Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Systém kontroly pravopisu pro webový editor

Plzeň 2020 Bc. Patrik Harag

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 20. května 2020

Bc. Patrik Harag

Poděkování

Tímto bych chtěl poděkovat Ing. Petru Slabému a Ing. Martinu Dostalovi, Ph.D. za jejich rady a připomínky v průběhu zpracování této práce.

Abstract

This master's thesis deals with the development of a spell checker for a web editor. The theoretical part provides an introduction to the problematic of spell checking systems, describes structures suitable for the representation of dictionaries and string similarity search algorithms. Necessary technologies such as Hunspell and GWT are also introduced. The practical part describes experiments with different representations of dictionaries, from which the system design emerged. It then describes the final implementation of the created spell checking system and demonstrates its integration on a sample application with an editor. The work also deals with testing of the created system.

Abstrakt

Tato diplomová práce se zabývá vývojem systému kontroly pravopisu pro použití ve webovém editoru. Teoretické část poskytuje úvod do problematiky systémů pro kontrolu pravopisu, popisuje struktury vhodné pro reprezentaci slovníků a algoritmy pro hledání podobných slov. Dále jsou představeny potřebné technologie jako Hunspell a GWT. Praktická část popisuje experimenty s různými reprezentacemi slovníků, ze kterých vzešel návrh systému. Dále pak popisuje finální implementaci vytvořeného systému pro kontrolu pravopisu a na ukázkové aplikaci s editorem předvádí jeho integraci. Práce se zabývá také testováním vytvořeného systému.

Obsah

1 Úvod

Systémy pro kontrolu pravopisu jsou nedílnou součástí textových procesorů, emailových klientů, komunikátorů a obecně většiny programů, u kterých je očekáván textový vstup v přirozeném jazyce. Jejich úkolem je upozornit nás zejména na jednoduché překlepy a časté chyby. Ušetřit práci nám mohou i ve fázi opravy, a to nabídnutím možných oprav.

Jedním z úspěšných systémů pro kontrolu pravopisu je Hunspell. Jedná se o svobodný software s otevřenými zdrojovými kódy. Postupně si jako knihovna našel cestu do mnoha populárních programů. Jeho rozšíření má za následek dostupnost slovníků pro velké množství jazyků.

Jak se aplikace přesouvají do webového prostředí, tak se s nimi přesouvá i potřeba po systémech pro kontrolu pravopisu. Prověřené, výkonné a svobodné nástroje typu Hunspell zde však chybí. Prohlížeče sice u běžných textových polí kontrolu pravopisu zajistí, ale při tvorbě vlastní komponenty nebo WYSIWYG editoru se použít nedají. Systémy obsažené v prohlížečích také neumožňují konfiguraci slovníků. To může být problém, protože není žádoucí, když je název společnosti nebo produktu označen jako chybné slovo.

Vytvořit systém pro kontrolu pravopisu pro použití ve webovém prostředí s sebou však přináší nové technické výzvy. Problémem je výkon JavaScriptu, ale při jeho typovém systému, i způsob reprezentace slovníků o milionech slov.

Cílem této práce je poskytnut úvod do problematiky systémů pro kontrolu pravopisu, seznámit čtenáře s frameworkem GWT a nástrojem Hunspell. Dále pak navrhnout a implementovat efektivní systém pro kontrolu pravopisu ve webovém prostředí, který bude podporovat slovníky systému Hunspell. A nakonec vytvořený systém otestovat a integrovat do aplikace postavené na platformě GWT.

2 Systémy pro kontrolu pravopisu

V oblasti počítačové lingvistiky rozlišujeme dva typy systémů:

- systémy pro kontrolu pravopisu,
- systémy pro kontrolu gramatické správnosti.

Slovník spisovného jazyka českého [10] definuje pravopis jako soubor pravidel o zaznamenávání spisovných jazykových projevů písmem a gramatiku jako stavbu jazyka řízenou pravidly o obměnách slov a o spojování slov ve věty. Jednoduchý systém pro kontrolu pravopisu lze vytvořit nad slovníkem správných slov. Oproti tomu tvorba systému pro kontrolu gramatické správnosti, je úkol výrazně obtížnější, a to jak z hlediska lingvistického, tak softwarového [24]. Tato práce se dále zabývá výhradně systémy pro kontrolu pravopisu.

Systémy pro kontrolu pravopisu se zaměřují na pravopisné chyby. Pravopisné chyby můžeme dělit na překlepy a fonetické chyby [15]. Překlepy vznikají tak, že byla omylem stisknuta jiná klávesa, byly stisknuty dvě klávesy, ve špatném pořadí a podobně, zatímco u fonetických chyb se slovo vyslovuje stejně jako zamýšlené slovo, ale písmena jsou ve slově chybně [15]. Obtížně odhalitelné jsou zejména fonetické chyby, při kterých je slovo zaměněno za homofon. Homofony jsou slova, která se v mluvené podobě shodují, ale v psané se liší. Například *být* a *bít*, *led* a *let* nebo anglická *two* a *too*. Analýza slov s chybami, provedená [4], odhalila, že přes 80 % pravopisných chyb spadá do následujících kategorií:

- chybějící písmeno,
- písmeno navíc,
- chybné písmeno (záměna),
- prohozená dvě sousední písmena.

K chybám může být přistupováno dvěma způsoby. Mohou být pouze detekovány nebo rovnou i opravovány. Oprava může být provedena automaticky nebo může být nabídnuta, případně může být nabídnuto více alternativ. Automatická oprava může způsobit, že správné slovo, které však není obsaženo

ve slovníku, bude nahrazeno jiným slovem. Tomuto fenoménu se říká *Cupertino effect* [36]. Paradoxně tak systém, který má před chybami chránit, zavádí nový druh chyb.

Systémy pro kontrolu pravopisu můžeme rozdělit podle jejich přístupu ke kontrole slov na kontextové a bezkontextové. Kontextová kontrola pravopisu bere v úvahu okolní slova, zatímco bezkontextová kontrola na slovo pohlíží izolovaně. Pouze kontextová kontrola může odhalit chyby, při kterých je slovo zaměněno za homofon.

Systémy pro kontrolu pravopisu můžeme také rozdělit podle principu jejich fungování na:

- *Slovníkové* – jejich jádro tvoří slovník správných slov. U jazyků s bohatou morfologií však může být paměťově velmi náročné nebo i nemožné kompletní slovník sestavit (viz podkapitola 2.1). Součástí systému tak mohou být i dynamická pravidla definující tvorbu či skládání slov.

- *Statistické* – jejich rozhodování se opírá o pravděpodobnost, s jakou se může dané slovo vyskytovat v kombinaci s jiným slovem (bi-gramy) nebo jinými slovy (tri-gramy, n-gramy).

- *Využívající strojové učení* – využívají mechanismy strojového učení, jako například neuronové sítě. Podobají se systémům založeným na statistice.

Cílem této práce je vytvořit systém pro bezkontextovou kontrolu pravopisu založený na slovníku, který kromě detekce chyby bude také umět nabídnout několik správných alternativ. Další podkapitoly proto budou zaměřeny na metody reprezentace slovníků a napovídání slov. Ještě předtím však dojde ke krátké odbočce na téma morfologie jazyka.

2.1 Vliv morfologie jazyka na kontrolu pravopisu

Morfologie jako lingvistická disciplína se zaměřuje na tvary slov, procesy slovotvorby a jejich výsledné tvary a podoby jejich celků či částí. Obecně se morfologií míní ta oblast gramatiky, která studuje povahu a chování morfémů. [29] Morfém je minimální (abstraktní a systémovou) významovou a/nebo gramatickou jednotku [29]. Může jím být afix nebo kořen slova (někdy také radix). Afix je morfém, který se připojuje ke kořeni slova. Upřesňují a dotvářejí význam lexémů a jsou pouze vázané, mohou se tedy vyskytovat

pouze ve spojení s kořenem [29]. Podle toho, jakým způsobem se připojují, rozlišuje:

- *Prefix* – předpona, umístěna před kořenem slova.
- *Sufix* – přípona, následuje za kořenem.
- *Cirkumfix* – kombinace prefixu a sufixu.
- *Infix* – vpona, vstupuje do kořene.
- *Interfix* – vkládá se mezi kořeny dvou slov (při skládání slov).

Podle morfologie můžeme dělit jazyky na: [6]

- *Syntetické* – charakterizuje je užívání afixů. Vyznačují se rozvinutým ohýbáním slov (skloňování, časování, stupňování apod.). Příkladem může být čeština, němčina a španělština.
- *Analytické* – charakterizuje je mnohoznačnost slov. Gramatická funkce je vyjádřena dodávanými pomocnými slovy, např. předložkami. Příkladem může být angličtina a švédština.

Z podstaty syntetických jazyků vyplývá, že obsahují zásadně více slov než jazyky analytické.

Jedním z forem slovotvorby je skládání slov. Se skládáním slov se lze setkat i v češtině, například u slova *vodotěsný*. Větší problém, z pohledu implementace systémů pro kontrolu pravopisu, představuje němčina, která má velmi volné skládání slov, které je navíc hojně využíváno. Extrémnějším příkladem může být slovo *Telekommunikationskundenschutzverordnung* (nařízení ochrany zákazníků telekomunikací). Takovýchto složených slov lze vytvořit téměř neomezený počet.

Morfologie jazyka má tedy zásadní vliv na složitost kontroly pravopisu. U analytických jazyků může k dobrým výsledkům stačit slovník o několika stovkách tisíc slov, zatímco u syntetických jazyků by slov musely být miliony, nebo dokonce nemusí být možné takový slovník rozumně sestavit. Některé systémy pro kontrolu pravopisu se proto vydaly cestou slovníku se slovy v základním tvaru a seznamu afixových pravidel, které vygenerují další varianty. Tento způsob využívá například Aspell, MySpell a Hunspell. Hunspell dále nabízí mechanismus pro volné skládání slov, který si dokáže poradit například s prezentovaným skládáním slov v německém jazyce.

2.2 Metody reprezentace slovníků

Tato podkapitola se bude zabývat datovými strukturami obvykle používanými pro reprezentaci slovníků či velkých seznamů slov:

- hashovací tabulka,
- binární vyhledávací strom,
- trie,
- ternární vyhledávací strom.

Datová struktura má rozhodující vliv i na možnosti napovídání podobných slov, proto se bude zvažovat i tento aspekt. Zvažovány budou také možnosti asociace příznaků[1], ke slovům uloženým ve slovníku, v dané datové struktuře.

2.2.1 Hashovací tabulka

Princip hashovacích tabulek je všeobecně znám. Jsou nedílnou součástí standardních knihoven všech běžných programovacích jazyků. Struktury založené na hashovacích tabulkách bývají implementovány za pomoci polí nebo spojových seznamů. Ve standardních knihovnách je nalezneme ve formě množin (*hash set*) nebo slovníků/map (*hash map*). Hash set by mohl být použit v případě, kdy není potřeba ke slovům asociovat příznaky, v opačném případě lze použít hash map. Pro reprezentaci slovníku se hodí zejména díky své schopnosti přístupu k prvkům v konstantní rychlosti.

Nevýhodou je, že tato struktura nenabízí možnost, jak ji využít při hledání podobných slov. Z principu hashovací funkce jsou slova pseudonáhodně roztroušena, bez ohledu na to, že se liší třeba jen v posledním znaku[2]. Další nevýhodou, která vychází ze skutečnosti, že jsou záznamy ukládány samostatně, je vyšší paměťová náročnost. U slovníku malého rozsahu tyto nevýhody nebudou patrné, zřejmě by se tak jednalo o první volbu. Ale i v opačném případě se může jednat o dobrou startovací metodu, před implementací komplexnějších struktur a pro porovnání.

[1] Příznaky se mohou týkat například mechanismu skládání slov nebo napovídání.

[2] Existují hashovací funkce, které jsou založeny na opačném principu, kdy jsou podobným položkám přiřazeny blízké nebo stejné hodnoty hashovací funkce[32]. Nicméně to by byl velmi neobvyklý způsob řešení tohoto problému.

2.2.2 Binární vyhledávací strom

Binární vyhledávací strom (BVS) je založen na binárním stromu, tedy každý uzel má nejvýše dva následníky. Uzly mají přiřazené hodnoty. Při vyhledávání je s hledanou hodnotou porovnávána hodnota v uzlu. Pokud je hledaná hodnota menší, pokračuje se levým podstromem, v opačném případě pravým podstromem. Půlení intervalu vede na logaritmickou časovou složitost.

BVS je možné použít pro reprezentaci slovníků, protože řetězce lze porovnávat. Počet porovnávaných písmen roste s tím, jak se vyhledávání blíží cíli [30]. Podobně jako hashovací tabulka ale nedává možnost, jak topologii struktury využít při hledání podobných slov. Požadavek na uložení příznaků může být snadno splněn, a to jejich uložením do vrcholu, vedle slova. Ve standardních knihovnách BVS nalezneme ve formě množin (tree set) nebo slovníků/map (tree map). Tree set by mohl být použit v případě, kdy není potřeba ke slovům asociovat příznaky, v opačném případě lze použít tree map.

BVS má nevýhodu, že při jeho sestavování může dojít k nevyvážení, a tím ke snížení výkonu. V extrémním případě může strom degenerovat do podoby spojového seznamu, s vyhledáváním s lineární časovou složitostí. To může nastat, pokud jsou prvky seřazené a vkládají se postupně. Podle [11] už i menší počet seřazených prvků na začátku datasetu může dramaticky ovlivnit výkon struktury. Z tohoto důvodu existují například *red-black tree* nebo *AVL tree*. Tyto stromy se udržují vyvážené za cenu pomalejšího přidávání prvků. Vyvážení však neřeší problém, kdy jsou často přistupované položky vnořené hluboko ve stromě. To je nutné vzít v úvahu, protože v přirozených jazycích malé procento slov tvoří většinu textu[3]. Tento problém se snaží řešit *splay tree* [26], který provádí vyvažování a navíc udržuje často přistupované uzly blíže u kořene. Splay tree však vyžaduje o něco více paměti, protože jeho efektivní implementace vyžaduje, aby každý uzel měl odkaz na svého předka [30].

Výsledky [11] ukazují, že v průměrném případě je BVS efektivnější než splay tree. [33] potvrzuje, že jsou obecně zhruba o 25 % pomalejší než BVS, ale nachází novou heuristiku pro vyvažování, která má o 3 % lepší výkon oproti BVS a o BVS říká: „*the possibility of worstcase performance means that it should not be used*“.

[3]V anglickém jazyce 10 slov tvoří zhruba 25 % veškerého textu, 100 slov 50 %, 50 000 slov 95 % a pro zbylých 5 % je potřeba slovník o více než milionu slov. [28]

2.2.3 Trie

Název trie vznikl zkřížením slov *tree* (strom) a *retrieval* (vyhledávání) [14]. Alternativní název je *prefix tree*. Jedná se o strukturu určenou speciálně pro ukládání řetězců. Každý uzel v trii představuje řetězec, který je definován posloupností hran, které do něj vedou. Kořenu odpovídá prázdný řetězec, druhá úroveň uzlů nese jednoznakové řetězce, třetí dvouznakové atd. Hrana tedy nese následující písmeno. Hrany vedoucí z uzlu jsou seřazené a unikátní. Uzel ve kterém končí slovo je označen příznakem. Spolu s koncem slova mohou být případně v uzlu uvedeny další příznaky, které se k danému slovu váží.

Existuje také upravená varianta, která se nazývá komprimovaná trie (někdy také *radix tree*). Je založena na myšlence, že trie často obsahují dlouhé cesty bez větvení. Tyto cesty mohou být komprimovány nahrazením jedinou hranou, která bude místo jednoho písmene nést celý řetězec. Musí však platit předpoklad, že žádné dvě hrany vycházející z jednoho vrcholu nemají společný prefix. Kompresí dojde ke snížení počtu uzlů, a tím i ke zrychlení vyhledávání a snížení paměťové náročnosti.

Podle [25] reálná je paměťová efektivita a výkon trie v porovnání s hashovací tabulkou řádově podobná. [31] se podařilo nahrazením hashovací tabulky komprimovanou trií dosáhnout výrazného zlepšení. Naopak v testech [11] se komprimovaná trie neosvědčila a hashovací tabulky dopadly lépe. Na základě rozdílných datasetů v uvedených článcích lze usoudit, že efektivita té či oné varianty zřejmě závisí na velikosti slovníku a charakteru uložených řetězců (zejména délce, podobnosti, apod.). [25] však poukazuje na to, že trie dává prostor pro další optimalizace a její struktura může být využita pro tvorbu efektivnějších algoritmů pro hledání podobných slov.

2.2.4 Ternární vyhledávací strom

Ternární vyhledávací strom (TVS) je méně známá varianta vyhledávacího stromu. Je založen na ternárním stromu, tedy každý uzel má nejvýše tři následníky. Na rozdíl od binárního vyhledávacího stromu již není obecný, ale je specializovaný na řetězce. Nepracuje se slovy jako s celkem (obecný klíč), ale pracuje s jednotlivými znaky. Tím se zase podobá trii. [1] říká: „*Ternary search tries combine the best of two worlds: the low overhead of binary search trees (in terms of space and running time) and the character-based eficiency of tries*“.

Při vyhledávání je s aktuálním znakem hledaného řetězce (začíná se od prvního) porovnáván znak v uzlu. Pokud je hodnota menší, tak se pokračuje

(se stejným znakem) levým podstromem, v opačném případě pravým podstromem. Pokud se znaky rovnají, tak se pokračuje s dalším znakem v hledaném řetězci, a to prostředním podstromem. Uzel ve kterém končí slovo je označen příznakem. Slovo může končit pouze v uzlu, který je prostředním potomkem. Jinými slovy: pokud algoritmus skončil vyčerpáním hledaného řetězce v uzlu, který je prostředním potomkem, a zároveň je na tomto uzlu nastaven příznak konce slova, znamená to, že slovo bylo nalezeno. Spolu s koncem slova mohou být případně v uzlu uvedeny další příznaky, které se k danému slovu váží.

Topologie TVS může být, podobně jako u trie, využita pro vyhledání všech slov začínající na daný prefix. To dává určité možnosti při hledání podobných slov, byť trie se v tomto ohledu zdá být vhodnější.

2.2.5 Shrnutí metod reprezentace slovníků

Každá z uvedených struktur má své výhody a nevýhody. Reálně se používají všechny. Velkou roli bude hrát to, jak efektivně se podaří danou strukturu implementovat na cílové platformě a jestli bude efektivnější než nativní *HashMap* ze standardní knihovny. Závěry citovaných článků byly vyneseny nad implementacemi v jazycích, jako je C. Ty se od JavaScriptu značně liší typovým systémem a správou paměti. Omezeními JavaScriptu se bude zabývat kapitola 4.

2.3 Metody napovídání slov

V předešlé podkapitole byly u jednotlivých struktur zvažovány možnosti v souvislosti s napovídáním podobných slov. Nyní bude toto téma probráno více do hloubky. Budou popsány tři metody pro hledání podobných slov.

2.3.1 Použití metriky

Metoda je založena na použití metriky, která číselně určí, jak jsou si dvě slova podobná. Vstupní slovo je za pomoci metriky porovnáno se všemi slovy ve slovníku a je vybráno n nejpodobnějších slov. Tuto metodu pro hledání podobných slov lze použít u struktur, které umožňují získání všech v ní uložených prvků. To znamená všechny struktury z podkapitoly 2.2.

Jako metriku lze použít nejdelší společný podřetězec, Hammingovu vzdálenost, Levenshteinovu vzdálenost, Damerau-Levenshteinovu vzdálenost a další. Nejčastěji využívaná je zřejmě Levenshteinova vzdálenost. Levenshteinova vzdálenost [13] určuje minimální počet operací, jejichž aplikací se

převede jedno slovo na druhé. Možné operace jsou nahrazení, vložení a odebrání znaku. Damerau-Levenshteinova vzdálenost má navíc operaci prohození dvou sousedních znaků.

Nevýhodou této metody je, že doba výpočtu lineárně roste s počtem slov ve slovníku. U slovníku s milionem slov se metrika bude počítat milionkrát. Výpočet lze urychlit pouze ukončením výpočtu metriky v případě, kdy je jasné, že se slovo nedostane mezi n nejpodobnějších slov (nepřekoná nejhorší z aktuálních n nejpodobnějších slov) nebo když je jasné, že hodnota přesáhne určitou mez[4].

2.3.2 Použití metriky a prořezávání

Předchozí metoda může být významně optimalizována při použití Levenshteinovy vzdálenosti jako metriky a trie jako struktury pro reprezentaci slovníku. Algoritmus pro výpočet Levenshteinovy vzdálenosti lze implementovat za použití dynamického programování. Jedná se o známý algoritmus Wagner–Fischer[5]. Ten umožňuje vypočítat Levenshteinovu vzdálenost po jednotlivých znacích. Pokud například máme několik slov se stejným prefixem, můžeme nad ním provést výpočet a uložit si mezivýsledek, a při zpracování jednotlivých slov mezivýsledek využít. Nedochází tak k opakování výpočtu pro společnou část slov. Toho lze naplno využít v trii, jelikož se jedná o prefixový strom – stačí ji projít do hloubky. Další výhodou tohoto přístupu je, že když mezivýsledek Levenshteinovy vzdálenosti napovídá, že výsledek bude vždy větší, než určitá mez nebo horší, než pro dosud nalezená podobná slova, lze zanechat procházení celého podstromu.

Urychlení je tedy dosaženo kombinací dvou důvodů:

- pro jednu předponu se Levenshteinova vzdálenost počítá pouze jednou,
- prořezávání – rozsáhlé části slovníku jsou vynechány.

2.3.3 Generování slov

Místo procházení uložených slov a jejich porovnávání se slovem s překlepem, je také možné jít na to zcela odlišně a ke slovu s překlepem vygenerovat slova, zpětnou aplikací možných chyb (chybějící písmeno, písmeno navíc, chybné písmeno nebo prohozená dvě sousední písmena) a tato vygenerovaná slova

[4]Levenshteinova vzdálenost se obvykle omezuje na 2 nebo 3. Obecně nemá smysl napovídat příliš odlišná slova.

[5]`https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm`

hledat ve slovníku. Tento způsob prezentuje Peter Norvig ve svém článku [22].

Tuto metodu pro hledání podobných slov lze použít u všech typů struktur. Má tu výhodu, že sama o sobě není závislá na velikosti slovníku. Nicméně na velikosti slovníku může být závislé ověření, jestli se slovo nachází ve slovníku. Je také výrazně méně efektivnější u jazyků s větší abecedou nebo pokud slovník obsahuje i nepísmenné znaky. Celkově výhodnost použití této metody vůči ostatním závisí na velikosti slovníku a abecedy. Hlavní nevýhoda však je její polynomiální závislost na délce slova – může výkonově selhávat u nadprůměrně dlouhých slov.

2.3.4 Shrnutí metod pro napovídání slov

Metoda využívající metriku bude efektivnější u menších slovníků, zatímco metoda využívající generování bude mít navrch u větších slovníků, ale pouze v případě, kdy abeceda nebude příliš velká a slovo extrémně dlouhé. V případě volby trie jako struktury pro reprezentaci slovníku bude zřejmě vhodnější použít metodu využívající metriku a prořezávání.

