

**Rapport MIF18**

MultiMIF 2015-2016

**

**Symp’le Event Manager**

Membres du Projet  
Jordan MARTIN Jules SAUVINET Alix GONNOT Landry Mélaine LEBATTO Tom DUSSEAUX Clément BLAISE

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

Sommaire

[1. La base de données relationnelle 3](#_Toc437077682)

[1.1. Les comptes utilisateurs 3](#_Toc437077683)

[1.2. Les ontologies 4](#_Toc437077684)

[2. La base de données RDF 6](#_Toc437077685)

[2.1. La modélisation d’une conférence 7](#_Toc437077686)

[2.2. Les données d’une conférence 7](#_Toc437077687)

[2.3. L’import des données 9](#_Toc437077688)

[2.3.1. Le format ICS 10](#_Toc437077689)

[2.3.2. Le format CSV 10](#_Toc437077690)

[2.3.3. Les formats « RDF » 10](#_Toc437077691)

L’application développée au cours de ce projet MultiMIF a pour but de permettre la conception d’un ensemble de données (dataset), à partir d’informations provenant de sources diverses. Ces données seront ensuite fournies à l’application Sympozer, un outil d’organisation d’évènements.

La conception d’un ensemble de données est complexe, celle-ci mettant en jeu l’import de données de formats hétéroclites, ou l’ajout de données via des interfaces de création d’entités. La construction du dataset sera une intégration continue de données fournies par différents utilisateurs.  Il est donc essentiel que l’application soit capable de persister les données et d’être flexible aux changements**.**

Pour pouvoir participer à la construction de l’ensemble des données, un utilisateur doit être **connecté**. C’est à dire que l’application doit pouvoir identifier cet utilisateur et posséder des informations à son sujet de façon **persistante** également. Ainsi, notre application est amenée à manipuler et stocker des données de nature très différentes (i.e. les utilisateurs, les évènements auxquels ils participent, etc.).

Nous avons choisi de stocker ces différents ensembles de données dans deux types de base de données distincts car nous avons des utilisations et des contraintes différentes sur la manipulation de celles-ci.

Les données liées au fonctionnement de l’application sont persistées dans une base de données de type **relationnel** et les données “métier” sont quant à elle stockées dans une base de données de type **RDF (Resource Description Framework).**

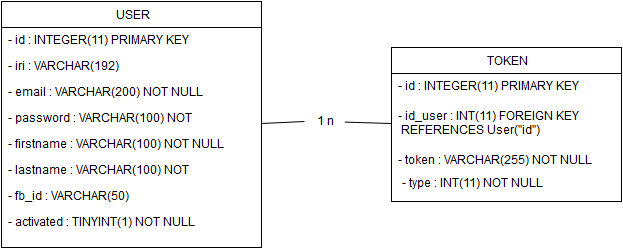
# La base de données relationnelle

## Les comptes utilisateurs

Nous avons choisi de stocker les données relatives aux comptes utilisateur dans une base de donnée relationnelle pour plusieurs raisons.

Tout d’abord, une base de données relationnelle est largement suffisante pour l’utilisation que nous comptons faire de ces données. Il nous a semblé inutile de représenter ces informations dans une base RDF, car les possibilités offertes par ce type de base de données ne seront jamais exploitées pour ce cas. En effet, le schéma de données est très simple puisqu’un compte utilisateur peut presque se résumer à un ensemble de chaînes de caractères. De plus, il n’existe pas de relations complexes entre objets dans ce schéma. Ainsi, l’utilisation d’une base de données RDF nous complexifierait inutilement la tâche.

Ensuite, les données représentant un compte utilisateur doivent respecter un certain nombre de contraintes. Par exemple, une même adresse mail ne peut pas être utilisée par plusieurs comptes. Or les bases de données relationnelles offrent des mécanismes permettant de s’assurer que ces contraintes sont respectées.



La table *User* contient les champs suivants :

* **id :** un identifiant unique, auto-généré lors de l’insertion en base de données et stocké sous forme d’entier.
* **email :** une adresse mail renseignée par l’utilisateur, stockée sous forme de chaîne de caractères. Elle doit être unique dans la base de données.
* **password :** le mot de passe renseigné par l’utilisateur, stocké sous forme de chaîne de caractères. Le mot de passe sera représenté par un hash (SHA1 + grain de sel) par sécurité, on ne doit pas pouvoir lire ce mot de passe “en clair” dans la base de données.
* **lastname :** le nom de famille renseigné par l’utilisateur, stocké sous forme de chaîne de caractères.
* **firstname :** le prénom renseigné par l’utilisateur, stocké sous forme de chaîne de caractères.
* **fb\_id :** l’identifiant du compte Facebook que l’utilisateur souhaite lier à son compte, stocké sous forme de chaîne de caractères.
* **activated :** un booléen qui code le fait qu’un utilisateur a activé son compte ou non.

