UCL
Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Réalisation d’un site web participatif pour la présentation de langages, paradigmes et concepts de programmation

Mémoire présenté par
Benjamin Georges

en vue de l'obtention du grade de master en
Master [120] en sciences informatiques, à finalité spécialisée

Promoteur(s)
Kim Mens

Lecteur(s)
Kim Mens, Guillaume Maudoux, Charles Pecheur

Année académique 2016-2017

Résumé

Dans ce mémoire, nous discutons de la réalisation d'un site web participatif pour la présentation de langages, paradigmes et concepts de programmation en appliquant les pratiques du développement logiciel. Nous parlons, entre autres, de l'analyse des besoins, du fonctionnement de l'application, des tests pratiqués, etc. Nous montrons également en quoi la solution proposée remplit les besoins du client.

Remerciements

Je remercie toutes les personnes qui m'ont aidé durant le développement de ce mémoire. En particulier, je suis reconnaissant à :

- Monsieur Kim Mens pour ses conseils qui m'ont aidé à réaliser ce mémoire.
- Aline Andrianne, Hélène Clouw et Fabienne Georges pour leur relecture.

Table des matières & table des figures

Table des matières

1. Introduction 4
1.1. Contexte 4
1.2. Impératif 4
1.3. Motivation 4
1.4. Objectifs 5
1.5. Approche 5
1.6. Aperçu de l'application 5
1.7. Plan du mémoire 7

2. Connaissances de base 8
2.1. JavaScript 8
2.2. NodeJS 8
2.3. Frameworks NodeJS 9
2.4. Frameworks côté client (front-end) 10
2.5. Vues dans PostgreSQL 10
2.6. Programmation asynchrone 11

3. Énoncé du problème 13

4. Solution 14
4.1. Analyse des besoins 14
4.2. Maquette du site web 15
4.3. Architecture du serveur 20
4.4. Modèles de la base de données 21
4.5. Architecture de l'application 24
4.6. Diagrammes de séquences 25
4.7. Exemple de détails d'implémentation 36
4.7.1. Interface de la base de données 36
4.7.2. Gestion des connexions avec le « pool » 37

4.8. Maintenabilité de l’application 38
4.9. Tests de l’application 40
4.9.1. Les différents types de tests 41
4.9.2. Résultats des tests 44

5. Validation 46
5.1. Objectifs fonctionnels 46
5.1.1. Être en ligne 46
5.1.2. Être participatif 46
5.1.3. Avoir un système de soumission 46
5.1.4. Permettre l’édition d’articles 46
5.1.5. Être muni d’un outil visuel pour les liens entre les entités 47
5.2. Objectifs non fonctionnels 47
5.2.1. Être codé simplement 47
5.2.2. Performances acceptables 47
5.2.3. Maniabilité 48
5.2.4. Maintenabilité 48
5.3. Objectifs implicitement demandés par le client 49
5.3.1. Sécurisé 49
5.3.2. Robuste 50
5.4. Objectif supplémentaire atteint 51

6. Travaux futurs 53
6.1. Fonctionnalités supplémentaires 53
6.2. Amélioration du cycle de développement 54

7. Conclusion 55

Références 56

Annexes 58
A. Annexe tables de la base de données 58
B. Annexe benchmarks 63

Tables des figures

Figure 1 : Capture d'écran montrant un graphique avec les liens entre langages, paradigmes, concepts, etc. 6
Figure 2 : formulaire pour suggérer un langage 6
Figure 3 : Exemple de code synchrone 11
Figure 4 : Exemple de code asynchrone 11
Figure 5 : Maquette de l'interface de validation d'un article 15
Figure 6 : Interface finale de la validation d'un langage 16
Figure 7 : Maquette de l'interface de correction d'un article 17
Figure 8 : Interface de correction d'un article 18
Figure 9 : Maquette de l'interface du profil 19
Figure 10 : Interface du profil 19

Figure 11 : Diagramme du serveur 20
Figure 12 : Diagramme de la table langage en patte de corbeau généré par SchemaSpy 21
Figure 13 : Diagramme du fonctionnement du serveur 24
Figure 14 : Diagramme de séquence de l'envoi d'un langage en validation....................................... 26
Figure 15 : Diagramme de séquence pour visualiser un langage en validation 27
Figure 16 : Diagramme de séquence de la mise en correction d'un langage 29
Figure 17 : Diagramme d'états d'un article 30
Figure 18 : Diagramme de séquence de la publication d'un langage (partie 1).................................. 32
Figure 19 : Diagramme de séquence sur la publication d'un langage (partie 2)................................. 34
Figure 20 : Diagramme de séquence pour l'inscription 35
Figure 21 : Schéma expliquant l'interface de la base de données 37
Figure 22 : Exemple de code illustrant le principe de pool 38
Figure 23 : Méthode existInDB documentée avec le système de tags de JSDoc 39
Figure 24 : Documentation générée pour la méthode existInDB avec JSDoc..................................... 40
Figure 25 : Exemple de tests pour une fonction calculant une factorielle d'un entier positif 42
Figure 26 : Exemple de tests unitaires conduits par Mocha 42
Figure 27 : Exemple de couverture de tests par Istanbul.. 43
Figure 28 : Résultats du benchmark sur la page générant le graphique... 45
Figure 29 : Recette Ansible pour l'installation de Postfix.. 52
Figure 30 : Diagramme de toutes les tables présentes dans le schéma auteur.................................. 58
Figure 31 : Diagramme de toutes les tables présentes dans le schéma concept 58
Figure 32 : Diagramme de toutes les tables présentes dans le schéma jointure 59
Figure 33 : Diagramme de toutes les tables présentes dans le schéma utilisateur 60
Figure 34 : Diagramme de toutes les tables présentes dans le schéma langage................................ 61
Figure 35 : Diagramme de toutes les tables présentes dans le schéma paradigme 62
Figure 36 : Résultats du benchmark pour la page d’accueil.. 63
Figure 37 : Résultats du benchmark pour la page d'un langage de programmation 64
Figure 38 : Résultats du benchmark pour la page de profil 65
Figure 39 : Résultats du benchmark pour la page d'administration 66
Figure 40 : Résultats du benchmark pour la page de validation d’un langage 67

1. Introduction

1.1. Contexte

L'objectif de ce mémoire était d'établir une chronique des langages de programmation. À l'origine, cette chronique devait montrer l'évolution et les influences entre les langages à travers le temps. S'y seraient retrouvés quelques morceaux de code visant à expliquer les différents concepts apportés par ceux-ci.

À la base, l'idée venait du livre « Les Chroniques de la Science-Fiction » [1]. Cet ouvrage essaie de tracer l'histoire de plusieurs licences de science-fiction en faisant une ligne du temps sophistiquée où il est possible de voir toutes les adaptations. Ensuite, l'auteur explique ce que chaque film apporte à la licence. Cette perspective permet de voir l'évolution temporelle.

Cependant, lors des premières semaines d'élaboration du mémoire, il a été remarqué qu'il existait des travaux traitant déjà de thèmes très proches, notamment le livre « Concepts, Techniques, and Models of Computer Programming » [2] ou le livre « Concepts in Programming Languages » [3]. Dès lors, le mémoire a pris une autre orientation (création d'un site web) tout en essayant de garder le même thème.

Monsieur Kim Mens, promoteur de ce mémoire, est aussi le titulaire du cours de « Programming paradigms : theory, practice and applications » (LSINF2335)[1]. Ce cours examine quelques langages de programmation d'un point de vue historique et les compare. Monsieur Kim Mens désirait se munir d'un outil qui permettrait de regrouper des langages de programmation, des paradigmes et des concepts à un même endroit et de voir les liens entre eux. Cet outil devrait servir aux étudiants pour améliorer leur compréhension du cours et des concepts.

1.2. Impératif

L'outil désiré est tenu de faire sens avec la matière enseignée et ainsi servir de support aux élèves. Ceux-ci doivent pouvoir consulter la page d'un langage de programmation, d'un paradigme ou d'un concept et y voir les informations liées. Il faut également que l'outil soit collaboratif pour permettre aux étudiants de participer à la vie du site. Celui-ci doit donc pouvoir supporter la charge d'une quinzaine d'utilisateurs.

1.3. Motivation

En plus d'apporter un outil pédagogique au cours, ce projet devrait permettre de faciliter le passage de connaissances aux étudiants, notamment grâce à un graphe montrant le lien entre les langages, paradigmes et concepts de programmation. En effet, ce sera un dispositif complété par eux et validé par le professeur. Cette approche devrait permettre une plus grande implication des étudiants dans l'apprentissage de la matière.

[1] https://uclouvain.be/cours-2016-LSINF2335.html

1.4. Objectifs

Les objectifs fonctionnels concernant cet outil sont les suivants :

L'outil doit :

- Être en ligne (site web) ;
- Être participatif ;
- Avoir un système de soumission (un contenu doit être validé avant d'être visible sur le site) ;
- Permettre d'éditer des articles ;
- Être muni d'un outil visuel qui permet de voir les relations entre langages, paradigmes et concept de programmation.

Les objectifs non fonctionnels concernant cet outil sont les suivants :

L'outil doit :

- Être codé simplement pour qu'il puisse être perfectionné par un étudiant ;
- Avoir des performances acceptables (exemple : une page ne doit pas mettre trop de temps à charger) ;
- Être facile à comprendre et à prendre en main.

1.5. Approche

L'approche utilisée pour développer l'outil est une approche semi-itérative : l'analyse des besoins et l'architecture de l'application sont réalisées en cascade. Tandis que l'implémentation et les tests se feront en itération, avec une approche inspirée de la méthode agile Scrum. Cette méthode demande trois rôles : le propriétaire du produit, le maître Scrum et les développeurs. Monsieur Kim Mens tiendra le rôle de propriétaire du produit tandis que je tiendrai le rôle de développeur. Le poste de maître Scrum sera occupé à la fois par Monsieur Kim Mens et moi-même.

1.6. Aperçu de l'application

Dans cette section, nous donnons un rapide avant-gout de l'application grâce à deux captures d'écran. La figure 1 présente le graphique qui est généré pour mettre en évidence les interactions entre les différents langages, paradigmes, concepts, etc. La figure 2 propose un aperçu du formulaire pour suggérer un langage sur le site.

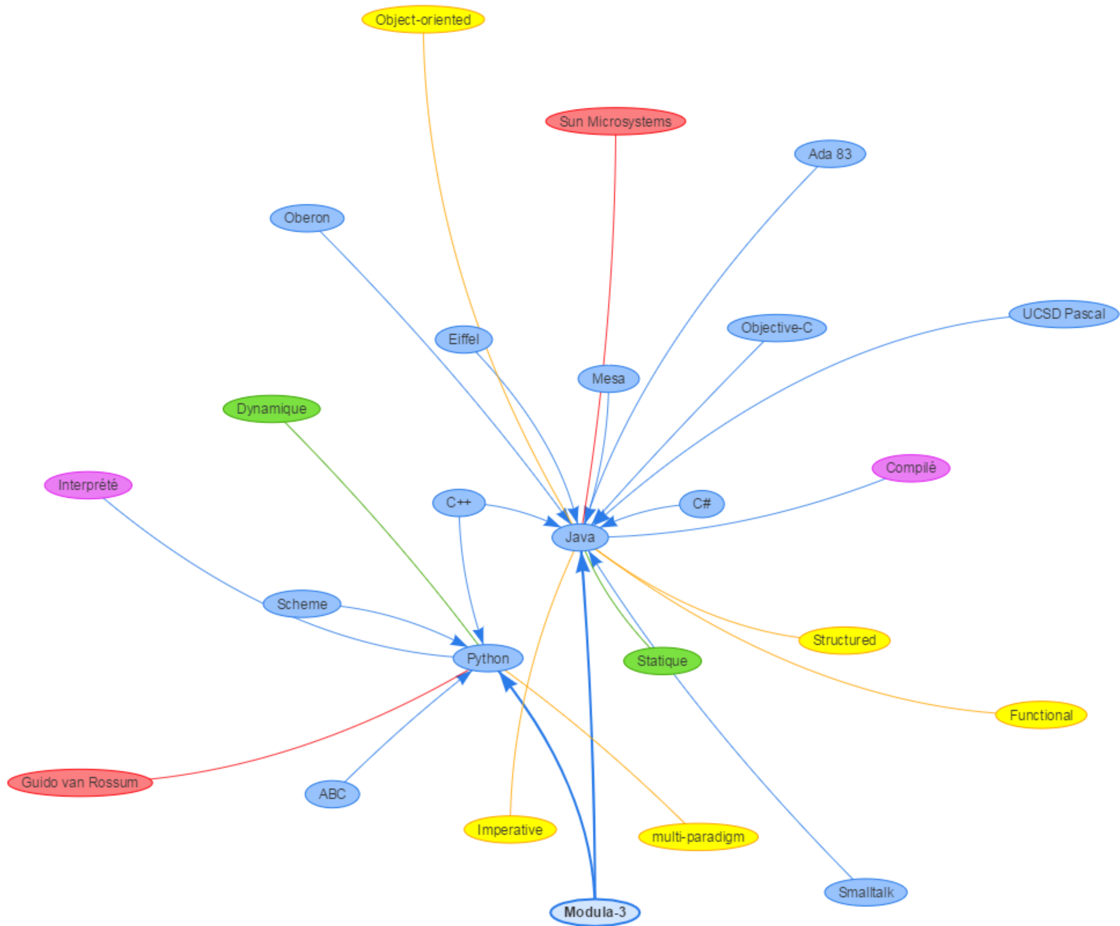

Figure 1 : Capture d'écran montrant un graphique avec les liens entre langages, paradigmes, concepts, etc.

Nom du langage de programmation | Vérifier si déjà en cours de rédaction | Ex: Java

Auteur(s) du langage de programmation | Ex: Sun Microsystems

Paradigme(s) du langage de programmation | Ex: Orientée objet, Structuré, Impératif, Fonctionel

websites | Ex: www.java.com

S'est inspiré de... | Ex: Smalltalk, C++, Eiffel, Ada 83, Mesa, Modula-3, Oberon, UCSD Pascal [...]

A inspiré... | Ex: C#, J#, Ada 2005, Gambas, BeanShell, Clojure [...]

Date(s) importante(s):

Debut période (année) | Fin période (année) | Evènement | +

Si annnée exacte: remplir les 2 champs avec la même année. Sinon, mettre l'année de début puis de fin de période.
Ex: début - 1995 | fin - 1995 | Evenement: première publication publique

Ce champs permet de donner des explications sur les dates rentrées
Ex: [source wikipédia] Java 1.0a fut disponible en téléchargement en 1994 mais la première version publique du navigateur HotJava arriva le 23 mai 1995 à la conférence SunWorld [...]

lines: 1 words: 0 0:0

Typage:

Dynamiquement typé | Statiquement typé

Figure 2 : formulaire pour suggérer un langage

1.7. Plan du mémoire

Dans la suite du mémoire, nous présentons les différentes connaissances de base nécessaires pour comprendre le fonctionnement des diverses technologies utilisées ainsi que la justification des choix opérés dans la sélection de ces technologies. Puis, nous faisons une analyse approfondie du problème rencontré par le client. Après quoi, nous expliquons le fonctionnement de la solution proposée. Ensuite, nous analysons si les besoins du client ont bien été satisfaits et si certains objectifs supplémentaires ont été réalisés. Enfin, nous terminons par la présentation de suggestions d’améliorations.

2. Connaissances de base

Dans cette section, nous allons discuter : de JavaScript qui est la technologie centrale du projet, de NodeJS qui est l'environnement JavaScript côté serveur, des frameworks NodeJS, des frameworks côté client, des vues en PostgreSQL et enfin de la programmation asynchrone.

2.1. JavaScript

Nous allons présenter JavaScript qui est la technologie principale du projet. Ce langage est à la fois utilisé du côté client et du côté serveur. Il convient donc bien de le comprendre, pour percevoir ainsi les contraintes qu'il impose au projet. Il a été créé par Brendan Eich pour Netscape en 1995 [4]. Netscape a soumis JavaScript à l'ECMA[2] pour qu'il soit standardisé. En juin 1997, l'ECMA propose l'ECMAScript comme standard [5]. Nous sommes aujourd'hui à sa huitième édition (ECMAScript 7) [6].

Actuellement, JavaScript est l'unique moyen de rendre interactive une interface HTML grâce notamment au système de listeners. JavaScript permet aussi d'utiliser l'Ajax[3] pour faire une requête au serveur sans recharger la page. Un exemple connu d'utilisation est la barre de recherche - avec correspondance dans la base de données – qui suggère des réponses en fonction de ce que l'utilisateur est en train de taper, sans que la page ne doive se recharger constamment.

Que ce soit pour les listeners ou l'Ajax, JavaScript utilise la programmation asynchrone qui permet de lancer une fonction lors du déclenchement d'un évènement. Pour les premiers, ce déclenchement peut être provoqué par un clic de souris, un déplacement dans la page ou même d'une frappe au clavier. Pour le second, le déclenchement survient lors de l'exécution une fonction lorsque la réponse du serveur est reçue (en cas de réussite, échec, etc.).

2.2. NodeJS[4]

L'application qui est du côté serveur fonctionne grâce à la technologie NodeJS. Il est donc nécessaire de la présenter pour mieux comprendre le fonctionnement de l'application.

NodeJS a été créé par Ryan Dahl [7] et est un environnement JavaScript du côté serveur. Il est principalement implémenté en C et C++. Il est axé sur 2 points : la performance et une utilisation faible de la mémoire. Pour arriver à ce résultat, il utilise un système d'évènements et de streams [8].

NodeJS utilise la programmation asynchrone pour les évènements, les opérations bloquantes (lecture ou écriture d'un fichier, par exemple) et les streams. Étant donné que NodeJS utilise le même paradigme que JavaScript ainsi que la même syntaxe, il est d'autant plus facile pour un programmeur front-end de passer en back-end et vice-versa.

NPM est le manageur de module officiel de NodeJS depuis la version 0.6.3 [9]. NPM permet d'installer facilement des modules en ligne de commande. Il permet aussi d'installer toutes les dépendances listées dans un fichier *package.json* via la commande « *npm install* » .

---

[2] European Computer Manufacturers Association
[3] Asynchronous JAvascript and Xml
[4] https://nodejs.org/en/

2.3. Frameworks NodeJS

L'utilisation de framework full-stack (un framework gérant à la fois le côté back-end et front-end comme Django[5], Laravel[6], PHPCake[7], Ruby on rails[8] …), n'a pas été retenue. En effet, ces frameworks imposent souvent les technologies à utiliser (moteur de templates, base de données, etc.). La contrainte de simplicité – car le logiciel est destiné à être perfectionné par des étudiants – nous a imposé d'éviter certaines technologies. Par exemple, la plupart utilisent des bases de données NoSQL par souci de performance. Le problème de cette catégorie de base de données est d'être plus compliquée que celles en SQL [10, p. 14], notamment parce qu'elle prône la non-utilisation de la normalisation et une gestion de l'inconsistance des données. Évidemment, ceci est subjectif et certains programmeurs pourraient penser l'inverse. Cependant, après plusieurs recherches sur le sujet, il apparaît qu'il est fortement conseillé de choisir une base de données de type SQL si aucun besoin du client ne va à l'encontre de ce choix (exemple : besoin de performance, besoin de forte disponibilité des données, etc.).

Par ailleurs, les frameworks NodeJS souffrent de la jeunesse de cette plateforme (NodeJS est sorti le 27 mai 2009 [11], il y a seulement huit ans). Beaucoup ne sont plus maintenus à jour, ce qui pose problème lors du développement d'une application. Contrairement à des frameworks full-stack connus et développés depuis longtemps, il y a un risque qu'un framework utilisé soit finalement abandonné. Donc pour être retenu, un framewrok devait être mature et permettre l'utilisation des technologies souhaitées.

ExpressJS[9] correspond parfaitement à ces deux critères. Maintenu depuis longtemps et avec une communauté très importante, il est déjà à la version 4.12.5 [12]. Ce framework ne s'occupe que du « routing » des pages. C'est-à-dire que lorsque le serveur reçoit une requête pour une page, il analyse l'URL demandée et lance une fonction qui y aura été associée. De plus, vu sa fonction principale, il n'impose aucune technologie.

Un autre framework utilisé est le moteur de template permettant de générer des pages HTML selon différentes règles. Le moteur choisi est EJS[10] [11], il a l'avantage d'avoir une syntaxe minimaliste et d'être en JavaScript. Contrairement à Pug[12] (anciennement Jade [13]), EJS ne possède pas une syntaxe propre. Il utilise du HTML et des balises délimitées par des «*<%*» et «*%>*» pour entourer le code JavaScript qui sert à générer la page.

[5] https://www.djangoproject.com/
[6] https://laravel.com/
[7] https://cakephp.org/
[8] http://rubyonrails.org/
[9] https://expressjs.com/
[10] Effective JavaScript templating
[11] http://ejs.co/
[12] https://pugjs.org/

2.4. Frameworks côté client (front-end)

Actuellement, il existe quatre grands frameworks populaires pour développer le côté client d'une application web : AngularJS, EmberJS, ReactJS et BackbonneJS. Ils sont reconnus pour être des frameworks « monopages » (SPA[13] en anglais). Les SPA se basent sur le principe de charger des morceaux de page via de l'AJAX ou de charger tous les éléments possibles en une fois, plutôt que de charger les pages une par une. Ceci a pour effet de réduire la charge réseau et CPU. En effet, le serveur ne s'occupe plus de rendre les pages HTML et laisse cette tâche au navigateur du client. Ainsi, il y a un gros chargement lors du premier accès au site web, pour charger tous les éléments nécessaires, le reste de la navigation se fait en enlevant et/ou en rajoutant des éléments à la page. Ceci peut être une solution pour les serveurs où le trafic est important (Twitter, Facebook …) pour soulager ces derniers. Comme ils ne sont plus obligés de s'occuper du rendu des pages et ne doivent en envoyer que des morceaux, la demande en ressources processeurs et réseaux est diminuée. La contrepartie de cette approche est sa complexité. En effet, l'utilisation de ces frameworks demande d'apprendre leurs spécificités (composants, data binding, etc.) propres (AngularJS a un système qui ne ressemble en rien à celui d'EmberJS). C'est ce qui constitue une des limites de cette solution.

Au final, il a été jugé plus approprié de ne pas utiliser un framework SPA. Les avantages des SPA ne sont pas exploitables dans la situation présente. Il n'y a, en aucun cas, une charge importante au niveau du site ni une demande de performance particulière. Leur utilisation augmenterait donc inutilement la complexité du projet. En effet, devoir apprendre à manier un framework, en plus du JavaScript, augmente la difficulté de maintenir ou changer le code. Or, ceci va à l'encontre de l'un des objectifs demandés : un code maintenable et extensible.

