Carleton University
School of Computer Science
COMP 4905
Honours Project
Fall 2016

MusicXML Parser

Author: Joel Sprinkle

Student ID: 100960023

Supervisor: Louis D. Nel

Abstract

MusicXML Parser

Author: Joel Sprinkle

Musicians use sheet music to represent musical compositions, known as scores, in a textual form. Traditionally, paper has been the medium for creating and sharing sheet music; however, this is changing. For over 10 years now developers have been working on and improving a product that stores sheet music digitally in plain text so that it can be easily shared and manipulated by musicians. This product is called MusicXML, it uses XML to store musical notation digitally, and it has become the standard open format for representing sheet music. The purpose of this project is to create a desktop application built in Java that can keep track of a library of MusicXML files in a database and parse these files on the fly. Covered in this report is the reasoning for creating the application, what tools were used, and how these tools were used to implement the project. As well, the design of the project, issues encountered when creating the application, and what was accomplished is detailed.

Supervisor: Louis D. Nel

Acknowledgements

I would like to thank everyone who helped me with this project, especially my supervising Professor Louis D. Nel who originally came up with the idea.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Sheet music is one way of representing music; it represents a score, a musical composition in written form. This written form covers all aspects of the musical score from the timing, to the melody, to the chords, and that is just scratching the surface. Sheet music is a way for composers to store their work so that it can be shared and played by others in the future. A lot of musical scores are not overly complicated and are fairly straightforward to read, however some scores can be very technical and complex. With the onset of the digital age, the question was asked: how can musical scores be stored, displayed, and possibly edited all digitally? Especially when taking into consideration how complex some scores can become, such as scores that have multiple parts or possibly changing tempos partway through. This question has been answered by the MusicXML format.

The MusicXML format stores musical scores using the XML (EXtensible Markup Language) format. Storing musical scores in the XML format allows the data, the musical notation and all aspects that describe the musical notation, to be stored in a very verbose manner. This is good because it allows the music to be described in a very detailed way. The basic aim of this project is to build a GUI (Graphic User Interface) application to parse and store MusicXML files into Java objects.

1.2 Motivation

The goal of this project is to create a desktop application where users can keep track of MusicXML files in a database. Users should be able to add entries, search the database, edit entries, and remove entries from the database. The application should also parse a selected MusicXML file, as well as let users export parsed MusicXML data back into an XML file.

In more detail, the project will be built in Java, using IntelliJ IDEA for the IDE and JavaFX to create the GUI. An embedded SQLite database will keep track of the MusicXML files, and GUI controls will allow for the addition, filtering, editing, and removal of entries in the database. The application will parse MusicXML data into a set of Java objects that can be accessed, for use with future projects, and so that the data can be exported back into a MusicXML file.

The two main reasons behind the goals of this application are, first, so that Louis can potentially use the project for his work with the music department. Second, Louis can also use this project as a teaching tool to introduce students to working with an embedded SQLite database in Java, the JavaFX library, and working with XML.

1.3 Outline

The rest of this report is organized into several sections. Chapter 2 covers background information on the project and the tools used to create the project. Chapter 3 covers why the selected tools were used, as well as the design of the project and any issues that were encountered. Finally, Chapter 4 covers what was accomplished, and what future work can be done to advance the development of the project.

Chapter 2

Background

Several APIs and pieces of software were used to create this project. Covered below is background information about the key systems that were used in the development of this project.

2.1 IntelliJ IDEA

IntelliJ IDEA is an Integrated Development Environment (IDE) that allows developers to create computer software using Java, a computer programming language. IntelliJ is mainly composed of a source code editor, a compiler, and debugging tools. IntelliJ also has several other parts that let developers use version control systems (VCSs), as well, there are analytical tools for inspecting and cleaning up code [Jet16].

2.2 JavaFX

JavaFX is a native Java library that allows for the creation of GUIs. JavaFX not only allows for the production of desktop applications on all modern operating systems, but Rich Internet Applications (RIAs) as well. The JavaFX library can be used on its own, where a developer creates the GUI completely programmatically, or it can be used in conjunction with JavaFX Scene Builder. Scene Builder is an application that helps developers create GUIs by easily allowing them to drag and drop controls onto a frame. The code for the frame is then generated automatically and stored in an *.FXML file. FXML is a specialized version of XML that defines a JavaFX applications GUI [Ora14a].

