

**Rapport MIF13 – MIF17**

MultiMIF 2015 – 2016

**

**Symp’le Event Manager**

**Groupe O**

Jordan MARTIN Jules SAUVINET Alix GONNOT

Landry Mélaine LEBATTO Tom DUSSEAUX Clément BLAISE

Lien forge : http://forge.univ-lyon1.fr/projects/p1201336-multimif/

**Sommaire**

[1. L’Architecture de l’Application 3](#_Toc437292830)

[2. Réalisation technique 5](#_Toc437292831)

[2.1. Les décisions prises 5](#_Toc437292832)

[2.1.1. Représentation des éléments d’une conférence 5](#_Toc437292833)

[2.1.2. Import des fichiers 6](#_Toc437292834)

[2.1.3. Compte et authentification 7](#_Toc437292835)

[2.1.4. Gestion des rôles 8](#_Toc437292836)

[2.2. Création d’entités 9](#_Toc437292837)

[2.3. Schéma de navigation 10](#_Toc437292838)

[2.4. Design patterns utilisés 11](#_Toc437292839)

[2.5. Ressources exposées en REST 12](#_Toc437292840)

[2.6. Configuration de Spring 12](#_Toc437292841)

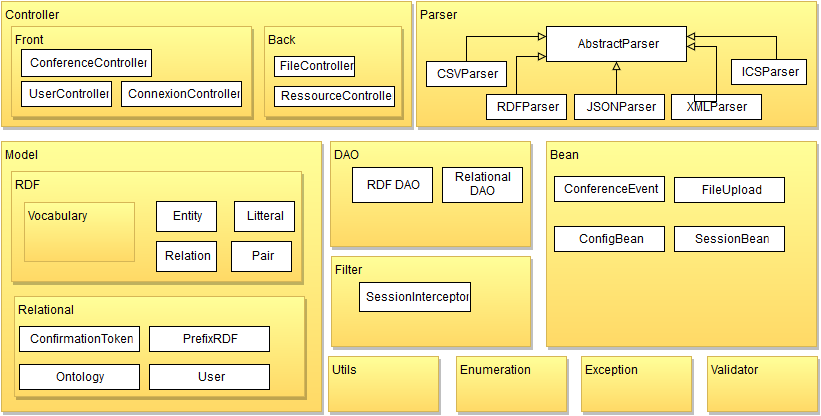
[3. Bilan 13](#_Toc437292842)

[3.1. Fonctionnalités implémentées 13](#_Toc437292843)

[3.2. Retour sur l’évaluation des risques 14](#_Toc437292844)

[3.3. Évolution de l’Architecture 14](#_Toc437292845)

# L’Architecture de l’Application



*Diagramme de package*

Les différents classes Java composant notre application ont été réparties dans différents package selon leur rôle.

Le package *model* contient toutes les classes dont les données sont sauvegardées en base de données. Par exemple les utilisateurs (*User*) ou les éléments d’une conférence (*Entity*). Les classes sont elles-mêmes réparties dans deux sous-package, l’un contenant les classes dont les données sont stockées dans la base de données relationnelle et l’autre les classes dont les données sont stockées dans la base de données RDF.

Le package *dao* contient les classes responsables des transactions avec les différentes bases de données, qui sont elles-mêmes réparties en deux catégories, les DAO dialoguant avec la base de données relationnelle et les DAO dialoguant avec la base de données RDF.

Le package Controller contient les différents contrôleurs Spring dont le rôle est de traîter les données envoyées via les requêtes du serveur.

Le package *controller* contient les différents contrôleurs. Les contrôleurs sont divisés en classe et sous-packages pour une meilleure organisation.

Le package *parser* contient les différents modules de parsing, un pour chaque format de fichier qui peut être importé via notre application. Chaque module de parsing hérite de la classe *AbstractParser* car ils ont tous besoin d’avoir accès aux mêmes objets pour fonctionner, comme les DAO ou la session en cours. En revanche, ils sont parfaitement indépendants car les traitements à effectuer sont très différents en fonction du type de fichier importé.

