Plateforme de gestion de diplômes

Dimitri Racordon

Nouvelles Technologies de l'information et de la Communication

Université de Genève

Table des matières

1 Introduction

1.1 Contexte

Dans le cadre de la réalisation de leurs projets de diplômes, il est souvent demandé aux étudiants de tenir leurs professeurs responsables informés de l'avancée de leurs travaux. En outre, la rédaction du leurs mémoires est souvent sujette à de nombreuses modifications et suggestions. La plupart de ces échanges entre professeur responsable et diplômant sont aujourd'hui fait par e-mail. En effet, ce média reste l'un des moyens les plus simples car il est rapide, ne demande pas que les deux acteurs soient disponibles au même moment et permet également un archivage des informations relatives à l'état d'avancement, ou encore des diverses versions d'un mémoire de diplôme.

Néanmoins, l'utilisation d'e-mail requiert une organisation très méticuleuse lorsque l'on souhaite retrouver rapidement une information. Il est également fréquent que les possibilités de mises en page soient sous-utilisées, soit par méconnaissance des outils de rédaction, soit parce qu'il est trop coûteux en temps de réaliser de telles mises en formes. Finalement, nous soulignerons encore que le mail est un media « fermé » qui ne permet pas la contribution d'autres acteurs que l'expéditeur et le/les destinataires.

Au vu de cette problématique, il a été proposé de mettre au point une plateforme web permettant de centraliser les échanges cités ci-dessus, au travers d'une interface homogène.

1.2 Objectifs

Notre plateforme de gestion, que nous avons appelée Cameleye, a pour objectif de simplifier la rédaction du *journal de bord* des étudiants, en proposant une sorte de blog où ceux-ci peuvent consigner diverses informations quant à l'avancement de leur travail, ou encore exposer les problématiques auxquelles ils font faces. Dans le but créer un média dépassant la simple interaction professeur/élève, Cameleye offre la possibilité à tout professeur ou étudiant de laisser des commentaires sur ces notes, pour par exemple proposer une suggestion, ou encore apporter une correction à une réflexion erronée.

En outre, Cameleye propose un système de dépôt de mémoires de diplômes. Les étudiants sont invités à mettre en ligne les différentes versions de leur mémoire que leurs professeurs peuvent ainsi consulter. Enfin, la plateforme offre la possibilité à ces derniers de laisser des annotations quant aux diverses versions successives proposées par l'étudiant.

2 Organisation générale de Cameleye

2.1 Cas d'utilisation

Nous proposons le diagramme des cas d'utilisation suivant :

