Faculty of Computer Science and Engineering,
Ss. Cyril and Methodius University

Project Report on "WordsInFocus"- A Digital Dictionary

Jana Ristovska

ID Number 203186

jana.ristovska@students.finki.ukim.mk

Ekaterina Sazdova

ID Number 201063

ekaterina.sazdova@students.finki.ukim.mk

Advanced Web Design

Prof. Dr. Boban Joksimoski

10 September 2023
Skopje, North Macedonia

Faculty of Computer Science and Engineering,
Ss. Cyril and Methodius University

Abstract

"WordsInFocus" is a digital dictionary application designed to modernize the documentation of our native language. This team project, made by Jana Ristovska and Ekaterina Sazdova, was developed as part of our university coursework for the subject Advanced Web Deisgn. The main goal of this university project was to create a web application dictionary that is a modernized version of the existing digital dictionary which would be easier to use.

The project used multiple of technologies: Python for web scraping, Docker for PostgreSQL database setup, Java Spring for the backend, and Vue.js for the frontend. The web scraping component utilized Chrome WebDriver to extract data from an existing online dictionary, extracting essential information such as words, word types, definitions, and usage examples. This data was then organized into JSON files by the web scraper, with each file representing a letter of the alphabet, making an easy data transfer to our PostgreSQL database.

The backend of "WordsInFocus" was built using Java Spring, following a structured architecture comprising model, controller, repository, and service layers. We introduced three main entities into our database model: 'Letter,' 'Word,' and 'Definition.' Each 'Letter' entity encapsulated a list of words, while each 'Word' entity included a list of definitions. This structured approach enabled efficient data retrieval and manipulation.

On the frontend side, Vue.js was employed to create an intuitive user interface. The application comprises components like 'Header,' 'Footer,' 'Letters,' 'Word,' 'LetterWithWords,' and 'PageNotFound.' The routing system, managed by 'router.js,' allows users to navigate smoothly through the application.

This project report delves into the technologies, and architectural decisions that contributed to the realization of "WordsInFocus."

10 September 2023
Skopje, North Macedonia

Table of Contents

Introduction

Language is not just a meant for communication, it is something that carries with itself the importance of culture, identity, and heritage. In a time where technology is evolving very fast, preserving our language and helping our native language to grow has become even more important than ever. "WordsInFocus" is created as part of our Advanced Web Design course to achieve these goals. In a world where languages evolve and adapt, it is very important that we have a tool that not only records our linguistic history but also helps its continued growth, bridging the gap between tradition and technology.

Project Description

Project Objectives

The primary objectives of "WordsInFocus" were:

- Modernize the Language Documentation: To create an updated and accessible digital dictionary of our native language.
- Ease of Access: To provide a user-friendly platform where individuals can easily access and explore the language.
- Learn Technologies: To gain practical experience in using a variety of modern technologies, including web scraping, data storage, backend development, and frontend design.

Project Components

The "WordsInFocus" project can be broken down into the following key components:

- Web Scraping with Python (and Data Organization with JSON)
- Database Setup with Docker and PostgreSQL
- Backend Development with Java Spring
- Frontend Design with Vue.js

Web Scraping with Python

We used Python, a versatile programming language, to build a web scraper capable of extracting data from the existing online dictionary for the Macedonian language - drmj.eu. The scraper navigates through the dictionary website, collecting essential information such as words, their types, definitions, and usage examples.

To efficiently manage the big amount of data collected during web scraping, we structured it into JSON files. Each file is dedicated to a specific letter of the alphabet, providing a logical and manageable data storage solution.

Database Setup with Docker and PostgreSQL

An important part of our project is in the storage and retrieval of the words data. We created a local PostgreSQL database using Docker, which offers a stable and scalable environment for data transfer and manipulation in the other parts of the application.

Backend Development with Java Spring

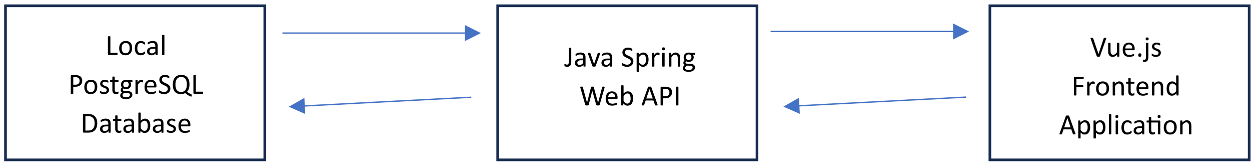
The Java Spring framework was used for the needs of our backend development. Our backend is structured into multiple layers, including models layer, web layer (controllers), persistance layer (repositories), and service layer. All of these layers together help create a good flow of data from the database to the frontend where it can be used.

