École Polytechnique Fédérale de Lausanne

Semester Project (Fall 2020) - Master of Robotics

Interfacing and expanding Kam4D, Kamusi's graph database

Diane Marquette SCIPER: 263789

Supervisors

Dr. Martin BENJAMIN

Prof. Karl ABERER

January 15th, 2021

in collaboration with

kamusi.org

Contents

1. Introduction 3

2. Project management 3

2.1. My initial project proposal 3

2.2. Kamusi’s needs 3

2.3. An unexpected obstacle: a cranky Vagrant 4

3. Setting up the development environment on my local computer 5

3.1. Setting up Vagrant for the first time 5

3.2. Running Vagrant 5

3.3. Setting up and running Neo4j 6

4. Updating the database public interface 9

4.1. The current solution 9

4.1.1. Description 9

4.1.2. Implementation 11

4.2. The new database: Kam4D 13

4.2.1. Key concepts: smurfs, ducks, lemurs and costumes 13

4.2.2. Querying data in Kam4D 14

4.3. The updated interface 16

4.3.1 Specifications 16

4.3.2. Possible implementation 17

5. Feeding Kam4D 18

5.1. Finding the most frequent words in French 18

5.2. Identifying words unknown to Kam4D 19

5.2.1. The first version of the script 19

5.2.2. Execution time & Query performance tuning 19

5.2.3. Which types of words are missing in Kam4D? 21

5.3. Finding the costumes of French verbs with Verbiste 21

5.3.1. Setting up Verbiste 21

5.3.2. Improving our script to identify words unknown to Kamusi 21

5.3.3. Adding the costumes of French verbs to Kam4D 23

5.4. Extracting meaningful example sentences from French corpora 24

6. Next steps 25

7. Conclusion 26

8. Appendix 27

8.1. Initial project proposal 27

8.2. Connection procedures (development server) 28

8.2.1. Connecting to the development server 28

8.1.2. Connecting to neo4j on the development server 28

8.2.3. Restarting nginx 28

8.3. Personal notes on back-end key concepts 28

8.3.1. Node.js 28

8.3.2. Middleware functions 29

8.4. Installing Verbiste 30

References 32

1. Introduction

The work presented in this report was done as part of my semester project during Fall 2020 within Kamusi. Kamusi is an NGO that is building a universal online dictionary. It aims to translate all the meanings of every word in all the world's languages.

The main goals of this report are to:

- present the work accomplished during this semester
- explain the decisions taken
- serve as a guide to any new student working on the Kamusi project

2. Project management

The project changed twice directions to adapt to the organization's priorities and the resources at our disposal.

2.1. My initial project proposal

Initially, I contacted Kamusi through Prof. Aberer's LSIR lab as I was very interested in their work and the games (linked to languages) they developed. I had been wanting to develop a software to help language learners focus on the most useful vocabulary to learn first. In a nutshell, my goal was to develop an interface where language learners could copy-paste an article. Users could then highlight the words they didn't understand or that they weren't comfortable using. The software would recognize key terms and suggest to the user the top k terms to remember from the analyzed article. Please refer to the project description in the appendix 8.1 for further details.

In addition, this project would provide valuable user feedback to Kamusi's dev team and help them identify terms that are potentially missing in the database. As a result, Dr. Martin Benjamin accepted to help me with this project.

2.2. Kamusi's needs

Kamusi already has user interfaces running on its website. The most important one is the table on its website landing page. It allows users to query data in its database (cf. section 4.1 for further details). However, Kamusi migrated to a new database management system during summer 2020. It was therefore urgent to update the website so that the queries exploited the features of the new database.

I decided to focus on this task as it would give users access to the power of the new database. Plus, it was a great starting point for me to understand how the front-end and the back-end work (as I had no previous experience with websites nor databases).

2.3. An unexpected obstacle: a cranky Vagrant

To work on Kamusi's website, I used a Vagrant virtual software development environment. Unfortunately, it stopped working during week 4 of the semester. It was supposed to be fixed in a week or two, but things didn't go as planned. The problem persisted as the student working on it was struggling with this bug.

Thankfully, we adopted a proactive approach to avoid wasting time and keep me busy. Dr. Martin Benjamin and I decided that I should focus on shorter tasks (that could be completed in a few weeks) to feed the database with missing data while waiting for Vagrant to be fixed. The plan was that as soon as Vagrant was up running again, I would go back to working on the website's user interface to finish it by the end of the semester. In the end, this never happened as the issue with Vagrant was resolved during the last week of the semester (it was caused by dependencies versions that were deprecated).

Nevertheless, I'm happy with what I accomplished during this semester and to have contributed to this online multilingual dictionary. In total, I developed five scripts:

- two to extract words from a list that are unknown to Kamusi's database (one without Verbiste and one with Verbiste)
- one to generate a CSV file with all the conjugated forms of French verbs (using Verbiste)
- two to generate a CSV file with example sentences extracted from a French corpus (one designed to be run locally and one to be executed using Google Colab)

Please refer to section 5 for further details.

3. Setting up the development environment on my local computer

3.1. Setting up Vagrant for the first time

The first step to complete in order to start working as a Kamusi developer is setting up the development environment on your local computer. This allows you to run Kamusi's website locally on your computer and therefore test the features that you added or updated.

The procedure is explained by Sina Mansour in this video and can be summarized by the following steps:

1. create a GitLab account (if you don't already have one)
2. fork the main repository from https://gitlab.com/kamusi/api-fork-from-lr.git
3. clone the forked repository (to get a copy of the folder on your computer)
4. rename the main folder ("api-fork-from-lr") to "API" (optional)
5. in the API/Kamusi_API/db folder, duplicate the `config (sample).js` file and rename it to `config.js` (it contains dev admin usernames and passwords that we don't need to change)
6. in the API/Kamusi_API/mailer folder, duplicate the `config sample.js` file and also rename it to `config.js` (this step isn't mentioned explicitly in the video, but I didn't manage to make Vagrant work without it)

By the end of step 6, everything should be configured correctly and you are ready to run Vagrant for the first time. Refer to the following section for the launch procedure.

3.2. Running Vagrant

To run Vagrant (including Kamusi website locally), follow these steps:

Launch procedure

1. open terminal
2. navigate to the `api-fork-from-lr/vagrant` folder
3. execute the `vagrant up` command
4. (if error) execute `vagrant ssh` and `pm2 start /kamusi/api/bin/www` (last line in setup_user_2.sh in vagrant folder)

5. access development environment using `localhost:6600` in web browser → you should see your local Kamusi landing page displayed

⚠ Vagrant continues to run in the background even if you close your terminal. Follow the following procedure to shut it down.

Shut-down procedure

1. open a new terminal
2. navigate to the `api-fork-from-lr/vagrant` folder
3. execute the `vagrant halt` command
4. check that `localhost:6600` no longer works

Update procedure

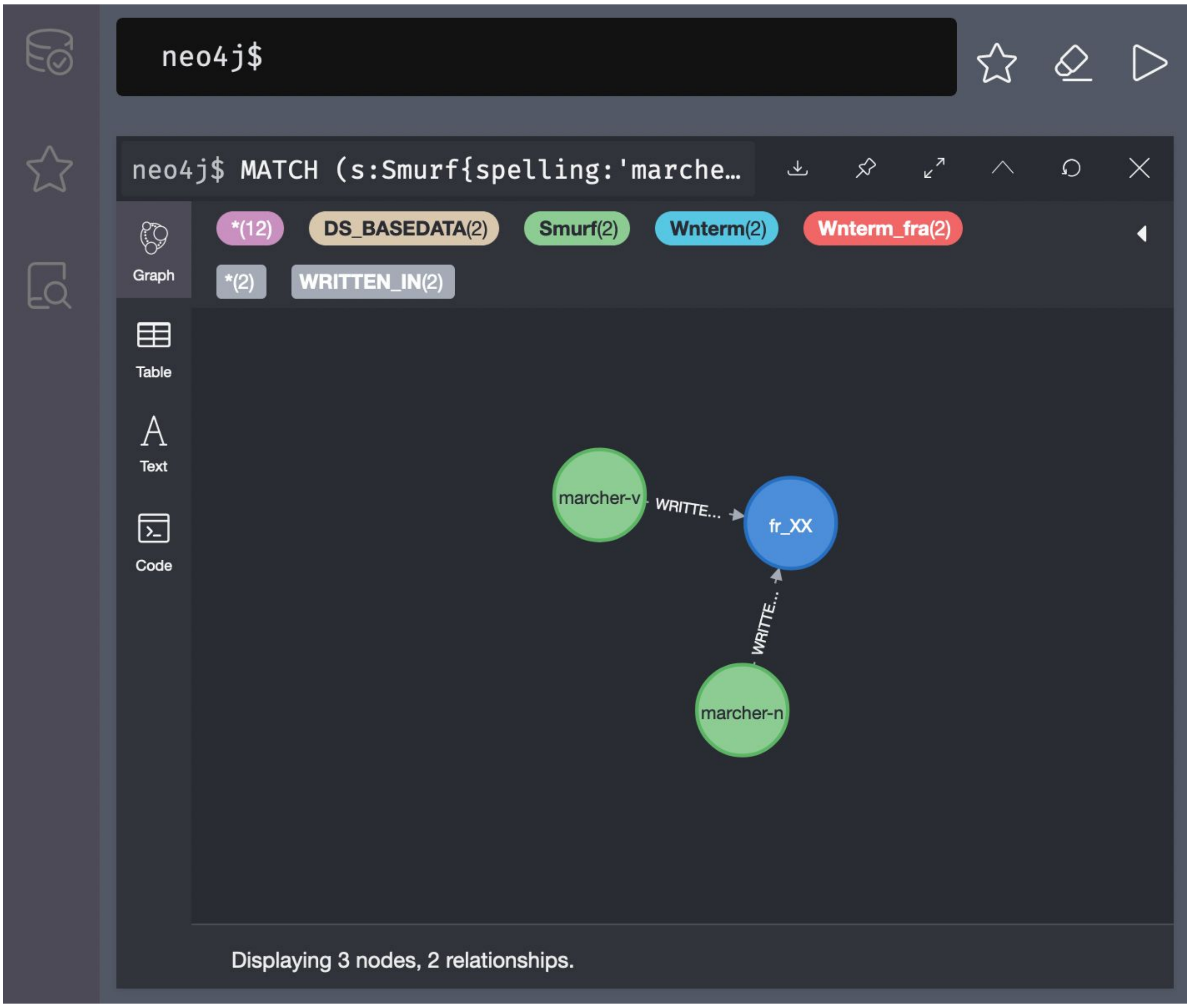
1. open a new terminal
2. navigate to the `api-fork-from-lr/vagrant` folder
3. execute the `vagrant destroy` command
4. execute the `vagrant up` command