Existují také pokročilejší metody, které jsou založené na tvorbě indexu. Ty fungují na principu, že ze slovníku jsou vygenerována chybná slova, až do určené editační vzdálenosti, a tato slova jsou následně uložena do indexu, spolu s odkazy na původní správná slova ve slovníku. [3][7] Implementace by byla náročná, zejména v souvislosti s cílovou platformou, také by se musela řešit distribuce indexu apod. Tento přístup by však mohl být zvážen, pokud by výše uvedené metody selhaly.

3 Systém pro kontrolu pravopisu Hunspell

Předchozí kapitola poskytla úvod do problematiky systémů pro kontrolu pravopisu, představila vhodné struktury pro reprezentaci slovníků a metody napovídání slov. Cílem této kapitoly je představit systém pro kontrolu pravopisu s názvem Hunspell. Součástí bude i popis formátu jeho slovníků, protože výsledný systém je má podporovat.

Hunspell[1] je systém pro kontrolu pravopisu vyvíjený v C++. Je založený na myšlence slovníku rozděleného na seznam slov v základním tvaru a seznam pravidel, které jsou na ně aplikována (přidávají jim předpony, přípony, skládají je dohromady...).

Vychází ze systému MySpell, který byl součástí OpenOffice, a který posléze nahradil. Vznikl původně pro maďarštinu, jakožto pro jazyk s komplexní strukturou slov, se kterou si původní systém nedokázal poradit. Postupně získal popularitu a našel si cestu do mnoha významných programů. Hunspell zachovává kompatibilitu se slovníky systému MySpell a má podobný formát slovníků – změny ve formátu slovníků jsou spíše aditivní, jako přidání nových příkazů, možností, apod. Navíc zavádí například podporu pro UTF-8, komplexnější skládání slov nebo syntaxi pro zápis morfologických metadat [20].

Hunspell zahrnuje knihovnu a nástroj pro příkazovou řádku. Využívají ho například macOS, OpenOffice, LibreOffice, Google Chrome, Mozilla Firefox, Mozilla Thunderbird, IntelliJ IDEA [12] a další. [20] Byť některé z nich zřejmě nepoužívají originální knihovnu, ale pouze do určité míry podporují slovníky Hunspellu. Rozšířenost Hunspellu má za následek dostupnost slovníků pro velké množství jazyků.

Dále budou popsány obecné mechanismy, na kterých je Hunspell založen. Teprve potom bude popsán formát slovníků.

3.1 Obecné mechanismy

V této podkapitole budou popsány následující mechanismy, kterými Hunspell disponuje:

- pravidla,

[1] `https://hunspell.github.io`

- nakládání s velikostí písmen,
- skládání slov,
- napovídání podobných slov,
- podpora pro ligatury.

Pravidla Jak již bylo zmíněno v úvodu této kapitoly, pravidla jsou v Hunspellu klíčovým mechanismem. Pravidla se přiřazují ke slovům ve slovníku. Některá pravidla vytvářejí nová slova přidáním afixů, jiná slovům nastaví příznaky (např. v souvislosti se skládáním slov nebo napovídání podobných slov, viz dále). Pravidlo také může být použito v definici jiného afixového pravidla. Tímto způsobem lze generovat slova s vícenásobnými příponami (např. *clue – clueless – cluelessness*). V případě pravidel nastavující příznak, je pak tento příznak nastaven všem z něho odvozeným slovům.

Nakládání s velikostí písmen Pokud slovník obsahuje „slovo“, potom i „Slovo“ (*capitalized form*) a „SLOVO“ (*uppercased form*) jsou považovány za správné. Ostatní kombinace velikostí písmen jsou považovány za nesprávné. Analogicky, pokud slovník obsahuje „Slovo“, potom i „SLOVO“ je správně, ale již ne „slovo“ nebo jiné. Pokud obsahuje „SLOVO“, potom pouze tato forma je správná. Uvedené chování se vztahuje i na slova vygenerovaná aplikací afixových pravidel.

Nechtěné varianty lze zneplatnit použitím pravidla *FORBIDDENWORD* nebo uppercased a capitalized varianty zcela zakázat pravidlem *KEEPCASE*. Tato pravidla budou podrobněji vysvětlena dále.

Skládání slov Hunspell obsahuje dva nezávislé mechanismy skládání slov:

- *Compound rules* – Založeno na principu tvorby vzorů, podobných regulárním výrazům. Jsou pomocí nich implementovány například řadové číslovky (např. *1001st*, *2054th*) v anglickém slovníku LibreOffice[2].
- *Compound flags* – Založeno na principu označení slov, které se mohou vyskytovat na začátku, uprostřed nebo na konci složeného slova. Všechny kombinace, splňují-li případná další omezení, jsou poté validní. Dále platí, že prefixy jsou povoleny pouze na začátku a sufixy pouze na konci složených slov.

[2] `https://github.com/LibreOffice/dictionaries`

Napovídání podobných slov Systém jakým Hunspell napovídá podobná slova není příliš dobře zdokumentován. Na základě dostupných informací a zběžného prozkoumání zdrojových kódů se jeho činnost dá zjednodušit do následujících kroků:

1. Hledá slova, která se liší pouze velikostí znaků.
2. Pokud ji slovník obsahuje, využívá nahrazovací tabulku častých chyb. Její součástí mohou být i jednoduché regulární výrazy.
3. Pokud slovník obsahuje tabulku pro fonetickou transkripci, zkusí ji využít.
4. Využívá informace o rozložení kláves, pokud je slovník obsahuje. Takto napovídá slova, která se liší o jeden znak.
5. Hledá slova s minimální editační vzdáleností.

Nemusí přitom dojít k vykonání všech kroků. Při nalezení dostatečného počtu podobných slov se hledání přeruší.

Do seznamu podobných slov se nedostanou slova označená pravidlem *NOSUGGEST*, *FORBIDDENWORD* nebo *NEEDAFFIX* (viz dále).

Podpora pro ligatury V typografii je ligatura spojení dvojice písmen. Historicky, již při knihtisku, tento pojem znamenal spojení písmen v jeden grafický objekt, který měl nějakým způsobem upravenou podobu [2]. V počítačové terminologii se pak jedná o nahrazení dvojice znaků jedním [2]. Příkladem může být například æ (ae), Æ (AE), œ (oe), Œ (OE) nebo fi (fi). Důsledek existence ligatur je, že může existovat více ekvivalentních vyjádření stejného slova. Hunspell tento problém řeší zavedením vstupní a výstupní konverzní tabulky.

3.2 Formát slovníků

Slovníky systému Hunspell tvoří dvojice textových souborů, které obvykle mají stejný název, např. `en_GB.dic` a `en_GB.aff`. Mají následující význam:

- Soubor s příponou *dic* obsahuje slovník (seznam slov) spolu s odkazy na pravidla, která se mají na vybraná slova aplikovat.
- Soubor s příponou *aff* pak obsahuje definovaná pravidla a obecná nastavení kontroly pravopisu a napovídání.

Jediným zdrojem o formátu slovníků jsou manuálové stránky (man pages) [21], které jsou zároveň hlavním zdrojem této podkapitoly.

3.2.1 Formát souboru se slovníkem

Soubor s příponou *dic* obsahuje seznam slov, jedno slovo na jednu řádku. Na první řádce je uvedeno číslo značící počet slov [3]. Každé slovo může být volitelně následováno lomítkem (/) a vektorem pravidel. Pokud má být lomítko součástí slova, umístí se před něj zpětné lomítko (\). Za slovem a pravidly mohou dále volitelně být morfologická metadata, oddělená mezerou nebo tabulátorem. Nástroje pro morfologickou analýzu[4] však nejsou pro funkci systému pro kontrolu pravopisu relevantní, proto se jimi ani nebudou další části této práce zabývat. Výpis 3.1 obsahuje příklad slovníku, který obsahuje slovo „foo“, slovo „bar“, na které jsou aplikována pravidla A a B a dvě slova s příklady morfologických metadat.

Výpis 3.1: Příklad obsahu souboru se slovníkem.

```
4
foo
bar/AB
word po:noun
mice st:mouse is:plural
```

3.2.2 Formát souboru s pravidly

Soubor s příponou *aff* obsahuje definovaná pravidla a obecná nastavení. Ukázka je uvedena ve výpisu 3.2. Vše je definováno pomocí příkazů. Použití příkazů a jejich pořadí je volitelné. Žádný z příkazů není povinný. Prázdné řádky mohou být vloženy kvůli čitelnosti. Podporovány jsou také řádkové komentáře, uvádí je křížek (#). Příkaz vždy začíná názvem a je následovaný parametry, které jsou odděleny mezerami nebo tabulátory. Příkaz může být jednořádkový (např. *SET* a *KEEPCASE*) nebo víceřádkový (např. *REP* a *SFX*). U víceřádkových příkazů je vždy součástí prvního příkazu parametr s počtem řádek, které jej následují. Další řádky opět začínají názvem příkazu.

Výpis 3.2: Příklad obsahu souboru s pravidly.

```
SET UTF-8

KEEPCASE A

REP 2
```

[3] Dle dokumentace však číslo není závazné, slouží pouze optimalizacím [21].

[4] Morfologická analýza je proces, při kterém se slovům zpětnou derivací morfémů, ze kterých se skládají, přiřazují lexikální nebo gramatické kategorie (například slovní druhy, pád, číslo, rod...). Morfologií se více zabývala kapitola 2.1.

```
REP shun$ tion
REP ph f

SFX B Y 2
SFX B 0 ed [^y]
SFX B y ied y
```

Hunspell obsahuje mnoho příkazů. Některé z nich však mají pouze omezené možnosti použití a téměř se nepoužívají, jsou nedostatečně zdokumentovány nebo nejsou relevantní pro potřeby této práce. Uvedeny budou ty nejdůležitější příkazy tak, aby byly zmíněny všechny klíčové mechanismy, které budou zapotřebí při tvorbě parseru a systému pro kontrolu pravopisu, který je zvládne realizovat. Příkazy můžeme rozdělit podle svého významu do následujících kategorií:

- příkazy obecného charakteru,
- příkazy pro afixová pravidla,
- příkazy pro skládání slov,
- příkazy pro napovídání slov.

Příkazy obecného charakteru

SET <kódování> Nastaví kódování souboru s pravidly a souboru se slovníkem. Typicky se jedná o první příkaz. Možné hodnoty: UTF-8, ISO8859-1 - ISO8859-10, ISO8859-13 - ISO8859-15, KOI8-R, KOI8-U, microsoft-cp1251, ISCII-DEVANAGARI.

FLAG <hodnota> Nastaví způsob pojmenování pravidel. Možné hodnoty:

- `UTF-8` – jeden znak ze znakové sady UTF-8.
- `long` – dva znaky z rozšířené ASCII znakové sady (8 bitů).
- `num` – číslo od 1 do 65000, při použití se oddělují čárkou.

Výchozí způsob pojmenování je jeden znak z rozšířené ASCII znakové sady (8 bitů).

FORBIDDENWORD <flag> Definuje pravidlo, kterým lze označit nevalidní slova. Slouží zejména k zachycení výjimek, které vznikly použitím některého z mechanismů generování slov.

KEEPCASE <flag> Definuje pravidlo, které u daného slova zakáže *uppercased* a *capitalized* varianty.

CHECKSHARPS Povolí podporu pro německé ostré S (ß). Při jeho převodu na velké písmeno pak bude nahrazeno za SS.

IGNORE <seznam znaků bez mezer> Nastaví znaky, které mají být u kontrolovaných slov ignorovány. To je zapotřebí například kvůli existenci nepovinných diakritických znaků u hebrejštiny nebo arabštiny.

ICONV <počet definic>

ICONV <pattern> <náhrada> Definuje vstupní konverzní tabulku. Slouží pro normalizaci v případě existence více znaků se stejným významem (Unicode ligatury a jejich vyjádření standardními znaky, různé alternativní varianty diakritických znamének...). Vstupní tabulka se aplikuje na slovo ke kontrole nebo na slovo ke kterému se hledají podobná slova ještě před všemi ostatními činnostmi.

OCONV <počet definic>

OCONV <pattern> <náhrada> Definuje výstupní konverzní tabulku. Aplikuje se na výsledný seznam podobných slov.

Příkazy pro afixová pravidla

PFX <flag> <povolit kombinace - Y/N> <počet definic>

PFX <flag> <část k odříznutí> <prefix> <podmínka> [<morf. data>...] Definuje prefixové pravidlo. Jedná se o víceřádkový příkaz s mírně odlišnou strukturou. Na prvním řádku se kromě obvyklého počtu definic uvádí i zda jsou povoleny kombinace se sufixovými pravidly (kartézský součin). Každý další řádek potom definuje jedno dílčí pravidlo. Dílčí pravidlo se skládá z řetězce, který má být odříznut na začátku slova, z prefixu, který se umístí místo něj a podmínky, jak musí slovo začínat, za které je dané dílčí pravidlo aplikovatelné. Prázdné řetězce se značí nulou. Prefix může být, stejně jako slova ve slovníku, následován lomítkem (/) a vektorem pravidel. Tato pravidla budou aplikována na nově vytvořené slovo. Podmínka je výraz podobný regulárnímu výrazu, který ze standardních mechanismů podporuje pouze tečku jako libovolný znak, množiny znaků zapisující se do hranatých závorek a jejich doplňky. V případě žádné podmínky (vždy splněno) se používá tečka.

SFX <flag> <povolit kombinace - Y/N> <počet definic>

SFX <flag> <část k odříznutí> <prefix> <podmínka> [<morf. data>...] Definuje sufixové pravidlo. Analogické prefixovým pravidům.

AF <počet definic>

AF <vektor pravidel> Vytvoří alias pro vektor pravidel. Vytvořený alias je poté adresován pořadovým číslem (od 1). Vektor pravidel by mohl být například `AB` z výpisu 3.1, potom by šlo zapsat `bar/1` místo `bar/AB`.

FULLSTRIP Povolí odříznutí více znaků při aplikaci sufixového pravidla. Standardně lze odříznout maximálně jeden znak.

NEEDAFFIX <flag> Slovo označené tímto pravidlem nemají smysl sama o sobě, ale jsou validní pouze s nějakým afixem.

Příkazy pro skládání slov (*compound rules*)

COMPOUNDRULE <počet definic>

COMPOUNDRULE <pattern> Definuje pravidlo pro skládání slov formou předpisu podobného regulárnímu výrazu. Výraz se skládá z názvů pravidel, řídících znaků a závorek. Řídící znaky jsou pouze hvězdička (*), libovolný počet výskytů předcházejícího výrazu, a otazník (?), volitelný výskyt předcházejícího výrazu. V případě adresování dvojicí znaků nebo číslem (příkaz *FLAG*) se používají závorky. Pravidla se nemusí nikde zvlášť definovat, pouze se použijí uvnitř výrazu a označí se jimi vybraná slova.

Příkazy pro skládání slov (*compound flags*)

COMPOUNDMIN <číslo> Nastavuje minimální délku slova, které může být součástí složeného slova. Výchozí hodnota je 3.

COMPOUNDBEGIN <flag> Značí slova, která mohou být na začátku složeného slova (pokud jsou delší než *COMPOUNDMIN*).

COMPOUNDMIDDLE <flag> Značí slova, která mohou být uprostřed složeného slova (pokud jsou delší než *COMPOUNDMIN*).

COMPOUNDLAST <flag> nebo COMPOUNDEND <flag> Značí slova, která mohou být na konci složeného slova (pokud jsou delší než *COMPOUNDMIN*).

COMPOUNDFLAG <flag> Značí slova,
která mohou být součástí složeného slova (pokud jsou delší než *COMPOUNDMIN*). Ekvivalentní kombinaci *COMPOUNDBEGIN*, *COMPOUNDMIDDLE* a *COMPOUNDLAST*.

ONLYINCOMPOUND <flag> Značí slova, která sice mohou být ve složeném slově, ale nemají smysl sama o sobě.

CHECKCOMPOUNDCASE Zakazuje velká písmena na hranicích slov uvnitř složeného slova. Funguje tedy jako filtr nesprávných slov.

Příkazy pro napovídání slov

TRY <seznam znaků bez mezer> Definuje znaky, které se mají zkoušet vkládat do slov. Case sensitive.

KEY <skupiny znaků bez mezer> Definuje skupiny sousedních znaků, které se mají zkoušet navzájem nahrazovat. Skupiny jsou odděleny svislou čárou (|). Slouží k detekci překlepů s ohledem na rozložení klávesnice. Příklad:

```
KEY qwertyuiop|asdfghjkl|zxcvbnm
```

REP <počet definic>

REP <pattern> <náhrada> Definuje nahrazovací tabulku, které slouží k detekci častých chyb. Ve vzoru se může nacházet stříška (^), značící začátek slova, a dolar ($), značící jeho konec. V nahrazujícím řetězci lze použít podtržítko (_) jako mezeru. Příklad:

```
REP 1
REP ^alot$ a_lot
```

MAP <počet definic>

MAP <seznam znaků bez mezer> Definuje skupiny znaků či řetězců, které lze považovat za podobné, a tak je u nich zvýšené riziko záměny. Příkladem mohou být Unicode ligatury a jejich vyjádření standardními znaky, ostré S (ß) z německé abecedy nebo stejné znaky s diakritikou a bez ní. Víceznakové řetězce se ohraničují závorkami. Příklad:

```
MAP 3
MAP uúü
MAP ööo
MAP ß(ss)
```

NOSUGGEST <flag> Značí slova, která jsou akceptovatelná kontrolou pravopisu, ale nemají se objevit mezi napovězenými slovy. Například pro vulgární nebo jinak nevhodná slova.

4 Potřebné JavaScriptové technologie

V předchozích kapitolách byly uvedeny systémy pro kontrolu pravopisu a představen systém s názvem Hunspell. Cílem této kapitoly je prozkoumat možnosti, jaké poskytuje JavaScript v souvislosti s implementací systému pro kontrolu pravopisu a s jakými omezeními je potřeba počítat. Zkoumány budou dvě oblasti:

- způsob uložení dat,
- paralelní zpracování.

4.1 Způsob uložení dat

V kapitole 2.2 byly popsány struktury pro reprezentaci slovníků. Tato podkapitola se zabývá možnostmi JavaScriptu v tomto ohledu. Probrány budou následující možnosti pro uložení dat:

- datové typy a objekty,
- `ArrayBuffer` a typovaná pole,
- Web Storage API,
- Indexed Database API.

4.1.1 Datové typy

ECMAScript 2015 (specifikace JavaScriptu) obsahuje následující primitivní datové typy: [5]

- `Boolean`,
- `Null`,
- `Undefined`,
- `Number`,
- `BigInt`,

- `String`,
- `Symbol`.

Primitivní datové typy se vyznačují tím, že jsou neměnné (*immutable*). Jediný klasický číselný typ je `Number` – 64 bitové číslo s plovoucí desetinnou čárkou.

Dále ECMAScript definuje datový typ `Object`. Vzhledem k tomu, že objekty jsou v JavaScriptu asociativní pole, zabírají v paměti více místa, než například struktury v jazyce C. Celková velikost objektu však závisí, kromě jeho obsahu, na běhovém prostředí. Specifikace se o ni nezmiňuje. Záleží například i na zarovnání dat v paměti a dalších skutečnostech [34][35].

4.1.2 ArrayBuffer a typovaná pole

JavaScript umožňuje vytvořit pole bajtů prostřednictvím typu `ArrayBuffer`. S jeho obsahem však přímo manipulovat nelze, to lze pouze pomocí typovaných polí. Typovaná pole obalují `ArrayBuffer` a umožňují přistupovat k prvkům stejně jako u klasických polí. Tabulka 4.1 obsahuje seznam typovaných polí specifikovaných v ECMAScript 2015. Jedná se již o starší rozhraní, a tak podpora prohlížečů nepředstavuje riziko.

Tabulka 4.1: Typovaná pole v ECMAScript 2015 [5].

Typ	**Velikost prvku v bajtech**	**Ekvivalentní typ v C**
`Int8Array`	1	*char*
`Uint8Array`	1	*unsigned char*
`Uint8ClampedArray`	1	*unsigned char*
`Int16Array`	2	*short*
`Uint16Array`	2	*unsigned short*
`Int32Array`	4	*int*
`Uint32Array`	4	*unsigned int*
`Float32Array`	4	*float*
`Float64Array`	8	*double*

4.1.3 Web Storage API

Web Storage API poskytuje mechanismus pro lokální ukládání dat typu klíč/hodnota, kde klíč i hodnota jsou řetězce. Existují dva typy:

- Session Storage – uchovává data pouze v rámci jedné session,
- Local Storage – uchovává data i mezi sessions (po zavření prohlížeče).

Data uložená do těchto úložišť se, na rozdíl od cookies, neposílají na server. Jejich velikost je obvykle omezena, nejčastěji na 5 MB (podle prohlížeče). Jedná se o starší rozhraní, podobně jako typovaná pole, a tak podpora ze strany prohlížečů nepředstavuje riziko.

4.1.4 Indexed Database API

Indexed Database API (*IndexedDB*) poskytuje mechanismus pro lokální ukládání dat podobně jako Web Storage API [17]. Liší se v tom, že:

- hodnota může být téměř libovolný datový typ,
- podporuje transakce,
- má asynchronní rozhraní,
- má mnohonásobně větší limit na velikost uložených dat (ale opět záleží na konkrétním prohlížeči).

Jedná se v podstatě o NoSQL databázi. Nedá se proto říci, že jde o náhradu Web Storage API, spíše o komplexnější nástroj. IndexedDB dnes podporují všechny populární prohlížeče, nicméně se jedná o novější rozhraní a ještě nedávno podpora nebyla samozřejmá.

„*IndexedDB is useful for applications that store a large amount of data (for example, a catalog of DVDs in a lending library) and applications that don't need persistent internet connectivity to work (for example, mail clients, to-do lists, and notepads).*“ [17]

4.1.5 Shrnutí možností uložení dat

Na implementaci datových struktur může mít významný vliv to, že JavaScript neobsahuje malé číselné datové typy. Jediný klasický číselný datový typ je 64-bitové číslo s plovoucí desetinnou čárkou. Neobsahuje ani datový typ pro znak, což může mít vliv na implementaci trie (musel by být použit `String` nebo `Number`, které zabírají více paměti). Dalším problémem by mohla být obecně větší velikost objektů v JavaScriptu.

Další možností je využít typovaná pole a vytvořit si pole bajtů. Do vytvořeného pole zakódovat strom nebo jinou strukturu. Tento způsob by však mohl mít dopady na rychlost.