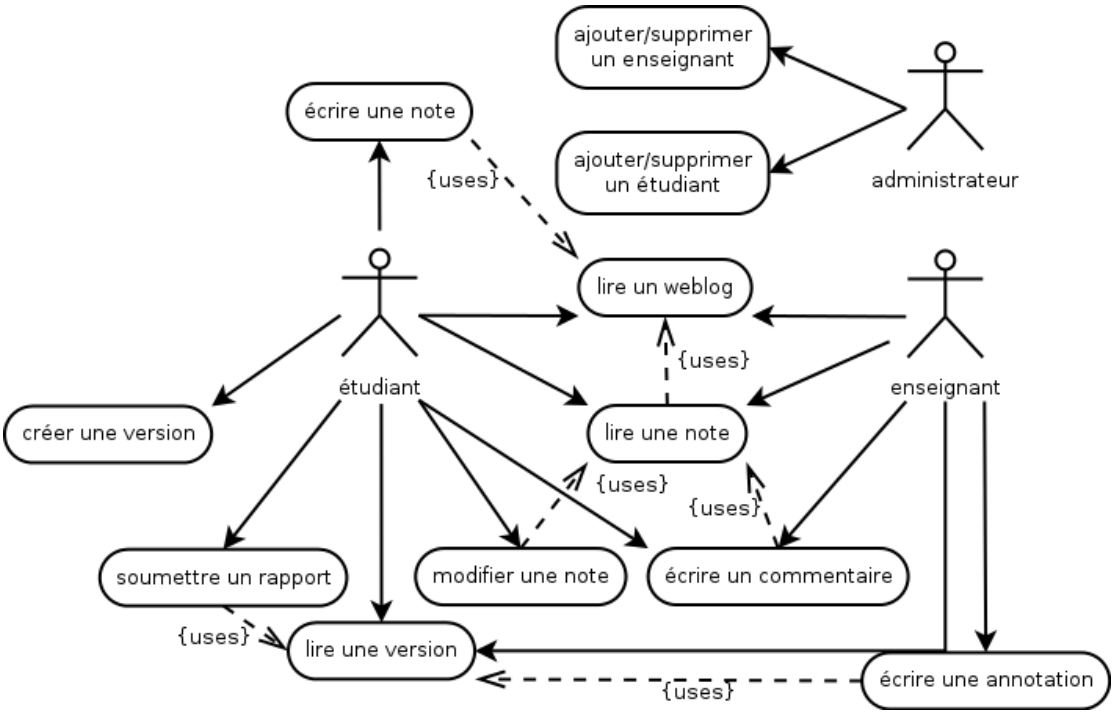

Figure 1 : diagramme des cas d'utilisation

Notre système comporte 3 acteurs, étudiant, enseignant et administrateur. A l'exception de l'administrateur qui n'a pour rôle que la création et suppression des divers utilisateurs, chaque acteur peut effectuer les mêmes actions, à quelques détails près.

En effet, seuls les étudiants sont censés créer eux-mêmes leurs différentes versions de mémoire de diplôme, pour y déposer leurs fichiers. De leurs côtés, seuls les enseignant sont censés pouvoir rédiger des annotations sur le travail de leurs étudiants.

2.2 Diagramme des classes

Nous avons réalisé notre plateforme à l'aide, entre autres, de la librairie Doctrine. Il s'agit d'un Object-Relation-Mapper, c'est-à-dire d'un outil qui permet de traduire un objet au sens de la programmation orientée objet, en schéma relationnel, compréhensible par un SGBD. Nous entrerons plus en détail dans les caractéristiques des librairies employées ultérieurement ; cette courte introduction a pour but d'expliquer l'absence de schéma relationnel dans ce document.

Pour la gestion des données de notre application, nous proposons le diagramme des classes suivant :

