

# Rapport - Projet informatique

Félix Bélanger-Robillard

April 15, 2017

# Contents

# 1 Introduction

Le présent projet avait pour but de créer un webservice servant les fonctionnalités de BiBler, puis de l'intégrer à ReLiS (Revue Littéraire Systématique) afin d'offrir une plateforme web permettant d'allier les fonctionnalités propres à ces deux logiciels, le tout sous la supervision de monsieur Eugène Syriani et avec la précieuse collaboration de Brice-Michel Bigendako qui s'occupe de ReLiS. Le projet intégrait des connaissances en PHP et Python. Le projet m'a aussi amené à travailler avec des serveurs de type Apache 2 et avec les systèmes de permissions Unix. Pour la rédaction du rapport, j'ai utilisé de nouveaux paquets LaTeX et au niveau de la documentation du projet, j'ai appris à utiliser Sphinx pour la générer.

-Mentionner tout, montrer qu'il y avait des challenges non trivial

-Section: intro( définition de concepts, termes et outils ), core, évaluation, conclusion, remerciements, références bibliographiques (technologies), annexes,

conclusion: résumé, puis future sabrina et khady, A+

## 1.1 Terminologie

BiBler est un outil de gestion de référence bibliographique. BiBler se sert de BibTeX, une librairie LaTeX de gestion référence, et offre une interface graphique afin de faciliter l'importation, l'exportation et la modification de références. BiBler offre également un aperçu des références et facilite l'insertion au sein de textes. BiBler génère également des clés d'identification pour les documents insérés qui peuvent ensuite être utilisés dans un document LaTeX.

ReLiS, Revue Littéraire Systémique, est un outil web rédigé en PHP dans le framework CodeIgniter. ReLiS vise à aider les chercheurs au moment de consulter la littérature d'un sujet précis. ReLiS permet à ces chercheurs de réduire l'ampleur de la tâche en automatisant en bonne partie la classification et avec des outils permettant un processus systématique de révision des articles.

## 2 Analyse des besoins

Le besoin initial était de mettre BiBler en webservice d'une part et de l'intégrer à ReLiS d'une autre part. La première décision prise à ce niveau, a été de conserver le code de BiBler et d'utiliser un framework Python afin de ne pas avoir à réécrire le code du logiciel et dans le soucis de pouvoir maintenir le webservice plus aisément. Avec le webservice directement dépendant du code source originel, la maintenance à ce niveau, avec une nouvelle version de BiBler, sera de remplacer la librairie à l'emplacement physique de l'importation pour le webservice et de s'assurer ensuite que les tests unitaires sont toujours valides.

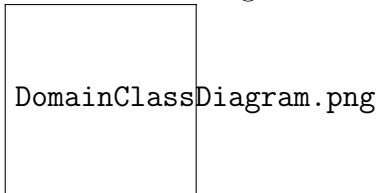
Afin de conserver au minimum le nombre de technologies employées, un framework qui n'utilise que Python était souhaité et, idéalement, avec un minimum de contraintes sur son utilisation. Il fallait également que l'application puisse être déployée sur un serveur Apache 2 à cause des contraintes des serveurs du département. C'est pour cette raison que Web.py [?] a été choisi comme framework pour le webservice. De cette façon, avec BiBler, tout comme pour son webservice, le seul langage utilisé demeure Python 3.

Au niveau de l'intégration avec ReLiS, les besoins étaient de pouvoir appeler le webservice, de formater correctement les fichiers BibTeX, de générer les clés pour les articles et de pouvoir intégrer l'abstract au niveau de la référence bibtex. Le dernier besoin a entraîné une maintenance au niveau de BiBler, puisque ce n'était pas un champ qui y était présent. La génération de clés était également une exigence partiellement difficile à remplir, car la décision avait été prise de ne pas conserver de données utilisateurs du côté du webservice. Pour cette raison, il fallait également ajouter au sein de ReLiS une vérification au niveau des doublons pour modifier leur clé, puisque les clefs des publications d'un auteur au cours d'une année allait être les mêmes.

