Université de Bordeaux

Master 1 Informatique

Projet de Programmation
Le Clicodrome de LEFFF

par Lionel CLEMENT

Mémoire

4 Avril 2019

réalisé par

BAKIR Fatima Ezzahra

JELLITE Oumayma

NEDELEC Guillaume

SYLLA Alfred Aboubacar

Enseignants responsables : Philippe NARBEL et Vincent PENELLE

Table des matières

Table des figures

1 Introduction

1.1 Contexte

Lionel Clément est un enseignant-chercheur du Labri (Laboratoire Bordelais de Recherche en Informatique) informatique, spécialisé dans le domaine de la linguistique formelle et du traitement automatiques des langues. Il a réalisé avec Benoît Sagot, chercheur dans le même domaine à l'INRIA (Institut National de Recherche en Informatique et en Automatique), le Lexique des formes fléchies du français, appelé le LeFFF.

Avant de détailler l'objectif de notre projet, nous allons mettre au clair quelques notions nécessaires à la compréhension du sujet. On appelle "lemme" un mot qui, de quelque nature que ce soit, qu'il soit composé ou simple, peut être référencé dans un dictionnaire. Dans la langue française, les formes fléchies correspondent aux conjugaisons et aux déclinaisons (forme au pluriel, forme féminine etc...) d'un lemme. Chaque forme fléchie est rattaché à un lemme particulier. Par exemple, "mangeront", "mange" et "mangera" sont trois formes fléchies du lemme "manger".

Le LeFFF contient un grand nombre de mots de différentes catégories (noms propres, des noms communs, des adverbes, des adjectifs, etc...). Lionel Clément le qualifie comme "une ressource complexe constituée de

- Lexème ou grammème (ie Prendre)
- Vocables (ie Prendre=saisir, Prendre=recevoir)
- Catégories syntaxiques (ie Verbe)
- Sous-catégories syntaxiques (ie Transitif, passivable)
- Catégories grammaticales (ie Nombre→ $\{sing, plur\}$, Personne→ $\{1, 2, 3\}$)
- Règles de flexion (ie Table de conjugaison de prendre - Stem=.*(pren|mett))
- Valence (ie objet nominal, oblique en à)
- Réalisation syntaxique (ie passif en par)
- Phraséologie (ie "Prendre ombrage", "Prendre ses jambes à son cou")
- Collocation (ie "Prendre une initiative")
- Fonctions lexicales (ie Magn : "Prendre une belle initiative")

Aujourd'hui ce lexique est présent dans un fichier texte et il n'existe pas d'outils permettant de faciliter son accès, sa modification ou son enrichissement. Le système actuel n'assure pas une bonne traçabilité des modifications effectués par les linguistes et ne permet aucun contrôle de ces changements.

L'objectif de ce projet est donc de transformer le lexique au format texte en une base de données et de développer une interface web, facilitant les interactions avec le lexique via cette base de données. L'avantage de cette solution web est que l'on ne sauvegarde dans la base de données, que les lemmes présents dans le lexique. On n'enregistre pas les formes fléchies afin de réduire considérablement la taille de la base. Les formes fléchies seront ensuite générées automatiquement par notre application à l'aide de règles, qui elles, seront enregistrées dans la base de données. Ces règles respectent le forma-

lisme PFM (Paradigm Function Morphology) qui permet d'associer une forme flexionnelle à un lemme, passé en entrée avec ses propriétés morphosyntaxiques.

Le projet peut être donc séparé en trois parties principales :
- L'importation du lexique dans une base de données
- La création de l'interface web qui interagira avec la base de données
- Le développement d'un interpréteur permettant de générer les formes fléchies d'un lemme donné en entrée selon des règles enregistrées en base de données.

1.2 Description de l'existant

1.2.1 Le lexique : LeFFF

Pour ce projet, nous avons a notre disposition différentes versions du Lefff dont la dernière version été réalisée par Benoît Sagot en 2010. Cette version est disponible dans 2 formats, à savoir [**?**] :
- le lexique **intentionnel**, qui décrit pour chaque entrée, son lemme (forme canonique + tableau d'inflexion) ainsi que des informations syntaxiques profondes (cadre de sous-catégorisation profonde + réalisations possibles + restructurations possibles) .
- le lexique **extensionnel**, construit automatiquement par compilation du lexique intentionnel.

Le lexique extensionnel est présenté sous la forme ci-dessous :

```
démariés adj démarier Kmp
démariés v démarier Kmp
démarqua v démarquer J3s
démarquage nc démarquage ms
démarquages nc démarquage mp
```

La première colonne correspond à un mot du lexique.

La seconde colonne présente la catégorie (**adj**(adjectif), **v**(verbe), **np**(nom propre), **nc** (nom commun)) du mot.

La troisième colonne présente le lemme auquel le mot est rattaché. On peut détecter si un mot est une forme fléchie d'un lemme si le contenu de la troisième colonne est différents de celui de la première.

La dernière colonne apporte des informations sur le mot par le biais de tags. Ces informations peuvent être le genre, le nombre ou encore le temps utilisé pour conjuguer un verbe. Une explication de ces codes est disponible sur le site de Lionel Clément [**?**].

Il existe un format extensionnel plus enrichi en tags que celui présenté ci-dessus. Vous le trouverez sur le site de Lionel Clément (version 2.1) ou de Benoît Sagot (format .elex).

Le format intentionnel contient plus d'informations puisqu'il contient les paramètres morphologiques des mots. Néanmoins, nous ne le détaillerons pas plus car il ne sera pas pris en charge par notre application.

1.2.2 Le formalisme PFM

D'après la documentation [**?**] de Olivier Bonami et Gilles Boyé, le formalisme Paradigm Function Morphology est présenté comme une théorie explicite de la morphologie flexionnelle qui est à la fois inférentielle (qui présente une conclusion à partir d'un fait, d'une situation) et réalisable. Les affixes ne sont pas traités comme des signes, mais comme des résultats de l'application d'une règle liant les caractéristiques morphosyntaxiques à une fonction phonologique qui modifie une base. Dans le formalisme PFM, le système flexionnel d'un langage est modélisé par une fonction de paradigme. Les fonctions de paradigme prennent en entrée une racine et un ensemble de fonctions, et retournent un ensemble phonologique.

La forme générique des fonctions paragigmes est : **PF(l, σ) = IV o III o II o I (l,σ)(ϵ)**

- l : lemme auquel on souhaite appliquer le formalisme
- σ : série de tags que l'on souhaite appliquer au lemme
- IV o III o II o I : Niveaux d'applications des règles

De manière plus formelle, le formalisme indique que pour un lemme donné avec une série de tags, une règle de chaque niveau d'application (et une seule uniquement pour chaque niveau), correspondant aux tags renseignés, va être appliqué au lemme.
Si pour un niveau d'application, aucune règle ne correspond, le formalisme passe au niveau d'application suivant.
Si pour un niveau d'application, plusieurs règles correspondent, celle qui s"applique sera celle qui comporte le plus de tags (la règle la plus spécifique).

Le format de base des règles est de la forme : **n, X, t $\Longrightarrow$ f(X)**

- n : niveau d'application de la règle
- X : lemme
- t : combinaison de tags pour laquelle la règle s'applique (Tous les tags doivent être renseigné par l'appel du formalisme pour que la règle puisse s'appliquer
- f(X) : la forme flexionnelle

Prenons un exemple concret du formalisme PFM avec les règles suivantes :

- I, chaîne,f $\Longrightarrow$ chaîne + "ne"
- II, chaîne, p $\Longrightarrow$ chaîne + "s"

On applique le formalisme sur le lemme "Sien" et lui appliquer le féminin et le pluriel :
PF(Sien, f,p) = IV o III o II o I (Sien, f,p)(ϵ) *Application de la règle I*
= IV o III o II (Sien, f,p)(**Sienne**) *Application de la règle II*
= IV o III (Sien, f,p)(**Siennes**) *Aucune règle à appliquer*
= IV (Sien, f,p)(**Siennes**) *Aucune règle à appliquer*
= **Siennes**

Dans cet exemple, on souhaite appliquer le féminin et le pluriel au lemme "Sien". La syntaxe montre l'application des règles de niveau 1 avant l'application des règles de niveau 2. Cette hiérarchie des niveaux d'application est nécessaire pour éviter toute incohérence. Ce système garantie que le résultat sera "Siennes". Il sera donc impossible d'obtenir "Siensne".
D'autres exemples sont disponible dans la documentation de Olivier Bonami [**?**].