Le package *bean* contient toutes les classes dont les instances sont destinées à être des « beans ».

Le package *filter* contient les filtres permettant d’effectuer des traitements après et avant requête.

Le package validator contient les validateurs permettant de valider les formulaires soumis par l’utilisateur.

Le package e*numeration* contient les différents Enum dont nous avons eu besoin au cours du développement de l’application.

Le package *exception* contient les exceptions personnalisées que nous avons dû mettre en place pour améliorer la sécurité et la gestion d’erreur de l’application.

Le package u*tils* contient nos classes utilitaires. Ces classes sont généralement composées de méthodes statiques et permettent de rendre des services précis à d’autres composant de l’application, évitant de dupliquer inutilement certaines portions de code.

Etant donné le nombre de packages présents dans notre application, nous avons choisi de ne pas représenter les relations entre packages dans le diagramme pour ne pas le rendre illisible mais plutôt de montrer les relations entre les différents composants de notre application dans le schéma ci-dessous.



Un de nos objectifs a été de faire une application modulaire, et ce, pour plusieurs raisons :

* Pendant le développement cela facilite la répartition des tâches et le travail en parallèle.
* Cela facilite l’utilisation de module déjà existant et le remplacement ou l’évolution de ceux que l’on développe.

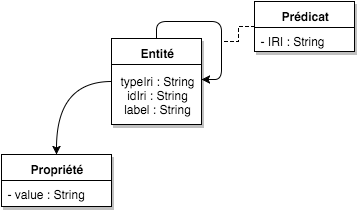
Etant donné que l’application se base sur le Framework Spring, elle comporte un élément dispatcher qui intercepte toutes les requêtes et dont le rôle est de redistribuer les requêtes aux contrôleurs appropriés.

Le module WebApp représente les vues et ressource statiques de notre application, on peut voir sur le schéma que ces pages web sont uniquement reliées aux contrôleurs et n’interagissent par exemple pas directement avec les DAO pour respecter la séparation des principes.

# Réalisation technique

## Les décisions prises

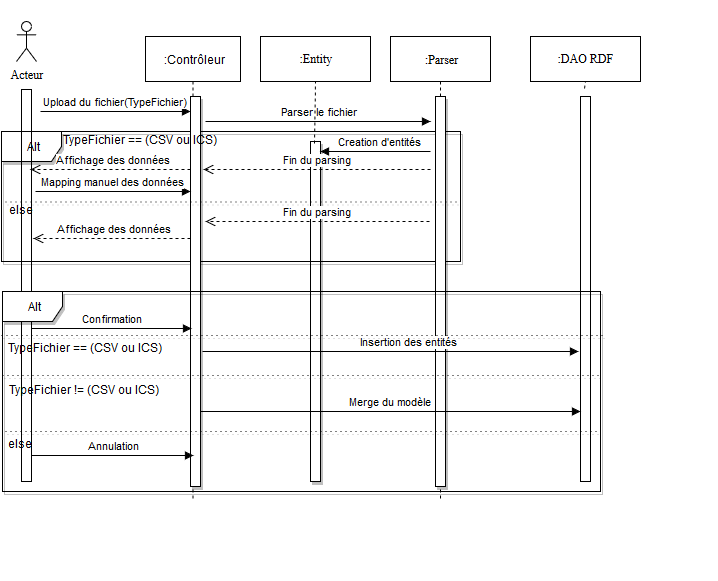
### Représentation des éléments d’une conférence



Pour représenter les différents éléments pouvant composer une conférence (Personne, Evènement, Publication), nous avons choisi de réaliser une implémentation très générique. Cette généricité nous permet d’alléger considérablement l’architecture de notre modèle car si nous avions voulu concevoir une implémentation plus proche de la réalité, nous aurions dû créer beaucoup de classes très similaires. Elle rend également l’application beaucoup plus flexible car il n’est par exemple pas nécessaire de créer une nouvelle classe pour gérer un type d’élément qui nous est inconnu, il suffira simplement de rajouter de nouvelles informations dans la base de données.