Contrairement aux autres champs, le champ *fb\_id* peut être nul. Un utilisateur n’est pas obligé de lier son compte Facebook à son compte local.

La table *Confirmation\_token* contient les champs suivants :

* **id :** un identifiant unique, auto-généré lors de l’insertion en base de données et stocké sous forme d’entier.
* **id\_user :** l’identifiant de l’utilisateur associé au token.
* **token :** la chaine de caractères représentant le token.
* **type :** un entier codant le type du token (0 = activation compte, 1 = oubli de mot de passe, etc).

## Les ontologies

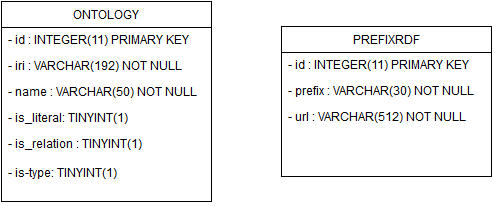
Outre les données relatives aux comptes utilisateurs, il nous a aussi fallu représenter dans une base de données relationnelles certaines informations un peu plus liées au “métier” de l’application. En effet, Nous verrons plus bas que les données liées aux conférences sont stockées dans une base de donnée de type RDF et que dans cette base interviennent des ontologies. Ces éléments permettent de typer les différentes ressources et de qualifier leurs relations.

Comme ces ontologies sont représentés dans la base RDF par des IRI, qui sont des chaînes de caractères longues et peu compréhensible, il était nécessaire de stocker la signification de ces IRI quelque part, de façon à pouvoir afficher ces informations de façon lisible dans les interfaces graphiques.

Nous avons également stocké les préfixes RDF de façon à pouvoir les intégrer dans le dataset. En effet, si le modèle Jena, qui représente un graphe RDF, ne connaît pas les préfixes utilisés dans ses triplets, et que l’on génère le dataset du modèle, il contiendra des préfixes auto-générés. Ainsi, nous obtenons un dataset avec des préfixes lisibles (ex : FOAF, SWC, …).

Nous avons représenté ces informations dans la base de données relationnelles pour plusieurs raisons :

Le premier est que nous voulions complétement isoler nos données métiers de nos données applicatives (nous reviendrons sur ce point dans la partie RDF). Deuxièmement, nous voulions avoir un modèle évolutif sur lequel il est facile de rajouter des contraintes ou des informations supplémentaires.



La table *Ontology* contient les champs suivants :

* **id :** un identifiant unique, auto-généré lors de l’insertion en base de données et stocké sous forme d’entier.
* **iri :** l’IRI représentant l’élément (ex : <http://xmlns.com/foaf/0.1/Person>) stockée sous forme de chaîne de caractères
* **name :** le nom en français du prédicat ou du type correspondant à l’IRI  (ex : Personne) stocké sous forme de chaîne de caractères.
* **is\_literal :** booléen qui indique si l’iri correspond à un prédicat pour une propriété
* **is\_relation :** booléen qui indique si l’iri correspond à un prédicat pour une relation
* **is\_type :** booléen qui indique si l’iri correspond à un type

La table *PrefixRdf* contient les champs suivants :

* **id :** un identifiant unique, auto-généré lors de l’insertion en base de données et stocké sous forme d’entier.
* **prefix :** le nom du préfixe RDF (ex : SEM, FOAF) stocké sous forme de chaîne de caractères
* **url :** la valeur du préfixe RDF (ex : [http://univ-lyon1.fr/sem#](http://univ-lyon1.fr/sem)) stockée sous forme de chaîne de caractères.

Pour ce modèle relationnel, nous avons décidé d’utiliser le SGBD **MySQL**. Notre choix s’est rapidement porté sur ce SGBD car des membres du projet possédaient déjà de bonnes connaissances sur celui ci. De plus, ce SGBD possède de nombreux outils pour faciliter sa manipulation (*phpmyadmin* ou logiciel natif par exemple).

