hladaného výrazu	user
/users	GET	Zoznam používateľov podľa hladaného výrazu	user

Tabuľka 5.11. Špecifikácia API - vyhľadávanie.

URI	Metóda	Popis	Prístup
/stats	GET	Detailné štatistiky o používatelovi	user
/stats/leaderboard	GET	Rebríček najaktívnejších používateľov	user

Tabuľka 5.12. Špecifikácia API - Štatistiky.

URI	Metóda	Popis	Prístup
/reports	POST	Nahlásenie používateľa	user
/reports	GET	Zoznam reportov	admin
/reports/:reportId	GET	Získanie reportu	admin
/reports/:reportId	PATCH	Zmena reportu (vyriešenie)	admin

Tabuľka 5.13. Špecifikácia API - správa reportov.

Kapitola 6
Implementácia

V tejto kapitole si popíšeme postupy na praktickú realizáciu dizajnu a špecifikácií na plne funkčný systém. Budeme sa zaoberať výberom nástrojov a technológií používaných v procese vývoja. Načrtneme osvedčené postupy z praxe používané počas fázy implementácie, architektonické vzory a štandardy kódovania. Kapitola tiež rozoberá ťažkosti, ktoré sa vyskytli počas vývoja a vysvetľuje metódy ich riešenia.

6.1 Použité technológie

Mikroslužba FactCheckAPI pracujúca s entitami je implementovaná v jazyku JavaScript, ktorý sa spúšta v prostredí NodeJS. Pre zjednodušenie implementácie rozhrania REST API v NodeJS je použitý framework NestJS.

6.1.1 NodeJS

NodeJS je softvérová platforma, ktorá umožňuje na nej vytvárať webové servery a webové aplikácie. Sama o sebe nie je web server ani programovací jazyk. Obsahuje zabudovanú HTTP serverovú knižnicu, čo znamená, že nepotrebujeme žiadny ďalší webový server (ako Apache alebo IIS) [36].

Tradičný problém s počítačmi v minulosti bol, že CPU dokáže spracovávať iba jednu požiadavku v jednom momente. Tento problém bol vyriešený už dávno viac-vláknovým programovaním, ktoré nám umožňuje použiť viaceré vlákna na jednom CPU. Striedanie medzi vláknami je rýchle, no má veľký overhead. NodeJS tento problém rieši tak, že používa iba jedno vlákno, riadené udalosťami (anglicky event-driven).

Namiesto vytvárania nového vlákna pre každú požiadavku, NodeJS používa iba jedno zdieľané vlákno pre všetky požiadavky. Ak napríklad vytvárame požiadavku na databázu, namiesto blokovania vlákna používame iba takzvané call-back funkcie, po skončení vykonávania predchádzajúcej požiadavky. Všetko ostatné, čo vykonáva náš program (ako napríklad čakanie na dáta zo súboru alebo nová prichádzajúca HTTP požiadavka), je spracované v pozadí NodeJS paralelne [36].

6.1.2 ExpressJS

ExpressJS je framework používaný na správu používateľských relácií (anglicky sessions) s prípadnou podporou pre MongoDB. ExpressJS je využívaný najmä pre abstrakciu a zjednodušenie nastavení webového serveru reagujúceho na požiadavky, ktoré spracuje a vráti relevantné odpovede [36].

6.1.3 Typescript

Aj napriek rýchlo narastajúcemu počtu softvérových projektov, ktoré používajú JavaScript, tento programovací jazyk stále zostáva ako nevhodná voľba pre udržovanie veľkých aplikácií. TypeScript je rozšírenie JavaScriptu, ktoré má za cieľ vyriešiť jeho problémy. TypeScript je nadmnožinou samotného JavaScriptu a teda každý JavaScript program je

zároveň aj TypeScript program. TypeScript obohacuje JavaScript o modulový systém, triedy, rozhrania a staticky typovaný systém. Cieľom TypeScriptu je zjedošušiť vývoj a odhalovanie problémov v kóde čo najskôr a tým zvyšuje efektivitu programátorov. Kompilátor TypeScriptu zabezpečuje kontroly programu a vydáva JavaScript súbory [37].

6.1.4 NestJS

Ako sa píše na oficálnych stránkach dokumentácie frameworku:

Nest (NestJS) je framework na vytváranie efektívnych, škálovateľných aplikácií na strane servera NodeJS. Používa progresívny JavaScript, je zostavený a plne podporuje TypeScript (ešte stále umožňuje vývojárom kódovať v čistom JavaScripte) a kombinuje prvky OOP (Object Oriented Programming), FP (Functional Programming) a FRP (Functional Reactive Programming) [38].

NestJS je aktuálne medzi najviac používanými frameworkami pre vývoj backendových riešení a mikroslužieb pomocou NodeJS, neustále rastie na popularite a má výbornú dokumentáciu. Využíva zabudovaný modulárny systém, čo umožňuje separáciu problémov a zvyšuje testovateľnosť aplikácie.

6.1.5 Mongoose

V aplikácií používame knižnicu Mongoose, ktorá zjednodušuje prácu s MongoDB. Mongoose je takzvaná ODM knižnica (Object Document Mapper) pre NodeJS a MongoDB. V praxi to znamená, že nám umožňuje definovať naše modely dokumentov iba na jednom mieste v našom kóde [39].

6.1.6 PlantUML

PlantUML je open-source nástroj a skriptovací jazyk pre tvorbu UML diagramov. Umožňuje vývojárom generovanie diagramov priamo z kódu vo vývojovom štúdiu. Nástroj podporuje rôzne typy UML diagramov vrátane diagramov tried, aktivít, sekvencií a diagram prípadov použitia. Generovanie diagramov priamo z kódu umožňuje správu diagramov priamo v našom verzovacom nástroji git a tým motivuje vývojárov upravovať tieto súbory priamo so zmenou v kóde. Nástroj PlantUML sme využili na vizualizáciu všetkých UML diagramov.

6.1.7 NPM - správca balíčkov pre JavaScript

NPM (Node Package Manager) je predvolený správca balíkov pre Javascriptové behové prostredie NodeJS. Používa sa na pomoc s inštaláciou, aktualizáciou a celkovou správou balíkov a ich závislostí JavaScriptových knižníc. Využitie riešení pre správu balíkov ako NPM výrazne zvyšuje produktivitu vývojárov, kvalitu kódu a jednoduchosť zdieľania knižníc vytvorených komunitov alebo organizáciami.

- **package.json** - Veľmi dôležitý súbor aplikácie. Nájdeme tu definované verzie balíčkov a závilostí, ktoré je nutné vyriešiť správcom balíčkov npm pri inštalácií. Okrem toho tu nájdeme aj skripty a príkazy, ktoré nám výrazne ulahčujú či už vývoj, tak aj testovanie a nasadzovanie. Pri opakovanej činnosti a vykonávaní určitých príkazov je vždy lepšie si vytvoriť script v tomto súbore. Nájdeme tu okrem toho aj metadáta o našom projekte (názov, autor, licencia, popis, odkaz na repozitár a podobne) a konfiguračné informácie (inštalácia projektu, ako ho používať, závislosti a pod.).
- **package-lock** - Je súbor automaticky generovaný správcom balíkov NPM, pre každú operáciu, ktorá upraví priečinok **node_modules** . Poskytuje detailný popis stromu

závislostí, ktorý bol vygenerovaný. Vďaka tomuto súboru dokážeme zabezpečiť, že následujúce inštalácie dokážu vygenerovať identický strom.

6.1.8 Git a GitHub

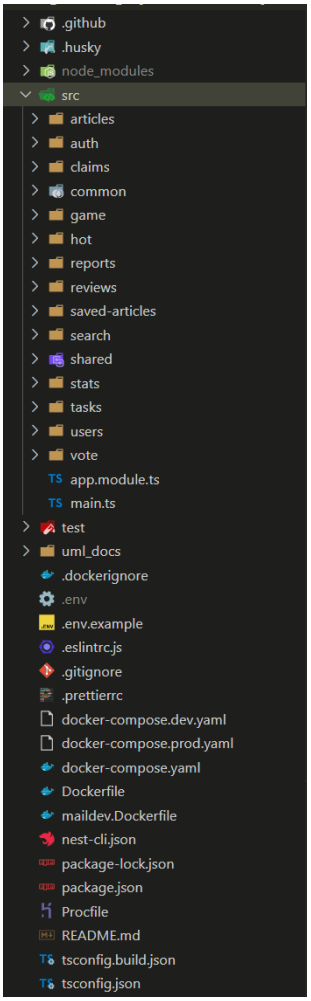
Git je distribuovaný systém pre riadenie verzií. Je to nástroj, ktorý umožňuje vývojárom sledovať zmeny v ich kóde v priebehu času, spoluprácu medzi viacerými členmi tímu a celkovo pomáha vývojárom spravovať rôzne verzie projektu. Ako webovú hostingovú službu pre zdieľanie a zverejňovanie kódu sme využili **GitHub**. Github okrem hostingu repozitárov poskytuje aj mnohé ďalšie funkcie pre správu projektov a spoluprácu. Ponúka taktiež možnosť zapojenia pipeline pre `continuous integration` a `continuous deployment` a to úplne bezplatne pre projekty s otvoreným kódom.

6.2 Zdrojový kód FactCheckAPI

6.2.1 Štruktúra kódu

Na obrázku 6.1 môžeme vidieť štruktúru kódu mikroslužby FactCheckAPI, ktorej úloha je práca so základnými entitami v databáze. Pri štruktúrovaní kódu boli využité tie najlepšie osvedčené postupy a zásady, pre udržanie prehľadnosti a kvality kódu. Napríklad dodržiavanie názvových konvencií, modulárny systém, integračné testy, unit testy a podobne.

- **.github, .husky, .eslintrc.js a .prerierrc** - Týmto súborom a priečinkom sa budeme venovať v následujúcich kapitolách, kde si vysvetlíme testovanie a kontinuálnu integráciu (CI - Continuous Integration).
- **node_modules** - Je priečinok modulov, ktorý je automaticky generovaný správcom balíčkov. Tento priečinok sa nachádza v každej NodeJS aplikácií a obsahuje kód všetkých externých balíčkov, respektíve modulov, ktoré v našom systéme používame. Správca balíčkov npm podľa definicií verzií balíčkov v súbore package.json automaticky stiahne správne a kompatibilné verzie. Typicky je tento priečinok veľký a nie je dobrý zvyk ho spravovať v našom git repozitári. Vďaka súboru package.json si každá čerstvá inštalácia stiahne aktuálne balíčky z centrálneho registra.
- **src** - Priečinok src je najdôležitejšia časť celého repozitára a je v ňom uložený celý zdrojový kód služby. Na základnej úrovni tohto priečinka je v každej NestJS aplikácií definovaný súbor *main.js* a *app.module.js*. V súbore app.module.js je definovaný koreňový (root) modul celej aplikácie. Súbor *main.js* je vstupný súbor aplikácie. Používa funkciu NestFactory na vytvorenie inštancie aplikácie Nest. V tomto súbore sa definuje konfigurácia aplikácie a načíta root modul (v našom prípade AppModule) a vytvorí sa hierarchia modulov, ktorá umožňuje frameworku Nest využívať techniku vkladania závislostí (Dependency Injection) a automatické riešenie závislostí medzi modulmi. Namiesto opakovaného písania rôznych definicií a funkcii sme vytvorili priečinok common, kde nájdeme pomocné a zdielané funkcie, ktoré sa používajú opakovane na rôznych miestach aplikácie. Obsahuje definície pre zdielané typy, middleware, pipes, validátory a podobne. V priečinku shared nájdeme zdieľané moduly v aplikácií ako konfiguračný modul, logovací modul a modul pre pripojenie k databáze MongoDB. Okrem týchto priečinkov sa tu už nachádzajú iba funkčné moduly aplikácie, kde každý rieši iba svoju uzavretú časť problému. Myšlienka týchto modulov je taká, že každý modul pracuje iba so svojou jednou entitou a dodržuje vývojový vzor oddeľovania obáv (Separation of concerns). Nie v každom module to je



Obrázok 6.1. Štruktúra kódu mikroslužby FacyCheckAPI.

