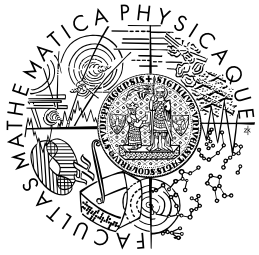

MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Adam Turčan

Webová aplikace pro konfigurovatelné zpracování textu s využitím externích služeb

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: doc. RNDr. Pavel Pecina, Ph.D.
Studijní program: Informatika

Praha 2026

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Poděkování.

Název práce: Webová aplikace pro konfigurovatelné zpracování textu s využitím externích služeb

Autor: Adam Turčan

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: doc. RNDr. Pavel Pecina, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Abstrakt.

Klíčová slova: zpracování přirozených jazyků, uživatelské rozhraní, webová aplikace

Title: Web application for configurable text processing using external services

Author: Adam Turčan

Institute: Institute of Formal and Applied Linguistics

Supervisor: doc. RNDr. Pavel Pecina, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Abstract.

Keywords: natural language processing, user interface, web application

Obsah

Úvod

Vývoj softvérových nástrojov pre špecifické vedecké domény so sebou prináša unikátne výzvy. Vyžaduje nielen hlboké pochopenie doménových dát, ale aj schopnosť navrhnúť intuitívne používateľské rozhranie (User Interface (UI)/User Experience (UX)) pre používateľov, ktorí nemusia byť technicky zdatní. Zároveň však narážame na problém, že riešenia „šité na mieru“ jednej doméne sú často príliš úzko špecializované, a teda ťažko prenositeľné do iných oblastí.

Táto bakalárska práca sa zaoberá kompletným životným cyklom vývoja platformy určenej na spracovanie textu s využitím externých Natural Language Processing (NLP) služieb, akými sú preklad, segmentácia, anotácia a semantické značkovanie. Opisuje cestu od prvotného návrhu používateľskej skúsenosti (UX) pre potreby projektu MEMORISE, cez implementáciu počiatočnej architektúry, až po jej následnú evolúciu do podoby univerzálneho, konfigurovateľného systému. Práca tak neprezentuje len finálny stav, ale dokumentuje iteratívny proces, v ktorom prvá implementácia slúžila ako kľúčový prostriedok na identifikáciu architektonických nedostatkov a validáciu interakčného dizajnu.

Projekt MEMORISE a technologické výzvy

Táto práca je zasadená do kontextu medzinárodného výskumného projektu MEMORISE, ktorého cieľom je zachovanie spomienok obetí holokaustu a sprístupnenie ich príbehov pre budúce generácie. Vzhľadom na to, že priami svedkovia a pamätníci holokaustu postupne odchádzajú, sa projekt zameriava na digitalizáciu viac ako 80 000 historických záznamov [1], vrátane denníkov obetí, svedectiev a multimediálnych materiálov.

V rámci pracovných balíkov projektu zameraných na dátovú infraštruktúru a integráciu Work Package 2: Data Infrastructure (WP2) a Work Package 3: Data Integration (WP3) vzniká komplexný proces pre spracovanie týchto dát. Cieľom je transformovať neštruktúrované historické texty do podoby prepojených dát a znalostných grafov, ktoré následne slúžia ako podklad pre vizualizáciu.

Tento proces využíva pokročilé metódy umelej inteligencie a spracovania prirodzeného jazyka (NLP). Automatizované služby zabezpečujú:

- **Segmentáciu textu**: Rozdelenie dlhých textov (napríklad denníkov) na logické celky.
- **Strojový preklad**: Sprístupnenie dokumentov v rôznych jazykoch.
- **Semantické značkovanie**: Identifikáciu tém a kontextu.
- **Rozpoznávanie pomenovaných entít (ďalej ako NER)**: Extrakciu mien osôb, geografických lokácií (napr. getá, tábory) a časových údajov.

Rola kurácie dát v NLP procesoch

Hoci moderné NLP modely dosahujú vysokú presnosť, pri práci s citlivými historickými dátami nie je plná automatizácia postačujúca. Výstupy modelov

často obsahujú chyby v rozpoznávaní entít (napr. zámena mena osoby za názov mesta), nepresnosti v preklade či nevhodnú segmentáciu, ktorá spôsobuje nelogické členenie textu a stratu kontextu. Tieto nedostatky by bez zásahu človeka mohli viesť k nesprávnej interpretácii historických faktov.