3.3. Setting up and running Neo4j

In order to work with the latest version of the database on Jérôme Baton's machine, he sent me a copy of the server to run locally on my computer.

Procedure

1. download the compressed archive (`neo4j-community-4.1.1__DIANE.zip` in my case) and decompress it (only the 1st time)
2. visit the sub-directory `/bin` of the extracted folder and execute `./neo4j start` in terminal (OR `./neo4j console` to get the output in same window)
3. visit http://localhost:7474 in your web browser
4. connection details (only the 1st time)
    a. neo4j://localhost:7687
    b. neo4j (username)
    c. password
5. to stop server, execute `./neo4j stop` in terminal (ctrl-c if launched with neo4J console)



```
neo4j$
```

```
neo4j$ MATCH (s:Smurf{spelling:'marche…
```

Displaying 3 nodes, 2 relationships.

Figure 1 - *Screenshot of the user interface accessible on http://localhost:7474 when the server is running (in this case, it shows the result of a query asking for the word "marcher" in French)*

To check the server status, go to sub-directory `/bin` and execute `./neo4j status`. It will return:

- "Neo4j is running at pid 2073" if the server is running (it can be another pid)
- "Neo4j is not running" if it has been properly stopped

To optimize the query delays while not monopolizing your computer's RAM, Neo4j has a built-in command that recommends you the best memory configuration for Neo4j given the machine you are using.

Procedure

1. visit the subdirectory `/bin` of the extracted folder
2. execute `./neo4j-admin memrec --memory=8g` in terminal (I specified 8g, as my computer has 8 GB of RAM)
3. go back to the main directory
4. modify the `neo4j.conf` file inside the `/conf` subdirectory according to Neo4j's recommendations

In my case, it gave me the following recommendations:

```
Based on the above, the following memory settings are recommended:
dbms.memory.heap.initial_size=3600m
dbms.memory.heap.max_size=3600m
dbms.memory.pagecache.size=2g
```

4. Updating the database public interface

4.1. The current solution

4.1.1. Description

The current database public interface corresponds to the landing page of Kamusi's website as shown in Figure 2.

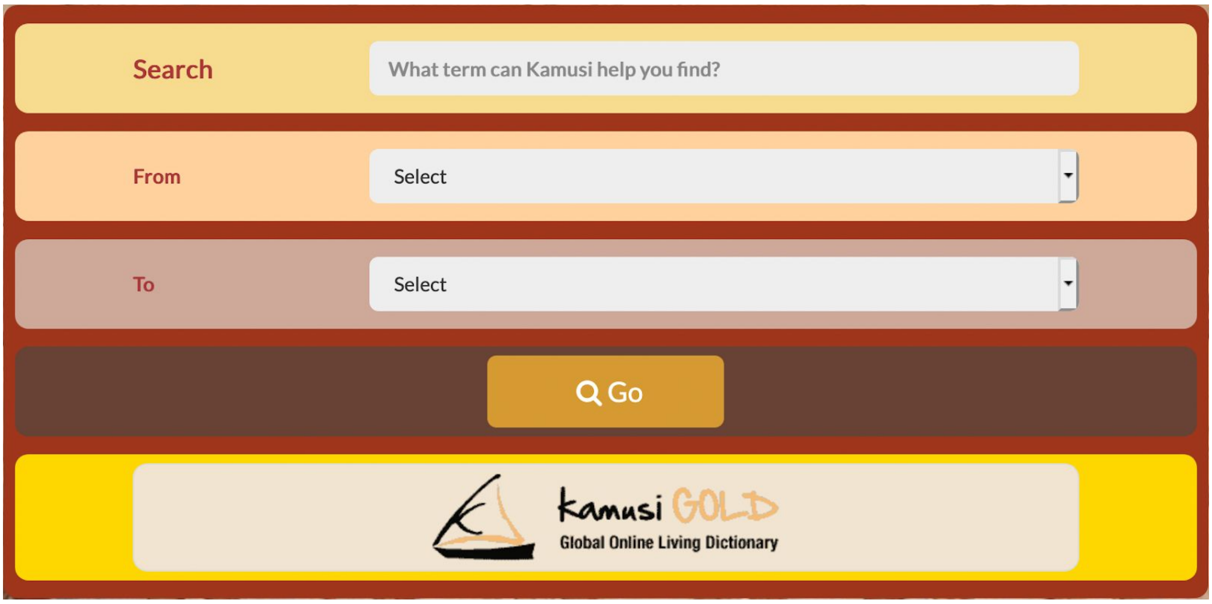
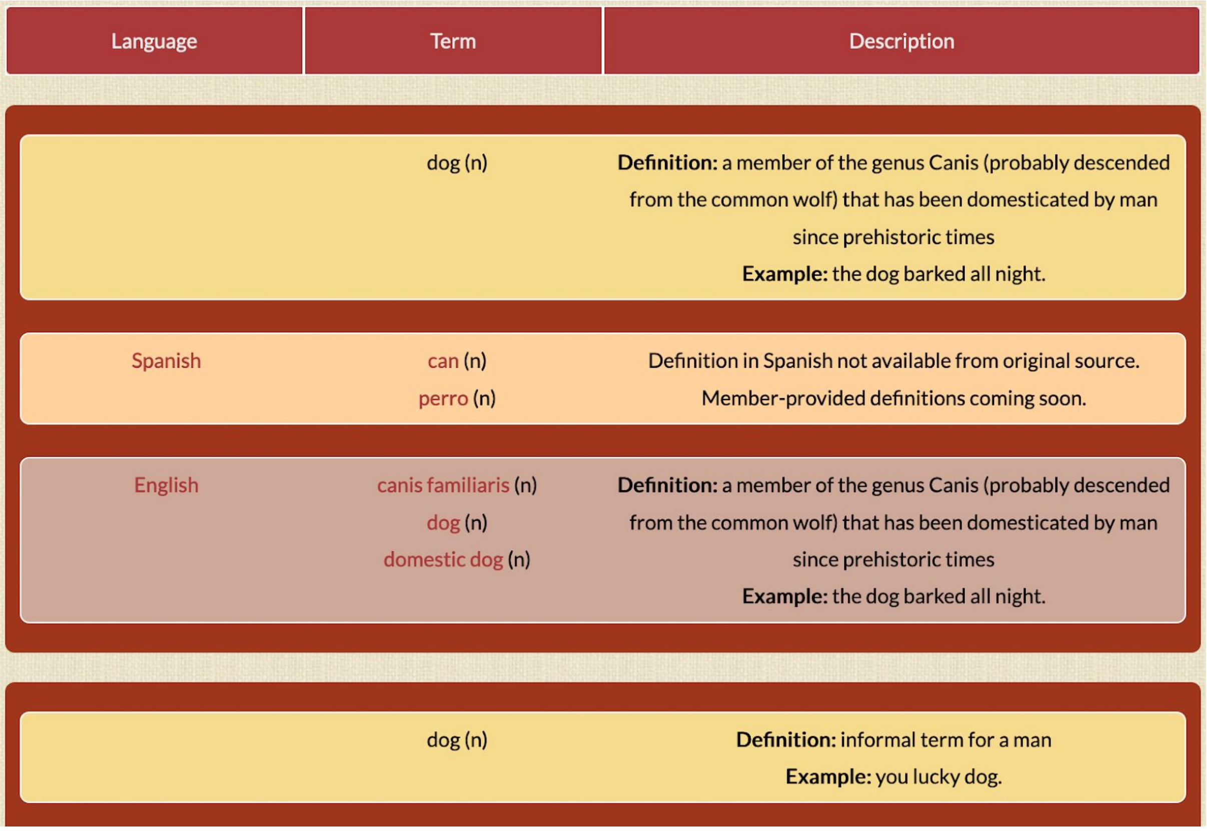

Search
What term can Kamusi help you find?
From
Select
To
Select
Go
kamusi GOLD
Global Online Living Dictionary

Figure 2 - *Screenshot of Kamusi's current database public interface*

In the **textfield** at the top of the table, the user can enter the **word to translate**. The second row enables the user to select the "**from**" **language** within a list of 44 languages. The third row corresponds to the **target language** ("to" language). The lists of languages (from & to) are displayed using **drop-down menus** which are quite tedious to use, especially on a smartphone.