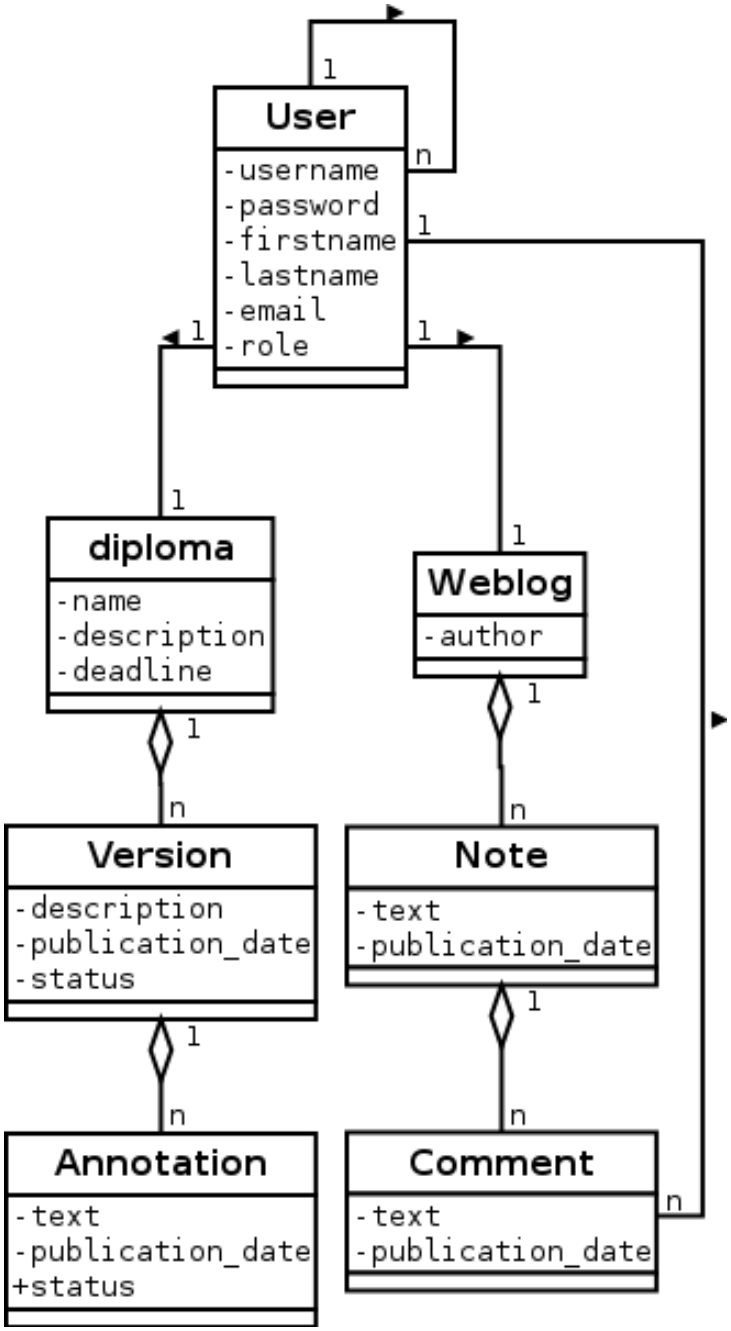

Figure 2 : diagramme des classes

Permettons-nous tout d'abord de préciser que pour des raisons de clarté, nous n'avons pas indiqué tous les *getters* et *accessers* sur le schéma ci-dessus. En effet, chaque membre privé peut être écrit ou lu au travers d'une méthode dite « magique » en langage PHP.

Comme nous pouvons le constater sur le schéma, nous ne faisons pas de distinction explicite entre élèves, enseignants et administrateurs. L'information est contenue dans le membre *role*.

2.3 Outils utilisés

Souhaitant profiter de ce projet pour découvrir de nouveaux outils, nous nous sommes fortement penchés sur l'utilisation de frameworks. Ceux-ci permettent un développement rapide, tout en respectant des standards souvent globalement reconnus. Nous faisons notamment référence au paradigme Modèle-Vue-Contrôleur.

2.3.1 Django

Nous avons commencé par nous pencher sur le framework dénommé Django. Ce dernier, réalisé en python, se veut aujourd'hui être une alternative au très célèbre framework Java, Spring.

Django est basé sur le paradigme Modèle-Vue-Contrôleur, qu'il exploite à son plein potentiel de par la nature orientée objet de python. En effet, chaque objet est un modèle dont la persistance est gérée par le framework, de manière totalement transparente, quelque soit le type de base de donnée employée. En d'autres termes, Django prend en charge la traduction de classes python en schéma SQL, laissant au développeur le soin de s'occuper uniquement de la logique de son application.

Django offre un système de vues, appelées *templates* au sein de la librairie, également basé sur une approche orientée objet. Chaque *template* peut être perçu comme une classe, disposant de certaines caractéristiques HTML telles que l'agencement de tel ou tel cadre. Celles-là peuvent être héritées par d'autres *templates* qui peuvent alors véritablement les surcharger en modifiant le comportement des balises HTML ou en les utilisant telles quelles. En outre, il est également possible de combiner plusieurs *templates* afin de générer une page HTML complète.

Ajoutons encore que Django propose un système d'écritures d'url intéressant, en permettant de mapper n'importe quelle expression à une adresse particulière. Ces expressions peuvent être tout-à-fait statiques, ou alors générées à partir d'expressions régulières.

Finalement, nous soulignerons que Django est une librairie très prometteuse, disposant d'un réel potentiel. Il ne nous paraît pas exagéré de penser qu'elle peut en effet être une alternative viable à Spring. Malheureusement, nous avons décidé d'abandonner son utilisation pour les besoins de notre application. En effet, même si la prise en main du framework n'est pas spécialement difficile, l'utilisation de python représentait un effort trop important, de part une méconnaissance certaine langage.

2.3.2 Zend Framework

Zend Framework est un framework PHP, réalisé par la société Zend. A l'inverse de Django, il ne propose pas d'Object-Relation-Mapper complet mais dispose néanmoins d'une armada d'outils permettant de simplifier l'écriture de code. Egalement basé sur un paradigme Modèle-Vue-Contrôleur, Zend Framework

propose lui aussi une architecture très robuste permettant l'extension de ses fonctionnalités au travers du mécanisme d'héritage.

Nous avons choisi Zend Framework car il s'agit d'un outil très bien documenté, proposant une vision très intéressante du fonctionnement d'une application complexe.