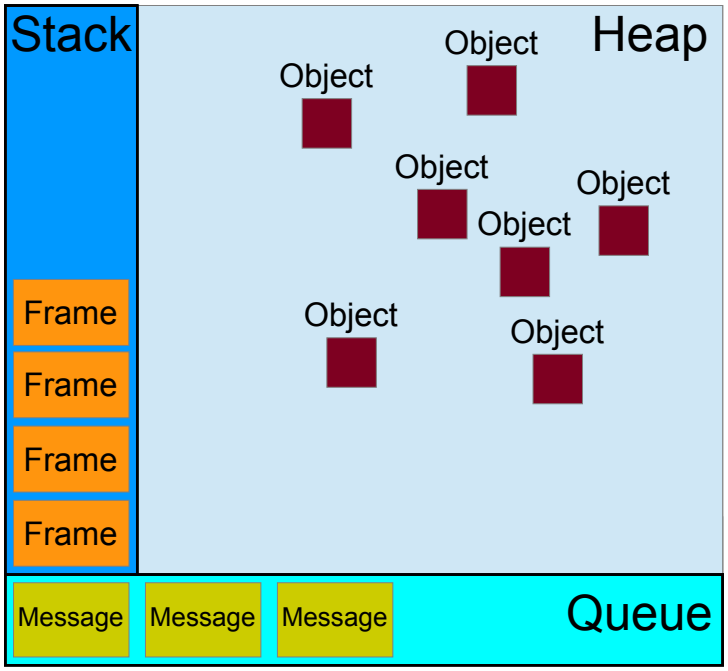
Poslední možností je využít Web Storage API nebo Indexed Database API. Výhoda těchto úložišť je jejich persistence. Je však nutné brát v úvahu omezenou velikost Web Storage API.

4.2 Paralelní zpracování

Od systému pro kontrolu pravopisu se obvykle očekává, že bude pracovat na pozadí a svojí činností nebude zpomalovat psaní textu.

JavaScript má řízení vykonávání kódu založené na modelu zvaném *event loop*. Tento model je velmi odlišný od modelů v běžných jazycích, jako je C nebo java. [16] Všechny události a úkoly se nejprve vkládají do fronty (*event queue* nebo někdy také *message queue* [16]). Event loop postupně vybírá úkoly z fronty a zpracovává je. Tímto způsobem jsou řešena asynchronní volání a přitom vše pracuje v jednom vlákně. Schéma běhového prostředí JavaScriptu je vidět na obrázku 4.1.

JavaScript je bezpochyby zaměřen na jednovláknové aplikace, přesto však nějaké možnosti pro psaní vícevláknových aplikací poskytuje.



Obrázek 4.1: Schéma běhového prostředí JavaScriptu [16].

4.2.1 HTML5 Web Worker

Jediný způsob, jak je možné v JavaScriptu nechat vykonat kód paralelně, je použít HTML5 Web Worker. Worker nemá přístup do aplikačního kontextu a nemůže tedy manipulovat s DOM stránky. Dále není ani možné sdílet data (objekty) mezi workerem a jeho okolím. Jediný způsob komunikace s workerem je prostřednictvím zpráv[1]. Ve zprávě mohou být odeslány primitivní datové typy, kolekce a jednodušší objekty (bez funkcí) [19]. Odeslaná data jsou kopírována, nikoli sdílena [18].

Příklad použití workera je uveden ve výpisech 4.1 a 4.2. Z aplikace je odeslána zpráva, worker ji zachytí a odpoví na ni. Odpověď zachytí aplikace a obsah zprávy vypíše.

Výpis 4.1: Kód aplikace. Inicializace Web Workera a komunikace s ním.

```
var worker = new Worker('/worker.js');
worker.addEventListener('message', function(e) {
    console.log(e.data);
});

worker.postMessage('Hello');   // vypíše Hello world!
```

Výpis 4.2: Kód Web Workera (`worker.js`). Odpovídá na zprávy.

```
self.addEventListener('message', function(e) {
    postMessage(e.data + ' world!');
});
```

JavaScript tedy dává k dispozici nástroj umožňující paralelní programování, byť poněkud méně pohodlný na použití. Při vývoji systému pro kontrolu pravopisu to znamená nutnost definovat protokol pro komunikaci s částí systému, která bude uvnitř workera. Přenášeno by přitom mělo být co nejméně dat, protože veškerá odeslaná data jsou kopírována, nikoli sdílena.

[1]Pokud nepočítáme nepřímou komunikaci například prostřednictvím serveru nebo Indexed Database API [18].

5 Google Web Toolkit

V předešlé kapitole byly zvažovány možnosti, které poskytuje JavaScript v souvislosti s vývojem systému pro kontrolu pravopisu. Tato kapitola s JavaScriptem také úzce souvisí. Cílem této kapitoly je představit framework Google Web Toolkit, který je používán aplikací, do které bude vyvinutý systém pro kontrolu pravopisu integrován a může být použit i systémem samotným.

Google Web Toolkit[1], zkráceně GWT, je open source framework pro tvorbu bohatých webových aplikací[2]. První verze byla vydána v roce 2006 [9]. Vznikal v době rozkvětu technologií jako Adobe Flash, JavaFX[3] a Microsoft Silverlight [27]. Jeho nejdůležitější vlastností je, že umožňuje vývoj webového front-endu v jazyce Java. Kód v jazyce Java je na úrovni zdrojového kódu překládán do jazyka JavaScript. Standardní knihovna Javy je přitom emulována. Oproti vývoji front-endu v JavaScriptu s sebou tento přístup přináší výhody, jako:

- silná typová kontrola Javy,
- určitá možnost využít nástroje, frameworky a knihovny z ekosystému Javy,
- možnost nasadit Java vývojáře na webový front-end,
- serverová část (JVM) může sdílet kód s klientskou částí (web),
- produkovaný kód optimalizovaný (generovaný ve více verzích) pro několik prohlížečových jader,
- stará se o problémy s kompatibilitou a odlišným chováním prohlížečů [27].

Má ovšem i několik nevýhod:

- obtížnější ladění (zavádí další vrstvu komplexity),
- vývoj GWT obvykle zaostává za vývojem Javy a její standardní knihovny[4].

[1] `http://www.gwtproject.org/`

[2] Historicky byl používán termín Rich Internet Application (RIA).

[3] Původně byl framework JavaFX navržen pro tvorbu RIA.

[4] Java 7 byla vydána 28. 7. 2011, ale GWT 2.6.0 s její podporou až 23. 11. 2014. Java 8 byla vydána 18. 3. 2014, ale GWT 2.8.0, který Javu 8 podporuje, až 20. 9. 2016. Java 9, ani začátkem roku 2020, ještě není kompletně podporována. [23][9]

GWT je komplexní webový framework a poskytuje i mechanismy jako *remote procedure call* (RPC), komponenty pro tvorbu uživatelského rozhraní, logování, internacionalizace, přístupnost a další.

V této kapitole bude dále popsáno, co obnáší vývoj v GWT, konkrétně:

- struktura GWT projektu a principy jeho sestavení,
- podpora Javy a její standardní knihovny,
- možnosti interoperability s kódem v JavaScriptu.

5.1 Struktura a sestavení projektu

Pro vývoj v GWT existuje GWT SDK obsahující potřebné knihovny a kompilátor, případně plugin pro vývojové prostředí Eclipse. Další možností je použít pouze Maven plugin, který zajistí kompilaci, ovšem bez možnosti plného využití ladících nástrojů. Pod úrovní Maven projetků GWT zavádí svůj systém modulů. Ty budou dále vysvětleny a spolu s nimi i související pojmy *entry point* a *linker*.

Moduly Třídy se sdružují do GWT modulů. Moduly jsou definovány v XML souborech s názvy ve formátu `<název-modulu>.gwt.xml` (ukázka viz 5.1). V těchto souborech bývají určeny:

- třídy, které do modulu patří,
- závislosti na dalších modulech,
- servlety pro RPC,
- parametry (například parametry překladu),
- logika nahrazování tříd[5],
- linkery (viz dále),
- entry pointy (viz dále).

[5]Může mít charakter podmíněného překladu, například „při kompilaci pro Internet Explorer nahraď třídu A třídou B“ nebo se může jednat o nahrazení tříd či částí knihoven, které nelze zkompilovat pod GWT – na tomto principu je založena emulace standardní knihovny Javy [8].

Výpis 5.1: Ukázka definice modulu.

```
<?xml version="1.0" encoding="UTF-8"?>
<module>
   <!-- Deps -->
   <inherits name='com.google.gwt.user.User'/>
   <inherits name="elemental.Elemental"/>

   <!-- Specify the app entry point class -->
   <entry-point class='cz.harag.app.ApplicationEntryPoint'/>
</module>
```

Entry point Pokud je v modulu definován alespoň jeden entry point, může být modul pomocí systému linkerů přeložen do JavaScriptu. Entry point je třída s konstruktorem bez parametrů a implementující rozhraní `EntryPoint`. Rozhraní má jedinou metodou `onModuleLoad()`, která je vstupní metodou modulu. Pokud je jich v modulu definováno více, budou spouštěny postupně, v definovaném pořadí [8].

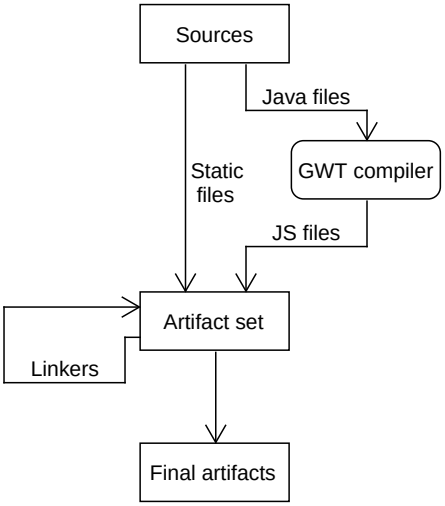
Linkery Linkery se starají o finální podobu generovaných artefaktů. Jsou odpovědí na velký počet různých případů užití GWT, vyžadující rozdílné přístupy. GWT obsahuje několik připravených linkerů a je také možné vytvořit vlastní. Linkerů může být použito i více.

Klíčovým pojmem je *artefact set*. Na počátku obsahuje statické soubory a výstupy kompilátoru. Linkery přistupují k artefaktům v artefakt set, transformují je, odebírají či generují nové. Nakonec zůstane finální množina artefaktů, jak ukazuje obrázek 5.1. [8]

5.2 Podpora Javy

Při vývoji aplikace na platformě GWT je nutné znát limitace, které vyplývají z odlišností Javy a JavaScriptu. Tyto skutečnosti je nutné brát v úvahu a počítat s nimi již od začátku vývoje. Hotový kód může být velmi pracné upravit tak, aby šel zkompilovat pod GWT. Nyní se podíváme na hlavní odlišnosti v podobě:

- datových typů,
- výjimek,
- vícevláknového programování,
- reflexe.



Obrázek 5.1: Schéma fungování linkerů.

Datové typy Přestože jsou všechny datové typy Javy podporovány, některé z nich nemají v JavaScriptu náhradu, a tak je jejich chování emulováno. Datové typy `byte`, `char`, `short` a `int` jsou reprezentovány jako 64-bitové číslo s plovoucí desetinnou čárkou a jejich přetékání je emulováno. Datový typ `long` je zase emulován dvojicí čísel [8]. Je potřeba tedy počítat s horším výkonem u těchto typů a zvážit jejich použití.

Výjimky Výjimky jsou podporovány, avšak ty, které produkuje JVM, nemohly být zcela emulovány, kvůli nemožnosti jejich rozlišení, pokud k nim dojde v JavaScriptu. Jedná se především o `NullPointerException`, `StackOverflowError` a `OutOfMemoryError`. Místo nich dojde k vyhození `JavaScriptException`. [8]

Vícevláknové programování Vzhledem k tomu, že jsou interpretery JavaScriptu jednovláknové, nemůže být tato logika přeložena. Klíčové slovo `synchronized` je proto ignorováno a metody třídy `Object` – `wait()`, `notify()` a `notifyAll()` nejsou podporovány [8]. Dále třídy ze standardní knihovny, které souvisejí s vícevláknovým programováním, buď nejsou podporovány nebo mají triviální implementaci.

Reflexe Reflexe je podporována pouze v omezené míře. Je sice možné standardním způsobem získat objekt typu `Class`, ale většina jeho metod není podporována. Například není podporováno získání seznamu definovaných metod nebo vlastností. Podporováno je například získání názvu třídy

nebo získání `Class` objektu rodičovské třídy. GWT dále nepodporuje ani dynamické načítání tříd [8].

5.3 Podpora standardní knihovny

GWT emuluje standardní knihovnu Javy, ale už z principu odlišné cílové platformy, odlišností popsaných výše a rozsahu standardní knihovny je podporována pouze její malá část. Níže je uvedena podpora klíčových tříd.

Podporované jsou například: [8]

- `java.io`
  - `InputStream`, `OutputStream` a další streamy
- `java.lang`
  - Základní rozhraní
  - Výjimky
  - Anotace
  - Obalové typy
  - `Math`
  - `String`, `StringBuilder`, `StringBuffer`
- `java.math`
  - `BigDecimal`, `BigInteger`
- `java.util`
  - Kolekce
  - `Date`
  - `Random`
- `java.util.concurrent`
  - Kolekce
  - Futures a executory
- `java.util.concurrent.atomic`
- `java.util.function`

- `java.util.logging`
- `java.util.stream`

Podporované nejsou například: [8]

- `java.io`
  - `File` a podobné (týkající se souborů)
- `java.lang`
  - `Thread`
- `java.util`
  - `Pattern`
- `java.nio`

5.4 Interoperabilita s JavaScriptem

Framework GWT programátora sice odstíní od přímého psaní JavaScriptového kódu, ale není natolik vysokoúrovňový, aby bylo možné zapomenout, co je pod ním. Složitější případy užití vyžadují určitou znalost JavaScriptu. Při vývoji v GWT nevyhnutelně narazíme na potřebu pracovat s DOM stránky nebo používat již hotové JavaScriptové knihovny či specifická JavaScriptová rozhraní – například HTML5 Web Workers, Web Storage API nebo typovaná pole z kapitoly 4.

GWT poskytuje několik možností, jak z Javy volat JavaScriptová rozhraní. Každá z těchto možností má jiné použití a v aplikacích založených na GWT se mohou a často používají všechny zároveň. V následujících podkapitolách tedy budou popsány:

- JavaScript Native Interface,
- overlay types,
- knihovna GWT User,
- knihovna GWT Elemental.

5.4.1 JavaScript Native Interface

JavaScript Native Interface (JSNI) je mechanismus GWT, který umožňuje vkládání JavaScriptového kódu přímo do Java tříd. Vypůjčuje si přitom část mechanismu Java Native Interface (JNI). Metody napsané v JavaScriptu jsou označeny klíčovým slovem `native` a jejich tělo je vloženo do speciálně formátovaného blokového komentáře. Tento způsob zaručuje soulad s existujícími vývojovými prostředími pro Javu. Příklad je uveden ve výpisu 5.2.

Výpis 5.2: Příklad nativní metody.

```
public static native void alert(String msg) /*-{
  // kód v JavaScriptu
  $wnd.alert(msg);
}-*/;
```

Jediný rozdíl je v použití `$wnd` místo `window` a `$doc` místo `document`. To je vyžadováno skutečností, že kód GWT je obvykle spouštěn uvnitř frame a do těchto proměnných jsou vkládány správné reference.

Volání Javy z JavaScriptu Z metod definovaných prostřednictvím JSNI lze zpětně přistupovat k metodám a proměnným definovaným v Javě. Vyžaduje to však speciální syntaxi:

```
[instance]@class::method(param-signature)(arguments)
[instance]@class::field
```

kde:

- *instance* – Je výraz vracející instanci. Při volání statické metody nebo přístupu ke statickému parametru se neuvádí.
- *class* – Plně kvalifikované jméno třídy. Například `java.util.Random`.
- *method* / *field* – Název metody nebo proměnné.
- *param-signature* – Parametry metody tak, jak je specifikuje standard JNI[6], ale bez návratového typu, který se uvádí na konci. Uvádí se kvůli rozlišení konkrétní verze metody při jejím přetížení.
- *arguments* – Parametry metody.

Příklad je uveden ve výpisu 5.3.

[6] `https://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/types.html`

Výpis 5.3: Příklad volání Java metod z JavaScriptu.

```
public native void bar(Example x) /*-{
  // instanční metoda
  x.@com.example.Example::myInstanceMethod(Ljava/lang/String
     ;)("Hello");

  // statická metoda
  @com.example.Example::myStaticMethod(Ljava/lang/String;)("
     Hello");

  // instanční proměnná
  var val = x.@com.example.Example::myInstanceField;

  // ...
}-*/;
```

5.4.2 Overlay types

Overlay types zavádí způsob mapování JavaScriptových objektů na Java objekty, s minimální režií a za použití JSNI. Základní myšlenka je ve vytvoření potomka třídy `JavaScriptObject`, který obaluje nativní JavaScriptový objekt a poskytuje metody pro práci s ním. V těchto metodách využívá JSNI a pomocí klíčového slova `this` přistupuje k metodám a parametrům původního JavaScriptového objektu.

Overlay types také mohou implementovat rozhraní. Lze tak například vytvořit společné rozhraní, jednu implementaci pro klientský JavaScript a druhou pro serverovou Javu. Tento přístup je místy použit i v knihovnách GWT (např. GWT User, viz níže).

Dalším využitím overlay types je parsování dokumentů formátu JSON.

5.4.3 Knihovna GWT User

GWT User[7] je hlavní modul frameworku GWT a závislost každé aplikace vyvíjené na platformě GWT. Obsahuje kód jak pro serverovou, tak i pro klientskou část.

V souvislosti s interoperabilitou s JavaScriptem poskytuje Java API například pro:

- DOM – `com.google.gwt.dom`,
- Typed Arrays – `com.google.gwt.typedarrays`,

[7] `https://mvnrepository.com/artifact/com.google.gwt/gwt-user`

- Přístupnost – `com.google.gwt.aria`,
- HTTP klient – `com.google.gwt.http`,
- Web Storage API – `com.google.gwt.storage`.

Různé další pomocné třídy obsahuje balíček `com.google.gwt.core`. Například overlay types pro JavaScriptová pole různých typů – `JsArrayUtils`, `JsonUtils`, `JsDate` a další.

Lze se všimnout konvence v pojmenování balíčků, kdy balíčky obsahující třídy použitelné pouze v klientské části obsahují slovo *client*, balíčky použitelné pouze v serverové části slovo *server* a balíčky použitelné v obou částech slovo *shared*. Tímto způsobem je dále rozčleněn například zmíněný balíček `com.google.gwt.typedarrays`, který obsahuje implementace pro serverovou i klientskou část a společné rozhraní.

5.4.4 Knihovna GWT Elemental

Protože knihovna GWT User neposkytuje Java API ke všemu a použití JSNI poněkud zhoršuje čitelnost a udržovatelnost kódu, vznikla knihovna GWT Elemental[8]. Elemental podporuje veškerá API z HTML5 prostřednictvím overlay types. Toho bylo dosaženo jejich vygenerováním z WebIDL specifikace webového enginu WebKit [8]. Jedná se tedy o velmi nízkoúrovňovou knihovnu, která kopíruje rozhraní v JavaScriptu.

Některá API jsou duplicitní s těmi v GWT User. Na druhou stranu generovaná API knihovny GWT Elemental jsou poněkud méně uživatelsky přívětivá. U duplicitních API tedy bude spíše zájem použít ty z GWT User.

[8] `https://mvnrepository.com/artifact/com.google.gwt/gwt-elemental`

6 Návrh systému a prototypování

V předchozích kapitolách byly uvedeny systémy pro kontrolu pravopisu, představen systém Hunspell a probrány potřebné technologie. Nyní se dostáváme k praktickému návrhu systému.

Jak již bylo nastíněno v předešlých kapitolách, cílem je vytvořit bezkontextový systém pro kontrolu pravopisu, založený na slovníku. Využívat přitom bude slovníky systému Hunspell, které jsou volně k dispozici pro velký počet jazyků. Vytvořený systém bude možné snadno integrovat do webového editoru – to znamená, že bude mít jednoduché a jasně definované API.

Systém musí být dostatečně rychlý na to, aby byl schopen po otevření stránky s editorem co nejrychleji zahájit kontrolu pravopisu a v reálném čase kontrolovat napsaný text. Nesmí přitom blokovat editor. Zároveň nesmí zabírat příliš mnoho paměti. Systém pro kontrolu pravopisu je obvykle pouze doplňkem rozsáhlejší aplikace a netvoří její business logiku. Na jeho provoz je tedy logicky vyčleněno méně prostředků. Není žádoucí, aby kontrola pravopisu vyčerpala prostředky na úkor hlavní funkčnosti aplikace.

Motivací k zahájení projektu, který vyústil v napsání této práce, byl nevyhovující systém pro kontrolu pravopisu webového editoru M/TEXT TONIC[1], vyvíjeného firmou KadeL Data servis. Ten pro kontrolu pravopisu používal svobodnou knihovnu typo.js[2], která nevyhovuje svojí rychlostí inicializace a paměťovou náročností.

Jádrem systému je slovník, proto je klíčové nejprve zvolit vhodnou strukturu pro jeho reprezentaci. V této kapitole budou dále implementovány prototypy slovníků, provedeny měření a nakonec prezentován konečný hrubý návrh systému.

6.1 Prototypování

Kvůli opakovatelnosti měření byl vytvořen projekt `app-performance-tests` s webovou aplikací, ve které jsou, na jednom místě, všechny testované metody/prototypy. Pro aplikaci a prototypy byl použit framework GWT. Cílem bylo rovněž ověřit, že GWT produkuje kód, který je dostatečně výkonný.

[1] `https://www.kadel.cz/mtexttonic.html`

[2] `https://github.com/cfinke/Typo.js/`

6.1.1 Způsob měření

U každé měřené veličiny bylo vždy provedeno 5 měření a z nich spočítán průměr a směrodatná odchylka. Mezi jednotlivými měřeními byla stránka znovu načtena (F5) a čekalo se, dokud se velikost haldy znovu nevrátí zhruba na hodnotu, kterou měla aplikace po spuštění. Mezi testovanými prototypy byl navíc vypnut a znovu zapnut prohlížeč.