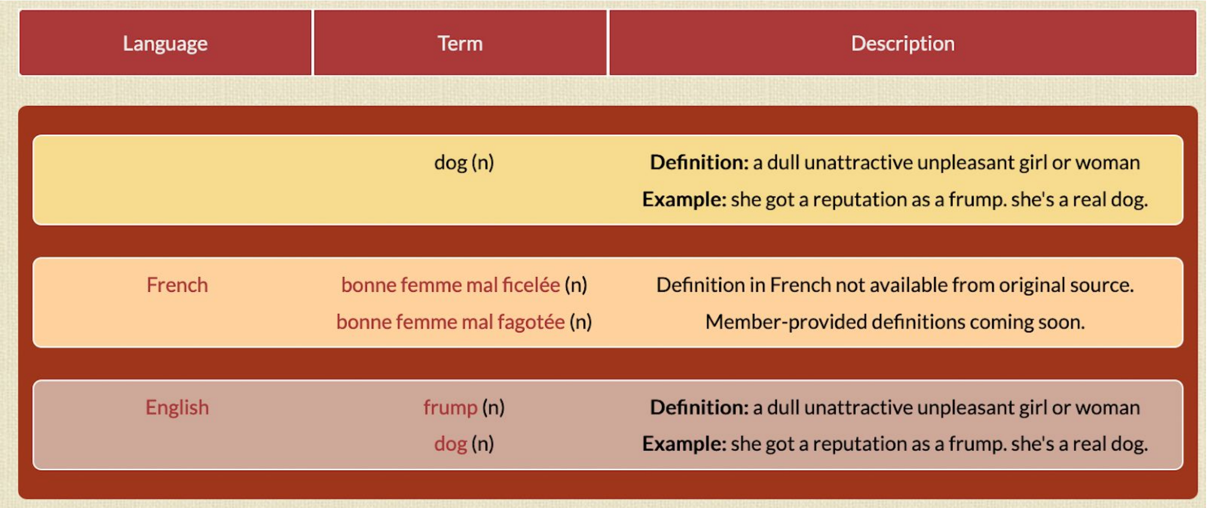
Regarding the translation procedure, it looks up the word to translate in the WordNet of the "from" language, then searches for its equivalent in the English WordNet. We then use the English translation to look for its equivalent in the "to" language Wordnet. The results of the query always give the English equivalent (which is especially important in case we infer knowledge).

The results displayed include the translation of all the different meanings of the queried word. Unfortunately, at time of writing, the meaning that is shown first isn't necessarily the most frequent one. The order in which the meanings of the word are displayed is random. In Figure 3, we can see that if we translate the word "dog" from English to Spanish, the first meaning (and translation) that appears is the most used one (i.e., the animal). However, as shown in Figure 4, if we ask Kamusi to translate "dog" from English to French, the first meaning for which a translation is given is "a dull, unattractive, unpleasant, girl or woman", which is rarely used.



Language	Term	Description
	dog (n)	**Definition:** a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times **Example:** the dog barked all night.
Spanish	can (n) perro (n)	Definition in Spanish not available from original source. Member-provided definitions coming soon.
English	canis familiaris (n) dog (n) domestic dog (n)	**Definition:** a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times **Example:** the dog barked all night.
	dog (n)	**Definition:** informal term for a man **Example:** you lucky dog.

Figure 3 - *Screenshot of the results displayed when a user asks to translate the word "dog" from English to Spanish*



Language	Term	Description
	dog (n)	**Definition:** a dull unattractive unpleasant girl or woman **Example:** she got a reputation as a frump. she's a real dog.
French	bonne femme mal ficelée (n) bonne femme mal fagotée (n)	Definition in French not available from original source. Member-provided definitions coming soon.
English	frump (n) dog (n)	**Definition:** a dull unattractive unpleasant girl or woman **Example:** she got a reputation as a frump. she's a real dog.

Figure 4 - *Screenshot of the results displayed when a user asks to translate the word "dog" from English to French*

4.1.2. Implementation

To understand the implementation of this public interface, we must dive deep into Kamusi's API on its GitLab repository.

Deciphering Kamusi's API was definitely one of the hardest tasks of this semester project as I had no previous experience in Javascript nor routing or back-end & front-end interfacing. In addition, there is unfortunately not a single README and I couldn't communicate with the developers that had worked on the code. At times (especially with the `neo4j.js` file in the `Kamusi_API/db` subdirectory), I truly felt like an archeologist excavating forgotten remains of an ancient civilization. Eventually, I managed to understand which files were important for the database public interface. Hopefully, the following analysis will help the next developer working on the code.

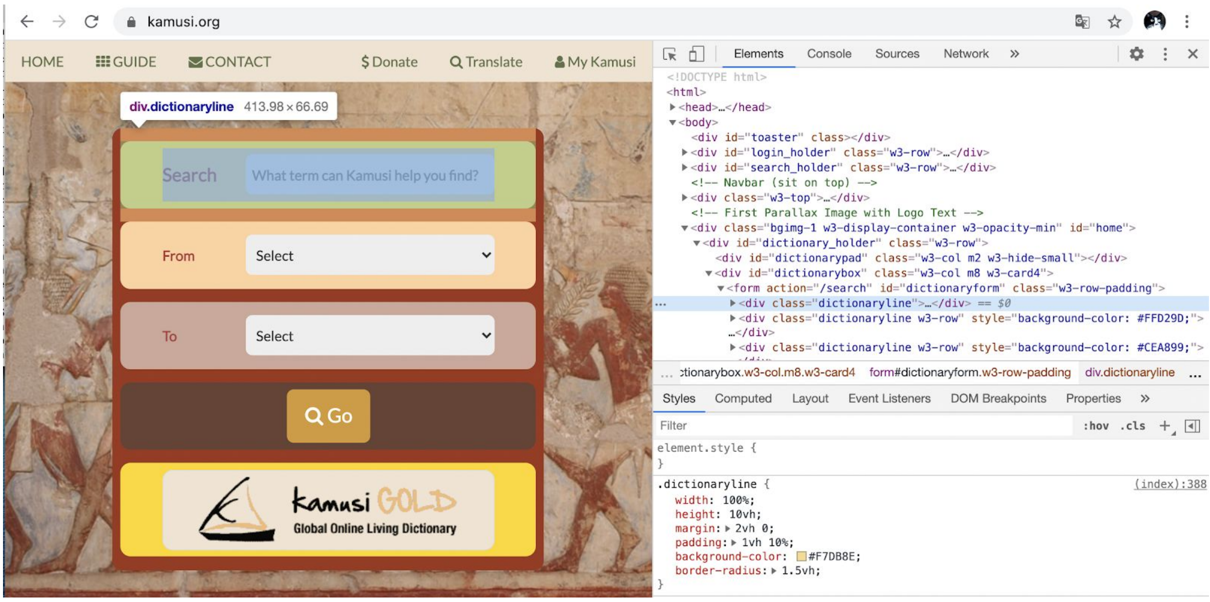
1) Web pages JavaScript templates

The two key files for the UI of the database public interface are in the KamusiAPI → views subdirectory :

- `splash.ejs` → corresponds to Kamusi's website landing page with the search box (as shown in Figure 2)
- `splashsearch.ejs` → corresponds to the web page with the table displaying the results of the user's query (as shown in Figure 3)

The .ejs extension stands for "Embedded JavaScript". It's a templating language that generates HTML markup with plain JavaScript.