2.3.3 Doctrine

Doctrine est un Object-Relation-Mapper (ORM), écrit en PHP. Il permet la traduction de classes PHP en schéma relationnel au travers de 3 méthodes :

1. Par annotation ; les informations relatives au mapping des classes sont indiquées directement dans la définition de celles-ci, au moyen de commentaires respectant une certaine syntaxe. Ce type de notation est également vastement employé dans Java.
2. Par fichier XML ; les informations relatives au mapping des classes sont indiquées dans un fichier XML séparé.
3. Par fichiers Yaml; à l'instar de la méthode XML, les informations de mapping sont indiqués dans un fichier séparé, au format Yaml dans ce cas.

Afin d'améliorer les performances de sont ORM, Doctrine utilise des classes appelées *Proxy* qui représentent en quelque sorte les signatures des classes utilisées. Les données réelles sont ainsi chargées uniquement lorsqu'elles sont demandées. En outre, Doctrine permet aussi de mettre une certaine quantité de données en cache pour éviter des appels trop fréquents à la base de données. A l'instar des *Proxy*, ce mécanisme est géré de manière totalement transparente.

Nous avons choisi Doctrine car il s'agit d'une librairie très largement plébiscitée sur la toile, en tant qu'ORM. En effet, dans l'optique de découvrir des technologies alliant performances et vitesses d'écritures, nous souhaitions nous défaire de la réalisation de requêtes SQL complexes afin de travailler à un niveau d'abstraction plus élevé, nous autorisant par ailleurs une plus grande flexibilité.

2.3.4 HTML Purifier

HTML Purifier est un outil relativement simple qui permet d'extraire le code éventuellement malicieux d'une chaîne de caractère.

Dans une optique de sécurité, nous souhaitions en effet prévenir de manière efficace les attaques de type Cross-Site-Scripting (XSS), sans pour autant amputer notre application de la possibilité de rédiger des notes stylisées. Nous avons choisi cette solution car elle s'adaptait parfaitement avec les autres outils que nous avons employés.

2.3.5 jQuery

jQuery est une librairie JavaScript conçue pour non seulement faciliter l'écriture de code, mais également assurer la compatibilité des scripts entre les différents navigateurs. Nous avons choisi jQuery pour le nombre incalculable de possibilités qu'elle propose, pour tout type d'application côté client.

2.3.6 Twitter Bootstrap

Twiter Bootstrap est un framework HTML5, permettant de mettre en place très simplement des interfaces graphiques complexes et conviviales. Principalement basé sur l'utilisation de CSS3, il s'agit d'un framework extrêmement modulable. En effet, chaque style proposé peut être réécrit pour correspondre à l'application développée. En outre, Twitter Bootstrap peut être utilisée conjointement avec jQuery pour créer des interfaces dynamiques tels que des menus déroulant, ou encore des systèmes d'alertes.

Nous avons choisi cette librairie pour son vaste choix de composants, ainsi que pour son adaptabilité. De plus, nous avons également été séduits par la qualité de sa documentation, contribuant à un développement simple et rapide.

3 Fonctionnement de Cameleye

3.1 Fonctionnement logique

Comme nous l'avons évoqué au chapitre précédent, notre application est principalement basée sur l'utilisation conjointe de Zend Framework et Doctrine.

Si nous nous intéressons à l'architecture proposée, nous constatons que le répertoire racine est organisé de la manière suivante :

```
cameleye/
	|-application/
		|-Bootstrap.php
		|-configs/
			|-application.ini
			|-routes.ini
		|-controllers/
			|-helpers/
				|-EntityManager.php
			|-…
		|-forms/
		|-layouts/
			|-filters/
			|-helpers/
			|-scripts/
				|-layout.phtml
				|-navigation.account.phtml
		|-models/
			|-Entities/
			|-Proxies/
		|-modules/
		|-services/
		|-views/
			|-filters/
			|-helpers/
			|-scripts/
	|-data/
		|-cache/
		|-logs/
		|-sqlite/
		|-thesis/
	|-docs/
	|-library/
		|-Cameleye/
		|-Doctrine/
		|-HTMLPurifier/
		|-Zend/
	|-public/
	|-scripts/
```

Nous ne détaillerons pas l'ensemble des répertoires dans ce document, mais nous intéresserons particulièrement aux éléments clefs dans le fonctionnement

d'une application Zend Framework. Nous noterons également que cette arborescence ne contient pas la totalité des fichiers et répertoires de notre application.