Ako uvádza metodika projektu MEMORISE (konkrétne **WP3** [2]), pre dosiahnutie vysokej kvality dát je nevyhnutný prístup *Human-in-the-loop*. To znamená, že automaticky generované metadáta musia byť validované a korigované ľudskými expertmi – kurátormi.

Pôvodná motivácia pre vznik tohto nástroja vychádzala práve z potreby poskytnúť expertom efektívne používateľské rozhranie pre:

- **Vizualizáciu výstupov z NLP Application Programming Interface (API):** Prehľadné zobrazenie segmentov, prekladov a navrhovaných entít.
- **Post-editáciu prekladov a segmentácie:** Možnosť opraviť gramatické a kontextové nepresnosti vzniknuté pri strojovom preklade a segmentácii.
- **Validáciu anotácií:** Rýchlu korekciu nesprávne klasifikovaných entít a úpravu priradených semantických značiek.
- **Semantické prepájanie (Linking[1]):** Manuálne dopĺňanie chýbajúcich väzieb na externé ontológie (napr. prepojenie na holandský tezaurus 2. svetovej vojny).

Efektívna interakcia s týmito funkciami prináša významnú pridanú hodnotu aj mimo samotnej kurácie dát. Nástroj slúži na zefektívnenie práce expertov, ktorí využívajú anotované texty pri tvorbe **odborných štúdií, článkov či vzdelávacích materiálov**. Namiesto zdĺhavého manuálneho značkovania a prekladania textov „od nuly“ získavajú predpripravený podklad, ktorý stačí následne len validovať. Tým sa ťažisko ich práce presúva od mechanickej činnosti k expertnej analýze a interpretácii prameňov. Tento posun je v súlade s aktuálnymi trendmi v edukácii a výskume, kde automatizácia rutinných manuálnych úloh umožňuje špecialistom sústrediť sa na činnosti s vyššou pridanou hodnotou [3, sek. 4.6.6].

Potreba takéhoto rozhrania sa však neobmedzuje len na oblasť digitálnej histórie. S masívnym nástupom NLP a generatívnych modelov do praxe, čo potvrdzujú aj aktuálne štatistiky exponenciálneho rastu publikácií a aplikácií [3, sek. 4.1], narážajú na identický problém mnohé iné domény.

Kľúčová je precíznosť dát – napríklad v **medicíne**, kde sa modely využívajú na analýzu rozsiahlych objemov dát z klinických záznamov a diagnostických správ (*patient records, diagnostic reports*) [3, sek. 4.6.4]. Analogickým príkladom je oblasť **práva a etiky**, kde experti riešia napríklad dopady generovaného obsahu na duševné vlastníctvo a potrebu tvorby dôkladných právnych rámcov (*legal frameworks*) [3, sek. 4.6.2].

V týchto oblastiach, rovnako ako v projekte MEMORISE, nemožno slepo dôverovať výstupom umelej inteligencie. Ako uvádzajú autori v kontexte kritických aplikácií, integrácia týchto nástrojov si vyžaduje „zachovanie ľudského dohľadu“ (*preserving human oversight*) [3, sek. 4.6.4], čím sa prístup *Human-in-the-loop* stáva nevyhnutnosťou.

[1] Ide o techniku prepájania dát, informácií alebo dokumentov na základe ich významu (sémantiky) a vzájomných vzťahov.

Vytvárať pre každú takúto doménu samostatnú, jednoúčelovú aplikáciu by však bolo neefektívne a dlhodobo neudržateľné. Existujúce riešenia často narážajú na obmedzenia flexibility, teda sú buď príliš všeobecné a nepokrývajú špecifické potreby (ako je súbežná úprava segmentácie a prekladu), alebo sú naopak pevne naviazané na konkrétnu dátovú schému.

