Sérialisation

Comment on sauve ? On fait quoi des gros modèles?

Points abordés

- Gros modèles -> séparations en différentes resources
- XMI/JSON fichier
  - format récursif
  - format par références
- Base de donnée relationnelle
- Base de donnée NOSQL
- Requêtes efficaces sur les gros modèles
- Comment charger un modèle (de quoi a-t-on besoin ?)

Sérialisation ? C’est quoi pour vous ?

Principe

La sérialisation est un procédé qui permet de rendre un modèle **persistant** pour stockage ou échange et vice versa. Ce modèle est mis sous une forme sous laquelle il pourra être reconstitué à **l'identique**. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer ailleurs.

C'est le procédé qui est utilisé par les beans pour sauvegarder leurs états.

L'opération inverse qui consiste à créer une nouvelle instance à partir du résultat d'une sérialisation s'appelle la désérialisation.

Sauver et charger

La sérialisation d'un modèle permet d'envoyer dans un flux les informations sur le type et l'état des éléments qui le compose pour permettre de le récréer ultérieurement.

- Algorithme de sérialisation

Si plus d'une référence sur un objet alors il n'est sérialisé qu'une fois pour réduire le stockage

Problèmes généraux

- les éléments sérialisés font généralement référence à d'autres éléments qui doivent aussi être sérialisés en même temps pour permettre de maintenir l'état de leurs relations.
- la gestion des références dans un graphe d'objets sérialisés surtout si un même objet est référencé plusieurs fois dans le graphe
- elle est généralement peu performante car elle utilise l'introspection
- son utilisation peut engendrer des problèmes de sécurité : sérialisation de données sensibles, désérialisation d'un objet dont la source est inconnue, ...

Quoi sauvegarder ? Des idées ?

Que doit-on sauvegarder ?

Un modèle peut rapidement contenir de nombreux objets

➡ Réduire au maximum à tout ce qui ne peut pas être calculé

Par exemple :

- les attributs ou références calculés (préfixés avec un “/”)
- pour les références bidirectionnelles sauvegarder un côté suffit
- ne pas sauver les valeurs par défauts des attributs
- ne pas sauver les types parfois si information facilement retrouvable dans le métamodèle

Sérialisation dans un fichier

La sérialisation d’un modèle dans un fichier est un moyen de sauver un modèle et de le partager.

Afin de pouvoir échanger des modèles sous forme de fichier facilement, il est nécessaire d’avoir un format commun, donnant suffisamment d’information et compris par tout le monde.