Tout d'abord, concentrons-nous sur le répertoire *application*. Celui-ci constitue le cœur de notre site web. Lorsqu'une page est générée, le fichier *Bootstrap.php* est le premier à être exécuté. Se basant sur les informations indiquées dans le fichier de configuration *application.ini*, il initialise les différents espaces de noms, ainsi que l'auto-loader, une classe de Zend permettant de charger dynamiquement une dépendance lorsque l'application en a besoin, sans que celle-ci soit explicitement indiquée dans des directives *include()* ou *require()*.

Ce même script est également responsable de l'initialisation de Doctrine. Comme nous souhaitions utiliser Doctrine conjointement avec Zend Framework, nous avons dû spécifiquement configurer Doctrine pour qu'elle utilise l'auto-loader de Zend en lieu et place du sien.

Une fois l'application mise en route, c'est Zend qui appelle automatiquement le contrôleur correspondant à la requête effectuée. Une url est toujours composée au moins du contrôleur ciblé, ainsi que de la vue demandée ; exemple : http://cameleye.etu/public/**diploma**/**view**/sergey.brin. Dans cet exemple, le contrôleur impliqué est *diploma*, sa vue est *view*.

Le contrôleur effectue les opérations logiques sur les objets le concernant puis appelle la vue ayant pour but d'afficher ses résultats. Chaque vue est constituée d'un fichier html, déposé dans le répertoire *application/views/scripts*.

Les différentes classes relatives aux données et gérées par Doctrine résident dans le répertoire *application/models/Entities*.

En dehors du répertoire *application*, nous avons le répertoire *data* qui contient les différents mémoires de diplômes déposés, ainsi que la base de donnée. Nous noterons que nous employons ici une base de donnée SQLite car son déploiement est très simple. Le répertoire *library* contient les différentes librairies PHP que nous employons. Nous noterons qu'il y réside une librairie *Cameleye*. En effet, pour les besoin de notre application, nous avons étendu certaines classes proposées par Zend afin notamment d'assurer la compatibilité avec Doctrine.

Finalement nous évoquerons encore le répertoire public, lequel contient les différentes images, scripts JavaScript et feuilles de style CSS de notre application. Lorsqu'une vue est appelée, le navigateur fait référence à ce répertoire qui, idéalement, devrait être le seul accessible par le serveur web.

3.2 Fonctionnement de l'interface

Sur sa page d'accueil, Cameleye propose directement deux liens pointant vers le module weblog, respectivement le module diploma.

La plupart des ressources de Cameleye sont disponibles pour tout visiteur anonyme (cf. figure 3). Néanmoins, l'interface permet à l'utilisateur de se connecter à tout moment, en cliquant sur le lien en haut à droite puis en saisissant son nom d'utilisateur, ainsi que son mot de passe.

