Sérialisation XML avec JAXB

Master Technologies de l'Internet 1ère année

Eric Cariou

Université de Pau et des Pays de l'Adour
UFR Sciences Pau – Département Informatique

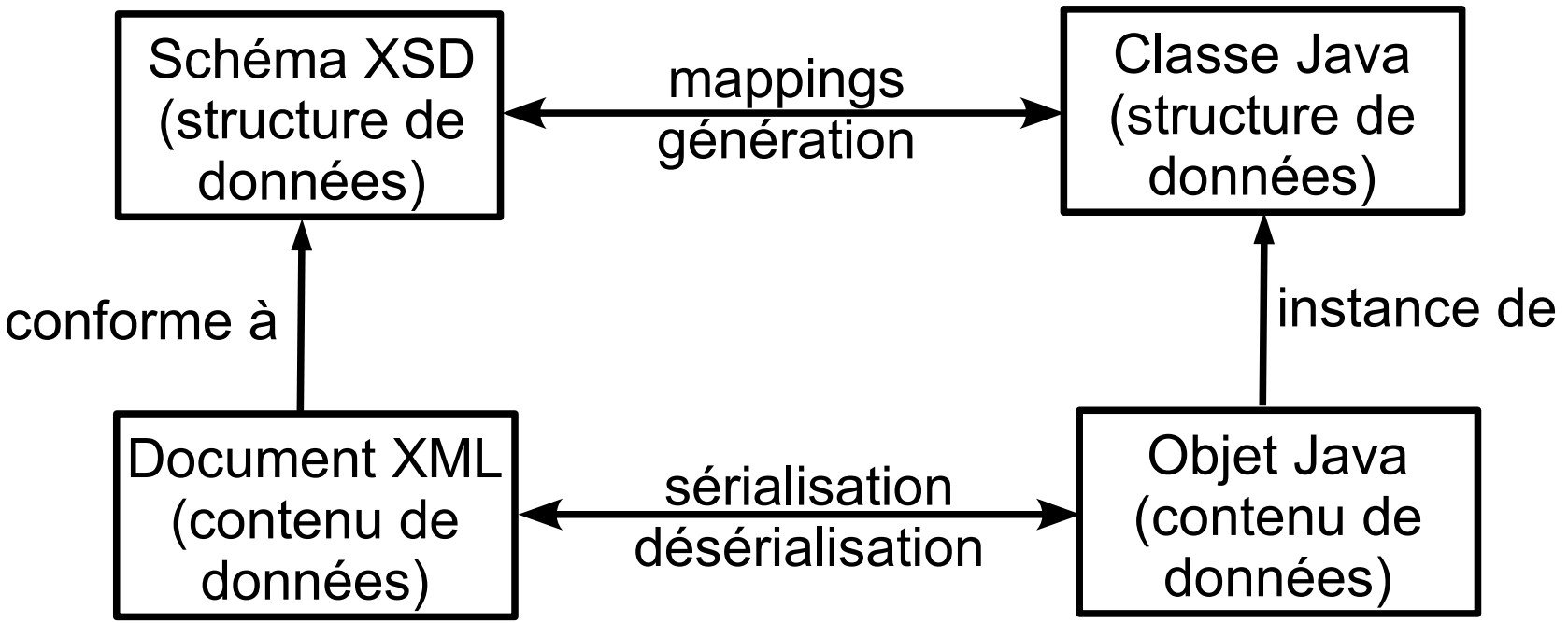
Eric.Cariou@univ-pau.fr

Octobre 2016

Sérialisation XML

- Stockage de données dans des fichiers XML
- JAXP
  - Java Architecture for XML Processing
  - Permet de parcourir un fichier XML en passant de nœud en nœud
  - Accès bas niveau, doit faire la correspondance à la main entre le contenu XML et les objets Java ainsi que le parcours du XML soi même
- JAXB
  - Java Architecture for XML Binding
  - Correspondance structure XML / structure classes
    - Via annotation des classes
  - Lecture directe d'objets à partir d'un contenu XML

Relations générales



- Le schéma XSD est optionnel
  - Peut déduire la structure du contenu XML (les données) à partir de la structure de la classe et inversement

Principes JAXB

- Package javax.xml.bind
- Deux modes de mappings
  - Annoter des classes Java existantes pour préciser comment/quoi sérialiser en XML
  - Générer un ensemble de classes Java à partir d'un schéma XML
- Dans le code de l'application
  - Utilisation de « marshaller », « unmarshaller » pour écrire ou lire le contenu d'objets dans du XML
  - Transparence complète vis-à-vis de la structure XML, ne manipule que des objets

Annotations JAXB

- Quelques annotations
  - @XmlRootElement : à placer sur une classe pour la définir comme le contenu principal/premier d'un fichier XML
  - @XmlTransient : sur un getter
    - Si on ne veut pas que le contenu de l'attribut soit sérialisé
    - Par défaut, tout attribut d'une classe est sérialisé
  - @XmlType : gestion des types XML
    - Par exemple pour ordonner les champs dans le fichier XML
  - @XmlEnum : pour mapper une énumération
  - ...