Le site utilise le framework Bootstrap[14], qui est un framework HTML, CSS et JS. Ce dernier permet de créer un site avec un design cohérent rapidement et sans demander de connaissances avancées. Il existe d'autres solutions de ce type : Semantic UI[15], Fondation 6[16],… Ils sont tous de base identique : ils offrent une série d'éléments HTML et CSS cohérents entre eux. Le choix se fera finalement par simple préférence de design. Ces frameworks répondent à un souci de créer un site web avec une apparence agréable à regarder sans devoir manier le HTML, CSS et JavaScript.

2.5. Vues dans PostgreSQL

Les vues en SQL permettent de stocker une requête. En appelant cette vue, PostgreSQL retourne le résultat de la requête. Il y a deux avantages à utiliser cette approche. La première est qu'elle permet de simplifier la requête du côté serveur (au lieu de faire la requête SQL, on appelle la vue). Deuxièmement, elle permet d'avoir une interface entre la base de données et le serveur. Ainsi, il est possible de changer la requête SQL stockée dans la vue (dans un objectif d'optimisation, par exemple) sans devoir changer le code du serveur puisque la vue continue d'exister et de renvoyer les mêmes données.

[13] Single-page application
[14] https://getbootstrap.com/
[15] https://semantic-ui.com/
[16] http://foundation.zurb.com/sites.html

Notons la présence de types de vues : les vues simples et les vues matérialisées. La première est la vue classique : à chaque appel à la vue, la requête associée est exécutée. Dans le second cas, le résultat de la requête est stocké et c'est ce résultat qui sera renvoyé après l'appel à la vue. Il faudra donc mettre la vue à jour en fonction des besoins et des changements effectués dans la base de données. Dans un souci de simplicité, il a été retenu d'utiliser le premier type de vue.

2.6. Programmation asynchrone

La programmation asynchrone est une façon de programmer telle qu'aucune opération n'est bloquante. Pour que cela soit possible, les opérations bloquantes sont déportées vers un thread pour que le processus principal puisse continuer à travailler. Cependant, cela crée des problèmes de synchronisation. Voici un exemple :

```
const fs = require('fs');

let contenue = fs.readFileSync('text.txt', {encoding: 'utf-8'});
console.log(contenue);
console.log('Hello World');
```

Figure 3 : Exemple de code synchrone

Dans ce premier morceau de code, nous connaissons l'ordre dans lequel les opérations seront effectuées. Ainsi, la première opération sera la lecture du fichier et l'exécution du programme sera bloquée jusqu'à ce que tout le contenu du fichier soit lu.

```
const fs = require('fs');

fs.readFile('text.txt', {encoding: 'utf-8'}, function(err, contenu){
    if(err){
        console.log(err);
    }
    else{
        console.log(contenu);
    }
});

console.log('Hello World');
```

Figure 4 : Exemple de code asynchrone

Dans ce second morceau de code, nous avons un exemple de code asynchrone qui est la réponse apportée au problème décrit plus haut. Ainsi, l'opération de lecture sera donnée aux threads pour que le script puisse continuer à être exécuté. Cela amène deux problèmes : déterminer l'ordre d'exécution des opérations et savoir quand l'opération est terminée. Tout d'abord, l'ordre entre l'affichage du contenu et celui du message « Hello World » est indéterminé. Ensuite, pour savoir quand l'opération sera finie, nous avons le système de « callback ». Cette approche consiste à donner une fonction à l'opération bloquante qui sera exécutée quand l'opération sera finie. On donnera à la fonction, via ses arguments, les résultats de cette opération. Ainsi, dans l'exemple, les variables « *err* » et « *contenu* » contiennent respectivement la valeur du message d'erreur (un chiffre, un texte ...) et le contenu du fichier. La fonction qui est donnée comme second argument à la fonction

readFile() est la fonction de callback qui sera donc appelée à la fin. Donc, si une erreur se produit, le message sera affiché, sinon ce sera le contenu du fichier qui le sera.

Un des premiers soucis du programmeur débutant dans l'asynchrone sera ce qu'on appelle le « callback hell ». Pour parler de ce phénomène, prenons un exemple : la lecture de plusieurs fichiers. Comme dans le second exemple, nous allons appeler la fonction *readFile()*. Ensuite dans le callback, nous allons appeler de nouveau *readFile()* pour lire le second fichier. Après quoi dans le second callback, nous allons encore utiliser *readFile()* et ainsi de suite pour lire chaque fichier. On s'expose alors à une imbrication de fonctions qui rendrait le code beaucoup moins compréhensible. En plus, il deviendrait moins maintenable de par sa complexité. JavaScript a donc adopté le principe des promesses [14, p. 483] qui permet d'aider à éviter cet empilement de callbacks. Elles ne seront pas détaillées ici, n'étant qu'une spécificité liée au code.

Ceci va influencer la façon de coder. En effet, il faudra essayer de minimiser les appels de fonctions bloquantes ou essayer de les grouper. Le site web effectue beaucoup d'opérations avec la base de données. Or, ce sont justement des appels bloquants. C'est pour cette raison que les vues ont été utilisées : combiner une série de requêtes en une seule.

3. Énoncé du problème

Monsieur Kim Mens désire se munir d'un outil en ligne permettant aux élèves de voir les différents liens entre langages, paradigmes, concepts… et leur permettant de suggérer de nouvelles entrées pour le site avec un système de validation. Ceci doit être possible via différents formulaires encadrant l'élève dans la rédaction de l'article. L'administrateur doit pouvoir le valider pour le publier sur le site ou demander à l'élève d'en corriger les erreurs. De plus, l'outil doit offrir la possibilité de visualiser les liens entre tous les éléments du site pour avoir une meilleure perception des liens entre eux.

Ceci répond à une nécessité de transmission de connaissances entre professeur et élèves. Ces derniers pourraient éprouver des difficultés à comprendre un chapitre du cours pour diverses raisons (un manque de connaissance dans le domaine ou une incompréhension d'un point théorique, par exemple). Il faut aussi tenir compte des élèves ayant des difficultés pour voir les liens entre les différents concepts vus au cours.

L'outil doit aider à pallier cette problématique. Le pan participatif du site va permettre de remédier au premier problème. Par exemple, un élève ayant eu des difficultés à comprendre un chapitre portant sur un langage de programmation et ayant réussi à surmonter ces difficultés via différentes recherches pourra transmettre ses connaissances (et donc le résultat de ses recherches) via les articles du site et ainsi potentiellement aider un autre élève qui éprouve les mêmes difficultés. Pour le second problème, la génération d'un graphique montrant l'ensemble des liens entre toutes les entités enregistrées sur le site permet de mieux visualiser leurs interactions.

4. Solution[17]

Dans cette section, nous discutons de l'analyse des besoins du client, des maquettes du site web avec le résultat final, de l'architecture du serveur en présentant les interactions avec les différents composants, des modèles de la base de données, de l'architecture de l'application, du fonctionnement de certaines fonctionnalités grâce à des diagrammes de séquences, de détails d'implémentation, de la maintenabilité de la solution proposée et enfin des tests utilisés pour valider l'application.

4.1. Analyse des besoins

Un des premiers objectifs est la simplicité de l'application pour que celle-ci puisse être étendue par un étudiant. En effet, l'application pourrait se voir greffer de nouvelles fonctionnalités s'il s'avère qu'une extension du projet pourrait être utile pour aider les étudiants. Nous pourrions, par exemple, envisager de rajouter des articles sur les différents IDE[18] existants. Il a été pris en compte que l'utilisation de multiples technologies augmentait la complexité d'un projet qui se voulait pourtant le plus simple et le moins complexe possible. Ces paramètres ont donc influencé le choix des frameworks, entre autres.

Dans ce cadre, certains choix technologiques n'étaient pas envisageables. L'utilisation des bases de données NoSQL (comme MongoDB) n'était pas appropriée. En effet, les bases de données de ce type mettent en avant la performance et la disponibilité des données au détriment du reste, ce qui amène à la « dénormalisation » des données. Une duplication des données à travers les tables a lieu, car le NoSQL n'autorise pas les opérations de type « join ». Étant des opérations lourdes et allant à l'encontre de la philosophie de ce type de base de données, ces mécanismes ne sont simplement pas implémentés. Dès lors, il faut veiller à ce que chaque information dupliquée soit bien mise à jour, supprimée, insérée… dans toutes les tables. Ainsi, les technologies de ce type sont moins facilement maintenables par rapport à des bases de données de type SQL [15].

Un des objectifs est d'avoir un site qui soit un minimum performant. En général, il est attendu qu'une page charge en moins de 3 secondes (environ) si aucune barre de chargement n'est visible sur le site [16]. Ce délai nous servira de référence pour savoir si le chargement d'une page est acceptable ou non. Un cas d'utilisation de ce délai se trouve dans l'explication de la figure 15.

En analysant les objectifs du projet, on peut en remarquer deux manquants (mais qui ont été réalisés) : la sécurité et la robustesse. En effet, il faut que le serveur web ne soit pas accessible par n'importe qui. Même si les données stockées ne sont pas sensibles, une personne mal intentionnée pourrait rendre le serveur inaccessible. De plus, il faut que celui-ci soit un minimum robuste : le serveur devrait, par exemple, pouvoir redémarrer suite à une panne.

[17] Adresse du dépôt public : https://github.com/georgesben/memoire et adresse du site web en ligne : http://alexandrie.ovh/

[18] integrated development environment

4.2. Maquette du site web

Dans cette section, nous présentons les différentes maquettes du site réalisées avec le logiciel Balsamiq[19]. Ce dernier permet de réaliser des maquettes brièvement pour avoir un rapide aperçu de la page web que l'on pourra présenter au client. À chaque fois, nous montrerons la maquette et le résultat final qui sera coupé dans un souci de limitation d'espace.

La figure 5 représente la maquette de l'interface pour la validation d'un article. L'interface ne devait être visible que pour les administrateurs. Elle était composée en quatre parties distinctes : à gauche le menu, la version de l'article sur le site web, puis le nouvel article proposé et, à droite, les remarques. Il devait ensuite y avoir une possibilité d'accepter ou de refuser l'article (celui-ci pouvant être corrigé par l'étudiant suite aux remarques que l'administrateur aurait écrites dans la section destinée à cet effet). Après présentation de la maquette au client, des améliorations ont été apportées : il devait y avoir une possibilité de sauvegarder le travail de correction pour le reprendre plus tard, le bouton « refuser » devait être renommé pour mieux correspondre à sa fonctionnalité (envoyer les remarques à l'utilisateur pour qu'il puisse corriger sa soumission) et il devait également être possible de supprimer l'article. De plus, il a été demandé de mettre en évidence les différences entre l'article sur le site et la nouvelle édition. Enfin, il devait y avoir un moyen de « rabattre » et d' « agrandir » (comme les fenêtres d'applications sur Linux, Windows ou Mac OS X) pour pouvoir adapter l'interface à des écrans de plus petites tailles (écran d'un ordinateur portable, par exemple).

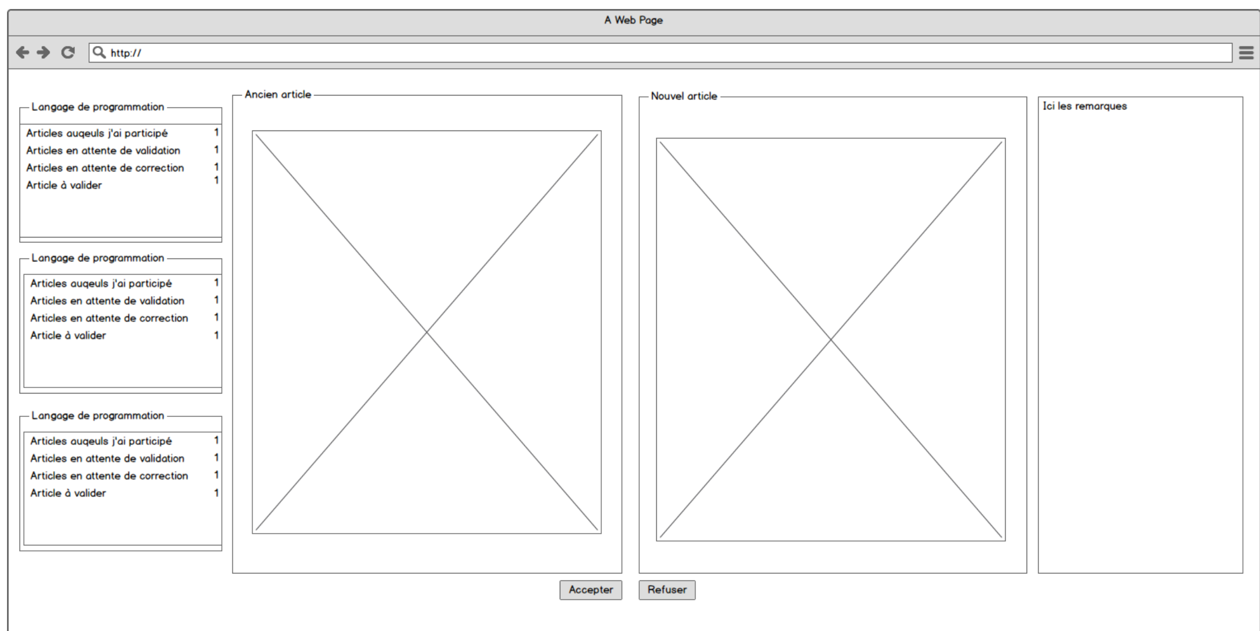

Figure 5 : Maquette de l'interface de validation d'un article

La figure 6 montre la version finale sur le site. On peut noter de nombreux changements. Tout d'abord, le menu de gauche a été changé. Il a été décidé qu'il était plus simple de diviser l'interface en deux. Une interface qui regroupe l'ensemble des fonctions utilisables pour tous les utilisateurs et une interface qui regroupe uniquement les fonctionnalités qu'un administrateur peut utiliser. Ainsi, le menu de gauche est moins dense et est plus lisible. Si l'administrateur souhaite accéder aux autres

[19] https://balsamiq.com/

fonctions, il peut y arriver via l'interface présente dans son profil (il doit cliquer sur le bouton menu en haut à droite, puis sur le lien profil).

Un autre changement concerne la partie « Ancien article » de la maquette qui est devenue une colonne pour comparer les deux articles sur le site. La comparaison se fait entre chaque champ du formulaire (par exemple : auteur(s) du langage, nom, histoire, etc.) avec un code couleur pour mettre en évidence les différences.

Enfin, la barre en haut des colonnes permet de « rabattre » une colonne libérant de l'espace pour les autres colonnes. Le bouton « réduire » est grisé si la colonne est déjà rabattue. Dans l'autre cas (où la colonne est donc agrandie), c'est le bouton « agrandir » qui est grisé. Évidemment, les boutons s'activent et se désactivent d'eux-mêmes en fonction des actions de l'utilisateur. À noter que dans la nouvelle version, les boutons « enregistrer pour reprendre plus tard » et « supprimer l'article » ont été rajoutés.

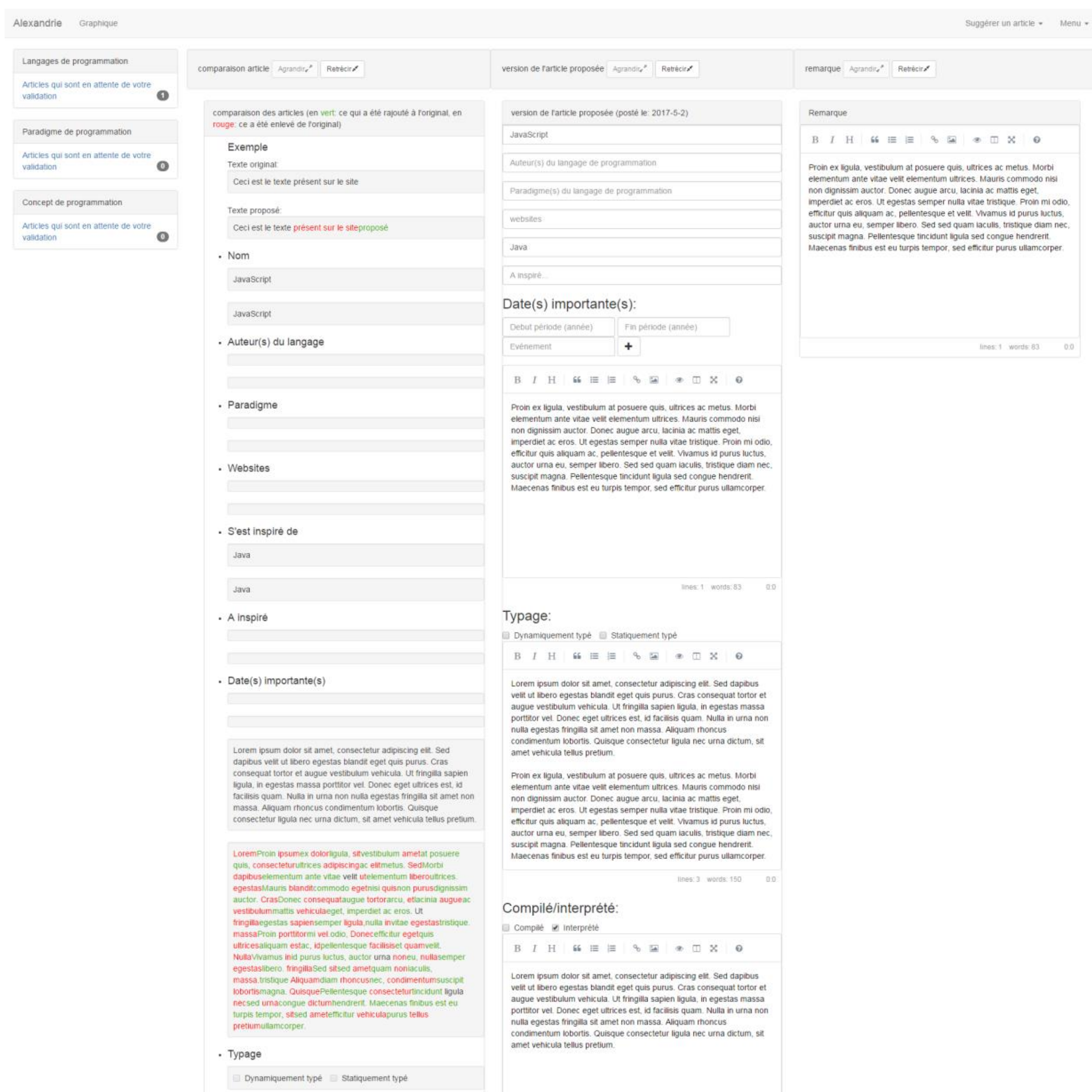

Figure 6 : Interface finale de la validation d'un langage

La figure 7 présente la maquette de l'interface pour corriger un article selon les remarques qu'un administrateur aurait faites. La maquette se divise en quatre sections : à gauche le menu, ensuite l'article présent sur le site, les remarques et à droite l'édition qui a été proposée. Enfin, les boutons « accepter » et « refuser » ont les mêmes fonctionnalités que ceux vus précédemment.

Cette maquette a été très importante pour le développement de l'application, car elle a permis de mettre en évidence un nouveau besoin non négligeable. En effet, après présentation et discussion de la maquette, il a été demandé de rajouter une fonctionnalité qui permet à l'utilisateur de sauvegarder des brouillons. Ce besoin n'avait jusqu'alors pas encore été exprimé. Même si la maquette ne concerne pas les brouillons (pour rappel, elle concerne l'interface de correction d'articles), elle a mis en avant ce problème par l'absence d'un lien « Article en brouillon » dans le menu de gauche.

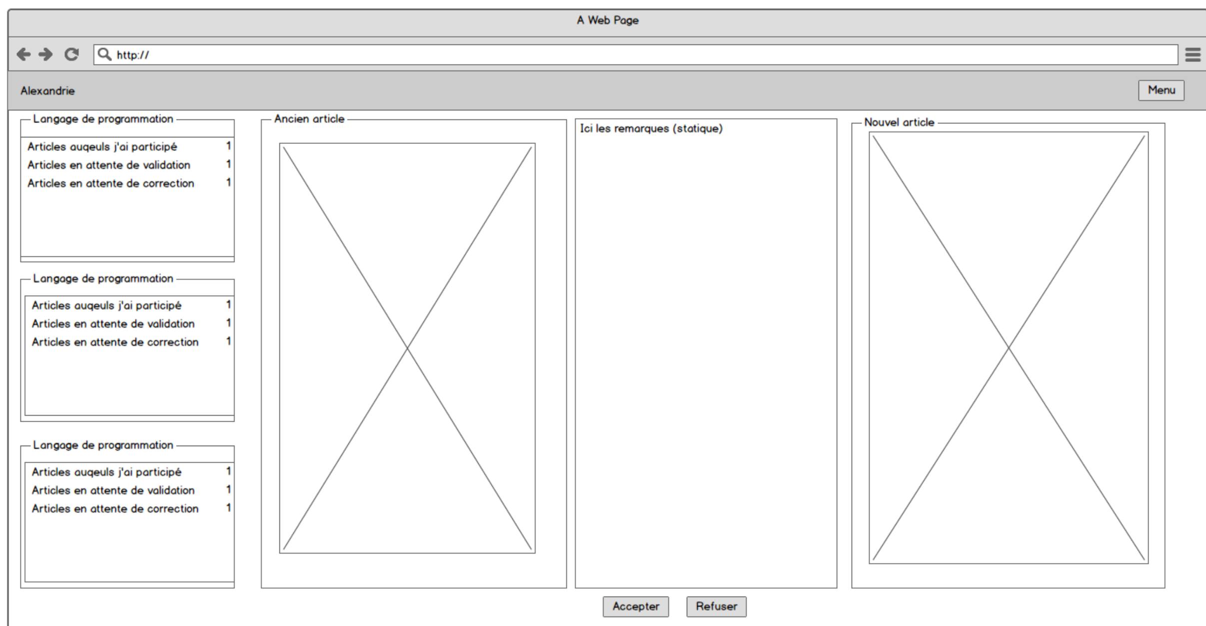

Figure 7 : Maquette de l'interface de correction d'un article

La figure 8 représente l'interface finale de correction d'articles présente sur le site. Nous pouvons noter différents changements. Désormais, Il est fait mention d'articles en brouillon, ce qui prouve bien que cette fonctionnalité a été prise en compte. Ensuite la colonne « Ancien article » de la maquette a disparu et la colonne « Remarque » a été déplacée à droite.

À nouveau, la barre au-dessus des colonnes a été rajoutée pour permettre d'agrandir ou réduire une colonne. De plus, le bouton « Enregistrer pour reprendre plus tard » a été ajouté.

Figure 8 : Interface de correction d'un article

La figure 9 présente la maquette de l'interface du profil. Cette page devait servir à l'utilisateur pour pouvoir modifier ses informations. La présence des boutons « vérifier » à côté du champ « Pseudo » et « Email » devait permettre de savoir si la valeur insérée était déjà enregistrée ou pas. En appuyant sur le bouton, un message devait apparaître pour indiquer si la valeur insérée (email ou mot de passe) était disponible. En était de même pour le bouton « valider » qui devait afficher si les nouvelles valeurs avaient bien été enregistrées ou non.