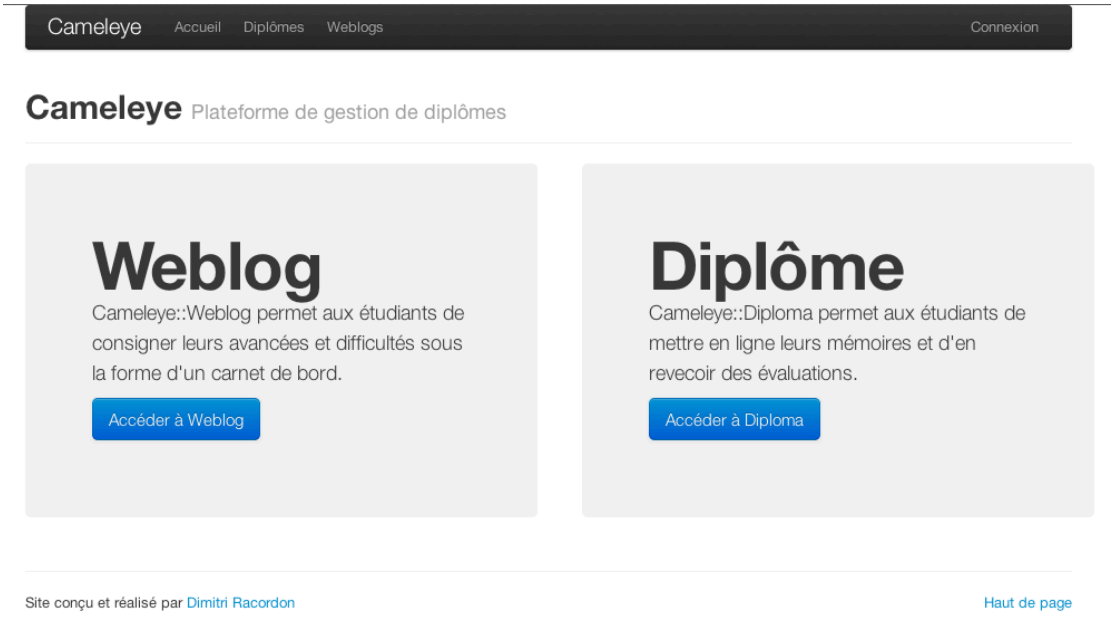

Figure 3 : page d'accueil

Lorsque l'on accède au module weblog, la liste des weblogs est affichée et un bouton invite l'utilisateur à consulter l'un d'entre eux. S'il est connecté et qu'il dispose des droits en écriture, alors il pourra éditer de nouvelles notes au travers de l'éditeur WYSIWYG, modifier et/ou supprimer les notes déjà existantes (cf. figure 4).

Lorsque l'on accède au module diploma, la liste des diplômes est affichée et un bouton invite l'utilisateur à consulter l'un d'entre eux, à l'instar du module weblog. Pour chaque diplôme est affichée une fiche descriptive, ainsi qu'une étiquette pour chaque version disponible (cf. figure 5). Un clic sur l'une d'entre elles permet d'afficher ladite version du mémoire de diplôme.

Si l'utilisateur est connecté et qu'il dispose des droits, il peut ajouter des annotations sur le mémoire de diplôme affiché (cf. figure 6).