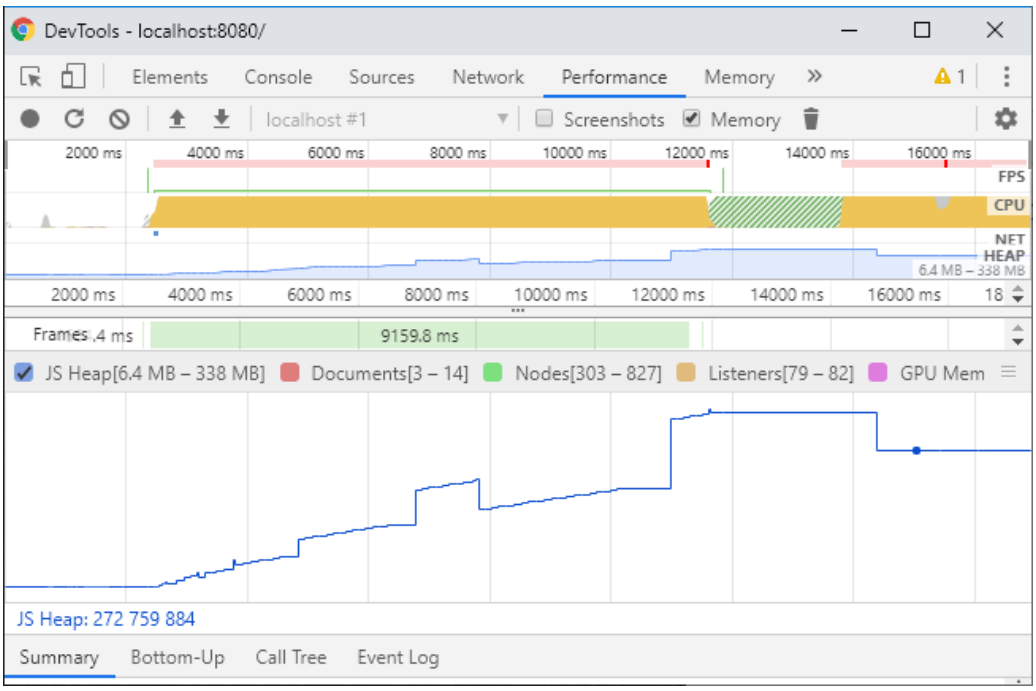
Hodnoty byly zaznamenány pro dva prohlížeče – Google Chrome (verze 80.0.3987.149) a Microsoft Edge (verze 44.17763.831.0). Časové veličiny byly měřeny programově. V případě paměti byly použity vývojářské nástroje a byla zaznamenávána velikost haldy. Klíčové jsou přitom dvě veličiny – maximální velikost haldy při načítání slovníku a „klidová“ velikost haldy po jeho načtení (odráží paměť, kterou bude systém perzistentně držet). Prohlížeče však alokují a uvolňují paměť odlišnými způsoby a jejich vývojářské nástroje se liší možnostmi a kvalitou podávaných informací. Tyto skutečnosti je nutné brát v úvahu při interpretaci výsledků.

Google Chrome (viz obrázek 6.1)

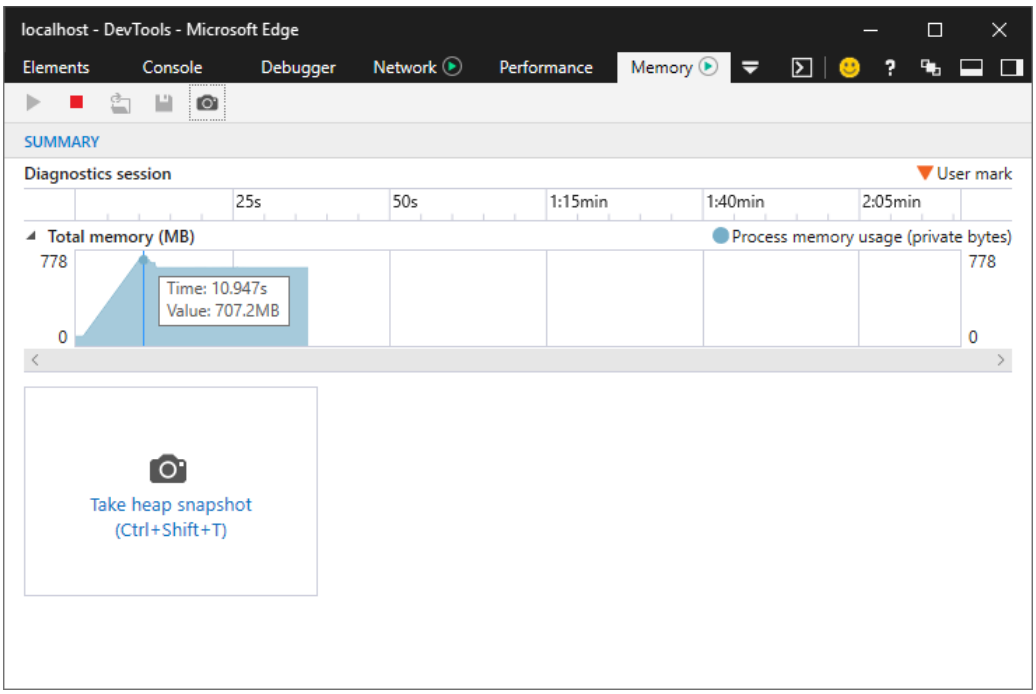
- Po spuštění webové aplikace s prototypy se velikost haldy pohybuje kolem 3,5 MB.
- Udává velikost haldy s přesností na bajty.
- Umožňuje spustit garbage collector (ikona popelnice v toolbaru), čehož bylo využíváno při měření klidové velikosti haldy. Velikost haldy poté, podle zkušeností, relativně přesně odpovídá skutečně využité paměti.

Microsoft Edge (viz obrázek 6.2)

- Velikost haldy po spuštění webové aplikace s prototypy se pohybuje kolem 75 MB.
- Udává velikost haldy s přesností na MB s jednou desetinnou čárkou, při překročení gigabajtu začne velikost udávat v GB.
- Na rozdíl od Google Chrome neumožňuje vynucené spuštění garbage collectoru. Zaznamenaná velikost haldy tedy může být o mnoho větší, než skutečně využitá paměť.
- V případě delších blokujících výpočtů, jsou v grafech vidět dokonalé linie, které jsou zřejmě způsobené proložením přímkou mezi hodnotami před započtením výpočtu a po jeho dokončení. Maximální velikost haldy tedy může být teoreticky ještě větší, než naměřená hodnota.



Obrázek 6.1: Vývojářské nástroje prohlížeče Google Chrome.



Obrázek 6.2: Vývojářské nástroje prohlížeče Microsoft Edge.

Konfigurace Měření byla prováděna na počítači se:

- systémem Windows 10 Pro (10.0.17763 Build 17763),
- procesorem Intel Core i5-4570,
- základní deskou Z87-G45 GAMING (MS-7821),
- diskem Apacer AS340 SSD 960GB,
- operační pamětí 16 GB RAM,
- grafickou kartou NVIDIA GeForce GTX 1660.

6.1.2 Výchozí pozice – typo.js

Jak již bylo zmíněno na začátku této kapitoly, editor M/TEXT TONIC používal pro kontrolu pravopisu knihovnu typo.js, proto s ní budou vytvořené prototypy porovnávány.

Typo.js[3] je jednoduchá JavaScriptová knihovna o rozsahu zhruba tisíce řádek kódu. Využívá slovníky systému Hunspell. Podporuje však pouze menší množinu několika klíčových příkazů. Slovníky parsuje až za běhu. Při načítání slovníků rozvíjí všechna afixová pravidla a slova uloží do hash mapy, kde klíčem je slovo a hodnotou množina příznaků. Podporuje pouze jednu z metod skládání slov, a to *compound rules* (viz kapitola 3.1). Pro napovídání využívá metodu generování slov (viz kapitola 2.3.3).

Byla provedena měření načtení českého slovníku[4] (cca 3 000 000 slov) a kontroly 100 000 slov. Výsledky jsou zaneseny v tabulce 6.1. Google Chrome dosahuje mnohem lepších výsledků, než Microsoft Edge. Kontrola je obecně dostatečně rychlá. Problém představuje doba trvání načítání slovníků a využitá paměť. Výsledky působí o to hůře, pokud přihlédneme ke skutečnosti, že měření byla prováděna na výkonném herním PC, zatímco kancelářské stroje koncových zákazníků takto výkonné pravděpodobně nebudou. Dále pak není neobvyklé, když dokumenty obsahují odstavce v různých jazycích. To by znamenalo znásobení uvedených hodnot.

[3] `https://github.com/cfinke/Typo.js/`

[4] Český slovník ze slovníků LibreOffice, dostupné z: `https://github.com/LibreOffice/dictionaries`

Tabulka 6.1: Výsledky pro knihovnu typo.js na prohlížeči Google Chrome a Microsoft Edge. Průměr pěti měření a směrodatná odchylka.

Prohlížeč	**Načítání**			**Kontrola**
	Paměť max [MB]	**Paměť klidová [MB]**	**Trvání [ms]**	**Trvání [ms]**
Google Chrome	372 ± 19	260 ± 0	8561 ± 577	44 ± 2
Microsoft Edge	705 ± 10	627 ± 4	7977 ± 82	190 ± 40

6.1.3 Prototypy slovníků

Jádrem systému je slovník, proto je klíčové nejprve zvolit vhodnou strukturu pro jeho reprezentaci. Vhodné struktury byly popsány v kapitole 2.2. U prototypů bylo vynecháno parsování Hunspell slovníků, protože cílem bylo zaměřit se na největší problém – uložení slov. Formát slovníků byl zjednodušen na seznam slov oddělených řádkami. U prototypů bylo zanedbáno uložení příznaků, protože všechny testované struktury lze rozšířit tak, aby je bylo možné je ke slovům asociovat.

Výkon prototypů byl změřen podobným způsobem, jako výkon knihovny typo.js. Z českého Hunspell slovníku (cca 3 000 000 slov) byl vygenerován wordlist, aby se zjednodušilo načítání. Byla provedena měření načtení slovníku a kontroly 100 000 slov. Výsledky jsou zaneseny v tabulce 6.2 a 6.3. Nicméně výsledky lze porovnávat s výsledky pro typo.js pouze orientačně, neboť typo.js při načítání navíc parsuje slovníky, se slovy ukládá i příznaky, při testování slov navíc zkouší různé velikosti písmen apod. Tyto skutečnosti mají samozřejmě vliv na paměť a dobu trvání operací.

Prototypování Postupně bylo implementováno pět prototypů slovníků:

- Jako první a nejjednodušší slovník, který byl implementován a změřen, je založen na struktuře hash set. V Javě byla použita třída `HashSet` z balíku `java.util`, kterou GWT podporuje. Tento prototyp byl zamýšlen spíše pro porovnání s ostatními strukturami.

- Další slovník byl založený na trii. Přestože se jedná o strukturu přímo navrženou pro ukládání slovníků, v prostředí JavaScriptu se ji nepodařilo implementovat efektivně. Výsledky jsou ve všech ohledech horší, než ty pro hash set.

- Následovala implementace komprimované trie. Ta se podle očekávání ukázala jako paměťově efektivnější, než její standardní varianta, ovšem

na úkor delší doby jejího budování a delší doby kontroly[5]. Ani jí se ovšem nepodařilo překonat hash set.

- Další slovník byl implementován pomocí seřazeného pole. Měl pouze zjistit, kolik paměti zaberou samotná slova (řetězce) v obyčejném poli. Podle očekávání byl nejvíce paměťově efektivní, nicméně pro reálné použití není taková struktura příliš vhodná.

- Poslední prototyp je založený na standardní trii, která je po sestavení optimalizována. Optimalizace je založena na myšlence, že koncovky slov se opakují, tudíž se opakují i podstromy v trii. Když se podstromy opakují, můžeme mít pouze jednu instanci, která bude ve stromě použita vícekrát. Po optimalizaci již není možné trii upravovat (přidávat nebo odebírat slova), ale to v případě slovníku není problém.

  Optimalizace přinesla velmi výrazné snížení paměťových nároků slovníku po jeho načtení. Maximální alokovaná paměť se samozřejmě nesnížila, protože trie se buduje standardním způsobem, až poté dojde k její optimalizaci. V prohlížeči Microsoft Edge se z nějakého důvodu vylepšení tak výrazně neprojevilo nebo jej jen nereflektovala velikost haldy. Nicméně, když víme, že lze trii takto zmenšit, stačí ji pouze dopředu zakódovat do pole bajtů a místo jejího vytváření při každém načítání, načíst pouze bajtové pole s již sestavenou trií uvnitř.

[5]Rychlost by se zřejmě ještě bývala dala vylepšit, ale kvůli ne příliš dobrým paměťovým nárokům neměly další optimalizace na rychlost smysl.

Tabulka 6.2: Výsledky pro implementované struktury na prohlížeči Google Chrome. Průměr pěti měření a směrodatná odchylka.

Struktura	**Načítání**			**Kontrola**
	Paměť max [MB]	**Paměť klidová [MB]**	**Trvání [ms]**	**Trvání [ms]**
Hash set	288 ± 24	142 ± 0	3790 ± 566	22 ± 3
Standardní trie	353 ± 2	255 ± 0	4913 ± 217	52 ± 4
Kompr. trie	282 ± 2	182 ± 0	16257 ± 332	317 ± 12
Seřazené pole	268 ± 21	98 ± 0	8085 ± 133	317 ± 12
Optimalizovaná standardní trie	364 ± 1	13 ± 0	6729 ± 201	48 ± 5

Tabulka 6.3: Výsledky pro implementované struktury na prohlížeči Microsoft Edge. Průměr pěti měření a směrodatná odchylka.

Struktura	**Načítání**			**Kontrola**
	Paměť max[1] **[MB]**	**Paměť klidová**[1] **[MB]**	**Trvání [ms]**	**Trvání [ms]**
Hash set	809 ± 19	679 ± 3	4452 ± 67	53 ± 1
Standardní trie	1434 ± 0	1126 ± 0	11311 ± 148	141 ± 23
Kompr. trie	1126 ± 0	1085 ± 56	43614 ± 2538	1573 ± 25
Seřazené pole	614 ± 19	501 ± 3	24586 ± 563	301 ± 23
Optimalizovaná standardní trie	1393 ± 56	846 ± 11	18258 ± 396	121 ± 3

[1]Při překročení 1024 MB se velikost začala zobrazovat v GB s jednou desetinnou čárkou, proto nulová směrodatná odchylka apod.

6.2 Konečný návrh

Z testování prototypů vyplynulo, že nejlepším řešením je dopředu zakódovat optimalizovanou trii do bajtového pole a v klientském JavaScriptu toto pole s trií pouze načíst. Web Storage API a Indexed Database API nakonec ani testovány nebyly, protože navržené řešení se zdá být lepší. Trie má také tu výhodu, že se jedná o prefixový strom, což se hodí například při hledání podobných slov, dokončování apod.

Systém bude operovat ve vlákně HTML5 Web Workera. Tak bude mít jeho činnost co nejmenší vliv na zbytek aplikace. Jediný způsob komunikace s workerem je prostřednictvím zpráv. Proto bude muset být specifikován protokol, založený na zprávách, který bude podporovat minimálně operace:

- načtení slovníku,
- kontrolu zadaných slov,
- nalezení podobných slov k zadanému slovu.

Technologie Aplikace, do které se vyvíjený systém pro kontrolu pravopisu bude primárně integrovat používá framework GWT, nicméně to neznamená, že GWT musí být použit i pro systém samotný. Od zbytku aplikace jej stejně bude oddělovat API workera. Použití čistého JavaScriptu by znamenalo větší kontrolu nad kódem. Pro tvorbu prototypů se ale GWT osvědčil. Dalším důvodem pro použití GWT je případná snazší údržba, především pokud by ji měli provádět vývojáři M/TEXTu – aplikace, do které bude systém integrován. Proto bylo rozhodnuto, že framework GWT použit bude.

Pro potřeby parsování slovníků bylo zvažováno použití některé volně dostupné knihovny. Nebyla však nalezena žádná s vhodnou licencí a především taková, která by šla snadno integrovat. Většinou jsou dostupné celé systémy pro kontrolu pravopisu, ze kterých by šlo obtížně získat, co je potřeba. Podpora příkazů Hunspellu v těchto systémech dále dosahuje různých úrovní. Proto je jednodušší a jistější implementovat vlastní parser.

Členění systému Systém se bude z pohledu použití skládat ze dvou částí:

- klientská část provozovaná uvnitř HTML5 Web Workera,
- nástroj, který umožní převod slovníků systému Hunspell do vlastního formátu.

Na úrovni modulů/maven projektů již ale systém takto striktně rozdělen nebude, protože se zde vyskytuje společná část – jádro systému. Využití

GWT totiž umožní její použití jak na webu, tak v nástroji pro převod slovníků, což je velká výhoda.

Ukázková aplikace Pro účely této práce bude dále vytvořena ukázková aplikace s editorem, využívající framework GWT, na které bude předvedena integrace s vyvinutým systémem pro kontrolu pravopisu. Ukázková aplikace vznikne z důvodu, že není možné do této práce zahrnout skutečnou komerční aplikaci. Ukázková aplikace však bude řešit stejné problémy, stejnými způsoby, jako aplikace skutečná.

7 Implementace systému pro kontrolu pravopisu

V předchozí kapitole byly popsány experimenty s různými reprezentacemi slovníku, ze kterých nakonec vzešel základní návrh systému. Cílem této kapitoly je popsat finální implementaci vytvořeného systému pro kontrolu pravopisu.

Jak už bylo zmíněno v předchozí kapitole, práci můžeme rozdělit na tři logické části:

- klientská část, provozovaná uvnitř HTML5 Web Workera;
- nástroj, který umožní převod slovníků systému Hunspell do vlastního formátu;
- ukázková aplikace s editorem a integrovaným systémem pro kontrolu pravopisu.

Na úrovni Maven projektů je situace o něco složitější, protože jich je více a některé jsou použité v rámci více logických částí. Jejich vztahy jsou vidět na obrázku 7.1. Vytvořeny byly následující projekty:

`spellchecker-core` – jádro systému, obsahuje implementaci slovníků, provádí kontrolu pravopisu a napovídání;

`spellchecker-parser-hunspell` – modul s Hunspell parserem, umožňuje načítat slovníky ve formátu Hunspell;

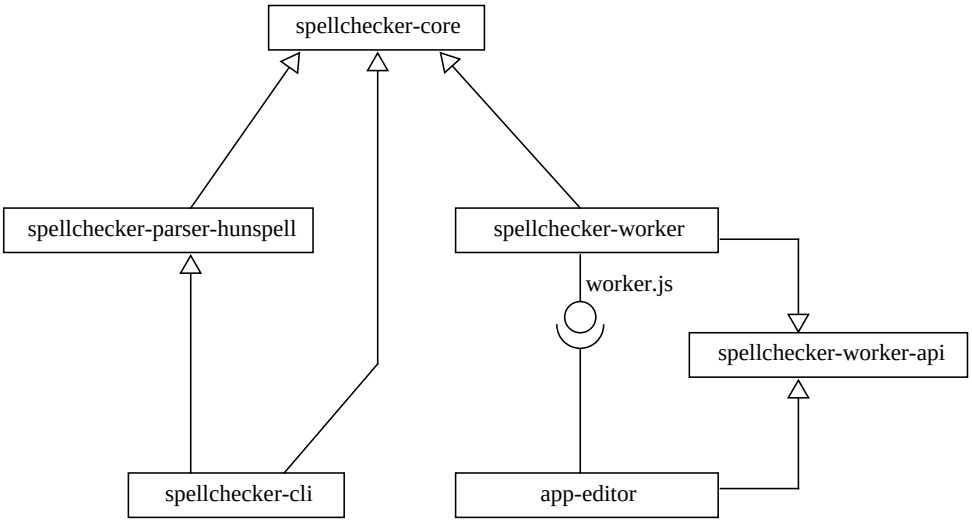
`spellchecker-cli` – nástroj pro konverzi slovníků z formátu Hunspell do vlastního formátu;

`spellchecker-worker` – kompiluje systém do JavaScriptu, výstupem je samostatně spustitelný skript, který lze provozovat jako HTML5 Web Worker;

`spellchecker-worker-api` – definuje formát zpráv pro komunikaci s workerem;

`app-editor` – ukázková GWT aplikace s editorem a integrovaným systémem pro kontrolu pravopisu.

V této kapitole bude dále nejprve popsáno jádro systému, následně zpracování slovníků systému Hunspell, klientská část a nakonec ukázková aplikace.



Obrázek 7.1: Diagram závislostí projektů.

7.1 Jádro systému

Jádro systému obsahuje zejména implementaci slovníků a provádí kontrolu správnosti a napovídání slov. Dále obsahuje následující mechanismy[1]:

- mechanismus pro volné skládání slov;
- podporuje použití regulárních výrazů pro definici akceptovatelných slov a výrazů;
- umožňuje pro konkrétní slovník povolit podporu pro ostrá S (malé ostré S se zapisuje jako „ß“, zatímco velké jako „SS“, což vyžaduje zvláštní zacházení);
- pro mechanismus napovídání slov umožňuje definovat nahrazovací tabulku, která slouží k detekci častých záměn;
- umožňuje označit slova, která jsou sice považována za akceptovatelná, ale není žádoucí, aby byly systémem napovídány;
- umožňuje explicitně definovat slova, která nejsou akceptovatelná (například pro odfiltrování výjimek ze slov, které vznikly aplikací mechanismu skládání slov);

[1]Implementované mechanismy samozřejmě z větší části kopírují mechanismy Hunspellu, aby bylo možné podporovat jeho slovníky.

- umožňuje označit slova, u kterých není akceptovatelná *capitalized* a *uppercased* varianta;
- umožňuje označit slova, která jsou akceptovatelná pouze jako součást složeného slova, nikoli samostatně;
- vstupní a výstupní konverzní tabulku.

Implementace Kořenovým balíkem projektu je `cz.harag.spell`, který obsahuje základní třídy a API pro práci se systémem pro kontrolu pravopisu. Dále obsahuje vnořené balíky:

- `cz.harag.spell.dictionary` – obsahuje rozhraní pro slovník a implementace slovníků;
- `cz.harag.spell.io` – implementace ukládání a načítání vlastního formátu slovníku;
- `cz.harag.spell.parser` – obsahuje rozhraní pro parser a jejich implementace.

Uvedené balíky, včetně jejich vnořených balíků, budou postupně popsány. Nejprve však bude podrobně popsán obsah kořenového balíku.

7.1.1 Základní rozhraní

Nejdůležitější třídy kořenového balíku jsou:

`SpellChecker` – Zapouzdřuje slovník (třída `Dictionary`, viz podkapitola 7.1.2) a konfiguraci (třída `Config`). Poskytuje metodu `check`, pro kontrolu slova a metodu `suggest` pro napovězení podobných slov. Odpovědnost za kontrolu slov přenáší na `CheckProvider` a analogicky odpovědnost za hledání podobných slov přenáší na `SuggestProvider`. Jediná další věc, o kterou se třída stará, je aplikace vstupní konverzní tabulky na slova přicházející metodou `check` a aplikace výstupní konverzní tabulky na nalezená napovězená slova, ještě před jejich vrácením, v metodě `suggest`.

`SpellCheckerBuilder` – Nástroj pro pohodlnější sestavování objektů `SpellChecker`. Nabízí takzvané *fluent* API. Vytvořeno za účelem použití v unit testech, ale i jinde.

`Config` – Uchovává konfiguraci. Konkrétně lze konfigurovat: vstupní a výstupní konverzní tabulky, nahrazovací tabulku pro mechanismus napovídání, patterny pro akceptovatelná slova, minimální velikost složeného slova, povolení kontroly velikosti znaků na hranicích slov ve složených slovech a povolení podpory pro ostrá S.

`Marker` – Výčtový typ s příznaky[2], které se ukládají ke slovům ve slovníku. Systém pro kontrolu pravopisu je využívá ke své činnosti. Nabývá sedmi hodnot: *KEEP_CASE*, *NO_SUGGEST*, *COMPOUND_BEGIN*, *COMPOUND_MIDDLE*, *COMPOUND_END*, *ONLY_IN_COMPOUND* a *FORBIDDEN*.