2 Analyse des besoins

Afin d'avoir une vision globale sur le projet, nous avons fait une analyse pour hiérarchiser les besoins du client. Nous avons décrit les besoins fonctionnels (fonctionnalités de l'application) et les besoins non fonctionnels (qualité logicielle, sécurité etc...).

2.1 Analyse des besoins fonctionnels

Pour apporter un contrôle des interactions des utilisateurs depuis le site web, un système de rôle doit être mis en place. Avant de présenter la liste des besoins fonctionnels identifiés, voici une présentation de ces différents rôles utilisateurs, qui vous aidera a mieux comprendre ces besoins :

- **Les visiteurs** : Ils ne peuvent que rechercher, consulter les mots du lexique, signaler des erreurs sur un mot ou exporter le lexique.
- **Les éditeurs** : Ce type d'utilisateur peut, en plus d'avoir les mêmes droits que les visiteurs, modifier un mot (ou ses attributs) du lexique. Ces actions ne peuvent être effectuées qu'en étant connecté (via un système d'authentification) sur le site.
- **Les éditeurs experts** : Ces utilisateurs sont des éditeurs pouvant supprimer définitivement des mots du lexique. Ces utilisateurs peuvent aussi ajouter, modifier et supprimer des catégories et des règles PFM.
- **Les administrateurs** : Ils ont un contrôle total de l'application. En plus d'agir comme des éditeurs expert, ce sont eux qui gèrent les différents utilisateurs du site (validation d'inscription, bannissement d'utilisateurs ou changement de rôle).

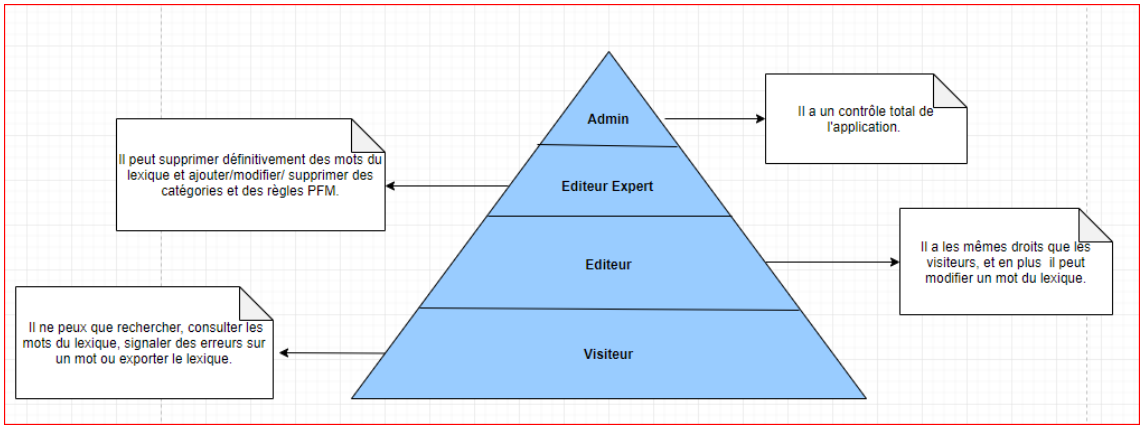


FIGURE 1 – Gestion des rôles

La liste des besoins présentée ci-dessous a été triée par domaine d'action. Les priorités de chaque besoin vont de la priorité 1 (la plus prioritaire) à la priorité 3 (la moins prioritaire).

2.1.1 Gestion du lexique LeFFF

Création d'une base de donnée
Priorité : 1
Il est nécessaire de créer la base de données ainsi que les tables et les relations entre les tables pour qu'elle soit prête à enregistrer les informations du lexique, nécessaires au bon fonctionnement de l'application.

Importer le LeFFF dans la base de données
Priorité : 1
Pour ce besoin, nous devons analyser le lexique qui nous est fourni (au format .txt ou .mlex) et en extraire les lemmes et leurs informations (tags, catégories...) afin de les enregistrer dans notre base de données préalablement configurée. Cette fonctionnalité est nécessaire pour filtrer le nombre d'entrée à enregistrer sur le grand nombre proposé par le lexique (environ 100 000 entrées à enregistrer sur les 500 000 entrées du lexique, dans le leFFF en version 2.1).

Exporter le Lefff
Priorité : 3
Tous les utilisateurs peuvent effectuer une exportation du lexique. Cette fonctionnalité sera disponible depuis notre application web et nécessitera l'utilisation d'un interpréteur PFM pour générer les formes fléchies des mots présents dans la base de données.

2.1.2 Manipulation du lexique

Rechercher un mot
Priorité : 2
Un champ de recherche, disponible pour tous les utilisateurs, permet de rechercher un mot du lexique. A la manière d'un moteur de recherche classique, la liste des mots enregistrés dans notre base de données, correspondant à ce qui a été recherché apparaîtront sur la page. Il sera ensuite possible à l'utilisateur de consulter en détail un mot, de signaler une erreur, de le modifier ou le supprimer (Si les droits de l'utilisateur le lui permet). Si aucun mot ne correspond en base de données, un message d'avertissement apparaîtra sur l'interface.

Consulter un mot
Priorité : 2
Disponible après avoir effectué une recherche, la consultation d'un mot permet d'obtenir un écran détaillant les caractéristiques du mot (catégorie, tags...) ainsi que toutes les formes fléchies générées pour ce mot. Si les droits de l'utilisateur le permet, les boutons d'ajout, de modification et de suppression apparaissent également sur cet écran.

Ajouter un mot
Priorité : 2
Cette partie est réservée au utilisateurs authentifiés (les administrateurs, les éditeurs

et les éditeurs experts). Un formulaire est proposé, où il est demandé de renseigner le mot à ajouter, sa catégorie (parmi celle enregistrées dans la base de données), et ses éventuels tags (saisie libre). Lors de l'ajout, des vérifications seront effectuées afin de vérifier si le mot n'existe pas déjà en base et que tous les champs renseignés ont des valeurs cohérentes.

Signaler un mot
Priorité : 3
Les visiteurs et les éditeurs ont la possibilité de signaler des erreurs sur un mot. Ce mécanisme permet d'envoyer une notification aux administrateurs pour leur faire part d'informations sur un mot en particulier. En l'absence de la fonctionnalité de suppression ou de modification (pour les visiteurs), ce service permet à ces utilisateurs de faire remonter des erreurs ou des demandes particulières relatifs à des mots.

Modifier un mot
Priorité : 2
En réutilisant le même formulaire que celui de l'ajout d'un mot, cette fonction de modification permet de modifier les informations d'un mot en pré-remplissant le formulaire avec les informations existantes. Elle est accessible à tous les utilisateurs connectés. Comme pour l'ajout, des vérifications seront effectuées à la soumission du formulaire (champs correctement remplis et doublons en base de données).

Supprimer un mot
Priorité : 2
Les éditeurs experts et les administrateurs peuvent supprimer un mot de la base de données. Il leur suffit simplement de rechercher un mot et de sélectionner l'option "Supprimer". Un message de confirmation apparaîtra pour confirmer la demande de suppression avant de l'exécuter.

2.1.3 Génération des formes fléchies

Priorité : 1
Ce besoin est le besoin principal de notre projet. En effet la génération dynamique des formes fléchies permet de réduire considérablement la taille du lexique à enregistrer dans la base de données. Pour répondre à ce besoin, il nous faut mettre en place un interpréteur permettant de traduire des règles PFM pour en générer des formes fléchies.