Exemple : sport et discipline

- Un sport est composé de disciplines
  - On veut enregistrer un sport avec ses disciplines
  - Classe Sport du package donnees
    - // la classe Sport est l'élément principal
      @XmlRootElement
      // ordre de la sérialisation des attributs
      @XmlType(propOrder = {"codeSport", "intitule", "disciplines"})
      public class Sport implements java.io.Serializable {

        private int codeSport;
        private String intitule;
        private Set<Discipline> disciplines;

        // getter, setter, constructeurs...
    - Si on ne veut pas sérialiser les disciplines

      @XmlTransient
      public Set<Discipline> getDisciplines() {
        return disciplines;
      }

Exemple : sport et discipline

- Classe Discipline du package donnees
  - Rien de particulier à préciser, on la définit comme un POJO classique, sans annotations
  - public class Discipline implements java.io.Serializable {

```
private int codeDiscipline;
private String intitule;
private Sport sport;

// getter, setter, constructeurs...
```

Contexte

- JAXBContext
  - Classe qui définit un contexte JAXB à partir d'une liste de classes sérialisables
  - Méthode statique newInstance, deux versions principales
    - static JAXBContext newInstance(Class...)
      - Une ou plusieurs descriptions de classes
    - static JAXBContext newInstance(String package)
      - Nom d'un package dans lequel se trouvent des classes générées à partir d'un schéma XML
  - Avec un contexte, on récupère un « marshaller » ou un « unmarshaller »
    - Marshaller : permet d'enregistrer des objets Java en XML
      - public Marshaller createMarshaller();
    - Unmarshaller : permet d'instancier des objets Java à partir d'un contenu XML
      - public Unmarshaller createUnmarshaller();

Marshaller

- Classe Marshaller
  - void marshal(Object obj, support) : sérialise l'objet obj avec support pouvant être
    - File : descripteur de fichier
    - OutputStream / Writer : flux binaire / texte de sortie
    - XMLEventWriter / XMLStreamWriter : flux spécialisés XML
    - Node : arbre DOM
    - …
- Classe Unmarshaller
  - Object unmarshal(support) : désérialise l'objet à partir d'un support pouvant être
    - File : descripteur de fichier
    - InputStream / Reader : flux binaire / texte d'entrée
    - XMLEventReader / XMLStreamReader : flux spécialisés XML
    - Node : arbre DOM
    - …

Exemple de sérialisation

```
// création du contexte pour la classe Sport
JAXBContext jc = JAXBContext.newInstance(Sport.class);
// création du marshaller
Marshaller marshaller = jc.createMarshaller();
// positionnement d'une propriété pour que le XML soit formaté en sortie
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

// création d'un sport avec deux disciplines
Sport sp = new Sport(12, "Ski");
Discipline ds = new Discipline(10, "Descente", sp);
sp.getDisciplines.add(ds);
ds = new Discipline(11,"Biathlon", sp);
sp.getDisciplines.add(ds);

// sérialisation du sport dans un fichier sport.xml
marshaller.marshal(sp,new File("sport.xml"));
```

Exemple de sérialisation

- L'exécution du code précédent donne le fichier suivant

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<sport>
   <codeSport>12</codeSport>
   <intitule>Ski</intitule>
   <disciplines>
      <codeDiscipline>10</codeDiscipline>
      <intitule>Descente</intitule>
   </disciplines>
   <disciplines>
      <codeDiscipline>11</codeDiscipline>
      <intitule>Biathlon</intitule>
   </disciplines>
</sport>
```

Exemple de désérialisation

- Si maintenant on désérialise le fichier « sport.xml »
  - ```
    // création d'un unmarshaller
    JAXBContext jc = JAXBContext.newInstance(Sport.class);
    Unmarshaller unmarshaller = jc.createUnmarshaller();

    // récupère l'instance du sport dans le fichier XML
    Sport sp = (Sport)unmarshaller.unmarshal(new File("sport.xml"));
    System.out.println(" Sport " +sp.getIntitule());
    for (Discipline ds : sp.getDisciplines()) {
        System.out.println(" -> "+ds.getIntitule());
    }
    ```
  - Cela donne l'affichage suivant

    ```
    Sport Ski
    -> Descente
    -> Biathlon
    ```
    ```

Génération d'un schéma

- A partir de classes annotées, on peut générer le schéma XML correspondant
  - Outil « schemagen » en ligne de commande
    $ schmagen -cp . donnees/Sport.java
  - Donne le fichier XSD suivant