`CheckProvider` – Řídí proces kontroly správnosti slova. Samotné hledání uvnitř struktury však již zajišťuje slovník (`Dictionary`). Proces kontroly má následující průběh:

1. Nejprve se ve slovníku hledá zadané slovo – tak jak je.
2. Pokud není nalezena přesná shoda, pokračuje se hledáním téhož slova s jinými velikostmi znaků:
   - Pokud vstupní slovo začíná velkým písmenem a ostatní jsou malá (*capitalized form*), tak se zkouší, jestli slovník neobsahuje stejné slovo se všemi malými písmeny (tj. pokud slovník neobsahuje přímo hledané „Slovo“, tak se zkouší „slovo“).
   - Pokud má vstupní slovo všechna písmena velká (*uppercased form*), tak se zkouší, jestli slovník neobsahuje stejné slovo s jinou libovolnou kombinací velikostí znaků (tj. pokud slovník neobsahuje „SLOVO“, tak se zkouší „slovo“, „Slovo“, „SLovo“...).
3. V případě povolené podpory pro ostrá S se navíc bere v úvahu převod ze „ß“ na „SS“ (protože například „STRASSE“ je uppercased varianta slova „Straße“).
4. Následně se zkouší, jestli slovo neodpovídá některému z regulárních výrazů.
5. Jako poslední se zkouší, jestli se nejedná o složené slovo. Vstupní slovo je všemi způsoby rozděleno na dvě části a tyto dvě části jsou samostatně hledány ve slovníku. To samé případně provede ještě s rozdělením na tři části.
6. Pokud dosud nebyla nalezena shoda, je slovo prohlášeno za chybné.

[2]Příznaky kopírují vybrané příkazy slovníků Hunspellu (viz kapitola 3.2.2).

U slov nalezených ve slovníku se samozřejmě kontroluje, že nemají nastaven příznak *FORBIDDEN* nebo *ONLY_IN_COMPOUND*.

`SuggestProvider` – Řídí proces hledání podobných slov. Samotné hledání uvnitř struktury však již zajišťuje slovník (`Dictionary`). Proces hledání podobných slov se skládá ze dvou fází.

V první fázi se hledají slova bez ohledu na velikost znaků a za použití nahrazovací tabulky. Takto nalezená slova mají větší potenciál být uživateli skutečně nápomocná, proto jsou ve výsledném seznamu na prvních místech.

V druhé fázi se hledají slova s nejmenší Levenshtainovo vzdáleností k zadanému slovu. Maximální přípustná vzdálenost je určena dynamicky podle délky slova. Důvodem je, že nemá smysl uživateli nabízet příliš odlišná slova, což by se jinak u krátkých slov reálně dělo. Po testování několika možností bylo nastaveno, že u slov o jednom znaku nebudou, na základě Levenshteinovy vzdálenosti, podobná slova hledána vůbec, u slov do pěti znaků maximálně do vzdálenosti 1, do osmi znaků 2 a nad osm znaků 3. U nalezených slov se nakonec ještě upraví velikost znaků podle vstupního slova. Pokud vstupní slovo začínalo na velké písmeno, tak i napovězená slova se budou snažit začínat na velké písmeno.

Slova označená příznaky *NO_SUGGEST*, *ONLY_IN_COMPOUND* nebo *FORBIDDEN* se mezi napovězená slova nedostanou.

7.1.2 Implementace slovníků

Z testování prototypů vyplynulo, že nejlepším řešením je zakódovat optimalizovanou trii do bajtového pole. Byly proto implementovány dvě struktury, a sice standardní trie, jež je tvořena z objektů, a trie zakódovaná v bajtovém poli. Při načítání slovníku je nejprve sestavována standardní objektová trie, která je po úplném načtení optimalizována a zakódována do bajtového pole, které po vytvoření již nelze modifikovat.

Implementace slovníků se nachází pod balíkem `cz.harag.spell.dictionary`. Přímo u kořene tomto balíku je umístěna rozhraní pro slovník, spolu s dalšími pomocnými třídami. Nejdůležitější jsou:

`Dictionary` – Rozhraní pro slovník. Obsahuje metody:

- `void add(String word, Set<Marker> markers)` – Přidá slovo do slovníku. Metoda je přetížená – rozhraní obsahuje také druhou verzi metody, bez parametru *markers*.

- `FindResult find(String word)` – Pokusí se vyhledat slovo ve slovníku a vrátí instanci třídy `FindResult` s výsledkem hledání.
- `Iterable<FindResult> findWithMapping(String word, MappingConfig config)` – Pokusí se vyhledat slovo ve slovníku za použití mapování (viz dále). Vrací sekvenci `FindResult` s výsledky.
- `void nearest(String word, int maxDiff, boolean ignoreCase, EditDistanceConsumer consumer)` – Vyhledá všechna podobná slova, až do maximální zadané editační vzdálenosti, vůči zadanému slovu. Výsledky jsou předávány konzumentovi.
- `int size()` – Vrátí počet slov ve slovníku.
- `void commit()` – Metoda, která je zavolána po dokončení sestavování slovníku. Slovník toho může využít a optimalizovat svoji vnitřní strukturu. Po zavolání této metody již nemohou být do slovníku přidána další slova.

`FindResult` – Objekt, který nese výsledky vyhledávání. Poskytuje metodu `found()`, která vrací logickou hodnotu a metody `getWord()` a `getMarkers()`. Poslední dvě uvedené metody vrací `null`, pokud slovo nebylo nalezeno.

`MappingConfig` – Rozhraní pro popis mapování. Umožňuje konfigurovat nahrazovací tabulku pro vyhledávání ve slovníku. Dále umožňuje nastavit, jestli se při vyhledávání ve slovníku má brát zřetel na velikost znaků.

`MappingConfigBuilder` – Nástroj pro pohodlnější sestavování objektů `MappingConfig`. Nabízí takzvané *fluent* API.

`DictionaryFactory` – Rozhraní tovární třídy pro slovníky.

`DictionaryFactories` – Třída, která sdružuje všechny tovární třídy pro slovníky na jednom místě.

Dále pak obsahuje vnořené balíky:

- `cz.harag.spell.dictionary.trie` – Obsahuje společný kód pro trii v objektové podobě a trii zakódovanou v bajtovém poli.
- `cz.harag.spell.dictionary.trie.object` – Obsahuje implementaci slovníku pomocí objektové trie.
- `cz.harag.spell.dictionary.trie.binary` – Obsahuje implementaci slovníku pomocí trie zakódované v bajtovém poli.

Obecné rozhraní trie Balík `cz.harag.spell.dictionary.trie` obsahuje společný kód pro trii v objektové podobě a trii zakódovanou v bajtovém poli. Rozhraní, které tvoří pomyslný most mezi těmito strukturami je `AbstractTrieApi`. Díky jednotnému rozhraní bylo možné implementovat algoritmy pro vyhledávání, procházení a hledání podobných slov pouze jednou. O jejich implementaci se starají třídy `TrieFindProvider`, `TrieIteratorProvider` a `TrieEditDistanceProvider`. Balík dále obsahuje abstraktní třídu `AbstractTrieDictionary` implementující rozhraní `Dictionary`, která využívá výše uvedené třídy a rozhraní, a od které jednotlivé slovníky dědí.

Implementace objektové trie Balík `cz.harag.spell.dictionary.trie.object` obsahuje implementaci slovníku pomocí objektové trie. Slovník představuje třída `TrieDictionary`, která dědí od `AbstractTrieDictionary`. `TrieDictionary` uchovává trii složenou z instancí `TrieNode`, která má následující parametry:

- `char value`,
- `boolean endOfWord`,
- `Set<Marker> markers`,
- `TrieNode[] children`.

Pole následníků se udržuje seřazené. Při přidávání slov do trie a při vyhledávání se používá algoritmus binárního vyhledávání.

V konstruktoru `TrieDictionary` je volitelně možné předat instanci `TrieOptimizer`. Jeho úkolem je optimalizovat strukturu trie při zavolání metody `commit()`. Jeho jediná implementace je `TrieOptimizerImpl`[3].

Poslední důležitou třídou v balíku je `TrieStatistics`. Ta umí projít trii, spočítat množství uzlů a sestavit abecedu. Tyto informace jsou nezbytné pro zakódování trie do bajtového pole, ale hodí se i pro testovací účely, apod.

Implementace trie zakódované v bajtovém poli Implementaci slovníku pomocí trie zakódované v bajtovém poli obsahuje balík `cz.harag.spell.dictionary.trie.binary`. Slovník představuje třída `ByteArrayTrieDictionary`, která dědí od `AbstractTrieDictionary`. Slovník může být vytvořen buď z instance `TrieDictionary` nebo z pole bajtů (načítání). Nízkoúrovňovou práci s bajty slovník deleguje na `ByteArrayTrieController`. Pro reprezentaci bajtového pole bylo vytvořeno rozhraní `ByteArray`. Třída,

[3]Princip funkce byl popsán již v kapitole 6.1.3

která jej implementuje, se jmenuje `ByteArrayImpl` a vnitřně používá obyčejné pole bajtů. Použití rozhraní umožní vytvoření speciální implementace pro klientskou stranu, která bude využívat typovaná pole. V balíku je ještě alternativní implementace slovníku, a sice `ByteArrayTrieAdaptiveDictionary`. Ta ve fázi budování slovníku vnitřně používá objektovou reprezentaci trie a při zavolání metody `commit()` trii zakóduje do bajtového pole.

Formát trie v bajtovém poli Trie je v bajtovém poli uložena jako hlavička a vlastní seznam uzlů. Struktura vypadá následovně:

- Hlavička obsahuje:
  - počet slov (4 bajty),
  - velikost ukazatelů v bajtech (1 bajt),
  - velikost abecedy (1 bajt),
  - abeceda (2 × <*velikost abecedy*> bajtů).
- Seznam uzlů, každý uzel obsahuje následující atributy:
  - znak = index znaku v abecedě (1 bajt),
  - příznaky[4] (1 bajt),
    0. příznak konce slova,
    1. *KEEP_CASE*,
    2. *FORBIDDEN*,
    3. *COMPOUND_BEGIN*,
    4. *COMPOUND_MIDDLE*,
    5. *COMPOUND_END*,
    6. *ONLY_IN_COMPOUND*,
    7. *NO_SUGGEST*,
  - počet následníků (1 bajt),
  - seznam ukazatelů na následníky (<*velikost adresy v bajtech*> × <*počet následníků*> bajtů).

Pole se dělí na hlavičku a seznam uzlů. Ve hlavičce je uložený počet slov ve slovníku, abeceda a velikost ukazatele. Prvním uzlem v seznamu je kořenový uzel. Uzel neobsahuje konkrétní hodnotu znaku, ale index do abecedy. Díky tomu je struktura paměťově efektivnější. Uzly jsou mezi sebou propojené absolutními ukazateli. Velikost ukazatele je jeden až čtyři bajty, podle

[4]Viz výčtový typ Marker, kapitola 7.1.1

velikosti trie. Každý uzel má v sobě seznam ukazatelů na své následníky. Pole následníků je seřazené. Při vyhledávání se proto může používat algoritmus binárního vyhledávání.

7.1.3 Ukládání a načítání slovníků

Dalším problémem bylo vybrat vhodný formát pro uložení slovníku do souboru. Požadavkem bylo především jednoduché načtení na klientské straně. Nakonec bylo rozhodnuto, že se slovník bude skládat ze dvojice souborů:

- binárního souboru se slovníkem (přípona *.bin*),
- textového souboru ve formátu JSON s konfigurací (přípona *.json*).

Implementace ukládání a načítání slovníků se nachází pod balíkem `cz.harag.spell.io`. Uložit a načíst pole bajtů je triviální, větší problém však bylo realizovat ukládání a načítání konfigurace do formátu JSON tak, aby byl kód GWT kompatibilní a nemusel být psán vícekrát.

Nakonec byla použita knihovna RequestFactory Client[5]. Jedná se o jednu z knihoven GWT, která se jinak obvykle používá v rámci RPC. Funguje na způsobu automatického mapování JSON na objekty (resp. rozhraní). Za tímto účelem bylo vytvořeno rozhraní `ConfigJSON`. To nese stejné informace jako `Config`, ale ve formátu vhodném pro uložení. Dalším důvodem pro tvorbu další reprezentace konfigurace bylo zajištění větší robustnosti. Třída `ConfigJSONConverter` pak převádí mezi `Config` a `ConfigJSON` a naopak. Vysokoúrovnější metody převod konfigurace z a do formátu JSON poskytuje třída `ConfigIO`. Komplexní seznam metod pro ukládání a načítání celých slovníků poskytuje třída `SpellCheckerIO`. Tato třída však není GWT kompatibilní, protože využívá některé nepodporované třídy z balíku `java.io`.

7.1.4 Parsery slovníků

Implementace parserů jsou pod balíkem `cz.harag.spell.parser`. Pro parsery bylo vytvořeno jednotné rozhraní, a sice `SpellcheckerParser`. Ten obsahuje metodu `parse`, která má na vstupu seznam souborů a jejím výstupem je instance třídy `SpellChecker`. Vzhledem k použití tříd z `java.io` nejsou parsery kompatibilní s GWT a tedy mohou být použity pouze na platformě Java.

[5] `https://mvnrepository.com/artifact/com.google.web.bindery/requestfactory-client`

V rámci systému pro kontrolu pravopisu byly implementovány celkem tři parsery. Pro každý implementovaný parser byl vytvořen vlastní balík:

- `cz.harag.spell.parser.binary` – pro načítání vlastního formátu slovníku, parser implementuje třída `ByteTrieSpellcheckerParser`;
- `cz.harag.spell.parser.wordlist` – pro načítání jednoduchých wordlistů, kde je jedno slovo na jednu řádku, parser implementuje třída `WordListSpellcheckerParser`;
- `cz.harag.spell.parser.hunspell` – pro načítání slovníků systému Hunspell, parser implementuje `HunspellSpellcheckerParser`.

Poslední uvedený parser již ale není součástí projektu `spellchecker-core`. Kvůli přehlednosti pro něj byl vytvořen samostatný projekt `spellchecker-parser-hunspell`. Jeho popisem za zabývá následující podkapitola.

7.2 Zpracování slovníků ve formátu systému Hunspell

Jak již bylo zmíněno v úvodu této kapitoly, cílem bylo vytvořit nástroj, který uživateli umožní převod slovníků systému Hunspell do formátu, který používá vyvinutý systému pro kontrolu pravopisu. Za tímto účelem byl implementován parser a aplikace pro příkazovou řádku.

7.2.1 Implementace parseru

Pro parser byl vytvořen projekt `spellchecker-parser-hunspell`. Načítání slovníku probíhá tím způsobem, že je nejprve zpracován soubor s pravidly (*.aff*), při kterém jsou načtena pravidla a další nastavení. Následně je zpracováván soubor se slovníkem (*.dic*). Podle použitých pravidel jsou ke slovům vygenerovány všechny příslušné afixové kombinace a nastaveny příznaky. Vygenerovaná slova jsou uložena do slovníku. Dochází tedy k dekompresi slovníku z afixového kompresního formátu.

Struktura balíků od sebe odděluje části zabývající se zpracováním souboru s pravidly a souboru se slovníkem. Projekt obsahuje následující balíky:

- `cz.harag.spell.parser.hunspell` – kořenový adresář,
- `cz.harag.spell.parser.hunspell.aff` – obsahuje třídy týkající se zpracování souboru s pravidly,

- `cz.harag.spell.parser.hunspell.dic` – obsahuje třídy týkající se zpracování souboru se slovníkem.

Obsah kořenového balíku Balík obsahuje zastřešující třídy a třídy použité v obou částech. Nejdůležitější třídy:

`HunspellSpellcheckerParser` – Implementace `SpellcheckerParser`. Na vstupu má seznam souborů, které podle přípon rozpozná a zpracuje. Na výstupu je instance typu `SpellChecker`. Využívá `HunspellParser`.

`HunspellParser` – Řídí proces zpracování. Nejprve vytvoří instanci `RuleSet`. Následně nechá `HunspellParserAff`, aby zpracoval soubor s pravidly a naplnil `RuleSet`. Nakonec předá řízení `HunspellParserDic`, který za pomoci `RuleSet` zpracuje soubor se slovníkem.

`Rule` – Rozhraní pro objektovou reprezentaci pravidla. Definuje metodu dávající jeho název a metodu která jej aplikuje.

`RuleVector` – Rozparsovaný vektor pravidel. Vektor pravidel se vyskytuje u slov ve slovníku nebo u afixových pravidel.

`RuleSet` – Obsahuje pravidla a veškeré další nastavení definované v souboru s pravidly.

`Word` – Slovo s množinou příznaků. Nese tyto informace v průběhu generování slov a než se slovo uloží do slovníku.

Zpracování souboru s pravidly Zpracování řídí třída `HunspellParserAff`, která soubor čte po řádcích a rozděluje je na části (podle bílých znaků). Na základě první části, která vždy určuje název příkazu, vybere z hashovací tabulky dílčí parser a nechá ho danou řádku zpracovat.

Dílčí parsery implementují rozhraní `PartialParser` s metodami `getCommandName()`, `parseCommand(String... parts)`, `onAffLoaded()` a `onDicLoaded()`. Při chybě zpracování vyhazují výjimky typu `HunspellFormatException`. Parsery obvykle v konstruktoru získávají instanci `RuleSet`, do které ukládají extrahovaná pravidla a další nastavení. Často se tak děje v metodách `onAffLoaded()` nebo `onDicLoaded()`. Je definováno celkem 16 tříd, implementující rozhraní `PartialParser`. Některé specificky pro jeden příkaz (například `PartialParserCmdREP` nebo `PartialParserCmdCOMPOUNDRULE`), jiné jsou obecnější (například `PartialParserMarker`). Dále budou uvedeny dílčí parsery, které jsou nějakým způsobem zajímavé.

Příkaz *SET*[6] zpracovává `PartialParserCmdSET`. `HunspellParser` vždy začíná slovník zpracovávat s kódováním UTF-8. Když `PartialParserCmdSET` nalezne příkaz *SET* s jiným kódováním, vyhodí výjimku `WrongEncodingException`, kterou `HunspellParser` zachytí, veškerý dosavadní postup zahodí (obvykle se ale jedná o první příkaz v souboru) a začne slovník zpracovávat znovu, se správným kódováním.

Afixová pravidla (*PFX* a *SFX*) zpracovává `PartialParserAffix`. Ten v konstruktoru přebírá hodnotu výčtového typu `AffixType`, nabývající hodnot `PREFIX` nebo `SUFFIX`, a podle toho zpracovává daný typ pravidel. Zpracovaná afixová pravidla jsou reprezentována jako `AffixRule` (implementuje rozhraní `Rule`). `AffixRule` obsahuje kolekci `AffixSubRule` odpovídající jejím dílčím pravidlům.

`PartialParserCmdCOMPOUNDRULE` zpracovává příkazy *COMPOUNDRULE*, první ze dvou mechanismů pro skládání slov. Zpracování má několik fází. Při zpracování souboru s pravidly se pouze ukládají patterny pro tvorbu složených slov. Po dokončení (metoda `onAffLoaded()`) se z patternů extrahují všechna použitá pravidla a pro každé z nich se vytvoří instance `CollectingRule` (implementuje `Rule`), která je vložena do `RuleSet`. Během procesu zpracování souboru se slovníkem, `CollectingRule` zachytává všechna slova označená daným pravidlem a ukládá je do kolekce. Po dokončení (metoda `onDicLoaded()`) se patterny pro skládání slov přetransformují na standardní regulární výrazy, tím způsobem, že se názvy pravidel nahradí konkrétními slovy. Příkladem může být pattern `n*mp`, z anglického slovníku[7], který se přetransformuje na regulární výraz:

```
(0|1|2|3|4|5|6|7|8|9)*(0|1|2|3|4|5|6|7|8|9)\-(0th|1st|2nd|3rd
   |4th|5th|6th|7th|8th|9th)
```

Zpracování souboru se slovníkem Soubor se slovníkem zpracovává třída `HunspellParserDic`. Čte soubor po řádcích a na každé řádce oddělí slovo od vektoru pravidel, vektor zpracuje a následně pravidla aplikuje. Vygenerovaná slova ukládá do slovníku. Dokáže si poradit i se soubory obsahující UTF-8 BOM (*Byte Order Mark*).

[6] Nastavuje kódování slovníku, viz kapitola 3.2.2

[7] `https://github.com/LibreOffice/dictionaries`

7.2.2 Úroveň podpory

Podporovány jsou všechny klíčové příkazy. Ignorovány jsou příkazy týkající se tokenizace, protože tokenizace se provídá mimo systém pro kontrolu pravopisu[8]. Dále jsou ignorovány příkazy týkající se morfologické analýzy a některé příkazy pro napovídání, které jsou příliš specifické pro mechanismus napovídání Hunspellu. Použití neznámého příkazu je zalogováno jako varování.

Podporované příkazy[9]: *AF*, *CHECKCOMPOUNDCASE*, *CHECKSHARPS*, *COMPLEXPREFIXES*, *COMPOUNDBEGIN*, *COMPOUNDEND*, *COMPOUNDFLAG*, *COMPOUNDFORBIDFLAG*, *COMPOUNDLAST*, *COMPOUNDMIDDLE*, *COMPOUNDMIN*, *COMPOUNDPERMITFLAG*, *COMPOUNDRULE*, *FLAG*, *FORBIDDENWORD*, *FULLSTRIP*, *ICONV*, *IGNORE*, *KEEPCASE*, *MAP*, *NEEDAFFIX*, *NOSUGGEST*, *OCONV*, *ONLYINCOMPOUND*, *PFX*, *PSEUDOROOT*, *REP*, *SET* a *SFX*.

Selektivně ignorované příkazy: *AM* (aliasy pro morf. metadata), *BREAK* (tokenizace), *CIRCUMFIX* (příznak pro morf. analýzu), *KEY* (napovídání), *MAXNGRAMSUGS* (napovídání), *TRY* (napovídání), *WORDCHARS* (tokenizace).

7.2.3 Nástroj pro převod slovníků

Pro nástroj byl vytvořen projekt `spellchecker-cli`. Umožňuje převod slovníků z formátu Hunspell do formátu slovníků, který používá vyvinutý systému pro kontrolu pravopisu. Jedná se o jednoduchou Java aplikaci, která využívá moduly `spellchecker-core` a `spellchecker-parser-hunspell`. Nevyužívá žádné další knihovny.

Všechny třídy jsou v balíčku `cz.harag.spell.cli`. Třída se vstupní metodou má název `Main`. Nástroj byl naprogramován tak, aby umožňoval snadné rozšíření. Aktuálně je jeho jediném účelem převod slovníků, ale do budoucna by mohl provádět, podle potřeby, i další operace. Byla také vyvinuta snaha o oddělení parsování a samotného vykonání, a to především z důvodu lepší testovatelnosti.

Hlavním artefaktem je soubor `spellchecker-cli-<verze>-jar-with-dependencies.jar`, který je samostatně spustitelnou Java aplikací. Způsob použití je popsán v příloze.

