Софийски университет „Св. Кл. Охридски”

Факултет по математика и информатика

Бакалавърска програма
„Софтуерно инженерство”

Предмет: XML технологии за семантичен Уеб

Зимен семестър, 2020/2021 год.

Тема №34: „XML to/from relational DB applications“

Курсов проект

Автор:

Борислав Карапанов, фак. номер 62280

януари, 2021

София

Съдържание

1 Въведение

Настоящият документ представлява документация на конзолно приложение на Java, което има следната функционалност:

- Извличане на данни от релационна база данни за продукти и търговци в XML документи с подходящо DTD.
- Разбор на XML документи, съдържащи данни за продукти и търговци, валидирайки ги с техните DTD-та и създаване на съответните таблици, съдържащи съответните данни, използвайки SQL.

Базата данни, която е използвана за реализация на приложението е **H2 Embedded Database**. Документация и упътвания за употреба могат да бъдат намерени на официалния сайт на H2 → https://www.h2database.com/html/tutorial.html.

Версията на Java е **Java 14**, а разбора на XML документите става със **Streaming API for XML (StAX)**.

В секция 2 от този документ е представен моделът на данните, с който ще работи приложението., както и начинът на употреба. Секция 3 съдържа дизайна на приложението. Секция 4 описва тестовете, които са приложени, за да се осигури необходимата надеждност. Секция 5 е заключителна.

Github repository на проекта може да бъде намерено на https://github.com/Borislav-K/XML-Project.

2 Анализ на решението

2.1 Работен процес

Приложението е стандартно конзолно приложение с prompt. Потребителят въвежда команда и тя се изпълнява. Входния интерфейс към конзолното приложение се състои от 3 команди:

Quit → Терминиране на програмата

convert –tosql <input_file> → Прочита се съдържанието на **<input_file>**. Валидира се, че това е добре структуриран и валиден XML файл, като в противен случай се връща подходяща грешка. След това, данните се прочитат и се вкарват в базата данни.

convert –toxml <output_file> → Прочитат се всички редове от таблиците в базата данни и се вписват в **<output_file>**. В резултатния файл се съдържа и вградено DTD, което отговаря на релационния модел на данни (виж 2.2.2).

Базата от данни е embedded и се намира в папката ${папка_на_проекта}/database.

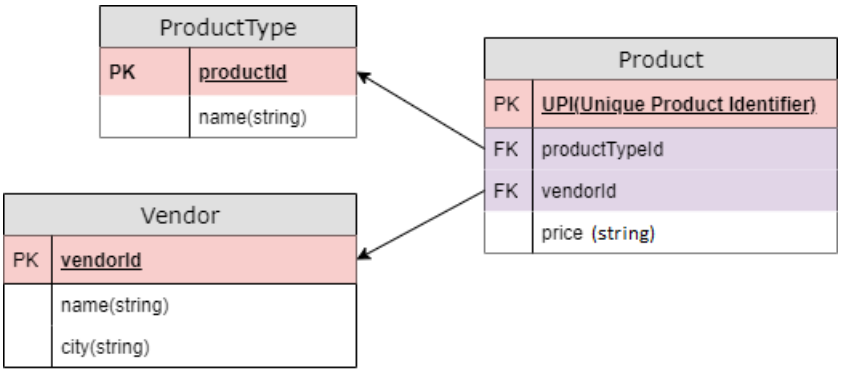
Ето и примерна употреба на приложението:

```
Enter command: s
Unknown command
Enter command: convert --toxml test1.xml
Conversion successful. The content of the database can be found at test1.xml
Enter command: convert --tosql resources/badly_structured.xml
The XML file is badly structured: Unexpected close tag </market>; expected </vendor>.
 at [row,col {unknown-source}]: [4,8]

Enter command: convert --tosql resources/invalid.xml
The XML file is not valid: Attribute 'productTypeId': Empty ID value
 at [row,col {unknown-source}]: [19,5]

Enter command: convert --tosql resources/example_shop.xml
Conversion successful. The XML file's content was added into the database

Enter command: convert --toxml test2.xml
Conversion successful. The content of the database can be found at test2.xml
Enter command:
```

2.2 Структура на съдържанието

2.2.1 Релационна схема на данните

Данните, които ще се съдържат в релационната база данни ще се съдържат в 3 таблици, които са показани в следната релационна схема:



ProductType е образец за продукт(ябълка, единица облекло и т.н.). Всеки тип на продукт има идентификатор **productId**, което е ключът на таблицата.

Vendor е конкретен търговец, който продава продукти. Всеки vendor има идентификатор **vendorId,** което е ключът на таблицата.