2.1.4 Gestion des règles

Les éditeurs experts et les administrateurs ont besoin de pouvoir gérer les règles PFM. En effet ces règles sont nécessaires à la génération des formes fléchies. Il faut donc proposer une interface permettant d'ajouter de nouvelles règles, les modifier ou encore les supprimer.

Ajouter / Modifier une règle
Priorité : 1
Ces écrans proposent un formulaire contenant les informations spécifique de la règle (niveau d'application, résultats ect...). dans le cas de la modification, les champs seront pré-remplis des informations existantes. Des vérifications seront effectués à la soumission du formulaires (validité des champs, gestion des doublons).

Supprimer une règle
Priorité : 1
Comme pour la suppression des mots, un message de confirmation apparaîtra pour confirmer la demande avant de l'exécuter.

2.1.5 Gestion des combinaisons de tags

Les règles PFM prennent des tags en paramètres pour détecter si elles doivent s'appliquer ou non. Les combinaisons de tags correspondent à des séries de tags appelé sur un mot pour génerer une forme fléchie. Les règles s'appliquent sur le mot si les tous ses tags (tags de la règle) sont reseignés dans les tags de la combinaison.

Prenons un exemple simple avec des règles :
- I, root, {a} $\Longrightarrow$ root + RES1
- I, root, {a,b,c} $\Longrightarrow$ root + RES2
- III, root, {b,c,d,e} $\Longrightarrow$ RES3 + root

On appelle ensuite le formalisme PFM avec 3 combinaisons de tags ({a}, {a,b,c,d} et {a,b,c,d,e})
Nous obtenons les cas suivants :
- PF(myWord, {a}) = myWordRES1
  Seul la première règle possède tous ces tags ("a") dans la combinaison renseignés. Seul cette règle s'appliquera.
- PF(myWord, {a,b,c,d}) = myWordRES2
  Dans ce cas, les 2 premières règles correspondent (La troisième possède "e" qui n'est pas renseigné dans la combinaison). Or les 2 règles ont le même niveau d'application. On choisi donc la règle la plus spécifique (avec le plus de tags), soit la seconde.
- PF(myWord, {a,b,c,d,e}) = RES3myWordRES2
  Toutes les règles correspondent, une sélection se fait entre les 2 premières règles comme dans le cas précédent.

Ces combinaisons sont donc nécessaires à la générations des formes fléchies. Si aucune combinaisons n'est renseignées alors aucune forme fléchie ne sera générée. Il y a une équivalence entre les formes fléchies et les combinaisons de tags : une combinaison de tags = une forme fléchie générée.
Nous devons donc ajouter à notre interface, de quoi ajouter des combinaisons, modifier les combinaisons existantes et la possibilité de les supprimer. Ces fonctionnalités ne sont disponibles qu'aux éditeurs experts et aux administrateurs.

Ajouter / Modifier une combinaison
Priorité : 1
Mise en place d'un formulaire avec des vérifications de doublons en base de données à la soumission du formulaire.

Supprimer une combinaison
Priorité : 1
Présence d'un message de confirmation de suppression avant d'exécuter la suppression.

2.1.6 Gestion des catégories

Les catégories permettent de classer les mots, les règles et les combinaisons de tags. En effet les règles qui s'appliqueront pour des verbes ne seront pas les mêmes que celles qui s'appliquent pour les noms communs. Il nous faut donc aussi fournir des interfaces permettant d'interagir avec les catégorie (disponible pour les éditeurs experts et les administrateurs).

Ajouter/Modifier une catégorie
Priorité : 2
Mise en place d'un formulaire avec des vérifications de doublons en base de données à la soumission du formulaire.

Supprimer une catégorie
Priorité : 2
Présence d'un message de confirmation de suppression avant d'exécuter la suppression.

2.1.7 Gestion des utilisateurs

Pour ajouter de nouveau utilisateurs et gérer les rôles de ceux existants, il est nécessaire de fournir une interface pour les administrateurs leurs permettant d'effectuer ces tâches.

Ajouter/Modifier un utilisateur
Priorité : 3
Mise en place d'un formulaire avec des vérifications de doublons en base de données à la soumission du formulaire.

Supprimer un utilisateur
Priorité : 3
Présence d'un message de confirmation de suppression avant d'exécuter la suppression.

2.1.8 Traçabilité des modifications

Priorité : 3
Afin de garder une trace de tous les changements effectués, le besoin de sauvegarder ces changements permet de pouvoir revenir à un ancien état du lexique en faisant les changements inverses de ceux enregistrés.

2.2 Analyse des besoins non fonctionnels

2.2.1 Généralisation de l'architecture

Priorité : 1
Ce projet a été proposé spécifiquement pour le LeFFF. Néanmoins il est important de réaliser l'architecture de notre application web de la manière la plus générale possible. Le but étant que notre solution fonctionne correctement pour n'importe quel langage (en supposant que les données enregistrées dans la base soient cohérentes).

2.2.2 Alerter les utilisateurs

Priorité : 2
Afin d'éviter un maximum les erreurs de manipulations, toutes actions entraînant la suppression d'éléments de la base de données seront précédées de l'affichage de message de confirmation afin d'alerter les utilisateurs de la suppression à venir. La mise en place de logs permettra de tracer les changements si une suppression a été effectuée par inadvertance.

2.2.3 Sécurité

Priorité : 1
Afin de protéger l'intégrité des données du LeFFF dans la base de données, nous sécuriserons la base et l'application web des problèmes de sécurité suivants :

- **L'erreur humaine** : en cas d'oubli de déconnexion, la session de l'utilisateur peut être utilisée pour effectuer des changements non voulus.
  Notre solution : Un système de déconnexion automatique permet de répondre à ce problème en déconnectant l'utilisateur automatiquement après 10 minutes d'inactivité.
- **Les injections SQL**, permettant de récupérer ou d'altérer des données, voir même prendre le contrôle de serveur. L'attaque par injection SQL consiste à injecter du code malicieux SQL qui sera interprété par le moteur de base de données dans un champ de saisie d'une application web.

Notre solution : Afin de se protéger de se genre d'attaque, nous avons choisi de développer l'application web en PHP avec le framework Symfony, qui comprend un l'ORM (Object-Relational Mapping) Doctrine. Cet ORM est une couche d'abstraction de base de données qui permet d'interagir avec le contenu de la base de données en manipulant des objets (informatique) du coté de notre application. Il comprend des modules de sécurité (génération de contraintes, formatage des données envoyées ect...) qui détecte les attaques par injection SQL.

— **Le Cross-Site Scripting (XSS)** : Cette faille permet d'injecter du code malicieux sur une page web. Par exemple sur un forum, si quelqu'un entre en commentaire un code Javascript nuisible, tous les autres utilisateurs qui accéderont à cette page seront impacté par ce code qui s'exécutera sur leur navigateur.
Notre solution : Pour remédier à cela, le framework Symfony possède aussi un système (filtrage des données d'entrées, échappement des données en sortie...) protégeant de ce genre d'attaque afin de filtrer tous ce qui pourrait ressembler à du code. C'est un procédé assez similaire à celui de la protection contre les injections SQL.

— **La falsification de requête inter-sites (CSRF)** : Cette faille permet à un attaquant de forcer ses victimes à effectuer certaines actions sur un site cible, sans qu'elles s'en aperçoivent. Pour cela, les attaquants cherche à ce que leur victime visite une page (en étant connecté) où des scripts malicieux seront exécutes sans que la victime s'en aperçoive à l'aide de balise image chargeant le script par exemple. La victime étant authentifié sur le site, les scripts pourront s'exécuter correctement et corrompre les données du site.
Notre solution : Voici la dernière faille de sécurité à laquelle Symfony nous protège. En effet le framework fait cela en générèrant des jetons ("tokens") au moment de l'authentification, différents pour chaque utilisateur et qui sont stockés dans la session. Ce jeton est ensuite vérifié a chaque action avec la base afin de vérifier que c'est bien une action voulue par l'utilisateur et non une attaque CSRF.