2.3 SQLite

SQLite is a powerful self-contained relational database. Using xerial's sqlite-jdbc driver, SQLite databases can be embedded into Java projects. One of the best features of SQLite databases is that they consist of a single file, which means that they are highly portable [Con16].

2.4 XML

XML is a human and machine-readable language, it was designed to hold data, where the tags enclosing the data describe the data itself, see figure 2.1. As well, XML is a general-purpose language, meaning that developers define the tags that encapsulate the data that is to be stored [Qui16, Dat16].

```
<note default-x="143">
  <pitch>
    <step>F</step>
    <octave>4</octave>
  </pitch>
  <duration>8</duration>
  <voice>1</voice>
  <type>quarter</type>
  <stem>none</stem>
  <notations>
    <slur number="1" type="stop"/>
  </notations>
</note>
```

Figure 2.1: A single musical note described in XML formatting [Mak16].

The XML format allows for the creation of Document Type Definitions (DTDs), which allow developers to define what elements are allowed in an XML file, as well as the structure of the XML file. XML Schema Definitions (XSDs) are similar to DTDs, except that XSDs provide a tighter restriction on what the structure of an XML file can look like. Both DTDs and XSDs can be used to validate XML files, to ensure that an XML file complies with a set standard [Qui16, Dat16].

2.5 WoodStox

Woodstox is a Java XML processor, which uses the StAX (Streaming API for XML) API. It easily lets developers not only parse but also write XML. Woodstox also contains configuration features, such as using less memory, or increased parsing speed. Woodstox is an open source project that is still being actively maintained and developed (at the time of writing this the latest release, version 5.0.3, was release August 23, 2016). However, I would like to note that Woodstox does not seem to be very well documented, this may be due to `http://woodstox.codehaus.org/` no longer being active [Fas16, Fas09].

2.6 MusicXML

MusicXML provides a way of digitally storing musical scores, in XML format. MusicXML is an open standard so that any application can read it, display it, edit it, or write it. This open standard lets MusicXML files be written in one application and then read by another. The MusicXML format is defined by both DTDs and XSDs so that developers can ensure correctness when reading or writing MusicXML files [Mak16]. Storing musical scores in the MusicXML format is advantages because it allows for a score to be edited more easily, and then generated on the fly as needed, see figure 2.2 [Mak16]. MusicXML contains two types of scores, partwise scores and timewise scores. In partwise scores, each part contains all of it measures in one location. Where as in timewise scores each measure contains each parts section for that measure [Mak16].

2.7 Other

2.7.1 version Control

GitHub is a repository based VCS that allows developers to manage source code, use the repository as a backup, and not only let developers share projects but lets teams work on the same project at the same time [Git16].

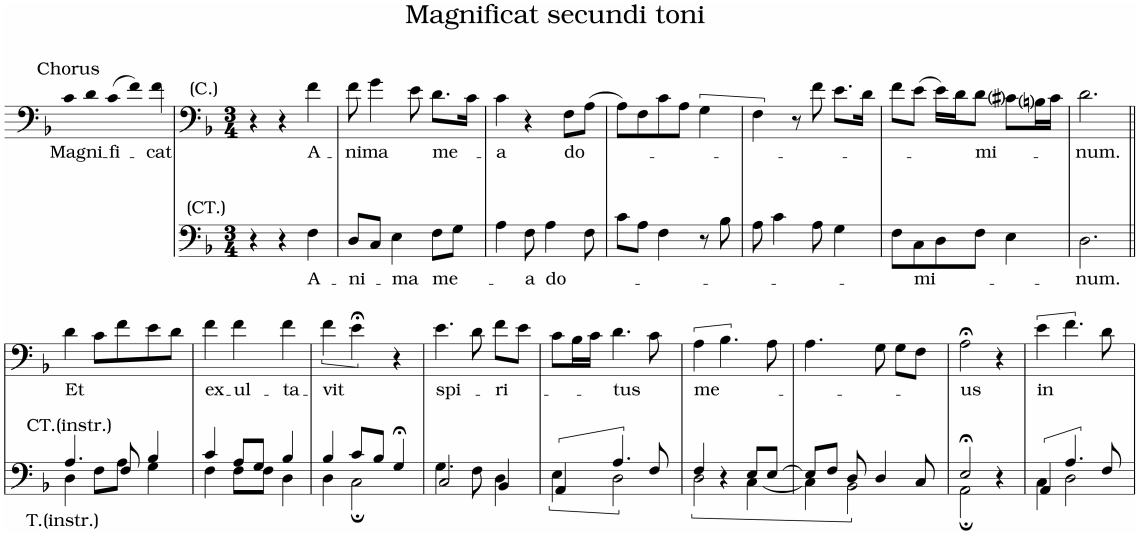

Figure 2.2: A MusicXML file, converted into readable sheet music [Mak16].