Product е конкретна инстанция на продукт, продаван от даден търговец. Всеки продукт може да бъде продаван от всеки търговец, на различна цена и всеки търговец може да продава 0..* продукти. Всяка инстанция на продукт има собствен идентификатор – **UPI** , за което можем да

мислим като за баркод. В таблицата Product имаме и връзки към 2 foreign ключа, а именно идентификаторите на типа на продукта и на конкретния търговец, който продава тази инстанция на продукта. Полето **price** не случайно е string, тъй като DTD така или иначе не поддържа типово ограничение за числени променливи.

И трите таблици се намират в базата от данни **market**. Всички посочени полета са задължителни.

2.2.2 DTD на XML съдържанието

Ето и как изглежда DTD-то, с което ще се валидират готовите XML документи, както и тези, които тепърва предстои да бъдат обработени и мигрирани към релационната база:

```
<?xml version="1.0" encoding="UTF-8" ?>
        <!ELEMENT market (productType*,vendor*,products?)>
        <!ELEMENT productType (name)>
        <!ELEMENT vendor (name,city)>
        <!ELEMENT products (product*)>
        <!ELEMENT product (price)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT city (#PCDATA)>
        <!ELEMENT price (#PCDATA)>
        <!ATTLIST productType productTypeId ID #REQUIRED>
        <!ATTLIST vendor vendorId ID #REQUIRED>
        <!ATTLIST product
                upi ID #REQUIRED
                productTypeId IDREF #REQUIRED
                vendorId IDREF #REQUIRED>
```

Както виждаме, XML документът ще се състои от коренов елемент **market**, който може да съдържа произволен брой образци на продукти и търговци и 0 или 1 елемента **products**. **Products** елементът служи като контейнер на всички инстанции на продукт и може да съдържа произволен брой **product** елементи. Всички идентификатори са атрибути и са задължителни. Foreign ключовете от релационната схема съответстват на **IDREF** атрибутите на елемента **product**. Данните, които не са идентификатори, са под-елементи на съответните тагове.

2.3 Тип и представяне на съдържанието

Съдържанието, което се обработва от приложението представлява множество XML файлове и редове в релационна база данни. Готовите XML файлове в github repository-то са авторски. Не се използва съдържание от трети лица.

3 Дизайн

Основните XML технологии, които се използват са DTD и StAX. Когато се подава XML документ за вход, той се валидира спрямо неговото DTD. Когато се създава XML документ от данните в базата, в него се вгражда следното DTD:

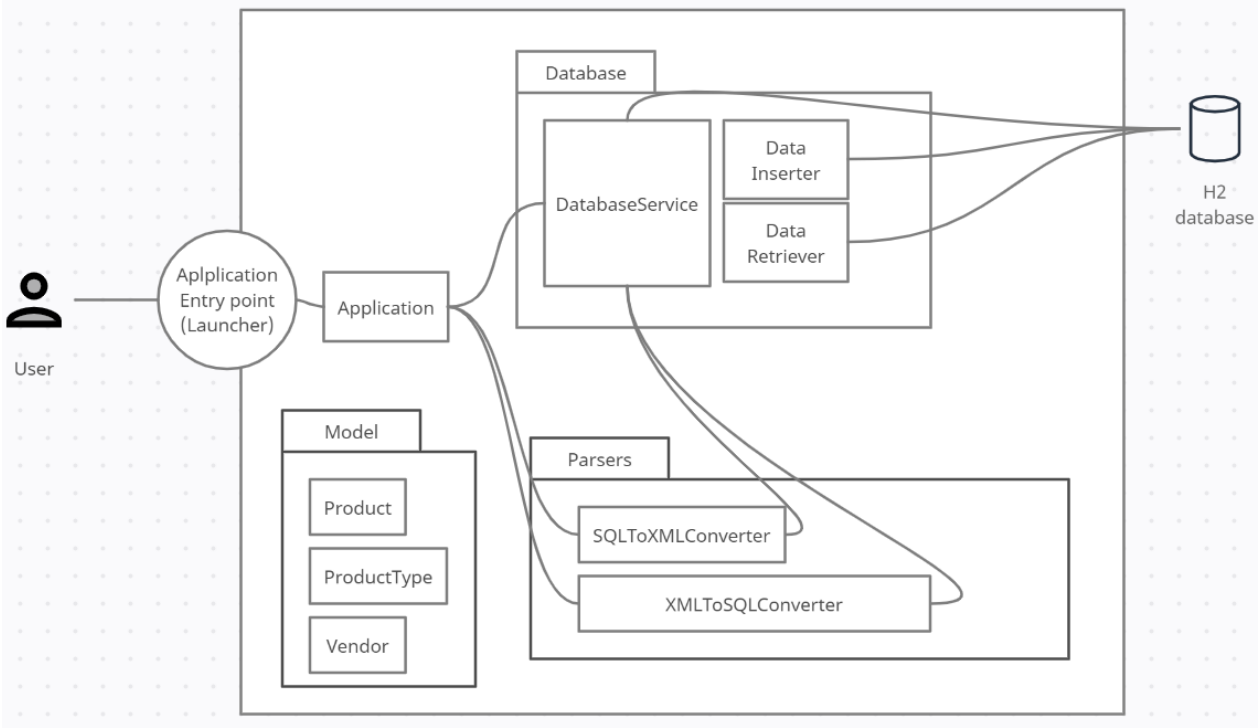
```
<!DOCTYPE market [
        <!ELEMENT market (productType*,vendor*,products?)>
        <!ELEMENT productType (name)>
        <!ELEMENT vendor (name,city)>
        <!ELEMENT products (product*)>
        <!ELEMENT product (price)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT city (#PCDATA)>
        <!ELEMENT price (#PCDATA)>
        <!ATTLIST productType productTypeId ID #REQUIRED>
        <!ATTLIST vendor vendorId ID #REQUIRED>
        <!ATTLIST product
                upi ID #REQUIRED
                productTypeId IDREF #REQUIRED
                vendorId IDREF #REQUIRED>
        ]>
```