As a JavaScript newbie, I used Google Chrome's "inspect" option to display the HTML code associated with each region on the webpage. It was useful to quickly bridge the gap between the website rendering and the code.



```
<!DOCTYPE html>
<html>
▶<head>…</head>
▼<body>
   <div id="toaster" class></div>
  ▶<div id="login_holder" class="w3-row">…</div>
  ▶<div id="search_holder" class="w3-row">…</div>
   <!-- Navbar (sit on top) -->
  ▶<div class="w3-top">…</div>
   <!-- First Parallax Image with Logo Text -->
  ▼<div class="bgimg-1 w3-display-container w3-opacity-min" id="home">
    ▼<div id="dictionary_holder" class="w3-row">
       <div id="dictionarypad" class="w3-col m2 w3-hide-small"></div>
      ▼<div id="dictionarybox" class="w3-col m8 w3-card4">
        ▼<form action="/search" id="dictionaryform" class="w3-row-padding">
...       ▶<div class="dictionaryline">…</div> == $0
          ▶<div class="dictionaryline w3-row" style="background-color: #FFD29D;">
          …</div>
          ▶<div class="dictionaryline w3-row" style="background-color: #CEA899;">
```

… :tionarybox.w3-col.m8.w3-card4   form#dictionaryform.w3-row-padding   div.dictionaryline …

```
element.style {
}
.dictionaryline {                                    (index):388
   width: 100%;
   height: 10vh;
   margin: ▶ 2vh 0;
   padding: ▶ 1vh 10%;
   background-color: #F7DB8E;
   border-radius: ▶ 1.5vh;
}
```

Figure 5 - *Screenshot of Kamusi's landing page with its associated HTML code (displayed on Google Chrome, by clicking on a region of the webpage and then selecting "inspect" in the menu)*

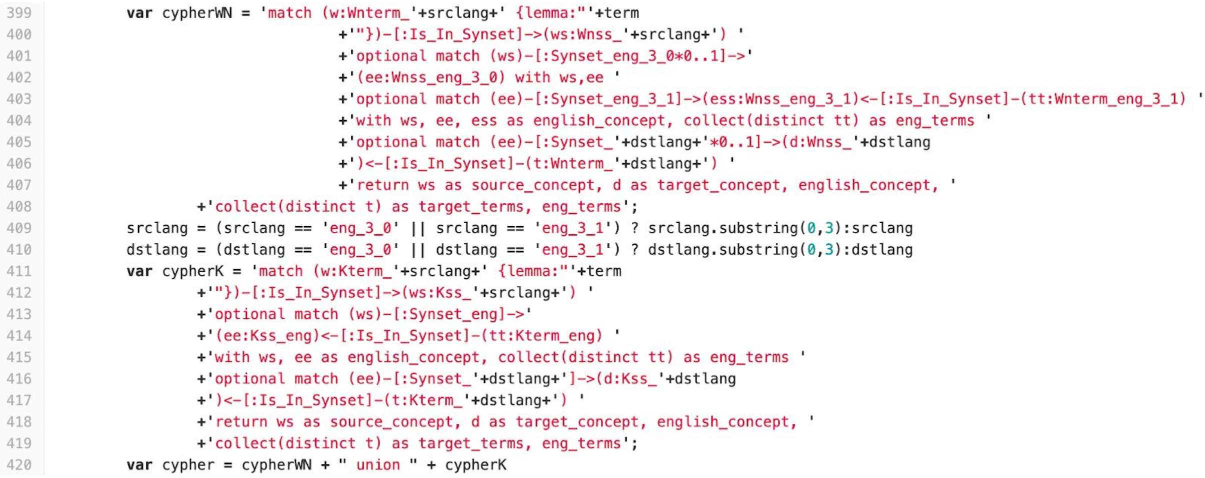
2) Routing

However, all the magic between the front-end and back-end happens thanks to the `index.js` file in the `Kamusi_API/routes` subdirectory. It handles the website startup and routing.

The `require()` method used in the first lines of the file is not part of the standard JavaScript API. It's a Node.js built-in function to load modules (similar to C's `include` and Python's `import`). As opposed to browser JavaScript in which scripts have direct access to the global scope, in Node.js, each module has its own scope. As explained in the following StackOverflow post, " a module cannot directly access things defined in another module unless it chooses to expose them. To expose things from a module, they must be assigned to `exports` or `module.exports`". Please refer to section 8.3.1 in the Appendix for more detailed notes on Node.js.
⇒ In a nutshell, `index.js` uses the Node.js environment and starts by importing the necessary modules for the routing.

The "splash" page (i.e., the landing page with the search box) section starts at line 180 of `index.js`. It calls the `splash.ejs` file using `res.render('splash')` to display the landing page when users enter kamusi.org/ in their browser.

If the user presses the "Go" button, the "splashsearch" page is displayed with the query results. It corresponds to the code from line 384 (`if (box == 'search') {`) to line 447 (with `res.render('splashsearch'` at line 441). The router requests the source language (`src_lang`, line 388) and the target language (`dst_lang`, line 389), as well as the word to translate (`term`, line 395) that have been saved by the landing page. Refer to section 8.3.2 in the Appendix for more detailed notes on middleware functions. As shown in Figure 6, the router then formulates two queries, `cypherWN` (lines 399 to 408) and `cypherK` (lines 411 to 419) to match words from WordNet or Kamusi specific terms (Kterm) respectively to the term to translate. The two queries are merged into a single one at line 420 and then sent to the database (line 420: `db.query(cypher, function(result)`). The result is given back to "splashsearch" for the rendering to the table with the proposed translations.



```
var cypherWN = 'match (w:Wnterm_'+srclang+' {lemma:"'+term
                        +'"})-[:Is_In_Synset]->(ws:Wnss_'+srclang+') '
                        +'optional match (ws)-[:Synset_eng_3_0*0..1]->'
                        +'(ee:Wnss_eng_3_0) with ws,ee '
                        +'optional match (ee)-[:Synset_eng_3_1]->(ess:Wnss_eng_3_1)<-[:Is_In_Synset]-(tt:Wnterm_eng_3_1) '
                        +'with ws, ee, ess as english_concept, collect(distinct tt) as eng_terms '
                        +'optional match (ee)-[:Synset_'+dstlang+'*0..1]->(d:Wnss_'+dstlang
                        +')<-[:Is_In_Synset]-(t:Wnterm_'+dstlang+') '
                        +'return ws as source_concept, d as target_concept, english_concept, '
        +'collect(distinct t) as target_terms, eng_terms';
srclang = (srclang == 'eng_3_0' || srclang == 'eng_3_1') ? srclang.substring(0,3):srclang
dstlang = (dstlang == 'eng_3_0' || dstlang == 'eng_3_1') ? dstlang.substring(0,3):dstlang
var cypherK = 'match (w:Kterm_'+srclang+' {lemma:"'+term
        +'"})-[:Is_In_Synset]->(ws:Kss_'+srclang+') '
        +'optional match (ws)-[:Synset_eng]->'
        +'(ee:Kss_eng)<-[:Is_In_Synset]-(tt:Kterm_eng) '
        +'with ws, ee as english_concept, collect(distinct tt) as eng_terms '
        +'optional match (ee)-[:Synset_'+dstlang+']->(d:Kss_'+dstlang
        +')<-[:Is_In_Synset]-(t:Kterm_'+dstlang+') '
        +'return ws as source_concept, d as target_concept, english_concept, '
        +'collect(distinct t) as target_terms, eng_terms';
var cypher = cypherWN + " union " + cypherK
```

Figure 6 - *Code snippet from* `index.js` *detailing the queries formulation*

4.2. The new database: Kam4D

During Summer 2020, Jérôme Bâton migrated Kamusi's old database based on WordNet and mySQL to a Neo4j graph database, named Kam4D. The graph nature of this new database allows us to connect words to their different meanings across thousands of languages.

4.2.1. Key concepts: smurfs, ducks, lemurs and costumes

The lexicographic structure implemented in this graph database is documented in detail in this paper (*Lexicography in Four Dimensions: Database Structure for a Matrix of Human Expression Across Time and Space)* written by Martin Benjamin [1]. However, for this semester project, only a few key concepts must be understood:

- **SMURFS (Spelling / Meaning Unit ReFerences)** → A *smurf* is "the unique intersection of a single spelling with a single meaning in a single language". In Kam4D, each smurf has its own reference number and corresponds to maximum 3 connected nodes:
    - a language (shared)
    - a spelling
    - a definition (unfortunately, currently rarely available; ≈ 3 million smurfs but only ≈ 30'000 definitions)

- **DUCKS (Data Unified Concept Knowledge Sets)** → A duck is a collection of terms that refer to the same concept (i.e., that share a meaning within a language (= synonyms) or between languages (= translations)).

- **Costumes** → A *costume* is one of the forms that a word can take. For example, "see", "saw", "seeing" are all costumes of the verb "see", while "apple" and "apples" are the two costumes of the noun "apple".

    - “Costumes” is an engineer-friendly name for what linguists call “inflections”.
    - Costumes often don’t exist yet in Kam4D (for example, only the infinitive form of verbs is available).
    - Costumes are stored in a “**wardrobe**”, such that all of the appropriate senses of the verb “see” can select their outfit from a wardrobe that consists of «see/ sees/ saw/ seeing/ seen».
- **LEMURS (LEMmatic Unit Reference)** → A lemur corresponds to a lemma but also encompasses a wider range of items.
    - It corresponds to a node connected to different wardrobes and smurfs sharing this spelling.
    - For wardrobes and costumes, all the action occurs relative to the lemur; since shapes have nothing to do with meaning, it makes no sense to repeat inflectional information for every smurf.