Motiváciou tejto práce je preto návrh a implementácia **univerzálnej, konfigurovateľnej platformy**. Cieľom je vytvoriť systém, ktorý dokáže dynamicky spracovávať výstupy z rôznych NLP služieb a prispôsobiť sa odlišným anotačným schémam iba prostredníctvom zmeny konfigurácie, bez nutnosti zásahov do zdrojového kódu.

Ciele práce

Cieľom práce nie je len vytvorenie jednorazovej aplikácie, ale demonštrácia procesu architektonickej generalizácie. Ciele sú stanovené v troch logicky nadväzujúcich krokoch:

1. **Návrh UX a implementácia doménového prototypu (Fáza 1):** Navrhnúť jednotný a intuitívny dizajn (UI/UX), ktorý zefektívni prácu anotátorov. Na základe tohto dizajnu implementovať funkčný systém spĺňajúci kľúčové požiadavky projektu MEMORISE.

2. **Analýza a identifikácia obmedzení:** Podrobiť východiskovú („baseline“) architektúru kritickej analýze a identifikovať technologické nedostatky, ktoré bránia škálovateľnosti a širšiemu využitiu.

3. **Architektonická evolúcia (Fáza 2):** Na základe analýzy transformovať systém na modulárnu klient-server platformu. Cieľom je oddeliť stabilné používateľské rozhranie od premenlivej biznis logiky, a tým umožniť dynamickú konfiguráciu nástroja pre rôzne domény bez nutnosti rozsiahlych zásahov do kódu.

Štruktúra práce

Text práce reflektuje tento evolučný postup a je rozdelený do šiestich kapitol:

- **Kapitola 1 (Teoretické východiská)** popisuje princípy tvorby moderných webových aplikácií, pričom sa zameriava na architektonické vzory a technológie využívané pri spracovaní textu a tvorbe používateľských rozhraní.

- **Kapitola 2 (Dizajn a počiatočná implementácia)** sa venuje prvej fáze vývoja. Popisuje proces návrhu UI/UX, ktorý zostáva konzistentný naprieč verziami, a dokumentuje počiatočnú architektúru systému.

- **Kapitola 3 (Analýza obmedzení a refactoring)** detailne analyzuje nedostatky počiatočného riešenia a definuje požiadavky na novú, **univerzálnu a konfigurovateľnú** architektúru.

- **Kapitola 4 (Návrh a implementácia cieľovej architektúry)** popisuje finálne technické riešenie – prechod na klient-server model, využitie asynchrónnych workerov pre tezaury a implementáciu konfiguračného mechanizmu.
- **Kapitola 5 (Overenie riešenia)** sumarizuje dosiahnuté výsledky, prezentuje testovanie v reálnej prevádzke a hodnotí, do akej miery nová architektúra splnila stanovené ciele flexibility.
- **Záver** zhrňuje hlavné prínosy práce, diskutuje o jej limitoch a navrhuje smery pre budúci výskum a rozvoj platformy.

1 Teoretické východiská

Táto kapitola predstavuje prehľad kľúčových konceptov vývoja webových aplikácií, architektonických vzorov a technológií, ktoré tvoria teoretický základ pre návrh a implementáciu prezentovanej platformy. Rozoberá techniky optimálneho vykresľovania prvkov na UI, synchronizáciu stavov a distribúciu dát.

1.1 Moderné webové technológie

Vývoj webových aplikácií prešiel od svojho vzniku zásadnou transformáciou. Zatiaľ čo počiatočná éra (Web 1.0) bola charakteristická statickými stránkami slúžiacimi primárne na konzumáciu obsahu, súčasné aplikácie ponúkajú interaktivitu porovnateľnú s komplexnými informačnými systémami, ktoré sú súčasťou nášho digitálneho života [4, sek. 1].

Historický posun od statických stránok k dynamickým bol spočiatku poháňaný serverovými skriptami Common Gateway Interface (CGI) a neskôr príchodom technológie AJAX, ktorá umožnila asynchrónnu výmenu dát bez nutnosti obnovovať celú stránku [4, sek. 1.1]. Skutočnú revolúciu však priniesli až moderné štandardy ako HTML5, CSS3 a nové verzie JavaScriptu (ECMAScript 6 (ES6)+), ktoré výrazne rozšírili možnosti prehliadačov [4, sek. 1.3].

