2017

Dynamic Database Schemas and Multi-Paradigm Persistence Transformations

Ryan D. Norton
Grand Valley State University

Dynamic Database Schemas and Multi-Paradigm Persistence Transformations

By
Ryan D. Norton
Dec, 2017

Dynamic Database Schemas and Multi-Paradigm Persistence Transformations

By
Ryan D. Norton

A project submitted in partial fulfillment of the requirements for the degree of

Master of Science in

Computer Information Systems

at

Grand Valley State University

December, 2017

_____

Dr. Jonathan Leidig **Date**

Table of Contents

Abstract

Today, countless businesses use relational databases to store essential information. That data, however, doesn't always come in the same structure. XML files, for example, may have various schemas for a document type, outlined by numerous XSD files. It may not always make sense to use a traditional relational database for this storage, as NoSQL solutions offer flexibility, speed, and powerful visualization tools. Often enough these documents and schemas are known and used by numerous branches or offices in a company, but need to be stored in a centrally located database. The goal of this work is to solve the problem of saving XML files of various schema types in the same database, by dynamically altering the schema of the database to accommodate the new file structures. In addition to relational database storage, the XML files are also mapped to a graph database to accommodate additional business needs such as visualizing relationships among the data using more powerful methods than traditional data stores. This project also aims to minimize the effort spent by a software developer persisting data with different schema types as well as time allocated to creating methods for storing newly added schemas to the data persistence workflow. It achieves this by automating the process, using several existing persistence frameworks such as Java Architecture for XML Binding (JAXB), Hibernate Object-Relational Mapping (ORM), and the Neo4J Object Graph Mapping Library (OGM). This work intends to integrate these technologies into a cohesive, easily configurable, highly extensible framework that provides a largely automated solution to dynamically mapping evolving data structures to multiple data persistence paradigms.

Introduction

My employer is an insurance company with the need to store policy books (books of many customer insurance policies) from a wide variety of independent insurance agents. Each agent stores his or her book of policies in a slightly different format (ex. some policies may contain discounts, while others may not). These books are often stored in Excel as CSV or XLSX files, which then get converted to XML (Extensible Markup Language) with an accompanying XSD (XML Schema Definition) file. XSD is a World Wide Web Consortium (W3C) recommendation that specifies how to formally describe the elements in an XML document. These policy books (that may or may not have a given data field) then need to be stored in a central database for later updates and retrievals. The schema of the database needs to be flexible and dynamically change to accommodate the persistence of new file schemas it may not currently have a column or table for. It also needs to do this without extraneous amounts of time devoted to developer configuration in order to maintain a quick turn-around time between policy books from different agents or States. In addition, the framework used to persist this data needed to be flexible enough to transform the data from relational to NoSQL persistence paradigms. This flexibility accommodates a need for analyzing and visualizing relationships between the data in a graph database (Neo4J) where relationships between nodes represent the data in a structure more consistent with the real world (Figure 1). This allows for new correlations to be revealed that were before not possible using traditional relational database management systems.

