Rapport TER M1 : Hotel Advisor

Réalisation d'une interface Web permettant de connaître la polarisation de termes se réferrant à un hôtel.

Étudiants

Bastien Carbonnier
Cédric Durand
Johann Golmard

Encadrants

Mathieu Lafourcade
Pierre Pompidor

Date de début : 9 Janvier 2019
Date de fin : 3 Juin 2019

Table des matières

Résumé

Le projet *Hotel Advisor* consiste à la création d'une application d'analyse d'avis (sur les hôtels) sous architecture MEAN. Pour y parvenir, le projet doit solutionner trois thématique :

- L'extraction de connaissance / text mining : une recherche sur la polarisation de texte et notamment l'analyse des formes négatives et des intensifieurs (adjectifs, adverbes).
- La mise en place de l'architecture MEAN
- La visualisation simple et claire de l'ontologie

Ce rapport regroupe la présentation de l'architecture MEAN, les démarches ainsi que les solutions que notre groupe *Decolaigle* a su trouver sur ces trois thèmes et quelque perspectives d'évolution de notre projet.

Mots-clés : Ontologie, Architecture MEAN, TALN, Scrapper, Polarisation

Remerciement

Nous tenons avant tout à remercier, *P. Pompidor* et *M. Lafourade* pour leurs bienveillances, leurs conseils avisés et leurs encadrements et qui ont permis la réalisation de ce projet.

Nous remercions également toutes les personnes qui nous ont apporté des retours et aidé à la relecture de ce rapport.

1 Introduction

1.1 Sujet

Le projet a pour objectif de réaliser une application qui analyse automatiquement des avis textuels sur des hôtels pour ensuite « colorer »les différents descripteurs des ontologies [1] représentant ces différents hôtels (par exemple : accueil, chambre, matelas, rangements, climatisation, salle de bain, buffet, boissons...).

Un utilisateur de l'application pourra avoir trois rôles :

- un profil d'administrateur
- un profil d'émetteur d'avis
- un profil de gérant d'hôtel

Le profil d'administrateur permet :

- d'instancier une ontologie générale prédéfinie en une ontologie spécifique à un hôtel
- d'accepter ou non les avis émis

Ensuite, le profil d'émetteur d'avis doit pouvoir :

- émettre un avis sous la forme d'un texte

Un émetteur d'avis doit obligatoirement indiquer son nom/prénom, la date de séjour et sa durée dans l'hôtel afin de vérifier la validité de son avis.

Et enfin, le profil de gérant d'hôtel permet :

- de visualiser l'ontologie colorée de son hôtel

1.2 Contraintes technique

Une contrainte technique nous a été imposée pour la réalisation de ce projet. Cette contrainte est d'utiliser l'Architecture MEAN, c'est à dire MongoDb, Node.js et Angular que nous allons expliquer dans les partie dédiées à ces technologies. Pour l'analyse linguistique que nous allons devoir réaliser, aucune technologie nous été imposée. Nous avons donc étudié deux technologies : *TreeTagger* et *Spacy*. Notre choix final après étude s'est porté sur *TreeTagger*.

1. Ontologie : Les ontologies sont des outils qui permettent de représenter précisément un corpus de connaissances sous une forme utilisable par une machine. Elles représentent un ensemble structuré de concepts. Les concepts sont organisés dans un graphe dont les relations peuvent être des relations sémantiques et/ou des relations de composition et d'héritage (au sens objet).

2 État de l'art

2.1 Étude de l'étiquetage Morpho Syntaxique

2.1.1 Présentation

En linguistique, l'étiquetage morpho-syntaxique (également appelé part-of-speech tagging en anglais) consiste à identifier pour chaque mot sa classe morphosyntaxique à partir de son contexte et de connaissances lexicales. Les étiqueteurs grammaticaux sont très nombreux pour les langues germaniques mais plus rares pour le français. Pour notre cas nous avons choisi de comparer les librairies TreeTagger et SpaCy.

2.1.2 Comparaison des librairies *TreeTagger* et *SpaCy*

Données utilisés :

Nous avons effectué ce test sur 15 avis comportant 2331 mots. Ce qui nous intéresse ici est de réaliser une étude relative afin de déterminer le meilleur des deux outils. Pour cela nous avons cherché les mots qui ont été étiquetés différemment (291 mots) afin de leur attribuer une étiquette de référence à la main. Cette méthode ne nous permettra pas de définir la précision globale des deux librairies mais nous saurons ainsi quelles librairies à mieux tagué les 291 mots.

Résultats obtenus :

Sur les 291 termes TreeTagger a une précision de $\simeq 53\%$ tandis que SpaCy a une précision de $\simeq 47\%$. Ces chiffres n'étant pas réellement concluant nous avons donc décidé de rechercher quelles natures grammaticales étaient les plus confondus par les deux étiqueteurs. Pour cela nous avons réalisé le graphique suivant :

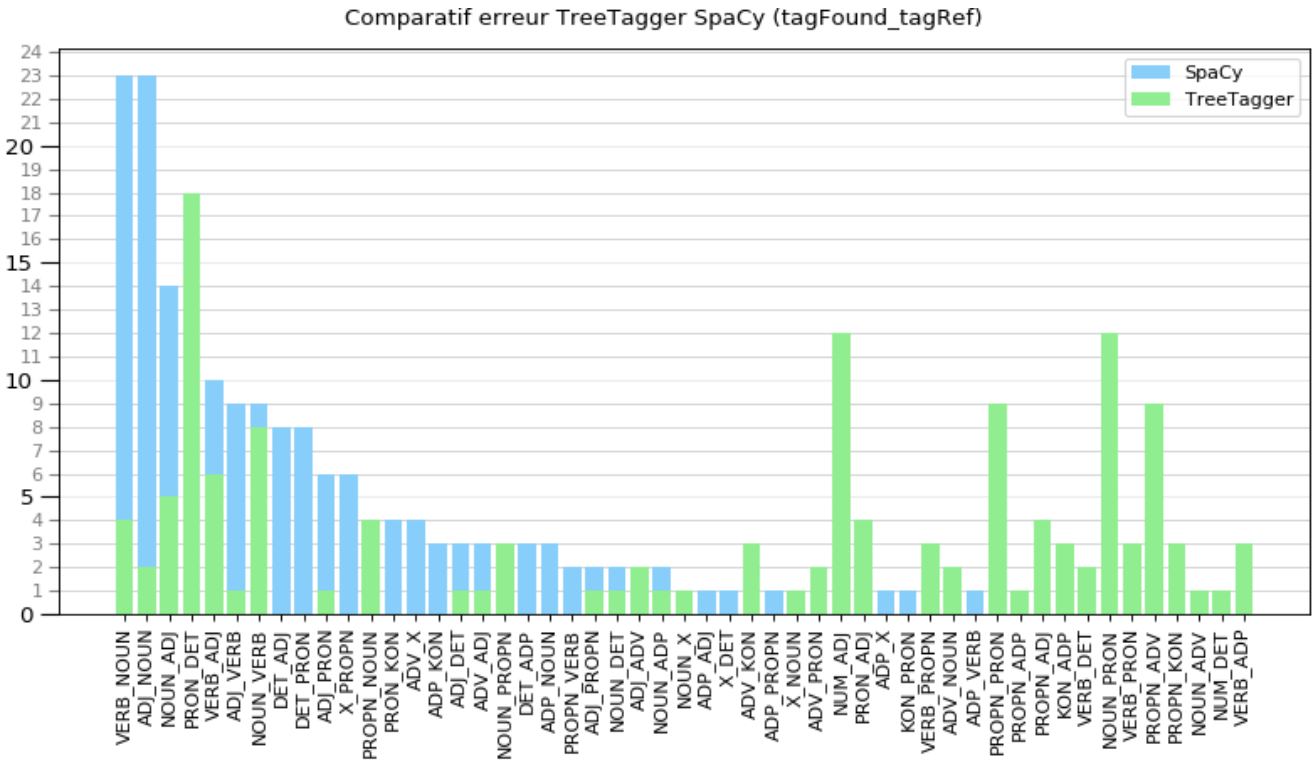


FIGURE 1 – Comparatif TreeTagger Spacy

Interprétation des résultats :

On observe que ces deux étiqueteurs ne font pas les mêmes erreurs. Celles les plus commises par SpaCy sont de prendre des noms pour des verbes, des noms pour des adjectifs et des adjectifs pour des noms. Tandis que celles de TreeTager sont de prendre des déterminants pour des pronoms, des adjectifs pour des nombres (ex : premier) et des pronoms pour des noms. Lors d'une première analyse, il semblait que les erreurs commises par SpaCy pouvaient avoir un impact plus négatif sur nos résultats que celles de TreeTagger.

En effet dans le cas de SpaCy, le fait de confondre un nom pour un verbe et un nom pour un adjectif fausse totalement la syntaxe de notre phrase et de ce fait impactera énormément notre polarisation par la suite. Tandis que dans le cas de TreeTagger qui prend des déterminants pour des pronoms, l'impact sera moindre et potentiellement rattrapable (lexique à rattraper limité).

Afin d'approfondir cette étude, nous avons choisi d'étudier les mots en question afin de déterminer quelles erreurs sont potentiellement acceptables et lesquels ne le sont pas.

Voici un tableau présentant les différents termes des catégories donnant le plus d'erreurs :

Prédictions SpaCy (pred_ref)			Prédictions TreeTagger (pred_ref)		
verb_noun	**adj_noun**	**noun_adj**	**pron_det**	**noun_adj**	**noun_pron**
mer	jardins	junior	chaque	futée	Il
mer	safari	grossier	cet	troisième	tout
déjeuner	buffets	bel	cet	solo	J
déjeuner	armoire	bel	cet	petits	Tous
mer	pente	chaude	cet	moderne	Il
coucher	bars	superbe	les		Il
bar	magasins	gratuite	chaque		Tous
déjeuner	restaurants	cool	cette		Il
forfait	vagues	correct	cet		Ils
mer	lits	médiocre	cet		Il
lever	bouilloire	abordables	le		Il
Suites	restaurant	sympa	cet		Il
vue	restaurants	bel	cet		
serviettes	douche	sympa	chaque		
déjeuner	lieux		cet		
bar	bureau		cet		
déjeuner	bémol		cet		
mer	dommage		cet		
jacuzzis	piscine				
vigil	séjour				
mer	soins				
déjeuner	massages				
fruits	Personnels				

FIGURE 2 – Erreurs commises par TreeTagger et SpaCy

Causes probables et solutions possibles

D'après ces termes on peut observer que les erreurs de l'étiqueteur SpaCy sont très certainement dues à une mauvaise analyse du contexte de la phrase auquel il importe trop d'importance. Certaines de ces erreurs sont peut-être dues à un lexique de départ inexact, car par exemple le terme "jardins" ne peut en aucun cas être un adjectif. Les fautes de SpaCy sont tellement hétéroclite que faire en sorte de les solutionner semble trop complexe et chronophage.

Les erreurs de TreeTagger doivent également être dues à un lexique de départ incorrect mais ces erreurs sont tout de même moins absurdes et plus facilement réparables. Notamment celle prenant des déterminants pour des pronoms qui au final peut être résolue en ajoutant notre propre lexique dans cet outil étant donné qu'il n'y a pas énormément de termes qui posent problèmes.

Conclusion

D'après les résultats, les erreurs commises par la librairie TreeTagger sont facilement rattrapable comparé à celle de SpaCy et modifie moins la sémantique de la phrase.

2.2 Scraping d'avis Tripadvisor

2.2.1 Besoin et étude de l'existant

Pour pouvoir utiliser l'étiquetage Morpho Syntaxique sur des commentaires d'hôtel, il faut en avoir un corpus. Il s'est vite imposé à nous d'utiliser un outil de Scraping [2] pour pouvoir récolter des commentaires sur un site de voyage. Nous avons choisi *Tripadvisor*, qui est l'un des sites de voyage les plus populaires dans le monde et qui possède de nombreux avis Français.

Tout d'abord, avant d'utiliser un outil de Scraping, nous avons regardé si l'API de Tripadvisor pouvait être utilisée pour récupérer les commentaires. Malheureusement, suite à la lecture de la documentation de l'API de *Tripadvisor*, la récupération d'avis sur un hôtel est impossible. Nous avons donc cherché un outil sur Python permettant de palier à ce problème, ce qui nous a conduit à : *Selenium*.

Selenium est un outil d'automatisation de navigateur web qui réalise des scripts dont l'exécution effectue automatiquement des actions programmés dans un navigateur web : cliquer sur des liens, remplir un formulaire, etc. et de récupérer les résultats de ces actions.

Nous aurions pu utiliser un autre outil en python appelé *Scrapy*, facile d'utilisation mais ne pouvant pas intéragir avec le Javascript. Or, dans *Tripadvisor*, un commentaire trop long n'est pas affiché entièrement et un bouton *Plus* s'affiche en dessous. Pour récupérer tout le commentaire, il est donc nécessaire de simuler un clic sur ce bouton *Plus*, d'où l'utilisation de *Selenium*.

2.2.2 Explication de l'algorithme

Le script, présenté dans l'**Annexe A**, démarre en lançant la fonction "parse(url, reponse)" avec *url*, l'url *Tripadvisor* de l'hôtel et *reponse*, le résultat de la fonction "sitedown(url)" qui retourne rien si nous n'arrivons pas à nous connecter sur le site, ou alors le résultat *HTML* de la page de l'hôtel. Dans "sitedown(url)" avant de retourner la page *HTML*, on applique la fonction "plusclick(url)" qui simule les cliques sur le bouton *Plus* de chaque commentaire, et qui retourne l'*HTML*.