### 3 Conception

Plusieurs choix ont été effectués par rapport à l'implantation du webservice par rapport à l'approche desktop de BiBler. Le webservice ne conserve aucune donnée envoyée au serveur, il effectue l'opération et retourne le résultat désiré. Afin d'arriver à ce résultat, une nouvelle instance de l'application est instanciée à chaque appel de méthode. Sans cette particularité, l'exportation de résultats sur une instance commune aurait retourné l'ensemble des fiches importées et comme on ne veut pas avoir à gérer le contenu utilisateur, ç'aurait été un effet indésirable. Comme on ne peut avoir d'instance commune sur plusieurs requêtes, il y a tout de même possibilité d'envoyer un fichier et de le traiter avec un seul appel au webservice et d'exporter ensuite un fichier.

Pour l'intégration avec ReLiS, puisque ReLiS est construit sur une architecture Modèle-Vue-Contrôleur (MVC), l'idée est d'ajouter un contrôleur spécifiquement pour ReLiS avec une vue associée. Tout en sachant les méthodes dévoilées par le webservice, il a été également pensé de faire appel au patron de proxy pour encapsuler les méthodes PHP nécessaires aux appels vers le webservice. De cette façon, les travaux futurs sur ReLiS ou tout autre application php se verraient grandement facilités. La conception mène donc au diagramme suivant:




## 4 Design

### 4.1 Modélisation

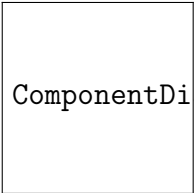
Le webservice dépend de BiBler et n'utilise que la classe BiBlerApp qui correspond à l'API de cette dernière. Ensuite, le webservice instancie un objet BiBlerApp et encapsule chacune des méthodes demandées. Chaque méthode est invoquée par un URL différent. Chacun de ses URLs réfèrent à une classe dont la méthode POST renvoie à la méthode correspondante de BiBler qui est encapsulée par la classe BiBlerWrapper du webservice pour assurer les instanciations locales nécessaires et les appels de méthodes de l'API. Les méthodes de l'API qui n'ont pas été encapsulées sont celles qui effectuent des query dans une application déjà existante qui contient des entrées. Elles pourraient être encapsulées, mais vu le choix de ne conserver aucune donnée, le résultat serait toujours nul.

Il a été décidé que le webservice possèderait deux fichiers. Le premier, application.py, servirait de contrôleur afin de rediriger les demandes du client vers les méthodes contenues dans la classe BiBlerWrapper, qui, comme son nom l'indique, servait d'emballeur aux méthodes de la classe BiBlerApp du code source de BiBler, ce qui correspond au diagramme de classes suivant, en incluant le Proxy:



ClassDiagram.png

Au niveau architectural, les liens entre BiBler, le webservice et ReLiS sont définis tels que présenté dans le diagramme de composantes suivant:



ComponentDiagram.png

Selon les besoins, il fallait étendre les fonctionnalités de BiBler pour accommoder les besoins spécifiques de ReLiS, mais l'interfaçage était déjà présent avec la classe BiBler-App, il fallait créer de toute pièce le webservice en utilisant l'interface fournie par

BiBlerApp et de créer une interface permettant à ReLiS de l'utiliser. Il fallait étendre également ReLiS pour pouvoir utiliser l'interface prochaine du webservice.

## 4.2 Justification de l'utilisation de POST vs GET

Les méthodes POST et GET sont des protocoles http servant au transfert d'information sur le web. On les distingue habituellement par le fait que le protocole GET transmet l'information via l'URI, tandis que le protocole POST cache l'information au moment de la transmission. Dans les faits, plusieurs autres différences font en sorte que l'un ou l'autre des protocoles est mieux adapté à notre cas d'utilisation qui est de soumettre une chaîne de caractère au webservice afin qu'il retourne également une chaîne de caractère au client.