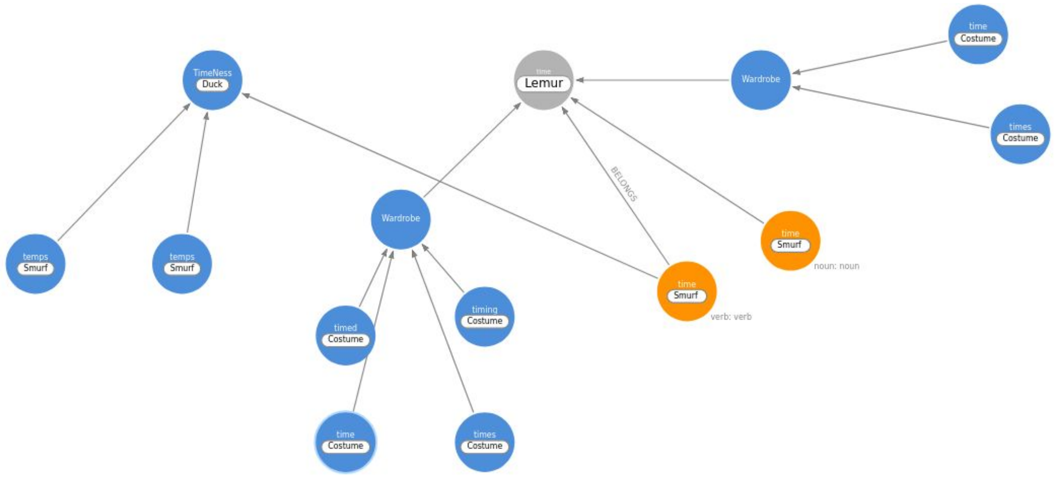


Figure 7 - *Relations between ducks, smurfs, costumes, wardrobes and lemurs*

4.2.2. Querying data in Kam4D

As I had no previous experience in back-end (let alone graph databases), I used Jérôme Baton’s and Rik Van Bruggen’s book “Learning Neo4j 3.x: Effective data modeling, performance tuning and data visualization techniques in Neo4j” [2]. It provided me a much needed overview of the key concepts and characteristics of graph databases and Neo4j.

To interact with Kam4D, we use Cypher, the query language of Neo4j. The following queries are especially useful:

Query to test if a word exists in Kam4D
```
MATCH (s:Smurf{spelling:'marcher'})-[:WRITTEN_IN]->
                                               (l:Language {code:'FRA'})
```

```
RETURN s,l
```

Query to test if a word written with an accent exists in Kam4D

```
MATCH (s:Smurf{lemma_accent:'même'})-[:WRITTEN_IN]->
                                          (l:Language {code:'FRA'})
RETURN s,l
```

For additional examples of Cypher queries for Kam4D, please refer to Jérôme Bâton's "Kam4D Cookbook".

The pattern of a Cypher query is:

```
(NODE1)-[:RELATION]->(NODE2)
```

In both example queries, we want to retrieve nodes with the label "Smurf" linked to nodes with the label "Language" by a "WRITTEN_IN" relation. In the first query, we are selecting nodes based on their "spelling" property, while in the second query, we focus on the "lemma_accent" property.

If we take "mémé" (i.e., "granny" in French) as an example, the query

```
MATCH (s:Smurf{spelling:'mémé'})-[:WRITTEN_IN]->
                                          (l:Language {code:'FRA'})
RETURN s,l
```

returns no result. This is because the "spelling" property of the node only stores its spelling without accents!

If we try again with the same query except that we write "mémé" as "meme" (i.e., without accents), we get the following output as shown in Figure 8 :

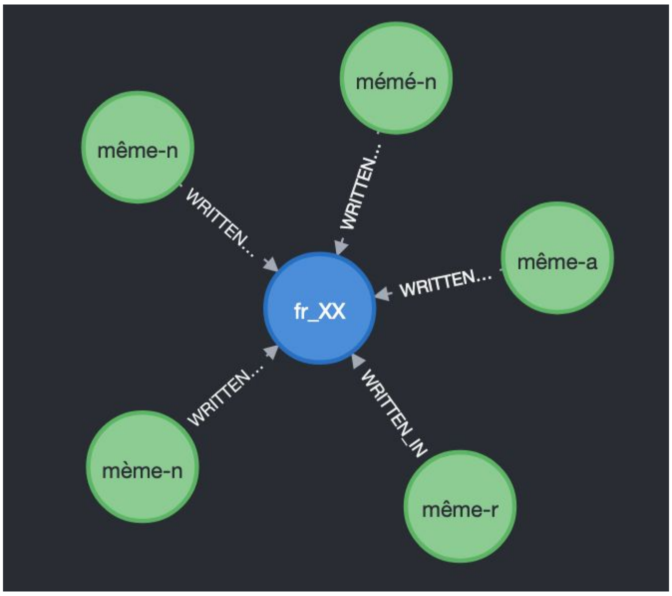


Figure 8 - *Graph representation of the output of query*
`MATCH (s:Smurf{spelling:'meme'})-[:WRITTEN_IN]->(l:Language{code:'FRA'}) RETURN s,l`

Last but not least, if we look for nodes with “mémé” as the value of their “lemma_accent” property, we retrieve only one node, i.e., the one corresponding to a noun meaning “granny” in French and written as “mémé”.

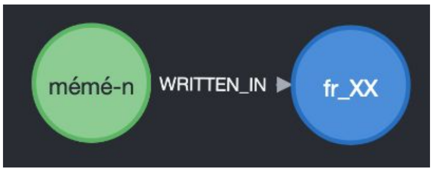


Figure 9 - *Graph representation of the output of query* `MATCH (s:Smurf{lemma_accent:'mémé'})-[:WRITTEN_IN]->(l:Language{code:'FRA'}) RETURN s,l`

At the time of writing, Kam4D Neo4j implementation is still under development. The model needs to continue to be improved as it’s still difficult to formulate queries due to:

- the large number of labels
    - 289 in total (output of query `CALL apoc.meta.stats() YIELD labelCount RETURN labelCount`) including “Turkish”, “Wordnet”, “Wnterm_gle”, “French”, “POS_duck”, “Smurf”
    - there is no list telling you that “English” and “French” are both labels for languages
- some duplicates of smurfs from Wordnet and Drupal (i.e., the old model that used mySQL) → it was already the case in the old database, but the problem has yet to be fixed

⚠ If you write a query with a typo, Neo4j will still execute the query without sending a warning and will display NULL as a result.

4.3. The updated interface

4.3.1 Specifications

Kamusi has two main goals for the new version of the database public interface.

First, the website should query the new Kam4D database. Instead of translating terms using WordNet, it should directly take advantage of the lexicographic structure of the graph database. To put it simply, it must find the duck number associated with the word the user is looking up and return its translation.

Second, the UI in itself should be simplified, especially the language selection. Instead of using dropdown menus as before, we could use predictive typing with a twist:

- “from” language → predictive typing ONLY for the languages where we have data (currently 44 languages out of the 7000 languages we have in predictive typing)
- “to” language → leave the entire 7000 options open to ask the user to provide a suggestion if we don’t have this term yet (then, once we have some validated data in that new language, we want to automatically add it to the available “from” languages)

4.3.2. Possible implementation

To begin with, as of December 2020, `index.js` includes the code to connect to the Neo4j graph database. Line 12 defines the variable `db` as follows:

```
var db = require('../db/neo4j');
```

If we have a look at the `neo4j.js` file in the Kamusi_API/db subdirectory, we see that line 3 also defines a `db` variable, this time using the username, password and server specified in SeraphConf in the `config.js` file (in the same Kamusi_API/db subdirectory). This implies that the website can already query data in Kam4D, but it only uses the nodes from WordNet.

TO DO: replace the two Cypher queries using WordNet (explained in section 4.1.2) in `index.js` with a query to:

1. find the wardrobe(s) containing the word to translate as a costume
2. get the lemur associated to this wardrobe
3. retrieve the smurf(s) associated to this lemur
4. find the duck(s) associated to the smurf(s)
5. return the smurf(s) in the target language connected to the same duck as the smurf(s) in the source language

Unfortunately, I wasn’t able to implement the updated version of the database public interface. Indeed, as explained in section 2.3, Vagrant stopped working during week 4 and was only fixed on December 19th (i.e., after the end of the semester). Therefore, the above comments are suggestions and ideas that I came up with after analyzing the repository code and some brainstorming, but, unfortunately, they were never tested or implemented.

5. Feeding Kam4D

While waiting for Vagrant to be fixed, we decided to change gears and come up with shorter tasks that would grow Kamusi's database. As a French native speaker and as French teacher, I chose to focus on expanding Kam4D's vocabulary in French.

5.1. Finding the most frequent words in French

My first goal was to find a list with the most frequent words in French. After doing some research, I decided to use WordLex [3] as it is freely available and downloadable from http://worldlex.lexique.org/.

In addition, it provides new frequencies based on Twitter, blog posts or newspapers for 66 languages (including French and English). These frequencies are reliable and modern alternatives to frequencies used until now (and based on books or movie subtitles).