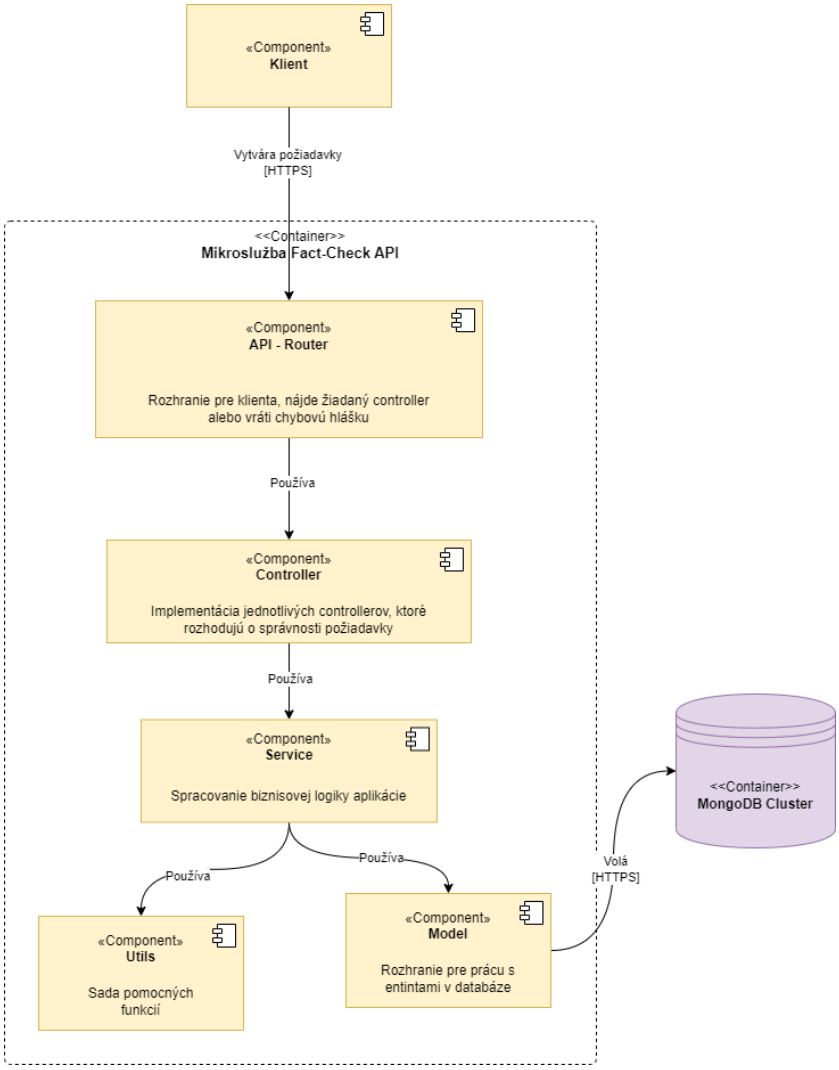
ale vždy možné, pretože niektoré už z definície musia používať entity iných modulov (ako napríklad modul pre štatistiky, hlasovania a iné).

- **test** - Tento priečinok obsahuje integračné end-to-end (e2e) testy, ktorým sa budeme venovať v kapitole o testovaní aplikácie.
- **uml_docs** - Definície všetkých UML diagramov a procesov v skriptovacom jazyku PlantUML a ich vydania vo formáte png.
- **.env** - Súbor, ktorý obsahuje premenné prostredia, ktoré ovplyvňujú beh a konfiguráciu aplikácie. Tento súbor nikdy nechceme zdielať verejne do nášho repozitáru, pretože môže obsahovať citlivé informácie. Príklad potrebných premenných vždy nájdeme v jeho verejnej kópií *.env.example*, ktorú si vždy musíme skopírovať a do prázdnych premenných doplniť hodnoty.
- **.gitignore** - Je súbor, kde sa definujú iné súbory alebo priečinky, ktoré nechceme ukladať v repozitári (ako napríklad priečinok node_modules alebo súbor .env). Tieto súbory sa teda budú ignorovať v našom verzovacom nástroji git.
- **Dockerfile** - Súbor, kde definujeme inštrukcie pre vybudovanie takzvaného Docker image (kontajnerizáciu), ktorý umožní spúštať našu Nest mikroslužbu ako kontajner. Pri procese vývoja je možné využiť tento súbor, no v našom prípade ho používame hlavne v procese nasadzovania.

- **docker-compose.yaml** - Súbor, kde definujeme inštrukcie vo formáte yaml. Týmito inštrukciami dokážeme nakonfigurovať a manažovať viaceré kontajnery, ktoré medzi sebou dokážu komunikovať. Využívame ho ako v procese vývoja (spúšťaním kvoli potrebe lokálnej verzie MongoDB) aj v procese nasadzovania, kde celý backend aplikácie sa spúšta ako docker-compose aplikácia (Mikroslužba FactCheckAPI, Web scrapping service, MongoDB, NginX, atď.).

6.2.2 Komponentový diagram a zapojenie čistej architektúry

Jednou z nefunkčných požiadavkov (zväčša na každú softvérovú aplikáciu, aj na tú našu) je udržiavateľnosť. Zabudovaný komponentový systém frameworku Nest priamo aplikuje najlepšie postupy a vytvára modulárny, dobre udržiavateľný kód oddelený do modulov. Každý z týchto modulov je ešte znova rozdelený do samostatných súborov pre oddelenie obáv a lepšiu testovateľnosť podľa teórie čistej architektúry.



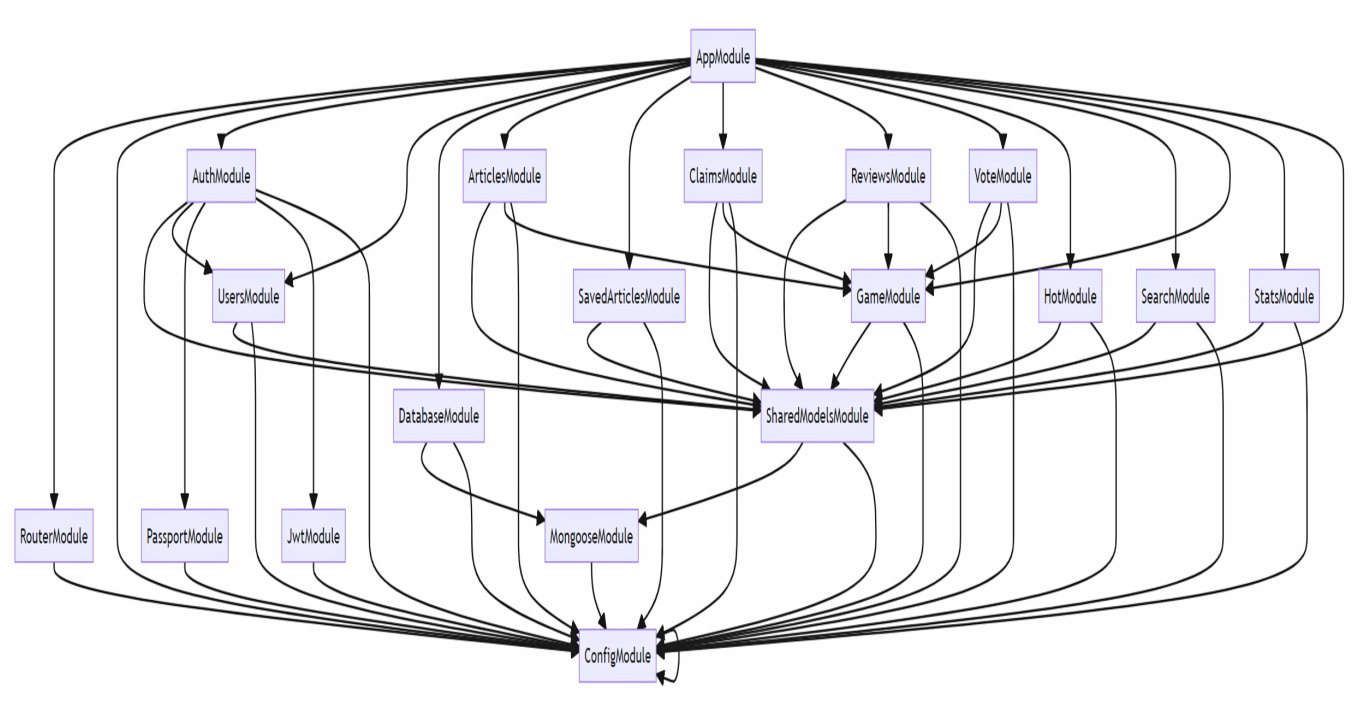
Obrázok 6.2. Diagram komponentov v jednotlivých moduloch aplikácie.

Na obrázku 6.2 môžeme vidieť UML diagram komponentov, v podstate každého modulu. Modul je rozdelený na samostatné súbory: definícia Modulu, Kontroler, Servis, Model a prípadne takzvané DTO objekty. Tieto časti frameworku Nest si rozoberieme v ďalších kapitolách.

6.2.3 Dependency Injection

Dependency Injection je technika prevrátenej kontroly (Inversion of Control - IoC), kde delegujeme vytváranie inštancií závislostí kontajneru IoC (v našom prípade behovému systému NestJs), namiesto toho, aby sme to robili manuálne my pomocou kódu. NestJs využíva Dependency Injection pomocou konštruktoru tried v čase vytvorenia inštancie. Poskytovatelia (služby, továrne a pod.) sú v predvolenom režime takzvané Singletony v rámci rozsahu daného importu. Po vytvorení inštancie poskytovateľa sa táto inštancia zdieľa v rámci modulu a vkladá sa tam, kde to je potrebné. Prípadne po importe aj v rámci iných modulov [38].

6.2.4 Zoznam modulov



Obrázok 6.3. Orientovaný graf modulov a závislostí v systéme (vrchol reprezentuje modul a hrany závislosti medzi nimi). Môžeme si všimnúť AppModule na vrchole obrázku ako koreňový modul aplikácie.