Ainsi, nous avons pu mettre en place notre base de données relationnelle très rapidement.

Pour manipuler la base de données via notre application et effectuer des requêtes, nous avons utilisé l’API **JPA** (avec les annotations) pour faire le mapping entre nos classes Java et nos tables relationnelles. Nous avons utilisé l’ORM **Hibernate** comme implémentation de JPA. Notre choix s’est orienté vers Hibernate pour sa très bonne intégration avec Spring (gestion de la configuration, injection des dépendances,etc.) et parce que nous avons tous déjà mis en place un mapping avec celui-ci.

# La base de données RDF

Les données “métier” de notre application sont représentées dans une base de données orientée graphe. Nous avons fait ce choix pour plusieurs raisons.

Tout d’abord, les relations entre les différentes entités d’une conférence sont assez complexes. Il existe des liaisons bidirectionnelles entre plusieurs d’entre elles (e.g. une personne participe à une conférence et une conférence possède un ensemble de personnes participantes). Si nous avions choisi de stocker ces informations dans une base de données de type relationnel, le schéma des données aurait été très complexe et les requêtes à effectuer coûteuses. Nous aurions eu le choix entre une implémentation très générique (presque sous forme de triplet finalement) et une implémentation figée et donc plus difficilement évolutive et peu flexible. Or les données que l’on manipule peuvent être très hétérogènes dans leurs attributs et relations entre elles.

Ensuite, ce choix nous a grandement facilité le développement des fonctionnalités d’import et d’export du dataset. En effet, notre application doit être capable de fournir à l’utilisateur une représentation en JSON/LD du dataset, format compréhensible par l’application Sympozer. Comme nos données sont déjà stockées dans un schéma RDF et que nous utilisons le Framework Jena, il n’est pas nécessaire de les convertir “manuellement” dans un nouveau format avant de les exporter car l’API Jena RIOT en permet une génération directe à partir de notre modèle. En effet, comme nous l’avions évoqué dans la partie relationnelle, les données présentent dans cette base RDF sont exclusivement des données métier ce qui nous permet de pouvoir exporter un dataset qui représente parfaitement notre base RDF. C’est un avantage non négligeable pour la cohérence des données lors d’un import.

Pour la persistance de ce modèle RDF, nous avons utilisé l’API **TDB** (TriplestoreDataBase) fournie par le Framework **Jena**.  Cette API nous permet d’avoir un stockage natif des données sous forme de triplets (sujet - prédicat - objet) dans des fichiers locaux. Cette API est plus performante que l’API SDB (qui elle exploite une base relationnelle).

## La modélisation d’une conférence

Notre application doit être en mesure de gérer des conférences. Une conférence est un type d’évènement correspondant à un ensemble de sous évènements. Chaque sous évènement peut également contenir un ensemble d’évènements.

Une conférence est représentée par un graphe qui correspond à un modèle Jena. Nous avons choisi de créer un dataset par conférence et de n’utiliser que le modèle par défaut de Jena. Cette solution nous permet de pouvoir exporter les données relatives à une conférence sans faire de filtrage puisque toutes les données du dataset sont alors renvoyées lors de l’export d’une conférence.

Les conférences étant indépendantes les unes des autres, nous avons opté pour cette solution qui simplifie la gestion des conférences et ne surcharge pas le dataset s’il doit être amené un jour à contenir un volume plus important de données.