Une fois réalisé, on récupère le nombre total de commentaire. *Tripadvisor* n'affiche pas tous les avis sur une seule page mais seulement cinq avis par page.

2. Scraping : Le scraping est une technique d'extraction de contenu de site Web, via un script ou un programme, dans l'objectif est leur utilisation dans un autre contexte.

Pour résoudre cela, on doit étudier le comportement de l'url quand on doit passer d'une page à une autre. On remarque un pattern spécifique par l'ajout d'un "-orX" après le nom de l'hôtel (X étant un nombre multiple de 5) :
https://www.tripadvisor.fr/Hotel_Review-g562819-d287443-Reviews-Occidental_Margaritas-Playa_del_Ingles_Maspalomas_Gran_Canaria_Canary_Islands-or0.html

On effectue alors une boucle qui s'incrémente de 5, allant de 0 juqu'au nombre d'avis et en modifiant l'url pour ajouter "-orY" (Y étant ici l'indice de la boucle) pour couvrir tous les avis d'un hôtel. Chaque url ainsi formée rentre dans une fonction nommée "parse_reviews(url)". Cette dernière fonction s'occupe de récupérer les avis et leurs notations de la page pour les stocker dans une variable globale au script, de type tableau.

Une fois tous les sites traités, un *CSV* est généré.

Cette algorithme nous a permis de récolter plus de 7747 commentaires sur 8 hôtels qui va nous permettre d'extraire les connaissances nécessaires pour l'étiquetage Morpho Syntaxique. Il existe malgré tout certaines limites à l'utilisation de cette algorithme, qui sont :

- le temps d'exécution qui peut varier selon le nombre de pages à traiter pour un seul hôtel *HTML*
- l'algorithme est dépendant des noms de balises *HTML* que *Tripadvisor* met sur son site

2.3 Réalisation d'une ontologie spécifique à l'hôtellerie

2.3.1 Choix d'une ontologie pertinente

Le but premier de notre projet étant d'afficher à l'utilisateur une ontologie dont les termes sont coloriés selon leur polarité, il nous fallait donc établir une ontologie pertinente, dont les termes sont restreints à notre domaine : l'hôtellerie. Une courte recherche sur le Web permet de mettre à jour tout un tas d'ontologies, dont la sémantique, l'héritage, voir même le domaine entier, ne correspond pas à nos attentes.

Fort heureusement notre encadrant *P. Pompidor* a mis à notre disposition une ontologie quasiment prête à être utilisée, après formatage des "enfants" de chaque terme, et de l'ontologie complète afin de la lier à un hôtel spécifique dans notre base de données. Cette ontologie est donc au format JSON, afin d'assurer son fonctionnement dans notre base de données MongoDB, sans oublier la multitude d'avantages que procure ce format objet, comme la lisibilité, et sa représentation sous forme d'arborescence qui illustre parfaitement l'essence même de notre ontologie.

2.3.2 Représentation de l'ontologie

Une fois l'ontologie déterminée et formatée, il nous fallait désormais trouver un moyen d'afficher cette ontologie à l'utilisateur, tout en restant lisible, dynamique et intuitive. Afin de rester cohérent avec l'ontologie, nous avons décidé de l'afficher à l'utilisateur sous forme d'arborescence. Pour cela plusieurs outils s'offraient à nous, dont *ngx-chart* et *D3.js*.

ngx-chart est un module Angular se basant sur la librairie *D3.js*, permettant d'afficher et manipuler des données selon plusieurs type de graphiques, dont le graphe orienté qui s'apparente le plus à ce que nous cherchions. Cependant, ce module restreint les interactions et la personnalisation des graphes, ce qui peut nuire à l'expérience de l'utilisateur et également notre affichage.

Nous avons donc porté notre choix sur la librairie dont elle s'inspire, et que nous avons rapidement survolé au cours de notre premier Semestre : *D3.js*. Cette librairie inclus de nombreux sous-module afin de travailler sur des balises *SVG* dans le code *HTML* d'une page. Elle offre ainsi toute une panoplie de choix pour représenter notre ontologie, sans aucune limitation, mis à part une documentation assez conséquente. Elle est également disponible en tant que module sur Angular.

Grâce à cette librairie, nous sommes parvenus au résultat présent à l'**Annexe N**. Selon la position du texte sur un noeud, l'utilisateur peut rapidement voir si ce noeud possède des noeuds fils, ou non, qu'il peut afficher avec un simple clic sur le noeud en question. Il peut également replier tous les noeuds fils, jusqu'à ne laisser affiché le noeud racine : "hôtel". La coloration d'un noeud indique la polarité de ce dernier, en fonction des commentaires présents dans la base de données et préalablement validés par un administrateur, et la moyenne des polarités de ses noeuds fils. Ainsi, si le noeud "hôtel" viendrait à se retrouver rouge, cela voudrait dire que la majorité des noeuds fils ont une polarité essentiellement négative.

Conclusion

L'ontologie que nous a fournit notre encadrant nous sert d'ontologie pertinente pour notre domaine d'étude, après formatage pour qu'elle puisse répondre à nos contraintes d'architectures. Afin de la représenter à l'utilisateur, nous avons opté pour *D3.js* afin de réaliser les fonctionnalités qui permettront d'afficher l'ontologie selon nos besoins, et d'assurer les fonctionnalités qui garantissent une expérience utilisateur agréable, sans compromettre le sujet du projet.

3 Présentation des différents outils utilisés

Cette partie présentera les différentes technologies utilisées sur ce projet. L'architecture MEAN étant imposée, il nous parait important d'expliquer leurs qualités pour notre projet

3.1 Jeux de Mots

JeuxDeMots est une plateforme dont le mécanisme principal suit le modèle du Game with a purpose (GWAP). Elle a été mise au point par une équipe de recherche en TALN du LIRMM pour construire un réseau lexico-sémantique [3] de la langue française. On y trouve de nombreux jeux tous orientés vers la production ou la validation de données lexicales et sémantiques.

Afin d'accéder aux données présentent sur JeuxDeMots nous utilisons le point d'accès au dump (rezo_dump) qui nous permet de récupérer pour un terme donné tous les noeuds et relations qui lui sont associé. Nous pouvons également filtrer les résultats en choisissant les types des relations à garder.

FIGURE 3 – Logo JeuxDeMots

3.2 L'architecture MEAN

L'architecture MEAN utilise quatres acteurs : le SGDB *MongoDB*, un serveur *Node.js*, le framework *Express* permettant de construire une application Web basée sur *Node.js* et le framework *Angular*.

Cette architecture permet :

- de découpler la partie client et la partie serveur, le client appelant le serveur grâce à des web services REST ;
- d'accéder aux données grâce au serveur *Node.js* et le serveur de bases de données NoSQL *MongoDB* ;
- de créer des pages web modulaire et réutilisable grâce à *Angular* ;

3. Un réseau lexical est une structure qui recense les relations lexicales et sémantiques qui existent entre les mots d'une même langue. Un tel réseau est un graphe, dont les noeuds correspondent à des termes et les arcs à des types de relations entre ces termes.

- et enfin, de développer toute cette architecture autour d'un seul langage, le *JavaScript* et de son extension *TypeScript* permettant une programmation par objet.

3.2.1 MongoDB

MongoDB est un système de gestion de base de données créé en 2007 faisant partie de la famille NoSQL (Not only SQL). *MongoDB* stocke les données sous le format BSON (Binary JSON) qui permet de sérialiser des objets *JavaScript* en binaire dans ce que l'on appelle un document. Un document est la représentation d'un objet *JavaScript* dans une collection. Un exemple d'une collection de documents est présent en **Annexe B**, correspondant aux hôtels de notre application.

Ce système a pour avantage de pouvoir gérer une grande volumétrie de données car on peut répartir une base de données sur plusieurs serveurs tout en assurant des accès rapide. C'est très utile pour notre projet, car lors de son déploiement, notre BDD va devoir stocker des milliers et des milliers d'avis, ainsi que de nombreuses ontologies.

MongoDB a aussi comme intérêt de permettre la gestion de données hétérogène qui sont difficile à pré-déterminer surtout aux niveaux des attributs si nous devions utiliser une base de données classique. C'est utile dans notre situation pour l'ajout de nouveau attribut dans une ontologie sans changer de logique métier.

3.2.2 NodeJs

NodeJS, créé en 2009, est un environnement construit sur le moteur *JavaScript V8* de *Chrome*. Il a été développé pour construire des applications web basées coté serveur en utilisant du *JavaScript*. Son architecture est modulaire, événementiel, principalement orienté réseau. Grâce à ses caractéristiques, il peut remplacer aisément un serveur web tel qu'*Apache*.

Node.js est mono-thread, ce qui veut dire qu'aucun code n'est exécuté en parallèle. On peut s'interroger sur l'utilisation d'un unique thread, dans le cas où l'ensemble du serveur est bloqué si un processus lourd est exécuté. Cependant Node.js gère les requêtes avec des événements. Cette spécification garantit que les processus ne se perturbent pas. Un processus qui demande beaucoup de ressources ne peut pas bloquer l'ensemble du serveur. Par conséquent, on évite ainsi la problématique dite de "*bound I/O*" [4]

Il possède donc deux grandes qualités :

- sa légèreté due à sa modularité ;

4. La problématique *bound I/O* explique que la latence globale d'une application est à cause de la latence des accès aux données, avant tout autre cause.

- son efficacité due à son architecture mono-thread.

Deux qualités importantes pour pouvoir manipuler efficacement tous les commentaires et ontologies dans notre projet.

3.2.3 Angular

Angular est un framework *JavaScript* développé par Google, créé en 2016 et qui permet de créer des applications web dites monopages [5]. Il est une réécriture complète de *AngularJS* et reprend certaines de ses fonctionnalités, telles que l'injection de dépendance, les directives (des composants pouvant être manipulés comme des balises), le système de routage, etc.

Il amène les améliorations suivantes :

- un système de module/composant. Une application contient des modules qui eux-mêmes peuvent contenir des composants. Un module correspond à une grande fonctionnalité, regroupant plusieurs composants, alors qu'un composant est une brique logicielle.
- de nouvelles directives.
- une amélioration du système de routage.
- une utilisation d'extension du langage *JavaScript* comme *TypeScript* permettant de programmer en utilisant des classes, interfaces, mais aussi de sécuriser la production de code JavaScript.

Nous n'allons pas préciser toutes les spécifications d'*Angular* mais allons parler de son paradigme important qui est le paradigme Modèle-Vue-Contrôleur **MVC**.

Il contient trois types de modules ayant des responsabilités différentes : les modèles, les vues et un contrôleur.

Les modèles représente les données à afficher et délivrées par le serveur *NodeJS* depuis une base de donnée *MongoDB* ou gérées localement. Ils représentent la logique métier de l'application.

Les vues contiennent la présentation graphique de l'application. Dans *Angular*, les vues sont des compositions de "sous-vues" associées aux composants ayant un rendu graphique.

Enfin, le contrôleur qui lui possède la logique concernant les actions effectuées par l'utilisateur. Pour *Angular* le contrôleur prend la forme d'un router qui invoque des composants par rapport à une demande de traitement effectuée défini par des route écris en *REST*.

5. Application web monopage est une application qui n'utilise qu'une seule page web. Le but de ce méchanisme est d'éviter le chargement d'une nouvelle page à chaque action demandée.

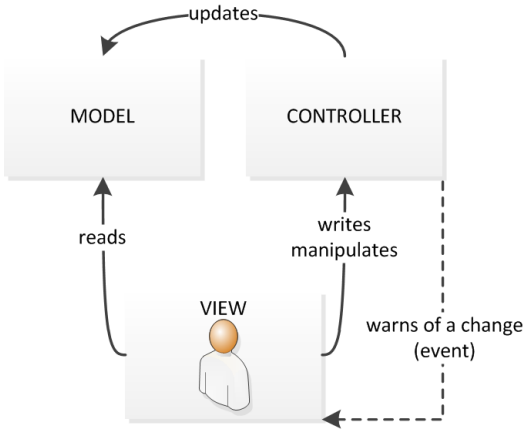


FIGURE 4 – Modele MVC

Angular possède donc plusieurs points fort pour notre projet. Tout d'abord, son découplage fort entre le client et le serveur permettant un développement plus rapide. Ensuite, sa modularité induite par ses modules et composants. Et enfin, le modèle MVC qui donne une bonne évolution structurelle de l'application et simplifie les tâches à développer.

4 Architecture du projet

4.1 Schéma global

Le projet a été décomposé en trois sous-projet.

Un sous-projet contenant l'application *Angular* nommé *Interface_ Hotel_ Advisor* et qui communique avec les deux autres au moyen de requête REST.

Un sous-projet *WebService_ TALN* qui est un serveur *NodeJS* et qui renvoi la polarisation d'un avis.

Et enfin, le sous-projet *WebService_ Reviews*, un autre serveur *NodeJS*, qui s'occupe de traiter les requêtes REST de l'application *Angular* et qui communique avec la base de donnée *MongoDB* pour traiter les opérations sur celle-ci. Ce serveur renvoi les résultats des différentes requêtes à l'application.