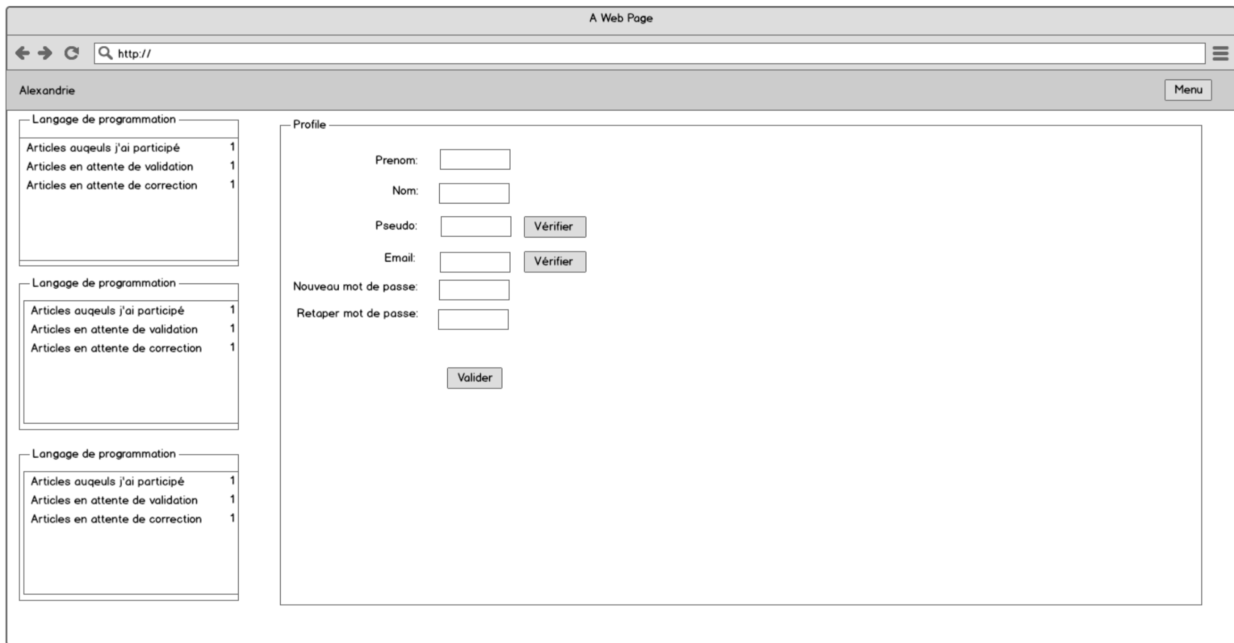

Figure 9 : Maquette de l'interface du profil

La figure 10 présente l'interface du profil sur le site web. La version finale est très proche de la maquette. Les champs « nom » et « prénom » ont été supprimés, car ils n'étaient pas employés sur le site. De plus, une explication a été ajoutée sur le fonctionnement de la page.

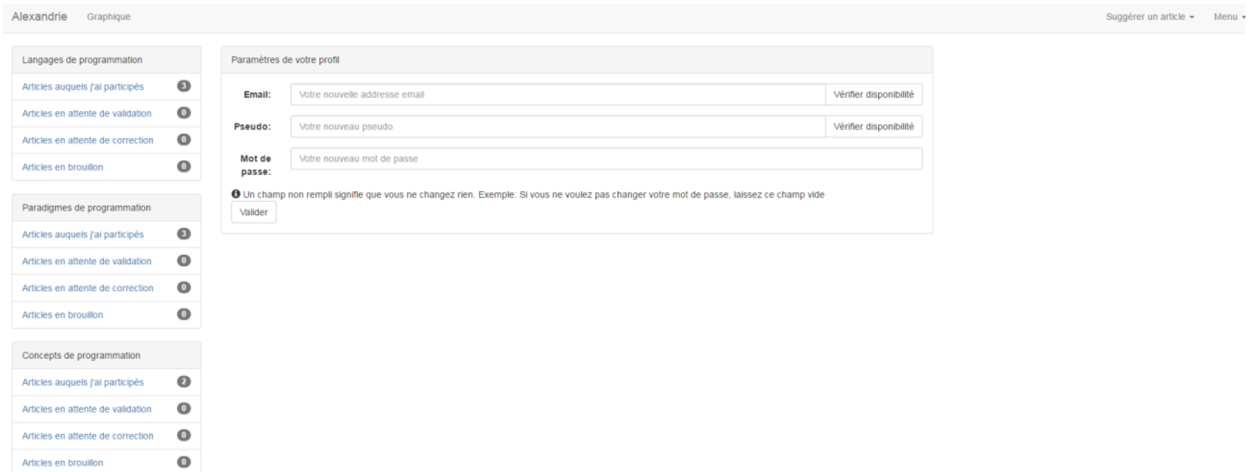

Figure 10 : Interface du profil

4.3. Architecture du serveur

La figure 11 présente l'architecture du serveur et propose une explication sur le traitement d'une requête au sein du serveur.

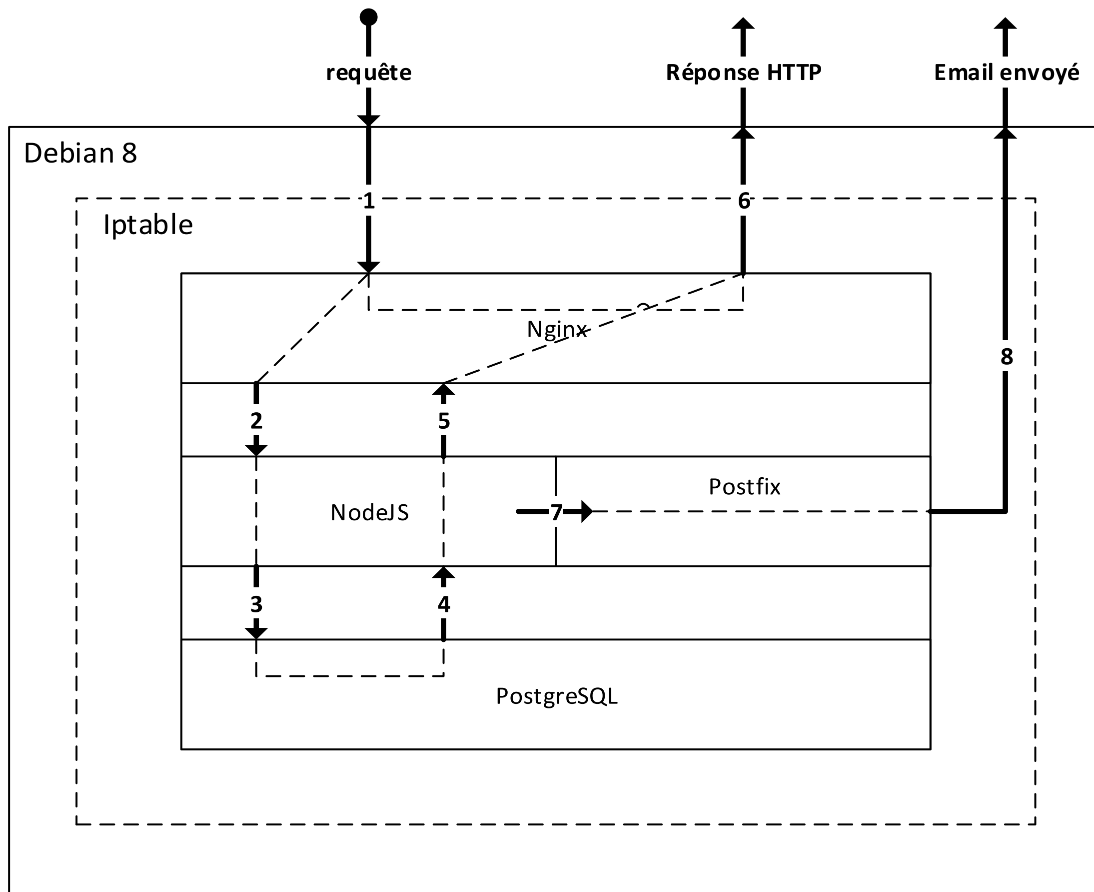

Figure 11 : Diagramme du serveur

Le serveur utilise Debian 8 comme système d'exploitation, Iptable comme firewall, Nginx comme reverse-proxy, NodeJS comme application, Postfix pour l'envoi de mails et PostgreSQL comme base de données. La gestion d'une requête se passe comme ceci :

1. Le serveur reçoit une requête HTTP qui est filtrée par Iptable. Si la requête arrive par le port 80 ou 443, elle est acceptée par le firewall. Sinon, elle est simplement abandonnée ;
2. NGINX traite la requête : si la réponse à cette requête se trouve dans son cache, la réponse est immédiatement renvoyée (étape 6). ;
3. Si besoin, NodeJS envoie une requête SQL à PostgreSQL ;
4. La réponse à la requête est envoyée à l'application en NodeJS ;
5. NodeJS envoie la réponse HTTP à Nginx ;
6. Nginx regarde s'il peut mettre la réponse en cache. Dans tous les cas, la réponse est envoyée au client ;
7. Il se peut que l'application doive envoyer un mail. NodeJS le transmet à Postfix ;
8. Postfix se charge d'envoyer le mail par le port 25 (port ouvert dans le firewall).

4.4. Modèles de la base de données

Pour pouvoir discuter des modèles de la base de données, un outil a été choisi pour générer l'image des différentes tables : SchemaSpy[20]. Cet outil est libre et sous licence Lesser GNU Public License 2.1[21]. Il a été écrit en Java et permet de construire les schémas représentant les tables présentes dans une base de données en s'y connectant. Il collecte les différentes informations à propos des tables et des colonnes (clé primaire, contraintes, valeur par défaut, propriétaire, etc.) et il utilise le programme Graphviz[22] pour créer une image (comme à la figure 12) et ainsi représenter visuellement la table.

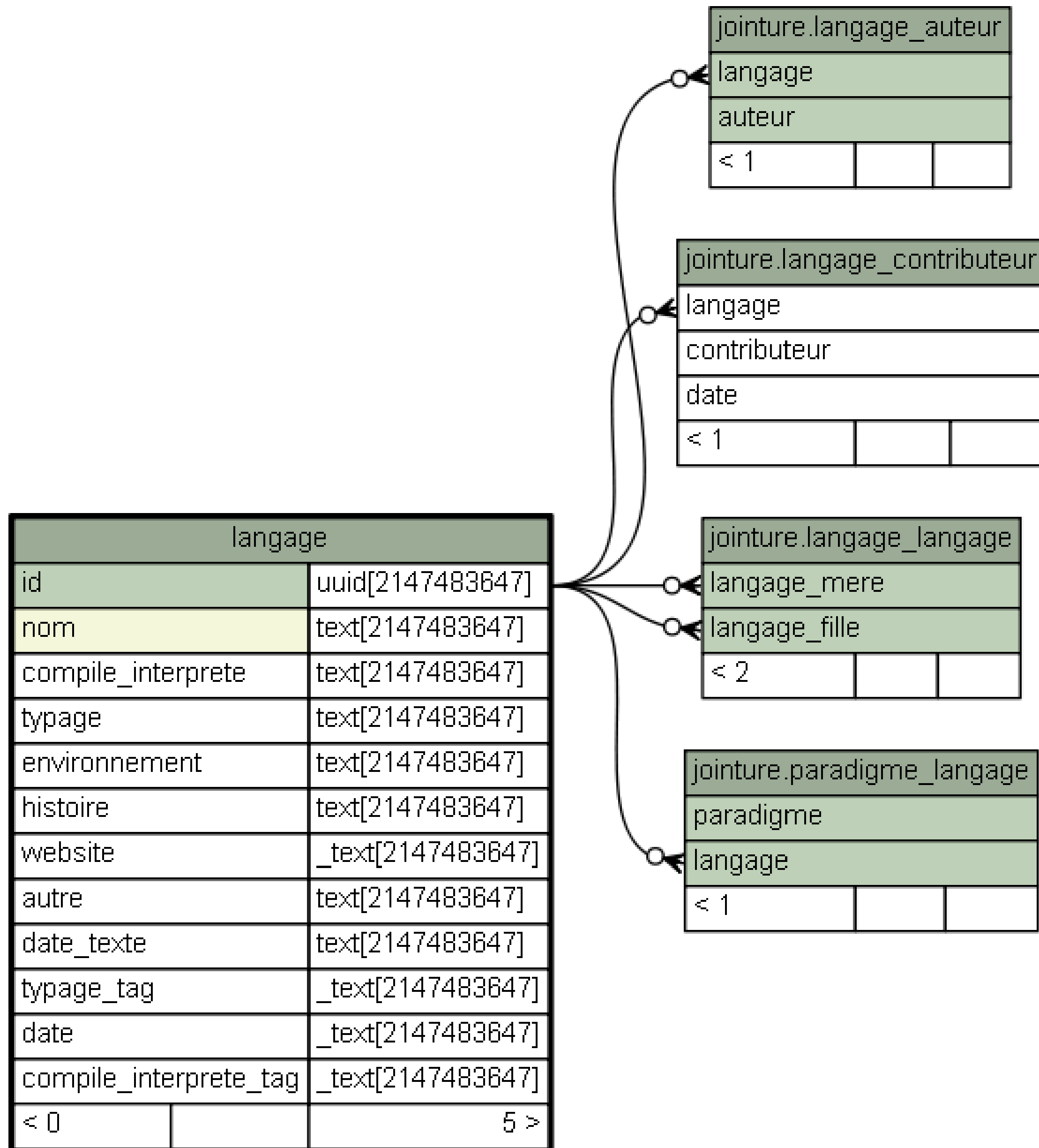

Figure 12 : Diagramme de la table langage en patte de corbeau généré par SchemaSpy

[20] http://schemaspy.sourceforge.net/
[21] https://www.gnu.org/licenses/lgpl-2.1.html
[22] http://www.graphviz.org/

La notation utilisée pour décrire la table est la notation « patte de corbeau » [17]. Cette notation sert à décrire l'unicité entre une entrée d'un champ et les colonnes des autres tables. Un cercle représente un 0 et la « patte de corbeau » représente le fait qu'il peut y en avoir plusieurs. Par exemple, si on prend la liaison entre la table *langage* et la table *jointure.langage_auteur*, on pourra dire que : « Un ID qui se trouve dans la table *langage* pourra se trouver 0 ou plusieurs fois dans la colonne *langage* de la table *jointure.langage_auteur* ». Les diagrammes des autres tables se trouvent en annexes par souci de lisibilité.

Pour démontrer que la base de données ne possède pas de données redondantes, nous allons devoir démontrer qu'elle est en forme normale 1, 2 et 3 de manière incrémentale. Ensuite, nous expliquerons pourquoi la base de données n'est pas en forme normale de Boyce Codd et nous en justifierons la raison. Nous n'irons pas plus loin dans l'analyse étant donné que la plupart des bases de données commerciales s'arrêtent à la forme normale de Boyce Codd [18, p. 536]. De plus, seule l'analyse de la table *langage* sera faite (faire la même analyse pour toutes les tables ne serait pas pertinent), mais la démarche expliquée peut être appliquée aux autres tables.

La méthodologie pour développer la base de données a été simple. Elle a été faite sur papier et analysée pour respecter les formes normales 1, 2 et 3. Dans la mesure où la base de données devait être modifiée pour correspondre aux changements de besoins du client, à chaque modification, elle a été examinée pour garantir la conservation des formes normales.

- Termes techniques
  - Attribut : « Les composants de même rang des lignes forment un attribut de la relation. Les attributs d'une relation portent des noms distincts. » [19, p. 72]
  - Relation : « Une relation est un ensemble nommé d'agrégats de n valeurs, chacune appartenant à un domaine. Nous appellerons ligne un tel agrégat (la théorie préconise le terme de n-uplet). » [19, p. 72]
  - DF : « Une dépendance fonctionnelle (DF) K → L existe dans une relation entre un ensemble K d'attributs (le déterminant) et un ensemble L d'attributs (le déterminé) si, dans tout couple de lignes qui ont mêmes valeurs de K, les valeurs de L sont aussi les mêmes. Une dépendance fonctionnelle exprime une propriété du domaine d'application. » [19, pp. 97–98]

- 1FN (forme normale 1) : « Une relation est en 1FN si les domaines d'attributs incluent seulement des valeurs atomiques (simples, indivisibles) et chaque tuple d'attribut est une seule valeur du domaine » [20, p. 25]

  Pour que cette forme soit respectée, il faut que les valeurs stockées soient atomiques (pas de tableau par exemple). Même si la table langage utilise des tableaux (représentés par un « _ » devant le type de données), ils sont perçus comme des valeurs atomiques. En effet, le système traite le tableau et non les données à l'intérieur. En réalité, le tableau est une facilité d'utilisation qui permet de stocker un ensemble de dates. Nous aurions pu utiliser une chaine de caractères pour conserver les dates, avec évidemment un choix approprié de séparateur. Dès lors, l'utilisation

d'un tableau pour les stocker s'est imposée, car il permettait d'éviter de devoir choisir un séparateur adéquat.

Par exemple, l'attribut « date » est bien atomique, car aucune requête n'est faite pour obtenir une partie de l'attribut. Ce qui en fait un attribut indivisible. Ceci est valide par rapport à l'analyse des besoins, car aucune recherche par date n'a été demandée. Cependant, si une telle fonctionnalité devait être implémentée, il faudrait revoir la structure de la base de données.

- 2FN (forme normale 2) : « Un schéma relationnel R est en 2FN si chaque attribut non-clé A dans R est entièrement fonctionnellement dépendant de la clé primaire de R » [18, p. 523]

  Pour que cette forme soit respectée, il faut que les attributs, qui ne forment pas la clé primaire, ne dépendent pas que d'une partie de cette clé. Étant donné que notre clé primaire est juste un ID (elle n'est donc pas composée de plusieurs attributs), il est impossible que les attributs ne dépendent seulement que d'une partie de la clé. La table respecte bien la forme 2FN.

- 3FN (forme normale 3) : « Selon la définition originale de Codd, un schéma de relation R est en 3FN s'il satisfait 2FN et si aucun attribut non clé de R est transitivement dépendant de la clé primaire » [18, p. 524]

  Pour que cette forme soit respectée, il faut que les attributs, qui ne forment pas la clé primaire, ne dépendent pas d'autres attributs qui ne forment pas la clé primaire. Tous les attributs, sauf ID, ne forment pas la clé primaire. Or, aucun attribut de cet ensemble ne dépend d'un autre. La table respecte donc bien la forme 3FN.

- FNBC (forme normale de Boyce Codd) : « Un schéma relationnel R est en FNBC si pour chaque dépendance fonctionnelle non triviale X → A tient dans R, alors X est une superclé de R » [18, p. 529]

  Si on regarde les DF de la table, nous remarquons que nous avons :

  {id} → {nom, compile_interprete, typage, environnement, histoire, website, autre, date_texte, typage_tag, date, compile_interprete_tag}

  {nom} → {id, compile_interprete, typage, environnement, histoire, website, autre, date_texte, typage_tag, date, compile_interprete_tag }

  Comme le nom du langage est unique, celui-ci peut agir comme clé primaire. Cependant, comme il ne fait pas partie de celle-ci, la table n'est donc pas en FNBC.

  Néanmoins, c'est un choix de design volontaire. En effet, il était prévu depuis la conception de pouvoir changer les informations d'un langage (via l'édition) et même le nom. Ce qui implique que si le nom était la clé primaire et qu'on souhaitait le modifier, il faudrait récupérer l'ancienne entrée, puis insérer une nouvelle entrée avec le nom changé et les anciennes valeurs, ensuite remplacer l'ancienne valeur par la nouvelle dans toutes les entrées des tables de jointures et enfin supprimer

l'ancienne entrée devenue inutile. Ce processus est long et peut devenir très vite compliqué si le nombre de tables de jointures à modifier est important. Actuellement, avec le système mis en place, il suffit de changer le nom dans l'entrée sans devoir éditer les tables de jointures.

4.5. Architecture de l'application

Le serveur possède une architecture MVC[23] : la partie modèle s'occupe de la logique de l'application, et la vue traite l'affichage des données. Le contrôleur, quant à lui, se charge de récupérer les données à partir des requêtes HTTP reçues et de les passer au modèle. Ce dernier lui renverra une réponse qu'il peut formater pour que les données correspondent au format attendu du côté de la vue.

Le modèle charge une configuration au lancement du serveur. Cette action lui permet de connaître les identifiants ainsi que l'adresse de la base de données. Ainsi il suffit de modifier le fichier de configuration, sans changer le modèle, pour se connecter à une autre base de données ou utiliser d'autres identifiants.

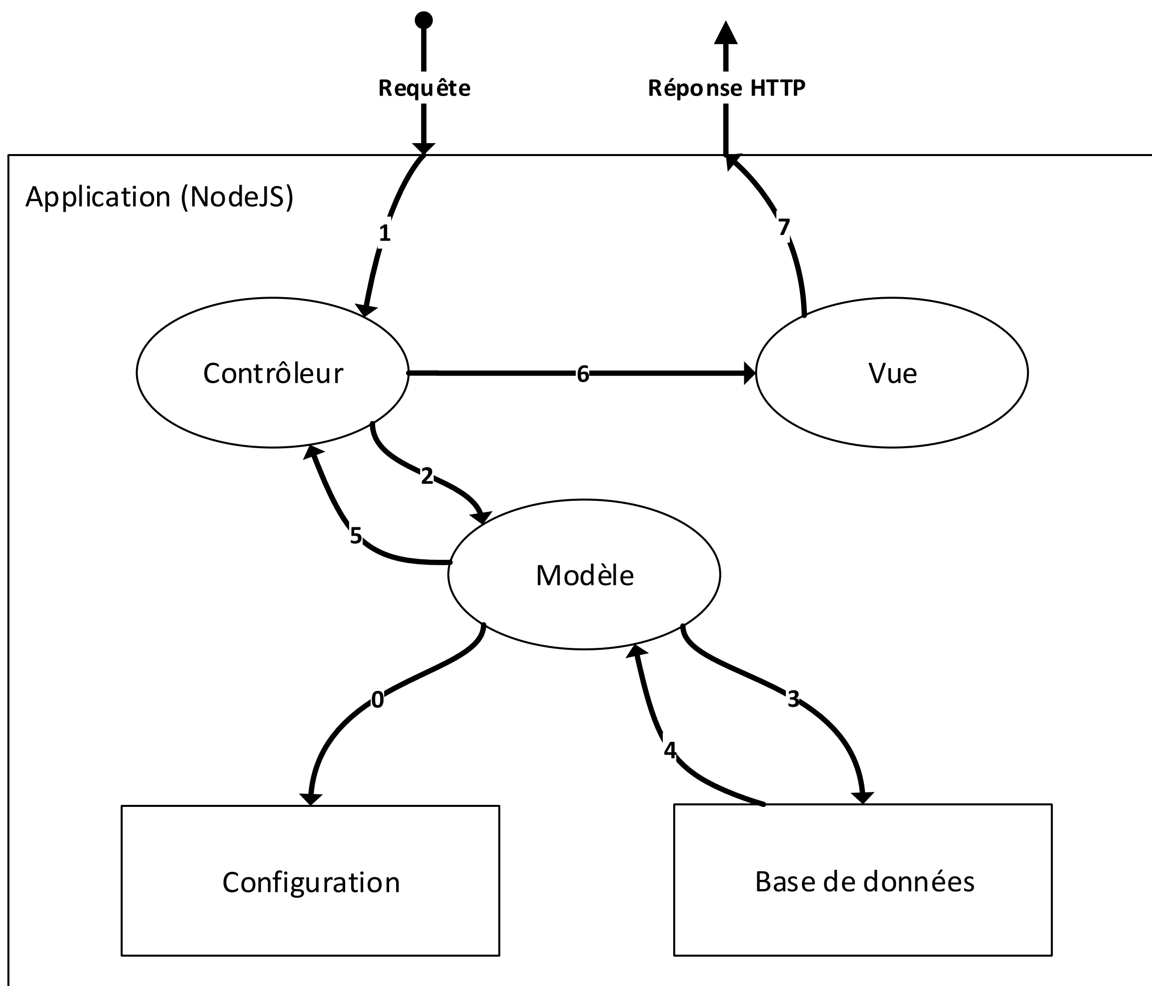

Figure 13 : Diagramme du fonctionnement du serveur

[23] Modèle-Vue-Contrôleur

Voici comment se passe le traitement d'une requête HTTP :

0. Au lancement de l'application, le modèle charge la configuration pour pouvoir se connecter à la base de données. Cette étape n'est effectuée qu'une seule fois durant le cycle de vie de l'application ;
1. Une requête est reçue par l'application. Le routing décide à quel contrôleur envoyer la requête selon l'adresse URL fournie avec la requête ;
2. Le contrôleur récupère les informations à partir de la requête et envoie les informations traitées au modèle ;
3. Le modèle interagit avec la base de données sur base de requêtes SQL ;
4. La base de données envoie une réponse pour chaque requête reçue ;
5. Le modèle envoie une réponse au contrôleur ;
6. Le contrôleur formate la réponse et l'envoie à la vue ;
7. La vue génère la page HTML et l'envoie au client.