2.7.2 Overleaf

This report was created using Overleaf, an online LaTeX tool that allows users to create documents in a web-app using LaTeX with a live preview of the document being generated. Because a live preview of the document is continually being generated, any errors in the document are found right away. Overleaf also allows for real-time collaboration on documents, similar to Google Docs but with LaTeX [Lim16].

Chapter 3

Methodology

Here the approaches that I used to create the project will be looked at. This includes, the reasoning behind tool selection, the methods used to implement the project, issues that were encountered, methods that were discarded, and how the project was tested.

3.1 Reasoning

3.1.1 JavaFX & IntelliJ IDEA

The main reason IntelliJ IDEA and JavaFX were chosen for this project is because Louis wanted to use them. That being said, both items were the most advantages products to use.

One of the main competitors of IntelliJ is Eclipse, both IDEs are robust, fairly straightforward to use, and have all the features one would expect from a major IDE. Eclipse was considered for use, as many Carleton students and staff at the university are familiar with the IDE. However it does not offer out of the box support for JavaFX like IntelliJ does. Eclipse requires additional software to be installed (e(fx)clipse) for JavaFX projects to be created [Bes16]. Because of IntelliJs native support for JavaFX, creating a greater ease of use over Eclipse, it was chosen as the IDE to use for the project.

When it comes to creating GUI applications in Java, Java Swing was the standard for a long time. However, Oracle has announced that while there will be continued support for Swing for some time, JavaFX is replacing Swing as the standard user interface library for Java [Ora16]. JavaFX also has some advantages over Swing, such as offering native actions for touch screens

interfaces, i.e. ”On Touch” and ”On Swipe.” As well, JavaFXs implementation of FXML files not only keeps the GUI design separate from the application logic, but is easy to read and maintain as well. Because of these advantages and Oracles dedication to JavaFX over Swing, JavaFX was to chosen as UI library for this project.

3.1.2 SQLite

SQLite was chosen over other possible databases for the project because it is open, easy to use, and it is used in the databases class at Carleton. Which means that Louis can use this project as a teaching tool. Other features of SQLite that make it a good choice are, no configuration is needed, a database is composed of a single cross-platform file on disk which makes its portable, and it is self contained [Con16].

3.1.3 Parsing

When it comes to parsing XML in Java, the Java API for XML Processing (JAXP) model contains three main approaches: Simple API for XML (SAX), Document Object Model (DOM), and Streaming API for XML (StAX). A fourth approach, Transformation API for XML (TrAX) can also be included if you need to transform XML documents using EXtensible Stylesheet Language Transformations (XSLT). However transformations are not the goal of the project, so TrAX is only being mentioned for completeness of the JAXP family tree. A summary of the different JAXP approaches are covered in figure 3.1.

The SAX API for parsing XML is based on a ”push model,” which means that XML data is pushed to the client as the parser encounters the elements in the XML. What this means is that data is pushed to the client whether or not the client is ready for the data. As well SAX is only capable of reading XML and not writing XML. However one benefit of SAX is that it can stream the XML document and does not require the full file to be in memory at the same time [Mic05].

The StAX API for parsing XML is based on a ”pull model,” which means that the client pulls data from the parser (calls parser methods) as the data is needed. What this means is that the client only gets XML data when it requests it. The StAX pull method is similar to the SAX push method, but has several advantages over the push method. The client is in control of the

flow of incoming data, the client can pull from multiple different libraries at the same time, and pull style libraries are generally smaller and easier to interact with then push style libraries. As well StAX libraries can both read and write XML documents [Mic05].

The DOM API for parsing XML is done by creating a tree of the XML document in-memory. The client can then manipulate the in-memory tree. The client can move freely throughout the tree updating, adding, or removing nodes in the tree as it goes. The DOM model is the most flexible for accessing XML elements in a document. However, it does have a drawback in that loading the whole document requires significantly more memory and possibly processing power than the SAX or StAX models [Mic05].