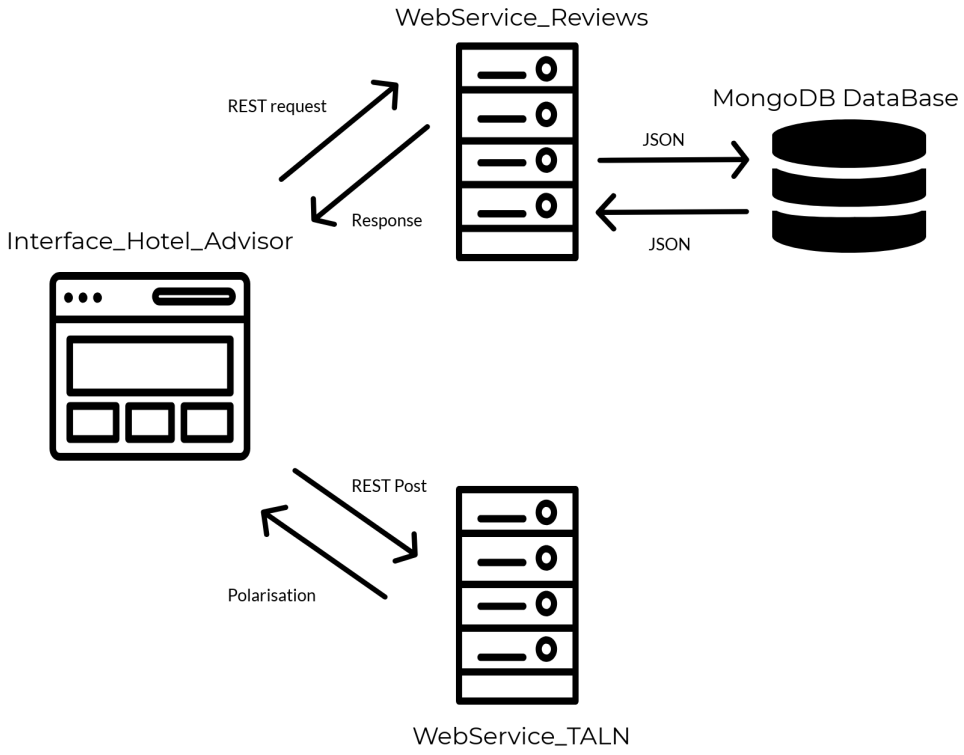


Figure 5 – Schéma global du projet

4.2 Webservice base de données

4.2.1 Schéma de la BDD

Le projet met en perspective 4 tables permettant de modéliser l'application : la table user, la table hôtel, table ontologie et la table commentaire.

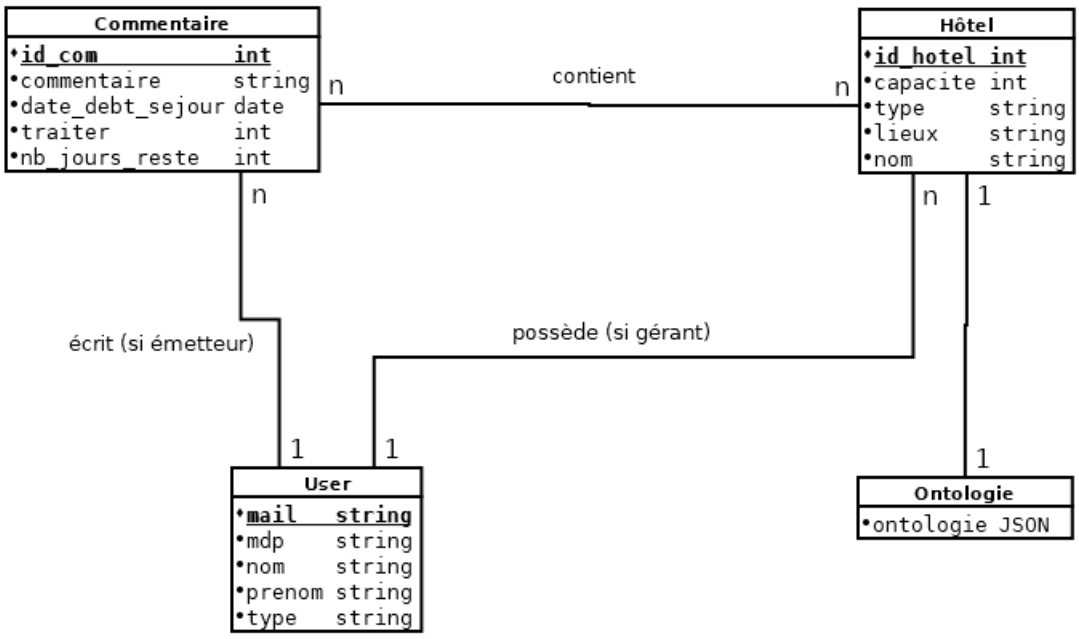


FIGURE 6 – Modélisation de la base de donnée de Hotel Advisor

Sur cette figure nous pouvons voir les différents attributs que composent nos tables. Toutes les clefs étrangères sont représentées implicitement avec les liens qui existent entre elles.

4.2.2 Web Service REST

Pour permettre l'interaction avec la base de données *MongoDB*, le serveur *Web-Service_Reviews* implémente des services web [6] REST.

REST (Representational State Transfer) est un style d'architecture permettant de construire des applications avec des conventions/bonne pratiques à respecter et qui utilise les spécifications originelles du protocole HTTP. Ces services web REST ont pour avantage d'être facile à comprendre et à implémenter, qu'un client HTTP suffit pour accéder à un service RESTful et que l'on peut utiliser des formats standards tel que le JSON. L'inconvénient majeur est la sécurité inexistante (on

6. Un web service est un programme informatique permettant la communication et l'échange de données entre applications et système hétérogènes dans des environnements distribués.

doit utiliser HTTPS et des authentifications) qui n'est pas prise en compte pour ce projet.

REST utilise des URI (Uniform Ressource Identifier) afin d'identifier des ressources. Notre application se doit de construire ces URI (donc ses URL) de manière précise en respectant si possible les quatre opérations que permet le protocole HTTP dit CRUD :

- Create : POST ;
- Read : GET ;
- Update : PUT ;
- Delete : DELETE ;

Donc, notre serveur *NodeJS* implémente des méthodes CRUD correspondant aux différentes tables que compose notre base de donnée. L'application *Angular* va ensuite appeler, grâce aux URL, un web service qui va permettre de modifier/consulter la base de données. Pour illustrer ceci, voici deux exemples de web service REST, l'un pour récupérer la liste des utilisateurs et l'autre pour changer un attribut sur la table commentaire :

```
1  /* GET sur /user */
2  app.get("/user", (req, res) => {
3    db.collection("user").find().toArray((err, documents)=> {
4      res.setHeader("Content-type", "application/json");
5        res.end(JSON.stringify(documents));
6    });
7  });
8
9  /* PUT sur /comTraiter/id=2 (2 ou autre) */
10 app.put("/comTraiter/id=:id",(req, res)=>{
11     let id = parseInt(req.params.id);
12     db.collection("commentaire").updateOne({"id_com":id},{$set :{"
     traiter":1}});
13     res.setHeader("Content-type","application/json");
14     res.end(JSON.stringify("Modification réussie !"));
15 });
```

4.3 Webservice Traitement Automatique du Langage Naturel

Dans cette partie nous allons nous intéresser au Webservice permettant la compréhension et la polarisation des avis.

4.3.1 Démarche adoptée

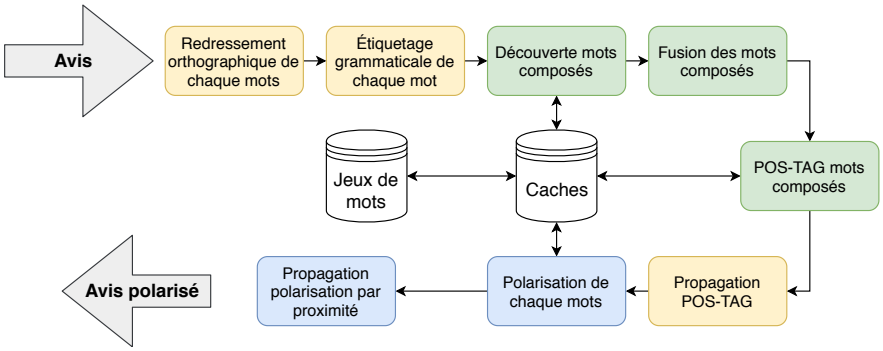


FIGURE 7 – Schéma global partie TALN

4.3.2 Redressement des mots

Lors de l'analyse préalable des avis que nous avions à polariser nous nous sommes aperçus qu'ils comptaient énormément d'erreur d'orthographe. Ce qui peut poser des soucis lors de la recherche de ces termes sur Jeux de mots. Pour palier à ce problème nous avons mis en place deux solutions.

Mise en place d'un correcteur orthographique

Nous avons trouvé un correcteur orthographique implémenté en Python (spellchecker) qui permet de corriger des erreurs d'orthographe basiques tel que l'oubli d'une lettre par exemple (recomande, proffessionnel...).

Ce module est basé sur la distance de Levenshtein [7] qui consiste à trouver toutes les permutations possible jusqu'à une distance de 2 du mot initial. Il effectue ensuite toutes ses permutations et renvoie le terme ayant la fréquence la plus élevée dans une liste référençant les termes ainsi que leurs fréquences respectives.

Cette algorithme ne permettra pas de corriger les erreurs comportant trop de caractères de différences avec le mot initial mais il pourrait nous permettre d'améliorer notre polarisation sur de nombreux cas.

7. La distance de Levenshtein est une distance, au sens mathématique du terme, donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre.

Utilisation du lemme [8]

Lors de l'étiquetage grammaticale, TreeTagger nous fournit le lemme du mot si celui-ci est connu. Nous pouvons ainsi l'utiliser afin de rechercher la polarité sur Jeux de mots si le mot de base n'est pas connu notamment pour les verbes. Prenons par exemple le mot « recommandons », sa polarité n'est pas défini dans Jeux de mots tandis que celle de son lemme (recommander) est présente.

Ainsi de nombreux termes conjugués, au pluriel ou féminin qui ne sont pas dans Jeux de mots ou mal référencés pourront être tout de même polarisés.

4.3.3 Gestion des mots composés

Présentation

Une étape importante lorsque l'on souhaite effectuer l'analyse d'une phrase rédigée en langage naturel est de savoir comment tokeniser celle-ci. Dans un premier temps nous avons tokenisé notre phrase par le symbole séparant les mots dans la langue française qui est l'espace. Mais il s'est avéré nécessaire de vérifier également si certains mots ne devaient pas restés groupés afin de préserver la sémantique.

Prenons par exemple le cas de l'ensemble de mots « salle de bain ». Si nous prenons chaque mot séparément nous perdons des informations qui peuvent être utiles aux étapes suivantes. En effet avoir les mots ["salle","de","bain"] polarisés et affichés séparément dans notre ontologie n'aidera pas le gérant de l'hôtel à analyser concrètement les avis.

Données utilisées

Pour pouvoir détecter les mots composés nous avons utilisé un dump d'une liste présente sur JeuxDeMots que nous avons enrichie avec certains termes spécifiques au domaine de l'hôtellerie.

Cette liste contenant plus de 600000 termes, il nous a donc semblé juste de trouver une structure de données adéquates permettant d'optimiser la recherche en temps. La structure choisie est un arbre des préfixes ayant pour noeud les différents mots présent dans nos mots composés.

Voici un exemple de sous-arbre de cette structure :

8. Le lemme est l'unité autonome constituante du lexique d'une langue. C'est une suite de caractères formant une unité sémantique et pouvant constituer une entrée de dictionnaire.

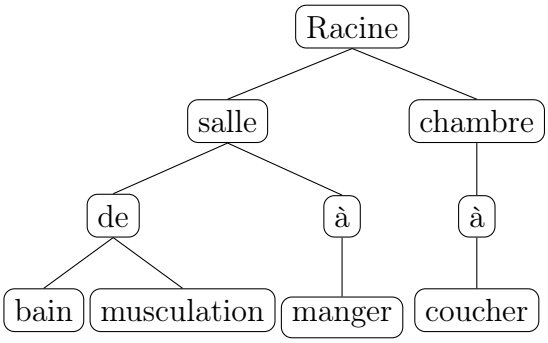


FIGURE 8 – Sous-arbre des préfixes des mots composés

Détection d'un mot composé

Afin de trouver tous les mots composés présents dans une phrase s de taille m (nombre de mots) nous avons implémenté un algorithme qui recherche pour chaque position dans la phrase le mot composé commençant à cette même position. Il nous retourne donc un tableau d'entiers de taille m contenant pour chaque position i dans la phrase s la taille du plus grand mot composé ayant pour premier mot le mot présent en s[i].

Algorithm 1 Recherche des mots composés

Input:
arbre_mc : arbre des préfixes contenant les mots composés
phrase : tableau de chaînes de caractères de taille m
Output: tableau d'entier de taille m contenant la taille maximum des mots composés pour chaque position

```
for k=0 ;k<phrase.length ;k++ do
  mc_tab = 0
end for
for i=0 ;i<phrase.length-1 ;i++ do
  if mot_courant est un fils de la racine de arbre_mc then
    size = 1
    for j=i+1 ;j<phrase.length ;j++ do
      if existe un chemin de phrase[i] à phrase[j] partant de la racine dans
      arbre_mc then
        if phrase[j] est un élément terminal dans arbre_mc then
          mc_tab[i] = size
        end if
        size++
      else
        break
      end if
    end for
  end if
end for
return mc_tab
```

FIGURE 9 – Algorithme de recherche des mots composés

4.3.4 Étiquetage morpho-syntaxiques

Après l'étude effectué dans l'état de l'art nous avons donc choisi d'utiliser la librairie TreeTagger qui nous fournit de meilleurs étiquetages grammaticaux que SpaCy. TreeTagger nous renvoyant une structure de données assez brut et difficilement utilisable en l'état nous avons choisi de définir notre propre structure qui se présente comme ci-dessous :

```
1     [   {index : 0, mot : "la", nature : "DET", lem : "le"},
2         {index : 1, mot : "chambre", nature : "NOUN", lem : "
      chambre"},
3         {index : 2, mot : "était", nature : "VERB", lem : "être"},
4         {index : 3, mot : "sale", nature : "ADJ", lem : "sale"}
5     ]
6
```

FIGURE 10 – Structure de données après pos-tag

Nous avons apporté une attention particulière à l'étiquetage morpho-syntaxique de notre phrase car c'est ce qui va être la base de notre propagation de polarité.