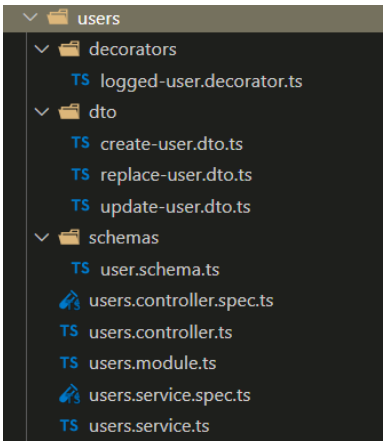
6.2.5 Základné pojmy a koncepty frameworku:

Predtým, ako si ukážeme príklad modulu používateľov si vysvetlíme základné pojmy a koncepty frameworku Nest, na ktoré sa budeme opakovane odkazovať v ďalších kapitolách.

- **Dekorátory** - ES2016 dekorátor je výraz, ktorý vracia funkciu. Ako argument môže prijať cielový objekt (target), názov a deskriptor vlastností (property descriptor). Používajú sa napísaním znaku @ a názvu dekorátora nad to, čo sa snažíme označiť týmto dekorátorom. Môžu byť definované pre triedu, metódu alebo vlastnosť [38]. Dekorátory sú známym konceptom v rôznych programovacích jazykoch, no vo svete JavaScriptu sú ešte stále relatívna novinka.
- **Kontrolery** - Kontrolery sú triedy, ktoré zabezpečujú spracovanie prichádzajúcich požiadavkov a odchádzajúcich odpovedí klientovi. Mechanizmus presmerovania frameworku Nest spravuje, ktorý kontroler bude spracovať zvolenú požiadavku.

- **Poskytovateľ** - Poskytovateľ (Provider) v NestJs je základným konceptom, ktorý umožňuje použitie, úpravu alebo manipuláciu s akoukoľvek funkciou poskytujúcou hodnotu, triedou alebo dokonca továrenskou funkciou vytvárajúcou inštancie tried. Poskytovatelia sú súčasťou vkladania závislostí (DI). Provider je označený dekorátorom @Injectable(). Mnohé funkcie NestJS, ako je middleware, filtre, pipes, guards a interceptors, sú postavené na poskytovateľoch [38].
- **Servis** - Je trieda označená dekorátorom @Injectable(). Servisy typicky zapúzdruje biznis logiku aplikácie a operácie ako požiadavky na dátabázu prípadne komplexné výpočty. Sú oddelené od kontrolerov kvôli separácií obáv (praktické využitie návrhového vzoru vrstiev) a znovu použiteľnosti funkcií inými komponentami alebo kontrolermi. Umožňuje jednoduché testovanie aplikácie, udržateľnosť a škálovateľnosť.

6.2.6 Príklad modulu používateľov



Obrázok 6.4. Súbory v priečinku modulu používateľov.

V tejto časti si ukážeme veľmi zjednodušený príklad kódu, ktorý vytvorí API koncový bod pre nájdenie používateľa podľa jeho identifikátora. Popíšeme si tu aj príklad dependency injection, ktorý nám zabezpečuje framework Nest.

V prvom kroku si vytvoríme schému kolekcie používateľov v databáze MongoDB. Následujúci kód je iba ukážkový, pre jednoduchosť pochopenia. V našom repozitári je možné nájsť aktuálnu verziu všetkých definicií, ktoré sú rozšírené o mnohé ďalšie atribúty, validácie vstupov a v tomto prípade aj metódy, ktoré sa spúšťajú pred každým uložením objektu používateľa (napríklad z dôvodu automatického hashovania hesiel) a ďalšie.

```
// users.schema.ts
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { HydratedDocument, Types } from 'mongoose';
import { Exclude, Expose, Transform } from 'class-transformer';

export type UserDocument = HydratedDocument<User>;

@Schema({ timestamps: true })
export class User {
  @Transform((params) => params.obj._id.toString())
  _id: Types.ObjectId;
```

```
  @Prop({ required: true, unique: true })
  email: string;

  @Prop({ required: true })
  @Exclude()
  password: string;

  @Prop({ type: [String], default: ['user'], required: true })
  roles: string[];
}

export const UserSchema = SchemaFactory.createForClass(User);
```

Môžeme si všimnúť elegantnú definíciu triedy používateľa využitím Typescriptu a dekorátorov. Teraz si vytvoríme vyššie spomínaný servis používateľov, kde sa bude nachdázať biznis logika aplikácie. Definujeme túto triedu použitím dekorátora @Injectable() a tým dáme vedieť frameworku Nest, že táto trieda môže byť využitá pri vyhodnocovaní závislostí.

```
// users.service.ts
import { Injectable } from '@nestjs/common';
import { User } from './schemas/user.schema';
import { Types } from 'mongoose';

@Injectable()
export class UsersService {
  constructor(@InjectModel(User.name) private userModel: Model<User>) {}

  async findOne(_id: Types.ObjectId): Promise<NullableType<User> > {
    return this.userModel.findById(_id);
  }
}
```

Už v našom servise si môžeme všimnúť využitie dependency injection a to priamo v konštruktore, definovaním triedy modelu a označením @InjectModel(User.name). V ďalšom kroku, v kontrolery požiadame framework Nest, aby automaticky vložil tohto poskytovateľa (UsersService) do triedy, kde ho vyžiadame.

```
// users.controller.ts
import { Controller, Get } from '@nestjs/common';
import { UsersService } from './cats.service';

@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get(':userId')
  @Public()
  @ApiParam({ name: 'userId', type: String })
  @HttpCode(HttpStatus.OK)
  findOne(
    @Param('userId', new ParseObjectIdPipe()) _id: Types.ObjectId,
```

```
  ): Promise<NullableType<User> > {
    return this.usersService.findOne(_id);
  }
}
```

Môžeme vidiet, že pri vytvorení kontroleru UsersController sme pridali dekorátor @Controller('users'), ktorý nám definuje cestu ku koncovým bodom v tejto triede a to pridaním segmentu `users` do url. V tomto prípade môžme zavolať koncový bod (pri lokálnom vývoji) ako http://localhost:3000/users/:userId.

A v poslednom kroku je už iba nutné zaregistrovať poskytovateľov v definícií modulu tak, aby ich IoC frameworku Nest dokázal nájsť:

```
// users.module.ts
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';
import { SharedModelsModule } from '../shared/shared-models.module';

@Module({
  imports: [SharedModelsModule],
  controllers: [UsersController],
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}
```

V prípade, že potrebuje poskytovateľov z iných modulov, ako napríklad v tomto prípade databázové modely, je nutné importovať moduly, v ktorých je daný poskytovateľ definovaný a exportovaný.

6.2.7 Implementácia procesu hlasovania

Po spustení testov výkonnosti sme zistili, že koncový bod pre vytvorenie hlasu vykonáva viacero požiadaviek na databázu a v prípade veľkého počtu súbežných používateľov sa systém začne zahlcovať a spomaľuje aj iné časti systému. Jedno z riešení by bolo horizontálne škálovanie, na čo je aplikácia FactCheck pripravená. No náš cieľový stroj pre nasadenie aktuálne nie je až tak výkonný, takže bolo nutné nájsť iné riešenie. Navrhli sme a pridali systém fronty správ (úloh) s použitím knižnice BullMQ [40] a databázy Redis. Každé nové volanie koncového bodu **/api/vote** sa uloží do fronty a začne spracovávať, keď sa dostane na rad. Nastavili sme maximálny počet úloh, ktoré môže systém začať spracovávať za sekundu a tým vyriešili tento problém zahltenia. Klientské aplikácie dostanú rýchlu odpoveď vo formáte:

```
{
    jobId: JobId, // identifikátor úlohy
    status: 'Job has been queued.'
}
```

Systém pridá úlohu na spracovanie do fronty. Klientské aplikácie budú teda musieť využívať takzvaný optimistický prístup a predpokladať, že úloha bude úspešne dokončená a nebudú čakať. Prípadne sa môžu dopytovať na stav podľa **jobId** na koncovom bode **/api/job/:jobId**, ktorý vráti aktuálny status úlohy. V následujúcej ukážke vidíme

kontroler, ktorý obsahuje iba jeden koncový bod pre hlasovania. Umožní pridať nový hlas na referencovaný objekt, prípadne zmeniť hodnotu už existujúceho hlasu:

```
// vote.controller.ts
@ApiTags('Votes')
@Controller({ version: '1', path: 'vote'})
export class VoteController extends BaseController {
  constructor(private readonly voteService: VoteService) { super(); }

 // POST /api/vote Endpoint dokáže hlasovať nad rôznymi kolekciami naraz.
 // Je preto nutné špecifikovať typ,
 // napríklad type=CLAIM a doplniť identifikátor tvrdenia,
  @Post()
  @ApiBearerAuth()
  @HttpCode(HttpStatus.CREATED)
  @ApiQuery({ name: 'id', type: String })
  @ApiQuery({ name: 'type', enum: VoteObjectEnum, required: true})
  vote(
    @Body() createVoteDto: CreateVoteDto,
    @LoggedUser() user: User,
    @Query('id', ParseObjectIdPipe) id: Types.ObjectId,
    @Query('type', new ParseEnumPipe(VoteObjectEnum))
    type: VoteObjectEnum,
  ): Promise<VoteJobResponseType> {
    return this.voteService.create(id, type, createVoteDto, user);
  }
}
```

Kontroler iba definuje koncový bod a následne volá metódu servisu - voteService.create(), ktorý zabezpečuje spracovanie biznis logiky. V tomto prípade validáciu a zapísanie úlohy do fronty:

```
// vote.service.ts
async create(
    referencedId: Types.ObjectId,
    type: VoteObjectEnum,
    createDto: CreateVoteDto,
    user: User,
  ): Promise<VoteJobResponseType> {
  // Ako prvé skontrolujeme, či referencovaný objekt existuje
    const countRef = await this.modelMapping[type].countDocuments({
      _id: referencedId,
    });

    if (countRef === 0) throw new NotFoundException({'Invalid ref'})

    const queueData: VoteQueueType =
        { referencedId, type, createDto, user};
    // Pridáme úlohu do fronty úloh
    const job = await this.votesQueue.add(queueData);
    // Vrátime číslo úlohy a status
```

```
    return { jobId: String(job.id), status: 'Job has been queued.' };
  }
```

Po pridaní úlohy do fronty správ sa spúšťa proces spracovania úlohy, ktorý reaguje na rôzne stavy úlohy. Úloha môže byť v stave:

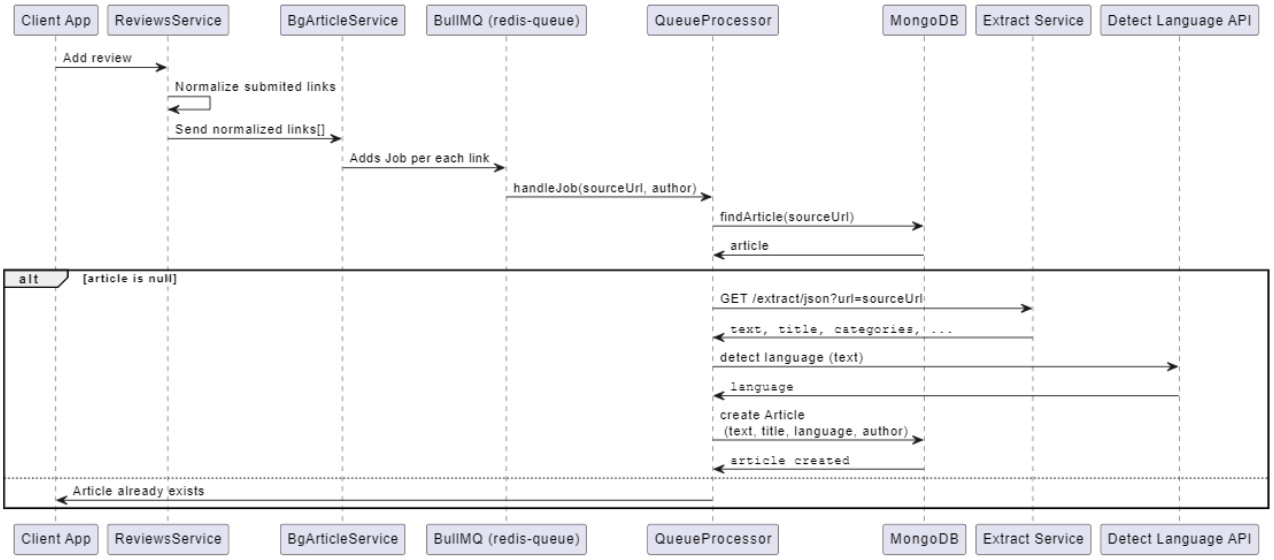
- **WAITING** - Úloha bola pridaná do fronty a čaká na spracovanie.
- **FAILED** - Pri spracovaní úlohy nastal problém a spracovanie nebolo úspešne ukončené.
- **COMPLETED** - Spracovanie úlohy bolo úspešne dokončené.

a rôznych ďalších podľa dokumentácie BullMQ [40].

```
//vote.processor.ts
@Processor(VOTES-QUEUE-NAME)
export class VoteQueueProcessor {
    private modelMapping;
    constructor(
        @InjectPinoLogger(VoteQueueProcessor.name)
        private readonly logger: PinoLogger,
        private readonly gameService: GameService,
    ) {}

    @OnQueueEvent(BullQueueEvents.WAITING)
    onJobAdded(jobId: number | string) {
        this.logger.info(`Job \${job.id} has been added to the queue.`);
    }

    @OnQueueEvent(BullQueueEvents.FAILED)
        onJobFailed(job: Job<VoteQueueType>) {
        this.logger.error(`Job \${job.id} has been FAILED!`);
    }

    @Process()
    async handleVote(job: Job<VoteQueueType>): Promise<any> {
        const { referencedId, type, loggedUser, createDto } = job.data;
        await this.unvote(referencedId, type, loggedUser);

        return this.vote(
            referencedId,
            type,
            loggedUser,
            createDto.rating
        );
    }

    @OnQueueEvent(BullQueueEvents.COMPLETED)
    onJobCompleted(job: Job) {
        this.logger.info(`Job \${job.id} has been completed!`);
    }
}
```

6.2.8 Implementácia - vytvorenie hodnotenia

Podobný problém ako pri hlasovaní nastával aj pri vytváraní nových hodnotení (reviews). Každá požiadavka musí urobiť niekoľko operácií ako: normalizácia URL, extrakcia textu zo zvolených zdrojov, rozpoznávanie jazyka textu pomocou vzdialenej API služby a nakoniec uložiť nové hodnotenie. Tento proces, ktorý je znázornený na následujúcom UML diagrame vykonáva väčší počet operácií. Z toho dôvodu sme aj pri implementácií tohto endpointu využili frontu správ.



Obrázok 6.5. UML sekvenčný diagram pre proces vytvorenia hodnotenia.

6.2.9 Implementácia lokalizácie

Jedna z požiadaviek od vývojára pre klientskú aplikáciu bola, aby backend aplikácie podporoval lokalizáciu a podľa nastaveného jazyka vrátil v prípade chybovej hlášky preložený text pre používateľa. Pre tento účel využijeme knižnicu I18n a vyžadujeme, aby klient do REST požiadavky pridal hlavičku **x-custom-lang**, ktorého hodnota bude skratka vyžadovaného jazyka. V prípade jazyka, ktorý aplikácia nepodporuje alebo vynechania hlavičky sa nastaví predvolený jazyk. V našom prípade anglický `en`. Ďalšie jazyky, ktoré v základnej verzií podporujeme sme zvolili slovenský `sk`. a český `cz`.

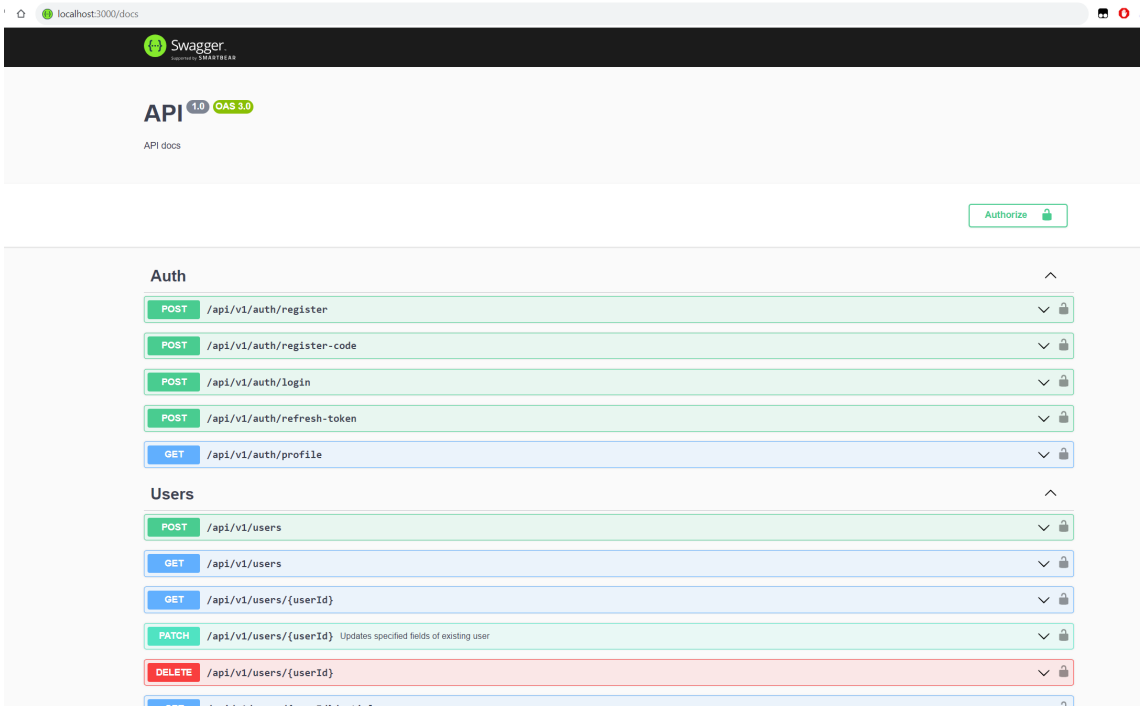
```
I18nModule.forRootAsync({
    inject: [ConfigService],
    resolvers: [new HeaderResolver(['x-custom-lang'])],
    useFactory: async (config: ConfigService) => ({
    fallbackLanguage: config.getOrThrow<string>('app.fallbackLanguage'),
    loaderOptions: {
        path: path.join(__dirname, '/common/i18n/'),
        watch: true,
        },
    }),
}),
```

Príklad použitia lokalizácie je v servisoch s využitím kontextu pomocou I18nService. V tomto kontexte nájdeme aktuálne zvolený jazyk z HTTP požiadavky.

```
    throw new NotFoundException({
      statusCode: HttpStatus.NOT_FOUND,
      message: this.i18nService.t('errors.invitation_not_found', {
        lang: I18nContext.current()?.lang,
      }),
    });
```

6.2.10 Swagger - Dokumentácia REST API

Mikroslužba FactCheckAPI vystavuje aj koncový bod **/docs**. Táto metóda zobrazí v prehliadači stránku vygenerovanú pomocou knižnice Swagger, ktorá slúži ako interaktívna (používateľsky prívetivá) a detailná dokumentácia webovej služby REST. Umožňuje vývojárom pochopiť a otestovať koncové body bez toho, aby museli manuálne vytvárať a definovať jednotlivé požiadavky. Nest poskytuje modul **@nestjs/swagger**, vďaka ktorému ho vieme jednoducho integrovať do frameworku. Tento modul následne poskytuje dekorátory pre anotáciu našich kontrolerov a metód. Swagger následne vygeneruje komplexnú špecifikáciu OpenAPI priamo zo zdrojového kódu služby. Táto používateľsky prívetivá forma dokumentácie zabezpečí hladkú a efektívnejšiu integráciu, testovanie a objavovanie existujúcich koncových bodov pre vývojárov klientských častí aplikácií.



Obrázok 6.6. Swagger - dokumentácia API.

Kapitola 7
Testovanie a zaistenie kvality

Testovanie a zaistenie kvality softvéru je v dnešnej dobe dôležitou súčasťou vývoja. Každý zložitejší softvérový produkt môže obsahovať chyby. Tieto chyby je niekedy náročné odhaliť iba manuálnym testovaním, prípadne sa môžu do systému spätne pridať až po otestovaní konkrétnej funkcionality. Správnym testovaním vieme zvýšiť kvalitu softvéru a teda dôveru všetkých strán.

7.1 Statická analýza kódu

Statická analýza kódu označuje techniky a nástroje, ktoré dokážu odhaliť chyby v kóde bez potreby ho vôbec spúšťať. Tento proces zabezpečuje zvýšenie kvality kódu dodržiavaním striktných štandardov a jednotným prístupom ku kódovaniu. Zobrazí potenciálne chyby, štylistické problémy a podozrivú syntax ešte predtým, než aplikáciu vôbec spustíme. Pre statickú analýzu Javascript a Typescript kódu využijeme nástroj **ESLint** [41], ktorý je vysoko prispôsobiteľný. My ho využijeme pre hľadanie problémových vzorov v kóde. Ako parser máme pre ESLint nastavený **@typescript-eslint/parser** a ako plugin **@typescript-eslint/eslint-plugin**. Uvedieme si niekoľko príkladov chýb, ktoré nám tento nástroj označí:

- Definovaná premenná, ktorá nie je použitá:

```
const a; // ESLint: 'a' is defined but never used
```

- Chýbajúca bodkočiarka:

```
const a // ESLint: Missing semicolon
```

- Nedosiahnuteľný kód:

```
function example() {
    return;
    const a; // ESLint: Unreachable code
}
```

Uviedli sme si iba zopár príkladov, ktoré bežne uvidíme. Sila nástroja ESLint ale ostáva v jeho flexibilite, ktorá nám umožňuje si zadefinovať aj vlastné kódovacie štýly (Napríklad známy štýl `Airbnb`).

Spolu s nástrojom ESLint využívame aj nástroj **Prettier**. Tento nástroj využívame pre automatické formátovanie kódu, aby sme si zachovali jednotný štýl naprieč celou aplikáciou. Prettier interpretuje náš kód a upravuje ho v súlade s nakonfigurovanými štýlmi, medzi ktoré patrí napríklad maximálna dĺžka riadku, dodržiavanie úvodzoviek, automatické pridávanie bodkočiarok na koniec riadkov a mnohé iné. Prettier uplatňuje konzistentný štýl programovania [42]. Inými slovami, Prettier využívame pre formátovanie kódu a ESLint pre statické hľadanie bugov.