Ceci est une explication générale du processus. Il y a bien évidemment des exceptions comme les requêtes AJAX qui ne nécessitent aucune génération de pages HTML.

4.6. Diagrammes de séquences

Dans les différents diagrammes de séquences de cette section, un bloc « En parallèle » non standard est utilisé (il est dessiné en pointillé pour éviter toute ambiguïté). Ce bloc permet de représenter le fait que l'ensemble des actions présentes dans le bloc peuvent se faire dans n'importe quel ordre. En effet, avec la programmation asynchrone, il n'y a plus aucune garantie de l'ordre d'exécution (voir le paragraphe 2.6 sur la programmation asynchrone pour plus de détails). Par contre, le bloc « En parallèle » garantit qu'à la fin de celui-ci, toutes les actions à l'intérieur sont achevées.

De plus, les réponses qui ne demandent aucun détail (résultat d'un appel de fonction, par exemple) ne sont pas représentées. Il en est de même pour les activations imbriquées qui ne sont pas reproduites dans un souci de clarté. À noter que, de nouveau, les réponses de la base de données ne sont pas représentées explicitement dans ces diagrammes pour que ceux-ci restent compréhensibles. Mais cela ne signifie pas que la base de données ne renvoie aucune réponse. En effet, à chaque requête, la base de données transmet une réponse pour indiquer si la requête a fonctionné ou pas. C'est grâce à cela que le modèle est capable de répondre au contrôleur et de lui signaler un éventuel problème.

Dans cette section, nous ne traitons que de la partie concernant les langages, mais les mécanismes qui sont expliqués sont également valables pour les paradigmes et les concepts.

Il convient également d'expliquer le processus de publication et d'édition avec la base de données. Les langages qui sont visibles sur le site sont stockés dans la table *langage*. Ce sont des langages qui ont été validés par l'administrateur et créés par un utilisateur ou par le site. Par exemple : dans la fiche du langage Java, si l'utilisateur fait mention du langage C++ dans les langages liés et que la fiche est validée, le site va générer automatiquement la page (vide) du langage C++ si elle n'est pas déjà présente. Les langages qui sont édités doivent, quant à eux, rester invisibles pour les visiteurs du site tant que l'administrateur ne les a pas validés. Il faut donc les conserver dans une autre table

(*langage_en_creation*) pour que l'administrateur puisse faire son travail. Ainsi, si l'édition du langage est validée, celle-ci remplacera la version du site (et donc celle stockée dans la table *langage*).

La figure 14 montre comment se passe l'envoi d'un langage de programmation en validation. Un langage mis en validation est un langage qui doit être expertisé par l'administrateur. Les données sont envoyées par une requête HTTP POST via un formulaire.

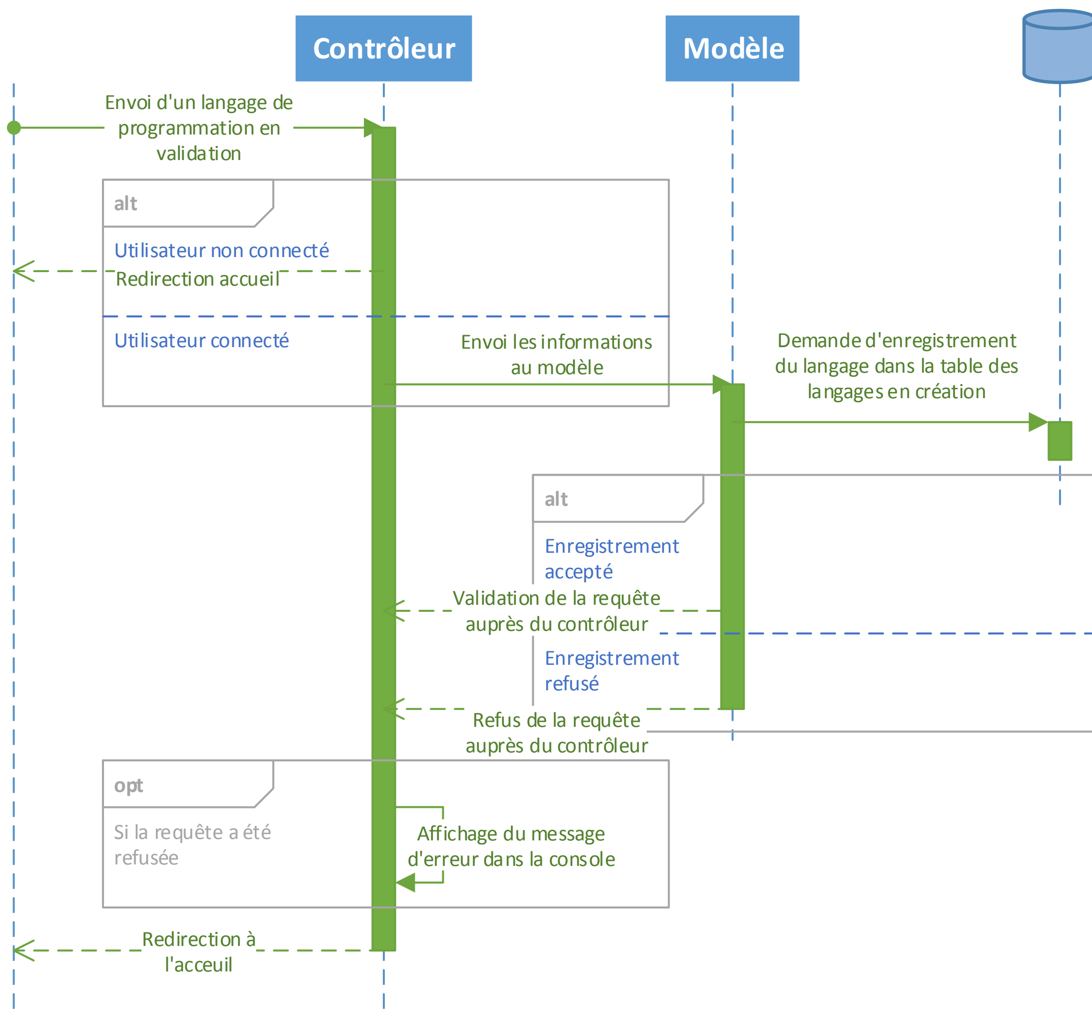

Figure 14 : Diagramme de séquence de l'envoi d'un langage en validation

Le contrôleur reçoit la requête une fois le routing fait. Il analyse si l'utilisateur est connecté ou pas, via les sessions actives du site. Si l'utilisateur n'est pas connecté, le contrôleur demande une redirection à l'accueil pour l'utilisateur. Dans l'autre cas, les données sont envoyées au modèle. Le modèle reçoit les informations et envoie une requête à la base de données pour insérer le langage dans la table *langage_en_creation*. Une réponse est envoyée de la base de données vers le modèle. Ce dernier a deux comportements : en cas de réussite de l'enregistrement, le modèle avertit le contrôleur que la requête a abouti, dans l'autre cas, le modèle avertit le contrôleur que la requête a échoué. Le contrôleur n'a donc plus qu'une chose à faire en cas d'échec : afficher le message d'erreur dans la console (pour pouvoir analyser les logs et ainsi régler le problème). Enfin, le contrôleur envoie une requête HTTP de redirection à l'accueil.

Seul un administrateur a les autorisations pour accéder aux langages qui sont en attente de validation. Cette étape lui permet de voir les informations sur le langage suggéré par l'utilisateur. La figure 15 montre le diagramme de séquence expliquant la procédure de cette fonctionnalité.

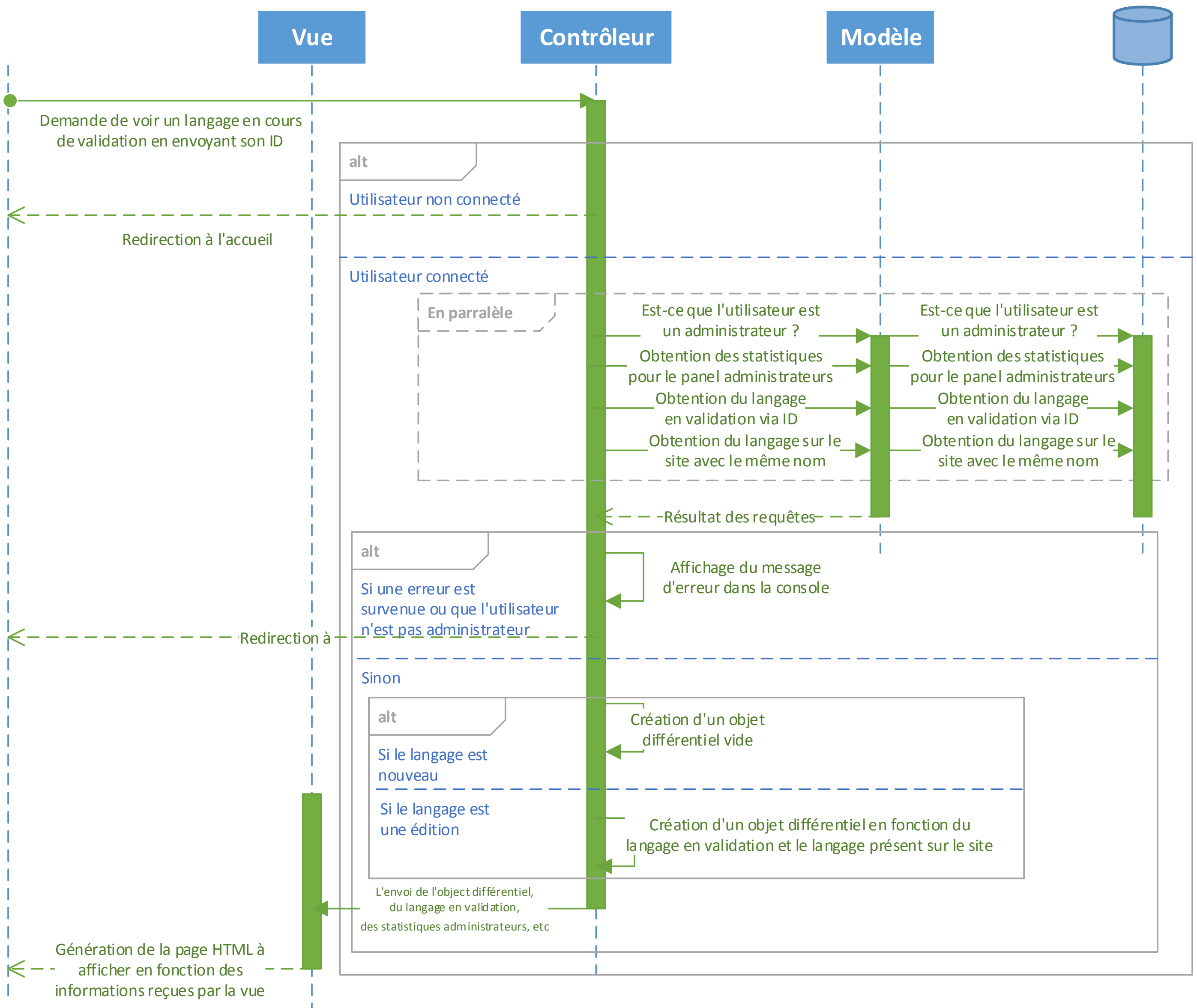

Figure 15 : Diagramme de séquence pour visualiser un langage en validation

Lorsque l'administrateur demande à voir le langage, la requête est envoyée au contrôleur après le routing. Le contrôleur vérifie si l'utilisateur est connecté, via les sessions actives du site. Si l'utilisateur n'est pas connecté, il est immédiatement redirigé à l'accueil.

Une fois la réponse reçue par le contrôleur, ce dernier l'analyse pour savoir quel comportement adopter. Si une erreur est survenue durant les interactions avec la base de données, le message d'erreur est affiché dans la console. Comme cette section du site ne doit être accessible qu'aux

administrateurs, le contrôleur vérifie également que l'utilisateur est bien un administrateur. Si l'utilisateur n'est pas un administrateur, il est aussitôt redirigé à l'accueil.

Lors de l'analyse des besoins, il a été mis en évidence qu'il devait y avoir un moyen d'exposer les différences entre une édition proposée pour un langage et la version sur le site. Il y a donc deux cas possibles : soit le langage proposé est nouveau, soit le langage proposé est en réalité une édition de celui présent sur le site. Comme le modèle a demandé à obtenir le langage avec le même nom que celui proposé et qu'il a passé le résultat au contrôleur, ce dernier sait immédiatement à quoi s'en tenir. Dans les deux cas, il va créer un objet différentiel. Si le langage n'existe pas sur le site, l'objet créé est un objet vide. Dans l'autre cas, le contrôleur fait usage de la librairie « diff » [21] qui utilise l'algorithme proposé par Myer [22] sur chaque composant de l'article avec l'élément correspondant dans l'édition proposée. L'algorithme essaie de déterminer si un élément a été rajouté ou enlevé. Après plusieurs tentatives, il s'est avéré qu'une analyse caractère par caractère pouvait prendre énormément de temps. Un test a montré qu'une analyse d'un texte conséquent pouvait atteindre environ 17 secondes. En demandant à l'algorithme d'examiner le texte par mot et non par caractère, l'exécution s'est faite en 1,2 seconde environ. C'est pour cette raison que les différences se font sur les mots et non les caractères.

Une fois que l'objet différentiel a été calculé, les informations nécessaires à la vue pour générer la page HTML lui sont transmises. Aussitôt que la page a été créée, elle est envoyée à l'administrateur.

Un administrateur doit pouvoir mettre un langage en correction. Quand ce dernier est mis en correction, cela signifie que l'auteur original doit apporter des modifications à son texte en fonction des remarques que l'administrateur aura faites. De plus, un administrateur peut directement apporter une modification dans certains cas : faute d'orthographe, mot manquant, etc. La figure 16 montre comment cette fonction est réalisée par l'application.

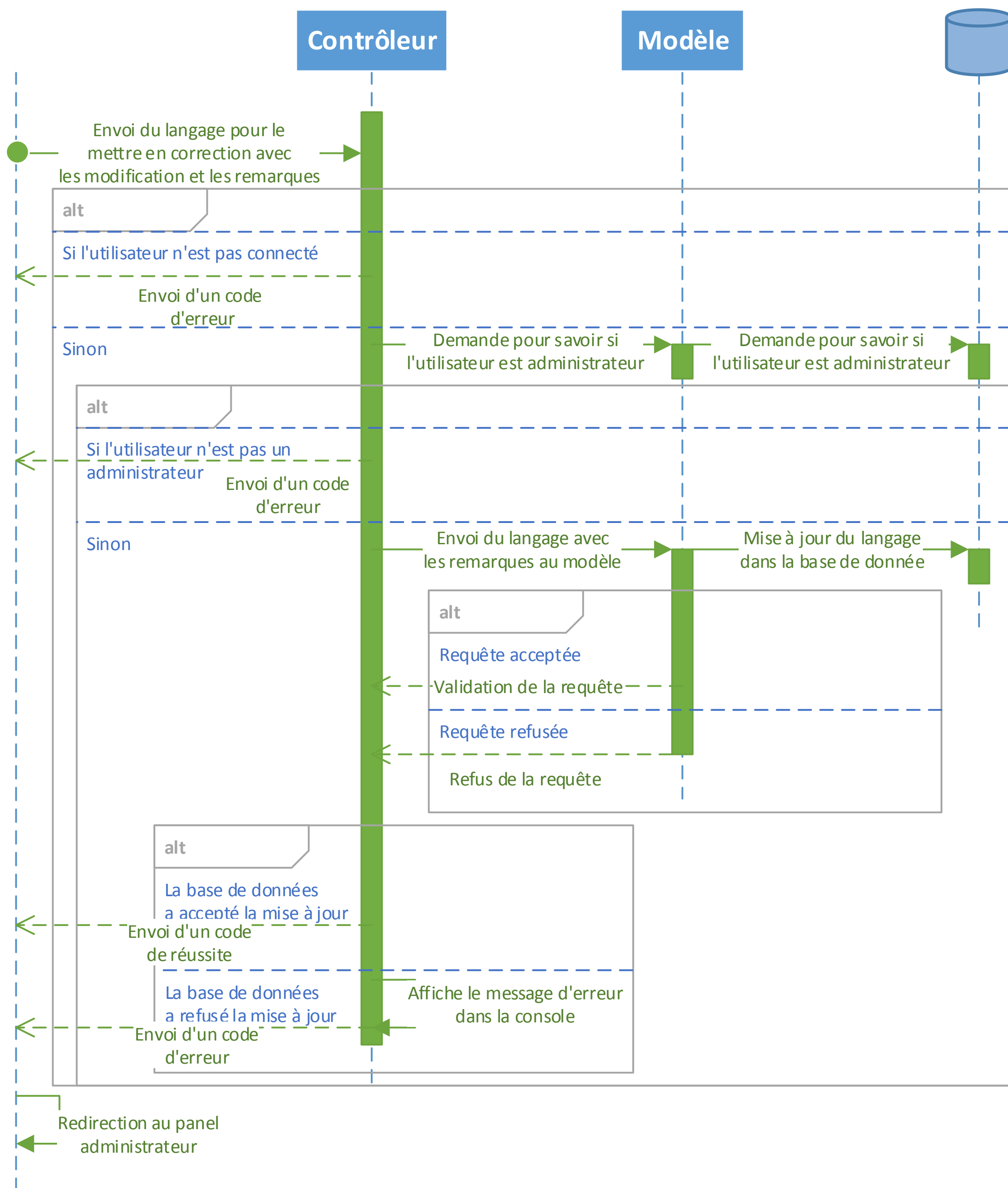

Figure 16 : Diagramme de séquence de la mise en correction d'un langage

Lorsque l'administrateur envoie le langage avec les remarques, la requête est acheminée au contrôleur après le routing. Celui-ci vérifie, via les sessions actives, qu'elle vient bien d'une personne connectée au site. Si c'est le cas, le contrôleur vérifie ensuite que la personne est bien un administrateur. Pour cela, il demande au modèle d'interroger la base de données en lui communiquant l'identifiant de la personne. Une fois la réponse récupérée, elle est passée au contrôleur qui va agir en conséquence. Dans les deux cas où l'utilisateur n'est ni connecté, ni un administrateur, celui-ci reçoit un code d'erreur.

Sinon, le contrôleur poursuit la suite des opérations. Il envoie au modèle le langage avec les remarques qu'il a reçues de l'administrateur. Le modèle transmet ensuite une demande de mise à jour à la base de données pour modifier l'état de l'article et les remarques qui lui sont associées. Un article a deux états possibles : soit il est en attente de validation, soit il est en attente de correction. Quand l'article est en attente de validation, il faut le modifier (via le message « Mise à jour du langage dans la base de données »). Le contrôleur est informé de la réussite de la mise à jour ou non. Si la requête a réussi, il envoie un code de réussite à l'administrateur. Dans le cas contraire, il affiche le message d'erreur dans la console et envoie un code d'erreur à l'administrateur. Enfin, dans tous les cas, l'administrateur est redirigé vers le panel administrateurs.

Il est intéressant de s'arrêter sur cette mécanique (mettre un langage en validation ou en correction) qui est l'une des fonctionnalités centrales du site. La figure 17 montre ce mécanisme via un diagramme d'états UML. En effet, elle permet de donner la dimension participative et interactive. Quand un langage est soumis pour la première fois ou qu'il soit une édition d'un langage présent sur le site, la nouvelle version est dans l'attente de validation. Si l'administrateur juge bon que l'utilisateur y apporte quelques modifications en suivant ses remarques, il met l'article en correction. L'utilisateur peut donc ensuite corriger son article en suivant les remarques et le proposer à nouveau. Dès lors, l'article passe à nouveau en attente de validation. La seule différence par rapport à avant est que l'article vient avec les remarques de l'administrateur pour qu'il puisse voir si l'étudiant a bien effectué les changements demandés. Ainsi, le document continuera à changer d'état tant que l'administrateur jugera qu'il est nécessaire d'apporter des corrections. Il est également possible de mettre un article en validation à partir d'un brouillon. L'auteur peut éditer son brouillon autant de fois qu'il le souhaite. Une fois satisfait de son travail, il peut le soumettre pour le valider par un administrateur. Quand celui-ci n'a plus de remarque à formuler concernant l'article, il peut le publier sur le site pour qu'il soit visible par tous les visiteurs.