— **Le déchiffrement de données** qui consiste à déchiffrer des données crypté. Cela peut permettre de trouver des mot de passe et donc prendre le contrôle d'une session utilisateur ou bien de découvrir des informations sensibles (informations bancaire, personnelles...)
Notre solution : Pour éviter cela, nous avons décidé de chiffrer les données avec des algorithmes irréversible (une fois crypté, aucune technique connue à ce jour peut décrypter le mot, il faut forcement connaître le mot initial) comme "bcrypt" ou "sha512".

— **L'écoute de communication réseau** : Requêter un site web avec un protocole HTTP permet à des intrus présent sur votre réseau de lire ou modifier le site internet que vous souhaitez consultez et présente donc un gros risque.
Notre solution : Utiliser le protocole HTTPS qui permet de chiffrer les données échangées et ainsi les protéger de toutes interceptions ou modification.

2.2.4 Format de fichier

Priorité : 2
Afin de pouvoir répondre aux besoins d'importation et d'exportation du LeFFF avec la base de données, il nous faut définir les format de fichiers compatibles avec ces fonctions. Nous avons fait le choix d'utiliser les formats texte et mlex (.txt et .mlex) car ce sont les formats disponibles au téléchargement du LeFFF actuellement. Ce choix nous permet de pouvoir exploiter le LeFFF existant sans avoir à le reformatter dans un format particulier.

2.2.5 Performances

Priorité : 3
Au niveau des performances, le besoin se porte sur le temps de génération des formes fléchies. En effet l'algorithme de génération des formes fléchies doit être le plus optimisé possible pour réduire le temps d'exécution. Il en va de même pour l'importation du lexique et l'exportation. Ces performances doivent être optimales pour combler les temps d'exécution des requêtes SQL sur lesquelles nous n'avons pas de contrôle.

2.2.6 Ergonomie

Priorité : 4
Un design très simple et épuré permet de faciliter l'utilisation de l'application par les utilisateurs. Cela leur permet de naviguer dans l'application de la manière la plus naturelle possible.

Des codes couleurs et des icônes distinctifs permettent aussi de guider facilement l'utilisateur dans l'interface (ex : le rouge pour la suppression)

3 Architecture et implémentation de l'application web

3.1 Architecture de la base de données

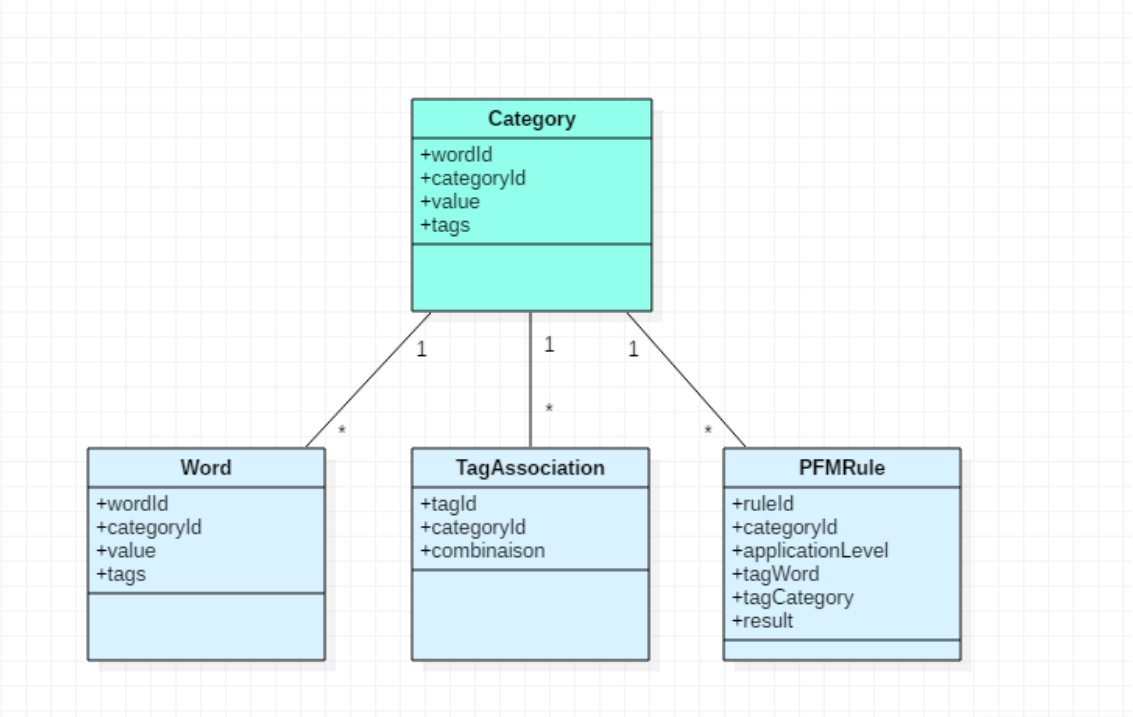


FIGURE 2 – Architecture : Base de données

Notre base de données est constituée de quatre tables :

- **Category** : Cette table contient le code et le nom de la catégorie. Le code est utilisé pour simplifier l'import du lexique puisque le LeFFF existant présente les catégorie avec des codes (par exemple : "adj" pour les adjectifs ou "v" pour les verbes).
- **Word** : Cette table est utilisé pour les mots du lexique. Elle y référence la valeur du mot, les tags qui peuvent lui être associé ainsi que l'identifiant de la catégorie auquel le mot appartient.
- **TagAssociation** : Cette table contient les combinaisons de tags utilisées pour générer les formes fléchies. Ces combinaisons sont liés à une unique catégorie via la clé étrangère "categoryId"
- **PFMRule** :Les règles PFM sont enregistrées dans cette table.
  Les règles sont liés à une unique catégorie par la clé étrangère "categoryId".
  Le résultat de la règle est enregistré grâce à l'attribut "result" (exemple de résultat ajoutant le suffixe "ent" à un mot : "{word}ent").
  "applicationLevel" représente le niveau d'application de la règle.
  Enfin nous avons modifié le fonctionnement du formalisme PFM en dissociant les tags d'une règle en deux parties distinctes :

— ”tagCategory” qui représente les tags de la règle que l’on va comparer avec les combinaisons de tags de la table ”TagAssociation” pour savoir si la règle s’applique.
— ”tagWord” qui représente des tags renseigné dans la table ”Word”. De la même façon que ”tagCategory”, si tous les tags d’un mot renseignés (dans la table ”Word”) sont présent dans l’attribut ”tagWord” de la règle, alors la règle pourra potentiellement s’appliquer. Dans le cas contraire la règle est ignorée par l’interpréteur PFM.

Cette séparation nous permet par la suite d’optimiser les performances de l’algorithme de génération des formes fléchies dont le fonctionnement sera expliqué peu après.

Cette architecture de la base de données permet de généraliser notre application à n’importe quelle langage. Au lieu d’enregistrer des informations sous formes d’attributs dans la base de données, toutes les informations nécessaires à la génération des formes fléchies sont enregistrés sous forme de tags. Cela permet de ne pas restreindre l’utilisation de l’application à un langage spécifique. Le bon fonctionnement de l’application ne dépend ainsi que de la cohérence des données présentes dans la base de données.