K týmto nástrojom používame ďalší nástroj, ktorý nám zabezpečí kvalitu kódu, ukladaného v repozitári. Nástroj **Husky** pred každým commitom do git repozitára spustí takzvaný `pre-commit`, v ktorom spustí statickú analýzu kódu ESLint a v prípade že nájde problémy, zruší príkaz a je nutné kód opraviť. Je ľahké zabudnúť spúšťať statickú analýzu pred vložením kódu do gitu. Nástroj Husky tento proces automatizuje [43].

7.2 Jednotkové test

Jednotkové testy majú skontrolovať funkčnosť určitej časti kódu nezávisle na okolitom prostredí. Zvyčajne sa takto testuje iba jedna funkcia jednej trieda, oddelená od všetkých závislostí. Tento typ testovania umožňuje rýchlu kontrolu malých častí softvéru. Pre implementáciu jednotkových testov sme zvolili knižnicu Jest, ktorá je aj odporúčaná frameworkom NestJS v oficiálnej dokumentácií [44]. Pomocou jednotkových testov sme otestovali iba zvolené časti aplikácie - servisy, ktoré obsahujú rozsiahlejšiu biznis logiku (napríklad servis pre autorizáciu a autentifikáciu, servis hlasovania a iné).

7.3 Integračné testy

Integračné testy (tiež nazývané e2e testy) sú úroveň testovania, kde sa menšie jednotky spoja a testujú ako celok. Takáto testovacia metodika sa využíva najmä pri vrstvovej architektúre. Účelom tohto typu testovania je odhalenie chýb medzi integrovanými vrstvami. V našom prípade budeme testovať správnosť odpovedí pre vystavené koncové body REST API. Overíme všetky metódy a množiny hodnôt, ktoré vstupy môžu nadobúdať. Overíme, či služba vráti odpoveď takú, akú od nej očakávame. V prípade zlého vstupu budeme očakávať chybovú odpoveď, v prípade správnej požiadavky očakávame správnu odpoveď s vrátenými dátami. Pre implementáciu integračných testov využijeme opäť odporúčanú knižnicu Jest [44]. Týmto spôsobom testovania overíme správnosť špecifikácie koncových bodov a taktiež správnosť vykonávania servisov a biznis logiky. Uvedieme si príklad integračných testov pre koncové body na vytvorenie nového článku. Po odoslaní správnych údajov, očakávame odpoveď s vytvoreným objektom:

```
it('should create a new article', () => {
  return request(httpServer)
    .post('/articles')
    .auth(userAccessToken, { type: 'bearer' })
    .send(article)
    .expect(HttpStatus.CREATED)
    .then((res) => {
      expect(res.body).toHaveProperty('_id');
      expect(res.body).toHaveProperty('createdAt');
      expect(res.body.text).toEqual(article.text);
      expect(res.body.sourceUrl).toEqual(article.sourceUrl);
      expect(res.body.sourceType).toEqual(article.sourceType);
      expect(res.body.lang).toEqual(article.lang);
      expect(res.body.nSaved).toEqual(0);
    });
});
```

Samozrejme je nutné testovať aj prípady, že požiadavka neobsahuje očakávané atribúty - prípadne v nesprávnej forme:

```
it('should report error when text is not provided', () => {
return request(httpServer)
  .post('/articles')
  .auth(userAccessToken, { type: 'bearer' })
  .send(_.omit(article, ['text']))
  .expect(HttpStatus.UNPROCESSABLE_ENTITY)
  .then((res) => {
    const { statusCode, errors } = res.body;
    expect(statusCode).toEqual(HttpStatus.UNPROCESSABLE_ENTITY);
    expect(errors).toHaveProperty('text');
  });
});
```

Integračné testy sme napísali pre všetky koncové body. Overili sme správne, respektíve očakávané hodnoty. Otestovali sme tiež hodnoty, ktoré aplikácia neočakáva a v takom prípade musí vrátiť chybovú hlášku (stavový kód 4xx).

- Celkový počet testovaných modulov: **8**
- Celkový počet integračných testov: **92**
- Otestovaných riadkov kódu: **85.59%**
- Otestovaných funkcií: **70.17%**

Na obrázkoch 7.1 a 7.2 a môžeme vidieť súbory jednotlivých modulov a im zodpovedajúce pokrytie integračnými testami.

File	Statements		Branches		Functions		Lines	
src	100%	30/30	100%	0/0	100%	0/0	100%	28/28
src/articles	92.3%	84/91	63.63%	7/11	93.33%	14/15	91.66%	77/84
src/articles/dto	100%	27/27	100%	0/0	100%	0/0	100%	27/27
src/articles/schemas	96.29%	26/27	100%	0/0	0%	0/1	96%	24/25
src/auth	89.69%	87/97	30%	3/10	87.5%	14/16	89.01%	81/91
src/auth/decorators	100%	10/10	100%	0/0	100%	2/2	100%	6/6
src/auth/dto	100%	25/25	100%	0/0	100%	0/0	100%	25/25
src/auth/guards	94.28%	33/35	80%	8/10	100%	7/7	93.1%	27/29
src/auth/schemas	87.5%	14/16	100%	0/0	0%	0/2	85.71%	12/14
src/auth/strategies	89.47%	17/19	0%	0/2	100%	2/2	88.23%	15/17
src/claims	90.1%	82/91	33.33%	4/12	92.85%	13/14	90.47%	76/84
src/claims/dto	100%	6/6	100%	0/0	100%	0/0	100%	6/6
src/claims/schemas	96.55%	28/29	100%	0/0	0%	0/1	96.29%	26/27
src/common	100%	5/5	75%	3/4	100%	2/2	100%	5/5
src/common/guards	71.18%	42/59	28.57%	2/7	71.42%	5/7	68%	34/50
src/common/helpers	100%	7/7	100%	0/0	100%	2/2	100%	7/7
src/common/interceptors	91.66%	11/12	50%	1/2	100%	4/4	91.66%	11/12
src/common/pipes	100%	9/9	100%	0/0	100%	1/1	100%	7/7
src/common/transformers	100%	3/3	75%	3/4	100%	1/1	100%	2/2
src/common/types	100%	7/7	100%	0/0	100%	2/2	100%	7/7
src/common/validators	71.87%	23/32	0%	0/2	50%	3/6	73.07%	19/26
src/game	94.44%	34/36	100%	1/1	80%	4/5	93.33%	28/30
src/game/enums	72%	18/25	46.15%	6/13	100%	2/2	72%	18/25
src/game/schemas	94.11%	16/17	100%	0/0	0%	0/1	93.33%	14/15
src/hot	100%	58/58	75%	6/8	100%	9/9	100%	52/52
src/hot/enums	57.89%	22/38	42.85%	6/14	100%	4/4	57.89%	22/38
src/invitations	72.22%	39/54	0%	0/4	20%	2/10	68.75%	33/48
src/invitations/dto	100%	7/7	100%	0/0	100%	0/0	100%	7/7
src/invitations/schemas	93.33%	14/15	100%	0/0	0%	0/1	92.3%	12/13

Obrázok 7.1. Pokrytie integračnými testami (1).

src/reports		67.85%	38/56	0%	0/5	20%	2/10	64%	32/50
src/reports/dto		100%	13/13	100%	0/0	100%	0/0	100%	13/13
src/reports/enums		100%	5/5	100%	2/2	100%	1/1	100%	5/5
src/reports/schemas		94.73%	18/19	100%	0/0	0%	0/1	94.11%	16/17
src/reviews		73.39%	80/109	20%	3/15	64.7%	11/17	72.27%	73/101
src/reviews/dto		100%	9/9	100%	0/0	100%	0/0	100%	9/9
src/reviews/enums		100%	6/6	100%	2/2	100%	1/1	100%	6/6
src/reviews/schemas		90.62%	29/32	100%	0/0	0%	0/3	90%	27/30
src/saved-articles		98.03%	50/51	83.33%	5/6	100%	8/8	97.77%	44/45
src/saved-articles/schemas		100%	15/15	100%	0/0	100%	1/1	100%	13/13
src/search		67.24%	39/58	0%	0/16	20%	2/10	63.46%	33/52
src/shared/config		100%	12/12	50%	11/22	100%	4/4	100%	8/8
src/shared/database		100%	16/16	50%	1/2	100%	3/3	100%	12/12
src/shared/logger		100%	11/11	50%	1/2	100%	2/2	100%	9/9
src/shared/mail		88.88%	16/18	100%	0/0	66.66%	2/3	85.71%	12/14
src/shared/shared-models		100%	15/15	100%	0/0	100%	0/0	100%	13/13
src/stats		90.9%	70/77	60%	12/20	100%	11/11	91.3%	63/69
src/tasks		100%	5/5	100%	0/0	100%	0/0	100%	3/3
src/users		63.72%	65/102	10%	2/20	37.5%	9/24	61.45%	59/96
src/users/decorators		100%	6/6	100%	1/1	100%	1/1	100%	6/6
src/users/dto		100%	28/28	100%	0/0	100%	0/0	100%	28/28
src/users/schemas		94.28%	33/35	0%	0/1	100%	2/2	96.87%	31/32
src/vote		96.66%	58/60	57.14%	8/14	100%	6/6	96.29%	52/54
src/vote/dto		100%	5/5	100%	0/0	100%	0/0	100%	5/5
src/vote/enums		100%	5/5	100%	2/2	100%	1/1	100%	5/5
src/vote/schemas		90%	18/20	100%	0/0	0%	0/2	88.88%	16/18

Obrázok 7.2. Pokrytie integračnými testami (2).

Na obrázkoch 7.1 a 7.2 môžeme vidieť pokrytie kódu integračnými testami. Jednotlivé stĺpce znamenajú následovné:

- Statements: Percento príkazov, ktoré boli vykonané. V JavaScripte sú príkazy napríklad priradenie premennej alebo volanie funkcie. Ak je percento nižšie ako 100%, znamená to, že niektoré príkazy sa nikdy nespustia.
- Branches: Percento vetiev kódu, ktoré boli spustené. Napríklad príkaz if vytvára dve vetvy kódu - jedna, pri ktorej je podmienka pravdivá a druhá, pri ktorej je nepravdivá.
- Functions: Percento funkcií, ktoré boli volané v testoch.
- Lines: Percento riadkov, ktoré boli spustené v testoch.

Vo všeobecnosti znamená, že vyššie percento pokrytia kódu je lepšie. Je treba podotknúť, že ani 100% pokrytie testov neznamená, že testy sú dokonalé. Testy okrem toho musia testovať aj rôzne hranice hodnôt, nesprávne vstupy a mnohé iné. Pokrytie 100% kódu testami teda neznamená, že kód neobsahuje žiadne chyby alebo problémy. Pokrytie testov je dobrým nástrojom na zistenie, ktoré časti kódu nie sú testované. Nie je to ale ukazovateľ kvality a účinnosti testov.