То е вграденият еквивалент на DTD от 2.2.2.

Валидацията при входен XML файл се поддържа автоматично от StAX. Тук може би е подходящият момент да се спомене, че стандартната имплементация на Streaming API-то не поддържа валидация(не може да се зададе свойство XMLInputFactory.IS_VALIDATING на true), поради което се използва open source библиотеката WoodStox -> https://github.com/FasterXML/woodstox.

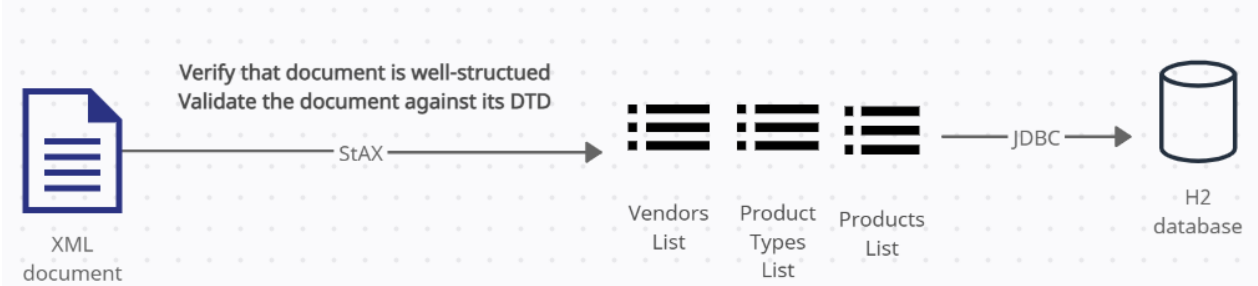
Няма съществени разлики между използването на стандартната имплементация на StAX и тази на woodstox – единствената, която касае този проект е, че имплементацията на WoodStox поддържа валидация на документи с DTD, което опрости решението. В противен случай, би трябвало да се използва DOM или SAX API-то, за да се валидира документа, което би усложнило имплементацията.

