UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
DOMENIUL DE LICENȚĂ INFORMATICĂ

REFERAT

Parsarea fișierelor XML folosind Java

STUDENT
Roșcăneanu George

BUCUREȘTI
Martie 2018

Cuprins

Introducere

Motivația

Am ales această temă deoarece am folosit personal Java pentru a parsa un fișier XML și mi-a plăcut funcționalitatea care a fost pusă la dispoziție din start în JDK, neavând nevoie să mai adaug alte librării.

Am fost foarte mulțumit de cât de ușor a fost să definesc elementele XML care doream să fie parsate, și de cât de repede reușește acest lucru.

Descrierea generală a infrastructurii JAXB

Java Architecture for XML Binding (JAXB) este o infrastructură software care le permite programatorilor să creeze clase Java care reprezintă nodurile dintr-un fișier XML.

Cele două facilități principale ale JAXB-ului sunt:

- **marshaling** : capabilitatea de a scrie obiecte Java în formatul XML
- **unmarshaling** : capabilitatea de a folosi un fișier(sau string, sau InputStream) XML ca să creeze obiecte Java

Această infrastructură este deja inclusă în Java SE Development Kit 8u161. Nu este necesară descărcarea unei librării secundare de parsare a fișierelor XML.

Cantitatea de cod necesară pentru a scrie clase care să fie folosite la *mashalling* și *unmarshalling* este relativ foarte mică în comparație cu implementările făcute de mână.

Viteza de procesare este de-asemenea foarte satisfăcătoare. De exemplu timpul necesar pentru a face *unmarshaling* pe un fișier XML de 111 KB unde sunt definite 29 de noduri unice și adâncimea cea mai mare al unei nod este 5, este de aproximativ 100-200 de ms (testat pe un calculator cu procesor Intel i5 4460).

În clasele Java, care vor fi folosite de către JAXB, pot fi definite în mod practic un număr nelimitat de atribute și subelemente ale elementului definit de către clasă.

Definirea unui element în Java

Elementele XML care sunt definite în Java au nevoie de ceva numit ***Java annotation*** (adnotare Java). Adnotările sunt texte care

```
@XmlRootElement
public class Customer {
    @XmlAttribute
    String name;
    @XmlElement
    int age;
    @XmlElement
    int id;
}
```

Figura 1

au la începutul lor simbolul **@**. JAXB se foloseşte de aceste adnotări definite de către programator ca să ştie cum sunt definite fiecare dintre subelementele şi atributele unui element XML.

De exemplu clasa *Customer* care a fost definită în Figura 1 poate fi folosită fie să genereze documentul XML definit în Figura 2, sau să parseze documentul definit în Figura 2 pentru a genera un obiect de tipul *Customer* definit în Figura 1.

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<customer name="Roscaneanu George">
    <age>24</age>
    <id>123456</id>
</customer>
```

Figura 2

Un lucru care poate fi imediat observat este într-adevăr cât de puțin cod a fost scris pentru definirea elementului.

La începutul clasei s-a adăugat adnotarea **@XmlRootElement**,care este folosită ca să anunțe JAXB-ul faptul că obiectele de tipul Customer pot fi parsate ca elemente XML *customer*.

O altă adnotare folosită, este **@XmlAttribute**, care este folosită pentru a defini un atribut pentru elementul *customer*. Ultima adnotare folosită este **@XmlElement**, care definește un subelement de-al lui *customer.*

```
@XmlRootElement
class Parent{
    @XmlAttribute
    String name;
    @XmlElement
    Child child;
}

@XmlRootElement
class Child{
    @XmlAttribute
    String name;
}
```

Figura 3

Un lucru de observat în legătura **@XmlRootElement** este de ce clasa cu numele *Customer* a fost scrisă în fișierul XML ca elementul *customer*(cu litera mică, nu literă mare).

Motivul de ce s-a întâmplat așa este pentru că atunci când adnotaţiile **@XmlRootElement**, **@XmlElement** şi **@XmlAttribute** sunt folosite în forma lor implicită, token-urile pe care sunt aplicate sunt automat convertite în camel-casing(prima literă este convertită implicit într-o literă mică).

Subelementele unui element pot fi de-asemenea şi ele la rândul lor alte clase Java care au fost adnotate cu **@XmlRootElement** ca în Figura 3. Cele două clase Parent și Child pot crea fișiere XML precum cele găsite în Figura 4

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<parent name="Daniel">
    <child name="Andreea"/>
</parent>
```