Tieto technologické inovácie umožnili presunúť ťažisko aplikačnej logiky zo servera na klienta, čím vytvorili priestor pre vznik architektonických modelov kladúcich dôraz na UX a výkon. Tento posun klientskej logiky vo svojom článku potvrdzuje aj Vijay Panwar, podľa ktorého súčasný stav vývoja kladie prioritný dôraz na používateľskú skúsenosť UX a výkon aplikácie. Tieto aspekty sú kritické pre moderné riešenia a vyžadujú si rôzne optimalizačné stratégie [4, sek. 2.3], ktorým sa venujú nasledujúce časti tejto kapitoly.

1.1.1 Jazyk TypeScript a statická analýza

S narastajúcou komplexnosťou klientskych aplikácií, na ktorú sme poukázali v predchádzajúcej sekcii, naráža vývoj v čistom JavaScripte na limity svojej udržateľnosti. Bierman a kol. [5, sek 1], vo svojej práci priamo konštatujú, že napriek celosvetovému úspechu ostáva JavaScript nevhodným nástrojom pre vývoj a údržbu rozsiahlych softvérových systémov.

Ako odpoveď na tieto nedostatky vznikol jazyk **TypeScript**. Ide o syntaktickú nadmnožinu JavaScriptu, ktorá obohacuje jazyk o systém modulov, tried a predovšetkým o statické typovanie.

Tento prídavok umožňuje vývojárom využívať pokročilé nástroje a podporu v Integrated Development Environment (IDE) (napr. inteligentné dopĺňanie kódu alebo refaktoring), akú poznáme z klasických objektovo orientovaných jazykov. Týmto sa predchádza runtime chybám pri manipulácii s komplexnými dátovými štruktúrami už počas písania kódu.

Kľúčovým prínosom TypeScriptu pre architektúru našej aplikácie je podpora typových rozhraní. Tie umožňujú definovať striktné kontrakty pre dátové štruktúry a objekty, čím sa zvyšuje modularita a znovupoužiteľnosť komponentov.

Je však dôležité poznamenať, že webové prehliadače nedokážu interpretovať TypeScript priamo. Kód je preto v procese transpilácie[1] (pomocou kompilátora alebo nástrojov ako Babel) prevedený na štandardný JavaScript.

Tento mechanizmus zabezpečuje spätnú kompatibilitu s celým existujúcim ekosystémom knižníc a beh v akomkoľvek prostredí [5, sek 1].

1.2 Architektúra webových aplikácií

Pri návrhu komplexných webových systémov sa dnes primárne rozlišuje medzi dvoma prístupmi [zdroj]: tradičným modelom Multi Page Application (MPA) a modernejšou architektúrou Single Page Application (SPA). Zatiaľ čo MPA prenáša väčšinu aplikačnej logiky na server a pri každej interakcii vyžaduje opätovné načítanie dokumentu, SPA umožňuje plynulejšiu interakciu tým, že rozhranie aktualizuje asynchrónne priamo na strane klienta. (tu pridam este par viet o komponentovej architekture a jej vyzname)Výber medzi týmito modelmi závisí predovšetkým od charakteru aplikácie, no pre systémy s vysokou frekvenciou drobných interakcií, ako sú napríklad textové editory, sa ako vhodnejší javí model SPA.

1.2.1 Virtuálny DOM a stratégie vykresľovania

Efektivita aktualizácie používateľského rozhrania v rámci architektúry SPA priamo závisí od zvolenej stratégie vykresľovania. Pre analýzu technologických možností sme zvolili tri dominantné stratégie, ktoré reprezentujú hlavné vývojové generácie frameworkov a podľa štatistík pokrývajú viac ako x % [zdroj, potrebujem preverit], trhu frontendových technológií:

Dirty Checking: Tento mechanizmus, známy napríklad zo starších verzií frameworku AngularJS, spočíva v cyklickej kontrole všetkých sledovaných premenných pri každej interakcii používateľa. Hoci je tento model jednoduchý na implementáciu, s rastúcim počtom sledovaných prvkov (napr. buniek v rozsiahlej tabuľke) jeho výkon rapídne klesá, čo vedie k spomaleniu odozvy aplikácie.