Chacune des méthodes offraient des avantages distincts. La méthode http GET permettait de faire une requête en passant directement les paramètres dans l'url de la requête. Au niveau de la mémoire, c'est également une opération qui peut être mise en cache afin de ne pas renvoyer d'informations inutiles au serveur en cas de requête partielle [?]. Sémantiquement, c'est également une opération où l'on demande au serveur de nous fournir de l'information. C'était donc une opération qui semblait initialement correspondre aux besoins du projet. Le méthode http GET a, par contre, une lacune importante quant au transfert de données:

The HTTP protocol does not place any a priori limit on the length of a URI. Servers MUST be able to handle the URI of any resource they serve, and SHOULD be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs. A server SHOULD return 414 (Request-URI Too Long) status if a URI is longer than the server can handle (see section 10.4.15).

Note: Servers ought to be cautious about depending on URI lengths above 255 bytes, because some older client or proxy implementations might not properly support these lengths. [?]

Même si la longueur n'est modulée par les protocoles eux-mêmes, on ne peut être assuré que tout client aura la même expérience en fonction des contraintes des serveurs



et proxy impliqués dans les opérations. C’est d’ailleurs ce qui a marqué le glas de la méthode GET au sein du webservice, lorsque des tests préliminaires avec plusieurs entrées BibTeX envoyés en simultané via le protocole GET n’ont pas retournés de résultat, alors que les mêmes opérations avec le protocole POST retournaient un résultat satisfaisant. Nous entrerons plus en détail dans la section sur les tests de performance à ce sujet. La gestion des configurations des serveurs et proxy du client sont un désagrément suffisant pour ne pas vouloir s’y ingérer et justifie en lui seul l’utilisation de POST. En même temps, POST garantie une plus grande sécurité de l’information, puisque l’information ne transite pas explicitement à par l’URI envoyé. [?] Sémantiquement, son utilisation est également justifiable, car même si l’on attend une réponse du serveur, le client lui soumet tout de même une série de données qui doivent être traitées.

### 4.3 Stabilité et abstraction

Si l’on considère seulement le webservice, sans son intégration, il est dépendant de la classe BiBlerApp et aucune classe n’en dépend. Son instabilité est donc de 1 et, comme son abstraction est 0, le webservice est complètement dépendant. Si on le considère dans son intégration, un lien afférent supplémentaire est ajouté du côté de ReLiS, ce qui lui confère un ratio de 12:0, ce qui ne semble pas idéal. Le rôle de médiateur du webservice lui procure une stabilité assez fragile puisqu’il est à la fois dépendant de BiBler et responsable vis-à-vis ReLiS. Par contre, l’idée derrière le webservice est d’interfacer pour le web l’API de BiBler, ce qui fait en sorte qu’une fois l’API dévoilée, la responsabilité incombe plutôt au client pour l’utilisation correcte du service offert. D’un autre côté, si le client est insatisfait, puisque le webservice ne retourne aucun contenu qu’il produit lui-même, le problème ne sera forcément dû au webservice en soi.

## 5 Implémentation

Au sein du fichier `application.py`, la variable `urls` est une collection de paires de strings correspondant respectivement à une URI et à une classe. Les classes présentes jouent le rôle de handler par rapport aux requêtes. La variable `app` instancie l'application à partir d'une méthode du framework qui prend en paramètre la variable `urls` et le résultat de la méthode `globals()` qui représente un dictionnaire du module actuel.

## 6 Tests et documentation

Les cas de tests unitaires étant les mêmes que pour BiBler, les oracles des Tests de BiBler ont été réutilisés pour les tests unitaires du webservice. La batterie de tests est beaucoup plus restreintes, puisqu'une grande partie des méthodes de BiBler n'ont pas été dévoilées dans son webservice. Les tests ont été rédigés grâce au module Pyunit pour Eclipse IDE. Le tout est colligé dans la classe TestAll qui exécute successivement l'ensemble des tests.