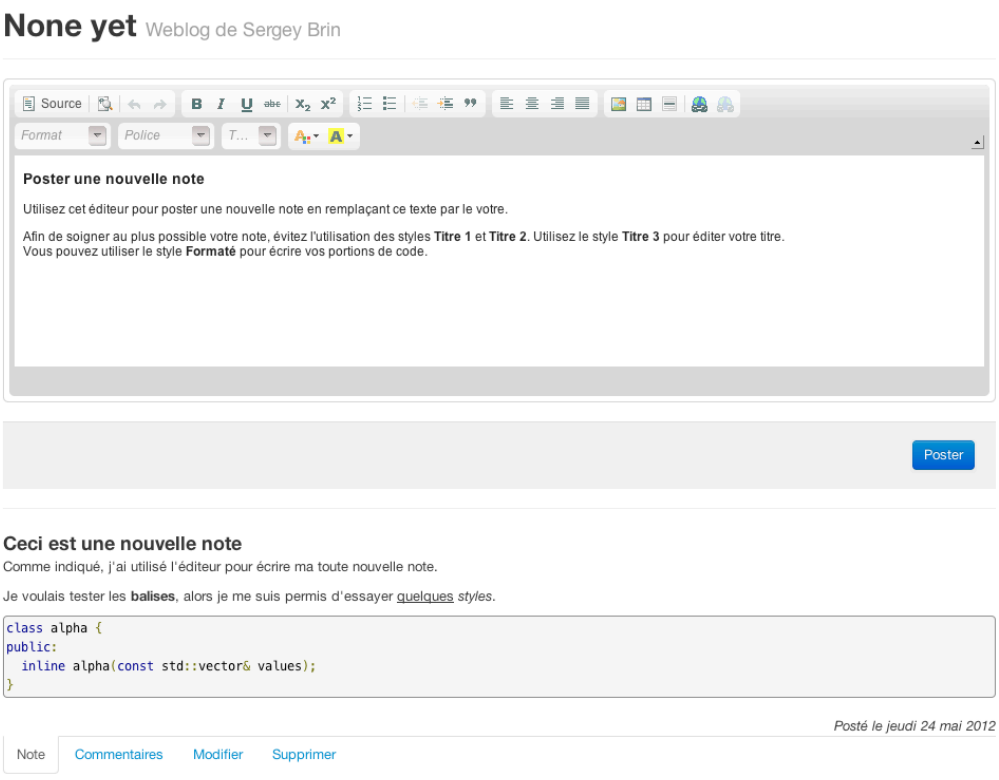

Figure 4 : interface weblog

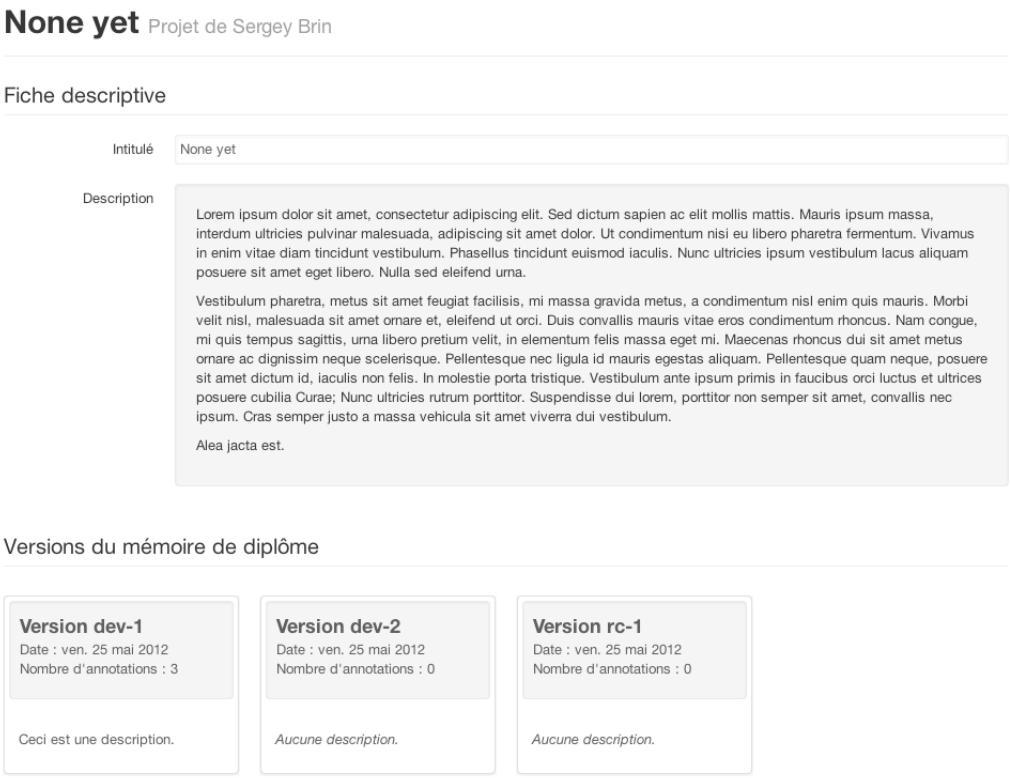

Figure 5 : affichage d'un diplôme

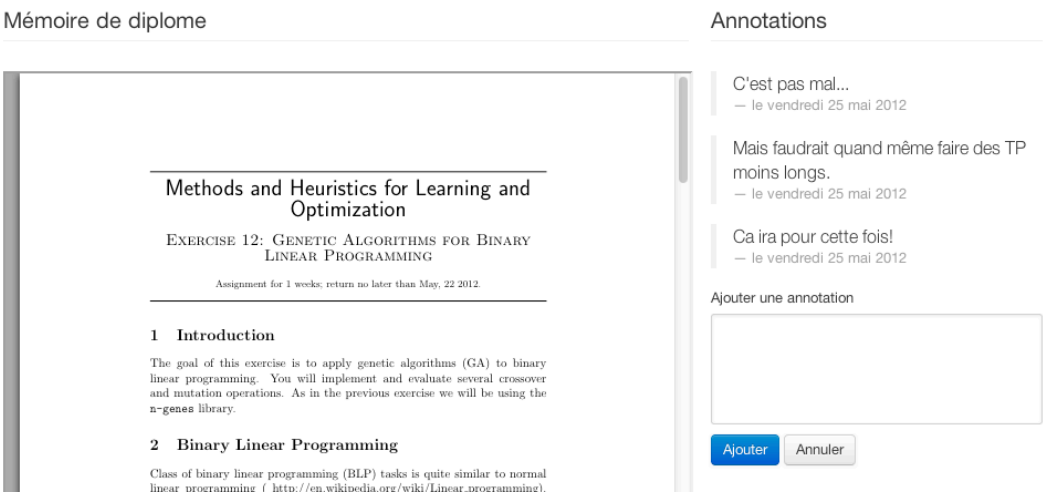

Figure 6 : affichage d'une version de mémoire

3.3 Tests

Zend Framework et Doctrine proposent des tests unitaires pour vérifier le fonctionnement du MVC, respectivement de l'ORM. De tels tests ont été menés sur l'ensemble de notre application, de sorte à vérifier les points suivants :

- Le modèle correspondant à l'url indiquée est appelé correctement.
- La vue correspondant à l'url indiquée est appelée correctement.
- L'insertion d'une entité dans la base de donnée est enregistrée correctement.
- La modification d'une entité dans la base de donnée est enregistrée correctement.
- La suppression d'une entité dans la base de donnée est enregistrée correctement.

En outre, nous avons réalisé « manuellement » un certain nombre de tests sur chacune des fonctionnalités implémentées, en testant à chaque fois de réaliser l'opération normalement, de réaliser l'opération en saisissant intentionnellement des données invalides ou malicieuses et de réaliser l'opération sans disposer des droits nécessaires.

Chacun des tests réalisés s'est révélé concluant.

4 Conclusion

4.1 Difficultés rencontrées

L'implémentation de multiples librairies de l'envergure de Zend Framework et Doctrine n'est pas chose aisée. En effet, même si ces outils proposent des architectures relativement robustes, il n'est pas toujours évident de savoir comment les entremêler.

Nous soulignerons également que Doctrine 2 est basé sur l'utilisation des *namespaces* PHP, une fonctionnalité apparue avec la version 5.3, alors que Zend Framework 1.11 utilise encore des règles de nommages dites *CamelCase*, où le nom de la classe contient le chemin pour y accéder. L'adaptation de l'auto-loader de Zend a donc constitué un défi de taille.