For each language, two files are made available. The first one includes the frequencies of all character strings in the corpus. The second one only takes into account the strings that were validated by the spell checker to the given language (it contains fewer orthographic and typographic errors than the raw frequency file but proper names of new words may have been removed). Because proper names are not Kam4D's focus, we will use the second file.

Each word frequency file contains the following information:

- the raw frequency
- the frequency per million words
- the contextual diversity (= the number of contexts in which a word appears [4])
- the percentage of contextual diversity

Country	**Collected Date**	**Blogs**		**Twitter**		**News**	
		#Words	#Docs	#Words	#Docs	#Words	#Docs
English (US)	2012	38.1	899'288	30.9	2'360'148	35.2	1'010'242
French	2011-12	35.2	880'655	28.9	2'023'279	20.1	358'001

Table 1 - *Number of words (in millions) and number of documents in the three new corpora for French and English [3]*

The corpora from which the word frequencies were computed can also be downloaded.

5.2. Identifying words unknown to Kam4D

Our goal is to write a script to identify the words from the frequency list that don't exist in Kam4D. The output of the script is a list of terms unknown to Kam4D sorted by frequency (from the most frequent to the least frequent).

5.2.1. The first version of the script

The first version of the script can be broken down into the following steps:

1. **Connect to Kam4D running locally** on Neo4j → we use the py2neo toolkit to work with Neo4j from within Python applications, such as Jupyter Notebook
2. **Import WordLex list of word frequencies and preprocess them** (words in WordLex are sorted by their frequency in blogs (from the most frequent to the least frequent))
    a. read each line
    b. get rid of the header
    c. tokenize each line
    d. extract words (we don't care about the word frequency stats)
3. **Query the database** to know whether the word exists in Kam4D with queries similar to `MATCH (s:Smurf{lemma_accent:'ragondin'})-[:WRITTEN_IN]->(l:Language {code:'FRA'}) RETURN s,l`

Note: When working on the script, a special focus was given to the **transferability of the code**. I wanted to code functions that could be easily used with datasets in other languages, even though the script has been developed and tested on a French dataset.

5.2.2. Execution time & Query performance tuning

With my configuration (i5, 8GB of RAM), running the code takes time. This is why, in the script, we only run the search among the 100 most frequent terms (which took me 17.2s), instead of the 187'814 words in the list. However, it's already enough to show that the script is working.

In order to improve our query performance, we analyzed the execution plan of our query. It is provided by Neo4j when we use the `EXPLAIN` Cypher instruction. For example, with the following query:

```
EXPLAIN
MATCH(s:Smurf{lemma_accent:'programmation'})-[:WRITTEN_IN]->
(l:Language {code:'FRA'}) RETURN s,l
```

we print the execution plan for a query asking whether the French word “programmation” exists in Kam4D.

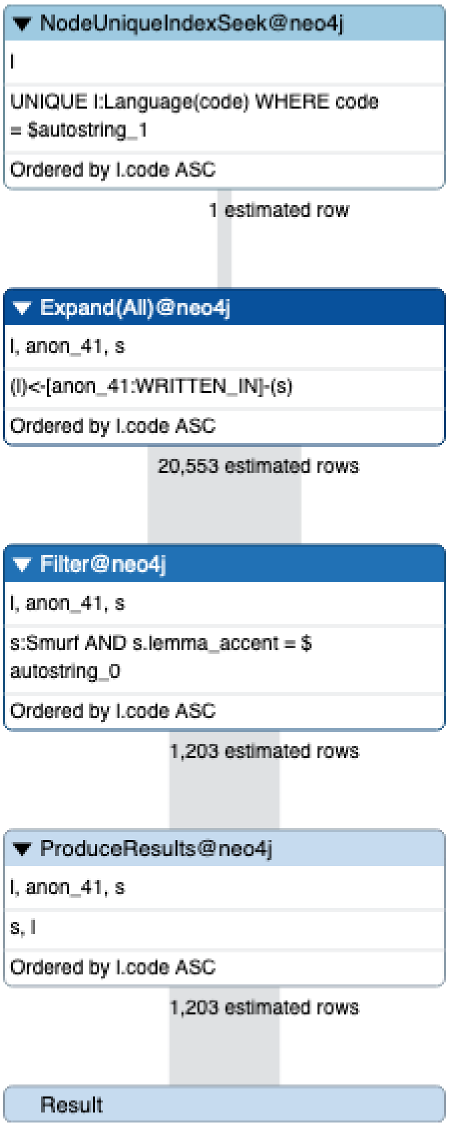


Figure 11 - *Execution plan without index*

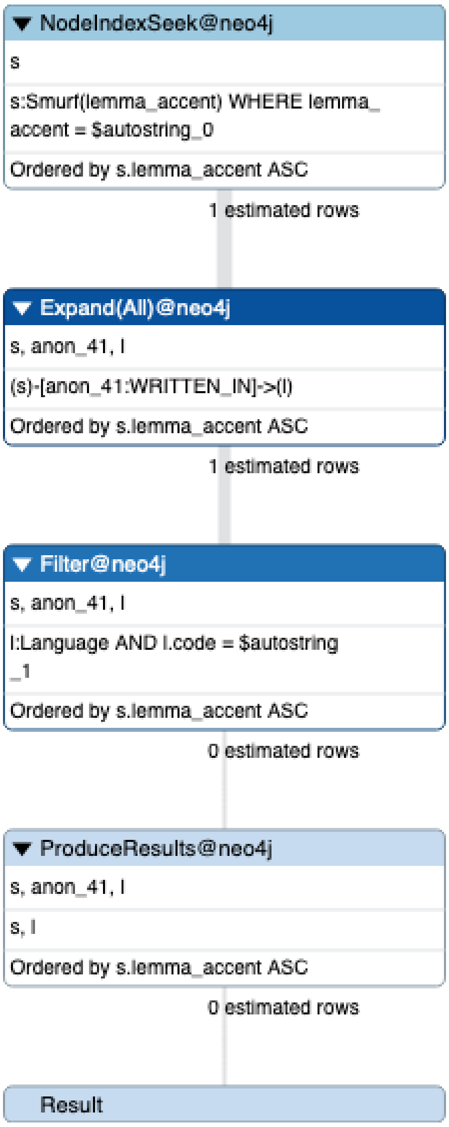

NodeIndexSeek@neo4j
s
s:Smurf(lemma_accent) WHERE lemma_ accent = $autostring_0
Ordered by s.lemma_accent ASC
1 estimated rows
Expand(All)@neo4j
s, anon_41, l
(s)-[anon_41:WRITTEN_IN]->(l)
Ordered by s.lemma_accent ASC
1 estimated rows
Filter@neo4j
s, anon_41, l
l:Language AND l.code = $autostring _1
Ordered by s.lemma_accent ASC
0 estimated rows
ProduceResults@neo4j
s, anon_41, l
s, l
Ordered by s.lemma_accent ASC
0 estimated rows
Result

Figure 10 - *Execution plan with index*

In Figure 11, we see that the second step of the execution plan is an ExpandAll command on 20’553 estimated rows. As shown in Figure 10, this can be drastically reduced to 1 if we create an index for the lemma_accent property of smurfs using the following command:

```
CREATE INDEX ON :Smurf(lemma_accent)
```

This is because in Neo4j, indexes are used to find the starting point of the queries. As a result, using an index avoids us to read too many smurfs to find the one we are looking for.

If we execute again the `detect_unknown_words()` function to find the terms from the 100 most frequent words in French that aren't in Kam4D, it only takes 1.46s to complete. This is more than 11,7 times faster!

5.2.3. Which types of words are missing in Kam4D?

The output of this first version of the script is the following list :

```
['de', 'et', 'le', 'à', 'l', 'les', 'en', 'd', 'des', 'que', 'une', 'je',
'du', 'il', 'dans', 'a', 'ce', 'qu', 'ne', 'au', 'sur', 'j', 'c', 'n', 'on',
'mais', 'se', 'avec', 's', 'nous', 'vous', 'ai', 'ou', 'elle', 'sont', 'me',
'cette', 'sa', 'mon', 'aux', 'ont', 'ça', 'ils', 'm', 'ses', 'ces', 'leur',
'était', 'sans', 'lui', 'suis', 'ma', 'donc', 'tous' 'où', 'alors', 'quand',
'moi', 'autres', 'peut', 'mes', 'entre', 'avait', 'tu', 'fois', 'cela',
'notre']
```

To be honest, I was shocked that 68 out of the 100 most frequent words in French were unknown to Kam4D! If we have a look at these unknown words, we find many:

- prepositions such as “et”, “à”, “en”, “avec”
- pronouns such as “il”, “j”, “je”, “elle”, “ça”, “m”
- conjunctions such as “où”, “mais”
- conjugated verbs (highlighted in blue in the list above) such as “a”, “ont”, “peut”

This makes sense because French data in Kam4D comes from Wordnet, which only has nouns, verbs, adjectives, and adverbs (Kamusi has many more parts of speech, but not in the data I'm familiar with). Plus, when the word changes costumes (e.g., if the verb is conjugated), Kam4D isn't able to recognize it because it only has the infinitive form of the verb.