Frontend Design with Vue.js

Vue.js was chosen to create a dynamic and user-friendly frontend interface. Our application is composed of multiple components, such as 'Header,' 'Footer,' 'Letters,' 'Word,' 'LetterWithWords,' and 'PageNotFound.' These components work together to provide a responsive and immersive user experience.

In the following parts of this project report, we will go into the technical details of each project part.

System Architecture and Implementation



The rest of this section will detail the implementation of all of the code located in the src folder in the repository:

Web Scraping

This part of the project is stored in the src/drmj-scraper folder of the git repository. In it is included the main.py file and the chrome driver.

The chromedriver is just an exe file of the latest version of the chrome web driver at the time of making this app. This file is needed to make the python script actually work so it can open chrome, open the drmj.eu page and scrape the data from it.

The main.py file is the actual web scraping script. It consists of multiple functions:

Main function

```
def main():
    for letter in letters:
        browser = webdriver.Chrome()
        browser.get('http://drmj.eu/letter/' + letter)
        fp = open("..\\WordsInFocusAPI\\src\\main\\resources\\wordsInJson\\" +
letter + '.json', 'w')
        fp.write("[")
        scrape_letter(browser, letter)

        browser.close()
```

This function Is the main function and it is the first function that is called in the execution of the script. It creates a browser variable, using the chrome web driver. For each letter of the alphabet it navigates to the appropriate route in order to retrieve its data. Then it navigates to a folder in the backend where the json files will be stored and opens the file that is named after the letter. If the file does not exist it creates it, if it does exist it rewrites it. It then starts writing some characters that will help format the json file properly. Then, it calls the function scrape_letter, passing in the letter it needs to scrape and the browser variable as arguments. After this loop finishes for each letter it then closes the browser.

Scrape letter function

```
def scrape_letter(browser, letter):
    fp = open("..\\WordsInFocusAPI\\src\\main\\resources\\wordsInJson\\" + letter
+ '.json', 'w')
    ranges = Select(browser.find_element(By.XPATH, "//select[@name='ranges']"))

    for index in range(len(ranges.options)):
        ranges = Select(browser.find_element(By.XPATH,
"//select[@name='ranges']"))
        ranges.select_by_index(index)
        words(browser, letter)

    fp.write("]")
```

As mentioned before, this function is called for every letter in the Macedonian alphabet. It opens the appropriate json file located in the backend API resources and continues to write to it. It then finds the word ranges element and iterates through all of the ranges for this letter, calling the words function for every word range. At the end it writes the closing character to the json file.

Words function

```
def words(browser, letter):
    lexems = Select(browser.find_element(By.XPATH, "//select[@name='lexems']"))
    words = lexems.options

    for index in range(len(lexems.options)-1):
        lexems = Select(browser.find_element(By.XPATH,
"//select[@name='lexems']"))
        lexems.select_by_index(index+1)
        meanings(browser, letter, words[index].accessible_name,
len(lexems.options), index)
```

This function finds the lexems element and iterates through all of the index in the range, calling the meanings function in each iteration. It passes the browser, letter, words name, options and index to that function.