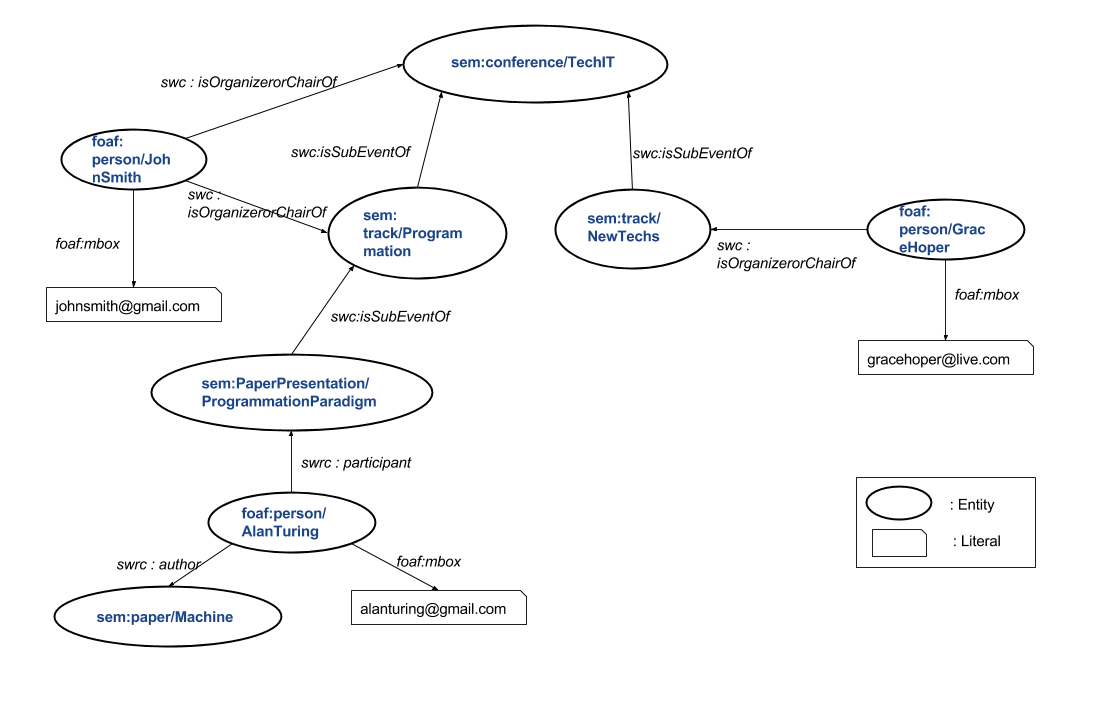
Cette solution implique que les données soient parfois dupliquées, on peut imaginer une personne qui participerait à plusieurs conférences par exemple, mais ce n’est pas très gênant car cela permet d’éviter les conflits de modification qui pourraient se poser avec une gestion à un seul modèle pour plusieurs conférences. En effet, avec cette solution, si une personne est supprimée du modèle par un Chair d’une des conférences alors qu’elle doit aussi être présente dans une autre conférence, on se retrouve avec des données incohérentes et les utilisateurs se gêneraient donc mutuellement.

## Les données d’une conférence

Nous nous sommes imposés une représentation des données très générique. Cette décision s’est avérée cruciale. En effet, les objets que nous seront amenés à manipuler sont très variés, il est très compliqué de proposer une implémentation complète. Ils peuvent évoluer au cours du temps, de nouveaux peuvent être ajoutés aux ontologies, voire créés par des utilisateurs.

Nous voulions avant tout mettre en place une base solide et flexible qui pourrait, sans que l’on ait besoin de revoir entièrement le modèle, être adaptable à d’autres domaines. Notamment au niveau des conférences, que l’on puisse gérer une nouvelle hiérarchie d’évènements avec d’autres contraintes que celles définies par les ontologies SWC et SWRC.

Ce choix a compliqué notre modélisation des ressources. En revanche, les structures de données que nous pouvons importer ou ajouter sont plus étendues que si nous avions figé notre modèle. Le modèle ainsi conçu est plus évolutif et flexible.

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.

Ci-dessus est représenté un exemple très simplifié de graphe pouvant être modélisé par notre structure de donnée. Les prédicats *RDFS:label* et *RDF:type* ont été omis afin d’assurer plus de lisibilité, sachant que ceux-ci doivent être présent pour chaque nœud.

Il existe deux méthodes d’insertion de données dans une conférence. La première consiste à importer des données d’un fichier externe. Selon le format du fichier, les données seront traduites en entités avant d’être insérées dans le modèle JENA ou bien directement converties en un modèle JENA. La seconde consiste à créer de nouvelles entités directement depuis l’interface de l’application.

Au niveau de la création de données via l’interface graphique, il n’est possible d’ajouter qu’une seule entité à la fois. Celle-ci peut contenir autant de relations et de propriétés que l’utilisateur veut créer. Toutefois, l’ensemble des relations que l’utilisateur souhaite ajouter doivent pointer vers des ressources déjà existantes. Si l’utilisateur souhaite ajouter deux entités ayant au moins une relation entre elles, il devra les créer séparément, puis les éditer pour mettre en place les liaisons. Nous n’avons pas trouvé de solution immédiate et réalisable dans les contraintes de temps données afin de satisfaire cette problématique.