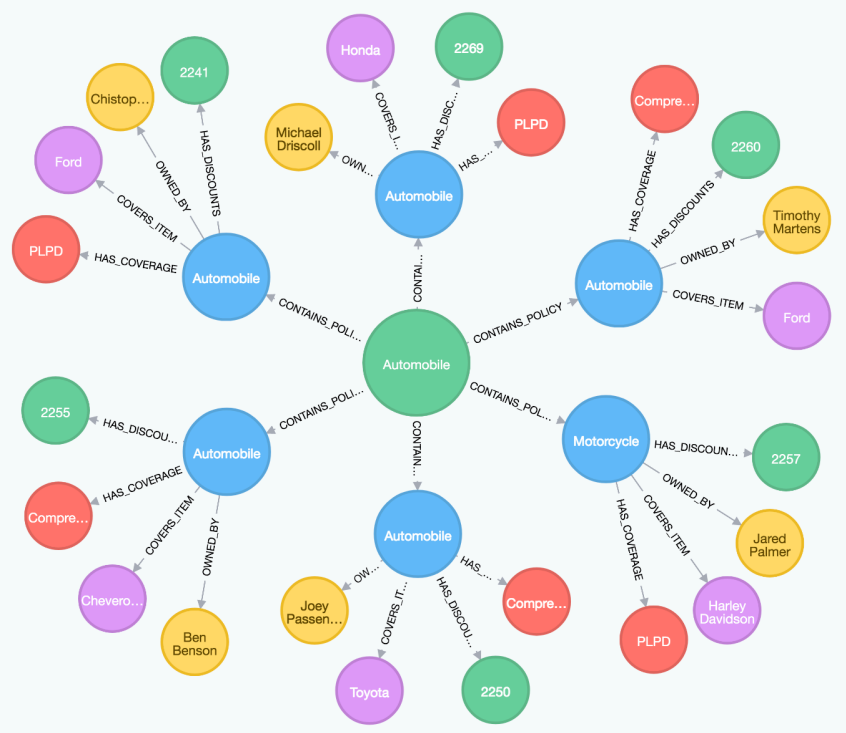

Figure 1: Neo4J Visualization

Background and Related Work

Many tools have been developed over the years to solve a portion of the problem of transforming data from an XML format and persisting it in a relational database, to a graph database and then back to XML again, but no single tool exists that will perform all of these tasks. Many tools use an XSD file to define what values are valid for an object, as well as the variable data types. FastXML's Jackson and Oracle's Java Architecture for XML Binding (JAXB) are popular solutions that can take XML and XSD files and unmarshall the data into Java classes and marshall them to to XML. JAXB was chosen because it is the most ubiquitous solution, is often incorporated into many Integrated Development Environments (IDEs), and has a large support community. JAXB quickly and efficiently serves its intended purpose, however, it does not persist data at all. Other tools must be relied upon to persist the generated Java objects such as Hibernate ORM (Object-Relational Mapping). The Hibernate framework maps annotated Java objects to a relational database via JDBC (Java Database Connectivity). While there are far fewer tools to transform XML documents to graph database paradigms, Neo4J (currently the most prevalent graph database platform) does offer a library for mapping Java objects to their NoSQL persistence platform. Neo4J OGM (Object-Graph Mapping) is a pure java library that can persist annotated domain objects to the Neo4J graph database. These tools work well to accomplish a specific duty, but in order to address the task as hand, a platform would need to be developed to integrate these tools to operate in unison.

Implementation

During the initial architecture design of the project I determined that transforming the XML data into Java objects would be the most flexible solution for persistence and transformation. The architecture went through several iterations of design before the final design was created (Figure 2). The application functionality was broken down into two segments: the Ingest/persist stage and the evaluate validity/rollback stage. These segments were broken down further into modules to maximize cohesion and reduce coupling. Modules in the architecture diagram are made up of either a single method, or multiple methods contained within a class. The application first receives the input XML and XSD files and then uses the JAXB framework to ingest the data. Next, it validates the XML content against the XSD to check for integrity violations. The program then unmarshalls the data into Java objects before determining what schema to dynamically configure for the data persistence. Once the schema is successfully configured, the application persists the data in the relational and NoSQL databases using the Hibernate and Neo4J OGM frameworks respectively. Next, the evaluation stage of the application retrieves the newly persisted objects and marshalls them back into XML to validate against the original XML and XSD files for accuracy as well as violations. The application either commits the data or rolls-back the transaction depending on the outcome of the evaluation process.