7.4 Záťažové testy

Posledná oblasť, ktorú si popíšeme v tejto kapitole sú záťažové testy. Pod týmto druhom testov si môžeme predstaviť rôzne scenáre. Častou interpretáciou záťažových testov je stresové testovanie, v ktorom vystavíme aplikáciu extrémnym záťažiam, aby sme videli ako aplikácia zvláda spracovať dáta a požiadavky v týchto podmienkach. Môžme testovať aj takzvané testy výdrže, pri ktorých aplikácia musí zvládať očakávanú zátaž, no

konzistentne po dlhší čas. Existujú aj iné formy záťažových testov, ako testovanie pri veľkom množstve dát už uložených v databáze. Pri takejto forme testovania môžme pozorovať znižovanie efektivity práce s databázou s rastúcim množstvom dát. Pri všetkých typoch testov sledujeme podobné parametre. Tie najdôležitejšie sú: počet transakcií za sekundu, počet súbežných požiadaviek, ktoré je aplikácia schopná spracovať a dĺžka odozvy. My sa v tejto kapitole zameriame na testovanie aplikácie, ktorá je pod očakávanou záťažou (súbežný počet používateľov). Uistíme sa tým, že produkt bude schopný fungovať aj pri väčšom počte používateľov.

7.4.1 Testovacia zostava

Záťažové testy budeme spúšťať v produkčnom prostredí, aby sme videli výsledky ktoré zodpovedajú realite. Produkčná aplikácia je spustená na virtuálnom stroji, ktorej parametre sú následovné:

- OS Debian 5.10.179-1
- 2 jadrový procesor Intel Core Processor (Broadwell, no TSX, IBRS), CPU @ 3.0GHz
- 6 GB Operačná pamäť
- 50 GB SSD

7.4.2 Priebeh testovania

V systéme je spustených 5 služieb, pričom všetky sú kontajnerizované pomocou nástroju Docker, ako uvádzame v kapitole Nasadenie. Pred spustením testov sme databázu naplnili 5000 používateľmi, 8000 článkami, tvrdeniami a hodnotenia - pre realistickejší pohľad na výkonnosť. Ako nástroj pre beh záťažových testov sme využili nástroj Artillery [45]. Je to nástroj s otvoreným kódom, ktorý umožňuje flexibilný a skriptom konfigurovateľný prístup k tvorbe scenárov testovania. Vytvorili sme si scenár testovania, ktorý napodobňuje používateľov klientských aplikácií - podobné realite. Každý scenár predstavuje jedného používateľa aplikácie, ktorý postupne vykonáva nasledovné akcie:

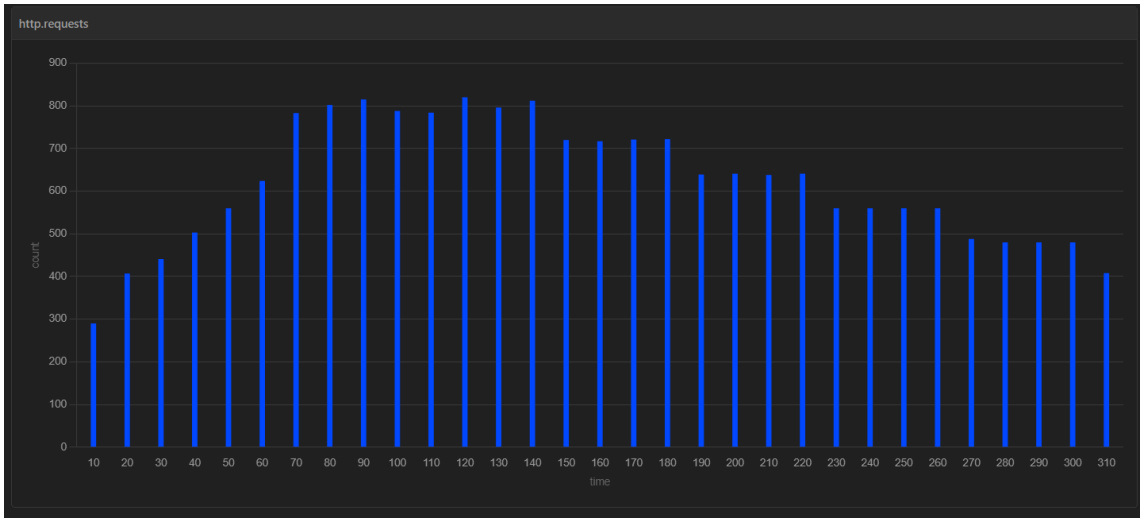
- Zaregistruje sa do systému
- Do systému pridá nový článok
- Pridá nové tvrdenie (k predtým vytvorenému článku)
- Do systému vloží hodnotenie ku tvrdeniu
- Zobrazí si 20 najnovších článkov
- Zobrazí si 20 najnovších tvrdení
- Požiada o zoznam 20 najviac pozitívne hodnotených tvrdení v posledných 7 dňoch
- Požiada o zoznam hodnotení k zvolenému tvrdeniu

Pomocou skriptovacieho jazyka `yml` sme vytvorili konfiguračný súbor pre Artillery. V tomto súbore si zadefinujeme fázy testovania:

```
load-test.yml
config:
  http:
    extendedMetrics: true
  target: https://factcheck.fel.cvut.cz
  phases:
    - duration: 60
      arrivalRate: 4
      rampTo: 8
```

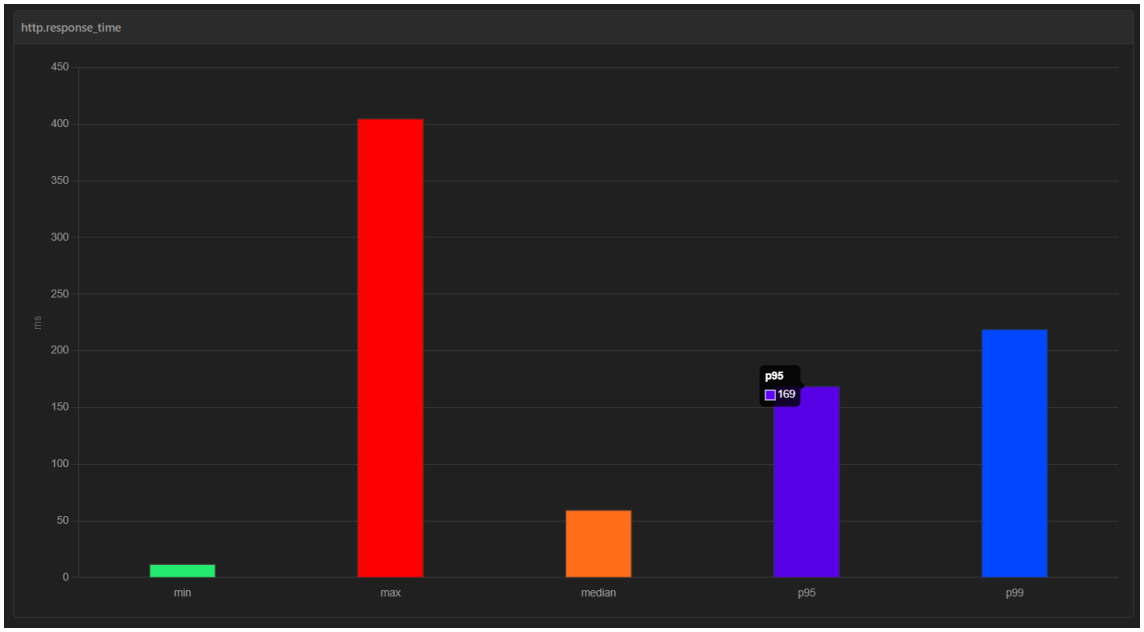
```
      name: 'Warm up'
    - duration: 200
      arrivalRate: 10
      name: 'sustained load'
    - duration: 60
      arrivalRate: 10
      rampTo: 5
      name: 'calm down'
  plugins:
    ensure: {}
    metrics-by-endpoint: {}
```

Môžeme vidieť, že testy prebiehajú v troch fázach. Prvá, v ktorej každú sekundu prídu 4 noví používatelia po dobu 60 sekúnd. Postupne sa zvyšuje počet používateľov zo 4 na 8. Druhá fáza testuje výdrž po dobu 200 sekúnd, pričom každú sekundu príde 10 nových scenárov. Testovanie ukončí tretia fáza, takzvaného schladenia. Tá trvá po dobu 60 sekúnd a 10 nových používateľov za sekundu postupne znižuje na 5 nových používateľov za sekundu. V našom scenári, 10 nových používateľov za sekundu znamená 80 nových požiadaviek za sekundu, respektíve 40 požiadaviek POST a 40 požiadaviek GET. Na obrázku 7.3 môžeme vidieť celkový počet odoslaných požiadaviek v čase. Vidíme, že počet požiadaviek postupne stúpal v zahrievacej fáze, následne sa udržoval a nakoniec začal postupne klesať.



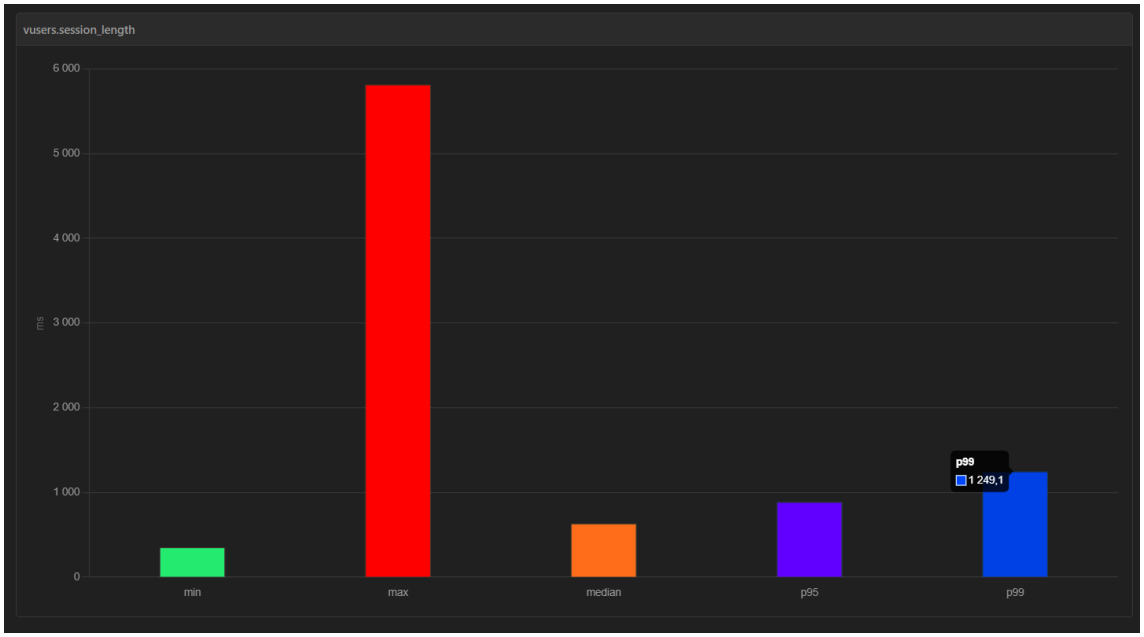
Obrázok 7.3. Artillery - Počet požiadavkov za sekundu.