3.1. Архитектура на приложението



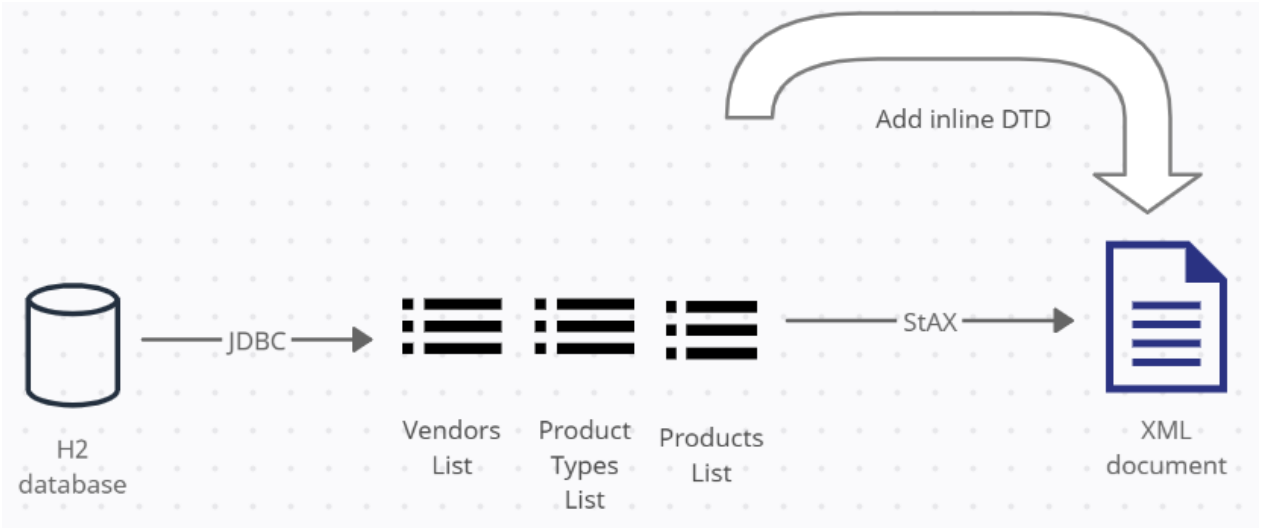
Горната диаграма представя на високо ниво архитектурата на приложението. Потребителят стартира приложението чрез main метода на Launcher класа. Той създава Application клас, който съдържа референции към всичкчи необходими компоненти за изпълнението на командите на потребителя. Model пакетът съдържа Plain Old Java Objects, които имат методи за записване в XML файл и за конвертиране в кортеж, готов за използване в INSERT заявка към базата данни. Database пакетът съдържа класове за установяване на връзка към базата и за двупосочна комуникация с нея. Интефейсът между database layer-а и останалата част на приложението е класът DatabaseService, който съдържа референции към Data Inserter и Data Retriever и делегира някои функционалности към тях. Parsers пакетът съдържа 2 класа, които използват StAX за конвертиране XML<-> SQL. Техните вход и изход за XML файл, и списък от обекти от тип model, които се взимат/вкарват в базата, в зависимост от data flow-а.

3.2. XML -> SQL data flow



Самият разбор на XML документа е много прост. Използва се iterator API-то на StAX. Всеки път, когато получим отварящ таг, който ни интересува, прочитаме цялото му съдържание и го записваме като Java обект в списък. Когато целият XML файл е прочетен, превръщаме списъка в SQL заявка(чрез обработка на string-ове, не се използва ORM). След това с една заявка към базата вкарваме съдържанието.

3.3. SQL -> XML data flow



Обратният процес е аналогичен. Първо прочитаме данните от базата данни и ги съхраняваме в списъци от обекти. След това отново използвайки StAX, записваме резултатите в XML документ. Тъй като на таблиците в базата сме наложили ограничения, които са по-стриктни от тези в DTD-то, новият XML документ е винаги валиден. Въпреки това, в него се добавя и вградено DTD, в случай, че биха се променяли данните в него от други приложения или хора.

4 Тестване

За имплементацията са реализирани unit тестове с Junit и Mockito за 2-та класа, които извършват конвертирането XML<->SQL. В тези тестове се mock-ва базата от данни.

За класа, който конвертира XML->SQL се подават низове, които репрезентират XML файлове и се валидира, че се извиква класът, който представлява интерфейса към базата от данни с подходящите списъци от данни.

За класа, който конвертира SQL->XML се подават списъци от данни, които биха дошли от интерфейса към базата от данни, и се проверява съдържанието на готовия XML файл.

Тестват се и случаите, в които входните файлове са невалидни или не са добре структурирани XML файлове.

5 Заключение и възможно бъдещо развитие

Освен StAX, друг начин за имплементация на този проект би била с DOM. Но, DOM има голям недостатък в този случай, защото разборът на XML файла може да стане последователно(без връщане назад) и поради това няма причина да се зарежда целият документ в паметта. При работа с големите документи и ограничена памет, DOM би се провалил като API за имплементация. Също така, производителността на StAX е по-добра от тази на DOM. SAX също би могло да се ползва, но то е API само за четене, т.е. за частта SQL->XML отново би трябвало да се използва друго API, което би било лоша практика, тъй като ще се усложни имплементацията – използвайки само StAX, имплементациите на двата потока от данни са сходни и консистентни една с друга.

За имплементацията на database layer-а от приложението взех решение да не използвам Object-Relational Mapping, тъй като имаме само 3 класа с по 2-3 свойства и използването на Hibernate би донесло твърде много boilerplate код и конфигурации, за да си заслужава. Вместо това съставям заявките към базата чрез parse-ване на низове, което не е никак сложно.

6 Разпределение на работата

N/A → Проектът е разработен от един студент, самостоятелно.

7 Използвани литературни източници и Уеб сайтове

1. WoodStox docs →https://github.com/FasterXML/woodstox
2. H2 Database tutorials → https://www.h2database.com/html/tutorial.html
3. StAX docs → https://docs.oracle.com/cd/E24329_01/web.1211/e24993/stax.htm#XMLPG177
4. JDBC docs → https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html