Fine-grained Reactivity: Moderné frameworky ako Svelte alebo SolidJS využívajú prístup, pri ktorom sa závislosti medzi dátami a Document Object Model (DOM) elementmi vyhodnocujú už počas kompilácie. Pri zmene hodnoty sa neporovnávajú stromy komponentov, ale priamo sa aktualizuje konkrétny textový uzol v prehliadači. Tento prístup je extrémne rýchly pri jednoduchých aktualizáciách, no môže byť menej flexibilný pri vysoko dynamických štruktúrach.

Virtual Document Object Model (VDOM): Funguje ako abstraktná pamäťová reprezentácia reálneho DOM. Proces aktualizácie využíva heuristický algoritmus porovnávania, ktorý dosahuje lineárnu zložitosť. Tento algoritmus

[1] Transpilácia (source-to-source compilation) je špecifický druh kompilácie, pri ktorom sa zdrojový kód napísaný v jednom programovacom jazyku transformuje do iného jazyka s podobnou úrovňou abstrakcie (napr. z TypeScriptu do JavaScriptu), na rozdiel od tradičnej kompilácie do nižšieho strojového kódu.

identifikuje minimálnu množinu rozdielov medzi starou a novou verziou virtuálneho stromu. Následne sú tieto zmeny hromadne a efektívne aplikované do reálneho prehliadača, čo zabezpečuje vysoký výkon aj pri vizuálne náročných a frekventovaných operáciách prekresľovania.

(Tento odsek asi presuniem az do kapitoly o architekture, kedze tam popisujem uz nasu volbu)Pre potreby našej platformy sa javí prístup použitia VDOM ako najvhodnejšia voľba, kedže jadrom aplikácie je vlastný textový editor, v ktorom používatelia v reálnom čase pracujú s farebne zvýraznenými entitami a hranicami segmentov textu. Tieto vizuálne prvky sú priamo závislé od pozícií znakov. Pri písaní alebo úprave textu dochádza k neustálym posunom týchto pozícií, čo si vyžaduje okamžité prepočítanie a prekreslenie veľkého množstva DOM elementov. Práve v tomto scenári sa prejavuje hlavný benefit VDOM, ktorý tento proces automatizuje a optimalizuje bez nutnosti manuálneho zásahu do štruktúry dokumentu.

1.2.2 Tok dát a správa stavu aplikácie

Ďalším kritickým aspektom v architektúre moderných webových aplikácií je mechanizmus distribúcie dát a správy stavu. Základným princípom v komponentovej architektúre je tzv. **jednosmerný tok dát**. V tomto modeli sú dáta distribuované hierarchicky zhora nadol – od rodičovských komponentov k potomkom.

Každý komponent môže disponovať aj vlastným vnútorným stavom, ktorý je zapuzdrený a ovplyvňuje len jeho vlastné vykresľovanie. Tento model je postačujúci pre menšie aplikácie, avšak pri komplexných systémoch, akým je aj naša platforma, naráža na limity udržateľnosti.

Problém Prop Drilling

S rastúcou hĺbkou stromu komponentov vzniká architektonický antipattern známy ako **Prop Drilling**. Ak potrebuje hlboko vnorený komponent prístup k dátam z koreňového komponentu, tieto dáta musia byť explicitne odovzdávané cez všetky medziľahlé vrstvy, aj keď ich tieto vrstvy vôbec nevyužívajú.

Tento proces **explicitného** posúvania dát vedie k problému vysokej previazanosti (tight coupling) a zbytočnému prekresľovaniu komponentov, ktoré v tomto reťazci slúžia len ako „prenosové kanály“.

Externá správa stavu

Ako odpoveď na tento architektonický nedostatok vznikli vzory pre **externú správu stavu**, ako napríklad architektúra Flux alebo jej moderné implementácie (Redux, Zustand). Tieto knižnice vyčleňujú aplikačný stav mimo hierarchiu komponentov do centralizovaného úložiska (tzv. *Store*).

Centralizované úložisko umožňuje komponentom pristupovať k potrebným dátam priamo, čím sa eliminuje zbytočná závislosť medzi jednotlivými úrovňami komponentového stromu. Zároveň sa tým jasne oddeľuje prezentačná vrstva od biznis logiky aplikácie.