Vous trouverez le code permettant de réaliser cette étiquetage grammaticale en Annexe C.

4.3.5 Propagation des étiquettes

La propagation des étiquettes grammaticales correspond à ajouter la notion de dépendance grammaticale entre certains termes de notre phrase. Elle est nécessaire pour faciliter par la suite notre analyse.

Dans cette démarche, pour chaque nom nous recherchons les adjectifs associés à celui-ci et également le verbe qu'il réfère. En suite pour chaque adjectif on recherche les adverbes qui modifient ou précise leur sens. Et pour finir pour chaque verbe on recherche les adverbes associés.

Prenons comme exemple la phrase suivante :

la chambre était très sale

Après avoir recherché ses étiquettes grammaticales nous obtenons la structure suivante :

```
1     [   {index : 0, mot : "la", nature : "DET"},
2         {index : 1, mot : "chambre", nature : "NOUN"},
3         {index : 2, mot : "était", nature : "VERB"},
4         {index : 3, mot : "très", nature : "ADV"},
5         {index : 4, mot : "sale", nature : "ADJ"}
6     ]
7
```

FIGURE 11 – Exemple de structure avant la propagation des étiquettes

Voici ci-dessous la structure choisi afin de représenter les connexions définit précédemment :

```
[   {index : 0, mot : "la", nature : "DET"},
    {index : 1, mot : "chambre", nature : "NOUN",
        index_verbe : [2],
        index_adj : [4]},
    {index : 2, mot : "était", nature : "VERB"},
    {index : 3, mot : "très", nature : "ADV"},
    {index : 4, mot : "sale", nature : "ADJ", index_adv : [3]}
]

```

FIGURE 12 – Exemple de structure après la propagation des étiquettes

Vous trouverez le code permettant cette propagation des étiquettes en Annexe D.

Maintenant que nous avons une structure qui lie chaque mot les uns avec les autres nous pouvons commencer la polarisation de notre phrase.

4.3.6 Polarisation des termes

Afin de polariser les termes de notre phrase nous avons utilisé le réseau lexical JeuxDeMots qui permet de récupérer pour chaque mot présent leur poids négatif, positif et neutre.

Pour commencer nous avons choisi de définir la polarité d'un terme comme étant un entier (-1 négatif, +1 positif). Mais après réflexion il s'est avéré que nous perdions énormément d'information très importantes pour la suite.

La deuxième approche adoptée a été de gérer la polarisation comme un vecteur normalisé ayant trois attributs (positif, négatif et neutre). Ainsi lors de la propagation de ces vecteurs nous sauront qu'elle est la probabilité qu'un terme soit négatif, positif ou neutre ce qui va améliorer nos prédictions.

Voici l'exemple précédent polarisé :

```
[   {index : 0, mot : "la", nature : "DET",
        pol: { neg: 0, pos: 0, neutre: 0 }},
    {index : 1, mot : "chambre", nature : "NOUN",
        index_verbe : [2],
        index_adj : [4],
        pol: { pos: 0.16, neutre: 0.84, neg: 0 }},
    {index : 2, mot : "était", nature : "VERB",
        pol: { pos: 0.75, neutre: 0.16, neg: 0.09 }},
    {index : 3, mot : "très", nature : "ADV",
        pol: { pos: 0.57, neutre: 0.37, neg: 0.06 }},
    {index : 4, mot : "sale", nature : "ADJ",
        index_adv : [3],
        pol: { pos: 0.02, neutre: 0.02, neg: 0.96 }}
]

```

FIGURE 13 – Exemple de structure après polarisation des termes

Maintenant que nous avons la polarité de chaque terme séparément nous allons voir comment propager cette polarisation sur les termes qui nous intéressent.

4.3.7 Propagation de la polarisation

Dans un premier temps nous avons choisi de propager les Adverbes sur les Adjectifs et les Adverbes sur les Verbes. Ensuite nous propageons les Adjectifs et les Verbes vers les Noms.

Concernant la propagation des adverbes nous avons une liste d'intensificateurs qui contient les adverbes ainsi qu'un coefficient d'intensification négatif ou positif. Si il est négatif nous multiplions l'attribut négatif de l'objet pol sinon nous appliquons ce coefficient sur l'attribut positif. Après avoir appliqué ce coefficient nous normalisons les valeurs des attributs de l'objet pol entre 0 et 1.

Pour ce qui est des adjectifs nous avons décidé de faire la moyenne des polarisations de tous les adjectifs se référant à chaque nom et de remplacer la polarisation du nom par celle-ci. Si nous avons choisi cette stratégie c'est tout simplement car nous avons remarqué que la polarité d'un nom dépend essentiellement de la polarité de ses adjectifs.

Voici par exemple la phrase présenté précédemment après propagation de la polarisation :

```
1     [   {index : 0, mot : "la", nature : "DET",
2             pol: { neg: 0, pos: 0, neutre: 0 }},
3         {index : 1, mot : "chambre", nature : "NOUN",
4             index_verbe : [2],
5             index_adj : [4],
6             pol: { pos: 0.01, neutre: 0.01, neg: 0.99 }},
7         {index : 2, mot : "était", nature : "VERB",
8             pol: { pos: 0.9, neutre: 0.06, neg: 0.04 }},
9         {index : 3, mot : "très", nature : "ADV",
10            pol: { pos: 0.57, neutre: 0.37, neg: 0.06 }},
11        {index : 4, mot : "sale", nature : "ADJ",
12            index_adv : [3],
13            pol: { pos: 0.01, neutre: 0.01, neg: 0.99 }}
14    ]
15
```

FIGURE 14 – Exemple de structure après propagation de la polarisation des termes

En analysant la structure, présentée ci-dessus, en se mettant dans le rôle du gérant de l'hôtel on peut observer qu'il y a un souci avec une chambre car ce terme a un poids négatif supérieur au neutre et au positif.

Après avoir obtenu cette structure elle est retournée au format JSON à l'émetteur de la requête.

4.4 Interface Web

Comme dit précédemment dans l'introduction de ce rapport, le projet comporte trois statuts utilisateurs. Cette section va montrer le workflow [9] de chaque utilisateur, les outils utilisés pour réaliser les interfaces et comment on représente l'ontologie.

4.4.1 Workflow

Ces schémas montrent les interactions que doivent réaliser chaque profil d'utilisateur pour aller d'une page à une autre.

Émetteur d'avis

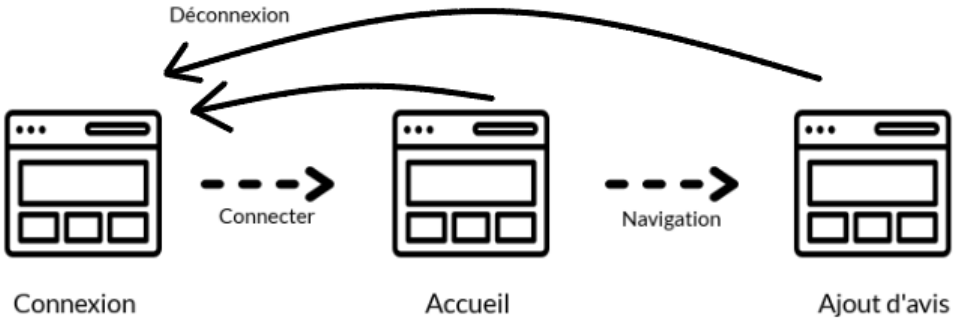


FIGURE 15 – Workflow de l'émetteur d'avis

Administrateur

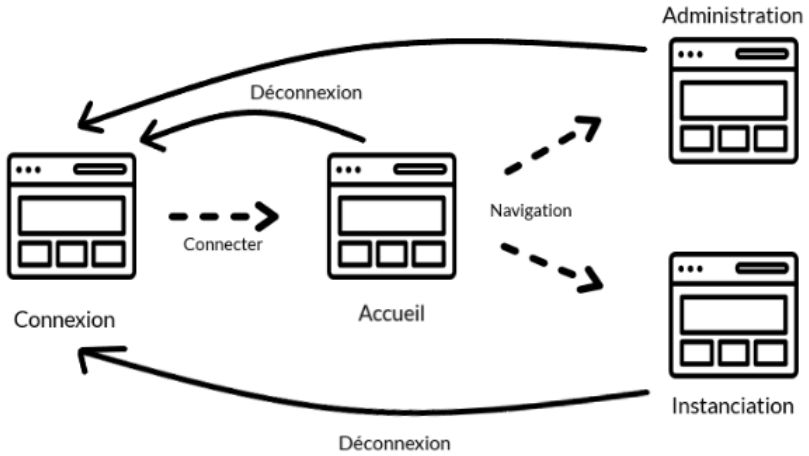


FIGURE 16 – Workflow de l'admin

9. Workflow est un anglicisme qui veut dire flux de travail. Il représente la suite de tâche à effectuer par une personne, un organisme, un groupe de personne.

Gérant

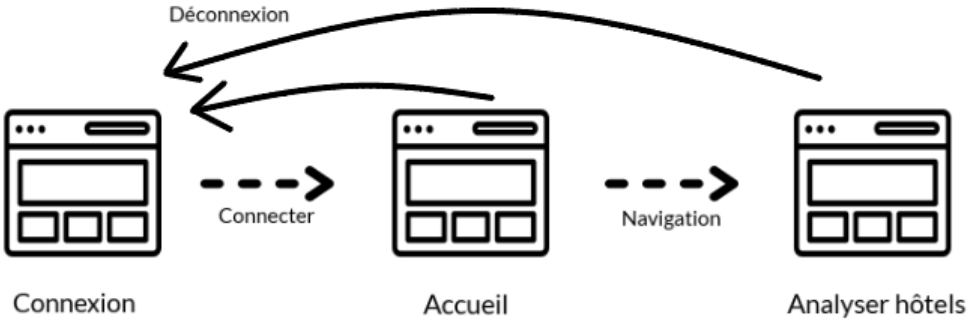


FIGURE 17 – Workflow du gérant

4.4.2 Outils d'interface

Toutes les interfaces réalisées durant ce projet ont été faites en majeure partie grâce à *Bootstrap*. *Bootstrap* est une collection d'outils utilisés pour la création de design de sites et d'application web. Nous avons choisis cette outil car il est très facile d'utilisation et contient une documentation claire et complète.

Ces interfaces vous sont présentées de l'Annexe E à l'Annexe N.

Un outil spécial, dont vous pouvez voir le résultat à l'Annexe M, a été utilisé pour gérer l'attente que génère l'affichage de l'ontologie quand un gérant doit en consulter une. En effet, avant d'afficher l'ontologie (Annexe N), l'application web envoie chaque commentaire lié à l'hôtel vers le service web TALN. Cela peut prendre plusieurs secondes, d'où l'utilisation d'un loader (chargeur) pour prévenir que l'application est en train de gérer sa demande. Nous avons utilisé pour cela *ngx-loading* qui est un paquet pour *Angular* permettant de créer et d'adapter des loaders.

4.4.3 Représentation de l'ontologie

Dans cette partie nous allons non pas détailler la façon dont a été réalisé l'arbre sur l'interface web avec toutes ses fonctionnalités interactives, qui consiste juste à de la manipulation *DOM*[10] à l'intérieur des balises *HTML SVG* grâce à *D3.js*, mais simplement expliquer la construction de l'arbre et le coloriage de chacun de ses noeuds.

10. DOM ou Document Object Model est une interface de programmation normalisée par le W3C, qui permet à des scripts d'examiner et de modifier le contenu du navigateur web

Construction arborescente

Après avoir récupéré l'ontologie de l'hôtel sélectionné avec le service *getOntologie(id :any)* présente dans les services d'*ontologie.service*, qui envoie une requête GET avec pour id celui de l'hôtel en question au serveur *Node.js* s'occupant de communiquer avec la base de données *MongoDB*, nous passons l'ontologie à *D3.js* depuis la fonction *ontologie()* du composant ontologie, qui permet de construire une arborescence *DOM* contenant chaque terme, leur fils, leur parent direct et ses synonymes, comme présent dans notre ontologie *JSON*. *D3.js* nous permet donc de construire chaque noeud au format *HTML*, encapsulé dans des balises *g* elles-mêmes contenues dans la balise *SVG*, permettant de dessiner graphiquement lesdits noeuds. L'avantage de cette méthode réside dans la surcouche *DOM* que pose *D3.js* sur notre objet JSON. Ainsi, si nous modifions la propriété *data* d'un objet *DOM* dans notre application, l'objet *JSON* le sera également.