Chaque entité est identifiée par une IRI unique, un type dont le nom apparaît dans l’IRI et un label.Le nom est décrit par le prédicat *RDFS:label* et le type par le prédicat *RDF:type* dont nous avons imposé la non nullité. Dans notre modèle Java, ces trois attributs sont fixés. En revanche, les autres relations et propriétés de nos ressources pouvant être très différentes en fonction du type, elles sont stockées, pour chaque entité, dans deux collections contenant des paires de chaînes de caractères symbolisant le prédicat et la ressource/valeur.

### Import des fichiers



*Diagramme de séquence d’import de fichier*

Ce diagramme décrit le processus d’import de fichier, les interactions entre les différentes classes et les actions de l’utilisateur au cours de ce processus.

On voit sur ce schéma que chaque type de fichier fait l’objet d’un processus d’import différent, cela est dû au fait que les fichiers ne contiennent pas du tout le même type d’information selon leur format.

En effet, les fichiers ICS ne décrivent que des éléments de type évènement tandis que les fichiers CSV peuvent décrire d’autres types d’éléments, mais un seul type d’élément par fichier. Les fichiers de type XML/RDF ou JSON/LD contiennent quant à eux des informations beaucoup plus complètes, décrivant des éléments de types variés et les relations entre ces éléments. Chaque fichier a donc besoin d’un traitement spécifique en fonction des données qu’il contient car ces informations ne sont pas toutes insérées de la même façon dans la base de données.

C’est pourquoi il nous a semblé important de mettre en place un module de parsing différent par format de fichier plutôt qu’un seul module de parsing effectuant un traitement très compliqué avec de nombreux branchements conditionnels.

De plus, cela nous permet facilement de gérer de nouveaux types de fichier. En effet, il suffira simplement de greffer un nouveau module sur ceux déjà existants.

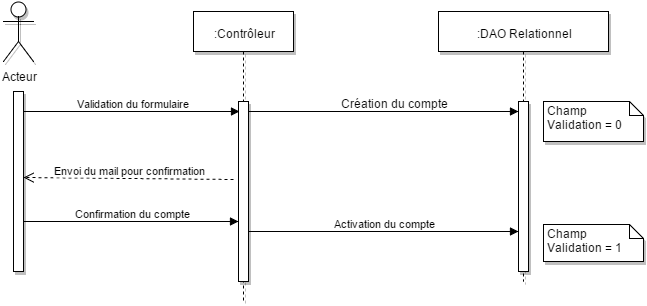
### Compte et authentification

La création d’un compte utilisateur se passe en deux étapes. La première est le remplissage du formulaire d’inscription avec les informations usuelles sur l’utilisateur. La validation de ce compte déclenche la génération d’un compte « inactif » et l’envoi d’un mail à l’utilisateur avec un lien lui permettant de valider son adresse mail et donc d’activer son compte. Pour sécuriser cette validation, le lien envoyé à l’utilisateur contient une clé unique qui est générée à chaque nouvelle inscription.

Il était essentiel pour nous de procéder à cette validation d’adresse mail. En effet, notre système de rôles et d’association des utilisateurs aux personnes des conférences sont basés sur l’unicité des adresses mail.

Le processus de création du compte est représenté dans le digramme ci-dessous.

Génération d’une clé unique



*Diagramme de séquence de la création de compte*

Avec son compte activé, l’utilisateur peut désormais se connecter et bénéficier de toutes les fonctionnalités de l’application.