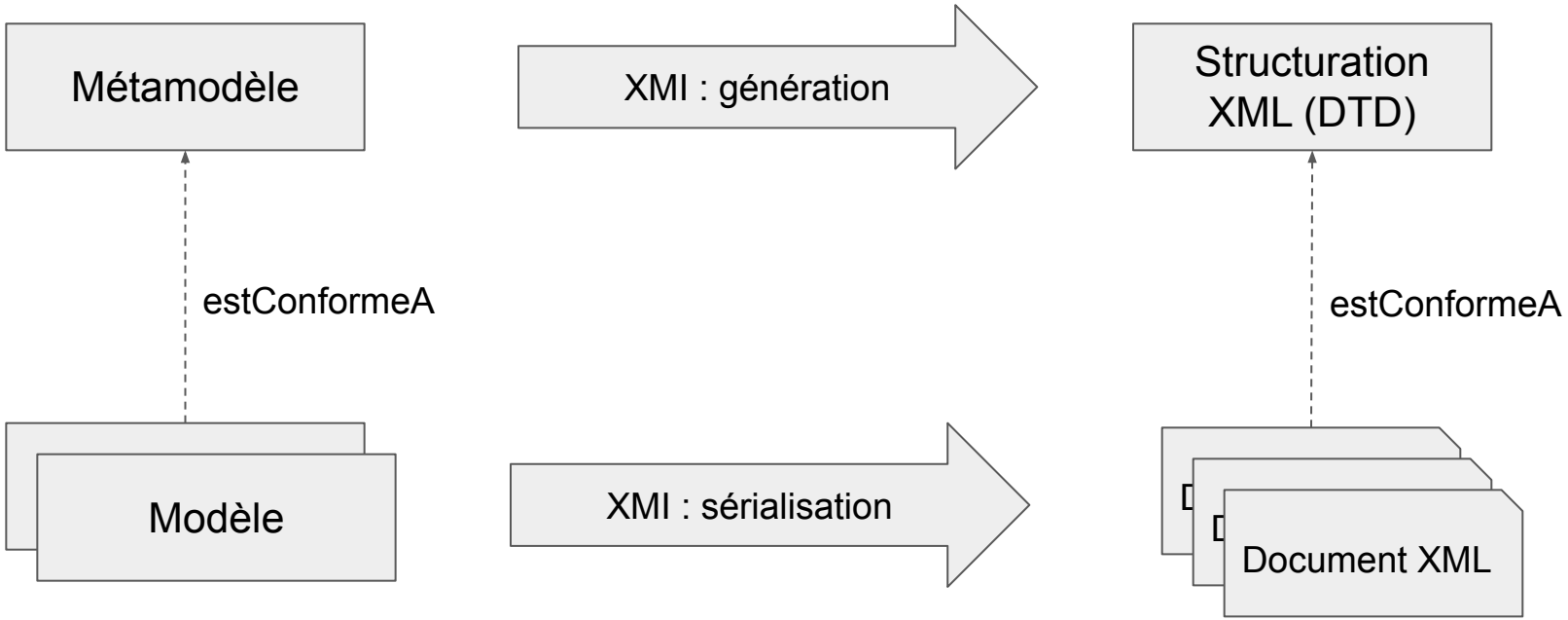
Par défaut, il y a 2 formats textuels bien acceptés dans la communauté : le format XMI (XML) et EMFJson (JSON).

Sérialisation de modèles et XML

Moyen : description de modèles avec XML

- XML (eXtensible Markup Language) du W3C :
  - Format pour la représentation textuelle de données struct.
  - Utilisation de balises < *xxx* > ... < /*xxx* >
- Grammaire définissant la validité d'un document XML :
  - DTD (Document Type Definition) :
    - Relations d'inclusion entre balises
  - XML Schema :
    - Doc XML définissant la structuration de documents XML
- XMI (XML Metadata Interchange) de l'OMG :
  - Standard pour représenter des (meta-)modèles au format XML
    - Sérialisation de modèles
  - Utilisation des métamodèles pour définir les DTD / Schema
    - Génération automatique des DTD / Schema

Mécanisme


Métamodèle
XMI : génération
Structuration
XML (DTD)
estConformeA
estConformeA
Modèle
XMI : sérialisation
Document XML

Récursif vs à plat

Récursif :

- Une racine
- La récursion suit l'arbre de composition ou les références
- Les autres références se font par identifiant
- Avantages :
  - pratique si on reste dans l'espace de récursion
  - reproduit la composition des éléments
- Inconvénients :
  - algorithmes de sérialisation et désérialisation complexes à écrire
  - gestion différente des relations composite et non-composite

A plat :

- Pas de racine
- Pas d'imbrication, tout au même niveau
- Toutes les références se font par identifiant
- Avantages :
  - Accès facile à n'importe quel type d'élément
- Inconvénients :
  - Besoin de parcourir plusieurs fois le fichier pour savoir qui contient quoi
  - Très verbeux

Format XMI

XML Metadata Interchange, (XMI) :

- un standard créé par l'Object Management Group (OMG) pour l'échange d'informations de métadonnées basé sur XML.
- utilisable pour toutes métadonnées dont le métamodèle peut être exprimé en Meta-Object Facility (MOF).
  - L'usage le plus commun de XMI est l'échange de modèles UML, mais peut aussi être utilisé pour la sérialisation de modèles d'autres langages (métamodèles).
- Tentative pour profiter des avantages de UML (et de sa définition dans MOF) et de XML