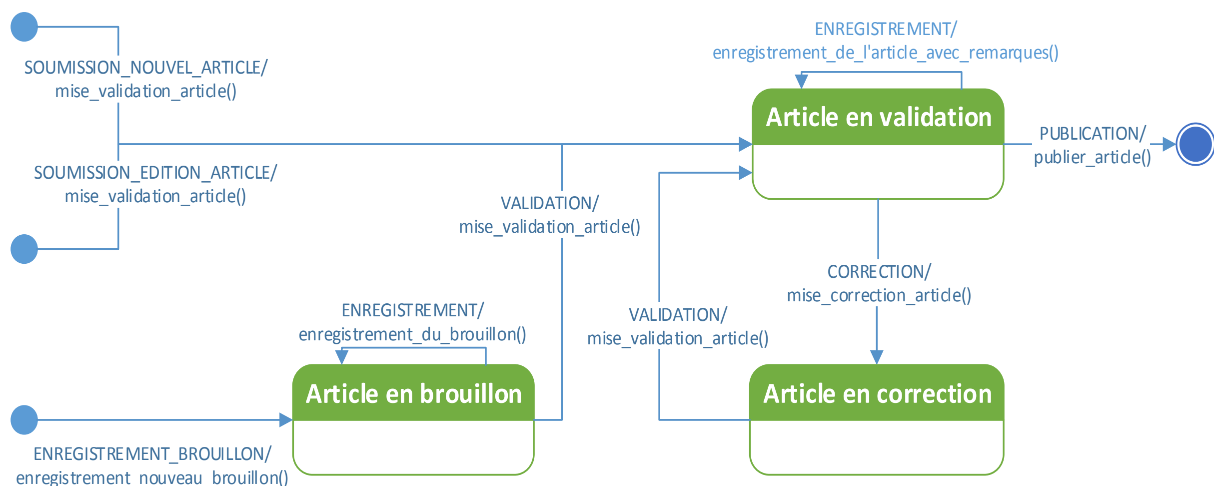

Figure 17 : Diagramme d'états d'un article

Il est aussi intéressant de noter que l'administrateur a d'autres choix au niveau de l'interface graphique : il peut enregistrer l'article avec les remarques pour le reprendre plus tard, le supprimer, le mettre en correction ou le publier sur le site. Nous ne parlerons pas des deux premiers cas, étant donné qu'ils sont très simples. Nous avons détaillé le troisième et nous parlerons du quatrième avec

la figure 18. Ces schémas ne représentent donc pas les choix qui s'offrent à l'administrateur, mais bien les mécaniques mises en place après qu'il ait fait sa sélection.

Un administrateur doit avoir la possibilité de publier un langage sur le site. Une fois que l'article lui paraît assez mature pour être en ligne, il doit avoir un moyen de l'afficher sur le site. Ce diagramme de séquence vise à expliquer ce mécanisme. Cependant, il convient de noter une particularité. Le mécanisme entier décrit était trop grand pour être sur une seule page. Il a donc été divisé en deux parties. La figure 18 représente la première moitié, tandis que la figure 19 la seconde moitié.

Lorsqu'un administrateur envoie un langage pour le publier sur le site via le formulaire, les informations sont transmises au contrôleur après le routing. Celui-ci vérifie que l'utilisateur est bien connecté et est un administrateur (dans un souci de place, la vérification que l'utilisateur est un administrateur n'est pas représentée contrairement à la figure 16). Si ce n'est pas le cas, la personne reçoit un code d'erreur. Sinon, le contrôleur envoie les informations au modèle. C'est ce dernier qui fera presque toutes les actions.

Le modèle va d'abord vérifier si le langage existe déjà sur le site pour pouvoir ajuster son comportement. En effet, si le langage existe, il doit reprendre son ID sinon, il doit en générer un. Ensuite, il va faire en parallèle trois boucles (figure 18). Dans le formulaire pour proposer un langage, il est demandé de renseigner les langages, paradigmes et auteurs liés. Le modèle doit donc s'assurer de savoir si pour chacun il s'agit d'une nouvelle entrée ou d'une entrée qui existe déjà sur le site. Dans la première boucle, il va interroger la base de données pour savoir si le langage lié au langage proposé est en ligne. Si oui, le modèle récupère l'ID, sinon, le modèle en génère un nouveau. Dans la seconde boucle ainsi que la troisième, le modèle va faire exactement la même chose, mais appliqué aux paradigmes et aux auteurs liés. Ceci est une phase de préparation étant donné que la base de données n'a pas été modifiée durant ces opérations. Maintenant que le modèle a récolté toutes les informations nécessaires, il va créer une transaction. Ainsi, la base de données reste dans un état consistant même si une des étapes qui suit venait à échouer. À l'origine, chaque action avec la base de données est une transaction. Mais on peut demander qu'une série d'actions forment une transaction. C'est ce qui est fait dans ce cas, car les futures opérations n'ont de sens que si elles réussissent ou échouent toutes. Donc la base de données entre dans un état spécial une fois que la transaction a commencé. Son activation en est alors bien plus longue que d'habitude.

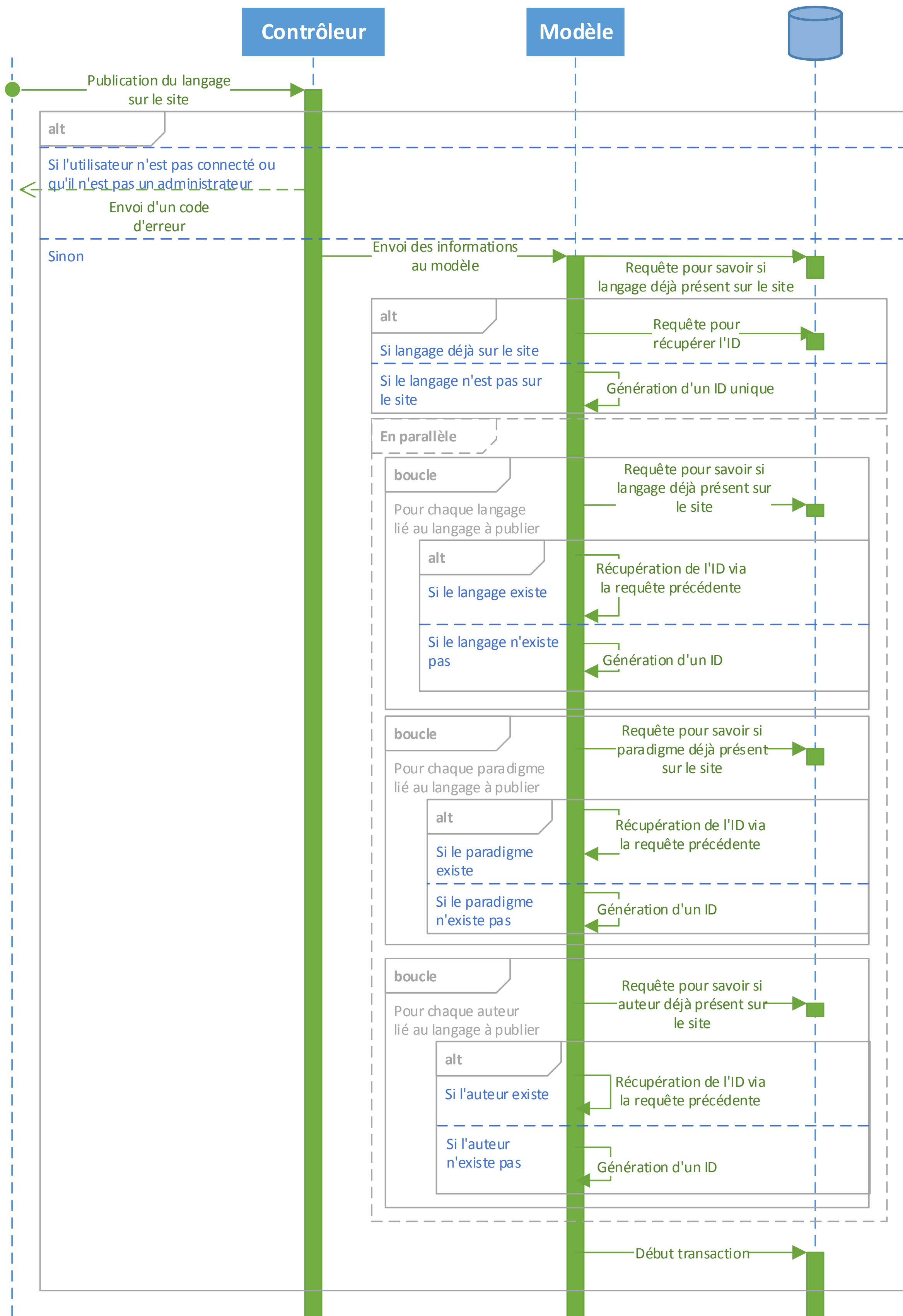

Figure 18 : Diagramme de séquence de la publication d'un langage (partie 1)

Le diagramme de la figure 19 reprend la fin du précédent. Il débute en ordonnant à la base de données de commencer la transaction. Ensuite, il y a un bloc « En parallèle » qui contient trois boucles ainsi que deux messages. Les boucles servent à rajouter les différents éléments manquants dans la base de données (langage, paradigme et auteur). Pour cela, le modèle va envoyer une requête SQL de type INSERT pour chaque élément en demandant à la base de données d'ignorer la requête si l'élément existe déjà (ce qui permet au final de rajouter uniquement les éléments manquants). Les deux messages servent à enlever le langage de la table *langage_en_creation* et à l'ajouter dans la table *langage*. Cette étape est indispensable pour pouvoir effectuer la mise à jour des tables de jointures. C'est pour cela que le modèle vérifie que toutes les opérations précédentes se sont bien déroulées. Si ce n'est pas le cas, il demande à la base de données d'effectuer un rollback et envoie une réponse au contrôleur pour le mettre au courant de l'échec de l'opération. Ce dernier transmet, dans ce cas, un code d'erreur à l'administrateur.

Dans la situation où tout s'est bien passé, le modèle effectue une mise à jour des tables de jointures. Dans un souci de clarté, il n'est représenté qu'un seul message avec « Mise à jour des tables de jointures ». En pratique, ceci est plus compliqué, car il s'agit en réalité d'une multitude de rajouts et de suppressions d'entrées en fonction des circonstances (nouveau langage ou mise à jour, par exemple). Il est intéressant de noter que seules les tables de jointures subissent des opérations de suppression et non les tables comme *langage*, *paradigme* ou *concept*. Concrètement, ce n'est pas parce qu'un langage n'est plus lié à d'autres que celui-ci n'existe plus. Ce qui explique pourquoi il n'y a pas de suppression dans ce diagramme de séquence. Cependant, l'administrateur a la possibilité de supprimer des entrées via le panel administrateurs.

Si un problème arrive au cours des mises à jour, une demande de rollback à la base de données est effectuée. Sinon, le modèle envoie un message pour terminer la transaction. À nouveau, il vérifie si elle a été acceptée ou non. Dans les situations où un problème serait apparu, le modèle informe le contrôleur que la requête a été refusée et ce dernier envoie un code d'erreur à l'administrateur. Autrement, le modèle avertit le contrôleur du succès de la requête et il envoie un code de réussite à l'administrateur. Finalement, l'administrateur est redirigé vers le panel.

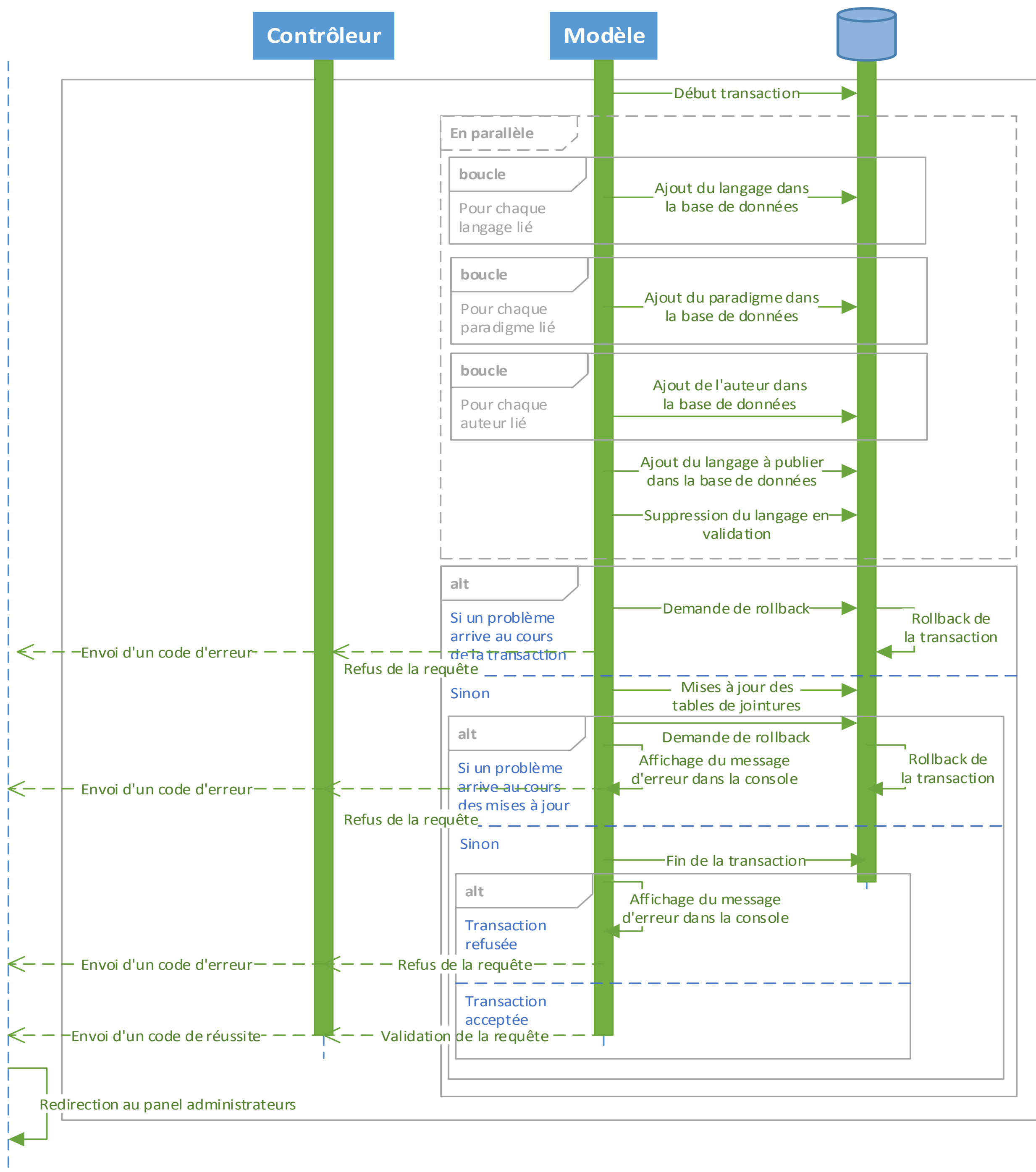

Figure 19 : Diagramme de séquence sur la publication d'un langage (partie 2)

Le système d'inscription du site est à deux niveaux : une première phase où l'utilisateur s'enregistre sur le site et une seconde phase où l'utilisateur valide son inscription via un lien envoyé par email. Ceci permet de pouvoir supprimer des entrées qui n'auraient pas été validées (si un utilisateur ne valide pas son inscription dans les 24 heures) et donc d'éviter de garder des données inutilement.

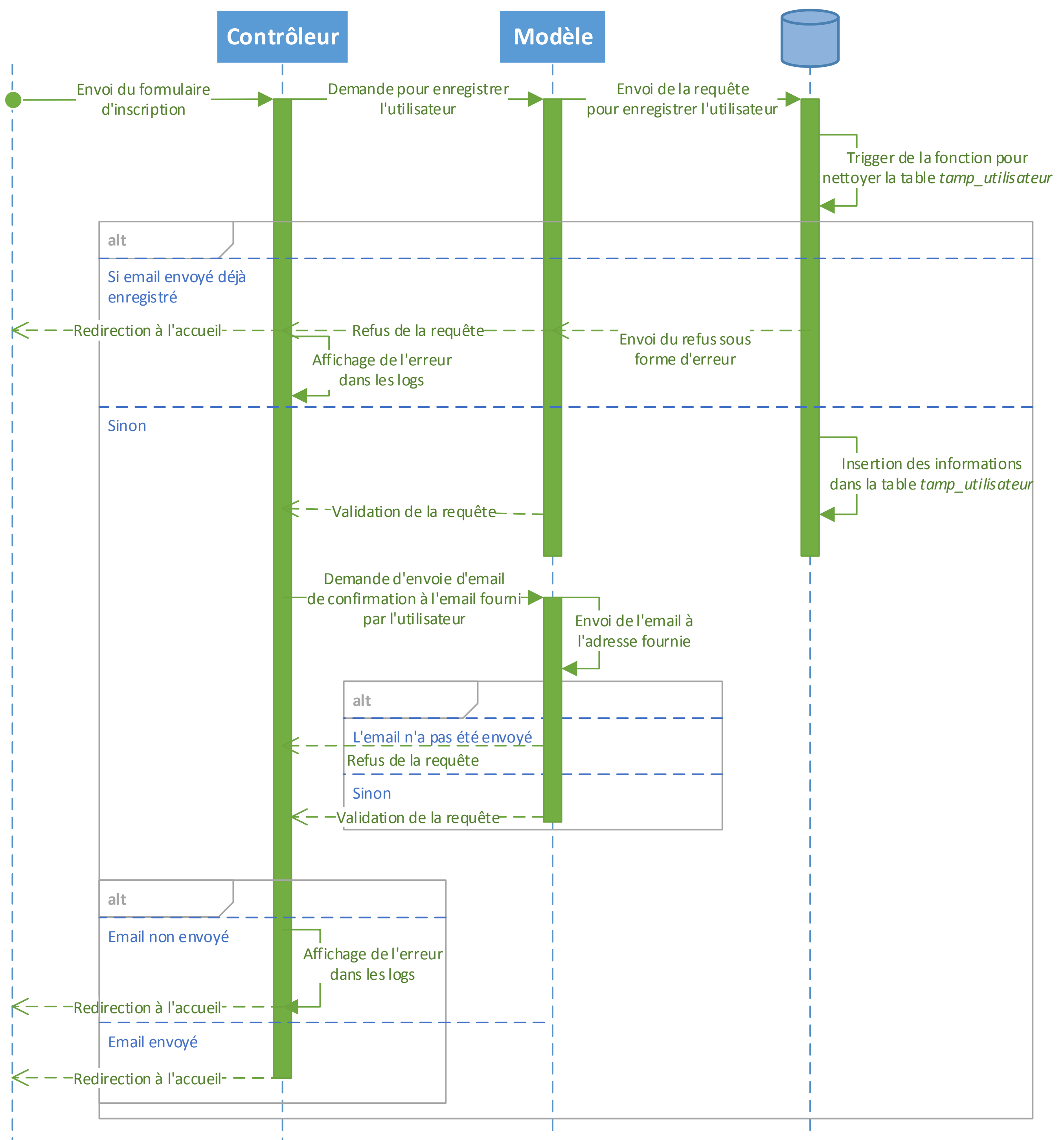

Figure 20 : Diagramme de séquence pour l'inscription

La figure 20 représente une demande d'inscription sur le site (première phase). Quand un utilisateur s'enregistre sur le site, via le formulaire, il envoie une requête HTTP au serveur. Après le routing, la requête est transmise au contrôleur. Ce dernier traite les informations contenues dans la requête et fait la demande au modèle pour enregistrer l'utilisateur dans la base de données. Le modèle va donc demander à la base de données d'enregistrer les informations dans la table *tamp_utilisateur*. Une tentative d'enregistrement enclenche une fonction de « trigger ». Il s'agit d'une fonction qui se

manifeste lors d'une certaine action (ici, l'insertion de données). Elle a pour but de supprimer toutes les entrées datant de plus de 24 heures. Une fois la fonction achevée, la base de données essaie d'enregistrer l'utilisateur et la réponse est envoyée au site. Si l'email était déjà enregistré, le modèle reçoit un refus de la base de données. Dans ce cas, il informe le contrôleur de l'échec de l'enregistrement. Ce dernier affiche l'erreur dans les logs et envoie une demande de redirection à l'accueil au client. Dans l'autre cas (succès de l'enregistrement), le modèle informe le contrôleur de la réussite de l'opération. Ensuite, ce dernier envoie une demande au modèle pour envoyer un email de validation à l'utilisateur. Le modèle prépare le mail grâce à la librairie « nodemailer »[24] et le signe en utilisant la norme d'authentification DKIM[25]. Le mail est ensuite transmis via Postfix. Le modèle informe ensuite le contrôleur de la réussite ou de l'échec de l'opération. En cas d'insuccès, le contrôleur affiche l'erreur dans les logs et envoie ensuite une demande de redirection. Dans la situation inverse, le contrôleur envoie simplement une demande de redirection vers l'accueil.

La seconde étape de l'inscription (la validation) se lance une fois que l'utilisateur clique sur le lien de validation présent dans le mail. Ensuite, le système déplace l'entrée correspondante à l'inscription de la table *tamp_utilisateur* à la table *utilisateur*. Aucun diagramme de séquence ne sera fourni pour cette étape. En effet, ne s'agissant que d'un simple déplacement de données, ce diagramme ne serait pas pertinent.

Ces divers diagrammes ont permis d'expliquer le fonctionnement d'une partie du site en montrant les différents mécanismes utilisés. Ainsi, nous avons également mis au jour les différentes relations entre modèle, vue, contrôleur et la base de données.

4.7. Exemple de détails d'implémentation

Dans cette section, nous discutons de deux détails d'implémentation : l'interface servant à communiquer avec la base de données et le système de pool. Ces deux exemples nous permettent de discuter de plusieurs points de l'application. L'interface de la base de données nous sert à montrer un exemple de choix de design et le système de « pool » est utile pour montrer un choix de management des ressources.

4.7.1. Interface de la base de données

La figure 21 présente l'interface utilisée par les contrôleurs. Le principal avantage de l'interface est de permettre une meilleure indépendance vis-à-vis de la base de données. En effet, le site utilise actuellement PostgreSQL. Mais les futurs développeurs pourraient opter pour une nouvelle base de données. Dès lors, tous les changements à effectuer seront concentrés au niveau des différents modules que l'interface charge. Les modules sont des fichiers JavaScript, qui contiennent le code logique, et sont chargés d'assurer les interactions avec la base de données. Par exemple, le module « db_langage » assure les interactions avec la base de données pour tout ce qui concerne les langages. Il est également intéressant de noter que les modules sont autonomes. Si un module a besoin de fonctions qu'il ne possède pas, il passe par

[24] https://nodemailer.com/
[25] DomainKeys Identified Mail

l'interface pour les appeler. Enfin, l'interface permet aux modules d'accéder aux ressources partagées, comme le « pool » de connexions avec la base de données.