La gestion de l’authentification est très simple. Nous avons créé un bean *SessionBean* qui est propre à chaque utilisateur tant qu’il est actif sur l’application (connecté ou non). Lorsqu’il se connecte, nous rajoutons cette information dans le *SessionBean*. Dès lors notre application peut, pour chaque requête, déterminer quel utilisateur a effectué la requête et s’il est connecté.

### Gestion des rôles

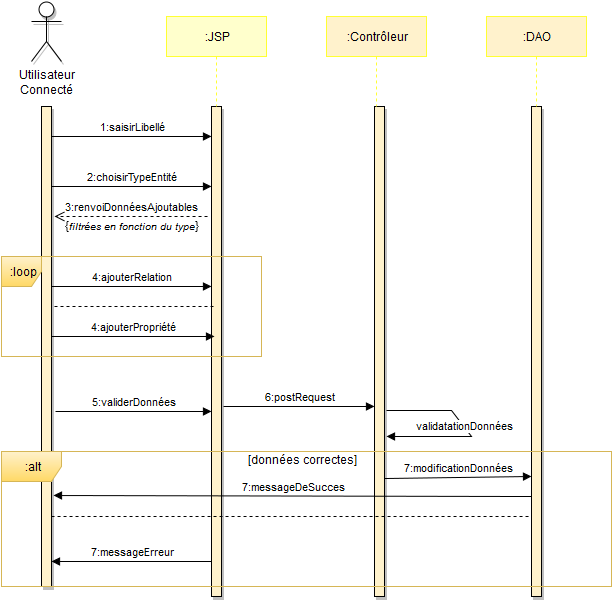
Les autorisations que possèdent les utilisateurs au sein de l’application ne peuvent pas être déterminées simplement en regardant s’il est connecté ou non. En effet, dans une conférence, il plusieurs catégories de personne ayant chacun des autorisations particulières sur des objets spécifiques.

Pour répondre à cette problématique, nous avons mis en place un système de rôle. Un utilisateur peut posséder plusieurs rôles et chaque rôle porte sur un objet en particulier. Nous avons par exemple le rôle *chair* qui donne à un utilisateur tous les droits de création/modification/suppression sur un événement. Ou encore *auteur* qui autorise la modification d’un champ *plus d’info* sur ses publications.

Ces rôles sont représentés dans la base de donnée RDF par un triplet <*personne*, *rôle*, *objet*> où l*’objet* est une conférence, une publication ou n’importe quelle ressource nécessitant des droits particuliers en fonction de la personne.

Dans certains cas, un rôle peut ne pas directement lier une personne à un objet. Par exemple, le *chair* d’un événement possède tous les droits sur les objets liés à cet événement. On distingue donc deux types de droit, ceux directs et ceux transitifs. Dans le premier cas, il nous suffit de regarder la personne qui possède le rôle correspondant à l’opération demandée sur l’objet. Dans le cas d’un rôle transitif, il nous faut parcourir tous les objets « parents » de celui concerné, pour y trouver ce même rôle.