Exemple de format XMI

```
<?xml version="1.0" encoding="ASCII"?>
<Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="ghmde">
  <users id="77096" name="louismrose">
    <repositories name="Epsilon-CLI" stargazers="1" watchers="1" forks="1" size="5656">
      <files path="test/org/eclipse/epsilon/cli/test/logging/DoesNotParse.eol" technologies="//@technologies.17">
        <commits date="2011-04-16T16:10:50Z" author="//@developers.0"/>
      </files>
      ...
```

Éléments stockés récursivement

Référence non-composite vers un objet via un chemin

Format JSON

JavaScript Object Notation (JSON) est un format de données textuelles

Un document JSON comprend deux types d'éléments structurels :
- des ensembles de paires « nom » (alias « clé ») / « valeur »
- des listes ordonnées de valeurs

Avantages :
- peu verbeux, donc lisible aussi bien par un humain que par une machine ;
- facile à apprendre, et à modifier car syntaxe réduite ;
- ses types de données sont connus et simples à décrire.

Inconvénients :
- peu de types généraux, besoin de passer par des sérialisation custom pour certains types primitifs
- pas extensible contrairement à XML limité dans le cas des MM
- pas de commentaire

Exemple de format JSON récursif

```
{
  "eClass": "http://www.eclipse.org/emf/2002/Ecore#//EPackage",
  "name": "pack",
  "nsURI": "http://test/1.0",
  "nsPrefix": "pack",
  "eClassifiers": [
    {
      "eClass": "http://www.eclipse.org/emf/2002/Ecore#//EClass",
      "eStructuralFeatures": [
        {
          "eClass": "http://www.eclipse.org/emf/2002/Ecore#//EAttribute",
```

Format MSE

MSE est un format à plat pour la sérialisation de modèles.

Visuellement, il est assez proche d'une représentation typée lisp ou encore JSON sous forme d'atomes. Il possède principalement les mêmes avantages que JSON.

Avantages :

- peu verbeux, donc lisible aussi bien par un humain que par une machine ;
- facile à apprendre, et à modifier car syntaxe réduite ;
- ses types de données sont connus et simples à décrire.

Inconvénients :

- peu de types généraux, besoin de passer par des sérialisation custom pour certains types primitifs
- pas extensible contrairement à XML limité dans le cas des MM

Exemple de format MSE (à plat)

```
((FamixJava.Namespace (id: 1)
    (name 'aNamespace'))
  (FamixJava.Package (id: 201)
    (name 'aPackage'))
  (FamixJava.Package (id: 202)
    (name 'anotherPackage')
    (parentPackage (ref: 201)))
  (FamixJava.Class (id: 2)
```

Format alternatif - JSOI

JSOI est un format alternatif au algos récursif et à plat. Le but est de considérer un maximum d'informations directement écrites dans le fichier sérialisé et de stocker les objets "par types" plutôt que suivant les relations composites. Il est aux croisés entre algos à plat et un stockage en base de donnée.

L'avantage d'un tel format et de faciliter le "lazy-loading" des objets, tout en permettant de trouver de l'information sur la structure générale des modèles sans pour autant charger tout le modèle en mémoire.

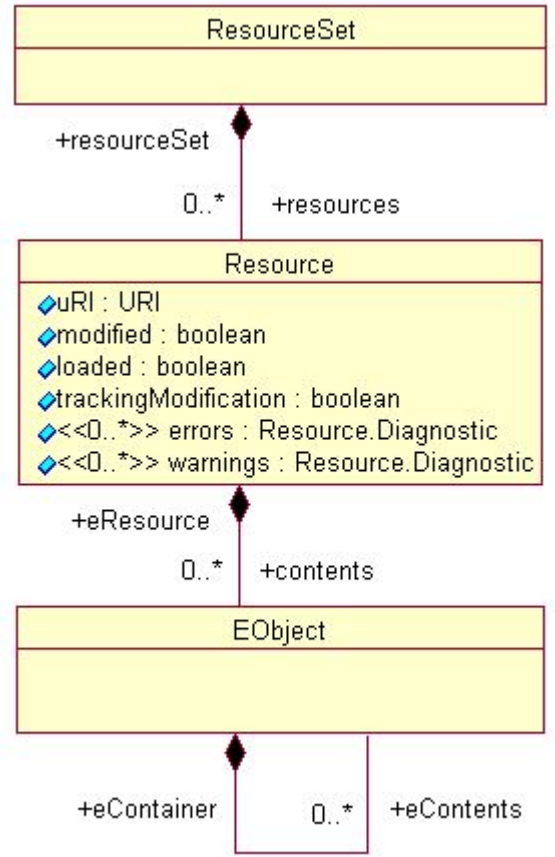
Le désavantage principal et sa complexité d'implémentation.