Feature	StAX	SAX	DOM	TrAX
API Type	Pull, streaming	Push, streaming	In memory tree	XSLT Rule
Ease of Use	High	Medium	High	Medium
XPath Capability	No	No	Yes	Yes
CPU and Memory Efficiency	Good	Good	Varies	Varies
Forward Only	Yes	Yes	No	No
Read XML	Yes	Yes	Yes	Yes
Write XML	Yes	No	Yes	Yes
Create, Read, Update, Delete	No	No	Yes	No

Table 3.1: Comparison of JAXP parsers [Mic05].

After comparing SAX, StAX, and DOM parsing methods, I ultimately decided to use the StAX method for parsing XML documents. StAX parsing provides all of the benefits of SAX parsing such as streaming XML files, and smaller processing requirements. With the added benefit of the pull model to get data only when needed and the ability to read and write XML documents. The DOM models added flexibility of being able to move freely throughout the document is not needed. Thus the extra memory usage and processing requirements of the DOM model does not make sense to use. As well keeping the parser small and efficient means that the project could more easily be ported to other platforms such as mobile.

3.2 Design

3.2.1 The Basics

The SQLite database is composed of a single table with 4 columns, an integer key, a songTitle, a composer, and a filePath to where the MusicXML file is located on disk. The database schema can be viewed or reset to its defaults using the createDB-Mac.sql script in the database folder of the project. For more information on creating or resetting the database see Appendix A.2.

The SQLite database is connected to the project through a single class, Database.java. For organization, any manipulation of the database is done through this class. Connecting to the database is done in the constructor of the class, if the connection to the database fails the project quits because of a ”Fatal Error.” The database class has separate methods for getting and manipulating the data in the database. Any other classes that want to manipulate the data in the database must ask the Database class to do so by calling one of its methods. For searching the database, to maximize matching of user input with database entries the songTitle and composer fields are selected using ”LIKE %<search text>%” for both. For security against any corruption or unwanted manipulation of the database, all INSERT/UPDATE statements are completed using prepared statements.

The GUI part of the project was built using JavaFX Scene Builder 2.0. The project consists of two scenes, the main GUI window is composed of a TableView to view the contents of the database, a TextField to search the database, and several buttons to manipulate the database. This main scene is setup and controlled by the musicXML.fxml and Controller.java files. The second scene shows details of a MusicXML file in the database and is brought up when a user clicks the ”Add Song” or ”Edit Details” buttons that are located in the main scene. The secondary scene is setup and controlled by the mXMLDetails.fxml and MXMLDetailsController.java files. The Scene Builder application provides a WYSIWYG interface for building GUIs, more information on setting up and using Scene Builder can be view in Appendix A.5.

In the main scene, the TextField is used for searching the database and uses an event listener to detect every time a character is added to or removed from the field. This way the database can be filtered with each new character. The TableView also uses an event listener so that not only mouse clicks are detected, but keyboard input as well. This lets users scroll through the

TableView with the up/down arrow keys. All other controls in the scenes use event handlers to handle mouse clicks.

3.2.2 Parsing, Storage, & Exporting

Parsing MusicXML files is done using the WoodStox API, which is built on top of the StAX API. The main parser for the application is located in the XMLParser.java class file, which is found in the parser package. The startParsing() method uses the WoodStox API to open up an XMLStreamReader2 stream (a file stream) tailored for reading XML files; in this case MusicXML files. Parsing of an XML file is done using a while loop and checking if the XMLStreamReader2 has another XML element to give to the client. Parsing is complete when there are no more elements in the stream (i.e. the loop finishes), getElements() is the static method that hosts this main while loop. At first glance, the getElements() method seems overly simple as there appears to be no actual if-statements in the while loop to catch specific elements in the XML. This was done on purpose to avoid a giant while loop that is filled with the if-statements of every element that is searched for in a MusicXML file, this keeps the code clean, and organized. It also generalizes the code, meaning that the getElements() method could be used for any XML document.

The getElements() method takes in three arguments; the XMLStreamReader2, which contains the location in the XML document that is currently being streamed, and two Interfaces. The two Interface arguments are the structures that enable the getElements() method to avoid using any embedded if-statements. Interfaces are Javas way of passing methods as arguments to other methods. There is a startMethod Interface and an endMethod Interface. The startMethod Interface is called when a new START_ELEMENT has been found in the XML by the streamReader, and the endMethod Interface is called when an END_ELEMENT has been found. The startMethods that are passed into the getElements() method are where the if-statements are located that look for specific elements in the XML. When an if-statement in the code locates an element of the XML that contains a subtree, the getElements() static method is called again; however, the Interfaces that are passed are specific to the subtree that was just located. This means that every subtree in a MusicXML file generates a call to the getElements() method, however, the if-statements that are used are context specific to search for subtrees that are only

possible within the current subtree, thereby improving run-time. In the trivial case, when an element is encountered that does not contain a subtree - a simple element containing only data - the elements data is simply collected and stored appropriately. When the parsing of a subtree is complete, the endMethod breaks the while loop which will force the return of the current call of getElements(). If parsing of the whole document is not yet complete, parsing of the parent subtree of the subtree that just finished will continue. For a better understanding of the getElements() method it can be viewed in Appendix B.