Coloriage de l'arborescence

Avant que l'arborescence ne soit affichée, l'application *Angular* va envoyer une nouvelle requête au serveur *Node.js* pour récupérer la liste des commentaires encore non traités par l'ontologie, mais validés par l'administrateur. Dans le cas où cette liste n'est pas vide, le composant *ontologie* va appeler la fonction *associatePola(root :any)* présentée ci-dessous :

```
1  associatePola(root:any){
2    var obj = this.searchNode(root);
3    if(obj != undefined){
4      if(root.numberOfCom == null){
5        root.numberOfCom = 1;
6        root.polarite = obj.pol;
7      }
8      else{
9        //calcul de la nouvelle moyenne de la polarité
10       root.polarite.neg = ((root.polarite.neg * root.numberOfCom)
     + obj.pol.neg)/(root.numberOfCom +1);
11       root.polarite.neutre = ((root.polarite.neutre * root.
     numberOfCom) + obj.pol.neutre)/(root.numberOfCom +1);
12       root.polarite.pos = ((root.polarite.pos * root.numberOfCom)
     + obj.pol.pos)/(root.numberOfCom +1);
13
14       root.numberOfCom++;
15     }
16   }
17   if(root.children){
18     for(let child of root.children){
19       this.associatePola(child);
20     }
21   }
22 }
```

FIGURE 18 – Association de la polarité d'un terme issue d'un commentaire avec celui présent dans l'ontologie

Cette fonction récursive permet d'explorer toute l'arborescence *DOM* de notre ontologie, affectant à chacun de ses termes la polarité issue d'un commentaire encore non traité. Pour cela, elle récupère tout d'abord l'objet *DOM* du terme de l'ontologie traité actuellement, ou de l'un de ses synonymes. Si le terme n'est pas présent dans l'un des commentaires, alors on relance directement la fonction sur l'un de ses fils, si il y en a. Si le terme est présent, alors nous calculons la polarité moyenne de ce terme, à partir de la polarité du commentaire, mais également celle sauvegardée dans la base de données avec l'ontologie de l'hôtel. Dans le cas où le terme n'a jamais été polarisé, nous affectons directement la polarité du commentaire au terme.

Que la liste des commentaires non traités soit vide ou non, le composant appelle ensuite la fonction *propagationPola(root :any)* présentée ci-dessous :

```
propagationPola(root:any){
  if(root.children != null){
    let arrayOfPola = [];
    let newPola = {pos:0,neg:0,neutre:0};
    for(let child of root.children){
      this.propagationPola(child);
      if(child.data.polarite != null){
        arrayOfPola.push(child.data.polarite);
      }
    }
    if(arrayOfPola.length != 0){
      let index=1;
      for(let tempPola of arrayOfPola){

        newPola.pos = ((newPola.pos * index) + tempPola.pos)/(index+1);
        newPola.neg = ((newPola.neg * index) + tempPola.neg)/(index+1);
        newPola.neutre = ((newPola.neutre * index) + tempPola.neutre)/(index+1);
        index++;
      }
      if(root.data.polarite == null){
          root.data.polarite = newPola;
          root.data.numberOfCom = 1;
      }
      else{
        let nbCom = (root.data.numberOfCom ? root.data.numberOfCom : 1);

        root.data.polarite.pos = ((root.data.polarite.pos * nbCom) + newPola.pos)/(nbCom +1);
        root.data.polarite.neutre = ((root.data.polarite.neutre * nbCom) + newPola.neutre)/(nbCom +1);
        root.data.polarite.neg = ((root.data.polarite.neg * nbCom) + newPola.neg)/(nbCom +1);

      }
    }
  }
}
```

FIGURE 19 – Propagation de la polarité des termes de l'arborescence via un parcours en profondeur

Cette fonction également récursive permet de propager la polarité d'un terme enfant, vers ses parents, grâce à un parcours en profondeur. Cette fonction commence par construire un tableau de polarité qui contiendra celle de ses enfants, remplit après l'appel récursif de la fonction sur ses enfants en question. Ce parcours en profondeur permet donc de propager la polarité depuis les feuilles de l'arborescence, jusqu'au noeud racine. Une fois qu'un terme a récolté la polarité propagée

de tous ses termes enfants, nous calculons à nouveau une moyenne de polarité, entre celles de ses enfants, et la dernière moyenne lorsque l'ontologie a été renvoyé par la base de données. Encore une fois, dans le cas où le terme n'a jamais été polarisé, nous affectons directement la polarité de la moyenne de ses enfants audit terme.

Une fois que l'ontologie a été correctement traitée et la polarité de ses termes propagées, nous sauvegardons cette dernière dans la base de données, grâce à la fonction *updateOntologie(onto :any)* de *ontologie.service* qui va transmettre l'ontologie modifiée au format JSON via une requête PUT vers le serveur *Node.js*, puis nous affichons l'ontologie (après chargement du *ngx-loader*) à l'utilisateur, c'est à dire le gérant.

5 Limites et évolutions

5.1 Limites

Tout au long des phases de développement et de débogage, nous avons pu appréhender plus précisément les limites de notre stratégie. Certaines de ces limites sont inhérentes au projet, d'autres sont extérieures au projet, c'est le cas par exemple du manque d'information sur la polarisation de certains termes.

Notre travail se basant sur une démarche empirique par rapport aux avis rédigés par des êtres humains, les principales limites du projet sont inhérentes à cette contrainte importante. Lors de la réalisation des différentes méthode de propagation nous nous somme basés sur de vrais avis ce qui nous a permis de définir des schémas globaux de propagation. Mais malgré cela il reste des cas qui ne sont pas pris en compte et qui ne pourront être pris en compte en utilisant cette méthode.

Une autre limite importante de notre projet est la gestion du redressement des fautes d'orthographes présentes dans les avis. Nous avons essayé d'en gérer le maximum mais il reste néanmoins celles concernant les oublis d'accents qui peuvent corrompre notre propagation de polarité.

Enfin une dernière limite de notre projet pourrait-être la façon dont nous récupérons les polarisations de chaque terme ainsi que les étiquettes grammaticales. En effet après avoir effectué certains tests sur des avis réels, il s'est avéré que de nombreux termes n'avait pas de polarité définit dans JeuxDeMots car ils étaient mal orthographiés ou tout simplement non renseignés. Concernant les étiquettes grammaticales, TreeTagger n'ayant pas une précision de 100%, il arrive que nous ayons des termes mal étiquetés grammaticalement ce qui peut générer de grosses erreurs dans nos résultats obtenus.

5.2 Évolutions

5.2.1 Une interface plus moderne

Nous pouvons constater que les interfaces réalisées sont simples et austères. En effet, nous n'avons pas eu de cours de réalisation d'interface graphique, ni de design web. Nous avons réalisé ces interfaces pour que leurs implémentations soient rapides et le design compréhensible afin d'avoir très vite une première version de l'application.

Afin d'améliorer cet aspect, nous devrions nous renseigner sur les différentes modes dans le monde du web design et nous former afin de créer des interfaces plus moderne.

Enfin, nous devrions créer un archivage semaine après semaine des ontologies d'un gérant pour qu'il constate leurs évolutions au cours du temps.

5.2.2 Amélioration de la propagation de polarité

Amélioration de la propagation par proximité

Afin d'améliorer notre propagation de polarité actuelle il faudrait améliorer les pré-traitements effectués sur les avis ce qui aura un impact certain sur l'étiquetage grammaticales ainsi que sur la polarisation de notre ontologie.

Utilisation d'un arbre syntaxique

Une évolution possible de notre application serait de gérer la propagation de la polarité grâce à un arbre syntaxique. Pour ce faire il faudrait tout d'abord construire l'arbre syntaxique de notre phrase en utilisant par exemple une grammaire de Chomsky. Ensuite on polariserait notre phrase en remontant l'arbre syntaxique.

Il serait utile d'effectuer une étude pour savoir si cela améliore réellement les résultats ou non.

6 Conclusion

Durant quatre mois, nous avons réalisé ce projet qui fut positif à bien des égards.

Lors de ce projet nous avons amélioré nos compétences en architecture logiciel (MEAN et MVC) et également appris l'utilisation de nouveaux outils tels que JeuxDeMots, Selenium et D3.js.

Nous nous sommes également perfectionnés au Traitement Automatique du Language Naturel et à ses différentes composantes telles que l'extraction d'informations, la lemmatisation, l'étiquetage morpho-syntaxique, l'utilisation d'un réseau lexical et la polarisation de terme.

Cette expérience nous a permis de comprendre l'importance d'une bonne communication entre les membres du groupe et de bien jalonner notre projet afin d'éviter les déconvenues et les éventuels retards pouvant apparaître lors de la réalisation du projet.

Notre architecture est fonctionnelle et nous considérons que les objectifs sont atteints. Mais du fait des caractéristiques inhérente de notre projet (démarche empirique notamment), elle doit être en constante évolution afin de fournir des résultats optimaux.

Table des figures

Annexes

A Algorithme Scraping Tripadvisor

```
1  import pandas as pa
2  from bs4 import BeautifulSoup
3  import requests
4
5  from selenium import webdriver
6  from selenium.webdriver.firefox.options import Options
7
8  options = Options()
9  #Evite que le navigateur firefox s'ouvre
10 options.add_argument('--headless')
11
12 driver = webdriver.Firefox(options=options)
13
14 #Bot qui permet de cliquer sur tous les boutons "plus" pour
       afficher le commentaire en intégralité.
15 #On retourne l'HTML généré.
16 def plusclick(url):
17   driver.get(url)
18   try:
19     driver.find_element_by_class_name('hotels-hotel-review-
       community-content-review-list-parts-ExpandableReview__cta--3
       _zOW').click()
20   except Exception as e:
21     print("Pas de plus")
22   return driver.page_source
23
24 def sitedown(url):
25   r = s.get(url)
26   if r.status_code != 200:
27     print('status_code : ', r.status_code)
28   else:
29     page = plusclick(url)
30     return BeautifulSoup(page, 'html.parser') #Biblio
       BeautifulSoup qui permet de parser un HTML et de sortir un
       objet pour faciliter le parsing de celui-ci.
31
32 def parse(url, reponse):
33   if not reponse:
34     print('Pas de reponse du site:', url)
35     driver.close()
36     return
37
38   #On récupère le nombre de commentaire mis sur la page de l'hôtel
       (commentaire étrangé inclus)
39   num_reviews = reponse.find("span", {"class":"reviewCount"}).text
40   num_reviews = num_reviews.replace("avis","")
41   num_reviews = num_reviews.replace("\xa0","") #pour les espaces
       si le nombres d'avis est au-dessus de 1000 (sur tripadvisor la
```

```
    forme est 1\xa0000\xa0avis = 1 000 avis)
42  num_reviews = int(num_reviews)
43
44  url = url.replace('.html', '-or{}.html')
45  for offset in range(0, num_reviews, 5):
46    url_ = url.format(offset)
47    doBreak = parse_reviews(url_, sitedown(url_))
48    if doBreak==1:
49      break
50
51 def parse_reviews(url, reponse):
52  print("URL :", url)
53  if not reponse:
54    print('Pas de reponse du site:', url)
55    driver.close()
56    return
57
58  liste_commentaire = enumerate(reponse.find_all("div", {"class":"
     hotels-hotel-review-community-content-review-list-parts-
     SingleReview__mainCol--29php"}))
59  #Test nécessaire car on récupère que les commentaires Français
     donc pas besoin de faire les autres pages
60  if len(list(liste_commentaire))!= 0 : #si il n'y a plus de
     commentaire, on passe à l'hotel suivant
61    #Obligé de refaire un enumerate car liste_commentaire est
     transformé en liste pour calculer le nombre de commentaire
62    for idx, review in enumerate(reponse.find_all("div", {"class":
     "hotels-hotel-review-community-content-review-list-parts-
     SingleReview__mainCol--29php"})):
63      rank = review.select_one('.ui_bubble_rating')['class'
     ][1][7:]
64      rank = float(review.select_one('.ui_bubble_rating')['class'
     ][1][7:])/10
65      item = {
66        'review_body': review.select_one('q').span.text,
67        'rating': rank,
68      }
69      results.append(item)
70    return 0
71  return 1 #on stop le traitement sur cette hôtel car on a récupér
     é tous les commentaires FR
```

B Une collection de documents avec *MongoDB*

```
[
  {
    "id_hotel":1,
    "mail":"gerant@gerant.com",
    "capacite":5000,
    "type":"Luxe",
    "lieux":"Montpellier",
    "nom":"Le Desperados"
  },
  {
    "id_hotel":2,
    "mail":"gerant@gerant.com",
    "capacite":300,
    "type":"Camping",
    "lieux":"Montpellier",
    "nom":"Le Camping Montpellierain"
  }
]
```

C Algorithme récupération des étiquettes grammaticales

```
import sys
import json
import treetaggerwrapper
import warnings

from spellchecker import SpellChecker

spell = SpellChecker(language="fr")

tagger = treetaggerwrapper.TreeTagger(TAGLANG='fr',TAGOPT="-token -lemma -sgml -quiet")

tags = {
    "VER:cond":"VERB",
    "VER:futu":"VERB",
    "VER:impe":"VERB",
    "VER:impf":"VERB",
    "VER:infi":"VERB",
    "VER:pper":"VERB",
    "VER:ppre":"VERB",
    "VER:pres":"VERB",
    "VER:simp":"VERB",
    "VER:subi":"VERB",
    "VER:subp":"VERB",
    "VERB" : "VERB",
    "SYM" :"SYM",
    "PUN:cit" : "PUNCT",
    "PUN" : "PUNCT",
    "PUNCT" : "PUNCT",
    "NAM" : "PROPN",
    "PROPN" : "PROPN",
    "NOM" : "NOUN",
    "NOUN" : "NOUN",
    "NUM" : "NUM",
    "PRP" : "ADP",
    "PRP:det" : "DET",
    "ADP" : "ADP",
    "PRO" : "PRON",
    "PRO:DEM" : "PRON",
    "PRO:IND" : "PRON",
    "PRO:PER" : "PRON",
    "PRO:POS" : "PRON",
    "PRO:REL" : "PRON",
    "PRON" : "PRON",
    "CONJ" : "KON",
    "CCONJ" : "KON",
    "SCONJ" : "KON",
    "KON" : "KON",
    "ADV" : "ADV",
```

```
49      "ADJ" : "ADJ",
50      "DET:ART" : "DET",
51      "DET:POS" : "DET",
52      "DET" : "DET",
53      "INT" : "INTJ",
54      "INTJ" : "INTJ",
55      "PART" : "PART",
56      "X" : "X",
57      "SENT" : "PUNCT",
58      "ABR" : "X"
59  }
60  def tagToObj(sentArray):
61      sentTab = []
62      for t in sentArray:
63          if len(t) >1 :
64              tag = t[1];
65              if t[2]=="ce" or t[2]=="chaque":
66                  tag = "DET"
67              sentTab.append({"mot" : t[0], "nature" : tags[tag],"
        tag":t[1],"lemme":t[2]})
68      return sentTab
69
70  def posTaggingTreeTagger(text) :
71      tags = tagger.tag_text(text)
72      taggedText = treetaggerwrapper.make_tags(tags,allow_extra =
        True)
73      return tagToObj(taggedText)
74
75  def spellCorrectSentence(sent) :
76      words = sent.split(" ");
77      words_corrected = []
78      for word in words :
79          if len(word.split("'")) < 2:
80              words_corrected.append(spell.correction(word))
81          else :
82              words_corrected.append(word)
83      return " ".join(words_corrected)
84
85  print(json.dumps(posTaggingTreeTagger(spellCorrectSentence(sys.
        argv[1]))))
```