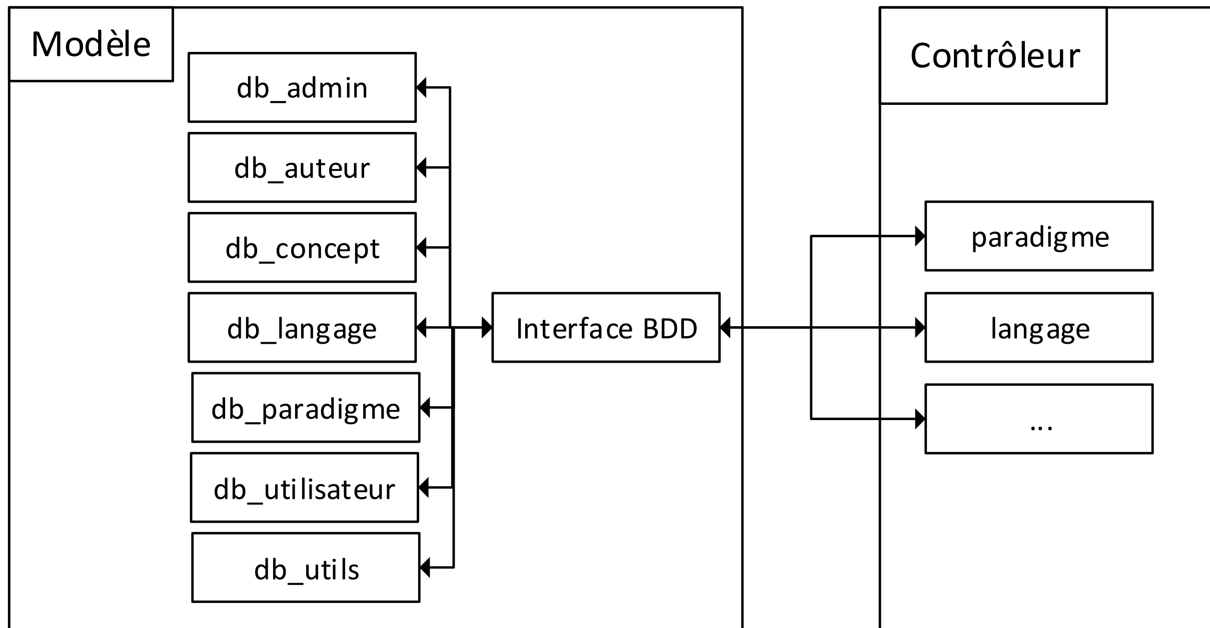

Figure 21 : Schéma expliquant l'interface de la base de données

4.7.2. Gestion des connexions avec le « pool »

Dans le code du projet, le « pool » permet de gérer le nombre de clients qui communiquent avec la base de données. Ainsi, ce système empêche la base de données d'avoir trop de connexions en parallèle. La figure 22 présente un exemple d'utilisation du « pool ». Le principe est le suivant : lorsqu'une fonction a besoin d'une ressource, celle-ci demande la ressource à le pool (appel à la méthode *pool.connect*) au lieu de la créer. Ensuite, lorsque le client n'est plus nécessaire, il est libéré (appel à la méthode *done*). En principe, cela permet de limiter le nombre de ressources créées et de pouvoir en réutiliser. La librairie « pg »[26] employée dans le projet utilise en plus un système de temps d'inactivité pour supprimer le client et ainsi libérer de la mémoire pour le système. Dans la configuration actuelle, si un client reste inactif plus de trois secondes, celui-ci est supprimé. Il est cependant important de remarquer qu'il est nécessaire de garder un même client pour une transaction. Il n'est donc pas possible de faire une transaction avec plusieurs clients pour accélérer le processus.

[26] https://github.com/brianc/node-postgres

```
const db = require('./../database');
const pool = db.pool;

function excuteQuery(query, valeurs) {
    return new Promise(function(resolve, reject) {
        pool.connect(function(err, client, done) {
            if (err) {
                console.log(err);
                return reject(err);
            }
            client.query(query, valeurs, function(err, result) {
                done();
                if (err) {
                    console.log(err);
                    return reject(err);
                }
                return resolve(result);
            });
        });
    });
}
```

Figure 22 : Exemple de code illustrant le principe de pool

4.8. Maintenabilité de l'application

Pour que l'application soit maintenable, il faut qu'il y ait des tests unitaires et de la documentation. Les tests permettent de vérifier si un changement dans le code a supprimé ou modifié un comportement dans l'application. La documentation, elle, aide les nouveaux développeurs à comprendre le fonctionnement des méthodes sans devoir lire l'implémentation.

Même si les tests unitaires aident à la maintenabilité de l'application, ces derniers sont analysés dans la section suivante qui concerne les tests de l'application.

Pour générer la documentation, l'outil JSDoc 3 a été utilisé (logiciel libre sous licence Apache) [24]. Il permet de créer des descriptions, par exemple, de méthodes ou de classes via un système de tags dans le code. La figure 23 fournit un exemple avec une méthode dont les paramètres ainsi que la valeur de retour sont décrits. De plus, le système de tags permet de décrire la méthode (publique, protégée ou privée) ainsi que de montrer un exemple qui se trouvera dans la documentation générée.

```
/**
 * Permet de savoir si une valeur existe dans une colonne d'une table
 * @protected
 * @param {string} table - Le nom de la table.
 * @param {string} colonne - Le nom de la colonne.
 * @param valeur - La valeur qui sera recherchée.
 * @returns {Promise} Renvoie une promise. Si réussite: le résulat est un
 * boolean. Si échec: le résultat est un string contenant le message d'erreur
 * @exports: db_composant/db_utils
 * @static
 * @example
 * //returns une promesse avec la valeur true si "maison" est dans la
 * //colonne "mot" de la table "dictionnaire"
 * existInBD("dictionnaire", "mot", "maisons)
 */
function existInDB(table, colonne, valeur) {
    return new Promise(function(resolve, reject) {
        let query = `SELECT count(*) FROM ${table} WHERE ${colonne} = $1`;
        pool.connect(function(err, client, done) {
            client.query(query, [valeur], function(err, result) {
                done();
                if (err) {
                    console.log(err);
                    return reject(err);
                }
                if (parseInt(result.rows[0].count) >= 1) {
                    return resolve(true);
                }
                return resolve(false);
            });
        });
    });
}
```

Figure 23 : Méthode existInDB documentée avec le système de tags de JSDoc

En installant JSDoc de façon globale via la commande « *npm install –g jsdoc* », puis en exécutant la commande suivante :« *jsdoc ./model -r -d docs* », JSDoc génère la documentation dans le dossier « docs » du projet pour l’ensemble des fichiers présents dans le dossier « model ». Elle est créée sous la forme d’une série de pages HTML pour être navigable avec un navigateur internet. La figure 24 montre la page générée en appliquant cette procédure.

Module: db_composant/db_utils

db_utils.

Source: db_utils.js, line 1

Methods

(protected, static) **existInDB(table, colonne, valeur)** → {Promise}

Permet de savoir si une valeur existe dans une colonne d'une table

Parameters:

Name	Type	Description
table	string	Le nom de la table.
colonne	string	Le nom de la colonne.
valeur		La valeur qui sera recherchée.

Source: db_utils.js, line 24

Returns:

Renvoie une promise. Si réussite: le résulat est un boolean. Si échec: le résultat est un string contenant le message d'erreur

Type
Promise

Example

```
//retourne une promesse avec la valeur true si "maison" est dans la
//colonne "mot" de la table "dictionnaire"
existInBD("dictionnaire", "mot", "maisons)
```

Figure 24 : Documentation générée pour la méthode existInDB avec JSDoc

JSDoc permet donc de générer une documentation précise (par exemple il est possible de donner le type et une description pour les arguments d'une fonction) de façon simple avec le système de tags.

Par manque de temps, il n'a pas été possible de créer la documentation pour tout le code du site. Nous avons donc décidé de restreindre la documentation seulement aux fonctions dites critiques. Les fonctions critiques sont les fonctions importantes dans le code. Dans ce projet, elles sont toutes situées dans la partie modèle ainsi que dans le fichier « *utils.js* ».

4.9. Tests de l'application

Les tests sont d'une importance capitale pour valider une application. Dans cette section, nous présentons donc les différents types de tests utilisés durant le développement de l'application ainsi que leurs résultats.

4.9.1. Les différents types de tests

Dans cette section, nous parlons des tests unitaires ainsi que de la couverture du code. Nous discutons des choix faits et de la configuration des outils.

Les tests unitaires permettent de vérifier le bon fonctionnement des fonctions présentes dans le code. Le fonctionnement est simple : le développeur est censé tester chaque fonction en les appelant et en comparant le résultat attendu avec celui obtenu. Il existe deux approches pour créer les tests unitaires : TDD[27] et BDD[28]. BDD diffère de TDD par son objectif de décrire le comportement de la fonction testée. En général, les tests suivant la méthodologie BDD sont donc plus clairs que ceux qui suivent la méthodologie TDD (selon l'auteur de la méthodologie BDD) [25]. Prenons deux exemples pour comparer les deux types de tests : « *assert(factoriel(3) == 6)* » et « *expect(factoriel(3)).to.be.eql(6)* » (les deux tests ont été écrits en utilisant la librairie « Chai » pour la gestion des « *assert* » et des « *expect* »)[29]. Le premier test correspond à la méthodologie TDD tandis que le second correspond à la méthodologie BDD. Le BDD se veut donc comme une évolution visant à combler certaines lacunes du TDD. Évidemment, ceci relève du côté subjectif : certains trouveront le TDD plus concis et donc plus simple que le BDD.

L'outil utilisé pour conduire les tests est « Mocha ». Cet outil est sous licence MIT, sa première version (en ne tenant pas compte des versions alpha) date du 22 novembre 2011 [26]. Il peut être installé avec la commande « *npm install –g mocha* ». Il suffit ensuite d'exécuter la commande « *mocha* » pour que celui-ci aille chercher les tests se trouvant dans le dossier « test ». La figure 25 présente un test minimal pour une fonction calculant la factorielle d'un entier positif. Il est intéressant de noter que Mocha offre la possibilité de créer les tests sans les compléter. Le troisième test n'est pas écrit et apparaîtra en « pending » lorsque la commande « *mocha* » sera effectuée. La figure 26, quant à elle, montre le résultat lorsque les tests sont exécutés.

[27] Test-driven development
[28] Behavior-driven development
[29] http://chaijs.com/

```javascript
const factoriel = require('../factorielle').factorielle;
const expect = require('chai').expect;

describe("Comportement de la fonction factorielle", function(){

    it('devrait renvoyer 1 pour factorielle de 0', function(done){
        expect(factoriel(0)).to.be.eql(1);
        done();
    });

    it('devrait renvoyer 6 pour factorielle de 3', function(done){
        expect(factoriel(3)).to.be.eql(6);
        done();
    });

    it('devrait lancer une erreur si argument négatif');

});
```

Figure 25 : Exemple de tests pour une fonction calculant une factorielle d'un entier positif

```
λ mocha

  Comportement de la fonction factorielle
    √ devrait renvoyer 1 pour factorielle de 0
    √ devrait renvoyer 6 pour factorielle de 3
    - devrait lancer une erreur si argument négatif


  2 passing (8ms)
  1 pending
```

Figure 26 : Exemple de tests unitaires conduits par Mocha

La couverture des tests est souvent effectuée en même temps que les tests unitaires. Il s'agit d'inspecter le code parcouru lors des tests. Ceci permet de déterminer le pourcentage du code testé selon une méthode choisie : couverture des fonctions, des instructions, des branches ou des lignes de code.

L'outil utilisé est Istanbul[30]. Il permet d'effectuer la couverture de code selon les quatre méthodes et de générer un rapport sous forme de fichier HTML. Pour installer l'outil, il faut utiliser la commande suivante : « *npm install –g istanbul* ». Pour exécuter l'analyse de couverture, il suffit de lancer la commande « *istanbul cover mocha* »[31]. La figure 27 montre le rapport HTML pour une fonction calculant la factorielle d'un nombre entier positif (même exemple que pour la figure 25).

[30] https://istanbul.js.org/

[31] Sur Windows, la commande sera « istanbul cover node_modules/mocha/bin/_mocha » après avoir installé mocha comme module du projet.

all files / example/ factorielle.js

100% Statements 6/6 **100%** Branches 0/0 **100%** Functions 1/1 **100%** Lines 6/6

```
1 1x function factorielle(n){
2 2x     let tmp = 1;
3 2x     for(let i = 1; i <= n ; i++){
4 3x         tmp *= i;
5        }
6 2x     return tmp;
7    }
8
9 1x exports.factorielle = factorielle;
```

Figure 27 : Exemple de couverture de tests par Istanbul

Il est important de noter qu'une couverture de 100% est en général impossible. En effet, une partie du code est souvent destinée à la gestion des erreurs. Il est donc logique que cette partie du programme ne soit pas explorée durant les tests. Istanbul propose une solution pour pallier ce problème en indiquant quel morceau de code il doit ignorer en utilisant des commentaires. Par exemple : pour ignorer un « if », il suffit d'insérer « */* istanbul ignore if */* » avant celui-ci. Ainsi, l'ensemble du « if » ne sera pas pris en compte lors de l'évaluation de la couverture.

Cependant, cette approche n'a pas été retenue. En effet, les commentaires dans un code ont pour but d'aider les développeurs à comprendre celui-ci. Or, ce n'est pas le cas avec les commentaires pour Istanbul. Il a donc été préféré de ne pas avoir une couverture de 100%, mais de conserver un code le plus compréhensible possible.

Les tests de performance comportent différentes catégories : tests de stress, tests de charge, tests de dégradation, etc. Dans le cadre du projet, l'objectif d'avoir des performances acceptables justifie l'utilisation d'un outil mesurant le temps de réponse des pages (tests de charge). L'outil utilisé est « ab » qui est un logiciel libre sous licence Apache[32]. Il permet de paramétrer le nombre de requêtes à effectuer, le nombre de connexions en parallèle, etc. La configuration de tests est la suivante : cent connexions en parallèle pour un total de dix mille requêtes. La charge est plus importante que le nombre d'utilisateurs décrit dans les besoins. Cependant, cela permet de s'assurer qu'il n'y aura aucun problème si le nombre d'élèves suivant le cours venait à augmenter.

Les tests d'intégration sont des tests qui visent à examiner l'ensemble du programme pour s'assurer que toutes les fonctionnalités sont opérationnelles. Contrairement aux tests unitaires qui sont utilisés pour s'assurer du bon fonctionnement d'un ensemble de fonctionnalités, les tests d'intégration servent à s'assurer du bon fonctionnement de l'ensemble du système. Ainsi, un module X pourrait parfaitement fonctionner selon

[32] https://httpd.apache.org/docs/2.4/programs/ab.html

des tests unitaires, mais une fois mis dans le système, le module Y ne fonctionne plus. Ce sont donc les tests d'intégration qui permettent d'alerter les programmeurs de ce problème.

Le projet a été développé sans test d'intégration continue. Dans ce type de test, tous les tests unitaires sont lancés, à chaque fois automatiquement, pour s'assurer que toutes les fonctionnalités (et donc tout le programme) fonctionnent. Or, nous n'avons utilisé aucun outil pour automatiser ceci. Un lancement manuel des tests avec la commande « *mocha* » était fait à chaque fois qu'une nouvelle fonction était implémentée.

Enfin, les tests d'acceptation visent à s'assurer que les besoins du client sont bien satisfaits. Ces tests consistent donc à demander au client de tester le prototype pour obtenir son avis. Ceci peut mener à des corrections ou des améliorations, voire à des changements de besoins. Ce dernier cas peut être le résultat du phénomène « IKIWISI[33] », car parfois le client comprend mieux ses besoins une fois un prototype testé [27]. Aucun outil n'a été utilisé pour encadrer les tests d'acceptation. Durant ces phases de test, le client envoyait ses remarques via email.

4.9.2. Résultats des tests

Dans cette partie, nous traitons des résultats des différents tests expliqués dans la section précédente. Dans un premier temps, nous décrivons la méthode utilisée, puis nous débattons des résultats et enfin, nous proposons une discussion des résultats.

Durant le développement du projet, chaque fois qu'une méthode (au niveau du Modèle dans l'architecture MVC) était entièrement implémentée, un test unitaire était écrit. Une fonction était considérée comme validée uniquement si tous les tests unitaires étaient réussis.

Les tests de performance ont montré des résultats bien au-dessus du minimum requis. En effet, le délai des réponses est inférieur à une seconde pour une charge supérieure à celle définie dans les besoins du client. La figure 28 montre un exemple de test employé pour mesurer le temps moyen de réponse (« Time per request (mean, accross all concurent requests) »). Cette mesure représente le temps moyen d'une réponse parmi les connexions faites en parallèle. Ceci simule les conditions si plusieurs utilisateurs naviguent sur la page en même temps. Ainsi, en se basant sur la figure, on peut dire qu'un utilisateur aura un délai de 87 millisecondes pour charger la page d'accueil lorsque cette même page est demandée par 99 autres connexions. Les autres tests sont en annexes. Seul le test concernant la page pour visualiser une proposition d'édition semble montrer un temps de réponse plus long. Ceci est provoqué par le calcul déterminant le texte qui a été ajouté ou enlevé entre la version sur le site et celle proposée.

[33] I know what I want when I see it

```
Server Software:        nginx
Server Hostname:        alexandrie.ovh
Server Port:            80

Document Path:          /graphique
Document Length:        7430 bytes

Concurrency Level:      100
Time taken for tests:   1048.312 seconds
Complete requests:      10000
Failed requests:        1
   (Connect: 1, Receive: 0, Length: 0, Exceptions: 0)
Total transferred:      77730000 bytes
HTML transferred:       74300000 bytes
Requests per second:    9.54 [#/sec] (mean)
Time per request:       10483.120 [ms] (mean)
Time per request:       104.831 [ms] (mean, across all concurrent requests)
Transfer rate:          72.41 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       44   79 375.2     49   15055
Processing:  4758 10339 8023.0   8550   74578
Waiting:      108 9109 5253.2   8479   70006
Total:       4807 10418 8074.3   8604   74638

Percentage of the requests served within a certain time (ms)
  50%   8604
  66%   8909
  75%   9091
  80%   9364
  90%  15298
  95%  23546
  98%  37216
  99%  55076
 100%  74638 (longest request)
```

Figure 28 : Résultats du benchmark sur la page générant le graphique

Il y a également eu deux tests d'acceptation : le premier a eu lieu mi-février et le second a été fait fin avril. Le projet a été déployé sur un serveur privé pour que le client (Monsieur Kim Mens) puisse le tester et ainsi profiter des retours de l'utilisateur.

Les tests unitaires couvrent 84,95 % (de lignes de code) des fichiers JavaScript constituant le modèle dans l'application. Comme annoncé et justifié dans la section 4.9.1, la couverture n'atteint pas 100%. Cependant, les tests couvrent bien tout le code qui n'est pas utilisé pour la gestion des erreurs.

Le test d'acceptation mené en février a permis de découvrir des problèmes de design. Notamment la nécessité d'avoir des zones permettant d'écrire du texte pour compléter certains champs qui étaient trop laconiques. C'est aussi à ce moment que la demande d'avoir une visualisation graphique des liens a été demandée. Le second qui a été mené en mai a permis de mettre en évidence plusieurs améliorations futures possibles pour le projet.

5. Validation

Dans cette section, nous discutons des besoins du client satisfaits par l'application. Nous allons les reprendre et, à chaque fois, justifier comment l'application arrive à les combler. Nous analysons d'abord les objectifs fonctionnels qui ont été clairement explicités par le client ainsi que les objectifs non fonctionnels formellement demandés. Ensuite, nous examinons les besoins non formulés par le client, mais qui étaient attendus. Enfin, nous terminons avec un objectif non demandé et non attendu qui a été rempli.

5.1. Objectifs fonctionnels

Dans leur livre « Applied software project management », Stellman et Green définissent les objectifs fonctionnels comme ce qui décrit les mécaniques internes du logiciel, par exemple la manipulation de données [28, p. 113]. Les points suivants sont les objectifs fonctionnels émis par le client :

5.1.1. Être en ligne

Pour que l'outil soit considéré en ligne, il doit être accessible via un navigateur. Il faut donc que le serveur soit configuré pour que les différents éléments composant l'architecture du système (voir figure 11) soient initialisés. Cet objectif a été atteint, les tests d'acceptation ayant pu être menés à terme.

5.1.2. Être participatif

L'outil permet de s'inscrire sur le site et de pouvoir suggérer des articles pour les différentes catégories. Le membre a le choix de sauvegarder son travail en cours de rédaction pour le reprendre plus tard. Ceci lui permet de travailler petit à petit sur article, ce qui est une facilité pour ceux qui disposent de peu de temps. Cet objectif a été atteint, les tests d'acceptation n'ayant relevé aucun problème de ce côté.

5.1.3. Avoir un système de soumission

L'outil permet à un administrateur de valider un article écrit par un utilisateur. Si le texte ne lui convient pas, il a alors la possibilité de demander à l'auteur de changer l'article en fonction des remarques qu'il aura formulées. Il peut se produire autant de demandes de corrections que nécessaire, permettant ainsi à l'auteur de fournir un article qui aura été amélioré de façon incrémentale. Cet objectif a été atteint, les tests d'acceptation n'ayant relevé aucun problème de ce côté.

5.1.4. Permettre l'édition d'articles

L'outil permet d'éditer des articles sur le site pour les mettre à jour ou corriger des erreurs qui auraient échappé à la vigilance de l'administrateur. L'édition d'un article n'est permise que si la personne est connectée au site. Dès lors, un bouton apparaît sur les articles pour accéder à un formulaire d'édition (déjà rempli avec les informations de l'article). Pour éviter des conflits avec les éditions, il n'est pas possible de proposer deux éditions simultanément sur un même texte. Ainsi, si une nouvelle version de l'article a été suggérée, le bouton d'édition de l'article se désactive et

indique qu'une édition est déjà en cours. Cet objectif a été atteint, les tests d'acceptation n'ayant relevé aucun problème.

5.1.5. Être muni d'un outil visuel pour les liens entre les entités

L'outil permet de voir les différents liens entre langages, paradigmes et concepts. Un exemple de graphique est visible sur la figure 1. Pour éviter que le graphe ait des nœuds isolés, les vues de la base de données contiennent des conditions qui servent à filtrer les résultats. En effet, un nœud isolé ne montre aucun lien avec les autres nœuds, ce qui alourdit le graphe inutilement. Le graphique met les nœuds en différentes couleurs en fonction de ce qu'ils représentent. Ainsi il est possible de distinguer rapidement et facilement les différentes entités. Cet objectif est considéré comme atteint, étant donné que les tests d'acceptation n'ont révélé aucun problème à ce sujet.

5.2. Objectifs non fonctionnels

Stellman et Green définissent les objectifs non fonctionnels comme les objectifs qui imposent des contraintes sur le design ou l'implémentation [28, p. 113]. Les points suivants sont les objectifs non fonctionnels énoncés par le client :

5.2.1. Être codé simplement

Le site web utilise le moins de technologies différentes possible. NodeJS, ExpressJS et EJS constituent l'ensemble de l'application. Tous les trois utilisent JavaScript, ce qui permet de gérer le front-end (l'interface HTML) et le back-end (l'application) avec un même et unique langage.