Meanings function

```
def meanings(browser, letter, word, lexemsLen, index):
    raw_html = browser.page_source
    html = BeautifulSoup(raw_html, "html.parser")
    name = html.select_one(".lexem").text.strip()
    fp = open("..\\WordsInFocusAPI\\src\\main\\resources\\wordsInJson\\" + letter
+ '.json', 'a', encoding="utf-8")
    fp.write('\n\t{')
    fp.write('\n\t\t\"word\": \"'+word+'\",')
    fp.write('\n\t\t\"name\": \"'+name+'\",')

    type = html.select_one(".grammar")
    formattedType = type.select_one("i").text.strip()
    fp.write('\n\t\t\"type\": \"'+formattedType+'\",')

    definitions = html.select(".definition")
    fp.write('\n\t\t\"definitions\": [')
    for i in range(0, len(definitions)):
        m = definitions[i].select_one(".meaning")
        meaning = m.text.replace(".","").replace("\n","").replace("\t","")
        formattedMeaning = '' .join((z for z in meaning if not
z.isdigit())).strip()

        e = definitions[i].select_one(".example")
        if(e == None):
            formattedExample = "No example"
        else:
            example = e.text.replace(".","").replace("\n","").replace("\t","")
            formattedExample = '' .join((z for z in example if not
z.isdigit())).strip()
        fp.write("\n\t\t\t{")
        fp.write('\n\t\t\t\t\"' + "definition_name" + '\": \"' + formattedMeaning
+ '\",')
        fp.write('\n\t\t\t\t\"' + "definition_example" + '\": \"' +
formattedExample + '\"')
        if((i)==len(definitions)-1):
            fp.write("\n\t\t\t}")
        else:
            fp.write("\n\t\t\t},")

    fp.write('\n\t\t]')
    fp.write('\n\t},\n')
```

This is the largest function that is in charge of writing all of the detailed information about the words, writing all of the definitions for the word and examples for each of them. It also formats the json files to be valid JSON and be readable.

Together all of these functions help scrape through the entire dictionary and write all the data to json files so it can later be used more easily and stored effectively into a database. Each json file that comes as a result is named **{letter}.json** and contains a list of all of the words, their type, their meanings and finally their definitions.

Data Organization

Although storing the data in json files is fine, storing it in a proper database is much better and more efficient. Because of this, we created a **PostgreSQL database** where we store all of the dictionary information.

To create the database, we needed to make a **docker container** where it can be stored. We decided on creating the container locally on our machines as it was an easier hosting option. To do this, we used this command in the CLI:

```
docker run --name local-psql -v local_psql_data:/var/lib/postgresql/data -p 54320:5432 -e POSTGRES_PASSWORD=my_password -d postgres
```

This automatically uses the postgres image for the type of machine we have to make a docker container for the postgres database. The command also specifies the connection details such as port number, password and where the container will be.

Connecting the backend application of the database was possible by configuring the application settings file and data sources section in IntelliJ.

This is part of the **application.properties** file where we specified the connection settings.

```
spring.datasource.url=jdbc:postgresql://localhost:54320/postgres
spring.datasource.username=postgres
spring.datasource.password=my_password
```

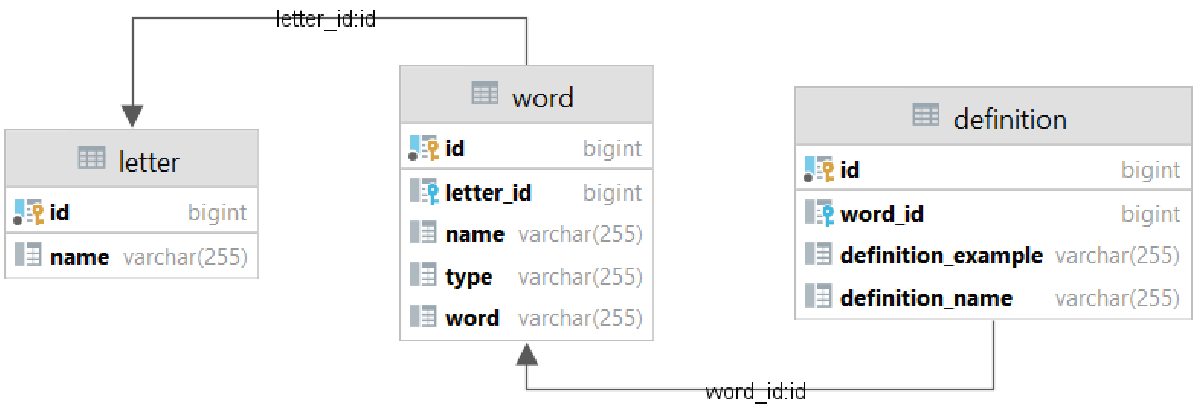
To map the json data to the database tables needed, we used the **backend models** (will be explained more in the next section) and an **Object Mapper.** This is the Java code that we wrote for this purpose:

```
ObjectMapper objectMapper = new ObjectMapper();
List<String> letters = new ArrayList<>(Arrays.asList(
        "а", "б", "в", "г", "д", "ѓ", "е",
        "ж", "з", "ѕ", "и", "ј", "к", "л",
        "љ", "м", "н", "њ", "о", "п", "р",
        "с", "т", "ќ", "у", "ф", "х", "ц",
        "ч", "џ", "ш"));

for (String letterValue:letters) {
    Letter letter = objectMapper
            .readValue(new File("src/main/resources/wordsInJson/"+letterValue+".json"),
Letter.class);

    letterService.save(letter);
}
```

Here is the resulting database tables that were created and their properties:



Backend

The backend consists of 4 main layers:

- Models layer
- Persistance layer
- Service layer
- Web layer

Models Layer

The entity models are: **Letter**, **Word** and **Definition**.