Na obrázku 7.4 vidíme štatistické metriky pre čas odpovedí. Vidíme, že minimálna dĺžka odpovede bola 17ms. Môžeme si všimnúť, že v jednom prípade sa dĺžka odpovede dostala až na 405ms. Najdôležitejšie sú pre nás ale hodnoty `p95` a `p99`. `P95` nám popisuje, že 95% požiadaviek dostalo odpoveď do 169ms a `p99`, že 99% požiadaviek trvalo menej ako 223ms.



Obrázok 7.4. Artillery - čas odpovede serveru.

Na obrázku 7.5 vidíme podobné metriky, no v tomto prípade pre dobu trvania jedného scenára. Inými slovami, ako dlho trvalo jednému používateľovi vykonať všetkých 8 HTTP požiadaviek.



Obrázok 7.5. Artillery - celkový čas jedného scenára.

Ako vyhodnotenie rýchlosti odpovedí použijeme hodnotenie Apdex. Hodnotenie apdex kategorizuje používateľov na základe rýchlosti odpovedí do skupín: spokojní, tolerujúci a frustrovaní [46]. A rýchlosť odpovede kategorizuje do jednej z piatich kategórií:

Index	Apdex Hodnotenie
0.94 to 1	Excellent
0.85 to 0.93	Good
0.70 to 0.84	Fair
0.50 to 0.69	Poor
0.00 to 0.49	Unacceptable

Tabuľka 7.1. Skóre Apdex.

Apdex skóre pre aplikáciu FactCheck na danom hardvéri vyšlo: **0.86 (good)** .

Kapitola 8
Nasadenie a CI, CD

Pre zaistenie kvality kódu je potrebné pridať procesy nepretržitej integrácie (CI) a nepretržitého nasadenia (CD). CI nám zabezpečí, že každá zmena ktorá bude do systému pridaná prejde testami (ak boli implementované) a spustí okrem toho všetky ostatné testy a ubezpečí nás, že ani malá zmena nám nespôsobí znefunkčnenie iných častí systému. Po úspešnom spustení testov spúšťame proces CD, ktorý nám zabezpečí efektívne nasadenie novej verzie bez potreby manuálneho zásahu programátora. CI a CD sa nám postarajú o ďalšie zaistenie kvality kódu, ktorý vývojár alebo vývojársky tím pridá. Poskytne nám aj rýchlejší a častejší feedback od používateľov, vďaka častému nasadeniu nových verzií. Okrem toho nám zabezpečí konzistentnosť testovania a nasadenia, lepšiu spoluprácu medzi tímami a v neposlednom rade šetria náklady a čas (vývojári by museli manuálne spúšťať testy, nasadzovať, riešiť prípadné problémy).

V našom repozitári sa nachádza priečinok `.github`. Tento priečinok poskytuje definície a šablóny spojené s GitHub repozitárom. Nájdeme tu súbory, ktorých cieľom je automatizácia a zefektívnenie procesov v rámci nášho repozitára, ako napríklad Continuous Integration (CI) a Continuous Delivery (CD) [47].

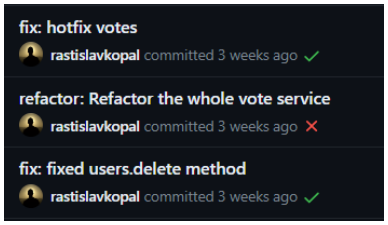
8.1 CI

Keďže sme už naprogramovali integračné testy, chceme pri každej `merge` požiadavke do `main` vetve repozitáru automaticky spúšťať tieto testy. Nastavili sme si preto CI pipeline, ktorá automaticky spúšťa nami napísané integračné testy. V prípade, že všetky testy prebehnú úspešne, môžme danú vetvu spojiť do vetve `main`. V opačnom prípade sa nám vráti chyba a je nutné kód opraviť. Pipeline je písaná v skriptovacom jazyku yaml.

```
name: FactCheckAPI CI
on:
  push:
    branches:
      - main
  pull-request:
    branches: [main]
jobs:
  tests:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [16.x]
        mongodb-version: ['6.0']
    steps:
      - name: Git checkout
        uses: actions/checkout@v3
```

```
      - name: Use Node.js 16.x
        uses: actions/setup-node@v3
      - name: Build App
        run: npm install
      - name: Cache node modules
        uses: actions/cache@v3
      - name: Start MongoDB
        uses: supercharge/mongodb-github-action@1.8.0
      - run: npm run test:e2e
```

Pri každom pridaní kódu do vetve `main`, alebo `merge` požiadavke sa spúšťa proces, ktorý na definovanom OS (ubuntu-latest) vytvorí MongoDB inštanciu (Verzie 6.0), nainštaluje našu službu (S verziou Node 16.x) pomocou príkazu **npm install**. Nakoniec spustí integračné testy príkazom **npm run test:e2e**. Na obrázku 8.1 môžeme vidieť úspech alebo chybu pri spúštaní integračných testov. V prípade chyby (červený znak 'x') je problém nutné vyriešiť pred ďalším pridaním kódu.



Obrázok 8.1. Spúštanie testov v CI pipeline v Github Actions.

8.2 Nasadenie

Pre nasadenie celého systému FactCheck sme si vytvorili nový repozitár, ktorý obsahuje definície potrebné k nasadeniu, premenné prostredia a konfiguračné súbory. Systém bude nasadený ako množina dockerizovaných služieb pomocou nástroja docker-compose.

```
// docker-compose.yaml
version: "3"

services:
  certbot:
    image: certbot/certbot
  nginx:
    image: rastokopal/fact-checking-fe:latest
    ports:
      - 80:80
      - 443:443
  api:
    image: rastokopal/fact-checking-api:latest
    ports:
      - 3000:3000
  extract-service:
    image: rastokopal/fact-checking-extract-service:1.0.0
    ports:
      - 8080:8080
```

```
  mongodb:
    image: mongo:latest
    ports:
      - 27017:27017
    command: [--auth]
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.1.2
    ports:
      - 9200:9200
  kibana:
    image: docker.elastic.co/kibana/kibana:8.1.2
    ports:
      - 5601:5601
  redis:
    image: redis:latest
    ports:
      - 6379:6379
```

Pre ilustračné účely boli vynechané definície premenných prostredia (ktoré konfigurujú jednotlivé služby), takzvané `volumes` a konfigurácie sietí (`networks`) - pre umožnenie komunikácie medzi službami. Spustením príkazu `docker-compose up -d` sa nám spustia všetky služby definované v `docker-compose.yaml`. Spustí sa nám MongoDB databáza a Redis s ktorými dokáže komunikovať náš FactCheckAPI servis. Môžme si všimnúť aj **extract-service**, ktorý poskytuje API pre automatizovanú extrakciu textu z URL. Ďalej tu používame servis **Certbot**, ktorý nám automaticky umožňuje obnovovať SSL certifikáty pre zabezpečenú komunikáciu pomocou služby **Nginx**. Službu Nginx máme konfigurovanú, aby fungovala ako API Gateway medzi jednotlivými službami. Službu ElasticSearch používame pre indexovanie logov z našich služieb a Kibanu pre vizualizáciu týchto dát a monitorovanie.

8.3 CD

Techniku Continuous Delivery využijeme pre automatizované nasadzovanie nových verzií. Je neudržateľné, aby vývojár po každej zmene kódu bol nútený sa pripojiť na server, pridať zmeny a reštartovať nasadenie. Všetko to zaberá čas a ľudský faktor prináša ďalšie chyby. Preto si zavedieme aj pipeline pre automatizované nasadenie, taktiež v jazyku yaml s využitím Github Actions.

```
name: FactCheckAPI CD
on:
  push:
    branches:
      - main
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Build and push Docker images
        run: |
```

```
        docker build -t rastokopal/fact-checking-api:latest .
        echo { secrets.DOCKER-PASSWORD } | docker login -u \
              { secrets.DOCKER-USERNAME } --password-stdin
        docker push rastokopal/fact-checking-api:latest
    - name: SSH and deploy
      uses: appleboy/ssh-action@master
      with:
        host: { secrets.SERVER-HOST }
        username: { secrets.SERVER-USERNAME }
        key: { secrets.SERVER-SSH-KEY }
        script: |
          docker pull rastokopal/fact-checking-api:latest
          cd { secrets.DEPLOY-FOLDER }
          docker-compose up -d --no-deps api
```

Keďže pre nasadenie aplikácie využívame kontajnerizované služby, zavedieme `build` týchto služieb automaticky po úspešnom spustení integračných testov z CI. Ako prvé si vybudujeme dockerizovaný obraz (`docker image`) podľa Dockerfile v repozitáre a odošleme tento nový `image` do zdielaného úložiska pre docker obrazy (DockerHub). Po úspešnom odoslaní sa pomocou SSH kĺuča pripojíme na náš server. Na serveri máme spustené produkčné nasadenie a v zložke so spustenými službami si stiahneme novú verziu zvoleného `image` a reštartujeme službu. V tomto bode máme novú verziu stiahnutú a nasadenú na serveri. Nová verzia je automaticky dostupná používateľom. Pipeline pritom využíva takzvané `secrets` pre premenné, ktoré nemôžu byť zverejnené. Ukladáme ich jednorázovo do Github repozitáru a už nikdy ich nie je možné znova zobraziť (Meno a heslo do DockerHubu, SSH kĺúč a podobne).

Kapitola 9
Budúcnosť platformy FactCheck

Na vývoji platformy FactCheck aktívne spolupracujeme s tímom z Centra umelej inteligencie (AIC) na Fakulte Elektrotechnickej na ČVUT. Tím ľudí z AIC sa aktívne venuje oblasti umelej inteligencie pre využitie priamo v overovaní faktov na internete. Budúci rozvoj platformy sa skladá najmä s integráciou vyvinutých NLP modelov členmi tímu. Tieto modely budú nasadené ako ďalšie služby, ktoré budú poskytovať REST API. Tím sa skladá z vývojového tímu platformy, študentov odboru umelej inteligencie a doktorandov pod vedením Ing. Jan Drchal, Ph.D. V spolupráci so spoločnosťami Avast Software s.r.o., Gen Digital Inc. a tímom ľudí z AIC vidíme v platforme veľký potenciál a rozvoj platformy máme v pláne aj naďalej. V tejto kapitole si popíšeme budúcnosť produktu, čo by malo byť ďalej pridané a akým smerom sa platforma môže uberať.