Enfin, nous ajouterons que l'utilisation de Latex au sein d'une application PHP représente un défi énorme. En effet, si PHP permet un développement presque identique quelque soit la plateforme considérée, il n'en est rien pour la compilation et l'interprétation de fichier tex. La plupart des outils proposés fonctionnent pour Linux, dans un cadre extrêmement retreint et rares sont les *parser* capables des réellement convertir un document latex entièrement en HTML.

4.2 Réflexion quant au travail effectuée

Malgré les quelques difficultés évoquées ci-dessus, ce travail m'a été extrêmement enrichissant. En effet, j'y ai eu l'occasion d'utiliser des librairies professionnelles dans un cadre concret.

Il est également agréable de voir à quel point la communauté est active autour de certains outils, offrant un précieux complément d'information à la documentation officielle qui souffre parfois d'une vision trop étroite du fonctionnement de tel ou tel composant.

4.3 Limitations

La prise en charge de Latex n'a pas pu être assurée. Je suis parvenu à installer un *parser* capable de convertir une formule Latex en image png, malheureusement l'opération n'est réalisable que sur des fragments de documents Latex et pas sur un fichier complet.

L'affichage des diplômes étant réalisé au travers d'une balise *iframe*, il est impossible de redimensionner le document affiché.

4.4 Améliorations envisageables

N'ayant pas pu achever correctement la gestion des rapports au format latex, il serait intéressant d'améliorer ce point en proposant soit un *parser* capable de compiler entièrement un fichier latex afin de le convertir en HTML, soit en proposant un meilleur système d'affichage des fichiers PDF. Il serait même envisageable de mettre au point un système d'annotation directement sur le fichier PDF.

D'autres pistes peuvent également être évoquées concernant par exemple l'interface graphique. L'ajout de la technologie Ajax peut constituer un pas intéressant vers une interface encore plus conviviale, notamment au niveau du dépôt des fichiers.

Enfin, de grosses améliorations peuvent être faites quant au retour d'erreur de la plateforme. Celles-ci sont parfois trop vagues et ne permettent pas nécessairement d'informer clairement l'utilisateur du problème rencontré.