Exemple de format JSOI

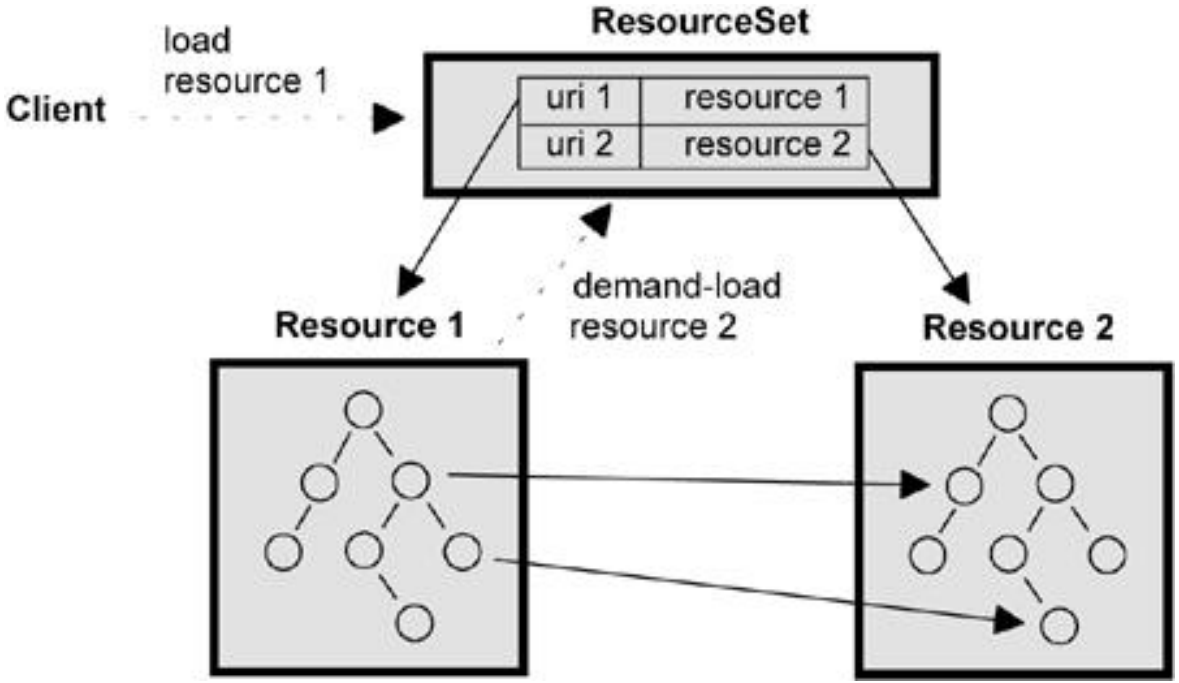
```
{
    ...
      "railway:RailwayContainer": {
        "size": 1,
        "elements": [ {
            "routes": [
                "jsoi:$.railway:Route.elements[0]"
            ],
            "regions": [
                "jsoi:$.railway:Region.elements[0]"
            ]
        }]
      },
```

Resource et ResourceSet

- EMF et PyEcore possèdent un concept de “Resource” et de “ResourceSet”
- Un “ResourceSet” représente une collection de “Resources”
- Une “Resource” représente techniquement un fichier ou autre entité qui contient un modèle
- Une “Resource” contient une ou plusieurs racines



Resource et ResourceSet



Obtenir une “Resource”

```java
ResourceSet rset = new ResourceSetImpl();
Resource resource = rset.getResource(URI.create("PATH_TO_XMI"));

EObject root = resource.getContents().get(0);
```

```python
rset = ResourceSet()
resource = rset.get_resource('PATH_TO_XMI')

root = resource.contents[0]
```