[8] Tokenizaci je lepší nechat na editoru. Jedná se o netriviální úlohu, která závisí na zvoleném jazyce. Druhým důvodem je, že už ji obvykle už obsahují kvůli implementaci zalamování řádků a slov.

[9] Příkazy slovníků systému Hunspell jsou popsány v kapitole 3.2

7.3 Klientská část

Klientská část systému pro kontrolu pravopisu operuje uvnitř HTML5 Web Workera. Tím je zajištěna responzivnost zbytku aplikace. Worker funguje tak, že čeká na požadavky, zpracovává je a vrací odpovědi. Požadavky se přitom myslí načtení slovníku, ověření slov nebo nalezení podobných slov. Požadavky se zpracovávají sekvenčně. Odpovědi tedy chodí v pořadí, v jakém byly zaslány požadavky.

7.3.1 Rozhraní pro komunikaci

Jediný způsob komunikace s HTML5 Web Workerem je prostřednictvím zpráv (viz kapitola 4.2.1). Proto bylo definováno rozhraní, které umožňuje komunikaci prostřednictvím zpráv ve formátu JSON. Existují tři typy operací:

- *init* – načte slovník ze souboru.
- *check* – ověří, zda se slova nacházejí ve slovníku. Vrací pole indexů chybných slov.
- *suggest* – napoví podobná slova k danému chybnému slovu. Vrátí maximálně 5 podobných slov. V případě, že je slovo správné nevrátí nic.

Každý zpráva obsahuje parametr *cmd*, který určuje typ operace. Každá odpověď dále obsahuje atribut *ok*, datového typu `boolean`. Ten značí, že zpracování požadavku proběhlo v pořádku. V případě, že došlo k chybě, bude odpověď obsahovat i atribut *msg* s chybovou zprávou.

Tabulka 7.1 obsahuje parametry požadavků a odpovědí. Ke každé operaci je uveden seznam parametrů požadavku a odpovědi. Všechny parametry požadavků jsou povinné

Výpis 7.1: Příklad sekvence požadavků a odpovědí na ně.

```
Požadavek: {"cmd":"init","language":"en_US","urlBin":"dicts/en_US.bin",
    "urlJson":"dicts/en_US.json"}
Odpověď: {"cmd":"init","language":"en_US","ok":true}

Požadavek: {"cmd":"check","language":"en_US","words":["Impementation",
    "hello"]}
Odpověď: {"cmd":"check","language":"en_US","ok":true,"incorrect":[0]}

Požadavek: {"cmd":"suggest","language":"en_US","word":"Impementation"}
Odpověď: {"cmd":"suggest","language":"en_US","ok":true,"words":[
    "Implementation","Implementations","Implantation","Implementation's",
    "Documentation"]}
```

Ve výpisu 7.1 je vidět příklad sekvence požadavků a odpovědí. Načtení anglického slovníku, ověření slova a vyhledání podobných slov.

Projekt `spellchecker-worker-api` obsahuje definice formátů požadavků a odpovědí pro komunikaci s HTML5 Web Workerem. Obsahuje *overlay types*, které lze serializovat do formátu JSON nebo naopak použít pro jeho zpracování. Využívá ho projekt `spellchecker-worker` (viz dále). V případě nasazení systému do webové aplikace na platformě GWT lze toto API také použít – tak se děje například v projektu ukázkové aplikace (viz další podkapitola).

Tabulka 7.1: Rozhraní workera.

Operace	**Parametry**	
Načtení slovníku	Požadavek	"cmd": "init", "language": "<identifikace jazyka>", "urlBin": "<URL k souboru *.bin>", "urlJson": "<URL k souboru *.json>"
	Odpověď	"cmd": "init", "ok": <úspěch operace>, "language": "<identifikace jazyka>"
Ověření slov	Požadavek	"cmd": "check", "language": "<identifikace jazyka>", "words": ["<slovo 1>",... "<slovo n>"]
	Odpověď	"cmd": "check", "ok": <úspěch operace>, "language": "<identifikace jazyka>", "incorrect": ["<index 1>",... "<index m>"]
Napovězení podobných slov	Požadavek	"cmd": "suggest", "language": "<identifikace jazyka>", "word": "<chybné slovo>"
	Odpověď	"cmd": "suggest", "ok": <úspěch operace>, "language": "<identifikace jazyka>", "incorrect": ["<návrh 1>",... "<návrh n>"]

7.3.2 Implementace klientské části

Projekt `spellchecker-worker` má za cíl, pomocí GWT, zkompilovat jádro systému (`spellchecker-core`) do JavaScriptu tak, aby jej bylo možné provozovat jako HTML5 Web Worker. Kromě samotné kompilace tedy také implementuje rozhraní, pomocí kterého lze s workerem komunikovat (zadávat mu úkoly).

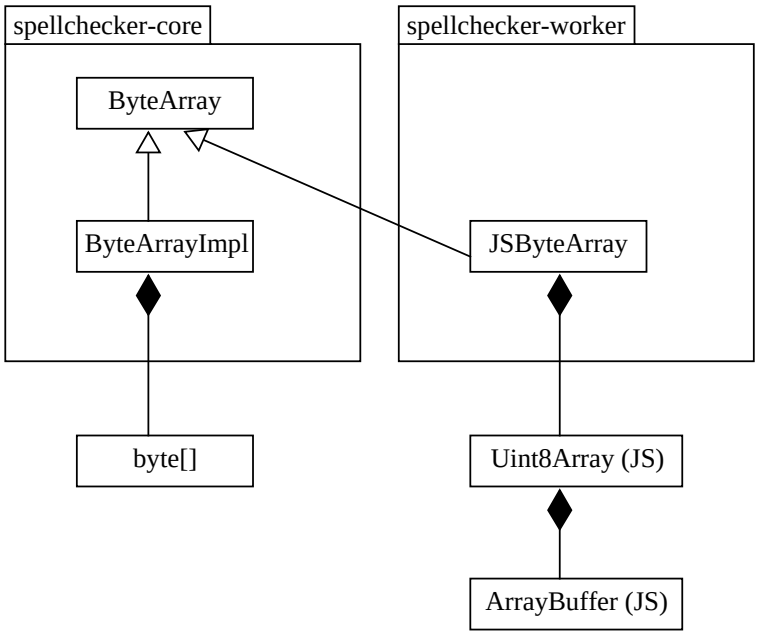
Projekt obsahuje balíky `cz.harag.spell.gwt.linker` a `cz.harag.spell.gwt.worker`. První z nich souvisí s kompilací do JavaScriptu a bude popsán až v podkapitole 7.3.3. Druhý balík obsahuje hlavní logiku – implementaci rozhraní workera a načítání slovníků.

Vstupem je metoda `onModuleLoad()` třídy `WorkerEntryPoint`. Metoda pouze zaregistruje handler pro příchozí požadavky a jejich samotné zpracování nechá na třídě `RequestProcessor`. Ta zprávy ve formátu JSON převede do objektové reprezentace a následně je zpracuje. Chybové stavy, jako nevalidní JSON nebo chybějící atributy, jsou ošetřeny.

Načtené slovníky jsou spravovány ve třídě `SpellCheckManager`. Načítání slovníků je realizováno prostřednictvím protokolu HTTP, pomocí JavaScriptového rozhraní `XMLHttpRequest` (jinak známé jako AJAX). Zajišťuje ho třída `SpellCheckerLoader`. Načítání je asynchronní a jeho dokončení je signalizováno zavoláním metod listeneru `SpellCheckerLoaderListener`. Slovník je úspěšně načten až po úspěšném načtení obou souborů, ze kterých se skládá.

- Konfigurace (soubor *.json*) je načtena s `responseType = 'text'`, tedy jako objekt typu `String`, a následně rozparsována s použitím rozhraním, které poskytuje modul `spellchecker-core` (viz kapitola 7.1.3).
- Slovník (soubor *.bin*) je načten s `responseType = 'arraybuffer'`, tedy jako objekt typu `ArrayBuffer`, ze kterého je vytvořeno typované pole `Uint8Array` (analogie `byte[]` v Javě). Pole je následně zabaleno třídou `JSByteArray`, která implementuje `ByteArray` pocházející ze `spellchecker-core`. S objektem implementujícím `ByteArray` je již možné načíst slovník s použitím rozhraní `spellchecker-core`. Obrázek 7.2 zobrazuje vztahy mezi zmíněnými třídami.

Pro minimalizaci množství nativního JavaScriptového kódu je použita knihovna GWT Elemental – především pro zaregistrování handleru workera a vytváření HTTP požadavků. Overlay types pro `ArrayBuffer` a `Uint8Array` a další pocházejí z knihovny GWT User.



Obrázek 7.2: Vztahy třídy `ByteArray` a jejích potomků. `ByteArrayImpl` je implementace pomocí pole bajtů, pro použití na JVM. `JSByteArray` je implementace využívající typované pole `Uint8Array`, pro použití na klientské straně.

7.3.3 Kompilace do JavaScriptu a sestavení

Struktura GWT projektů a jejich kompilace byla popsána v kapitole 5.1. V rámci projektu `spellchecker-worker` jsou definovány tři GWT moduly:

- `cz.harag.spell.SpellChecker` – Obsahuje jádro systému pro kontrolu pravopisu. Jinými slovy se jedná o projekt `spellchecker-core` definovaný jako GWT modul. Neobsahuje části, které souvisejí s parsery slovníků a prací se soubory. Dále nahrazuje třídu `cz.harag.spell-.PatternRegex`, která využívá frameworkem GWT nepodporovanou třídu `java.util.Pattern`, za implementaci využívající JavaScriptový `RegExp` a dále pak třídu `com.google.web.bindery.autobean.vm.AutoBeanFactorySource`, z knihovny pro práci s JSON dokumenty, která nešla zkompilovat.

- `cz.harag.spell.gwt.WorkerLinker` – Speciální GWT linker, který musel být vytvořen, protože generování kódu pro HTML5 Web Worker není běžný způsob použití frameworku GWT. Jeho princip spočívá v tom, že místo několika výstupů optimalizované pro různé prohlížeče

a dalších podpůrných skriptů, generuje pouze jeden samostatně spustitelný skript.

- `cz.harag.spell.gwt.Worker` – Obsahuje výkonný kód a hlavní funkcionalitu Web Workera. Ke svému sestavení používá `WorkerLinker` a má také závislost na modulu `SpellChecker`.

Kompilaci a sestavení řídí Maven a funguje následovně:

1. Kompilace do JavaScriptu je spuštěna prostřednictvím pluginu `gwt-maven-plugin`, ve fázi `prepare-package`. Kompiluje se modul `cz.harag.spell.gwt.Worker`. Ostatní moduly jsou mezi jeho závislostmi.

2. Plugin `maven-resources-plugin` ve fázi `package` přesune výsledný skript do výstupního adresáře.

3. Následně je ve fázi `package`, pomocí pluginu `maven-jar-plugin`, vytvořen artefakt s klasifikátorem `worker`[10], jehož obsahem je soubor `worker.js`. Tento samostatný script obsahuje celou klientskou část systému pro kontrolu pravopisu, včetně všech závislostí. To znamená, že jej lze jej lze použít na libovolné webové stránce, ne jen ve webové aplikaci na platformě GWT.

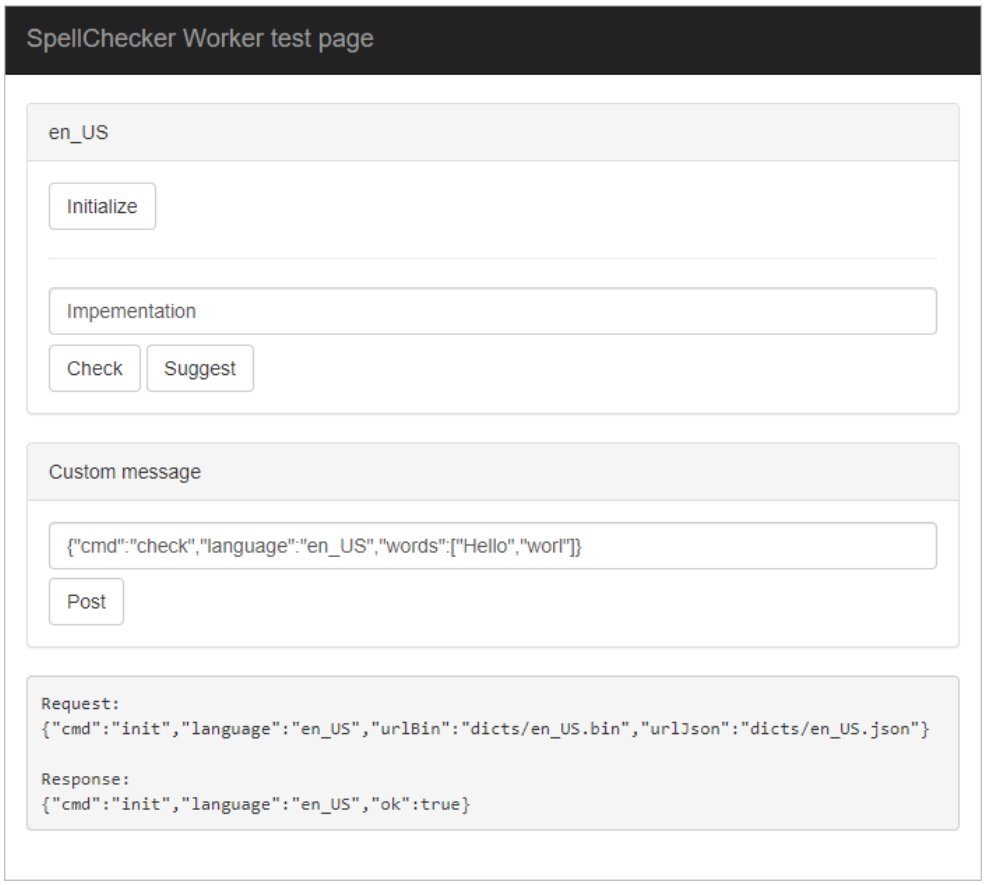
7.3.4 Testovací aplikace

Během implementace a ladění klientské části chyběla možnost, jak jednoduše a rychle ověřit funkčnost workera, bez nutnosti sestavování a spouštění celé komplexní aplikace s editorem (které se věnuje následující podkapitola).

Za tímto účelem byl vytvořen projekt `app-worker-test-page`, který obsahuje jednoduchou webovou stránku s nasazeným systémem pro kontrolu pravopisu. Lze z ní poslat požadavky workerovi, i v surové podobě, a zkontrolovat odpovědi. Umožňuje snadné manuální testování vytvořeného workera. Byla použita při ladění a testování – zejména pro kontrolu ošetření nevalidních vstupů apod. Jedná se také o příklad nejjednodušší integrace systému, bez GWT a dalších technologií, pouze za použití jednoduchého web serveru.

Obrázek 7.3 ukazuje vzhled aplikace. V prvním panelu jsou připravené akce pro anglický slovník. V druhém panelu je pak možné sestavit vlastní požadavek v surové podobě. Dolní část obsahuje výpis poslední provedené akce.

[10] `spellchecker-worker-<verze>-worker.jar`



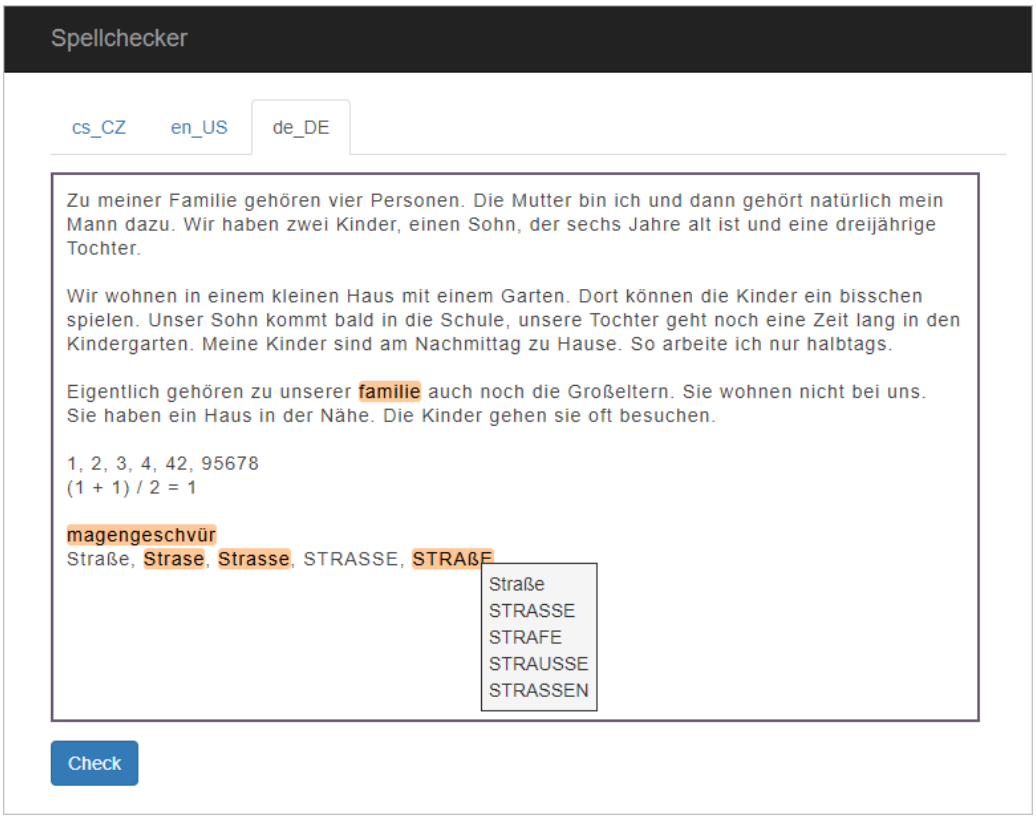
Obrázek 7.3: Vzhled testovací aplikace.

7.4 Ukázková aplikace

Projekt `app-editor` obsahuje ukázkovou GWT aplikaci využívající vyvinutý systém pro kontrolu pravopisu. Jedná se tedy o příklad integrace. Ukázková aplikace vznikla z důvodu, že není možné do této práce zahrnout skutečnou komerční aplikaci. Ukázková aplikace však řeší stejné problémy, stejnými způsoby, jako aplikace skutečná.

Aplikace obsahuje několik připravených slovníků[11]. Pro každý tento slovník zobrazuje jednoduchý editor, který v sobě integruje vyvinutý systém pro kontrolu pravopisu. Je tedy možné vyzkoušet jeho funkčnost. Podporováno je ověření slov i napovídání (po pravém kliknutí na chybné slovo). Uživatelské rozhraní je zobrazeno na obrázku 7.4. Editor samotný je také pouze pro ukázku, protože tvorba editoru nebyla předmětem této práce. Reálně byl systém integrován do komplexního WYSIWYG editoru, to však z pohledu kontroly pravopisu není podstatné.

[11] Zdroj: `https://github.com/LibreOffice/dictionaries`



Obrázek 7.4: Ukázková aplikace s editorem a nasazeným systémem pro kontrolu pravopisu.

7.4.1 Implementace ukázkové aplikace

Kořenovým balíkem je `cz.harag.spell.app`, který má jediný vnořený balík `client`. Ten obsahuje vstupní třídu `SampleApplicationEntryPoint` s metodou `onModuleLoad()`. Metoda pouze vyhledá elementy v šabloně podle jejich ID a na jejich místě nechá inicializovat panel s editorem. Samotnou inicializaci již ale provádí třída `EditorPanel`. Ta vytvoří panel s editorem a tlačítkem spouštějící kontrolu, a nakonec načte ukázkový text. Dále obsahuje vnořené balíky:

- `cz.harag.spell.app.client.dict` – Obsahuje třídy pro správu slovníků a komunikaci s Web Workerem.
- `cz.harag.spell.app.client.editor` – Obsahuje třídy týkající se komponenty editoru.

Implementace správy slovníků Na pomyslné nejnižší úrovni se nachází třída `WorkerWrapper`. Ta komunikuje přímo s Web Workerem a posky-

tuje rozhraní pro práci s ním. Zajišťuje jeho inicializaci, zavádí obsluhu odpovědí přicházejících od Web Workera a předává mu požadavky. Požadavky a odpovědi přitom mapuje na jejich objektové reprezentace z projektu `spellchecker-worker-api` (viz kapitola 7.3.1). Při vytváření Web Workera je do URL přidáván parametr, který zabraňuje cachování [12]. Cachování nejenže dělalo problémy při vývoji, ale mohlo by docházet i k problémům u koncových zákazníků, kdy by po nasazení nové verze produktu mohl být používán starý kód (není možné spolehlivě zajistit vyprázdnění cache v klientských prohlížečích po nasazení nové verze).

Třídu `WorkerWrapper` využívá správce slovníků `WorkerDictionaryManager`, implementující rozhraní `DictionaryManager`. Obsahuje metody `loadDictionary` a `getDictionary` vracející objekt slovníku. Metoda pro načtení slovníku je neblokující a volající se o výsledku informován callbackem. Slovníky jsou identifikovány unikátním jménem.

Slovník je reprezentován rozhraním `Dictionary` a je implementován třídou `WorkerBasedDictionary`. Slovník obsahuje metody `check` a `suggest`. Obě tyto metody jsou neblokující a výsledek je vrácen callbackem. Své neobsloužené požadavky si spravuje slovník sám, správce slovníků pouze informuje o jejich dokončení. Slovník obsahuje frontu handlerů pro `check` i `suggest`. Vzhledem k tomu, že jsou požadavky ve Web Workerovi zpracovávány sekvenčně, je přiřazení dokončeného požadavku k příchozí odpovědi realizováno prostým výběrem z fronty.

Implementace editoru Editor je v základu standardní HTML textové pole, před které je umístěn panel se zvýrazněním chybných slov. Na textové pole je navěšena obsluha posunutí jeho zobrazené oblasti, která stejným způsobem posune i překrývající panel. Panel je, s výjimkou ploch značících chybná slova, průhledný a nezachytává žádné vstupní akce. Zvýrazněná místa jsou pak poloprůhledná a zachytávají vstupní akce[13]. Lze na nich vytvořit kontextového menu s návrhy správných slov. Panel obsahuje stejný text jako textové pole, jen neviditelný – tak je dosaženo umístění zvýrazněných ploch na správná místa. Zvýrazněná plocha je pak pouze pozadí příslušného úseku textu (`background-color`). Aby vše fungovalo, mají textové pole a překrývající panel nastavené stejné rozměry, okraje, řádkování, font a další vlastnosti. Panel se zvýrazněním se formátuje při každé změně v editoru, ale kontrola správnosti slov probíhá pouze v případě změny ve slovech. Netýká se tedy přidání oddělovače (interpunkční znaménka, mezery...), pokud nezpůsobí vznik dalšího slova. Analogicky odebrání oddělovače.

[12] URL pak vypadá například jako: `/js/worker.js?v=0.6514152187196`.

[13] Menší nevýhoda: přes zvýrazněná slova se nedá prokliknout do textu.