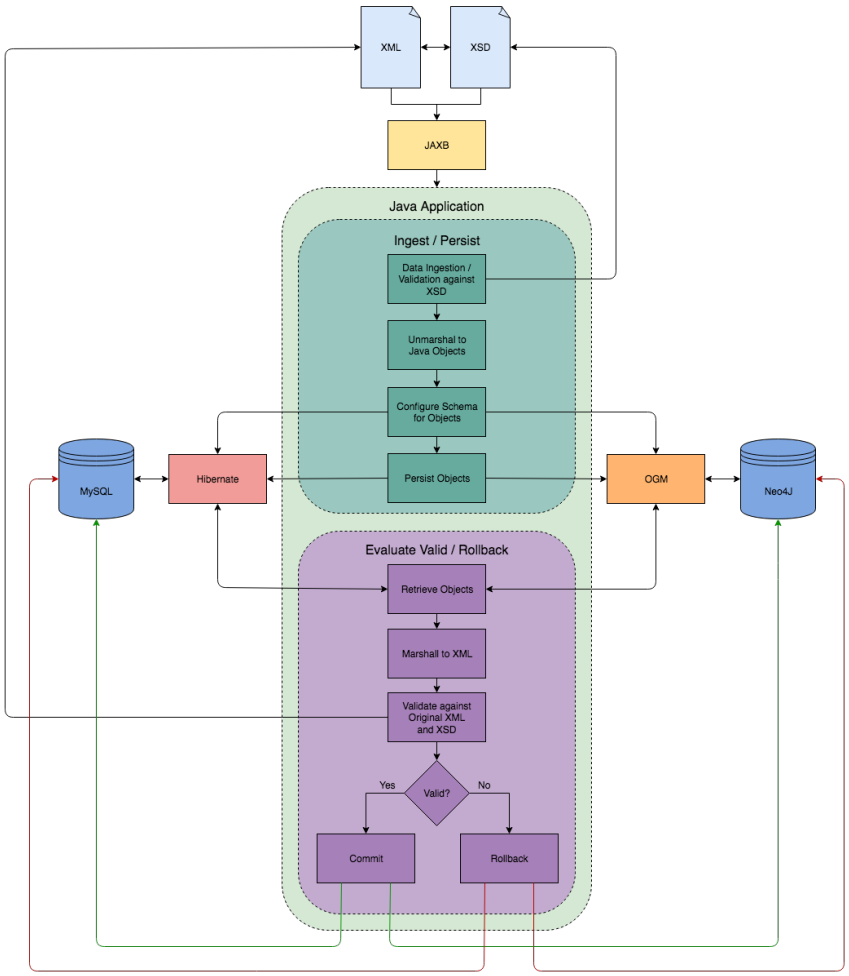

Figure 2: Project Architecture Diagram

The project was written entirely in Java as all of the tools needed for the project either natively supported Java or were written in Java themselves. I also have the most familiarity with the language over any other Object-Oriented language in my repertoire. The Java objects created, needed to accurately represent the original data and therefore required validation for consistency as well as integrity violations throughout the application's runtime process in order for the platform to have substantial value. JAXB was used to transform these XML/XSD files into plain old Java objects (POJOS) due to its aforementioned flexibility, support, and easy incorporation into my Integrated Development Environment (Eclipse Neon 3 Release (4.6.3). Next, Hibernate ORM was chosen for the relational mapping due to its vast library of methods as well as its excellent session and transaction management methods used to persist POJOS. The Database Management System (DBMS) chosen for the relational database was MySQL due to its ease of use and my familiarity with the system and its connection drivers. Neo4J was chosen as the graph DB because of its relatively simple, yet powerful persistence layer and visualization tools. Lastly, the Neo4J OGM library seemed to be an obvious choice since it was developed by Neo4J and offered a vast range of tools, including excellent session and transaction management as well as direct access to the proprietary Neo4J Cypher languages for both embedded and remote Neo4J databases. All dependencies were brought into the project using the Maven dependency manager. I had no prior experience with any of the tools I attempted to integrate into the project with the exception of those previously mentioned. In fact, it was also my first time working with XML and XSD files. Since the XML policy books contain customer proprietary information, a sample dataset of XML and XSD files was created for use in this project (Figure 2).