Créer une nouvelle “Resource”

```java
EObject obj = ...; // Une racine quelconque

ResourceSet resourceSet = new ResourceSetImpl();
Resource resource = resourceSet.createResource(URI.create("PATH_TO_XMI"));
resource.getContents().add(obj);
```

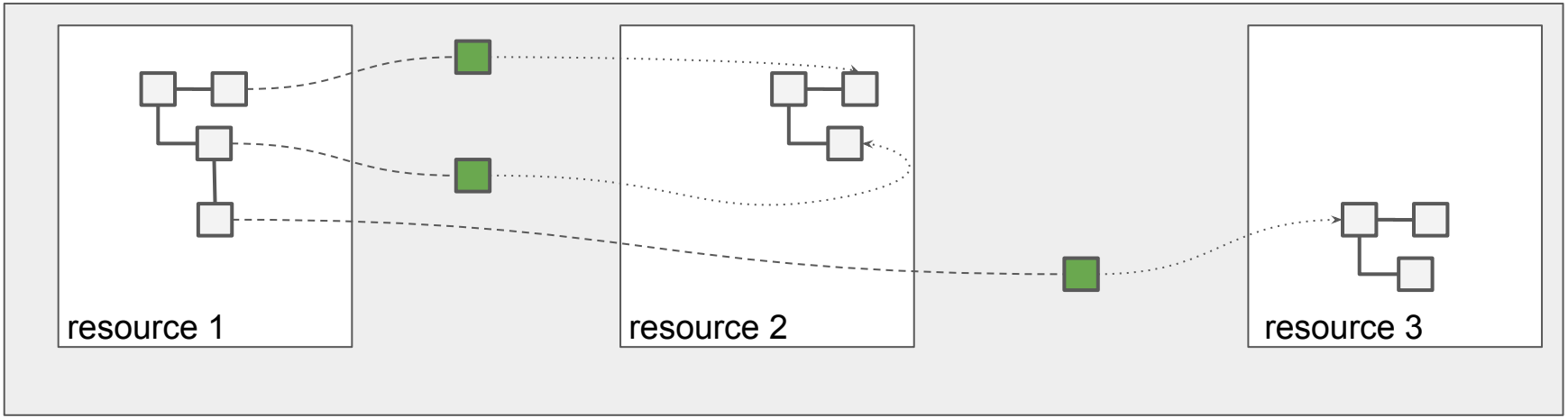
```python
obj = ...  # Existing resource

rset = ResourceSet()
resource = rset.create_resource('PATH_TO_XMI')
resource.append(obj)
```

Multiples ressources et proxies

On peut partager un modèle sur plusieurs resources. Ça permet de gérer les gros modèles et avoir des données qu'on peut charger de façon modulaire grâce à des proxies.

Les proxies sont généralement résolus à l'usage (lazy-loading)