1.2.3 Komunikácia so serverom

Keďže v SPA je prezentačná vrstva logicky oddelená od dátovej vrstvy, komunikácia prebieha prostredníctvom definovaného aplikačného rozhrania (API).

Najrozšírenejším štandardom je architektúra Representational State Transfer (REST)[zdroj]. V tomto modeli klient pristupuje k dátam prostredníctvom metód protokolu Hypertext Transfer Protocol (HTTP), ako sú GET, POST, PUT či DELETE. Výmena dát zvyčajne prebieha vo formáte JavaScript Object Notation (JSON), ktorý je natívne podporovaný jazykom JavaScript. Využitie architektúry REST zabezpečuje vysokú mieru interoperability(mozno pridat vysvetlivku) medzi klientskou aplikáciou a serverovými službami.

1.3 Návrhové vzory a princípy modularity

Modularita je základným predpokladom pre udržateľnosť a rozšíriteľnosť softvérových systémov. V kontexte vývoja moderných klientskych aplikácií sa pri organizácii kódu opierame o overené návrhové vzory, ktoré pomáhajú organizovať kód do logických, testovateľných a znovupoužiteľných celkov.

Pri návrhu architektúry sa najčastejšie uplatňujú nasledujúce vzory [zdroj]:

- **Module Pattern:** Tento vzor využíva princíp zapuzdrenia na vytvorenie privátneho rozsahu platnosti. Umožňuje rozdeliť kód do samostatných jednotiek a explicitne definovať, ktoré časti (funkcie, premenné) tvoria verejné rozhranie a ktoré ostávajú skryté pred okolitým svetom. Tým sa minimalizuje riziko konfliktov v globálnom mennom priestore.
- **Observer Pattern:** Definuje závislosť typu jeden-k-mnohým, kde zmena stavu jedného objektu automaticky notifikuje všetky závislé objekty. Tento mechanizmus zabezpečuje konzistenciu medzi dátovým modelom a používateľským rozhraním bez nutnosti silnej previazanosti, čo umožňuje reagovať na zmeny v dátovom modeli bez nutnosti manuálneho prekresľovania rozhrania.
- **Oddelenie prezentácie a logiky:** Vzor špecifický pre vývoj používateľských rozhraní, ktorý navrhuje striktné rozdelenie komponentov na dve kategórie. Prvá kategória zodpovedá za získavanie a spracovanie dát, zatiaľ čo druhá kategória sa stará výhradne o vizualizáciu na základe prijatých parametrov.

Okrem konkrétnych vzorov je pre kvalitu kódu kľúčové dodržiavanie architektonických princípov, ktoré definujú zodpovednosti jednotlivých modulov.

1.3.1 Princípy SOLID

Skratka SOLID predstavuje päticu princípov objektovo orientovaného návrhu, ktoré sú však plne aplikovateľné aj v kontexte komponentovej architektúry. Ich dodržiavanie vedie k systému, ktorý je odolnejší voči zmenám a ľahšie udržiateľný.

S – Single Responsibility Principle (SRP): Každý modul alebo komponent by mal mať iba jeden dôvod na zmenu. V praxi to znamená, že jedna časť systému by nemala riešiť nesúvisiace úlohy, napríklad formátovanie dát a zároveň ich načítavanie zo servera.

O – Open/Closed Principle (OCP): Softvérové entity by mali byť otvorené pre rozšírenie, ale uzavreté pre modifikáciu. To znamená, že novú funkcionalitu by malo byť možné pridať vytváraním nových modulov alebo kompozíciou existujúcich, bez nutnosti zásahu do už otestovaného zdrojového kódu.

L – Liskov Substitution Principle (LSP): Podtypy musia byť zastupiteľné za svoje bázové typy. V kontexte typovaných jazykov to zaručuje, že ak komponent závisí od určitého typového rozhrania, musí byť schopný fungovať s akoukoľvek jeho korektnou implementáciou bez neočakávaných chýb.