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="sport" type="sport"/>

    <xs:complexType name="sport">
        <xs:sequence>
            <xs:element name="codeSport" type="xs:int"/>
            <xs:element name="intitule" type="xs:string" minOccurs="0"/>
            <xs:element name="disciplines" type="discipline" nillable="true" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

     <xs:complexType name="discipline">
        <xs:sequence>
            <xs:element name="codeDiscipline" type="xs:int"/>
            <xs:element name="intitule" type="xs:string" minOccurs="0"/>
            <xs:element ref="sport" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Limites du mapping XML

- Le schéma généré est syntaxiquement valide mais sémantiquement faux
  - Un sport référence des disciplines et une discipline référence un sport : références croisées
  - Si on génère les classes Java (voir juste après) avec ce schéma et qu'on essaye de sérialiser un sport avec des disciplines, ça plante

    com.sun.istack.internal.SAXException2: A cycle is detected in the object graph. This will cause infinitely deep XML: Sport{codeSport=12, intitule=Ski} -> Discipline{codeDiscipline=11, intitule=Biathlon, sport=Sport{codeSport=12, intitule=Ski}} -> Sport{codeSport=12, intitule=Ski}]
  - Données en XML = arbre avec éléments imbriqués
    - Ne peut pas gérer une référence croisée entre deux éléments, les deux éléments se retrouveraient imbriqués mutuellement l'un dans l'autre
  - Solutions
    - Supprimer la référence vers un sport dans la classe Discipline
    - Ou marquer le getter du sport comme transient :

      ```
      @XmlTransient
      public Sport getSport() { return sport }
      ```

      Dans la balise XML définissant une discipline, ne génère alors plus la ligne `<xs:element ref="sport" minOccurs="0"/>`

Limites du mapping XML

- Retour sur exemple de sérialisation (transparents 10 à 12)
  - La classe Discipline a un attribut de type Sport
  - Dans les balises de disciplines, aucune référence vers leur sport
    - A la sérialisation, le moteur JAXB a ignoré de lui-même la référence croisée
    - … mais pas le générateur de schéma
  - Lors de la désérialisation d'un sport, en conséquence, l'attribut sport d'une discipline n'est pas positionné
  - ...

```
for (Discipline ds : sp.getDisciplines()) {
    System.out.println(" -> "+ds.getIntitule() + " de sport "+ds.sport);
}
```

donne maintenant

```
Sport Ski
-> Descente de sport null
-> Biathlon de sport null
```

Limites du mapping XML

- Format de représentation des données
  - Java : graphe d'objets avec des références entre objets
  - XML : arbre avec imbrication des éléments
- Quand on prévoit de sérialiser en XML
  - Faire attention à la structure des classes coté Java
  - Utiliser une structure à sérialiser formant un arbre

Génération de classes annotées

- Opération inverse, on génère des classes Java annotées à partir d'un schéma XSD
  - Exemple de schéma (similaire dans le principe au précédent avec un sport composé de disciplines)
  - Fichier « sport.xsd »

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="sport">
      <xs:complexType>
         <xs:sequence>
            <xs:element type="xs:byte" name="codeSport"/>
            <xs:element type="xs:string" name="intitule"/>
            <xs:element ref="discipline" maxOccurs="unbounded" minOccurs="0"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="discipline">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="codeDiscipline" type="xs:byte"/>
            <xs:element name="intitule" type="xs:string"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Génération de classes annotées

- Exécution de l'outil « xjc » en ligne de commande
  $ xjc -p donnees sport.xsd
  - Génère les deux classes Sport et Discipline

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {"codeSport","intitule","discipline"})
@XmlRootElement(name = "sport")
public class Sport {
    protected byte codeSport;
    @XmlElement(required = true)
    protected String intitule;
    protected List<Discipline> discipline;
…}
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {"codeDiscipline","intitule"})
@XmlRootElement(name = "discipline")
public class Discipline {
    protected byte codeDiscipline;
    @XmlElement(required = true)
    protected String intitule;
… }
```

Génération de classes annotées

- Une factory d'objets est également générée avec une méthode de création de chacune des classes métier générées
  - @XmlRegistry

```
public class ObjectFactory {

    public ObjectFactory() {
    }

    public Sport createSport() {
        return new Sport();
    }

    public Discipline createDiscipline() {
        return new Discipline();
    }
}
```

Utilisation de classes générées

- Les classes Sport et Discipline sont des POJOs classiques
  - On les utilise normalement
- Si on crée un contexte par rapport à un package
  - Le moteur JAXB s'attend à trouver une classe ObjectFactory pour gérer la création des objets
- Exemple pour le package « donnees »
  - `JAXBContext jc = JAXBContext.newInstance("donnees");`
  - Le reste du code de marshalling/unmarshalling ne change pas ensuite