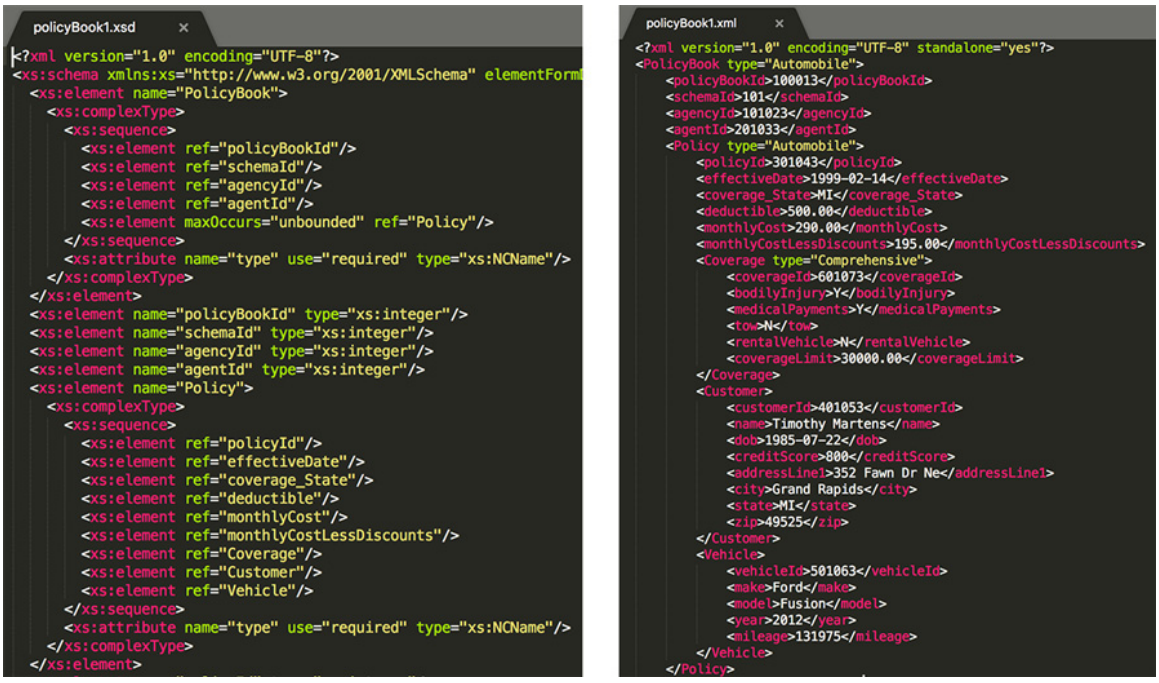

Figure 2: Sample XSD and XML files

The first operation the program performs is looping through each XML file and validating it against its corresponding XSD file to ensure that it does not violate any of the schema integrity constraints. It does this by

importing Java's XML Validation API and using schema validator methods to ensure that the XSD constraints have not been violated. Next it unmarshalls the policy book contained in the XML file into Java objects. These Java objects are then persisted in MySQL using Hibernate. Hibernate retrieves a session from its session factory and opens it in order to begin a transaction to persist the data. Based on the agency ID found in the XML file, a schema is selected to persist the objects and they are written to the databases. At this point the policy book that was just stored in the database is retrieved and marshalled into an XML file to be then validated against the original. The validation process utilizes a robust library called XMLUnit to ensure that the contents are identical. XMLUnit was configured to normalize the XML file, expand entity references, ignore white space, attribute order and comments in order to accurately compare the contents of the file rather than the formatting. If the retrieved policy book could not be validated successfully, the transaction was rolled back and a message was displayed to the user. If the validation was successful the transaction was then committed to the database. The same workflow was applied to the storage and validataion of the XML files to the Neo4J database. All XML files at this point were validated against the original XSD file one last time to ensure that no data has violated any of the integrity constraints and that no anomalies have been introduced to the data. A message then gets generated regarding the successful persistence of data or failure thereof. I decided to create a very simple program GUI using the Java SwingBuilder Library in order for the user to easily upload an entire directory of policy books at once as well as receive the outgoing application messages (Figure 3).