3.2 Présentation de l’application web

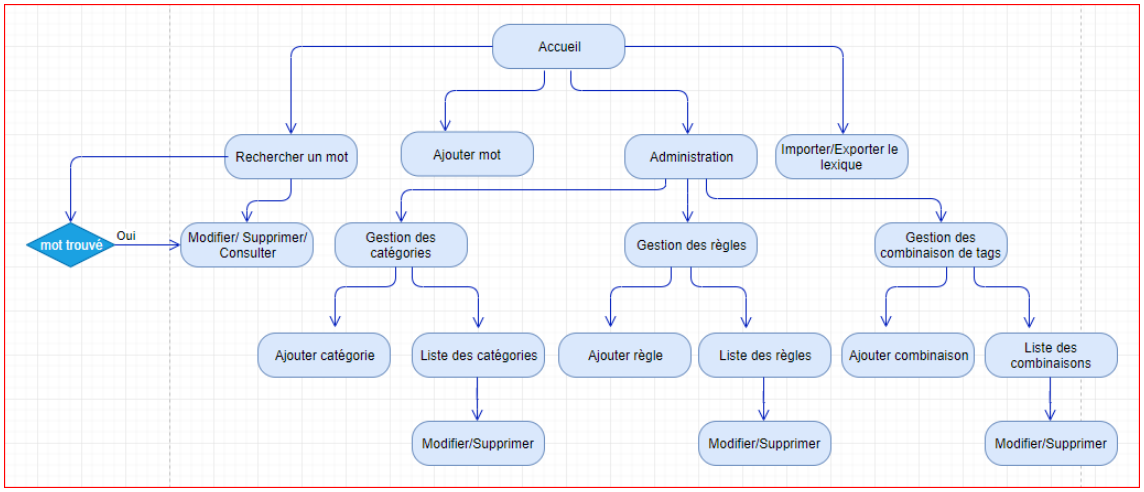


Figure 3 – Accessibilité des fonctionnalités sur l’application web

Ce diagramme présente la navigation entre les différentes fonctionnalités de l’application web.

4 Implémentation de l'application web

Dans cette partie, nous allons vous présenter notre implémentation des fonctionnalités d'import d'un lexique d'export ainsi que l'algorithme de génération de formes fléchies. Le reste des services de l'application est un système CRUD (Create, Read, Update, Delete) sur les données de la base de données. Nous allons vous présenter un exemple de ces services relatifs au mots. Les autres fonctionnalités CRUD fonctionnent sur le même modèle et ne seront donc pas présentées.

4.1 Le système de CRUD

4.1.1 L'écriture : Create

Pour insérer de nouvelles données en base, on crée un nouvel objet correspondant à l'entité souhaiter et on lui ajoute les valeurs voulues. Enfin on enregistre les données en base de données grâce aux méthodes persist($object) et flush() fournies par l'ORM Doctrine.

```php
$word = new Word();
$word->setCategory($category);
$word->setValue($parametersAsArray['value']);
$word->setTags($parametersAsArray['tags']);
$em = $this->getDoctrine()->getManager();
$em->persist($word);
$em->flush();
```

FIGURE 4 – Création d'une nouvelle entrée dans la base de données

4.1.2 La lecture : Read

Pour la lecture, nous utilisons Doctrine, L'ORM de Symfony pour récupérer les données de la base et les renvoyer au client.

```php
$word = $this->getDoctrine()->getRepository( persistentObject: Word::class)->findOneBy(['id' => $idWord]);
```

FIGURE 5 – Lecture des données de la base de données

4.1.3 La modification : Update

Pour modifier des données, on récupère (par la fonction de lecture) l'objet voulu grâce à son identifiant. On effectue ensuite les modifications directement sur l'objet grâce à des setters. Enfin on enregistre les données dans la base grâce aux méthodes persist($object) et flush().

4.1.4 La suppression : Delete

Pour la suppression, les données sont récupérées par lecture. Ensuite on utilise les méthodes de doctrine remove($object) et flush() pour enregistrer les changements.

```
$word = $this->getDoctrine()->getRepository( persistentObject: Word::class)->findOneBy(['id' => $idWord]);
if($word != null) {
    $entityManager = $this->getDoctrine()->getManager();
    $entityManager->remove($word);
    $entityManager->flush();
    $response->setStatusCode( code: Response::HTTP_OK);
    $response->setContent( content: null);
}
```

FIGURE 6 – Suppression d'une entrée dans la base de données

4.2 Implémentation de l'interpréteur

Afin de pouvoir générer les formes fléchies d'un mot, nous avons mis en place un algorithme d'interprétation des règles PFM. Voici une présentation du fonctionnement de l'algorithme

4.2.1 Récupération des données

Dans un premier temps, on se charge de récupérer les données nécessaires à la génération des formes fléchies. Les données nécessaires sont :

- Le mot
- Les tags du mots
- Les règles PFM de la même catégorie du mot
- Les associations de tags de la même catégorie du mot

7

```php
/*** Récupération des règles / tags du mot / combianaison de Tags de la catégorie */
$rules = $word->getCategory()->getRules();
$tagsWord = $word->getTags();
$tagsCombinations = $word->getCategory()->getTagsAssociations();
```

FIGURE 7 – Interpréteur PFM : Récupération des données utiles

4.2.2 Premier filtrage des règles

La prochaine étape consiste à filtrer les règles afin de ne conserver que les règles dont les tags correspondent à ceux renseignés par le mot.8

```php
/*** Filtrages des règles selon la correspondances avec les tags du mot + Récupération des règles définissant le radical du mot*/
$usefulRules = [];
foreach ($rules as $rule) {
    $selected = false;
    //Si TOUS les tags (mots) de la règle sont renseignés dans les tags du mot OU Si aucun tag de mot n'est rensigné dans la règle
    if(!array_diff(explode( delimiter: ";", $rule->getTagWord()), explode( delimiter: ";", $tagsWord)) || empty($rule->getTagWord()) ) {
        $selected = true;
    }

    //Est ce que le mot en lui-même est un tag de la règle
    if(in_array($word->getValue(), explode( delimiter: ";", $rule->getTagWord()))) {
        $selected = true;
    }
    if($selected) {
        array_push( &array: $usefulRules, $rule);
    }
}
```

FIGURE 8 – Interpréteur PFM : Filtrage selon les tags du mot

4.2.3 Début des traitements

A partir de cette étape, on parcourt toutes les combinaisons de tags enregistrés pour la catégorie du mot. Chaque combinaison résultera sur une forme fléchie.

Second filtrage des règles
Pour chaque association de tags renseignés, on réitère l'étape de filtrage des règles en comparant les tags de la règle avec la combinaison de tags courante. A l'issue de cette étape, toutes les règles restantes sont compatibles pour générer une forme fléchie pour la combinaison de tags sélectionnée.9

```php
foreach ($usefulRules as $r) {
    //Si TOUS les tags de la règle sont renseignés dans la combinaisons OU aucun tag n'est rensigné dans la règle
    if(!array_diff(explode( delimiter: ";", $r->getTagCategory()), explode( delimiter: ";", $tags)) || empty($r->getTagCategory()) ) {
        array_push( &array: $ruleToApply, $r);
    }
}
```

FIGURE 9 – Interpréteur PFM : Filtrage selon les combinaisons de tags

Tri des règles par niveau d'application
Parmi les règles restantes, on les tries par niveau d'application pour faire une sélection de la règle à appliquer pour chaque niveau d'application.10

```php
//Tri des règles dans leur ordre d'application (croissant)
usort( &array: $ruleToApply, function(PFMRule $a, PFMRule $b) {
    if($a->getApplicationLevel() == $b->getApplicationLevel()){ return 0 ; }
    return ($a->getApplicationLevel() < $b->getApplicationLevel()) ? -1 : 1;
});
```