5.3. Finding the costumes of French verbs with Verbiste

We cannot fix all these issues at once. So, we decided to tackle the costumes of French verbs first. Indeed, we found Verbiste, an open source French conjugation system. The package includes a C++ library and two programs that can be run from the command line or from another program, as well as a GNOME interface.

5.3.1. Setting up Verbiste

Building the package from source on Mac wasn't a piece of cake. Please refer to section 8.4 in the Appendix to learn more about the methods I tried and the bugs I had to fix. I eventually made it work partly thanks to MacPorts.

5.3.2. Improving our script to identify words unknown to Kamusi

As explained in section 5.2.3, some terms among the most frequent terms in French were considered as unknown to Kam4D by our Python script. However, “ai”, “ont”, “avait” are all conjugated forms (and therefore costumes) of the verb “avoir” (= to have) in French, that *does* exist in Kam4D.
Therefore, if a word from the list is unknown to Kam4D, we should try to “deconjugate” it using Verbiste to find out whether it’s a costume of a known verb or if it’s really an unknown word or verb.

The tricky part was interfacing Python with Verbiste as its library is in C++. Even though C++ is a programming language I’m comfortable with, I had never wrapped C or C++ for Python. Luckily, we didn’t need to wrap Verbiste’s C++ API for Python in the end as we found a workaround. We take advantage of Verbite’s two commands `french-conjugator` and `french-deconjugator` that we can execute in the terminal. The trick is to use these commands in IPython (i.e, in the Jupyter Notebook) by prefixing them with the `!` character. For example:

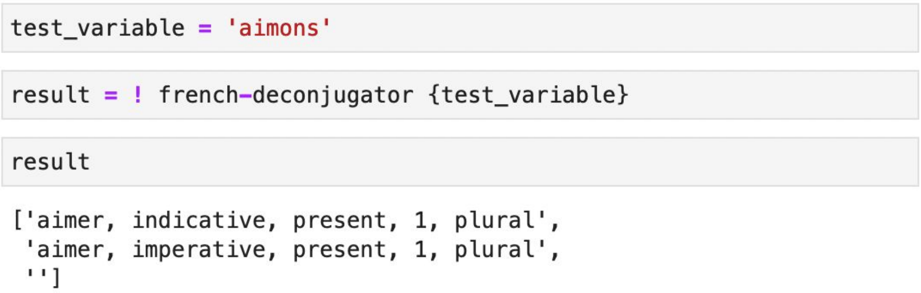
```
verbiste_results = ! french-deconjugator {word}
```

Passing Python variables such as the word / verb to “deconjugate” into the shell is possible using the `{varname}` syntax. In addition, we can save the output of any shell command such as `french-deconjugator` to a Python list using the assignment operator (`=`). For a more in-depth presentation of IPython and shell commands, please follow this link.

For each conjugated form of a French verb matching the input term, the `french-deconjugator` command prints:

- the infinitive form of the verb
- its mode (infinitive, indicative, subjunctive, imperative, conditional or participle)
- its tense (present, future, imperfect or past)
- its person (1st, 2nd, 3rd, male or female)
- its number (singular or plural)

For example, if we give “aimons” as input, `french-deconjugator` returns the following result:



```
test_variable = 'aimons'
```

```
result = ! french-deconjugator {test_variable}
```

```
result
```

```
['aimer, indicative, present, 1, plural',
 'aimer, imperative, present, 1, plural',
 '']
```

Figure 12 - *Code snippet showing how the* `french-deconjugator` *command can be used in Jupyter Notebook*

If the input isn’t a conjugated form of a French verb, the command returns a list with an empty string `['']`.

In the second version of our script to find unknown words to Kam4D, if the query in Kam4D didn’t return a result, we try to “deconjugate” the word using Verbiste. Then:

- if Verbiste matches a conjugated form of a verb to this word, we extract the infinitive of the verb and query again Kam4D using this infinitive → if the query still returns no result, we add the word to the unknowns list
- if Verbiste doesn’t match a conjugated form of a verb to this word, we add the word to the unknowns list

The output list generated by the second version is shorter:

```
['de', 'et', 'le', 'à', 'l', 'les', 'en', 'd', 'des', 'que', 'une', 'qui',
'je', 'du', 'il', 'dans', 'ce', 'qu', 'ne', 'au', 'sur', 'j', 'c', 'n', 'on',
'mais', 'se', 'avec', 's', 'nous', 'vous', 'ou', 'elle', 'me', 'cette', 'sa',
'mon', 'aux', 'ça', 'ils', 'm', 'ses', 'ces', 'leur', 'sans', 'ma', 'donc',
'tous', 'où', 'alors', 'quand', 'moi', 'autres', 'mes', 'fois', 'notre']
```

We can see that all the conjugated verbs included in the output list of the first script disappeared. This is exactly what we expected as these words were costumes of basic verbs which infinitives already exist in Kam4D.

5.3.3. Adding the costumes of French verbs to Kam4D

We decided to go one step further and solve the problem at the root. Instead of using Verbiste to deconjugate costumes of verbs, we could add them to Kam4D once and for all. This way, if we query Kam4D with a conjugated form of a verb, the database automatically detects that it is a costume of a verb and returns the smurf associated with this verb’s infinitive.

We use the second built-in command of Verbiste: `french-conjugator`. It conjugates French verbs to all existing simple tenses (e.g., present of the indicative, imperfect of the indicative, present of the conditional), but doesn’t include “composed tenses” (“temps composés” in French) such as “passé composé” or “plus-que-parfait” (as the auxiliary verb used to build them depends on the verb and sometimes on the context). A code snippet showing the output of `french-conjugator` for “coder” can be found here.

Our script can be broken down into the following steps:

1. **Extract verb infinitives from XML file** (`verbs-fr.xml` in Verbiste’s `data` subdirectory) → it includes 7015 lemma
2. **Create template matching the output structure** of the Verbiste `french-conjugator` command
3. **Conjugate each verb from the XML file** using `french-conjugator` and **add costumes to template table**
4. **Append filled in table to dataframe** containing all the costumes of all verbs

The panda dataframe is then converted to a CSV file with the following columns:

- infinitive form of the verb
- conjugated form (1 per row)
- mode (possible values: inf / sub / impe / ind / cond / part)
- tense (possible values: pres / fut / impa / pas)
- person (possible values: 1st / 2nd / 3rd / m / f)
- number (possible values: sg / pl)

Please follow this link to download the CSV file including 350’751 costumes generated by the script. I then sent the output CSV file to Jérôme Baton who took care to import the data to Kam4D.

Unfortunately, I spilled a glass of water on my computer during the Christmas break. It killed it. All my files were saved in the cloud except for the Python script to generate the CSV file. I apologize for not being able to provide it.

5.4. Extracting meaningful example sentences from French corpora

Back in section 5.1, we mentioned that the corpora from which the word frequencies were computed were also made available. These documents constitute a good source of example sentences.

In addition, we found Lexique 383, a new dataset for French with documentation. It includes 142’694 entries with 47’342 unique lemma. The dataset is quite rich as in addition to the word, it includes 33 supplementary columns with further information such as the word “grammatical class” (“classe grammaticale” in French, abbreviated as “cgram” in the dataset).

We focused on rare words that only appear 3 times in the entire WordLex corpus. Example sentences are especially valuable for these words as they usually have a specific usage context. Plus, we don’t get overwhelmed by hundreds of potential example sentences.

The script to code has 3 main steps:

1. **extract words from WordLex that appear 3 times** (in the entire corpus, newspaper, Twitter and blog posts combined)
2. **check if these words exist in Lexique 383** (sanity check because Lexique 383 is way “cleaner” than WordLex that has a lot of garbage (typos, words from other languages than French, numbers, emojis, etc.)
3. if it’s the case, **extract example sentences from** the **WordLex** corpus

To parse the documents, we used the `sentence-splitter` package. It relies on a heuristic algorithm trained for different languages, including French, to split plain text into a list of sentences.

In addition, extracting sentences for each word appearing 3 times in the corpus and included in Lexique 383 takes time (approximately 23 seconds for 10 words, and we have more than 5’000 to process)! Therefore, I created a version of the script that could run using Google Colab. This allowed me to execute our script for free on Google’s cloud servers, instead of heating up my laptop.

The sentence splitting wasn’t always successful (see script) but I still managed to extract 13’574 example sentences from the corpus. Please refer to the following CSV file.

6. Next steps

The most urgent task to complete now that Vagrant is fixed would be to implement the new queries for Kam4D’s public interface. Refer to section 4.3.2 for more details.

7. Conclusion

Things didn’t go as planned (but compared to the rest of 2020, they went fine!) and I ended up not working on the project I had proposed at the beginning of the semester nor finishing the update of the database public interface.

Nevertheless, I’m very pleased with how this semester project turned out. I learned so much about graph databases, interfacing back-end with front-end, working in a team of developers, and the list goes on. My struggles to understand the GitLab repository taught me the importance of documentation and READMEs, as well as keeping in mind that code is read much more often than it is written.