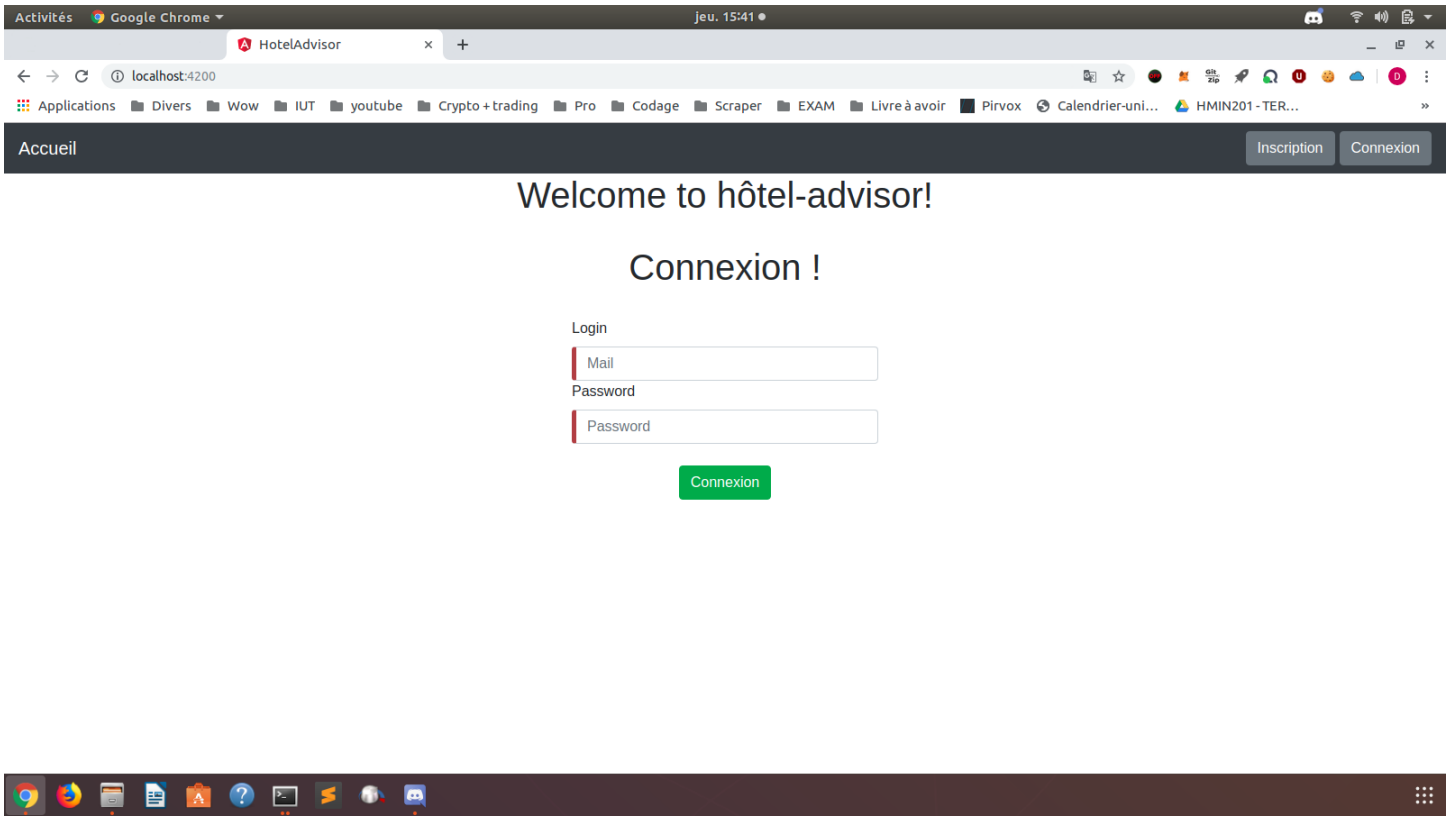
D Algorithme propagation des étiquettes grammaticales

```
# Liste des auxiliaires mal orthographiés ne pouvant être corrigés par spellchecker
listAux = ["etes", "ete", "etais", "etait", "etions","etiez", "etaient"]

# Trouve les adjectifs associés aux noms
def findAdjectifForNouns(phrase,i_nom):
    index_adj = []
    i_d = i_nom+1
    i_g = i_nom-1
    deja_trouve = False

    while i_d <len(phrase):
        if phrase[i_d]["nature"]=="ADJ":
            index_adj.append(i_d)
            deja_trouve = True
            i_d+=1
        elif phrase[i_d]["nature"]=="ADV":
            i_d+=1
        elif (phrase[i_d]["lemme"] != "<unknown>" and ((phrase[i_d][["lemme"]=="être") or (phrase[i_d]["lemme"]=="avoir"))) or phrase[i_d]["mot"] in listAux:
            i_d+=1
        elif phrase[i_d]["mot"]=="que":
            i_d+=1
        elif phrase[i_d]["nature"]=="PRON":
            i_d+=1
        elif phrase[i_d]["nature"]=="CCONJ":
            i_d+=1
        else:
            break

    deja_trouve = False
    while i_g >=0:
        if phrase[i_g]["nature"]=="ADJ":
            index_adj.append(i_g)
            deja_trouve = True
            i_g-=1
        elif phrase[i_g]["nature"]=="ADV":
            i_g-=1
        elif (phrase[i_g]["lemme"] != "<unknown>" and ((phrase[i_g][["lemme"]=="être") or (phrase[i_g]["lemme"]=="avoir"))) or phrase[i_g]["mot"] in listAux:
            i_g-=1
        elif phrase[i_g]["nature"]=="CCONJ":
            i_g-=1
        else:
            break
    return index_adj
```

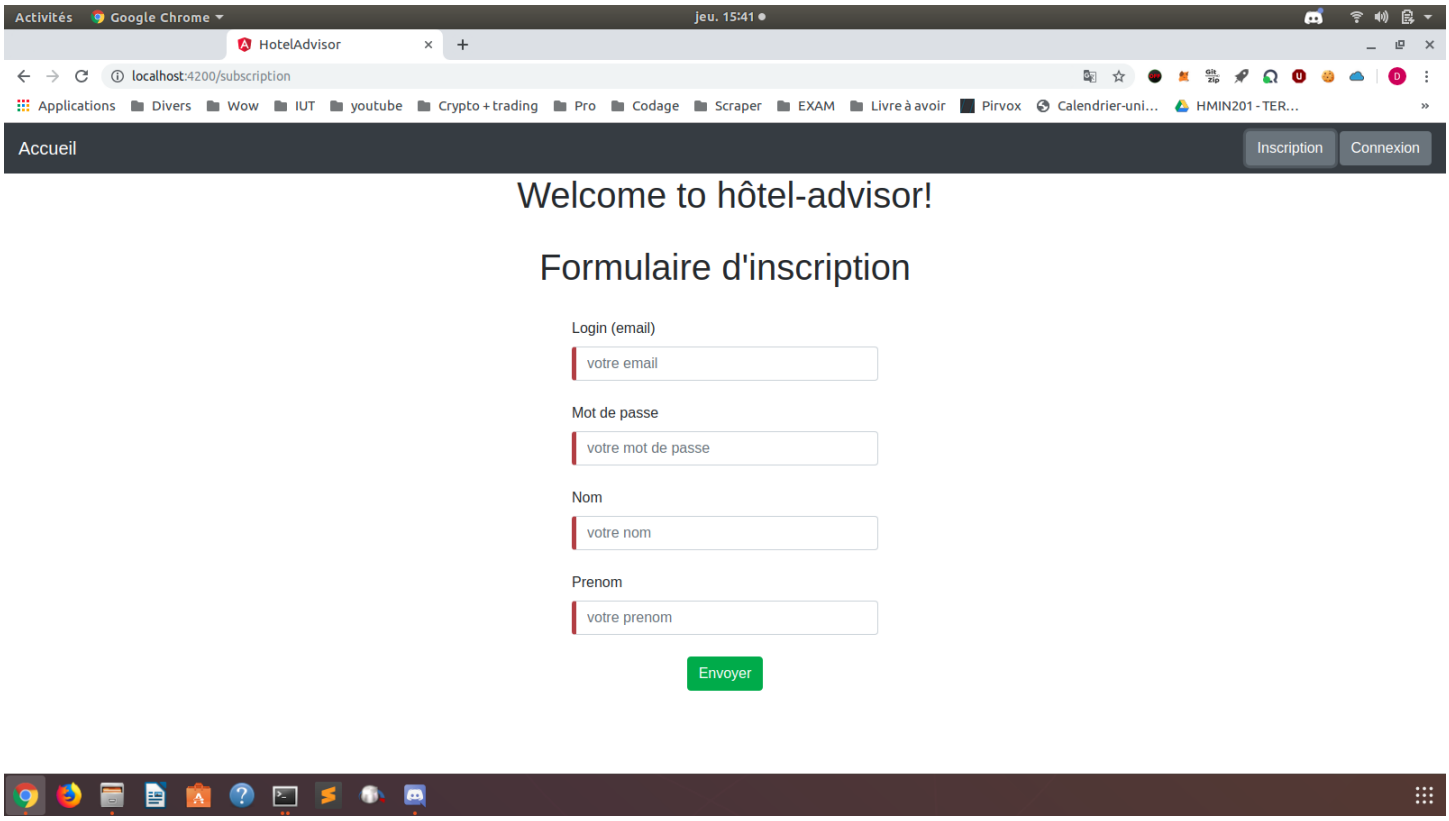
```
44
45 # Trouve les adverbes associés aux adjectifs
46 def findAdverbsForAdjectifs(phrase,i_adj):
47     index_adv = []
48     i_d = i_adj+1
49     i_g = i_adj-1
50
51     while i_d <len(phrase):
52         if phrase[i_d]["nature"]=="ADV":
53             index_adv.append(i_d)
54             i_d+=1
55         else:
56             break
57
58     while i_g >=0:
59         if phrase[i_g]["nature"]=="ADV":
60             index_adv.append(i_g)
61             i_g-=1
62         else:
63             break
64     return index_adv
65
66 # Trouve les verbes associés aux noms
67 def findVerbeForNouns(phrase,i_nom):
68     index_verbe = []
69     i_d = i_nom+1
70     i_g = i_nom-1
71     deja_trouve = False
72
73     while i_d <len(phrase):
74         if phrase[i_d]["nature"]=="VERB" and not ((phrase[i_d]["
    lemme"] != "<unknown>" and ((phrase[i_d]["lemme"]=="être") or (
    phrase[i_d]["lemme"]=="avoir"))) or phrase[i_d]["mot"] in
    listAux):
75             index_verbe.append(i_d)
76             deja_trouve = True
77             i_d+=1
78         elif phrase[i_d]["nature"]=="DET":
79             i_d+=1
80         elif phrase[i_d]["nature"]=="ADV":
81             i_d+=1
82         elif phrase[i_d]["mot"]=="que":
83             i_d+=1
84         elif phrase[i_d]["nature"]=="PRON":
85             i_d+=1
86         elif phrase[i_d]["nature"]=="CCONJ":
87             i_d+=1
88         else:
89             break
90
91     deja_trouve = False
92     while i_g >=0:
93         if phrase[i_g]["nature"]=="VERB":
```

```
94              index_verbe.append(i_g)
95              deja_trouve = True
96              i_g-=1
97          elif phrase[i_g]["nature"]=="DET":
98              i_g-=1
99          elif phrase[i_g]["nature"]=="ADV":
100             i_g-=1
101         elif phrase[i_g]["nature"]=="CCONJ":
102             i_g-=1
103         else:
104             break
105     return index_verbe
106
107 # prise en compte de la négation ou autre
108 def findAdverbsForVerbes(phrase,i_adj):
109     index_adv = []
110     i_d = i_adj+1
111     i_g = i_adj-1
112
113     while i_d <len(phrase):
114         if phrase[i_d]["nature"]=="ADV":
115             index_adv.append(i_d)
116             i_d+=1
117         else:
118             break
119
120     while i_g >=0:
121         if phrase[i_g]["nature"]=="ADV":
122             index_adv.append(i_g)
123             i_g-=1
124         else:
125             break
126     return index_adv
127
128 list = json.loads(sys.argv[1])
129
130 for index,item in enumerate(list):
131     if item["nature"] == "NOUN":
132         item["index_adj"] = []
133         item["index_adj"] = findAdjectifForNouns(list,index)
134         item["index_verb"] = []
135         item["index_verb"] = findVerbeForNouns(list,index)
136         list[index] = item
137     elif item["nature"] == "ADJ":
138         item["index_adv"] = []
139         item["index_adv"] = findAdverbsForAdjectifs(list,index)
140         list[index] = item
141     elif item["nature"] == "VERB":
142         item["index_adv"] = []
143         item["index_adv"] = findAdverbsForVerbes(list,index)
144         list[index] = item
145
146 print(json.dumps(list))
```