L'ensemble du système utilise Debian 8, Iptable, Nginx et PostgreSQL. Les deux premiers participent à la plupart des configurations des serveurs web. Debian est réputé pour être une des versions de Linux les plus stables. Iptable permet de créer un pare-feu avec des règles dont la syntaxe est facilement compréhensible. Nginx permet de lancer un serveur simplement. Enfin PostgreSQL, comme base de données, permet d'utiliser le langage SQL qui est standardisé. Comme expliqué dans la section 4.1, les bases de données de type SQL sont plus facilement accessibles pour un étudiant que les bases de données NoSQL.

Nous venons de montrer qu'il y a eu un réel effort pour garder l'ensemble du système le plus abordable possible en utilisant le moins de technologies différentes, les plus standardisées et les plus stables.

5.2.2. Performances acceptables

L'outil arrive à répondre dans un délai inférieur à trois secondes pour les requêtes HTTP. Nous avons établi dans la section 4.1 que le délai maximum tolérable pour le chargement d'une page était d'environ trois secondes. Ceci a été possible facilement grâce au paradigme de programmation asynchrone (cf. la section 2.6 sur la

programmation asynchrone pour l'explication sur les performances). Grâce aux tests de performance, il nous est possible de démontrer que cet objectif a été atteint.

5.2.3. Maniabilité

Il n'est pas possible de prouver que l'outil est facile à prendre en main pour toutes les personnes qui l'utiliseront car c'est un avis subjectif. Cependant, nous allons essayer de montrer que nous avons pris ce point en considération durant tout le développement de l'application. Par exemple, le menu du profil a été coupé en section pour qu'il soit mieux organisé.

L'outil a connu deux tests d'acceptation. Le premier a permis de mettre en évidence des erreurs de design et de les corriger. Par exemple, le champ date (dont la structure était jour/mois/année) ne convenait pas. À quoi correspondait-il ? Est-ce que la date devait être celle de la première publication ou celle de son prototype ? Finalement, cette partie du formulaire a été revue en permettant à l'utilisateur de commenter les dates qu'il a encodées dans un champ dédié aux explications de celles-ci. Lors du second test, il n'y a eu aucun signalement concernant cette partie. C'est ainsi que les problèmes d'ergonomie ont été en partie réglés (le reste ayant été corrigé au fur et à mesure du développement). Le deuxième test a aussi montré d'autres problèmes d'ergonomie. Cette fois-ci, seuls ceux dont la solution pouvait être implémentée, dans le délai du mémoire, ont été résolus.

De plus, il ne serait pas étonnant d'avoir de nouveaux problèmes qui émergeraient d'un troisième test d'acceptation, voir même d'une phase de test public. Cependant, cela ne signifie pas que le site n'est pas facile à prendre en main, mais que son ergonomie peut être améliorée. Durant le deuxième test, il n'y a pas eu de remarques ou questions concernant le fonctionnement du site. Ce qui permet de montrer que son ergonomie est suffisamment évoluée pour une utilisation publique.

5.2.4. Maintenabilité

Comme pour le point précédent, il n'est pas possible de prouver que l'application est maintenable, car ceci dépend de l'avis de chacun. Néanmoins, il est envisageable de prouver que des efforts ont été faits pour aller dans ce sens.

Tout d'abord, la documentation est fournie pour chaque méthode publique, dans la partie modèle de l'application. Ceci signifie que si un développeur désire savoir le but d'une méthode ou les données qu'elle a pour arguments, il lui sera possible de lire la documentation et ainsi ne pas devoir comprendre l'implémentation. Pour un développeur, il s'agit d'un énorme gain d'énergie et de temps.

Ensuite, l'application comporte des tests unitaires. Ces derniers permettent de tester les fonctionnalités comme signalé précédemment. Dans le cadre de la maintenance d'une application, ils peuvent être utilisés pour rapidement tester si une modification a rendu une ou plusieurs fonctions invalides. De plus, ces tests sont automatisés via une commande pour qu'ils puissent être tous lancés en même temps. Ainsi le futur

développeur pourra tester au plus vite si une modification ou de nouvelles fonctions ne posent aucun problème.

Enfin, le caractère modulable de la solution permet de réduire le nombre de changements nécessaires lors d'une modification du code. Nous avons présenté l'architecture MVC de l'application dans la section 244.5 qui permet de séparer les composants selon leur rôle. À ceci s'ajoute l'interface du modèle (décrite dans la section 4.7.1) pour dialoguer avec la base de données.

5.3. Objectifs implicitement demandés par le client

Certains objectifs ne sont pas demandés explicitement, mais sous-entendus par convention. Par exemple, il n'a pas été formulé que la connexion au serveur soit sécurisée (ici, nous parlons de nous connecter en SSH et non d'accéder au serveur par le port 80 avec son navigateur). Il n'a pas été non plus demandé que le système soit robuste (par exemple : si le serveur redémarre, l'application doit faire de même). Pourtant, ce sont des objectifs attendus et qui doivent être remplis un minimum pour être considérés comme validés. Nous allons donc définir pour chaque point le minimum à accomplir pour qu'il soit accepté.

5.3.1. Sécurisé

Pour que le système soit considéré comme sécurisé, il faut que le serveur ne soit accessible que pour les personnes autorisées, que les différents composants ne soient pas exposés inutilement (au risque de créer une brèche dans le système) et que les fonctions d'administrateurs ne soient pas utilisables par une personne qui essaierait d'usurper l'identité de quelqu'un d'autre.

Le premier point est rempli facilement, car les systèmes d'exploitation mis sur les serveurs d'hébergeurs demandent toujours un mot de passe (fourni par email, généralement) pour se connecter via SSH. À noter qu'il est possible d'augmenter la sécurité en remplaçant la connexion avec mot de passe par une connexion avec clé publique et privée. Cette option est intéressante si une seule personne a l'autorisation de se connecter au serveur. Actuellement, le serveur n'autorise pas cette connexion, mais l'outil de déploiement (dont nous parlons dans la section sur les objectifs supplémentaires atteints) permet d'installer cette option.

Le second point est rempli avec l'outil Iptable. En effet, ne pas exposer les composants du serveur inutilement revient à mettre en place un système de pare-feu. Il a été configuré de telle façon que toutes les requêtes sont abandonnées (la réponse n'est pas envoyée) si elles ne correspondent pas à une des règles établies (par exemple, utiliser le protocole TCP et demander une connexion via le port 80, ce qui équivaut aux requêtes d'un navigateur internet).

Le dernier point est validé, car le système de sessions empêche de forger une identité (usurper l'identité de l'administrateur). En effet, le cookie envoyé au client est composé d'un identifiant de session et de la signature faite par le serveur. La signature empêche de créer une session, car le serveur pourra vérifier l'authenticité avec la

signature : en comparant la signature reçue avec celle calculée. Il devient possible de voir si la session a été initialisée par le serveur ou non. Comme à chaque accès d'une fonctionnalité réservée à l'administrateur, le système vérifie si la requête émane bien d'un administrateur via le système de sessions. Ainsi il n'y a aucun risque d'utilisation de cette fonctionnalité par un utilisateur qui n'est pas un administrateur.

Le système utilise le middleware Helmet[34] qui permet de modifier les en-têtes HTTP pour une meilleure sécurité. Par exemple, sans lui, ExpressJS envoie un en-tête nommé « X-Powered-By » avec la valeur « Express ». Helmet permet d'enlever cet en-tête, ce qui empêche de donner des informations sur le système à une personne potentiellement malintentionnée. Un autre exemple est l'en-tête « X-Content-Type-Options » avec la valeur « nosniff » qui est ajouté à chaque réponse du serveur. Cet en-tête permet d'éviter le « sniffing » des données envoyées par le serveur[35]. Ce middleware permet donc d'augmenter le niveau de sécurité du site.

Enfin deux autres protections ont été implémentées. La première empêche les attaques XSS[36] en filtrant les entrées des formulaires. La solution mise en place consiste à retirer les balises « *<script>* » et « *</script>* » du texte. Ainsi, il est impossible de mettre un script JavaScript dans les données du site. La deuxième sécurité consiste à paramétrer les requêtes SQL pour éviter les injections SQL. La librairie utilisée pour communiquer avec la base de données fournit ce genre de requêtes[37]. Ces deux mécanismes renforcent encore le niveau de sécurité du site.

5.3.2. Robuste

Pour que le système soit considéré comme robuste, il fallait qu'il puisse redémarrer automatiquement en cas de panne de courant. Normalement, un serveur se relance tout seul dans la plupart des cas. Cependant, il faut encore exécuter le programme servant d'application. Pour cela, deux scripts sont exécutés au démarrage : le premier sert à remettre les règles du pare-feu et le second lance l'application en NodeJS. Ainsi l'objectif de la robustesse est rempli.

Nous avons réussi à aller plus loin qu'un simple redémarrage automatique : le système utilise PM2 (Process Manager 2)[38] pour gérer les arrêts de l'application. En effet, lorsqu'il remarque qu'un processus qu'il a démarré s'est interrompu, il le relance. Ce scénario est, par exemple, possible si une personne tente une attaque sur le site. Il se peut que l'attaquant arrive à faire arrêter le processus NodeJS et donc l'application. Dans ce cas, PM2 relancera l'application sans aucune intervention humaine. Ainsi, le système est plus robuste face à un problème technique.

---

[34] https://github.com/helmetjs/helmet
[35] https://developer.mozilla.org/fr/docs/Web/HTTP/Headers/X-Content-Type-Options
[36] cross-site scripting
[37] https://github.com/brianc/node-postgres/wiki/Parameterized-queries-and-Prepared-Statements#parameterized-queries
[38] http://pm2.keymetrics.io/

Dans le développement de l'application, l'utilisation de fonctions qui n'échouent jamais a permis d'augmenter la robustesse du site. En effet, certaines fonctions ont été créées de sorte qu'elles gèrent les exceptions qui pourraient être produites. Ces fonctions sont non-critiques d'un point de vue fonctionnel, mais sont utilisées largement sur le site. Par exemple, la fonction qui permet d'avoir les statistiques des articles sur la page de notre profil est utilisée sur plus de la moitié du site. Cependant, elle n'est pas essentielle aux fonctionnalités. Imaginons que les statistiques soient indisponibles (problème au niveau du disque dur, par exemple), un utilisateur doit pouvoir accéder à son profil malgré ce problème technique. Certes, son expérience sera détériorée dans ce cas, mais elle sera moins dégradée que si son profil n'était plus du tout accessible. Il est important de noter que cela ne dissimule nullement le problème. En effet, dans notre exemple, en cas de problème, les statistiques affichées sont remplacées par des « ? ». Ainsi, il y a bien un retour visuel qu'un problème est survenu et l'utilisateur peut avertir l'administrateur de ce souci.

Enfin, le système est capable de gérer l'indisponibilité des fichiers sur les CDN[39]. Ce cas peut sembler peu probable, mais il est réaliste. Certains fichiers de style et JavaScript sont vitaux pour le site. Si nous prenons les fichiers pour Bootstrap, sans eux, nous ne pouvons rien afficher. Pour pallier ce problème potentiel, les fichiers manquants sont chargés via le serveur en cas d'échec de chargements. Ensuite, ces fichiers seront mis en cache par Nginx pour soulager l'application NodeJS.

5.4. Objectif supplémentaire atteint

Dans les objectifs réalisés, un objectif n'était ni mentionné explicitement ni implicitement dans les besoins du client. Il convient de le mentionner, car cet objectif rentre dans la catégorie des objectifs non fonctionnels décrits dans le standard ISO 9126[40] qui a été remplacé par le standard ISO 25010[41]. La facilité d'installation est reprise dans le critère de portabilité de ce dernier.

Pour faciliter l'installation, il a été choisi d'utiliser un automate de déploiement. Comme leur nom l'indique, ces programmes permettent d'automatiser toutes les tâches nécessaires pour l'installation d'un site web (ou autre) sur un serveur. Le choix s'est porté sur Ansible[42] parmi d'autres choix comme Chef[43] ou Puppet[44]. L'avantage d'Ansible par rapport aux autres est sa facilité syntaxique qui ne demande pas de connaître un langage particulier. Il utilise la syntaxe YAML[45] contrairement aux deux autres mentionnés qui utilisent le langage Ruby. Ansible utilise des recettes qui sont composées d'un ensemble de tâches. La figure 29 montre un exemple de recette.

---

[39] content delivery network
[40] https://www.iso.org/fr/standard/22749.html
[41] https://www.iso.org/fr/standard/35733.html
[42] https://www.ansible.com/
[43] https://www.chef.io/
[44] https://puppet.com/fr
[45] Yet Another Markup Language

```yaml
---
- name: installation postfix
  become: yes
  apt:
    name: postfix
    state: present

- name: edition configuration postfix
  become: yes
  lineinfile:
    dest: /etc/postfix/main.cf
    regexp: "myhostname = UNKNOWN"
    line: "myhostname = alexandrie.ovh"

- name: restart postfix
  become: yes
  shell: "service postfix restart"
...
```

Figure 29 : Recette Ansible pour l'installation de Postfix

Concrètement, l'installation du serveur se fera de la façon suivante : il faudra exécuter la commande « *sudo ansible-playbook playbook.yml -i host –k* »[46], puis rentrer le mot de passe du serveur. Le temps d'installation de tous les composants du projet varie en fonction de plusieurs paramètres (CPU, RAM, bande passante, etc.). Ansible va donc faire les opérations suivantes automatiquement : installation du pare-feu, de la base de données, de Postfix, de NodeJS, de Nginx et de l'application. Une fois cet ensemble de tâches terminé, le site web est entièrement opérationnel. Ainsi, la phase d'installation du serveur (qui est généralement longue et fastidieuse) se voit entièrement automatisée.

[46] Nous supposons que la commande ansible-playbook a été installée auparavant.

6. Travaux futurs

À l'instar de tout projet, la solution peut être améliorée. Il est envisageable d'apporter des fonctionnalités supplémentaires, d'améliorer l'environnement de développement ou certaines fonctionnalités. Ceci ne signifie pas que le projet n'a pas atteint ses objectifs, bien au contraire (cf. section 5 sur la validation des objectifs). Cependant, proposer des pistes pour parfaire le système permet d'aider les futurs développeurs à améliorer celui-ci.

6.1. Fonctionnalités supplémentaires

Le dernier test d'acceptation a suggéré une fonctionnalité qui serait une amélioration intéressante : pouvoir rajouter des champs libres au formulaire. Actuellement, celui-ci dispose d'une section appelée « autre » qui peut contenir toutes les informations qui n'auraient pas lieu d'être dans les autres champs. Cependant, pouvoir ajouter des champs permettrait de mieux mettre en évidence les différentes sections d'un texte. Ainsi, il serait possible d'ajouter une section portant sur la sécurité ou sur la gestion de la concurrence pour certains langages. Toutefois, un changement de cet ordre amène à réexaminer la question du type de base de données à adopter. En effet, étant donné que le formulaire aurait un contenu de base qui pourrait être augmenté par des champs libres ajoutés, la structure des données n'est plus clairement définie. Dès lors, une base de données NoSQL, qui est plus flexible concernant la forme des données, pourrait être un choix judicieux.

Une autre fonctionnalité, qui pourrait être ajoutée au site, est un système de validation à plusieurs administrateurs, à l'instar d'un jury qui validerait un travail. Ceci permettrait d'apporter différents points de vue au travail d'un étudiant. Cette approche autoriserait aussi d'ajouter l'avis d'un expert d'un domaine précis à l'évaluation de l'article. Par exemple, si dans le document il est fait mention de la dimension sécurité d'un langage, il serait intéressant d'avoir l'avis d'un expert à ce sujet. Cette fonctionnalité demanderait de revoir le cycle de vie d'un article. Il faudrait voir comment seraient faites les remarques (un membre d'un jury peut-il voir les remarques des autres ?) ainsi que le système de validation (tous les membres du jury doivent valider l'article pour qu'il soit publié ou faut-il une majorité ?). De plus, il faudrait décider du nombre de membres pour valider un article et déterminer si ce nombre peut changer durant le cycle de vie de l'article.

Il serait aussi intéressant d'étudier la possibilité de rajouter l'opportunité de lier un langage avec un ou plusieurs concepts. Actuellement, un langage peut être lié à d'autres langages ou paradigmes, mais pas directement à un concept. Même si ceci n'était pas défini dans les besoins initiaux, il est envisageable de voir un concept lié uniquement à un langage. Par exemple, Java a introduit le concept d'interface. Même si depuis les interfaces se sont manifestées dans d'autres langages de programmation (ADA, C#, Objectif-C, PHP, etc.), on peut dire qu'à l'époque ce concept était bien lié à Java. Pour cette raison, il serait intéressant de rajouter cette option au site.

Une dernière fonctionnalité supplémentaire abordée dans cette section concerne l'upload de fichiers. Il serait intéressant de pouvoir envoyer les fichiers utilisés dans l'article. Actuellement, il est possible d'y faire référence dans le texte sous forme de lien. Mais si le fichier venait à être supprimé, il ne serait plus possible pour les lecteurs d'accéder à cette ressource. Ceci serait aussi un gain de temps pour les étudiants qui désireraient voir les documents utilisés (ils ne devraient plus chercher la source sur le net, vu qu'elle serait jointe à l'article). Cette fonctionnalité amène des questions d'ordre de sécurité, puisque le site va pouvoir distribuer des fichiers. Il faudra donc s'assurer que les fichiers

soient sans risque pour les utilisateurs. L'utilisation de service d'analyse comme VirusTotal[47] pourrait être une option à envisager. Ce type de service propose d'évaluer les fichiers avec un indice de méfiance. Plus il est élevé, plus le fichier est à risque. Pour obtenir cet indice, VirusTotal scanne le fichier avec plusieurs antivirus. Cependant, d'autres approches pourraient être envisagées (utilisation d'un unique scan antivirus, par exemple).

6.2. Amélioration du cycle de développement

Comme notifié précédemment, des tests d'intégration continue n'ont pas été réalisés durant le projet. Pour une équipe de développement réduite à une personne, il n'est pas nécessaire d'automatiser l'exécution de tous les tests unitaires. Cependant, si cette équipe venait à grandir, il serait conseillé de mettre en place un tel système. Jenkins[48], Travis[49] ou Coveralls[50] sont différents exemples possibles. Le principe est de connecter le dépôt où est stocké le code et d'exécuter tous les tests unitaires dès qu'un changement (un push) est effectué. De plus, ceci permet d'éviter d'occuper la machine du développeur pour effectuer les tests (qui peuvent prendre du temps si la taille du projet devient plus conséquente).

Il serait possible également de mettre en place d'autres tests comme les tests end-to-end. Ces tests évaluent si le « flot » de l'application correspond au comportement attendu. Par exemple un utilisateur non connecté au site, s'il clique sur le menu, puis sur « connexion », rentre les bons identifiants dans le formulaire, puis clique sur le bouton valider, il devrait voir un lien lui permettant de suggérer un article. Ceci est le flot qui consiste en une série d'actions qui peuvent transmettre des informations entre elles. Pour tester ceci, on utilisera un navigateur piloté automatiquement. Un exemple connu est Selenium[51] : il donne la possibilité d'automatiser une série de tests pour vérifier plusieurs comportements de l'application. Ces tests permettent de mettre en évidence des problèmes liés à l'interface que les tests unitaires ne couvrent pas forcément. Un bouton manquant, ne faisant pas partie de la logique de l'application, ne sera pas détecté par un test unitaire, mais bien par un test end-to-end. Ainsi si la taille du projet devenait importante, il serait intéressant d'automatiser ces tests.

Enfin, il serait pertinent de procéder à un bêta test qui consiste à demander à des utilisateurs finaux de tester l'application et d'obtenir leur avis. En effet, les tests d'acceptation ont été menés avec le client uniquement. Il serait donc intéressant d'effectuer un bêta test avec des étudiants pour avoir plus d'opinions sur l'application. Ceci permettrait de potentiellement découvrir plus de soucis de design ou technique. Le bêta test autorise aussi de hiérarchiser l'ajout de certaines en suivant des demandes dans les retours des testeurs. Cette phase de test permettrait à l'application de passer en « release candidate ». En « release candidate », l'application serait considérée comme assez mature pour être délivrée au client : aucune nouvelle fonctionnalité ne devrait être ajoutée, seuls des bugs seraient corrigés. La phase de bêta test n'a pas pu être menée dans le cadre de ce mémoire par manque de temps. Ceci pourrait donc être envisagé par les développeurs qui reprendront le travail.

---

[47] https://www.virustotal.com/fr/
[48] https://jenkins.io/
[49] https://travis-ci.org/
[50] https://coveralls.io/
[51] http://www.seleniumhq.org/

7. Conclusion

Notre problématique se centrait sur la transmission de connaissances entre professeur et élève au début de ce mémoire. La concrétisation d'une solution pertinente a nécessité la définition d'objectifs précis à atteindre pour que le produit réponde bien au problème. Nous avons ensuite présenté l'accomplissement du travail effectué en discutant de l'architecture du serveur et de l'application. Par la suite, nous avons exploré les différents tests exploités ainsi que leurs résultats. Par après, nous avons démontré que l'application répondait aux exigences du client et qu'elle remplissait même des objectifs supplémentaires.

Nous avons montré que le produit est actuellement valide et qu'il pourrait être utilisé dans le cadre du cours « Programming paradigms : theory, practice and applications » (LSINF2335) donné par Monsieur Kim Mens. Bien qu'il soit utilisable, nous avons identifié différentes pistes d'amélioration que ce soit au niveau des fonctionnalités proposées qu'au niveau du cycle de développement.

Ce travail s'inscrit dans une démarche qui a pour but d'aider des étudiants dans leur apprentissage et nous espérons avoir apporté notre pierre à l'édifice. En aucun cas le projet ne peut remplacer un cours, mais il peut servir de complément. Nous espérons donc que ce projet, mené avec enthousiasme, suscitera l'envie à d'autres de le continuer.

Références

[1] S. Baxter, G. Haley, M. Renier, and I. Lecigne, *Les chroniques de la science-fiction: une histoire en images de toute la sf depuis ses origines*. Muttpop {é}ditions, 2015.

[2] P. Van-Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. Prentice-Hall, 2004.

[3] J. C. Mitchell, *Concepts in Programming Languages*. Cambridge University Press, 2002.

[4] A. Marc, “INNOVATORS OF THE NET: BRENDAN EICH AND JAVASCRIPT.” [Online]. Available: http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fcgi.netscape.com%2Fcolumns%2Ftechvision%2Finnovators_be.html. [Accessed: 28-Apr-2017].

[5] Ecmas. ECMA-Kommittee, “A general purpose, cross-platform programming langugae, Standard ECMA-262.” 1997.

[6] Mozilla, “ECMAScript,” 2017. [Online]. Available: https://developer.mozilla.org/fr/docs/Web/JavaScript/Language_Resources. [Accessed: 28-Apr-2017].

[7] Ryan Dahl, “node-v0.x-archive/AUTHORS,” 2010. [Online]. Available: https://github.com/nodejs/node-v0.x-archive/blob/cb32883537bc7c499d08d57d005a6261ddb3cfd3/AUTHORS. [Accessed: 28-Apr-2017].

[8] S. Tilkov and S. Vinoski, “Node.js: Using JavaScript to Build High-Performance Network Programs,” *IEEE Internet Comput.*, vol. 14, no. 6, pp. 80–83, Nov. 2010.