- Integrácia služby s využitím NLP modelov pre vyhľadávanie podobných tvrdení. Služba bude poskytovať REST API a klientským aplikáciam zobrazí zoznam podobných tvrdení. Táto služba sa bude využívať pri pridávaní nových tvrdení, aby sme znížili možnú redundanciu dát.
- Vylepšenie možnosti full-text vyhľadávania. Aktuálna implementácia využíva full-text index z lokálneho MongoDB dockerizovaného obrazu, ktorý nie je najefektívnejší. Cloudové riešenia ako MongoDB Atlas poskytujú výrazne lepšie výsledky pre vyhľadávanie. Prípadne sa dá využiť vyhľadávanie v kombinácií s vyššie spomínanou službou Elasticsearch, ktorá poskytuje lepši algoritmus pre skóre relevancie pri vyhľadávaní v textoch.
- Rozšíriť gamifikáciu platformy o prvky dlhodobej motivácie. Jeden z návrhov na zlepšenie bolo pridanie možnosti výziev. Používatelia dostanú plán na časové obdobie, ktoré treba dodržiavať a tým si zvýšia svoju reputáciu viac. Príklad výzvy môže byť napríklad `horúci týždeň`, kedy každý deň musí používateľ pridať aspoň `x` hodnotení. Po úspešnom dokončení výzvy používateľ dostane `y` reputácie. Možností podobných výziev je mnoho, ktoré ponúka aj napríklad vzdelávacia platforma Duolingo.
- Pridanie rolí `expert` do systému s tým, že ich hodnotenia budú v zozname hodnotení vždy zobrazované vyššie. Hodnotenia expertov budú mať taktiež väčšiu váhu a rovnako ich hlasy vo fázy hodnotenia. Máme v pláne do systému implementovať možnosť zbierania reputácií s ešte vyššou motiváciou. Motivácia má byť taká, že po dosiahnutí určitej hranice sa stanú taktiež špecialisti na hodnotenie. Tento proces bude musieť byť znova analyzovaný, implementovaný a otestovaný.

Kapitola 10
Záver

V tejto práci sme si ukázali kompletný cyklus vývoja softvérového systému. V prvých kapitolách sme sa venovali analýze existujúcich riešení a aplikácií s podobným cieľom. Zhodnotili sme ich silné a slabé stránky a popísali sme si dôvody pre návrh vlastného softvérového produktu pre využitie crowdsourcingu v overovaní faktov na internete s prvkami gamifikácie. Následne sme si predstavili a vysvetlili teóriu potrebnú pre návrh moderných, dobre udržiavateľných a škálovateľných backendových aplikácií.

V ďalšej kapitole sme vytvorili analýzu funkčných a nefunkčných požiadavkov v spolupráci s vedúcim práce Ing. Jan Drchal, Ph.D. a Ing. Roman Bútora - kolega pre vývoj klientskej časti aplikácie. Platformu sme vyvíjali agilným spôsobom počas 2 semestrov, počas ktorých sa požiadavky často menili. Výrazná zmena systému nastala po prvom kole testovania systému používateľmi - študentmi žurnalistiky z FSV UK. Študenti žurnalistiky popísali viaceré problémové časti systému. Do návrhu sme následne zakomponovali ich poznámky a návrhy spolu s prvkami gamifikácie pre udržanie používateľov na platforme. Následne sme boli schopní vytvoriť finálny návrh backendového riešenia.

Po návrhu základnej časti platformy sme navrhli technológie, ktoré budeme používať pri implementácií a postupne vyvinuli služby, ktoré poskytujú REST API pre komunikáciu s klientskými aplikáciami. Spolu s nimi sme pravidelne dodávali dokumentáciu nášho rozhrania REST API a umožňovali tým vývojárom na strane klienta jednoduchú integráciu. S kolegom Ing. Romanom Bútorom sme postupne vyvinuli kompletnú funkčnú platformu a webovú aplikáciu. Počas práce sme spolu často diskutovali rôzne problémy a riešenia, či už na klientskej strane aplikácie alebo na strane backendu.

Počas implementácie REST API sme kontinuálne vyvíjali aj integračné testy pre koncové body, ktoré automaticky testovali všetky časti systému pri každej väčšej zmene. Pridali sme procesy CI a CD pre automatizáciu testovania a nasadzovania aplikácie. Následne sme vyhodnotili pokrytie kódu integračnými testami.

V neposlednom rade sme vytvorili produkčnú verziu platformy, ktorá uchopí výhody kontajnerizácie a elegantne nasadí všetky služby do produkčného prostredia, bez potreby väčšieho zásahu vývojárskeho tímu.

Na záver sme zhrnuli budúcnosť platformy FactCheck. Popísali sme si rozšírenia, ktoré by bolo vhodné do produktu pridať. Na týchto rozšíreniach budeme s tímom pracovať aj naďalej, po ukončení štúdia.

Literatúra

[1] Soroush Vosoughi, Deb Roy a Sinan Aral. The spread of true and false news online. *Science*. 2018, 359 (6380), 1146-1151. DOI 10.1126/science.aap9559.

[2] Marcos Rodrigues Pinto, Yuri Oliveira de Lima, Carlos Eduardo Barbosa a Jano Moreira de Souza. *Towards fact-checking through crowdsourcing.* In: *2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. 2019. 494–499.

[3] Claire Wardle. *Fake news. It's complicated.* 2018. `https://firstdraftnews.org/articles/fake-news-complicated/`. Accessed on 07.12, 2023.

[4] Inc. Wikimedia Foundation. *Contributing to Wikipedia.* `https://en.wikipedia.org/wiki/Wikipedia:Contributing_to_Wikipedia`. Accessed on 07.12, 2023.

[5] S Penoyer, B Reynolds, B Marshall a P W Cardon. Impact of users' motivation on gamified crowdsourcing systems: A case of Stackoverflow. *Issues in Information Systems*. 2018 , 19 (2), 33–40.

[6] *FactCheck.org - A Project of The Annenberg Public Policy Center*. `https://www.factcheck.org/`. Accessed on 07.17, 2023.

[7] *Politifact*. `https://www.politifact.com/`. Accessed on 07.17, 2023.

[8] *Truth Setter - CROWD-SOURCING TRUTH*. `https://truthsetter.com`. Accessed on 07.17, 2023.

[9] *CaptainFact - Collaborative Fact Checking platform*. `https://captainfact.io/`. Accessed on 07.12, 2023.

[10] Mary Shaw a David Garlan. *Characteristics of Higher-Level Languages for Software Architecture.* . CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[11] Khalid Aljasser. Implementing design patterns as parametric aspects using ParaAJ: the case of the singleton, observer, and decorator design patterns. *Computer Languages, Systems & Structures*. 2016, 45 1–15.

[12] Nenad Medvidovic a Richard N Taylor. *Software architecture: foundations, theory, and practice.* In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 2010. 471–472.

[13] Nada Salaheddin a Nuredin Ahmed. MICROSERVICES VS. MONOLITHIC ARCHITECTURES [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES. 2022,

[14] Komal Modhiya. Introduction to DBMS, RDBMS, and NoSQL Database: NoSQL Database Challenges. *RDBMS, and NoSQL Database: NoSQL Database Challenges (March 6, 2021)*. 2021,

[15] Ying Bai. *SQL Server Database Programming with Java: Concepts, Designs and Implementations*. Springer Nature, 2022.

[16] Neal Leavitt. Will NoSQL databases live up to their promise?. *Computer*. 2010, 43 (2), 12–14.

[17] Rick Cattell. Scalable SQL and NoSQL data stores. *Acm Sigmod Record*. 2011, 39 (4), 12–27.

[18] Inc. HazelCast. *What is the CAP Theorem?g*. `https://hazelcast.com/glossary/cap-theorem/l`.

[19] Zachary Parker, Scott Poe a Susan V Vrbsky. *Comparing nosql mongodb to an sql db*. In: *Proceedings of the 51st ACM Southeast Conference*. 2013. 1–6.

[20] *MongoDB - Schema Design Best Practices*. `https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/`. Accessed on 07.17, 2023.

[21] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen a Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*. 2017,

[22] Mark Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. " O'Reilly Media, Inc.", 2011.

[23] Robin Wieruch. The Road to GraphQL. *Independently published*. 2018,

[24] Reuven M Lerner. At the forge: 12-factor apps. *Linux Journal*. 2014, 2014 (245), 5.

[25] P Sriramya a RA Karthika. Providing password security by salted password hashing using bcrypt algorithm. *ARPN journal of engineering and applied sciences*. 2015, 10 (13), 5551–5556.

[26] Alex Biryukov, Daniel Dinu a Dmitry Khovratovich. *Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications*. In: *2016 IEEE European Symposium on Security and Privacy*. 2016. 292-302.

[27] Rolf Oppliger. *SSL and TLS: Theory and Practice*. Artech House, 2016.

[28] Krishna Shingala. Json web token (jwt) based client authentication in message queuing telemetry transport (mqtt). *arXiv preprint arXiv:1903.02895*. 2019,

[29] Jonathan Dyson. *Conjoining FURPS and MoSCoW to Analyse and Prioritise Requirements*. 2019. `https://www.linkedin.com/pulse/conjoining-furps-moscow-analyse-prioritise-jonathan-dyson/`.

[30] Catherine Sotirakou, Theodoros Paraskevas a Constantinos Mourlas. *Toward the Design of a Gamification Framework for Enhancing Motivation Among Journalists, Experts, and the Public to Combat Disinformation: The Case of CALYPSO Platform*. In: *HCI in Games: 4th International Conference, HCI-Games 2022, Held as Part of the 24th HCI International Conference, HCII 2022, Virtual Event, June 26–July 1, 2022, Proceedings*. 2022. 542–554.

[31] *Stackoverflow - Empowering the world to develop technology through collective knowledge*. `https://stackoverflow.com`.

[32] Adrien Barbaresi. *Trafilatura: A Web Scraping Library and Command-Line Tool for Text Discovery and Extraction*. In: *Proceedings of the Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and*

the 11th International Joint Conference on Natural Language Processing: System Demonstrations. Association for Computational Linguistics, 2021. 122–131. https://aclanthology.org/2021.acl-demo.15.

[33] elasticsearch. *elasticsearch/elasticsearch*. 2015 . https://github.com/elasticsearch/elasticsearch.

[34] Robert C. Martin (Uncle Bob). *The Clean Architecture*. 2012. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[35] David Jaramillo, Duy V Nguyen a Robert Smart. *Leveraging microservices architecture by using Docker technology.* In: *SoutheastCon 2016*. 2016. 1–5.

[36] SL Bangare, S Gupta, M Dalal a A Inamdar. *Using Node. Js to build high speed and scalable backend database server.* In: *Proc. NCPCI. Conf.* 2016. 19.

[37] Gavin Bierman, Martin Abadi a Mads Torgersen. *Understanding typescript.* In: *ECOOP 2014, Object-Oriented Programming: 28th European Conference, Uppsala, Sweden*. 2014. 257–281.

[38] Kamil Mysliwiec. *Nest.js, A progressive Node.js framework for building efficient, reliable and scalable server-side applications.* 2017. https://nestjs.com/.

[39] Simon Holmes. *Mongoose for Application Development*. Packt Publishing Ltd, 2013.

[40] *BullMQ - The fastest, most reliable, Redis-based distributed queue for Node.* https://api.docs.bullmq.io/. Accessed on 09.20, 2023.

[41] *ESLint - Static code analyzer*. https://eslint.org/docs/latest/. Accessed on 07.27, 2023.

[42] *Prettier - an opinionated code formatter*. https://prettier.io/docs/en/index.html. Accessed on 07.27, 2023.

[43] Typicode. *Husky*. 2021. [Online]. Available: https://www.npmjs.com/package/husky.

[44] *Jest - a delightful JavaScript Testing Framework with a focus on simplicity.* https://jestjs.io/. Accessed on 07.27, 2023.

[45] *Artillery - Never Fail To Scale.* https://www.artillery.io/docs. Accessed on 08.02, 2023.

[46] J Brady. The apdex index vs traditional management information decision tools. 2009,

[47] Inc. GitHub. *GitHub Workflows*. https://docs.github.com/en/actions. 2023.