Mis à part cette contrainte, tous les chaînages entre entités sont réalisables et quand l’utilisateur choisit un prédicat à ajouter correspondant à une relation, un filtrage est effectué sur le type de la ressource à faire correspondre. Par exemple, si l’utilisateur souhaite créer une personne et lui ajouter le prédicat *FOAF:knows*, il aura accès à l’ensemble des personnes déjà existantes en base. En revanche, l’utilisateur est responsable de la cohérence des données qu’il veut insérer pour les propriétés puisqu’elles pointent vers des littéraux.

Les droits d’un utilisateur dans une conférence sont représentés par les rôles possédés par les personnes dont l’adresse mail correspond à celle de l’utilisateur. Il est d’ailleurs obligatoire de fournir une adresse mail unique lors que la création d’une personne dans une conférence. A chaque accès à une ressource, on effectue une vérification pour savoir si l’utilisateur correspond à une personne de la conférence et si oui, on cherche s’il possède des droits de modifications pour pouvoir lui proposer les bonnes options.

La hiérarchie des évènements est représentée à l’aide du prédicat *isSubEventOf*. Avec ce prédicat, il est possible de parcourir les évènements père à partir d’un nœud évènement. Cela nous permet aussi de définir les droits CRUD sur les ressources. Par exemple, si une personne est chair de l’évènement courant (représenté par le prédicat *isOrganizerorChairOf* qui relie la personne et l'évènement), nous lui attribuons les droits de modification sur l’évènement, sinon nous parcourons les évènements “parent” pour savoir si la personne est chair d’un de ces évènements et lui attribuons les droits en conséquence.

Pour les entités autres que les évènements, il suffit de voir dans quel évènement cette entité est contenue puis d’effectuer la même recherche afin de déterminer les droits CRUD sur la ressource.

En l’état actuel des choses, pour savoir si un utilisateur peut modifier une autre personne par exemple, on parcourt les relations de la personne pour chercher si elle est reliée à un évènement sur lequel l’utilisateur possèderait des droits de modification. Ce système fonctionne si l’on utilise uniquement des prédicats qui vont des Person aux Events pour symboliser les relations entre ces deux types d’entités et est donc très sensible à l’utilisation qui est faite des prédicats. Il peut donc trouver ses limites, notamment sur des données qui aurait étés importées via des fichiers.

On aurait pu résoudre ce problème en utilisant le moteur d’inférence de Jena car celui-ci nous aurait permis de détecter beaucoup plus facilement les relations entre les ressources et donc gérer plus efficacement les droits utilisateurs.

## L’import des données

L’utilisateur peut importer des données aux formats RDF/XML, JSON-LD, ICS, CSV ou XML avec une feuille de style XSLT qui transformera son fichier XML au format RDF.

### Le format ICS

Un fichier ICS contient des données relatives à des évènements. Ci-dessous, un exemple d’évènement décrit selon le vocabulaire ICS.

BEGIN:VEVENT  
DTSTART:20151204T140000Z  
DTEND:20151204T170000Z

LOCATION:Batiment Nautibus  
SUMMARY:Soutenance Multimif

DESCRIPTION:Chaque groupe de multimif présentera son application

CATEGORIES:Soutenance  
END:VEVENT

Les champs DTSTART, DTEND, SUMMARY, LOCATION, et DESCRIPTION sont reconnus par notre parser ICS et automatiquement associés à des propriétés d’une ressource de type évènement. Ce traitement est suivi d’une étape de validation pendant laquelle l’utilisateur doit définir le type de chaque évènement décrit dans le fichier (TalkEvent, DemoEvent,etc.).

### Le format CSV

Un fichier au format CSV doit décrire un ensemble d’entités du même type. Comme pour le format ICS, l’import passe par une étape de validation au cours de laquelle l’utilisateur doit choisir le type d’entité contenu dans le fichier (Person, TalkEvent, Track, etc.) ainsi que la signification de chaque colonne (Label, Nom, Email, etc.) avant de pouvoir ajouter ces informations dans la base de données.

### Les formats « RDF »

Notre choix de base de donnée s’est retrouvé conforté ici puisque l’interface Jena RIOT offre des méthodes de parsing pour les fichiers aux formats XML, RDF, et JSON.

En l’état actuel des choses, il est nécessaire que les formats XML et JSON soient construit avec les ontologies SWC, SWRC et FOAF pour pouvoir bénéficier de la gestion des droits. Néanmoins, il est possible de représenter n’importe quelle autre ontologie.