Nejdůležitější třídou je `Editor`. Poskytuje metody pro nastavení textu, provedení ověření správnosti slov a pro získání editoru jako instance GWT `Widget`, který lze následně umístit do GWT aplikace. Ostatní třídy v balíčku jsou využívány přímo třídou `Editor`. Určit hranice slov má na starost třída `Tokenizer`. Bere v úvahu běžné bílé znaky, interpunkční znaménka a čísla. Vnitřně používá regulární výrazy. `HighlightingFormatter` se stará o formátování překrývajícího panelu se zvýrazněním chybných slov. `ContextMenuProvider` vytváří na daném místě kontextové menu se seznamem podobných slov.

7.4.2 Sestavení a nasazení

Klíčovým úkolem při sestavování bylo zajistit, aby se stáhnul artefakt obsahující skript pro Web Worker, rozbalil se a umístil do adresáře, odkud k němu bude mít aplikace při běhu přístup.

Stejně jako ostatní projekty, i tento využívá Maven. Build script obsahuje nastavení kompilace GWT, a to prostřednictvím pluginu `gwt-maven-plugin`. Dále pak stažení artefaktu s Web Workerem (`spellchecker-worker`) a jeho rozbalení na správném místě. To zajišťuje `maven-dependency-plugin` jeho cílem `unpack`. Ve fázi `package` je standardně produkován WAR s webovou aplikací. Ten je možné nasadit na aplikační server na Javě 8.

8 Testování

Předchozí kapitola popisovala finální implementaci vytvořeného systému pro kontrolu pravopisu. Cílem této kapitoly je popsat, jakým způsobem probíhalo ověřování kvality systému. Testování lze rozdělit na:

- jednotkové testování,
- testování klientské části,
- testování systému na slovnících evropských jazyků,
- testování výkonu (rychlost, paměťová náročnost),
- funkční testování.

8.1 Jednotkové testování

Projekty, které bylo z jejich podstaty možné otestovat jednotkově, používají knihovnu JUnit pro tvorbu jednotkových testů. Především tedy jádro systému (`spellchecker-core`) a parser Hunspell slovníků (`spellchecker-parser-hunspell`). U těch jsou velmi často použity parametrizované testy. Ale několik testů má i nástroj pro příkazovou řádku (projekt `spellchecker-cli`) a ukázková aplikace (projekt `app-editor`).

Co se týče jádra systému (projekt `spellchecker-core`), za zmínku stojí například `BasicDictionaryTest` (ukázka viz výpis 8.1) a `AdvancedDictionaryTest`, což jsou parametrizované testy, které stejným způsobem ověřují funkčnost všech implementací slovníků (rozhraní `Dictionary`). `ParametrizedSuggestTest` zase prochází seznam stovek slov s překlepy, ke každému hledá až 10 podobných slov a výsledky porovnává s referenčními daty. Jeho úkolem je upozornit na jakékoliv změny v mechanismu napovídání. `BackwardCompatibilityTest` testuje zpětnou kompatibilitu binárního formátu slovníků.

V případě parseru Hunspell slovníků (projekt `spellchecker-parser-hunspell`) lze mluvit spíše o integračních data-driven testech. Test se vždy skládá z minimalistického slovníku a seznamu správných a chybných slov. Při spuštění je slovník načten a všechna slova v seznamu správných slov musí být systémem pro kontrolu pravopisu označena jako správná, analogicky u seznamu chybných slov. Testy tak ověřují parsování slovníků i vyhodnocování.

Testů je velké množství a obvykle ověřují správnou interpretaci jedné konkrétní vlastnosti (určité pravidlo nebo nastavení). Podobný typ testů existuje i pro napovídání. Část testovacích dat pochází z originálních testovacích dat Hunspellu[1], což by mělo zajistit správnost interpretace podporovaných příkazů.

Jednotkovým testům byla od počátku vývoje věnována velká pozornost, a tak pokrývají poměrně vysoké procento řádek kódu. Jádro systému je vlastními testy pokryto z 88 % (1123/1271). Pokud k němu přidáme testy Hunspell parseru popsané výše, je pokryto 93 % (1178/1271) řádek kódu. Hunspell parser samotný má pokrytých 92 % (629/687) řádek kódu.

Výpis 8.1: Část testu `BasicDictionaryTest` pro ukázku.

```
@RunWith(Parameterized.class)
public class BasicDictionaryTest {

  @Parameterized.Parameters
  public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
      { DictionaryFactories.trieDictionary() },
      { DictionaryFactories.trieDictionaryOptimized() },
      { DictionaryFactories.byteArrayTrieDictionary() },
      { DictionaryFactories
            .byteArrayTrieDictionaryOptimized() },
    });
  }

  private final DictionaryFactory provider;
  private Dictionary dictionary;

  public BasicDictionaryTest(DictionaryFactory provider) {
    this.provider = provider;
  }

  @Before
  public void init() {
    this.dictionary = provider.create();
  }

  @Test
  public void testSizeSimple() {
    dictionary.add("a");
    dictionary.add("aa");
    dictionary.add("aaa");
```

[1] https://github.com/hunspell/hunspell/tree/master/tests

```
    dictionary.add("aab");
    dictionary.commit();

    assertEquals(4, dictionary.size());
  }

  // + další testy
}
```

8.2 Testování klientské části

Dalším výzvou bylo testování klientské části. Tedy skriptu vygenerovaného GWT, který lze nasadit jako HTML5 Web Worker. Cílem bylo vytvořit automatizované testy, které zvládnou ověřit, že se skript vygeneroval správně a funguje, dále pak ověřit jeho rozhraní a načítání slovníků. Cílem nebylo dopodrobna ověřovat funkce kontroly pravopisu jako takové, ty již ověřují jednotkové testy. Testy je zapotřebí provést v prohlížeči. Řešením bylo použít jednoduchou testovací stránku s nasazeným systémem, která byla představena v kapitole 7.4.1, a vytvořit automatizované testy. Klíčové je, že stránka umožňuje zasílání zpráv workerovi v surové podobě.

Pro implementaci testů byl použit Robot Framework[2]. Jedná se o obecný nástroj pro automatizaci a tvorbu automatizovaných testů. Pro automatizaci v prostředí webového prohlížeče používá známé Selenium[3]. Robot Framework má vlastní jazyk pro tvorbu automatizačních skriptů, který má tu výhodu, že umožňuje poměrně rychlou a jednoduchou tvorbu testů. Bylo vytvořeno celkem 11 testů, které testují především rozhraní workera. Testována byla také například reakce na chybějící povinné parametry, reakce na neexistenci souborů slovníku apod.

Výpis 8.2 ukazuje jeden test, který má název *Test command not set*, včetně konfigurace (kompletní spustitelný skript). Test funguje tak, že nejprve vykoná proceduru s názvem *Initialize*, která nastaví prodlevu mezi příkazy na 250 milisekund, otevře prohlížeč a klikne na tlačítko pro načtení slovníku. Dále vloží vstup a odešle ho. Nakonec porovná odpovědi. Kompletní skript se liší pouze v tom, že obsahuje dalších 10 podobných testů.

[2] `https://robotframework.org`
[3] `https://www.selenium.dev`

Výpis 8.2: Skript Robot frameworku s konfigurací a jedním testem.

```
| *** Settings ***
| Library | SeleniumLibrary

| *** Keywords ***
| Initialize
| | Set Selenium Speed | 0.25
| | Open browser | http://localhost:8000/index.html | Chrome
| | Click Button | id:btn-load

| *** Test Cases ***
| Test command not set
| | Initialize
| | Input Text | id:input-message | \{"language":"en_US"\}
| | Click Button | id:btn-post
| | ${response} | Get Text | //span[@id='response']
| | Should Be Equal As Strings | ${response} | \{"ok":false,"
   msg":"Command type not set"\}
| | Close All Browsers
```

8.3 Testování systému na slovnících evropských jazyků

Systém byl otestován Hunspell slovníky devíti významných evropských jazyků. Pro otestování byly použity slovníky z projektu LibreOffice[4]. Slovníky byly převedeny do vlastního formátu systému pro kontrolu pravopisu. Tabulka 8.1 zobrazuje informace o velikosti původních a konvertovaných slovníků. Pro lepší představu o rozsahu jsou kromě velikosti souborů uvedeny také počty kořenů (počet slov v souboru s příponou *.dic*) u původních slovníků a počty slov ve výsledném slovníku, po rozvinutí afixových pravidel.

Žádný ze slovníků neobsahoval příkazy, se kterými by systém neuměl pracovat. Pouze některé slovníky obsahovaly nezdokumentované příkazy popisného charakteru (*VERSION*, *NAME*, apod.). Dále některé slovníky obsahovaly drobné chyby. Například český slovník obsahuje několik odkazů na nedefinovaná pravidla.

Z testovaných slovníků vyčnívá italský slovník. Italské předpony a přípony podle všeho umožňují velkou variabilitu při tvorbě slov. Tím, že se afixy mnohokrát opakují, výborně zafunguje komprese a výsledný slovník je malý. Konverze si však vyžádá zhruba 3 GB operační paměti. Nicméně konverze slovníku je jednorázová operace.

[4] `https://github.com/LibreOffice/dictionaries`

Tabulka 8.1: Velikosti původních a konvertovaných slovníků.

Slovník	**Hunspell slovník**		**Konvertovaný slovník**	
	Velikost (.dic + .aff)	**Kořenů slov**	**Velikost (.bin + .json)**	**Slov**
cs_CZ	2,3 MB + 98 KB	166 564	1,2 MB + 1 KB	3 125 547
sk_SK	3,4 MB + 100 KB	247 005	1,4 MB + 2 KB	2 022 120
de_DE	4,4 MB + 20 KB	258 200	3,3 MB + 2 KB	1 389 554
en_US	0.6 MB + 4 KB	49 524	0,4 MB + 4 KB	123 615
en_GB	1,0 MB + 34 KB	87 140	0,8 MB + 2 KB	221 172
es_ES	0,8 MB + 165 KB	68 226	0,6 MB + 2 KB	710 388
pl_PL	4,6 MB + 249 KB	308 298	2,2 MB + 4 KB	3 765 798
fr_FR	1,1 MB + 260 KB	80 853	1,1 MB + 8 KB	1 159 523
it_IT	1,3 MB + 82 KB	95 185	1,1 MB + 2 KB	34 566 152

Velikost všech slovníků se po konverzi zmenšila. Lze tedy prohlásit, že způsob reprezentace slovníků vyvinutého systému je paměťově efektivnější, než komprese pomocí afixových pravidel, kterou používají slovníky systému Hunspell.

Všechny slovníky byly po převedení zběžně manuálně otestovány na textech v příslušném jazyce. S vybranými slovníky pak byly testovány další aspekty systému, jako výkon a schopnost plnit funkci systému pro kontrolu pravopisu – viz dále.

8.4 Testování výkonu

Pro výkonnostní testy výsledného systému byl, podobně jako u výkonnostních testů prototypů v kapitole 6.1, použit český slovník. Měření bylo také provedeno stejným postupem. Měření byla provedena na prohlížečích Google Chrome a Microsoft Edge. Měřeno bylo:

- načtení slovníku – maximální a klidová paměť[5], čas;
- kontrola 100 000 slov – čas;
- hledání podobných slov, ke slovu „nejnehozpodárnějšími“ – čas.

[5]Klidová velikost haldy po načtení odráží paměť, kterou systém perzistentně drží.

Tabulka 8.2: Výsledky pro knihovnu typo.js na prohlížeči Google Chrome a Microsoft Edge. Průměr pěti měření a směrodatná odchylka.

Prohlížeč	**Načítání**			**Kontrola**	**Napovídání**
	Paměť max [MB]	**Paměť klidová [MB]**	**Trvání [ms]**	**Trvání [ms]**	**Trvání [ms]**
Chrome	372 ± 19	260 ± 0	8561 ± 577	44 ± 2	1841 ± 19[1]
Edge	705 ± 10	627 ± 4	7977 ± 82	190 ± 40	2521 ± 27[1]

[1]Typo.js má napovídání implementované pomocí generování slov (viz 2.3.3). To má za následek, že výkonově selhává u delších slov.

Tabulka 8.3: Výsledky pro vytvořený systém na prohlížeči Google Chrome a Microsoft Edge. Průměr pěti měření a směrodatná odchylka.

Prohlížeč	**Načítání**			**Kontrola**	**Napovídání**
	Paměť max [MB]	**Paměť klidová [MB]**	**Trvání [ms]**	**Trvání [ms]**	**Trvání [ms]**
Chrome	4 ± 0	4 ± 0	26 ± 1	50 ± 40	126 ± 14
Edge	95 ± 4	95 ± 4	40 ± 6	24 ± 13	229 ± 19

Systém byl měřen na ukázkové aplikaci (popsána v kapitole 7.4). Za účelem porovnání jsou přiloženy také výsledky pro knihovnu typo.js. Knihovna typo.js byla měřena v testovací aplikaci vytvořené pro testování prototypů slovníků (viz kapitola 6.1.2). Bylo doměřeno pouze napovídání. Aplikace samy o sobě vytvářejí určité zatížení, které ovlivňuje výsledky. Je ale natolik nízké, že jej můžeme přehlédnout. Obě aplikace měly po načtení, v rámci jednoho prohlížeče, téměř stejnou velikost haldy – Google Chrome zhruba 4 MB a Microsoft Edge zhruba v rozmezí od 90 do 100 MB.

Výsledky měření knihovny typo.js, které byly v kapitole 6.1.2 zhodnoceny jako nevyhovující, jsou uvedeny v tabulce 8.2. Výsledky měření vyvinutého systému pro kontrolu pravopisu jsou uvedeny v tabulce 8.3.

Na obou prohlížečích je načtení téměř okamžité a slovník po načtení prakticky nemá vliv na velikost haldy. Podobně i ověření stovek tisíc slov trvá řádově desítky milisekund. Vyhledání nadprůměrně dlouhého slova trvá nízké stovky milisekund, což je také naprosto dostatečné, vzhledem k tomu, že se jedná o operaci prováděnou pro konkrétní slovo, až na žádost uživatele.

Zajímavé je, že český slovník má velikost 1,24 MB, ale velikost haldy v prohlížeči Google Chrome po načtení slovníku vzroste pouze o desetiny megabajtu. Možnost, že na haldě bylo volné místo, a proto nedošlo k jejímu zvětšení, byla vyvrácena pokusem, při kterém bylo načteno více různých slovníků s celkovou velikostí mnohem větší, než byla velikost haldy. Paměť je tedy zjevně alokována mimo haldu. Můžeme se pouze domnívat, jestli je to způsobeno tím, že se jedná o objekt typu `ArrayBuffer` nebo tím, že se jedná o data přijatá přes Ajaxové volání (`XMLHttpRequest`).

8.5 Funkční testování

Cílem funkčního testování bylo ověřit, že systém plní svoji roli, jako systém pro kontrolu pravopisu. Dílčím úkolem bylo zajistit, aby bylo napovídání slov maximálně přínosné pro uživatele. Dále bude uvedeno několik testovacích případů, které předvedou, jak systém reaguje na typické scénáře.

Triviální překlepy Jako příklad bude uvedeno několik slov s triviálními překlepy a slova, která systém napověděl jako možné opravy. Byl použit český slovník. Slova s překlepy:

- buldozr (buldozer) – buldozer, buldoci, buldoka, buldok, buldoku;
- pžemyslovský (přemyslovský) – přemyslovský, přemyslovsky, přemyslovské, přemyslovským, přemyslovská;
- doložeým (doloženým) – doloženým, doložím, dovoženým, dolože, položeným;
- dolni (dolní) – dotni, dozni, Dolfi, dožni, dolně;
- sitý (sytý) – bitý, lity, pitý, sbitý, setý.

Jak je vidět na posledních dvou příkladech, někdy je možností příliš mnoho a správné slovo se mezi napovězená slova nedostane. Jejich počet je totiž omezen na 5. Chybějící diakritika, záměna měkkého a tvrdého *i*, záměna *s* a *z* apod., by mezi napovězenými slovy mohla být upřednostněna použitím mechanismu nahrazovací tabulky, protože se zjevně jedná o časté chyby. Nicméně použitý český slovník této možnosti nevyužívá.

Napovídání u kratších slov Při testování bylo zjištěno, že u kratších slov byl systém někdy schopen napovědět i zcela odlišná slova, což nepůsobilo dobře. Proto se přistoupilo k možnosti určit maximální Levenshteinovu vzdálenost dynamicky, podle délky slova. U slov o jednom znaku nejsou podobná slova hledána vůbec, u slov do pěti znaků maximálně do vzdálenosti 1, do osmi znaků 2 a nad osm znaků 3. Příklady:

- ť – *<nic>*;
- bžž – baž, běž;
- lokoomotiav (lokomotiva) – lokomotiv, lokomotiva, lokomotivu, lokomotivo, lokomotivy.

Chybná velikost znaků Systém jako první napovídá slova, která se liší pouze ve velikosti znaků. Příklady:

- jitka (Jitka) – Jitka, pitka, Řitka, Jitko;
- java (Java) – Java, Jana, Jasa, Jav;
- who (WHO) – WHO, WMO, ho, Wh;
- wysiwyg (WYSIWYG) – WYSIWYG;
- wysiwyk (WYSIWYG) – WYSIWYG.

Poslední příklad ukazuje, že systém si je schopen poradit i v situaci, kdy má slovo chybnou velikost znaků v kombinaci s překlepem.

Zachování velikosti znaků Systém se při napovídání snaží zachovat velikost znaků, pokud je to možné. Příklady:

- buldozr (buldozer) – buldozer, buldoci, buldoka, buldok, buldoku;
- Buldozr (Buldozer) – Buldozer, Buldoci, Buldoka, Buldok, Buldoku;
- BULDOZR (BULDOZER) – BULDOZER, BULDOCI, BULDOKA, BULDOK, BULDOKU.

Podpora pro ostrá S Německý jazyk má specialitu, která se týká ostrého S. Malé ostré S se zapisuje jako „ß“, zatímco velké jako „SS“. To vyžaduje zvláštní zacházení při kontrole i napovídání. Na německém slovníku byly otestovány varianty slova Straße:

- Straße – *<správně>*;
- Strasse – Straße, Strapse, Stresse, Trasse, Rasse;
- STRAßE – Straße, STRASSE, STRAFE, STRAUSSE, STRASSEN;
- STRASSE – *<správně>*;
- straße – Straße, Strafe, Strauße, Straßen.

Obecně kvalita kontroly pravopisu velmi závisí na kvalitě slovníku. Kvalita slovníku je věc, se kterou systém nic nenadělá. Nedá se však říci, že větší slovník znamená kvalitnější kontrolu. Pokud slovník obsahuje velmi vzácná slova, specifické pojmy, názvy a podobně, zvyšuje se tím riziko, že překlep nebude rozpoznán, protože se zrovna „trefil“ do něčeho, co je ve slovníku. Podobně napovídání může být „zaplaveno“ exotickými slovy, které uživatel ani nezná.

Bylo vyvinuto velké úsilí, aby byla napovídaná slova pro uživatele relevantní. Výraznějšího zlepšení by bylo možné dosáhnout už jen pokud by byly k dispozici dodatečná data. Například frekvenční slovník, který by slovům ve slovníku přiřadil relativní četnost výskytu v daném jazyce. Pak by bylo možné výsledky řadit a v případech, kdy je možností příliš mnoho, zajistit, aby se mezi výsledky dostala slova, která jsou více pravděpodobná.

8.6 Shrnutí testování

Projekty, které bylo z jejich podstaty možné otestovat jednotkově, používají knihovnu JUnit pro tvorbu jednotkových testů. Jednotkovým testům byla po celou dobu vývoje věnována velká pozornost. U klíčových částí bylo naměřeno pokrytí přes 90 %.

Dále byly vytvořeny automatizované testy klientské části – skriptu vygenerovaného GWT, který lze nasadit jako HTML5 Web Worker. Pro jejich tvorbu byl použit Robot Framework. Testy zvládnou ověřit, že se skript vygeneroval správně a funguje, dále pak ověřují jeho rozhraní a načítání slovníků.

Systém byl otestován Hunspell slovníky devíti významných evropských jazyků. Slovníky byly převedeny do formátu vytvořeného systému pro kontrolu pravopisu a byla zhodnocena jejich velikost. Vlastní formát se ukázal jako paměťově efektivnější. Podporu Hunspellu lze zhodnotit jako dostatečnou, protože žádný ze slovníků neobsahoval příkazy, se kterými by systém neuměl pracovat.

Testování výkonu bylo zaměřeno na rychlost a paměťovou náročnost. Testy byly provedeny na prohlížečích Google Chrome a Microsoft Edge. Bylo změřeno, že systém s načteným slovníkem o milionech slov zabírá v paměti několik málo megabajtů. Ověření stovek tisíc slov trvá řádově desítky milisekund a vyhledání podobných slov nízké stovky milisekund.

Funkční testování ověřilo, že systém plní svoji roli, jako systém pro kontrolu pravopisu. Bylo vyvinuto velké úsilí, aby byla napovídaná slova pro uživatele relevantní. Hlavní roli však vždy hraje kvalita slovníku.

9 Závěr

Po první, úvodní kapitole byl ve druhé kapitole čtenář seznámen s problematikou systémů pro kontrolu pravopisu. Probrány byly struktury vhodné pro reprezentaci slovníků a algoritmy pro hledání podobných slov. Ve třetí kapitole byl představen systém pro kontrolu pravopisu s názvem Hunspell. Byly popsány obecné mechanismy, na kterých je založen a popsán byl také formát jeho slovníků. Čtvrtá kapitola probírala potřebné JavaScriptové technologie, v souvislosti s implementací systému pro kontrolu pravopisu. Probrány byly také omezení, se kterými je potřeba na této platformě počítat. Pátá kapitola představila framework Google Web Toolkit (GWT). Ten umožňuje vyvíjet webové frontendové aplikace v Javě.

Praktickou část zahájily experimenty s různými reprezentacemi slovníků, které byly popsány v šesté kapitole. Prototypy ukázaly, že přímočaré implementace struktur v JavaScriptu selhávají, a to především z pohledu paměťové náročnosti. Z experimentů nakonec vzešel základní návrh systému. Jako nejlepší řešení se ukázalo dopředu zakódovat optimalizovanou trii do bajtového pole a v klientském JavaScriptu toto pole s trií pouze načíst. Trie má také tu výhodu, že se jedná o prefixový strom, což se hodí například při hledání podobných slov.

V sedmé kapitole byla popsána finální implementace vytvořeného systému pro kontrolu pravopisu a na ukázkové aplikaci s editorem byla předvedena jeho integrace. Systém samotný lze rozdělit na dvě logické části, a sice klientskou část provozovanou uvnitř HTML5 Web Workera a nástroj, který umožňuje převod slovníků systému Hunspell do vlastního binárního formátu. Klientská část je postavená na frameworku GWT.