Here is one of the model classes as an example:

Word.java:

```
@Data
@Entity
@RequiredArgsConstructor
public class Word {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String word;

    private String name;

    private String type;

    @OneToMany(mappedBy = "word")
    private List<Definition> definitions;

    @ManyToOne
    private Letter letter;
}
```

This entity represents a word in the dictionary. It is identified by it's id, which is of type Long and is a value generated by the database. The @Id annoation above it lets the database know to use this as the primary key for this entity. The database generated these ids in a serial manner, so the first word added to the table has id 1, the next id 2 and so on and so forth…

Apart from name and type properties which are just basic strings, it also has properties that represents the relationships with the other two entities. It has a one to many relationship with the Definition entity, meaning that one word can have many definitions. The mappedBy property of this relation helps the database know how to create the mapping and relations when storing the data. It has a many to one relationship with the Letter entity meaning that the Word is one of many words that a Letter can have.

The annoations at the top of the class have the following purpose:

- @Entity marks the Word class as an entity class,
- @Data generates getters and setters for the properties of the word and
- @RequiredArgsConstructor generates a constructor with the required arguments so this code does not have to be written in the file by the developers.

The models layer also includes a **DTO (data transfer object)** subfolder, for models created to map the data that is needed when sending to the frontend, not to map the database so they're a little different than the entity models.

WordDto.java:

```java
@Data
public class WordDto {

    private Long id;

    private String word;

    private String name;

    private String type;

    private Long letterId;
}
```

This class is different to the one of the entity, Word.java, in that it does not contain a reference to the Letter object, instead just info about the letter'd ID. It also does not have a list of definitions, because that can be quite large and does not need to be sent in every request, only when necessary.

Persistance Layer

The persistance layer was very simple to implement by utilizing the already existing JPA functionalities.

We created three repositories, one for each entity, and each of these repositories inherits from the **JPARepository** which has many pre-built functions that can be used in the project, such as search for items by the primary key or fetching all of the items from a table.

Here's an example of one of the repositories:

LetterRepository.java:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface LetterRepository extends JpaRepository<Letter, Long> {
    Optional<Letter> findByName(String name);
}
```

We did not need to implement any of the methods as this is just an interface and JPA Repository does the implementation for us. This is very useful and makes development easier, faster and less prone to bugs.

The only methods that we needed to even specify in the files were ones that are more specific, such as searching by a property other than just finding the entity by it's primary key. So for example, the findById() method does not need to be specified anywhere.

Service Layer

The service layer is where the business logic lies and it consists of interfaces and their implementation.

The interfaces are used in the other layers, so that those layers are independent of the implementation.

We created three services for this project: **WordService**, **LetterService** and **DefinitionService**.

Here's an example of one of the service interfaces, the word service, which only has method definitions:

WordService.java (the interface):

```
public interface WordService {

    WordDto findById(Long letterId, Long id);
    WordDto findByWord(Long letterId, String word);
    List<WordDto> findAllByLetter(Long letterId);

}
```

And here is the implementation class:

WordServiceImpl.java (the implementation):

```
@Service
public class WordServiceImpl implements WordService {

    private final WordRepository wordRepository;
    private final LetterRepository letterRepository;

    public WordServiceImpl(WordRepository wordRepository,
                           LetterRepository letterRepository, LetterService
letterService) {
        this.wordRepository = wordRepository;
        this.letterRepository = letterRepository;
    }

                                    . . .
```

In order for Spring to recognize it is a service, the @Service annotation is needed, otherwise it would be seen as just any other class and it would cause a problem.

The service communicates with the persistance layers, so it has dependencies injected via the constructor for the repositories needed – WordRepository and LetterRepository.

An example of one of the methods in this service:

```
    @Override
    public WordDto findById(Long letterId, Long id) {

        List<WordDto> wordDtos = this.findAllByLetter(letterId);

        Optional<WordDto> wordDtoOptional = wordDtos.stream()
                .filter(w -> Objects.equals(w.getId(), id))
                .findFirst();

        if(!wordDtoOptional.isPresent()){
            return null;
        }

        return wordDtoOptional.get();
    }
```

All of the methods have the @Override annotation because they are first defined in the interfaces and only implemented here. The method shown in the example above is to find a word by it's ID and return the DTO of it if it is found, hence the optional return type. It uses functions from the persistance to fetch the words from the database and filter through them. It also checks the letter ID of the word because in the following web layer each request related to words or definitions also includes information about the entity higher up in the hierarchy. This means that when searching for a word, the information about the letter is passed as well. Same thing when searching for a definition, information for the word and letter is passed along in the request.

Web Layer

The web layer is what communicates with the frontend Vue.js application. It consists of three rest API controllers, one for each data entity. These controllers communicate with the service layer in order to retrieve the data needed when processing the request they receive.

WordRestController.java

```
@RestController
@CrossOrigin("http://localhost:8080/")
@RequestMapping("/api/letters/{letterId}/words")
public class WordRestController {

    private final WordService wordService;
    private final LetterService letterService;

    public WordRestController(WordService wordService,
                              LetterService letterService) {
        this.wordService = wordService;
        this.letterService = letterService;
    }
```

This code snipped is the beginning of the WordRestController class. It has a @RestController annoation, signaling to Spring that it is a controller for requests. It has a @CrossOrigin annoation which takes a parameter – the route of the frontend application. This tells the backend application that requests coming from that route are okay to process. If it does not have this annoation, then there will be a CORS error on the frontend. Lastly the @RequestMapping annoation spciefies on what mapping this controller is designated to handle requests.

As the web layer communicates with the service layer, it has dependency injections for the needed services, in this case WordService and LetterService. These services are the interfaces, not the implementations, and they are injected via the constructor.

Here is one of the methods from the WordRestController:

```
@GetMapping("/{id}")
public ResponseEntity<WordDto> getById(
        @PathVariable Long letterId,
        @PathVariable Long id){

    LetterDto letterDto = this.letterService.findById(letterId);

    if(letterDto == null){
        return ResponseEntity.notFound().build();
    }

    WordDto wordDto = this.wordService.findById(letterId, id);

    if(wordDto == null){
        return ResponseEntity.notFound().build();
    }

    return ResponseEntity.ok().body(wordDto);
}
```

This method handles **GET** requests on the mapping **/api/letters/{letterId}/words/{id}** and returns the word with the specified ID. First it tries to find the letter the word belongs to via the letterId, if it doesn't it returns a **404 Not Found** to the API user. If that is successful, it tries to find the word. If it cannot find it, it also returns a 404. Lastly, if everything goes okay it returns **200 OK** and returns a DTO of the requested word in the body of the response request.

Frontend

The final part of the project is the Vue.js frontend application. The code for this part is located in the wordinfocus sub-directory of the src folder in the repository.

Components

The Vue application consists of multiple components that are then nested into the App component and into eachother in order to render the user interface. These vue components are: HeaderComponent, FooterComponent, PageNotFoundComponent, WordComponent, LetterWithWordsComponent and LettersComponent.

Let's first look at the contents of the main App.vue component:

App.vue

```
<template>
  <div>
    <HeaderComponent />

    <router-view v-slot="{ Component }">
      <keep-alive :max="5">
        <component :is="Component"
      :key="$route.fullPath" />
      </keep-alive>
    </router-view>

    <FooterComponent />
  </div>
</template>

<script>

import FooterComponent from "@/components/FooterComponent";
import HeaderComponent from "@/components/HeaderComponent";
import LettersComponent from "@/components/LettersComponent.vue"

export default {

  name: 'App',

  components: {

    HeaderComponent,
    FooterComponent,
    LettersComponent,

  }

}

</script>
```

The template of the App component has other components nested in it, which are imported and declared as components in its script. It also has a router-view, which is where other components are nested into depending on the specified route in the browser by the user.

Let’s now see one of the other components, which can be nested into this section.

We will look at the Letters Component as an example. This is the component that is nested into the middle of the App component when the user opens the application i.e. this is the home / landing page. On this page we display all of the thirty one letters in the Macedonian alphabet, organized in rows.

LettersComponent.vue:

```
<template>
    <div class="container">
        <div class="row" v-for="(group, i) in groupLetters" v-bind:key="i">
                <router-link  class="letter" v-for="letter in
this.letters.slice(i * itemsPerRow, (i + 1) * itemsPerRow)" :key="letter.id"
                    :to="{
                    name: 'Letter',
                    params: { l: letter.name}
                }">
                    {{letter.name}}
                </router-link>
        </div>
    </div>
</template>
<script>
    export default{
        name: "LettersComponent",
        data() {
            return {
                itemsPerRow: 8,
                letters : [],
                url: "/",
            }
        },
        methods:{
            async getLetters(){
               const lettersApi =await fetch('http://localhost:9090/api/letters')
               const result = await lettersApi.json();
               this.letters = result;
            }
        },
        computed: {
            groupLetters () {
                return Array.from(Array(Math.ceil(this.letters.length /
this.itemsPerRow)).keys())
            },
        },
        beforeMount() {
            this.getLetters()
        },
    }
</script>
```

The template part does what was explained before, it displays all of the letters in rows, with 8 items per row. We get these letter in the getLetters() method which makes an **asynchronous API call** to the mapping http://localhost:9090/api/letters. We call this method before mounting the component, so that when the page renders the letters are displayed. This is done with the **beforeMount()** method.

The API call to fetch the letters is asynchronous because that allows users to make multiple requests at the same time without waiting for the previous request to finish exeucting. This means that the server may process multiple requests at the same time, decreasing the API's average response time to incoming requests. This is not something that is important right now as this is an application hosted locally and used only by ourselves, but if it were used by many users this is extremely important.

Routing

The routing is done in the router.js file which is located in the router directory of the frontend source code.

The routes are the following:

- "/" which is the home page i.e. the page where the Letters component is nested
- "/letter/:l" or '/letter/:l #', "/letter/:l. : w/:t" which is where the LetterWIthWords component is nested into the App component

```
import { createRouter, createWebHistory } from 'vue-router'

const routes = [
    {
      path: "/",
      name: "Letters",
      component: () => import("../components/LettersComponent.vue"),

    },
    {
      path: "/letter/:l",
      alias: ['/letter/:l#', "/letter/:l.:w/:t"],
      name: "Letter",
      component: () => import("../components/LetterWithWordsComponent.vue"),
      props: true,
    }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
})
```

This second route is passed properties via props which make the route variable depending on the letter or word.

Conclusion

In conclusion, this project is not only a tool, but it also helped us learn a lot about different kinds of technologies and would help in preserving our language and helping our native language to grow. Through this project, we we wanted to connect tradition and technology.

Achievements

- **Comprehensive Digital Dictionary:** With working hard on web scraping, data organization / database setup and creating the application, we have successfully created a web application digital dictionary that encompasses a wide range of words, their meanings, and usage examples.
- **Technology Learning:** The project has been a learning curve that allowed us to gain skills in a diverse set of technologies, including Python, Docker, Java Spring, and Vue.js. These skills will definitely serve us well in our future learning and work.
- **Structured Architecture:** "WordsInFocus" has a structured architecture, embracing the principles of modularity and scalability.
- **User-Friendly Frontend:** The Vue.js frontend gives an intuitive and visually appealing user interface, inviting people to explore our language's wealth easily and help them learn.

Challenges and Growth

Our work was not without any challenges. We encountered many technical obstacles, navigated through learning new things, and struggled with the complexity of data extraction and manipulation. These challenges were however opportunities for growth and helped us learn how to solve problems better.

Future Possibilities

As we think back on our work in the "WordsInFocus" project, we see that our work is far from complete. There are countless possible future enhancements. Some potential things include:

- **Expansion of Content:** Continue to expand and enrich the dictionary with more words, idiomatic expressions, and linguistic resources for learning and such.
- **User Engagement:** We can implement features for user contributions, feedback, and interacting to create a sense of community around our native language.
- **Language Learning Tools:** Developing language learning tools and resources that use the dictionary's data for educational purposes.
- **Cross-Platform Accessibility:** Extending "WordsInFocus" to other platforms, such as mobile applications on different devices, to reach a wider audience.