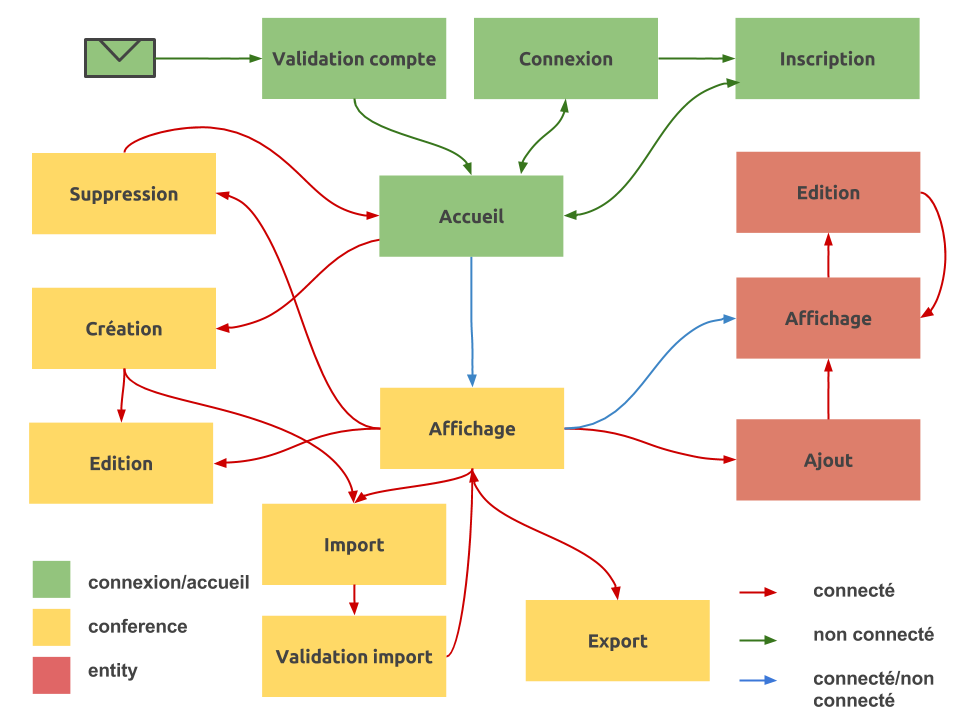
## Création d’entités



*Diagramme de séquence de création d’une entité*

Ce diagramme présente la création d’une entité par un utilisateur connecté. L’autre précondition pour créer une entité est que l’utilisateur doit être chair de l’évènement dans lequel va créer l’entité. Pour ce faire, l’utilisateur doit saisir obligatoirement un libellé, puis le type de l’entité à ajouter. La liste des propriétés et relations relatives au type d’entité lui apparaît. Une fois le type sélectionné, l’utilisateur peut ajouter autant de propriété et de relation à cet entité qu’il le désire. Puis, il valide la création de l’entité. Si les données sont correctes, alors un message de succès lui est renvoyé, sinon, un message d’erreur l’invitant à ressaisir de bonnes informations lui est affiché.

## Schéma de navigation

****

*Schéma de navigation*

Ce diagramme représente les liens entre les différentes pages et ressources de l’application.

Il s’agit d’une version simplifiée et un code couleur permet de simplifier sa lecture. Les pages ont été divisées en trois catégories :

* Vert pour les pages inhérentes à l’application en générale
* Jaune pour les pages relatives à une conférence
* Rouge pour les pages correspondant à une ressource spécifique

Les flèches représentent des actions de l’utilisateur, chaque couleur correspond à un type d’utilisateur :

* Rouge pour l’utilisateur connecté à l’application
* Vert pour un utilisateur non-connecté
* Bleu pour un utilisateur connecté ou non-connecté

A partir de la page d’accueil n'importe quel utilisateur pourra parcourir les conférences et les entités de la conférence. Un utilisateur non-connecté a accès à la fenêtre de connexion et de création de compte. Lorsque l’utilisateur créé son compte, il est redirigé vers la page d’accueil qui lui indique qu’un email lui a été envoyé. S’il clique sur le lien contenu dans son mail, il arrivera sur la ressource de validation de compte qui le redirigera vers la page d’accueil en le connectant.

Un utilisateur connecté peut créer une conférence, il peut ainsi saisir le nom de sa conférence et préciser s’il souhaite importer les données définissant sa conférence ou les rentrer à la main. Selon le choix il sera dirigé vers la fenêtre d’édition de conférence ou vers celle d’import de données. La fenêtre d’importation lui demande le format du fichier à importer et des champs supplémentaires sont ajoutés selon le format. Après avoir chargé le fichier l’utilisateur est redirigé vers la page de validation des données afin qu’il puisse vérifier leur cohérence avant qu’elles soient persistées.

L’utilisateur connecté a aussi la possibilité d’éditer et de créer des entités au sein d’une conférence.