In addition, it has been rewarding to contribute to Kamusi’s database by providing:

- a list of frequent words missing in Kam4D
- a list of 350’751 costumes of 7’015 French verbs
- a list of 13’574 example sentences for more than 5’000 french words

and their matching Python scripts available in this GitHub repository.

I would like to end this report by giving a special thanks to:

- Dr. Martin Benjamin for his unshakable support during this entire semester and for pursuing this meaningful endeavor of building a multilingual dictionary;
- and Jérôme Bâton for his clear explanations on Neo4j and for investing his precious free time to build Kam4D’s database.

Thank you for making a difference.

8. Appendix

8.1. Initial project proposal

Semester Project Idea: Learning Useful Vocabulary that Matters to You

Pain point: When reading articles or webpages, language learners are often stymied by the new words they come across. They may understand the main points of the text. However, they usually don't know which new words they should focus on.

POTENTIAL SOLUTION (based on SlowBrew)

Format: Kamusi front-end app
Type of files processed: Texts
Targeted users: People learning a foreign language

Core functionalities

1. Recognize key terms in a written document (i.e., most frequent words in the language or words, terms with ambiguous meanings (such as the "radius" of a circle versus the "radius" bone of an arm, inflected forms when we are able to connect with an NLP toolkit such as Freeling or NLTK, and multiword expressions (including separated multi-word expressions).
2. Build a tool for users to highlight in orange the words for which they're unsure of the meaning and in red those with which they're completely unfamiliar. Simultaneously, we're memorizing the words known by the user and can, therefore, better estimate his/her level in this language. Concepts that the user flags as difficult can also be marked as priorities for inclusion in language learning games elsewhere in Kamusi, together with the usage example from the given text.
3. When users select a word, we can display using Slowbrew the most probable translation in their mother tongue, or allow them to submit a meaning if they're not satisfied with Slowbrew's suggestions.
4. Propose 5 new important words / expressions from the text to the user.
5. Learn which terms are missing from Kamusi data. We will offer the user the opportunity to add as much information about the term as they can, but that is likely outside the competency of learners. More pertinent will be a feature to store the missing terms in our db, along with the original context, so that they can be placed in the queue for advanced speakers to consider the term (conveniently, already attached to its usage example) for inclusion in the lexicon.

Bonus

Once we gathered enough data to precisely estimate the user's level and using the data of other users with a similar language level, we could highlight in a lighter color the words we suspect he/she might not know.

8.2. Connection procedures (development server)

8.2.1. Connecting to the development server

Procedure

1. open terminal and execute `ssh lou@188.138.127.93` (password: LOU)
2. type `exit` to end the connection

8.1.2. Connecting to neo4j on the development server

Procedure

1. connect to the development server
2. execute `cypher-shell -u neo4j` in terminal
3. password = nodes2020

8.2.3. Restarting nginx

Procedure

1. connect to the development server
2. execute `sudo systemctl restart nginx` in terminal

8.3. Personal notes on back-end key concepts

8.3.1. Node.js

General features

- a JavaScript runtime environment built on Chrome's V8 JavaScript engine → everything you need to execute a program written in JavaScript
- uses an event-driven, non-blocking I/O model → asynchronous programming
- npm = Node.js' package ecosystem

I/O (= input / output)

- can be anything from reading / writing local files to making an HTTP request to an API
- takes time and hence blocks other functions (if we're not in a non-blocking I/O model)

What can Node.js do?

- generate dynamic page content
- create, open, read, write, delete, and close files on the server
- collect form data
- add, delete, modify data in your database

`require()` → equivalent in Node.js to `include` in C or `import` in Python

- accepts a parameter "path"
- not part of the standard JavaScript API, but in Node.js
- loads modules that come bundled with Node.js like file system and HTTP from the Node.js API
- loads third-party libraries like Express and Mongoose that you install with `npm`
- lets you require your own files and modularize the project

Modules

- in Node.js, each module has its own scope → a module cannot directly access things defined in another module unless it chooses to expose them
- to expose things from a module, they must be assigned to `exports` of `module.exports`
- for a module to access another module's `exports` or `module.exports`, it must use require()

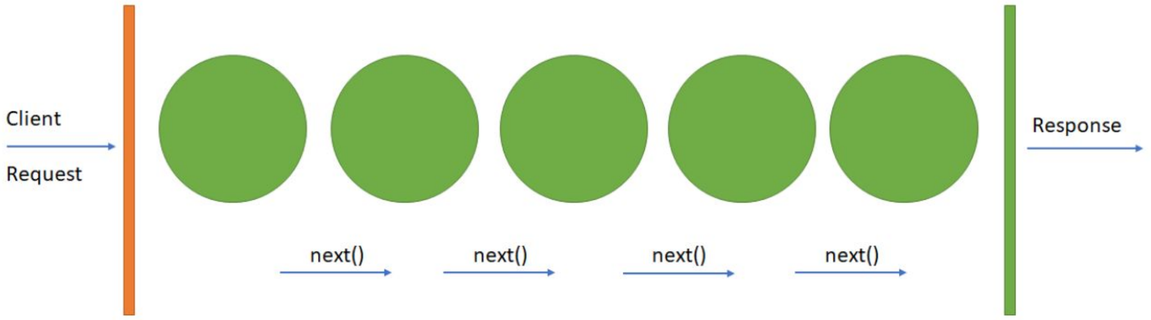
Sources

- *Node.js Introduction* by w3.schools.com
- *What exactly is Node.js?* by Priyesh Patel (Medium)
- *What is "require" ?* (*Stack Overflow*)

8.3.2. Middleware functions

Definition of middleware → functions that have access to

- the request object (`req`)
- the response object (`res`)
- the next middleware function in the application's request-response cycle → commonly denoted by a variable named `next`



Example of a request - response life cycle (green circles = Express servers)

Tasks performed by middleware functions

- execute code
- make changes to the request and the response objects
- end the request-response cycle
- call the next middleware in the stack using `next()` → necessary if the current middleware function does not end the request-response cycle, else the request is left hanging

Types of express middleware (non-exhaustive list)

- application level middleware → `app.use`
- router level middleware → `router.use`

Sources

- *How Node JS middleware Works?* by Selvaganes (Medium) → explanation of the concept
- *Writing middleware for use in Express apps* (Express documentation) → example of middleware functions (code)

8.4. Installing Verbiste

Installation - Method 1

👩🏼💻 installed the Mac OSX Verbiste application that installs automatically the Mac OS X Verbiste command line programs in `/usr/local` but when I try running a command such as `french-conjugator aimer`, I get the following error:

```
zsh: bad CPU type in executable: french-conjugator
```

→ making sure the path is well defined using `export PATH=$PATH:/usr/local/bin` didn't help

→ it might be due to the fact that an incorrect CPU type is written in the executable (the package was originally designed for Mac OS X 10.4, Tiger and a Power PC G4 processor), but I can't find a way to solve this issue

Installation - Method 2

👩🏼‍💻 build package from code source using instruction in INSTALL file

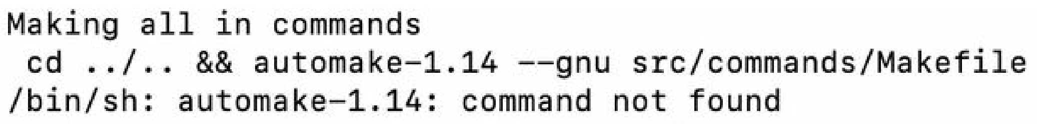
🐛 Error when running `./configure` from verbiste-0.1.47 folder (downloaded from the official Verbiste website)

```
configure: error: Package requirements (libgnomeui-2.0 >= 2.0.0)
were not met:
No package 'libgnomeui-2.0' found
```

Solution

1. install MacPorts (= open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading either command-line, X11 or Aqua based open-source software on the Mac operating system)
2. execute `sudo port install libgnomeui` in terminal (MacPorts wiki)

🐛 Error when running make command



```
Making all in commands
 cd ../.. && automake-1.14 --gnu src/commands/Makefile
/bin/sh: automake-1.14: command not found
```

Solution = in aclocal.m4 file, change am_api_version to ‘1.16’ for Automake & other constants to use the automake version installed on my computer

References

[1] Benjamin Martin, Bâton Jérôme et McKeen Greg. « Lexicography in Four Dimensions: Database Structure for a Matrix of Human Expression Across Time and Space ». (Google Doc)

[2] Baton, Jérôme, et Rik Van Bruggen. *Learning Neo4j 3.x: Effective Data Modeling, Performance Tuning, and Data Visualization Techniques in Neo4j*. Second Edition, Packt, 2017.

[3] Gimenes, Manuel, et Boris New. « Worldlex: Twitter and Blog Word Frequencies for 66 Languages ». *Behavior Research Methods*, vol. 48, nº 3, septembre 2016, p. 963‑72. *DOI.org (Crossref)*, doi:10.3758/s13428-015-0621-0.

[4] Huang, Xin. « The role of word frequency and contextual diversity in visual word recognition: a mini review ». *New Frontiers in Ophthalmology*, vol. 3, nº 6, 2017. *DOI.org (Crossref)*, doi:10.15761/NFO.1000185.