Poslední, osmá kapitola se věnovala ověřování kvality vytvořeného systému. Kromě obvyklých jednotkových testů a testů klientské části byla pozornost věnována také funkčnímu testování. Dále byl systém otestován devíti slovníky důležitých evropských jazyků. Systém byl také podroben testům výkonu, které potvrdily, že jeho paměťová náročnost je naprosto minimální. Slovník s miliony slov zabírá v paměti pouze několik málo megabajtů. Výkonnostní testy také dokázaly, že je systém dostatečně rychlý. Že systém může fungovat v reálném čase dokazuje i ukázková aplikace s editorem.

Všechny body zadání byly splněny, systém byl komplexně otestován a je připraven na produkční nasazení.

Přehled zkratek

API . . . . . Application Programming Interface

BVS . . . . . Binární vyhledávací strom

DOM . . . . . Document Object Model

GWT . . . . . Google Web Toolkit

HTTP . . . . Hypertext Transfer Protocol

JAR . . . . . Java Archive

JNI . . . . . Java Native Interface

JSNI . . . . . JavaScript Native Interface

JSON . . . . JavaScript Object Notation

RPC . . . . . Remote Procedure Call

TVS . . . . . Ternární vyhledávací strom

URL . . . . . Uniform Resource Locator

WYSIWYG . „What you see is what you get“

Literatura

[1] Bentley, J. L. – Sedgewick, R. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, s. 360–369, 1997.

[2] Beran, V. *Typografický mauál, IV. vydání.* Kafka design, 2005.

[3] Bocek, T. – Hunt, E. – Stiller, B. *Fast similarity search in large dictionaries. TECHNICAL REPORT – No. ifi-2007.02.* University of Zurich, 2007.

[4] Damerau, F. J. A technique for computer detection and correction of spelling errors. *Communications of the ACM.* 1964, 7, 3, s. 171–176.

[5] Ecma. *ECMA-262 6th edition, ECMAScript 2015 Language Specification* [online]. Ecma International, 2015. [cit. 2020-04-22]. Dostupné z: `https://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`.

[6] Frantíková, I. *MORFOLOGICKÁ TYPOLOGIE JAZYKŮ – AGLUTINAČNÍ JAZYKY* [online]. 2008. [cit. 2020-04-14]. Dostupné z: `http://www.indianskejazyky.cz/files/Aglutinacni_jazyky.pdf`.

[7] Garbe, W. *1000x Faster Spelling Correction algorithm* [online]. 2012. [cit. 2020-04-15]. Dostupné z: `https://medium.com/@wolfgarbe/1000x-faster-spelling-correction-algorithm-2012-8701fcd87a5f`.

[8] GWT contributors. *Developer's Guide* [online]. . [cit. 2020-04-22]. Dostupné z: `http://www.gwtproject.org/doc/latest/DevGuide.html`.

[9] GWT contributors. *The GWT Release Notes* [online]. . [cit. 2020-04-22]. Dostupné z: `http://www.gwtproject.org/release-notes.html`.

[10] Havránek, B. *Slovník spisovného jazyka českého IVIII.* Praha: Academia, 1989. ISBN: 12/2-8816-21-037-89.

[11] Heinz, S. – Zobel, J. Performance of data structures for small sets of strings. In *Australian Computer Science Communications*, 24, s. 87–94. Australian Computer Society, Inc., 2002.

[12] JetBrains. *JetBrains Space – IntelliJ IDEA 2020.1 – Spelling* [online]. JetBrains s.r.o., 2020. [cit. 2020-04-15]. Dostupné z: `https://www.jetbrains.com/help/idea/spelling.html`.

[13] Levenshtein, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*. February 1966, 10, s. 707.

[14] Mareš, M. – Valla, T. *Průvodce labyrintem algoritmů*. CZ. NIC, zspo, 2017. ISBN tištěné verze: 978-80-88168-19-5.

[15] Martins, B. – Silva, M. J. Spelling correction for search engine queries. In *International Conference on Natural Language Processing (in Spain)*, s. 372–383. Springer, 2004.

[16] MDN contributors. *MDN web docs / Concurrency model and the event loop* [online]. 2020. [cit. 2020-04-22]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`.

[17] MDN contributors. *MDN web docs / IndexedDB / API Basic concepts* [online]. 2019. [cit. 2020-04-22]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB`.

[18] MDN contributors. *MDN web docs / Web Workers API* [online]. 2019. [cit. 2020-04-22]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API`.

[19] MDN contributors. *MDN web docs / Web Workers API / The structured clone algorithm* [online]. 2020. [cit. 2020-04-22]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm`.

[20] Németh, L. – contributors. *Hunspell - About* [online]. 2020. [cit. 2020-04-15]. Dostupné z: `https://hunspell.github.io`.

[21] Németh, L. – contributors. *hunspell(4) - Linux man page* [online]. 2020. [cit. 2020-04-15]. Dostupné z: `https://linux.die.net/man/4/hunspell`.

[22] Norvig, P. *How to Write a Spelling Corrector* [online]. 2016. [cit. 2020-04-15]. Dostupné z: `https://norvig.com/spell-correct.html`.

[23] Oracle. *Java Releases* [online]. Oracle Corporation. [cit. 2020-04-22]. Dostupné z: `https://java.com/en/download/faq/release_dates.xml`.

[24] Petkevič, V. *Kontrola české gramatiky (český grammar checker)*. Univerzita Karlova, Filozofická fakulta, 2014.

[25] Salifou, L. – Naroua, H. Design of A Spell Corrector For Hausa Language. *International Journal of Computational Linguistics (IJCL), 5(2), 14-26.* 2014.

[26] Sleator, D. D. – Tarjan, R. E. Self-Adjusting Binary Search Trees. *J. ACM*. July 1985, 32, 3, s. 652–686. ISSN 0004-5411. doi: 10.1145/3828.3835.

[27] Smeets, B. – Boness, U. – Bankras, R. *Beginning Google Web Toolkit*. Apress, 2008. ISBN: 978-1-4302-1032-0.

[28] Sterbenz, C. *Only 10 Words Make Up 25% Of The English Language* [online]. Business Insider, 2013. [cit. 2020-04-15]. Dostupné z: `https://www.businessinsider.com/zipfs-law-and-the-most-common-words-in-english-2013-10`.

[29] Svobodová, I. *Morfologie současného portugalského jazyka I.* Brno: Masarykova univerzita, 2014. ISBN: 978-80-210-7008-0.

[30] Thenmozhi, M. – Srimathi, H. An analysis on the performance of tree and trie based dictionary implementations with different data usage models. *Indian Journal of Science and Technology*. 2015, 8, 4, s. 364.

[31] Tian, X. et al. The Performance Optimization of Hadoop during Mining Online Education Packets for Malware Detection. In *2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, s. 305–309. IEEE, 2016.

[32] Wang, J. et al. Hashing for Similarity Search: A Survey. *CoRR*. 2014, abs/1408.2927.

[33] Williams, H. E. – Zobel, J. – Heinz, S. Self-adjusting trees in practice for large text collections. *Software: Practice and Experience*. 2001, 31, 10, s. 925–939.

[34] Zeunert, M. *How much memory do JavaScript arrays take up in Chrome?* [online]. 2016. [cit. 2020-04-22]. Dostupné z: `https://www.mattzeunert.com/2016/07/24/javascript-array-object-sizes.html`.

[35] Zeunert, M. *Understanding the size of an object in Chrome/V8* [online]. 2017. [cit. 2020-04-22]. Dostupné z: `https://www.mattzeunert.com/2017/03/29/v8-object-size.html`.

[36] Zimmer, B. *THE CUPERTINO EFFECT* [online]. 2006. [cit. 2020-04-14]. Dostupné z: `http://itre.cis.upenn.edu/~myl/languagelog/archives/002911.html`.

Přílohy

Obsah DVD

Přiložené DVD obsahuje text práce a všechny vytvořené projekty. Kde je to potřeba, projekty obsahují soubor `readme.md` s instrukcemi pro sestavení, nasazení apod.

Adresářová struktura

- `/`
  - `spellchecker` ..... Adresář s projekty
    - `spellchecker-core` ..... Jádro systému
    - `spellchecker-parser-hunspell` ..... Parser Hunspell slovníků
    - `spellchecker-worker` ..... Klientská část
    - `spellchecker-worker-api` ..... API workera
    - `spellchecker-cli` ..... CLI nástroj
    - `app-performance-tests` ..... Aplikace s prototypy slovníků
    - `app-worker-test-page` ..... Aplikace pro testování workera
    - `app-editor` ..... Ukázková aplikace s editorem
    - `dictionaries` ..... Slovníky používané v rámci této práce
  - `poster` ..... Adresář posterem
  - `thesis` ..... Adresář s diplomovou prací
    - `data` ..... Veškerá data z měření výkonu
    - `latex` ..... Zdrojové soubory diplomové práce

Uživatelská příručka

Předpoklady
Nejprve je nutné mít nainstalovaný správný software:

- JDK 8 (ne starší, ne novější),
- Maven 3.

Další kroky
Prvním krokem by mělo být jít do adresáře `spellchecker` a spustit příkaz `mvn install`. Tím dojde ke spuštění testů a sestavení všech artefaktů. Poté je možné prozkoumat jednotlivé projekty:

- ukázkovou aplikaci s editorem,
- aplikaci s prototypy slovníků,
- aplikaci pro testování Web Workera,
- nástroj pro příkazovou řádku.

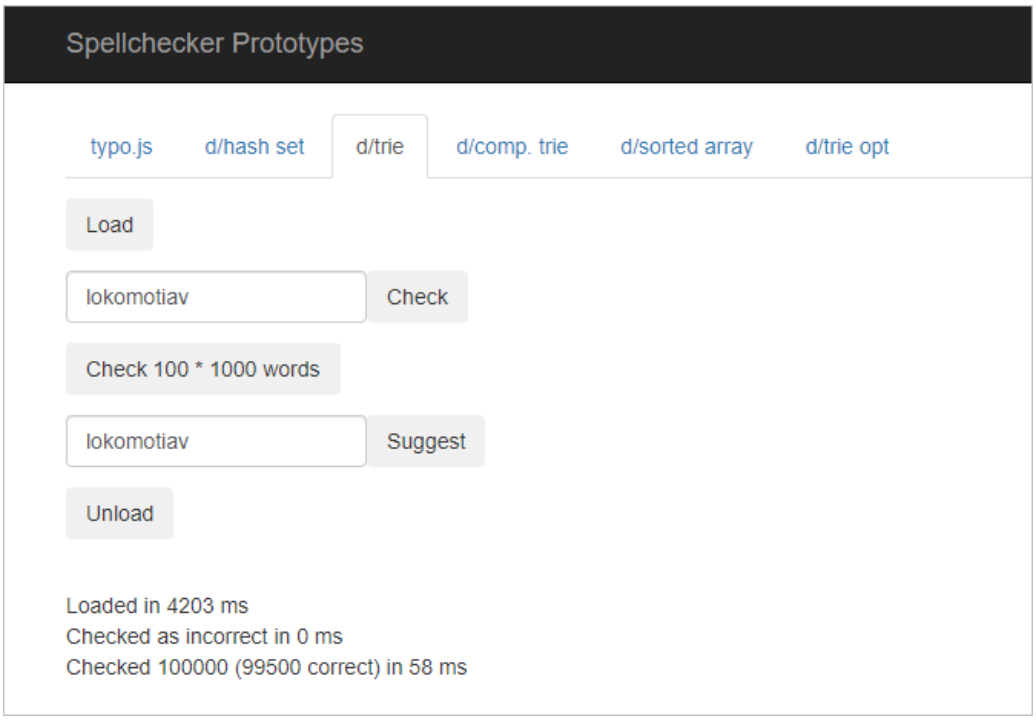
Ukázková aplikace

Ukázková aplikace s editorem byla popsána v kapitole 7.4 a její zdrojové kódy se nacházejí v projektu `app-editor`.

Slovníky Aplikace má vestavěný český, anglický a německý slovník. Případné přidání dalšího slovníku by znamenalo:

- přidání slovníku do `src/main/webapp/dictionaries`,
- přidání výchozího textu do `src/main/webapp/examples`,
- rozšíření `index.html` o další panel,
- inicializace panelu ve třídě `SampleApplicationEntryPoint`.

Nasazení Ve fázi package je standardně produkován WAR s webovou aplikací. Ten je možné nasadit na libovolný aplikační server na Javě 8 (testován byl Tomcat 7.2.2 a Tomcat 8.5.38). Žádné speciální nastavení není potřeba.



Obrázek 9.1: Aplikace pro testování prototypů

Build script dále obsahuje `tomcat7-maven-plugin`, který lze využít pro lokální nasazení na embedovaný Tomcat 7.2.2. Sestavení WAR, spuštění serveru a zavedení aplikace se tak provede jediným příkazem:

```
mvn tomcat7:run-war
```

Aplikace bude dostupná na adrese: `http://localhost:8080/`

Aplikace s prototypy slovníků

Projekt `app-performance-tests` s GWT aplikací s implementací prototypů slovníků a nasazeným typo.js. Byla použita pro měření v rámci kapitoly 6.

Vzhled aplikace ukazuje obrázek 9.1. Obsahuje panel pro každou testovanou metodu/prototyp. Logy jsou vypisovány ve spodní části a rovněž do konzole. Před operacemi jako kontrola slova je nutné nejprve načíst slovník (tlačítko „Load“).

Pro sestavení a nasazení platí stejný postup, jako u ukázkové aplikace.

Aplikace pro testování workera

Projekt `app-worker-test-page` obsahuje jednoduchou webovou aplikaci, pro snadné manuální testování Web Workera. Jedná se zároveň o příklad

nejjednodušší integrace. Byla zmíněna v rámci kapitoly 7.3.4.

Před spuštěním je nutné do adresáře nakopírovat sestavený skript pro Web Workera. K tomu je připraven skript `do-copy-worker-js.bat`.

Dále ke svému fungování vyžaduje nějaký HTTP server. Lze využít například Python. Pro ten je připraven skript `do-run-python-http-server.bat`. Aplikace bude dostupná na adrese: `http://localhost:8000/index.html`

Nástroj pro příkazovou řádku

Projekt `spellchecker-cli` obsahuje nástroj pro práci se slovníky, který lze obsluhovat z příkazové řádky. Byl zmíněn v rámci kapitoly 7.2.3. Umožňuje převod slovníků z formátu Hunspell do binárního formátu systému pro kontrolu pravopisu. Hlavním artefaktem je soubor `spellchecker-cli-<verze>--jar-with-dependencies.jar`, který je samostatně spustitelnou Java aplikací.
Spouští se následujícím způsobem:

```
java -jar <název.jar> <příkaz> <par. příkazu 1>... <par. pří
    kazu n>
```

Formát a množství parametrů příkazu je závislé na daném příkazu. Aktuálně jediným příkazem je `convert`, pro již zmíněný převod. Použití může být například:

```
java -jar <název.jar> convert -i en_US.aff en_US.dic -o en_US
```

Pokud převod proběhne úspěšně, dojde k vytvoření souborů `en_US.bin` a `en_US.json`.

Spuštění bez parametrů má za následek vypsání nápovědy:

```
Usage:
java -jar <filename.jar> convert -i <input 1> <input 2> -o <
    output>

Files <output>.bin and <output>.json will be created.
```

Programátorská příručka

Všechny projekty používají maven. Vzájemné vztahy maven projektů jsou vidět na obrázku 9.2. Vytvořeny byly následující projekty:

`spellchecker-core` – jádro systému, obsahuje implementaci slovníků, provádí kontrolu pravopisu a napovídání;

`spellchecker-parser-hunspell` – modul s Hunspell parserem, umožňuje načítat slovníky ve formátu Hunspell;

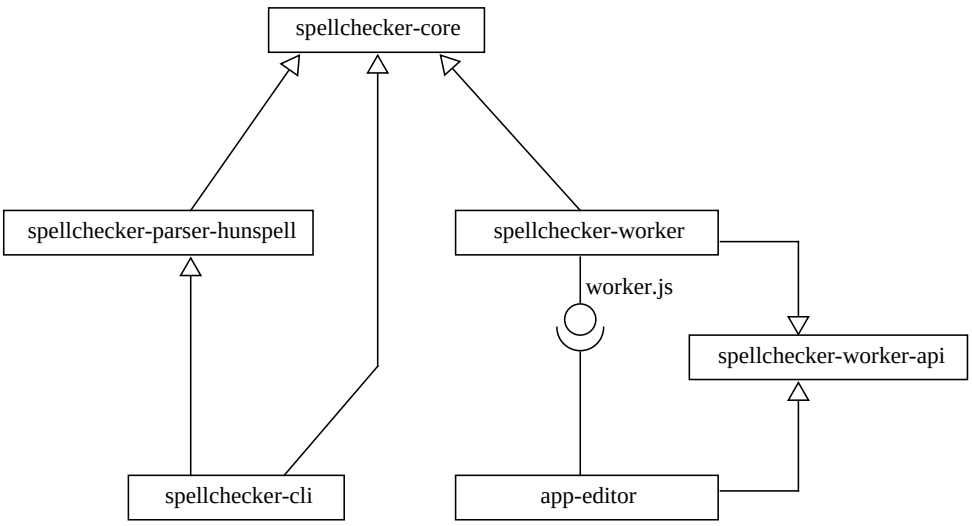
`spellchecker-cli` – nástroj pro konverzi slovníků z formátu Hunspell do vlastního formátu;

`spellchecker-worker` – kompiluje systém do JavaScriptu, výstupem je samostatně spustitelný skript, který lze provozovat jako HTML5 Web Worker;

`spellchecker-worker-api` – definuje formát zpráv pro komunikaci s workerem;

`app-editor` – ukázková GWT aplikace s editorem a integrovaným systémem pro kontrolu pravopisu;

`app-performance-tests` – aplikace s prototypy slovníků, která byla použita v rámci kapitoly 6.



Obrázek 9.2: Diagram závislostí projektů.

Dále se ve stejném nadřazeném adresáři vyskytují následující adresáře, které už ale nejsou maven projekty:

app-worker-test-page – webová stránka pro testování workera, obsahuje také testy klientské části (viz dále);

dictionaries – slovníky používané v rámci této práce, obsahuje také slovníky vybraných evropských jazyků, které byly testovány v rámci kapitoly 8.3.

Implementací se více zabývala kapitola 7, kde je popsáno členění systému, jednotlivé projekty a nejdůležitější třídy. Dále je k dispozici javadoc, který generují jednotlivé projekty.

Testy klientské části

Testování klientské části bylo probíráno v kapitole 8.2. Pro implementaci testů byl použit Robot Framework[1]. Jejich zprovoznění se skládá z následujících kroků:

1. Instalace python & pip
2. Instalace Robot Framework

```
pip install robotframework
pip install robotframework-seleniumlibrary
```

3. Instalace driveru prohlížeče

   Například Chrome driver[2]. Driver uložit na *Path* nebo umístit do adresáře s testy.

4. Spuštění testů

   Spuštění všech testů:

```
robot ./
```

   Pro nastavení složky s výstupy lze použít '–outputdir':

```
robot --outputdir ./output ./
```

   Spuštění konkrétního testu:

```
robot --outputdir ./output ./test.robot
```

[1] http://robotframework.org

[2] https://sites.google.com/a/chromium.org/chromedriver/downloads

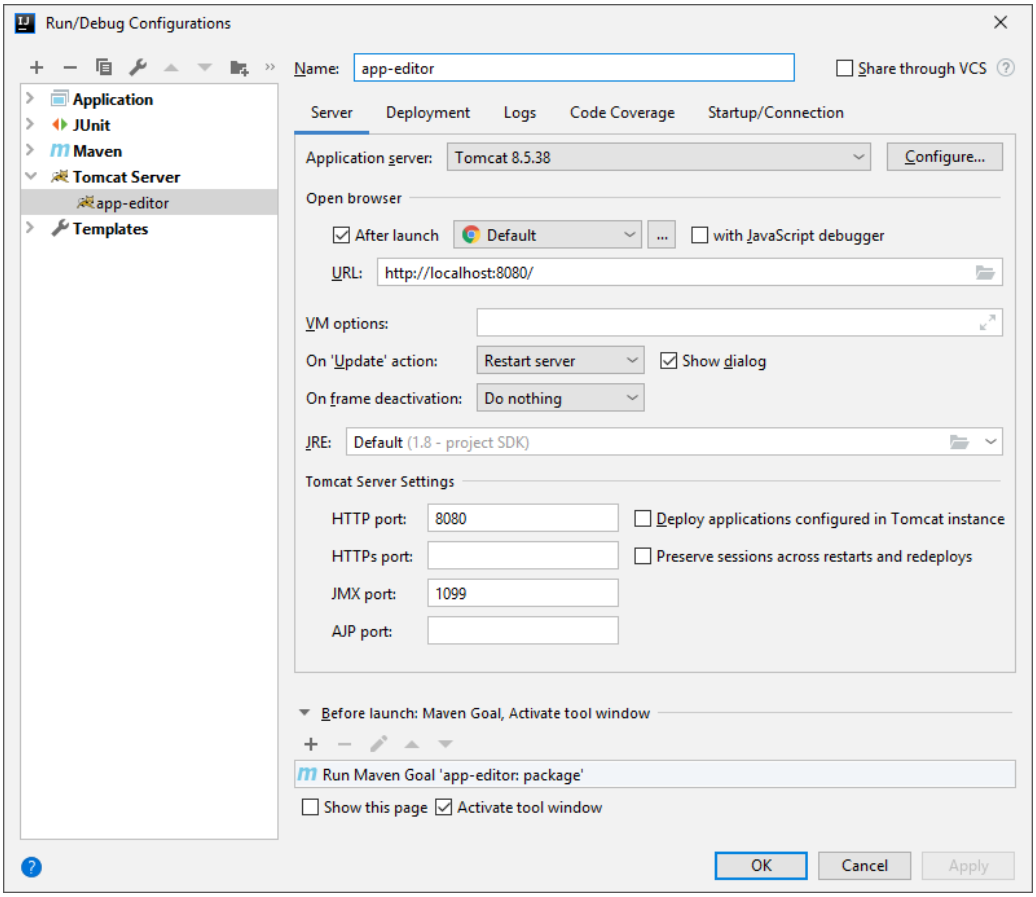
Vývojové prostředí

Všechny projekty používají maven a v adresáři `spellchecker` existuje i nadřazený `pom.xml`. Neměl by tedy být problém importovat celý nadřazený maven projekt se všemi projekty do libovolného vývojového prostředí. Import spolehlivě funguje minimálně u vývojového prostředí IntelliJ IDEA.

Lokální nasazení webových aplikací

Způsob, jak nasadit webové aplikace `app-editor` a `app-performance-tests` již byl popsán v uživatelské příručce. Při vývoji ale bývá pohodlnější použít nástroje pro nasazování, které poskytuje použité vývojové prostředí.

V případě ukázkové aplikace (`app-editor`) je nutné před nasazením projekt sestavit mavenem (např. příkazem `mvn package`). Bez toho nedojde k nakopírování skriptu pro Web Worker, který vygenerovalo GWT. To je nutné v běhové konfiguraci explicitně nastavit. Příklad konfigurace je zobrazen na obrázku 9.3.



Obrázek 9.3: Běhová konfigurace ukázkové aplikace v IntelliJ IDEA.