## Design patterns utilisés

* **Inversion de contrôle/Injection de dépendances**

Le pattern IOC est le cœur de Spring. Partie intégrante du pattern IOC, l’injection de dépendance propose une gestion automatique du cycle de vie des instances d’objets. Ce pattern s’utilise via annotation @Autowired. Nous l’avons notamment employé pour les DAO, le *SessionBean*, et *ConfigBean*.

* **DAO**

Le principe de ce design pattern est de créer des objets, les DAO, responsables de la persistance. Ce sont ces objets qui se chargeront de faire communiquer le modèle et la base de données.

* **MVC**

Le modèle de données et la vue, matérialisée par les JSP, sont complètement indépendants. Cette séparation est possible à l‘aide les différents contrôleurs Spring mis en place dans l’application. Toutes les requêtes en provenance d’une vue sont interprétées par le contrôleur associé et c’est lui qui effectuera les traitements sur le modèle.

* **Factory**

Nous utilisons ce pattern dans les Dao pour créer des objets à partir des *factory* de Jena.

Ex : ResourceFactory.*createResource*(entityIri);

* **Singleton**

Nous utilisons ce pattern implicitement grâce aux annotations @Repository et @Component pour instancier la majorité de nos beans de manière unique. C’est le cas notamment pour les DAO et les parseurs.

* **Proxy**

Ce pattern utilise la programmation orienté aspect (AOP) pour l’interception des appels de méthodes. Ce pattern est très utile pour la configuration des beans qui ont besoin de s’injecter un autre bean qui possède un scope différent.

Nous l’avons par exemple utilisé avec notre *SessionBean* (scope session) qui avait besoin d’être injecté dans des DAO (scope application). C’est donc un objet proxy qui est injecté à la place du bean et qui, pour chaque appel de méthode sur ce proxy, va le transmettre au bon bean. Il fait donc office de multiplexeur.

* **Front Controller**

Ce design pattern permet de faire passer toutes les requêtes vers un dispatcher qui s’occupera de les rediriger vers le contrôleur approprié. Dans notre application, il est mis en place dans le fichier web.xml.

* **Helper**

Pour mutualiser certains traitements répétitifs, le pattern *helper* est utilisé pour créer des classes “utilitaires”. Ces classes sont composées de méthodes statiques destinées à rendre des services précis à d’autres composants de l’application.

Par exemple, la classe *PasswordUtil* permet d’effectuer facilement le cryptage des mots de passe dans la classe *UserDaoImpl.*

## Ressources exposées en REST

Toutes les ressources sont exposées en HTML et en JSON.

Récupération d’une ressource :

http://<hostname>/ressource/<nom\_conference>?iri=<iri\_ressource>

Récupération de toutes les ressources d’un type spécifique :

http://<hostname>/ressource/<nom\_conference>?type=<iri\_du\_type\_d’ontologie>

La représentation de ces ressources dépend de l’entête HTTP *Accept. (*HTML si égal à *text/html*, JSON sinon).

## Configuration de Spring

Nous avons fait le choix de déporter un maximum de configuration dans les annotations Java plutôt que dans les fichiers XML. Cela nous permet de simplifier grandement les fichiers de configuration et de faciliter la compréhension et la modification de celle-ci par les autres développeurs. On distingue ainsi, les beans “de gestion” (DataSource, TransactionManager, etc.), qui restent configurés en fichier XML, des beans “métier” pour lesquels les annotations apportent des simplifications.