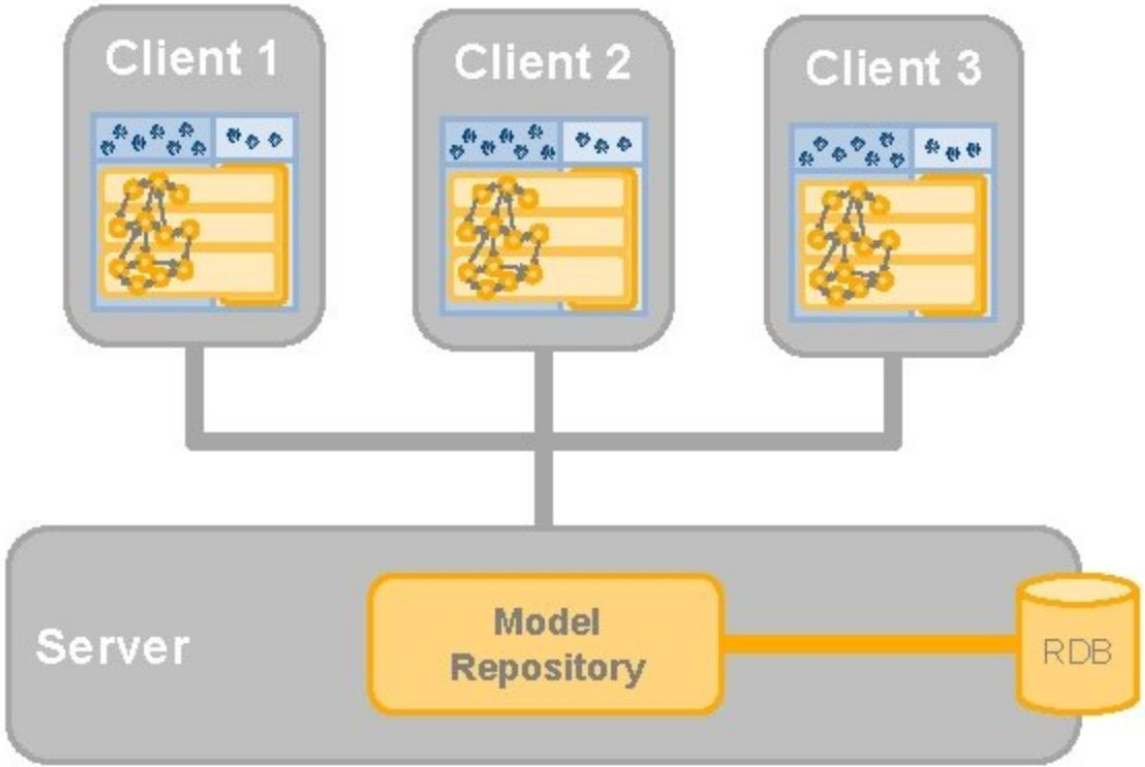


BDs vs fichiers

Même si la représentation d'un modèle en base de données est lourde, elle propose beaucoup d'avantages :

- avoir des gros modèles qui ne tiennent pas forcément en mémoire
- charger des portions de modèles uniquement
- stabilité du support (outils pour le maintien de la BD)
- support des transactions (possible plus ou moins sans BD, mais complexe)

Exemple - CDO



Sauvegarde dans une base de données relationnelle

Avantage :

- peut contenir un très grand nombre d'éléments
- permet des interrogations rapides et efficaces
- existence d'un schéma

Inconvénients :

- Problèmes d'efficacité si requêtes avec beaucoup de jointures
- Difficulté de traitement des relations récursives
- Difficulté de gestion des relations opposites
- Difficulté de gestion du polymorphisme

Sauvegarde dans une base de données relationnelle

Conséquence aux problèmes soulevés :

- la manière dont les éléments sont sauvés dans la base sont très différents de la manière dont les éléments sont représentés en mémoire !

Sauvegarde dans une base de données relationnelle

- Chaque type ou élément du métamodèle correspond à une table
- Chaque référence entre ces éléments correspond à une clé étrangère si relation 1-1 ou nouvelle table si *-*
- Chaque objet ou élément du modèle correspond à une ligne dans une table
- Chaque référence entre ces objets correspond à une clé étrangère ou une ligne dans une table.
    - Attention à l'ordre dans lequel les éléments sont ajoutés dans la base : il n'est pas possible de référencer quelque chose qui n'existe pas
- Le polymorphisme est en général géré par une entrée spéciale dans la table

Base de donnée NoSQL

Le NoSQL permet la dénormalisation.

En particulier, il permet de manipuler des objets structurés (contenant d’autres objets)

Différents types de bases de données NoSQL (document, clé-valeur, graphe…)