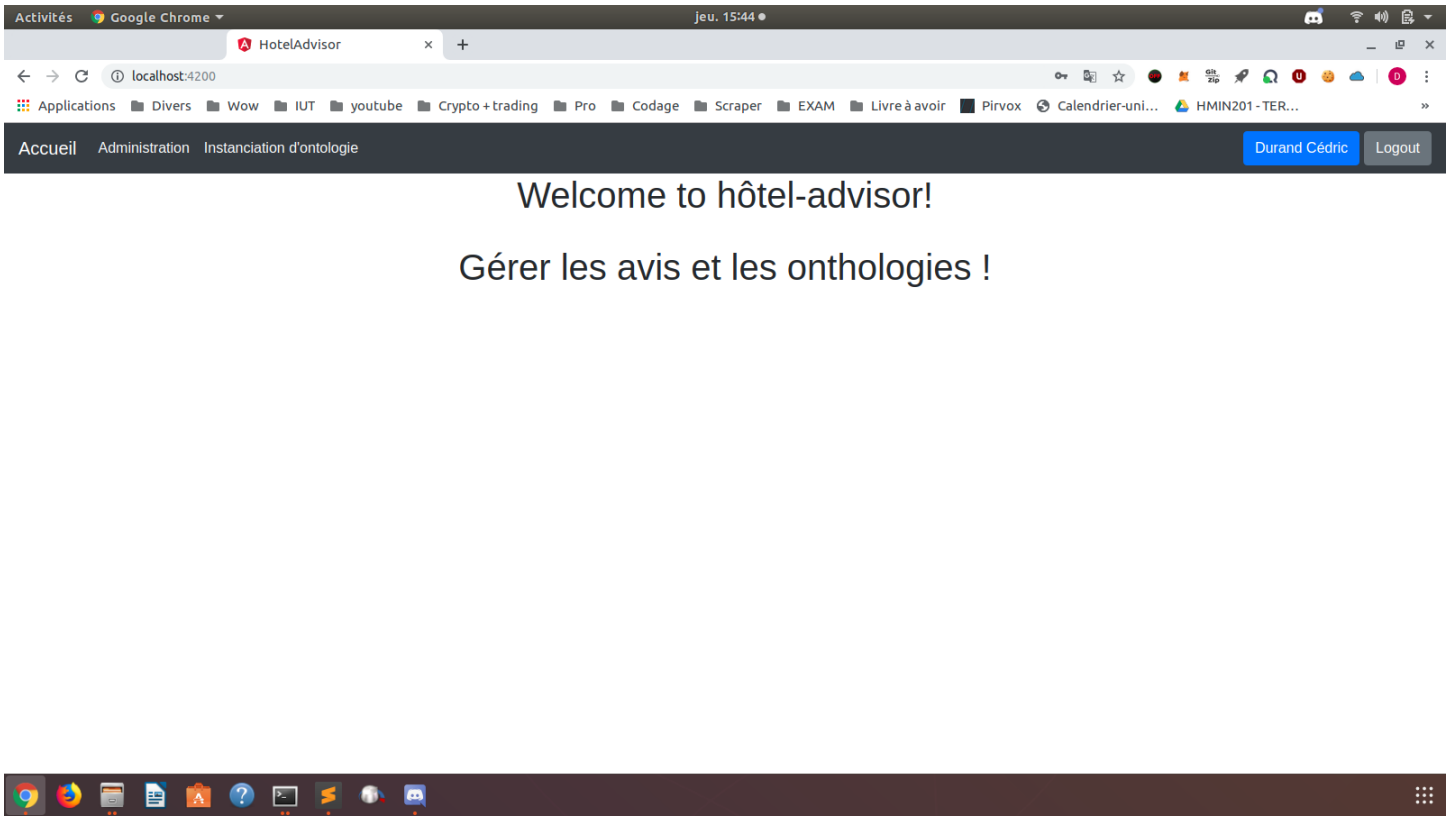
E Interface connexion



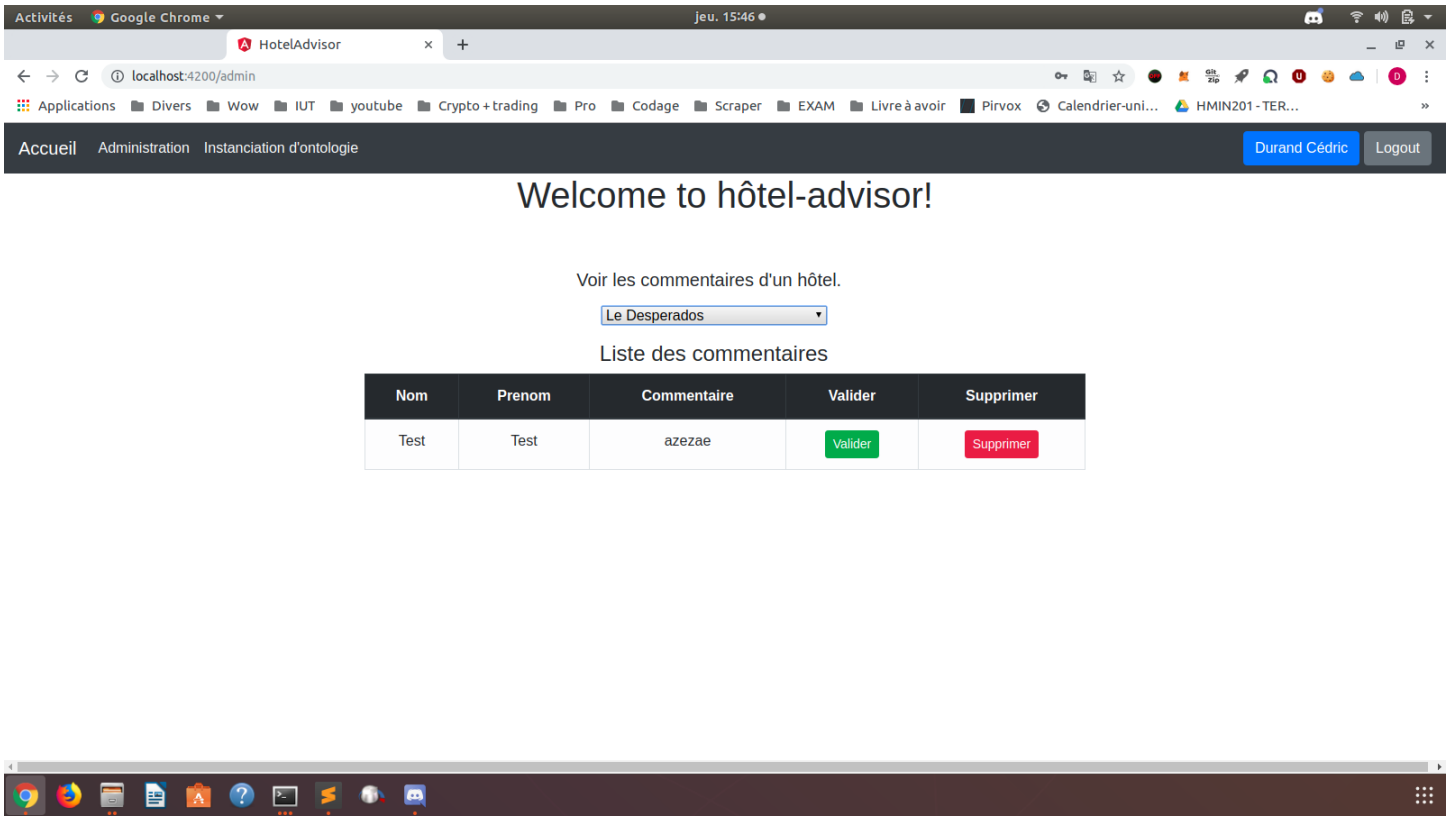
F Interface inscription



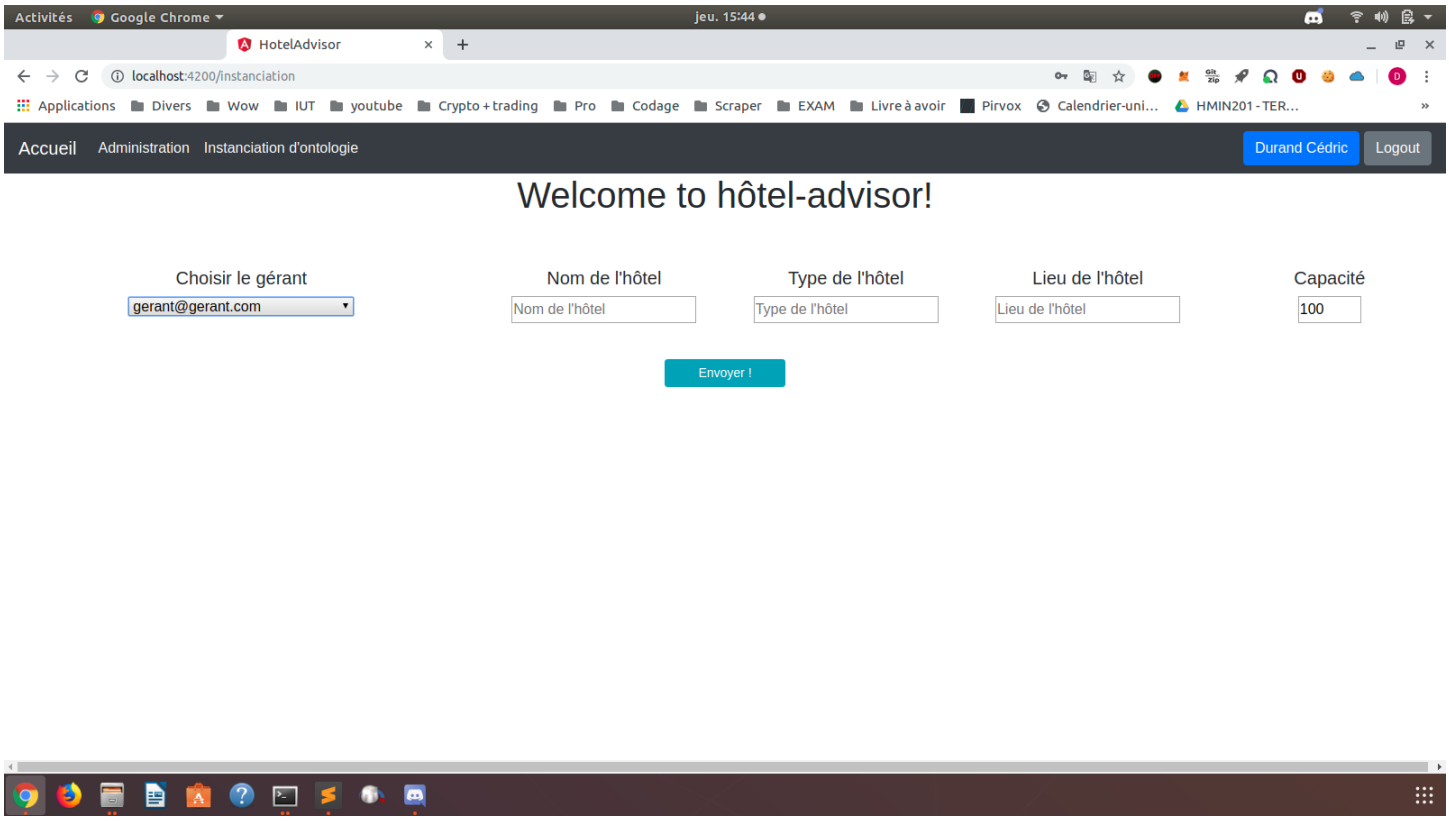
G Interface accueil admin



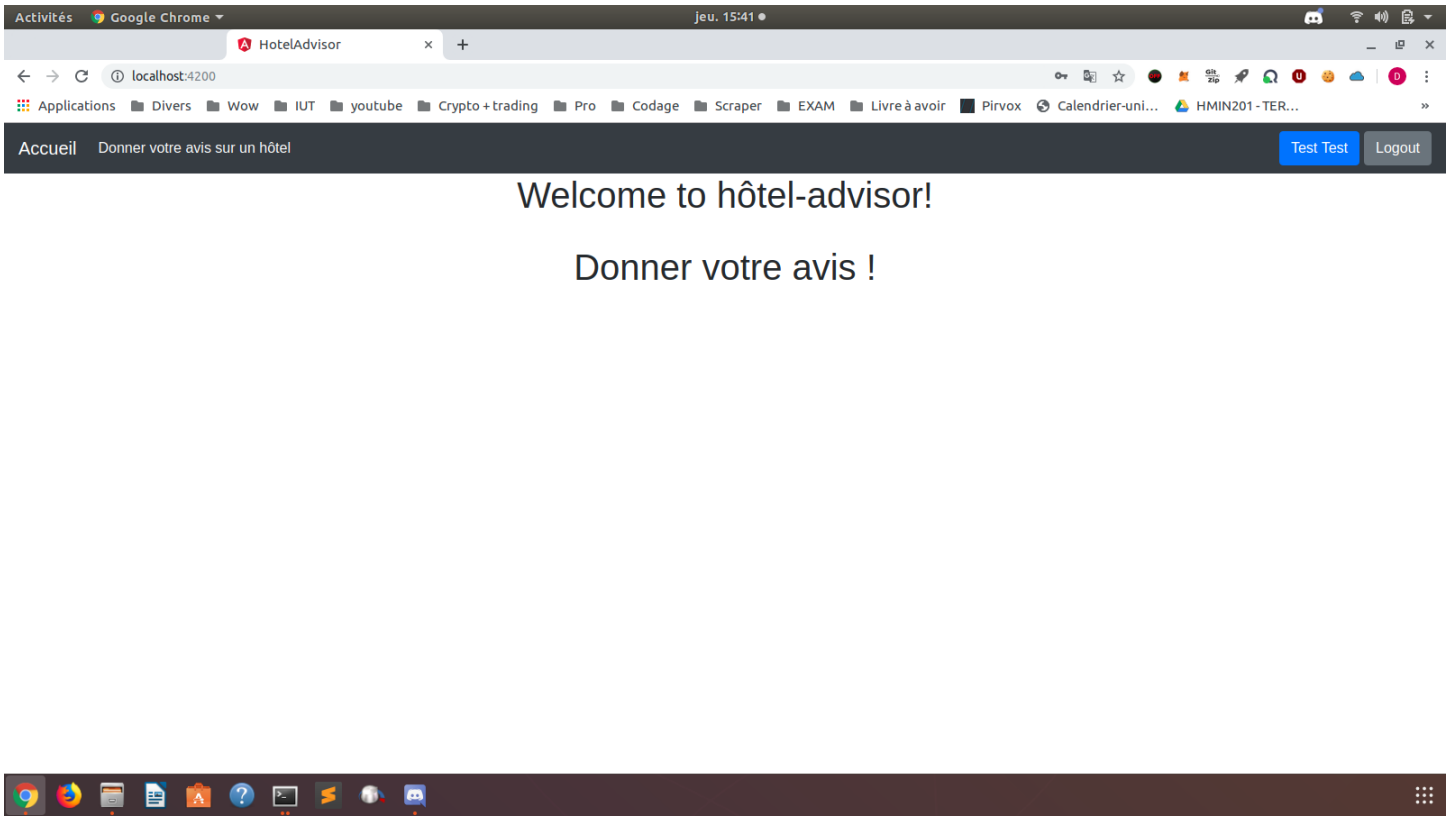
H Interface administration de commentaire