FIGURE 10 – Interpréteur PFM : Tri des règles par niveau d'application

Sélection des règles à appliquer
Quand plusieurs règles ont le mêmes niveau d'application, on regarde laquelle est la plus spécifique pour la choisir. S'il existe 2 règles ou plus étant aussi spécifique l'une que l'autre, une exception est levée car aucun choix ne peut être fait.11

```php
//Pour chaque niveau d'application, si il y a plus d'une règles, on choisi la plus spécifique
foreach ($newRulesTab as $sortedRule) {
    if(count($sortedRule)> 1) {
        $selectedRule = $sortedRule[0];
        for($i=1; $i < count($sortedRule); $i++) {
            $nbTagI = count(explode( delimiter: ";", $sortedRule[$i]->getTagCategory()));
            $nbTagSelect = count(explode( delimiter: ";", $selectedRule->getTagCategory()));
            if ($nbTagI > $nbTagSelect) {
                $selectedRule = $sortedRule[$i];
            } else if($sortedRule[$i]->getTagCategory() != "" && $selectedRule->getTagCategory() == "" ) {
                $selectedRule = $sortedRule[$i];
            } else {
                throw new Exception( message: "Error : Conflict beetween 2 rules with the same number of tags.");
            }
        }
        array_push( &array: $ruleToApply, $selectedRule);
    }
    else {
        array_push( &array: $ruleToApply, $sortedRule[0]);
    }
```

FIGURE 11 – Interpréteur PFM : Sélection de la règle la plus spécifique

Application des règles
Enfin, chaque niveau d'application ne possède qu'une règle. Il suffit seulement de les appliquer dans l'ordre en remplaçant la chaîne {word} par le résultats des règles précédentes (en partant d'une chaîne vide à la base) et en y ajoutant les informations complémentaires. 12

```php
foreach ($ruleToApply as $rule) {
    if(strpos($rule->getResult(), needle: "{word}") !== false || strpos($rule->getResult(), needle: "{radical}") !== false) {
        if(strpos($rule->getResult(), needle: "{word}") !== false) {
            $newForm = str_replace( search: "{word}", $newForm, $rule->getResult());
        }
        else {
            $newForm = str_replace( search: "{radical}", $newForm, $rule->getResult());
        }
    } else {
        $newForm = $rule->getResult();
    }
}
```

FIGURE 12 – Interpréteur PFM : Application des règles

4.3 Exporter le lexique

L'algorithme d'export13 du lexique est très simple. En parcourant tous les mots de la base de données, on ajoute une ligne concernant le mot. Ensuite on cherche à générer les formes fléchies du mot et on ajoute une ligne dans le fichier pour chaque forme fléchie générée.
Le fichier d'export respecte ainsi le format ci-contre :

```
<mot> <code-catégorie>["<categorie-name">] lemme="<mot>" {<mot-tags>}
<forme-fléchie-générée> f["forme fléchie"] lemme="<mot>" {<tags-combinaison>}
<forme-fléchie-générée> f["forme fléchie"] lemme="<mot>" {<tags-combinaison>}
<mot> <code-catégorie>["<categorie-name">] lemme="<mot>" {<mot-tags>}
<forme-fléchie-générée> f["forme fléchie"] lemme="<mot>" {<tags-combinaison>}
<forme-fléchie-générée> f["forme fléchie"] lemme="<mot>" {<tags-combinaison>}
...
```

Les tags sont séparés par des points-virgule.

```php
// Pour chaque mot du lexique
foreach ($words as $word) {
    // On ajoute une ligne concernant le mot
    $line = $word . "\t" . $word->getCategory()->getCode() . "[\"" . $word->getCategory()->getName() . "\"]\tlemme=\"" . $word . "\" \t{" . $word->getTags() . "}\n";
    $result .= $line;
    // On génère ses formes fléchies
    foreach ($pfmInterpretor->generateInflectedForm($word, getTags: true) as $inflectedForm) {
        // On ajoute une ligne pour chaque forme fléchie générée
        $line = $inflectedForm["value"] . "\tf[\"forme fléchie\"]\tlemme=\"" . $word . "\" \t{" . $inflectedForm["tags"] . "}\n";
        $result .= $line;
    }
}
```

FIGURE 13 – Export du lexique

4.4 Importer un lexique

L'import d'un lexique est disponible pour deux format de fichiers :

- Le format de l'export (voir ci-dessus)
- Le format mlex consultable sur le site de Benoît Sagot

```
<mot> <code-catégorie> <lemme> <tags-mots>
<mot> <code-catégorie> <lemme> <tags-mots>
<mot> <code-catégorie> <lemme> <tags-mots>
...
```

Le principe de l'algorithme est de consulter chaque ligne et de détecter si la ligne doit être enregistré en base de données. Si le lemme (<lemme>) est égale au mot (<mot>) alors on enregistre l'entrée dans la base de données. On enregistre aussi la catégorie si elle n'est pas déjà référencée dans la base de données. Voici un exemple de code remplissant cette tâche.

```
// Si le mot n'est pas une forme fléchie
if ($word == $lemme) {
    $cat = $doctrine->getRepository( persistentObject: Category::class)->findOneBy(["code" => $category]);

    //formatage des tags
    if($tagsWord != null) {
        $tags = "";
        foreach (str_split($tagsWord) as $tag) {
            $tags .= $tag .";";
        }
        $tags = substr($tags, start: 0, length: -1);
    }

    //La catégorie n'existe pas en base, on l'ajoute !
    if ($cat == null) {
        $cat = new Category();
        $cat->setCode($category);
        $cat->setName($category);
        $em->persist($cat);
    }

    //Ajout du mot
    $newWord = new Word();
    $newWord->setValue($word);
    $newWord->setCategory($cat);
    $newWord->setTags($tags);
    $em->persist($newWord);
}
```

FIGURE 14 – Import du lexique - format MLEX

4.5 Les besoins non réalisés

A l'issue de ce projet, nous n'avons pas eu le temps d'implémenter certains besoins. En effet la gestion des utilisateurs avec des comptes utilisateurs et le système d'authentification n'a pas été mis en place.
L'absence de cette fonctionnalité a engendré l'absence des rôles utilisateurs (visiteur, éditeur, éditeur expert et administrateur). Suite à cette absence, nous avons décidé d'autoriser l'accès à tous les utilisateurs à toutes les fonctionnalités implémentées.
Le signalement de mot n'a pas été mis en place puisque tous les utilisateurs peuvent modifier ou supprimer les données d'un mot.
Enfin la création d'un historique des modifications avec une possibilité d'inverser certaines modifications a aussi été mis de côté afin de privilégier le développement des fonctionnalités principales de l'application web.

4.6 Outils de développement

4.6.1 Application web - Back-end et Interpréteur PFM

Pour la partie back-end de notre application le choix du langage de programmation s'est porté sur PHP [1]. Ce langage, spécifique au développement web, est facile à prendre en main. De plus c'est un langage rapide avec beaucoup de fonctionnalités et de documentation.

Nous avons ensuite choisi d'utiliser le framework Symfony [2] dans la version 4. En effet ce framework intègre de nombreuses fonctionnalités permettant de répondre à certains des besoins identifiés précédemment :

- Doctrine, un ORM (Object Relational Mapping) permettant de manipuler les données de la base sous forme d'objet. En manipulant des objets, on évite les reqûetes SQL brutes, nous protégeant ainsi des injections SQL.
- Des fonctionnalités de sécurité contre les attaques XSS ou CSRF.
- Une architecture MVC (Modèle Vue Controleur) permettant ainsi de bien diviser notre code. En effet le entités sont implémentées séparément, et le contrôleur permet de traiter les actions.
- Une intégration facile de nouveaux services. L'interpréteur PFM ainsi que les méthodes d'import et d'export d'un lexique ont ainsi pu être ajoutées simplement.
- Une grande communauté et beaucoup de documentation permettant d'intégrer des packages existants à notre application pour simplifier le travail (exemple : afin d'éviter des problème de CORS (Cross-origin resource sharing) nous avons utilisé le package "NelmioCorsBundle" [3] développé par "nelmio").

4.6.2 Application web - Front-end

La partie front de l'application a été réalisé en Javascript avec le framework Angular 7 [4]. Ce framework nous a permis de hiérarchiser notre code sous forme de composants réutilisable. L'utilisation de Typescript [5] par Angular a permis, de créer, de manière structurée, des modèle de données. Il permet aussi création de composants et de services à l'aide de commande à exécuter dans un terminal. Un système de gestion des routes permet d'appeler dynamiquement les composants sur la page sans avoir à régénérer le DOM (Pas de rafraîchissement de page à chaque changement d'url) permettant ainsi une navigation dynamique sur l'application.