I – Interface Segregation Principle (ISP): Klienti by nemali byť nútení závisieť od rozhraní, ktoré nepoužívajú. Namiesto vytvárania robustných, všeobjímajúcich rozhraní je vhodnejšie definovať viacero menších a špecifických kontraktov.

D – Dependency Inversion Principle (DIP): Moduly vyššej úrovne by nemali závisieť od modulov nižšej úrovne; oba by mali závisieť od abstrakcií. Tento princíp je kritický pre testovateľnosť aplikácie, pretože umožňuje jednoduché nahradenie konkrétnych implementácií za iné.

1.4 Algoritmy a paralelizmus

Spracovanie textu na klientskej strane prináša špecifické výzvy v oblasti výkonu. Keďže JavaScript v prehliadači operuje na jednom hlavnom vlákne, náročné algoritmické operácie môžu spôsobiť zamrznutie používateľského rozhrania.

1.4.1 Problematika Fuzzy vyhľadávania

doplním počas pisania kapitoly o navrhu a implementacii prvotneho prototypu pre MEMORISE

1.4.2 Event Loop a asynchrónne spracovanie

JavaScript je z princípu jednovláknový jazyk. Využíva model založený na tzv. **Event Loop**, ktorá riadi vykonávanie kódu, spracovanie udalostí a vykresľovanie UI. Všetky operácie zdieľajú jedno hlavné vlákno. Akákoľvek dlhotrvajúca synchrónna operácia zablokuje Event Loop. Dôsledkom je, že prehliadač nemôže reagovať na vstupy používateľa ani prekresľovať obrazovku, čo vedie k zlej používateľskej skúsenosti.

1.4.3 Web Workers

Na riešenie problému blokovania hlavného vlákna poskytuje moderný webový štandard API **Web Workers**. Web Workers umožňujú spúšťať skripty na pozadí, v samostatnom vlákne, oddelene od hlavného vlákna stránky. Vďaka tomu môžu prebiehať náročné výpočty paralelne bez toho, aby ovplyvnili plynulosť UI.

Komunikácia medzi hlavným vláknom a workerom prebieha prostredníctvom zasielania správ, čo eliminuje problémy so súbehom, ktoré sú bežné pri viacvláknovom programovaní so zdieľanou pamäťou. V našom riešení využívame Web Workers na asynchrónne načítanie a prehľadávanie rozsiahlych ontologických dát.

Literatura

1. MEMORISE Consortium. *MEMORISE Project Objectives* [online]. [cit. 2026]. Dostupné z : https://memorise.sdu.dk/about-memorise/.
2. MEMORISE Consortium. *MEMORISE Project Methodology: Work Package 3 – Data Infrastructure* [online]. [cit. 2026]. Dostupné z : https://memorise.sdu.dk/about-memorise/.
3. Sharma, Chetan; Sharma, Shamneesh; Bhardwaj, Vivek; Dhaliwal, Balwinder Kaur. Generative artificial intelligence for sustainable development: predictive trend analysis in key sectors using natural language processing. *Discover Applied Sciences*. 2025, roč. 7, č. 596. Dostupné z doi: 10.1007/s42452-025-07207-7.
4. Panwar, Vijay. Web Evolution to Revolution: Navigating the Future of Web Application Development. *International Journal of Computer Trends and Technology (IJCTT)*. 2024, roč. 72, č. 2, s. 34–40. Dostupné z doi: 10.14445/22312803/IJCTT-V72I2P107.
5. Bierman, Gavin; Abadi, Martín; Torgersen, Mads. Understanding TypeScript. 2014, roč. 8586, s. 257–281. Dostupné z doi: 10.1007/978-3-662-44202-9_11.

Seznam použitých zkratek

API	Application Programming Interface
DOM	Document Object Model
JSON	JavaScript Object Notation
NLP	Natural Language Processing
REST	Representational State Transfer
SPA	Single Page Application
UI	User Interface
UX	User Experience
WP2	Work Package 2: Data Infrastructure
WP3	Work Package 3: Data Integration
SPA	Single Page Application
ES6	ECMAScript 6
CGI	Common Gateway Interface
IDE	Integrated Development Environment
DOM	Document Object Model
VDOM	Virtual Document Object Model
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
MPA	Multi Page Application