Nous avons déclaré notre configuration Spring dans un fichier *servlet-context*. Dans celui-ci nous retrouvons les éléments principaux suivant :

* <**context:component-scan base-package="...” /**> : Indique à Spring quels packages il va devoir scanner à la recherche de classe avec des annotations.
* <**mvc:interceptors … /**> : Déclaration d’un filtre qui s’occupe d’injecter notre SessionBean dans le contexte de chaque requête afin qu’il soit accessible dans toutes les JSP.
* <**context:property-placeholder /**> : Permet d’utiliser très simplement les fichier *.properties.*
* <**bean … />** : Déclaration de bean. Nous retrouvons notamment ceux pour la gestion de la persistance relationnelle. Par exemple, le bean *Datasource* permet de définir les informations de connexion à la BDD.
* <**tx:annotation-driven /**> : Active la gestion automatisée des transactions par Spring pour les méthodes annotées avec @Transactional.
* <**mvc:annotation-driven**/> : Active le support des certaines annotations telles que @Controller.

# Bilan

## Fonctionnalités implémentées

|  |  |
| --- | --- |
| **Fonctionnalité** | **Statut** |
| Création de Compte | Développé |
| Connexion au compte local | Développé |
| Connexion via Facebook | Développé |
| Connexion via Twitter | Non développé |
| Utilisation de Gravatar | Non développé |
| Création d’une Conférence | Développé |
| Edition/Suppression d’une conférence | Développé |
| Import des données | Développé |
| Export du dataset | Développé |
| Visualisation des données et Navigation | Développé |
| SparQL EndPoint | Non développé |

Au terme du développement de notre application, celle-ci remplissait un certain nombre des fonctionnalités requises dans le cahier des charges. Cependant, nous avons manqué de temps pour implémenter les fonctionnalités restantes. Nous étions d’ailleurs conscients, dès les premiers jours du projet, que le temps nous manquerait et que nous devrions faire des choix concernant les fonctionnalités à implémenter.

C’est pourquoi nous avons décidé que certaines d’entre elles, telles que la connexion via Twitter ou l’utilisation de Gravatar pour les comptes utilisateur serait développées uniquement s'il nous restait du temps, d’autant plus que leur absence ne nuisait pas au fonctionnement global de l’application, et lui permettait quand même de remplir son but premier, à savoir la conception du dataset.

Nous n’avons pas seulement manqué de temps mais aussi d’expérience et de pratique, notamment dans l'utilisation du Framework Jena. En effet, notre gestion des droits utilisateur n’est pas optimale et possède quelques limites car nous avons eu du mal à comprendre comment extraire certaines informations de la base de données RDF. Nous nous sommes par exemple rendus compte trop tard qu’il existait un mécanisme de Jena qui nous aurait beaucoup facilité la mise en place de cette gestion de droit. Si nous avions eu plus d’expérience dans l’utilisation de ce Framework, nous serions directement partis dans la bonne direction.

## Retour sur l’évaluation des risques

Parmi les risques que nous avons évoqués dans le cahier des charges, l’un d’eux a été mal évalué. En effet, nous étions conscients que, ne connaissant pas beaucoup le Framework Spring, nous aurions besoin de temps pour comprendre comment l’utiliser. Nous ne pensions en revanche pas que cela nous ferait aussi perdre du temps sur la mise en place de la base de données. En effet, pour interagir avec la base de données, nous utilisons l’ORM Hibernate que nous avons déjà tous utilisé mais la mise en place de cet ORM dans une application avec ou sans Spring se fait de façon de façon très différente et nous avons ainsi passé du temps à configurer correctement notre application pour le faire fonctionner.

## Évolution de l’Architecture

Finalement, notre modélisation du cahier des charges était juste puisqu’il n’y a pas beaucoup de variations entre l’architecture prévue et l’architecture effectivement mise en place. Seul le module de parsing nous a imposé de revenir sur nos décisions. En effet, nous avions considéré la possibilité de créer une classe abstraite *Parser* et d’implémenter le design pattern *Factory* pour gérer les différents parseurs correspondant aux différents formats de fichier à importer. Cependant, dans la pratique, les parseurs ont besoin d’informations trop disparates pour pouvoir passer par une Factory et nous avons dû les gérer de façon indépendante.