Figura 4

Definirea explicită a numelor elementelor și atributelor

Există situații când un XML trebuie să conțină nume care au explicit prima literă o majusculă sau, pur și simplu nu se poate definii un obiect sau clasă cu un cuvânt anume deoarece acela este un cuvânt cheie al sintaxei Java (de exemplu cuvântul *class*).

```
@XmlRootElement
class Teacher{
    @XmlAttribute
    String name;
    @XmlElement(name="class")
    ClassElement classElement;
}
@XmlRootElement(name="class")
class ClassElement{
    @XmlAttribute
    String name;
    @XmlElement(name="Student")
    Student student;
}

@XmlRootElement(name="Student")
class Student{
    @XmlAttribute
    String name;
}
```

Figura 5

În Figura 5 a fost definit elementul *Student*(atenție la faptul că *Student* este scris cu literă mare explicit), elementul *class*(cuvântul *class* face parte din lista de cuvinte cheie a limbajului Java, deci acesta a trebuit să fie și el declarat explicit) și elementul *teacher*.

Ce este interesant de observat, este pentru ca obiectele Java să fie transformate corect în elemente XML, atunci în clasa lor trebuie să definim până și subelementele lor cu numele explicite dorite.

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<teacher name="Popescu Dan">
    <class name="Informatica">
        <Student name="Roscaneanu George"/>
    </class>
</teacher>
```

Figura 6

În Figura 6 se poate observa cum elementele *class* și *Student* au fost create cu numele lor explicite din Java, în timp ce clasei *Teacher* ia fost asignat în mod explicit numele de element *teacher*.

Explicarea adnotării @XmlValue

Pentru elementele XML care au atribute, nu au alte subelemente și conțin o primitivă drept valoare trebuie folosită adnotarea **@XmlValue**.

```
@XmlRootElement
class Bike{
    @XmlAttribute
    float wheelSize;
    @XmlAttribute
    float weight;
    @XmlValue
    String name;
}
```

Figura 7

Numai un singur **@XmlValue** poate să apară într-o clasa Java.

Un exemplu de folosire a acestei adnotări sunt în Figura 7 și Figura 8, unde avem elementul *bike* care conține atributele *wheelSize* și *weight*, iar în câmpul *name* este stocată o valoare string.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bike wheelSize="29.0" weight="11.0">Pegas Mare</bike>
```

Figura 8

Definirea colecțiilor de subelemente

Majoritatea fișierelor XML vor definii elemente XML care au mai multe subelemente de același tip (de exemplu : un element de tip *teacher* va avea mai multe subelemente de tip *student*).

JAXB permite o definire foarte ușoară a acestor colecții folosind obiecte Java de tip colecție.

În Figura 9 se poate observa cum elementul *teacher* are o colecție de subelemente de tipul *student*. Un exemplu de fișier XML care a fost generat folosind aceste clase poate fi văzut în Figura 10

```
@XmlRootElement
class Teacher{
    @XmlAttribute
    String name;
    @XmlElement
    ArrayList<Student> student;
}

@XmlRootElement
class Student{
    @XmlAttribute
    String name;
}
```

Figura 9

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<teacher name="Popescu Dan">
    <student name="Roscaneanu George"/>
    <student name="Ene Vlad"/>
    <student name="Dimoiu Cosmin"/>
</teacher>
```

Figura 10

Convertirea din Java în XML