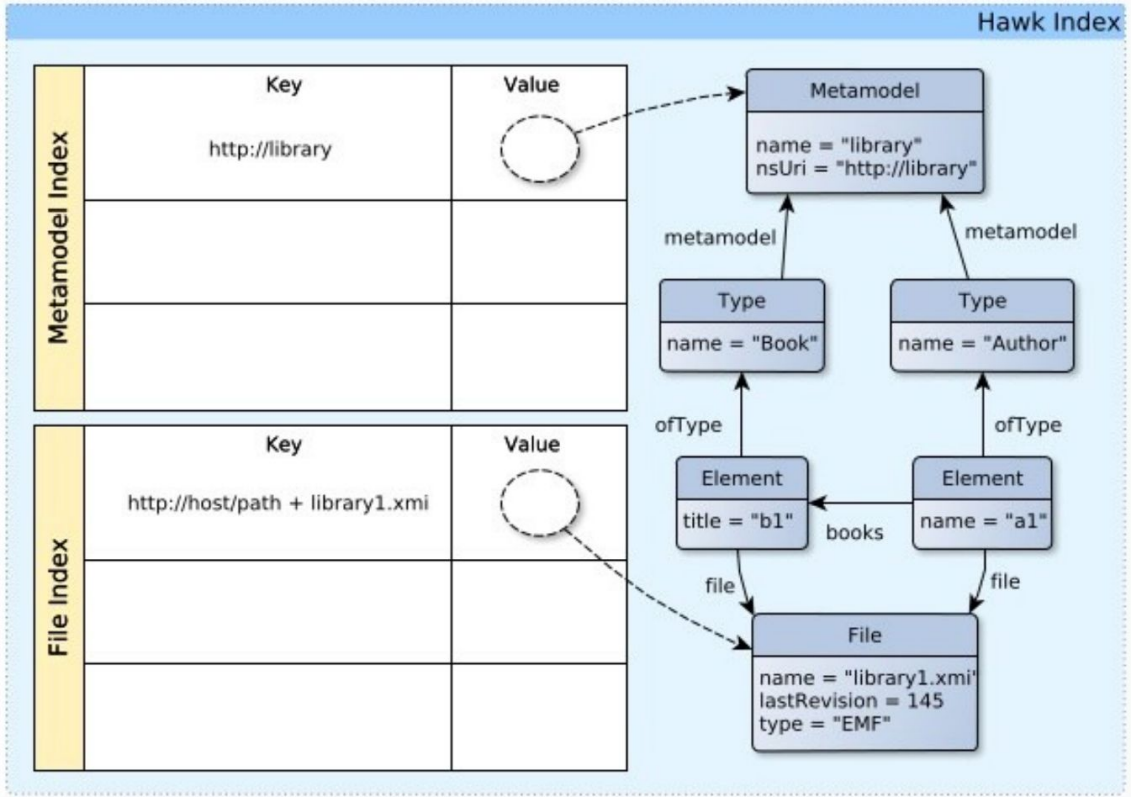
Le MM sert de schéma à la base de données.

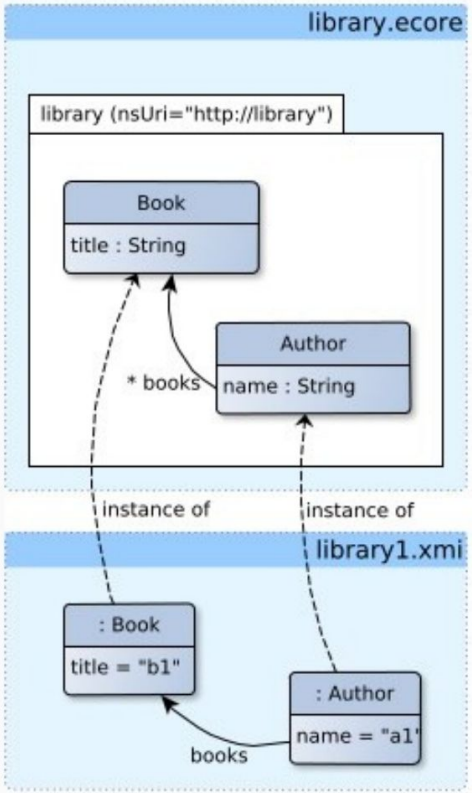
Quelles sont les problématiques ?

- Les bases de données NoSQL n'impose pas l'utilisation d'un schéma de base

Il est nécessaire, si on veut embarquer toutes les informations d'un modèle dans la BD, de stocker à la fois le modèle et le méta-modèle.