La documentation a été rédigée en docstring rst [?], puis générée avec l'outil Sphinx-python. Une fois la documentation rédigée correctement en rst, Sphinx peut être utilisé. la commande `$ sphinx-quickstart` invoquée dans un terminal crée un répertoire avec une configuration de base pour la génération de la documentation. Le répertoire contient trois sous-répertoires build, Makefile et source. La configuration peut être modifiée au sein du fichier `config.py` du sous-répertoire source. La commande `$ sphinx-build -b html sourcedir builddir` permet de générer les fichiers rst à partir du code source python. Finalement, la commande `$ make html` invoquée à partir du répertoire initialement créé par `$ sphinx-quickstart`. La dernière commande construit les fichiers html à partir des fichiers compris dans le répertoire source. À partir du fichier `index.html`, on peut maintenant naviguer la documentation du code source et également choisir de lire des extraits ici du code source.

## 7 Intégration et déploiement

Les méthodes d'export ont pour attrait de pouvoir envoyer un fichier BibTeX et de le transformer en fichier BibTeX corrigé, en CSV, en HTML ou en SQL. Par contre, cela nécessite l'écriture dans un fichier, ce qui cause problème dans des cas d'écriture en simultanée. Au moment de la rédaction, il avait été choisi de ne pas intégrer ses fonctionnalités au webservice. Par contre, par l'utilisation du patron d'état, il serait possible d'assurer qu'une seule requête client n'accède au fichier à la fois. Par contre, cela ralentirait nettement la vitesse de retour du webservice puisque les clients demandant l'écriture sur un fichier en cours d'écriture devraient attendre que celui-ci soit disponible afin l'exécution de sa requête.

L'utilisation attendue du webservice est via une requête http POST vers une méthode du webservice. Dans le cas de ReLiS ou d'un service basé sur PHP, la fonction suivante permet de faire une requête, attendu que \$url contient l'url de la méthode désirée et que \$data contient les données à transmettre sous forme de String:

---

```
function httpPost($url, $data)
{

    $ch = curl_init( $url );
    curl_setopt( $ch, CURLOPT_POST, 1);
    curl_setopt( $ch, CURLOPT_POSTFIELDS, urlencode($data));
    curl_setopt( $ch, CURLOPT_FOLLOWLOCATION, 1);
    curl_setopt( $ch, CURLOPT_HEADER, 0);
    curl_setopt( $ch, CURLOPT_RETURNTRANSFER, 1);

    $response = curl_exec( $ch );
    return $response;
}
```

---

Pour encore plus simplifier la chose, le patron de Proxy a été implémenté dans une petite librairie php afin d'encapsuler davantage les requêtes. Une classe Singleton BiBlerPost qui contient un attribut privé URL et un attribut privé instance sert à

invoquer les méthodes du webservice et les méthodes publiques de la classe ont le même nom que les classes correspondantes du webservice et ne demandent qu'un argument qui correspond au contenu à envoyer pour traitement. On peut modifier l'URL qui correspond à l'emplacement du webservice avec la méthode `setURL` qui prend un URI en argument. Des modifications subséquentes au webservice pourrait étendre la classe actuelle afin d'incorporer un plus grand nombre de méthodes ou de modifier le comportement des méthodes actuelles par override afin de correspondre à une modification éventuelle de l'interface fournie par le webservice. Avec le Proxy actuel, le client n'a donc qu'à déterminer l'emplacement web du webservice appelée et à appeler les méthodes désirées sur l'instance unique de la classe.

Pour l'intégration avec ReLiS, un champ `abstract` a été ajouté au code originel de `BiBler`. Pour ce faire, `abstract` a été ajouté à l'énumération des champs possibles. Une classe `Abstract` héritant de `Field` a également été ajoutée et `abstract` a été ajouté aux champs possibles de divers éléments, comme `Article`.

## 8 Maintenance

Impact, maintenance performance

## 9 Conclusion

### 9.1 Remerciements

Un merci infini à ceux qui m'ont aidé et accompagnés dans cette tâche:

Eugène Syriani, Ph.D., pour la supervision et les conseils précieux

Brice-Michel Bigendak, pour ReLiS et le débogage en PHP, L'équipe technique du DIRO, pour le support technique et le serveur