[9] NodeJS, “v0.6.3: 2011.11.25, Version 0.6.3 (stable),” 2011. [Online]. Available: https://github.com/nodejs/node/releases/tag/v0.6.3. [Accessed: 29-Apr-2017].

[10] N. Leavitt, “Will NoSQL databases live up to their promise?,” *Computer (Long. Beach. Calif).*, vol. 43, no. 2, 2010.

[11] “Première release de NodeJS.” [Online]. Available: https://github.com/nodejs/node/releases/tag/v0.0.1. [Accessed: 11-May-2017].

[12] Express, “Release Change Log,” 2017. [Online]. Available: https://expressjs.com/en/changelog/4x.html#4.15.2. [Accessed: 29-Apr-2017].

[13] Pugjs, “Github Pug.” [Online]. Available: https://github.com/pugjs/pug#rename-from-jade. [Accessed: 29-Apr-2017].

[14] E. International, “ECMAScript® 2015 Language Specification.” Geneva, p. 545, 2015.

[15] A. Nayak, A. Poriya, and D. Poojary, “Type of NOSQL databases and its comparison with relational databases,” *Int. J. Appl. Inf. Syst.*, vol. 5, no. 4, pp. 16–19, 2013.

[16] F. F.-H. Nah, “A study on tolerable waiting time: how long are web users willing to wait?,” *Behav. \& Inf. Technol.*, vol. 23, no. 3, pp. 153–163, 2004.

[17] R. Barker, *CASE Method: Entity Relationship Modelling*, no. vol.~1. Addison-Wesley, 1990.

[18] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Addison-Wesley, 2011.

[19] J. L. Hainaut, *Bases de données - 3e éd.: Concepts, utilisation et développement*. Dunod, 2015.

[20] B. Lambeau, “LINGI2172 – Databases Logical Database Design Part I – Normal Forms & Normalization.” p. 59.

[21] K. Decker, “Github jsdiff.” [Online]. Available: https://github.com/kpdecker/jsdiff. [Accessed: 05-May-2017].

[22] E. W. Myers, “An O (ND) difference algorithm and its variations,” *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[23] T. H. Cormen, *Introduction to Algorithms, 3rd Edition:* MIT Press, 2009.

[24] “JSDoc Github.” [Online]. Available: https://github.com/jsdoc3/jsdoc. [Accessed: 13-May-2017].

[25] D. North and others, “Introducing bdd,” *Better Software, March*, 2006.

[26] “Mocha release 0.0.1.” [Online]. Available: https://github.com/mochajs/mocha/releases/tag/0.0.1. [Accessed: 13-May-2017].

[27] B. Boehm, “Requirements that handle IKIWISI, COTS, and rapid change,” *Computer (Long. Beach. Calif).*, vol. 33, no. 7, pp. 99–102, 2000.

[28] A. Stellman and J. Greene, *Applied software project management*. “ O’Reilly Media, Inc.,” 2005.

Annexes

Les annexes sont divisées en deux parties. La première présente les images générées par SchemaSpy de toutes les tables de la base de données. Tandis que la seconde permet de voir les résultats des différents benchmarks (générés par l'outil « ab ») utilisés pour tester les performances de l'application.

A. Annexe tables de la base de données

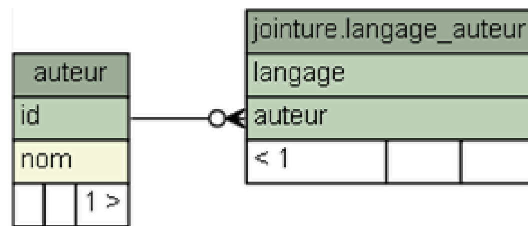

Figure 30 : Diagramme de toutes les tables présentes dans le schéma auteur

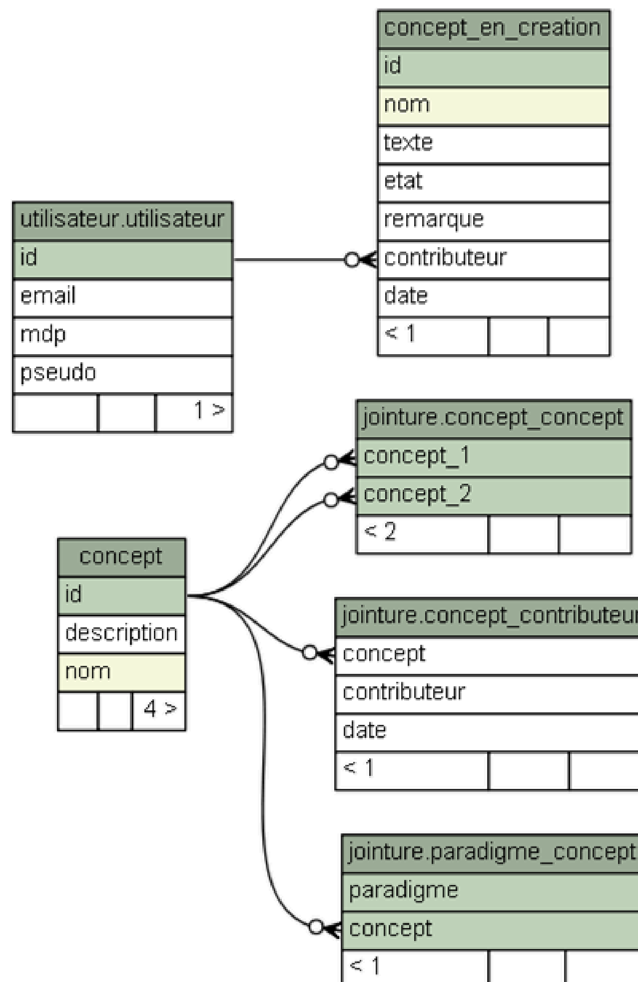

Figure 31 : Diagramme de toutes les tables présentes dans le schéma concept

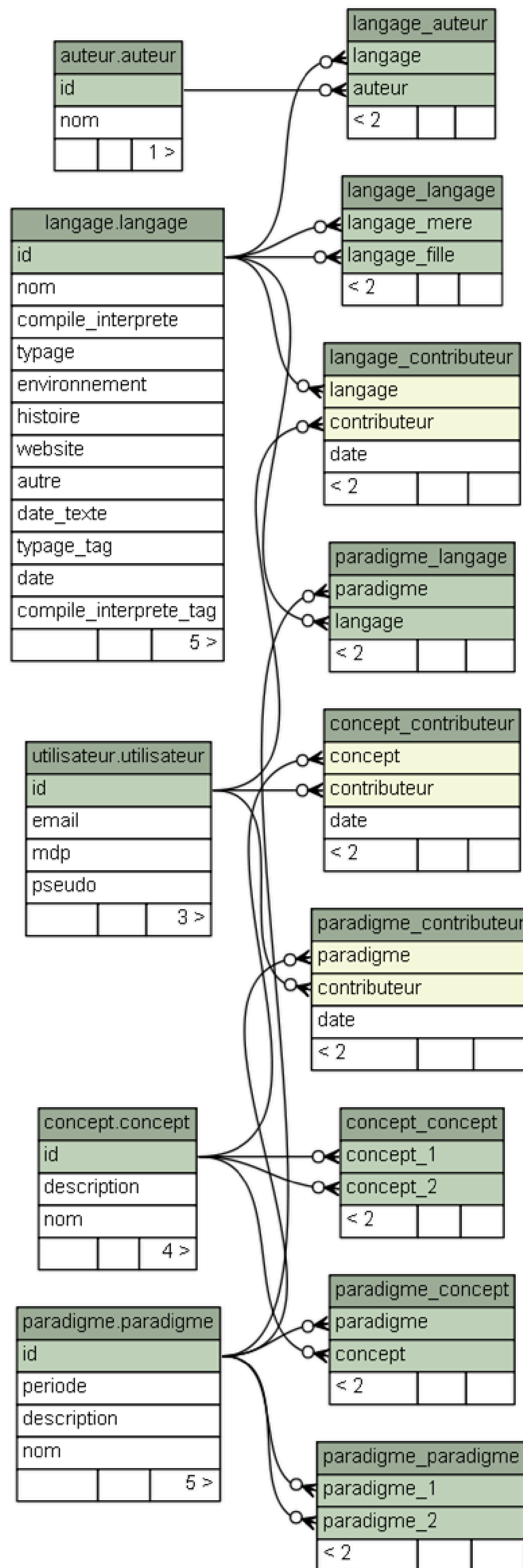

Figure 32 : Diagramme de toutes les tables présentes dans le schéma jointure

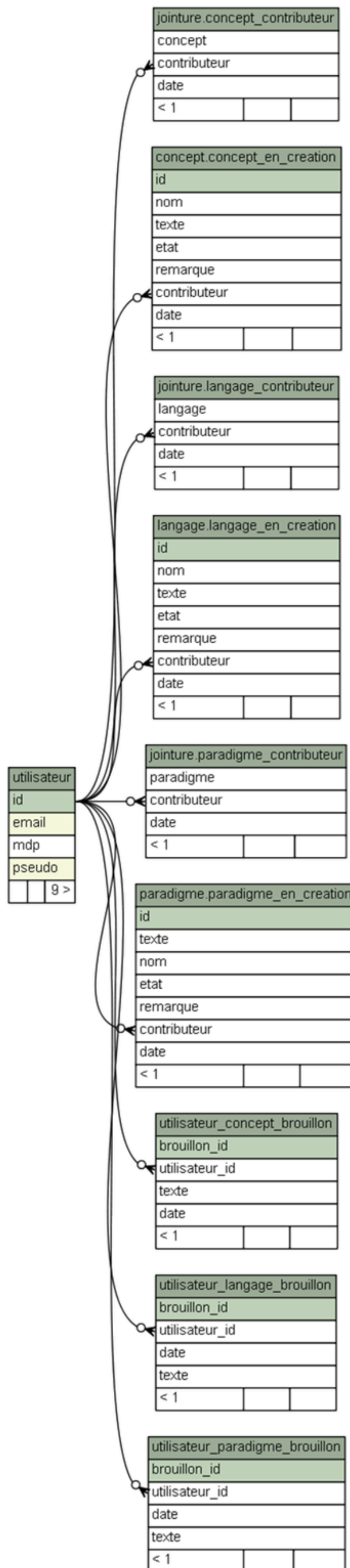

Figure 33 : Diagramme de toutes les tables présentes dans le schéma utilisateur

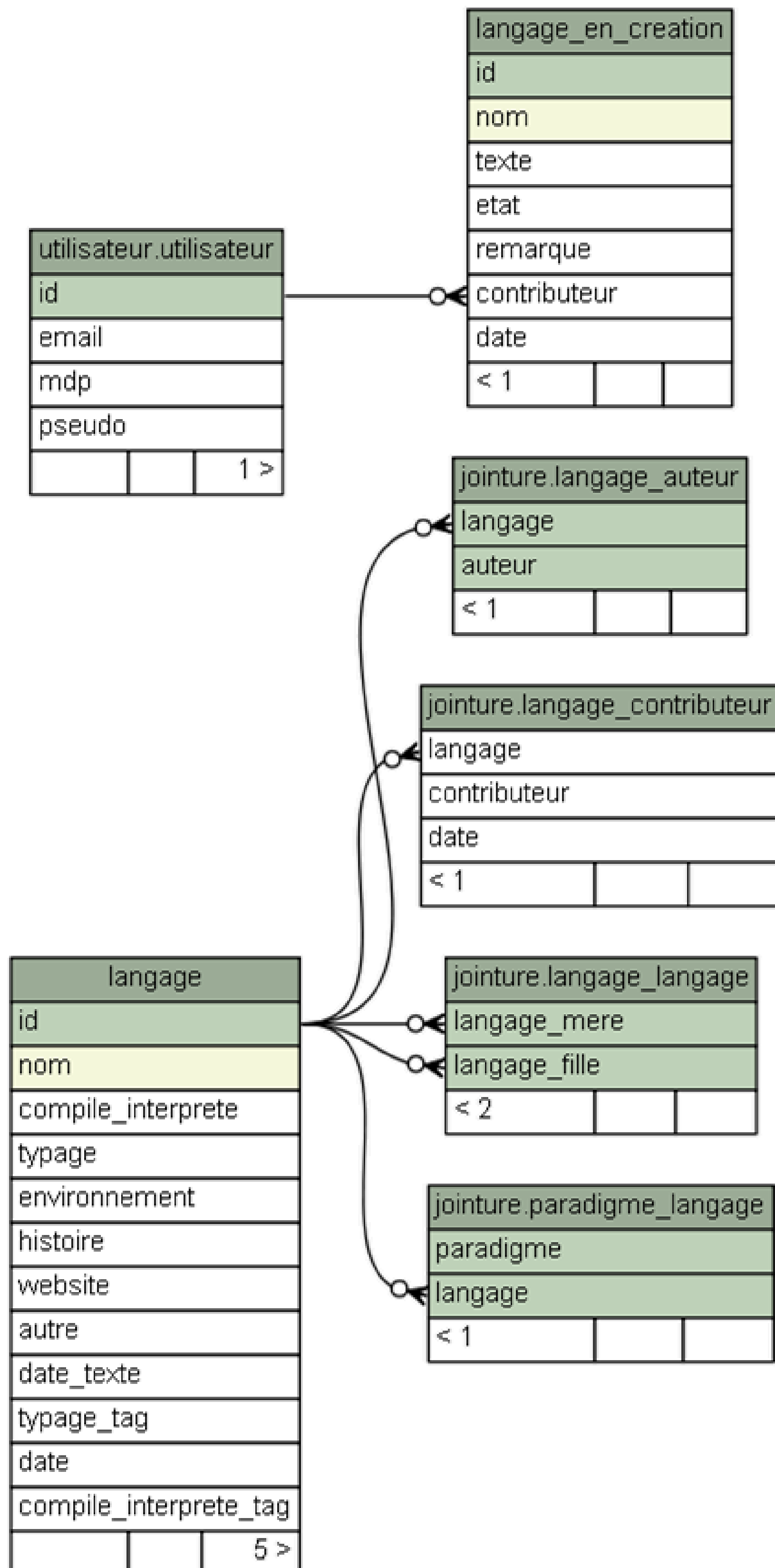

Figure 34 : Diagramme de toutes les tables présentes dans le schéma langage

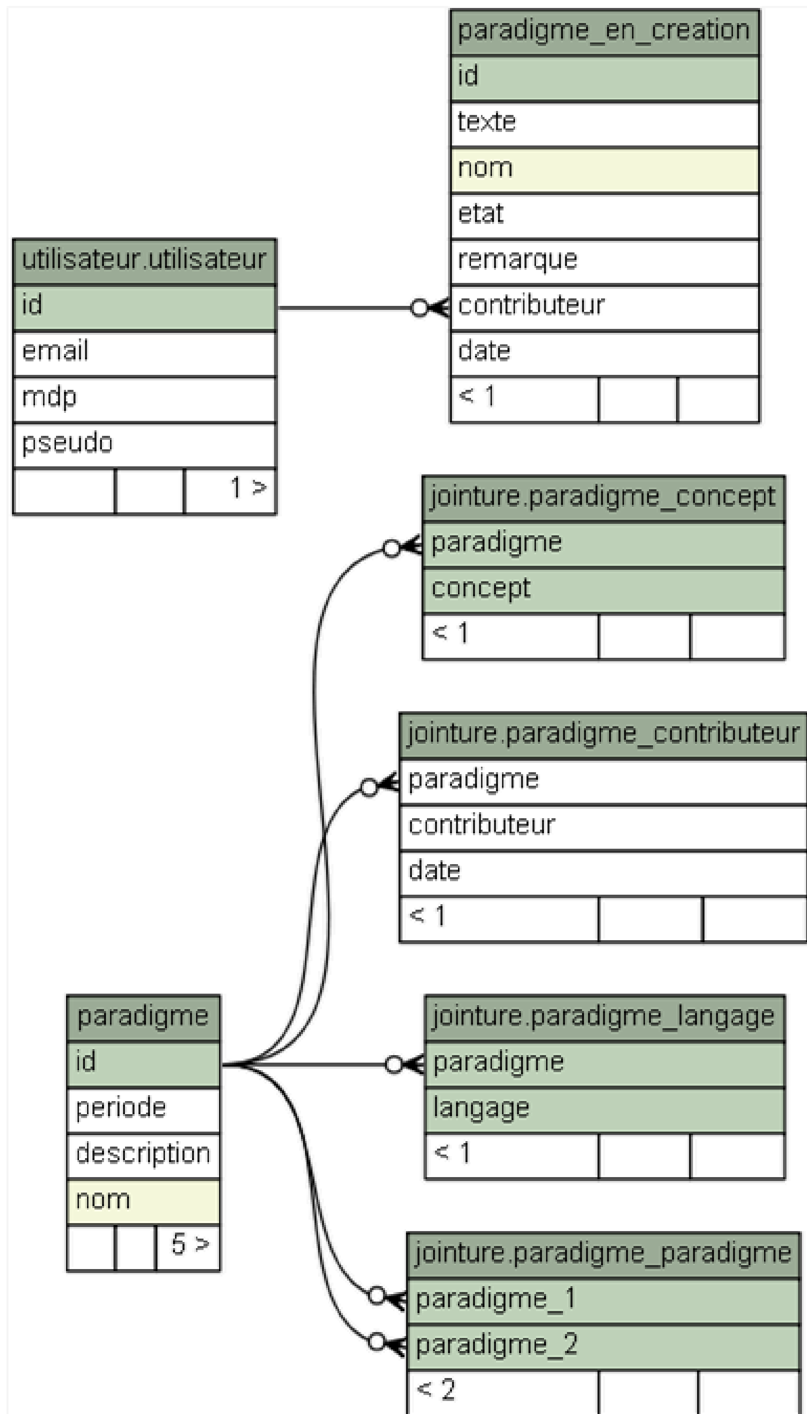

Figure 35 : Diagramme de toutes les tables présentes dans le schéma paradigme

B. Annexe benchmarks

```
Server Software:        nginx
Server Hostname:        alexandrie.ovh
Server Port:            80

Document Path:          /
Document Length:        6047 bytes

Concurrency Level:      100
Time taken for tests:   651.298 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      63900000 bytes
HTML transferred:       60470000 bytes
Requests per second:    15.35 [#/sec] (mean)
Time per request:       6512.983 [ms] (mean)
Time per request:       65.130 [ms] (mean, across all concurrent requests)
Transfer rate:          95.81 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       44   65 284.6     48    9094
Processing:    93 6397 2832.8   4979   22270
Waiting:       62 3537 2847.2   3053   16935
Total:        143 6462 2848.4   5030   22317

Percentage of the requests served within a certain time (ms)
  50%   5030
  66%   5569
  75%   7876
  80%   7956
  90%   9385
  95%  13972
  98%  15324
  99%  16989
 100%  22317 (longest request)
```

Figure 36 : Résultats du benchmark pour la page d'accueil

```
Server Software:        nginx
Server Hostname:        alexandrie.ovh
Server Port:            80

Document Path:          /langagesProgrammation/246d93a4-65a7-477b-b0ae-7a550933cfeb
Document Length:        13042 bytes

Concurrency Level:      100
Time taken for tests:   1152.912 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      133860000 bytes
HTML transferred:       130420000 bytes
Requests per second:    8.67 [#/sec] (mean)
Time per request:       11529.116 [ms] (mean)
Time per request:       115.291 [ms] (mean, across all concurrent requests)
Transfer rate:          113.38 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       40   74 223.4     60    9069
Processing:  5103 11399 1870.5  11293   20362
Waiting:      230 10994 1653.6  11195   20357
Total:       5161 11473 1862.5  11357   20412

Percentage of the requests served within a certain time (ms)
  50%  11357
  66%  11602
  75%  11861
  80%  12168
  90%  13379
  95%  14921
  98%  16891
  99%  17709
 100%  20412 (longest request)
```

Figure 37 : Résultats du benchmark pour la page d'un langage de programmation

```
Server Software:        nginx
Server Hostname:        alexandrie.ovh
Server Port:            80

Document Path:          /monProfil/fa467859-4abf-474e-bf45-aee766dad114
Document Length:        13399 bytes

Concurrency Level:      100
Time taken for tests:   611.406 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      137430000 bytes
HTML transferred:       133990000 bytes
Requests per second:    16.36 [#/sec] (mean)
Time per request:       6114.061 [ms] (mean)
Time per request:       61.141 [ms] (mean, across all concurrent requests)
Transfer rate:          219.51 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       38   52  30.8     48    3044
Processing:  4516 6016 436.2   5996    9204
Waiting:     2316 5847 522.7   5887    8982
Total:       4565 6068 436.5   6048    9251

Percentage of the requests served within a certain time (ms)
  50%   6048
  66%   6182
  75%   6273
  80%   6329
  90%   6475
  95%   6616
  98%   6779
  99%   8286
 100%   9251 (longest request)
```

Figure 38 : Résultats du benchmark pour la page de profil

```
Server Software:        nginx
Server Hostname:        alexandrie.ovh
Server Port:            80

Document Path:          /administration
Document Length:        7011 bytes

Concurrency Level:      100
Time taken for tests:   731.971 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      73540000 bytes
HTML transferred:       70110000 bytes
Requests per second:    13.66 [#/sec] (mean)
Time per request:       7319.708 [ms] (mean)
Time per request:       73.197 [ms] (mean, across all concurrent requests)
Transfer rate:          98.11 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       40   73 112.8     68    3466
Processing:   119 7214 1182.9   6849   12455
Waiting:      102 4179 2244.6   4147   12432
Total:        178 7287 1187.8   6922   12519

Percentage of the requests served within a certain time (ms)
  50%   6922
  66%   7203
  75%   7505
  80%   7728
  90%   9416
  95%   9865
  98%  10527
  99%  11098
 100%  12519 (longest request)
```

Figure 39 : Résultats du benchmark pour la page d'administration

```
Server Software:        nginx
Server Hostname:        alexandrie.ovh
Server Port:            80

Document Path:          /administration/getLangageValidation/763af35e-27b5-4b24-812b-256c35da11d3/Java
Document Length:        45491 bytes

Concurrency Level:      100
Time taken for tests:   7037.391 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      458350000 bytes
HTML transferred:       454910000 bytes
Requests per second:    1.42 [#/sec] (mean)
Time per request:       70373.911 [ms] (mean)
Time per request:       703.739 [ms] (mean, across all concurrent requests)
Transfer rate:          63.60 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       40   64 129.7     47    3090
Processing:  5123 69974 6326.9  69406   85222
Waiting:     4934 69602 6322.5  69024   85039
Total:       5171 70038 6325.6  69494   85268

Percentage of the requests served within a certain time (ms)
  50%  69494
  66%  71291
  75%  73198
  80%  74483
  90%  77566
  95%  79764
  98%  82068
  99%  83899
 100%  85268 (longest request)
```

Figure 40 : Résultats du benchmark pour la page de validation d'un langage

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique **www.uclouvain.be/epl**