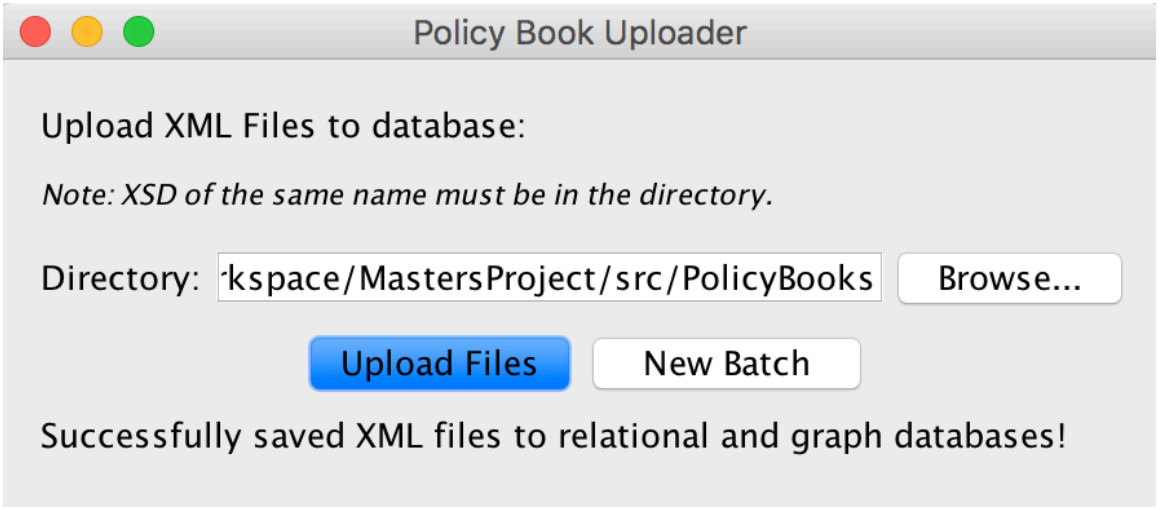

Figure 3: The Graphic User Interface

Conclusions and Future Work

The Java program created was capable of successfully transforming a structured document file (XML) to a relational persistence paradigm using MySQL, a NoSQL persistence platform using Neo4J, and back to an XML file without introducing any foreign anomalies to the data. The data was also validated against the constraints of the XSD file throughout the process to ensure that the data stayed consistent. When a new file needed to be stored in the system, the developer was simply required to annotate any new data fields to properly map the data to the persistence layers and add them to the configuration file. The configuration file

helps tell MySQL and Neo4J how to save the data when opening a session and beginning any transactions needed for persistence. The program was extremely successful at accomplishing its goal of using a batch process to dynamically persist and transform a large volume of XML files (containing various document structures and rules) between multiple persistence paradigms. It was able to transform the XML data into a graph database for deep analysis and visualization. Finally, achieved its goal of being flexible enough to handle new file schemas with minimal updates made by the developer creating minimal downtime.

Future work for the project will include the option for a user to upload individual XML files as well as batch operations for more flexibility. It will also retrieve the XSD file using an HTTP REST call to retrieve it from the webserver rather than forcing the user to provide it with each upload. While the application does provide significant logging on the backend to inform developers of any problems that have occurred, the user is only notified of minimal information. Messages to the user will be made more descriptive regarding the exact cause and error message for simpler troubleshooting. Finally, the application currently only rolls back a single policy book when it cannot be validated; the future implementation will roll back all policy books that were attempted to be uploaded in a batch until the user can resolve the specific issue. This will simplify the need to determine which books were properly saved and which books need to be uploaded again.

Bibliography

- FastXML, "Jackson Project Home @github", 2017, https://github.com/FasterXML/jackson.
- Ed Ort and Bhakti Mehta, "Java Architecture for XML Binding (JAXB)", March 2003, http://www.oracle.com/technetwork/articles/javase/index-140168.html.
- NumberFormat, "Using JAXB to Convert Between XML and POJO's", February 1, 2009, https://numberformat.wordpress.com/2009/11/01/using-jaxb-to-convert-between-xml-and-pojos/.
- Hibernate, "Hibernate ORM, What is Object/Relational Mapping?", 2017, http://hibernate.org/orm/what-is-an-orm.
- Neo4J, "The Neo4j Developer Manual v3.3", 2017, http://neo4j.com/docs/developer-manual/current.
- Neo4J, "OGM User Manual", 2017, https://neo4j.com/docs/ogm-manual/2.1/introduction.
- Elliotte Harold, "The Java XML Validation API, Check your documents for conformance to schemas", February 10th, 2010, https://www.ibm.com/developerworks/library/x-javaxmlvalidapi/index.html.
- Oracle, "Package javax.xml.validation JavaDoc", https://docs.oracle.com/javase/7/docs/api/javax/xml/validation/package-summary.html.
- Tim Bacon and Stefan Bodewig, "XMLUnit Java User's Guide", 2014, http://xmlunit.sourceforge.net/userguide/html/index.html.

Appendices

Links to tools and software used:

Java: http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

Eclipse: http://www.eclipse.org/downloads/eclipse-packages/

JAXB: https://github.com/javaee/jaxb-v2

MySQL Server/Workbench: https://dev.mysql.com/downloads/

Hibernate ORM: https://github.com/hibernate/hibernate-orm

Neo4J: https://neo4j.com/download/

Neo4J OGM: https://github.com/neo4j/neo4j-ogm

XMLUnit: www.xmlunit.org

Maven Repository paths: https://mvnrepository.com/