```
private static void parseToFile(Object o,String filePath){
    File file = new File(filePath);
    try {
        JAXBContext jaxbContext = JAXBContext.newInstance(o.getClass());
        Marshaller jaxbMarshaller = jaxbContext.createMarshaller();

        jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,true);
        jaxbMarshaller.marshal(o,file);
    } catch (JAXBException e) {
        e.printStackTrace();
    }
}
```

Figura 11

Obiectul de tip **JAXBContext** este necesar folosirii JAXB-ului. Odată ce s-a inițializat o instanță de obiect care știe să folosească clasa care i-a fost dată ca parametru, vom putea genera obiecte de tip **Marshaller**.

Obiectele de tip **Marshaller** sunt folosite pentru a serializa obiecte în date cu formatul XML. Primul argument al metodei *marshal* este însuși obiectul pe care dorim să îl serializăm, iar al

```
Student st1 = new Student();
st1.name = "Roscaneanu George";
Student st2 = new Student();
st2.name = "Ene Vlad";
Student st3 = new Student();
st3.name = "Dimoiu Cosmin";
Teacher t = new Teacher();
t.name = "Popescu Dan";
t.student = new ArrayList<>();
t.student.add(st1);
t.student.add(st2);
t.student.add(st3);
parseToFile(t,"file.xml");
```

Figura 12

doilea argument poate fi un obiect de tipul **Node**, **File**, **Result**, **Writer**, **OutputStream**, **XMLEventWriter**, **ContentHandler** sau **XMLStreamWriter**.

În Figura 11 este prezentat cum se poate scrie un obiect Java(care a fost definit ca **@XmlRootElement**) într-un fișier.

Dacă proprietatea *JAXB_FORMATTED_OUTPUT* nu este setată cu valoarea *true*, atunci textul din fișier va fi scris tot pe o singură linie.

Dacă folosim clasele definite în Figura 9, pentru a obține un fișier care are același conținut cu ce este prezentat în Figura 10, se poate folosii codul definit în Figura 12.

Convertirea din XML în Java

```
private static <T> T parseFromFile(Class<T> cl, String filePath){
    T object = null;
    try {
        JAXBContext context = JAXBContext.newInstance(cl);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        object = (T) unmarshaller.unmarshal(new File(filePath));
    } catch (JAXBException e) {
        e.printStackTrace();
    }
    return object;
}
```

Figura 13

Odată ce s-a inițializat o instanță de obiect **JAXBContext** care știe să folosească clasa care i-a fost dată ca parametru, vom putea genera obiecte de tip **Unmarshaller** (prezentat în Figura 13).

Obiectele de tip Unmarshaller sunt folosite pentru a deserializa obiecte Java care au fost scrise în formatul XML. Singurul argument al metodei *unmarshal*(varianta care returnează un obiect de tipul *Object*) este un obiect de tipul **File**, **URL**, **Node**, **Reader**, **Source**, **InputStream**, **InputSource**, **XMLEventReader** sau **XMLStreamReader**. Obiectul care este returnat de către aceasta variantă de metodă trebuie să fie convertit de către programator la tipul de care are nevoie.

Sfaturi pentru optimizarea procesului de convertire

În Figura 11 și Figura 13 metodele definite de mine folosesc în mod ineficient obiectele **JAXBContext**, **Marshaller** și **Unmarshaller**. Sunt create și distruse de fiecare dată când se începe un proces de convertire. Aceste obiecte pot fi refolosite, dar am decis să încapsulez funcționalitatea lor în funcții pentru a arăta tot ce este necesar pentru a le folosi, și pentru a permite persoanelor care doresc să învețe să folosească JAXB să le fie mai ușor să le aplice în cod prima dată.

Obiectul **JAXBContext** trebuie creat doar odată pentru fiecare clasă la se dorește să se facă conversie (este recomandat ca această instanță să fie stocată undeva cu acces global). De-asemenea acest obiect este thread-safe.

Obiectele **Marshaller** și **Unmarshaller** nu necesită mult timp ca să fie construite și nu sunt thread-safe. Recomandabil este să fie creată câte o astfel de instanță pentru fiecare fir de execuție pentru a nu avea nevoie de obiecte de sincronizare (de exemplu mutex, sau block-uri synchronized).

Bibliografie

1. https://docs.oracle.com/javase/tutorial/jaxb/
2. https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxb/index.html