Exemple - Hawk



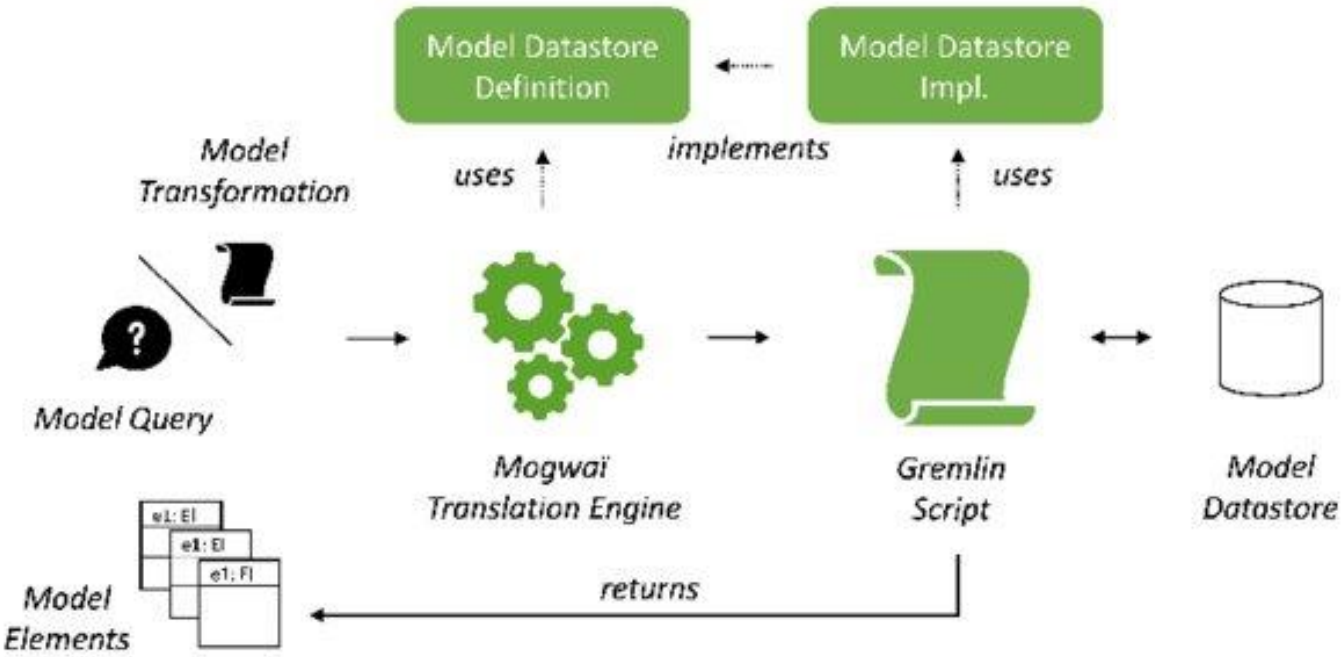


Navigation et requêtes sur gros modèles

Les bases de données sont très intéressantes pour pouvoir conserver des modèles commes elles proposent des moyens rapides pour questionner les données et sont taillées pour traiter des données extrêmement grosses.

- Pour pouvoir questionner un modèle, on peut passer par SQL (par exemple), mais comme un modèle est peut-être stocké de manière différente de sa représentation connue par les humains, alors il faut “traduire” les requêtes OCL en requête SQL spécifique

Navigation et requêtes sur gros modèles



Charger un modèle ?

Les problématiques de chargement de modèles sont transversales à la sérialisation. Nous ne traiterons pas exactement de comment ces algorithmes sont implémentés.

- Comment pensez-vous pouvoir effectuer un chargement de modèles ?
- De quoi a-t-on besoin si on charge un modèle sérialisé “à plat” et “récursivement” ?
- S’il y a plusieurs méta-modèles en jeux, comment les différencier ?

Ce que vous devez savoir faire

- Connaître les mécanismes de sérialisation
- Connaître différents formats de sérialisation avec leurs avantages et inconvénients
- Comprendre comment se fait la sérialisation et le chargement d’un modèle