1. https ://www.php.net/
2. https ://symfony.com/
3. https ://github.com/nelmio/NelmioApiDocBundle
4. https ://angular.io/
5. https ://www.typescriptlang.org/

4.6.3 Base de données

Pour le choix du Système de Gestion de Base de Données (SGBD), nous avons choisi MySQL [6] car il est très simple d'utilisation et il a été recommandé par le client. Ce SGBD s'intègre aussi facilement à notre framework Symfony permettant une configuration très simple. Niveau performance, ce choix est contestable lorsqu'il y un très grand nombre d'entrées. En effet la vitesse d'exécution des requêtes SQL en lecture ou en écriture est plus lente que d'autre SGBD comme PostgreSQL [7] par exemple. Ce choix est donc compréhensible au niveau facilité d'utilisation mais contestable au niveau des performances.

5 Analyse du fonctionnement et tests

Afin d'assurer le bon fonctionnement de notre application et d'être certains qu'elle répond bien aux besoins du client, des tests unitaires et des tests de performances ont été mis en place.

5.1 Test Unitaires - Front-end

Les tests de la partie front-end de l'application ont été réalisé par les outils Karma [8] et Jasmine [9], tout deux intégré dans le framework Angular. Jasmine permet d'écrire des tests qui sont ensuite exécutés par Karma. Ce dernier peut être configuré afin de pouvoir visualiser les résultats des tests dans le navigateur.
Les principaux tests concernent les composants. Ils permettent de vérifier que leur création a bien été réalisée et qu'ils ont bien été initialisés (via l'appel de la méthode ngOnInit()). Ensuite des tests interviennent pour vérifier que les composants ont été correctement générés sur la page web.
Voici un exemple15 de tests du composant ListCategory permettant de tester si les catégories (**CATEGORIES**) sont bien initialisés par le service. Le test *should set Categories correctly from service* vérifie si le nombre de catégorie est de 3 .

```
CATEGORIES = [
  {id: 1, code: 'adj', name: 'Adjectif'},
  {id: 2, code: 'v', name: 'Verbe'},
  {id: 3, code: 'nc', name: 'Nom Commun'}
];
```

6. https ://www.mysql.com/fr/
7. https ://www.postgresql.org/
8. https ://karma-runner.github.io
9. https ://jasmine.github.io/

Enfin, le dernier test *should create a tr for each category* permet de vérifier que le tableau html pour lister les catégories est bien créé.

```
beforeEach( action: () => {
  fixture = TestBed.createComponent(ListCategoryComponent);
  component = fixture.componentInstance;
  mockCategoryService.getCategories.and.returnValue(of(CATEGORIES));
  fixture.detectChanges();
});

it( expectation: 'should create', assertion: () => {
  expect(component).toBeTruthy();
});
it( expectation: 'should set Categories correctly from the service', assertion: () => {
  expect(fixture.componentInstance.categories.length).toBe( expected: 3);
});
it( expectation: 'should create a tr for each category', assertion: () => {
  const tab = fixture.nativeElement.querySelectorAll( selectors: 'tr.mat-row');
  expect(tab.length).toBe( expected: 3);
});
```

FIGURE 15 – Test Unitaire : Liste des catégories

Les tests des services nécessite d'envoyer des données à un serveur et de traiter la réponse. Pour simuler cette réponse, on utilise *HttpTestingController*. Voici un exemple 16 de tests d'un service :

```
describe( description: 'CombinationService', specDefinitions: () => {
  let httpTestingController: HttpTestingController;
  let service: CombinationService;
  beforeEach( action: () => {
    TestBed.configureTestingModule( moduleDef: {
      imports: [ HttpClientTestingModule ],
      providers: [
        CombinationService,
      ]
    });
    httpTestingController = TestBed.get(HttpTestingController);
    service = TestBed.get(CombinationService);
  });

  describe( description: 'getCombinaison', specDefinitions: () => {
    it( expectation: 'should call getCombinaison with the correct URL', assertion: () => {
      service.getCombination( idCombination: 32).subscribe( next: res => {
        expect(res['combinaison']).toEqual( expected: 'imparfait;3ps');
      });
      const req = httpTestingController.expectOne( url: environment.BACK_END_URL + '/get/combination/32');
      req.flush( body: {id: 32, category_id: '30', combinaison: 'imparfait;3ps'});
      httpTestingController.verify();
    });
  });
});
```

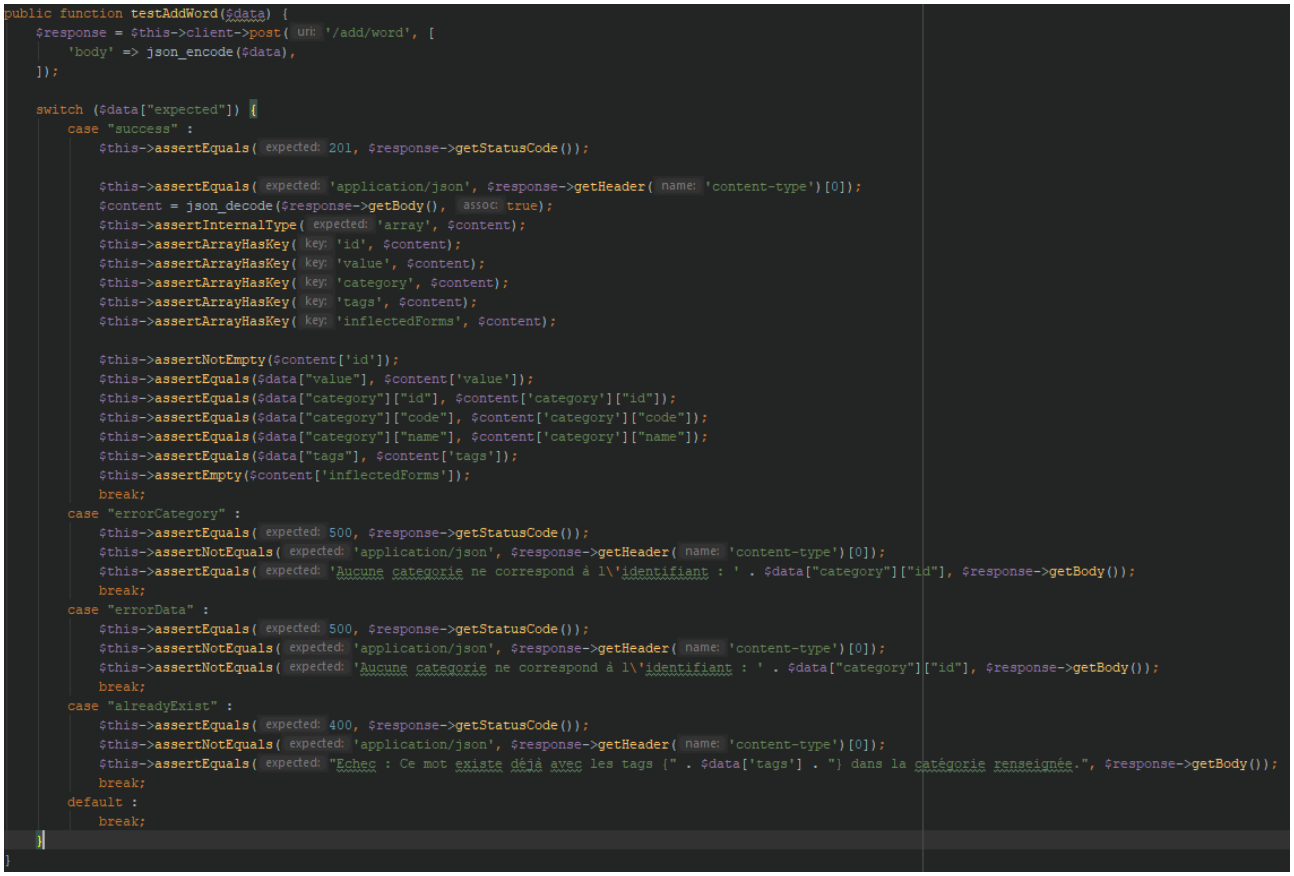
FIGURE 16 – Test Unitaire : Service des combinaisons

Pour vérifier que notre méthode *getCombinaison* qui récupère une combinaison, on appelle cette méthode avec un identifiant et on teste si la réponse attendue est la même que celle renvoyée par notre serveur simulé (*HttpTestingController*). La méthode *exceptOne* permet de tester qu'une seule requête est envoyé. La méthode *verify* permet de vérifier la réponse.