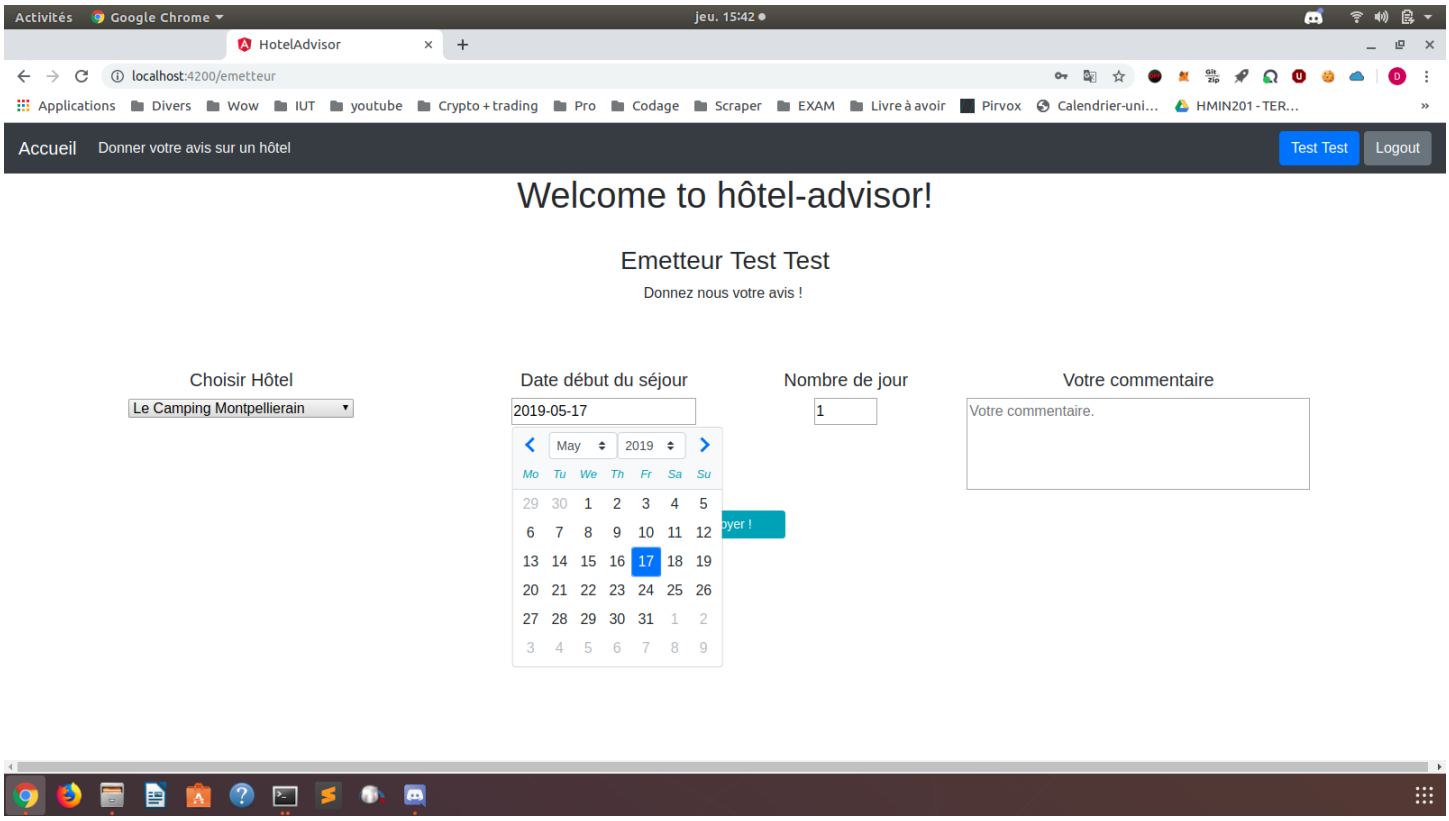
I Interface instanciation



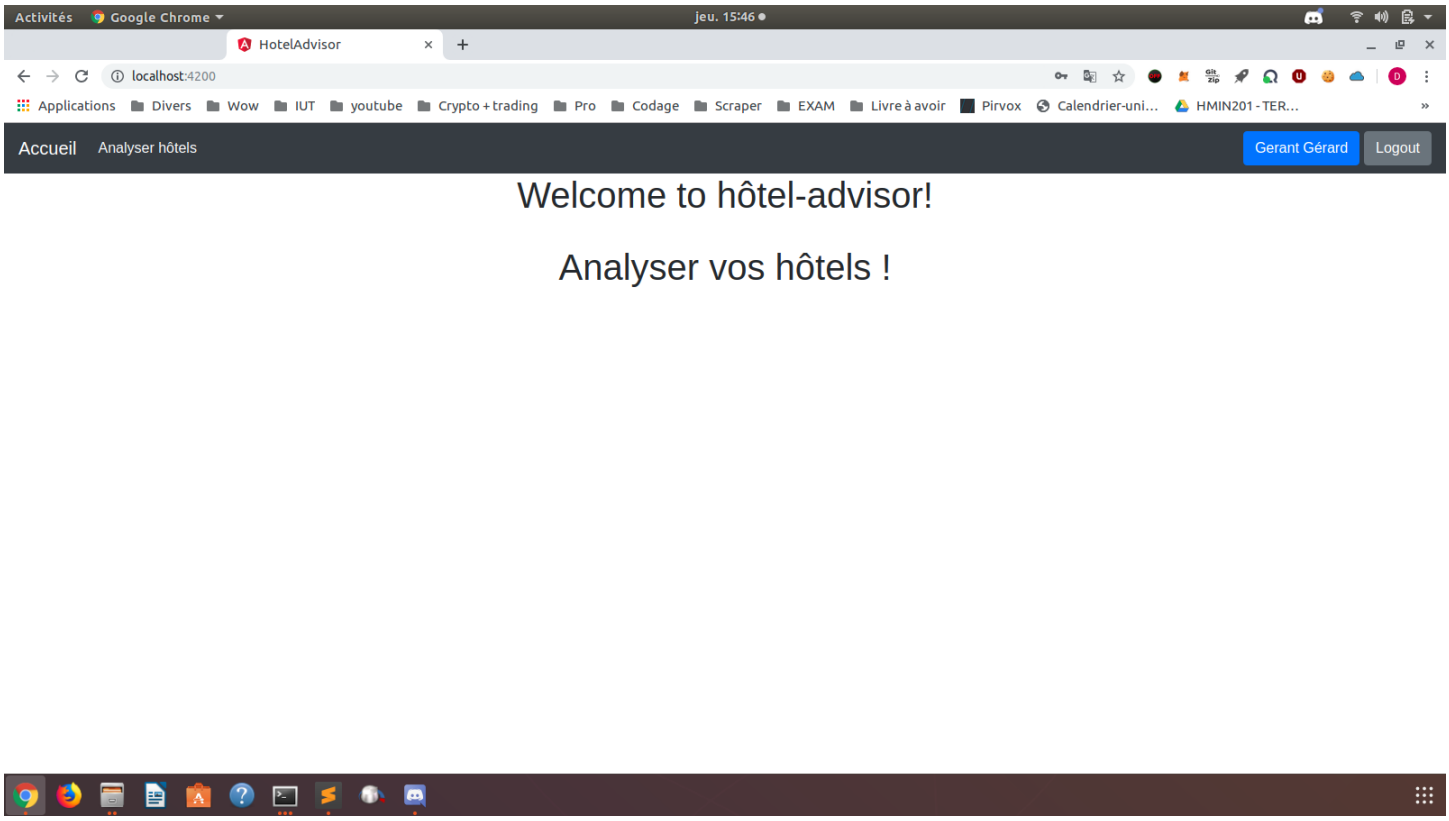
J Interface accueil émetteur



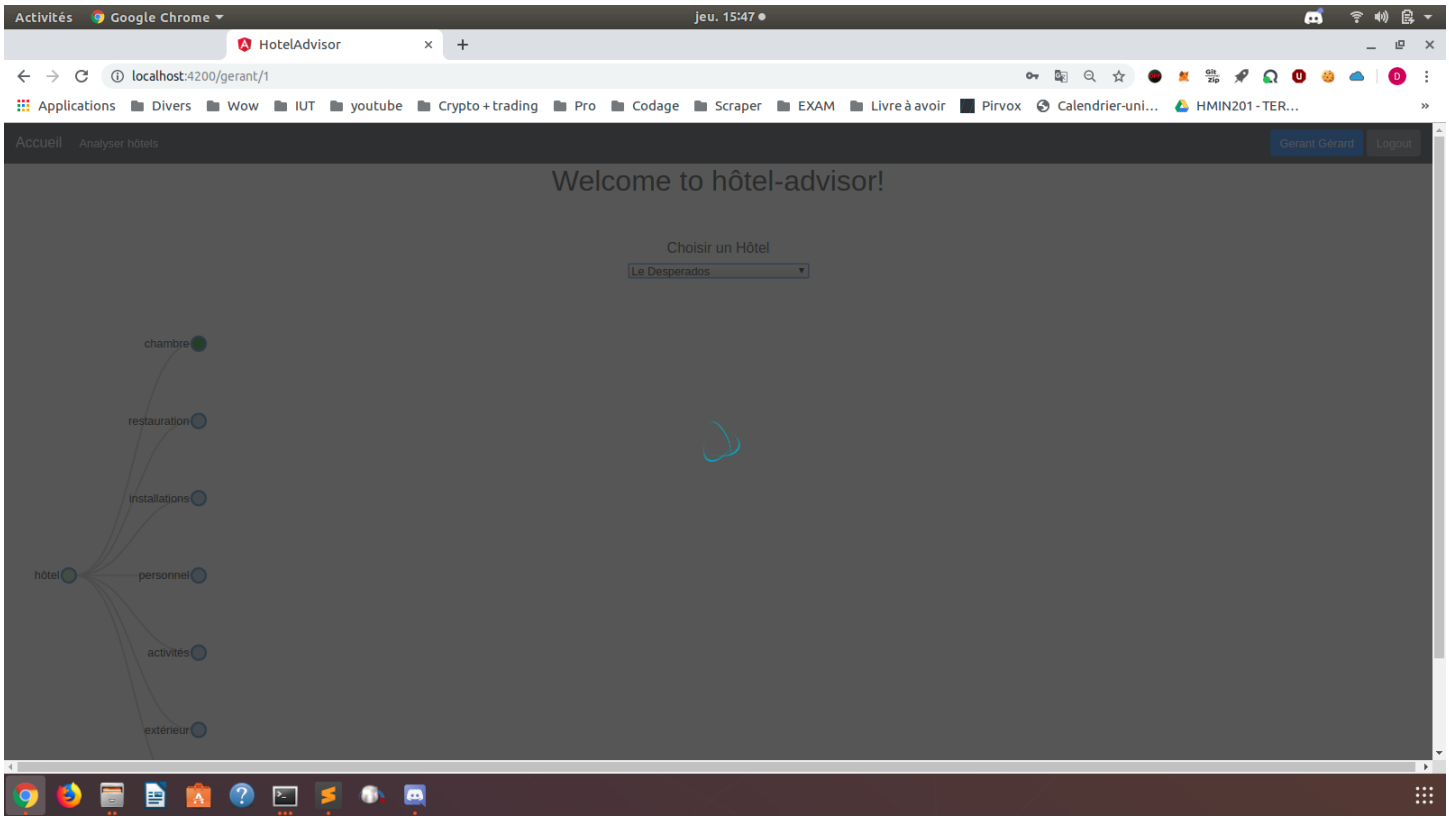
K Interface d'ajout d'avis



L Interface accueil gérant



M Interface d'attente de l'ontologie



N Interface d'analyse d'avis avec l'ontologie