5.2 Test Unitaires - Back-end

Au niveau du back-end de l'application, les tests ont été réalisé à l'aide du framework de tests unitaire PHPUnit [10] et Guzzle [11] permettant de simuler un client pour notre application. Voici un exemple de tests permettant de vérifier le fonctionnement de l'ajout d'un mot :

10. https ://phpunit.de/
11. http ://docs.guzzlephp.org/en/stable/



```php
public function testAddWord($data) {
    $response = $this->client->post( uri: '/add/word', [
        'body' => json_encode($data),
    ]);

    switch ($data["expected"]) {
        case "success" :
            $this->assertEquals( expected: 201, $response->getStatusCode());

            $this->assertEquals( expected: 'application/json', $response->getHeader( name: 'content-type')[0]);
            $content = json_decode($response->getBody(), assoc: true);
            $this->assertInternalType( expected: 'array', $content);
            $this->assertArrayHasKey( key: 'id', $content);
            $this->assertArrayHasKey( key: 'value', $content);
            $this->assertArrayHasKey( key: 'category', $content);
            $this->assertArrayHasKey( key: 'tags', $content);
            $this->assertArrayHasKey( key: 'inflectedForms', $content);

            $this->assertNotEmpty($content['id']);
            $this->assertEquals($data["value"], $content['value']);
            $this->assertEquals($data["category"]["id"], $content['category']["id"]);
            $this->assertEquals($data["category"]["code"], $content['category']["code"]);
            $this->assertEquals($data["category"]["name"], $content['category']["name"]);
            $this->assertEquals($data["tags"], $content['tags']);
            $this->assertEmpty($content['inflectedForms']);
            break;
        case "errorCategory" :
            $this->assertEquals( expected: 500, $response->getStatusCode());
            $this->assertNotEquals( expected: 'application/json', $response->getHeader( name: 'content-type')[0]);
            $this->assertEquals( expected: 'Aucune categorie ne correspond à l\'identifiant : ' . $data["category"]["id"], $response->getBody());
            break;
        case "errorData" :
            $this->assertEquals( expected: 500, $response->getStatusCode());
            $this->assertNotEquals( expected: 'application/json', $response->getHeader( name: 'content-type')[0]);
            $this->assertNotEquals( expected: 'Aucune categorie ne correspond à l\'identifiant : ' . $data["category"]["id"], $response->getBody());
            break;
        case "alreadyExist" :
            $this->assertEquals( expected: 400, $response->getStatusCode());
            $this->assertNotEquals( expected: 'application/json', $response->getHeader( name: 'content-type')[0]);
            $this->assertEquals( expected: "Echec : Ce mot existe déjà avec les tags {" . $data['tags'] . "} dans la catégorie renseignée.", $response->getBody());
            break;
        default :
            break;
    }
}
```

FIGURE 17 – Test Unitaire : Ajouter un mot

Comme présenté sur la figure 17, l'ajout de mot est effectué par des 4 ensembles de données :

- Un ensemble qui fonctionne
- Un ensemble possédant une catégorie inconnue
- Un ensemble avec des valeurs erronée
- Un ensemble avec des données existantes

On peut ainsi voir que tous les cas d'ajout sont testés. Selon les cas, le contenu de la réponse est testé. Le framework PHPUnit nous permet de tester le contenu de la réponse, son statut ainsi que toutes les en-têtes pouvant être renseignées. Par manque de temps, tous les tests n'ont pas pu être implémenter pour cette partie. Néanmoins ils devraient être similaires à ceux implémenter pour le controleur "WordController".

5.3 Tests de performances

Les tests de performances ont été réalisés manuellement. Ils nous ont permis de tester la rapidité des traitements, notamment pour l'import d'un lexique (action prenant actuellement le plus de temps quand l'import prend en compte un grand nombre de données). Le but de nos tests est de détecter si la lenteur du traitement de l'import est due à l'implémentation de nos algorithmes ou par les délais des requêtes SQL (sur lesquels

nous n'avons pas de contrôle).
Nous avons obtenus les résultats suivants (les temps obtenus sont des moyennes de plusieurs tests) :

En traitant l'import sans exécuter de requêtes SQL

- Fichier de 500 lignes dont 499 enregistrements devraient être sauvegardés en base de données : 4.5 secondes
- Fichier de 1000 lignes dont 998 enregistrements devraient être sauvegardés en base de données : 6.4 secondes
- Fichier de 5000 lignes dont 4986 enregistrements devraient être sauvegardés en base de données : 21.8 secondes
- Fichier d'environ 550 000 lignes dont environ 120 000 enregistrements devraient être sauvegardés en base de données : 8.3 minutes

En traitant l'import en exécutant les requêtes SQL

- Fichier de 500 lignes dont 499 enregistrements devraient être sauvegardés en base de données : 13 secondes
- Fichier de 1000 lignes dont 998 enregistrements devraient être sauvegardés en base de données : 20 secondes
- Fichier de 5000 lignes dont 4986 enregistrements devraient être sauvegardés en base de données : 1.4 minutes
- Fichier d'environ 550 000 lignes dont environ 120 000 enregistrements devraient être sauvegardés en base de données : > 2h30

Ces temps ne prennent pas en compte la suppression des données de la base effectués avant l'import. Cette opération peut elle aussi prendre du temps lorsque la base contient un nombre important de données.

Ces résultats nous ont montré que les problèmes de performances ne proviennent pas de nos algorithmes mais bien du très grand nombre de requêtes SQL a exécuter. Ces tests révèlent un inconvénient important dans le choix du système de gestion de base de données (SGBD) MySQL. Néanmoins, notre implémentation de l'application web ne limite pas son fonctionnement à ce SGBD, laissant ainsi la possibilité de changer facilement le SGBD.

6 Conclusion

A l'issue de projet, nous avons fourni une application web répondant aux besoins principaux du client. En effet, malgré quelques besoins non implémentés, notre solution permet de d'importer un lexique dans une base de données et d'en consulter son contenu en en générant ses formes fléchies.

En plus des besoins identifiés mais non implémentés, nous avons détecté des améliorations possibles à cette application :

- L'import et l'export d'un lexique pourrait prendre en charge les règles PFM et les combinaisons de tags.
- En cas de conflit entre 2 règles PFM de même niveau et étant aussi spécifique l'une que l'autre (situation censé être impossible mais aucun contrôle n'est effectué en base pour gérer cela) une erreur pourrait être renvoyée pour avertir l'utilisateur. Actuellement la première règle s'applique par défaut et peut donc générer une forme fléchie incohérente.
- L'utilisation d'un autre système de gestion de base de données pourrait permettre d'améliorer les performances (vitesse d'exécution) des fonctionnalités de génération des formes fléchies, d'import et d'export d'un lexique.

Pour conclure sur ce projet, nous avons eu du mal à comprendre le formalisme PFM et à trouver une manière de l'exploiter efficacement pour répondre à nos besoins. Ces incompréhensions nous ont fait perdre du temps dans l'implémentation de l'application.
De plus, rendre notre solution générique à n'importe quel langage nous a demandé une conception longue et rigoureuse qui a provoqué de nombreux changements dans l'architecture de notre base de données.
Néanmoins, ce projet nous a permis d'en apprendre beaucoup sur le formalisme PFM , la flexion des mots en général et de mettre en application nos compétences en programmation.