One issue I encountered while parsing MusicXML files was how to store the data efficiently. Originally I was breaking down the XML data into classes based on the structure of the MusicXML XSD, with major subtrees getting there own class and then arbitrarily breaking down minor subtrees into their own classes or not based on how much data the subtree potentially holds. This method of parsing proved to be very time consuming, it also created a lot of random classes that were not reusable.

I solved this storage issue by generalizing how the XML data is stored. The data is now broken down and stored in three different classes, Attributes, Elements, and ComplexElements. XML tags contain attributes in the same way that HTML contains attributes. Thus the Element class contains a list of Attributes, as well as the data stored between the tags. The Element class only holds a simple element, i.e. it holds the data for an XML tag that does not contain a subtree. Where as the ComplexElement class contains a list of Attributes, as well as a list of the Elements/ComplexElements that are contained within the current element.

For exporting the parsed MusicXML data back to a file, the Woodstox API is used again. The ExportXML.java class uses Woodstox to create an XMLStreamWriter that is linked to a file that the user has selected. Then the Element and ComplexElement objects created while parsing a document are looped through and written to the file by calling methods from the XMLStreamWriter. Looping through the Element and ComplexElement objects is possible because they are stored sequentially in the order that they were first created in.

3.3 Design Pattern

JavaFX lends itself nicely to the Model-View-Controller (MVC) design pattern. In the MVC design pattern, the model manages logic and data, the view displays information to the user,

and the controller is the link between the model and the view. The controller takes input and then tells either the model or the view what to do. Using JavaFX, the model consists of application-specific objects that manipulate data, the view consists of a FXML file, and the controller consists of a Java class that implements an initialize() method [Ora14b]. For this project the model objects consist of the database as well as the XMLParser. Then there are two views, one for the main scene and one for a secondary scene. As well, there are also two controllers, one for each view.

3.4 Testing

Testing of the GUI was done manually as there are only three buttons and one text field to test. Manipulation of the database was tested using the GUI interface and then checking the results in the tableView, as well as manually checking the database by loading it in sqlite3. For testing the parser and exporter functionalities of the application, the two parts were built in conjunction with each other. Building the two parts at the same time let me build in some new functionality to the parser and then export the parsed data to a file. I was then able to use the diff command (diff -w <file1> <file2>) to compare the newly created file with the original file that was parsed in. As well for testing I created several XML files, these files are not valid MusicXML files, however they do contain every Element and attribute possible in the MusicXML format. These testing files are located in MusicXMLFiles/Testing/ directory. For further testing of the parser and export functions I also tested parsing and exporting a lot of the valid MusicXML files that are in the MusicXMLFiles folder.

Chapter 4

Results

The results section outlines what the project accomplished as well as ideas that could be implemented in the future to improve the project.

4.1 Achieved

Each point below specifies one aspect of the project that was achieved in the project:

- When the application starts up it connects to an SQLite database and selects all of the entries in alphabetical order by song title. If the database cannot be found or accessed for some reason, the application quits due to a fatal error.
- Once the main GUI window of the application loads, the contents of the database are displayed in a tableView.
- When a user selects an item in the tableView, the contents of the file are automatically parsed. When parsing of the file is complete, some XML data of the file is displayed in the textArea. If a user does not have permission to read a file, or the file does not exist, an error is displayed in the textArea.
- Users can filter/search the tableView (database) by simply typing in the search field. There is no search button to push, each character added to or removed from the search field changes the search results.

- The ”Add Song” button allows user to add new entries to the database. Clicking the button opens up a new window where users can enter the song title, the composer and the file path to the location of the MusicXML file on disk.

- The ”Edit Details” button allows users to edit an entry of a selected item in the tableView. Clicking the button opens up a new window where users can edit the details of an existing entry in the database. If no item is selected in the tableView, clicking the button will do nothing.

- The ”Export Song” button allows users to export a selected item from the tableView. Clicking the button opens up the operating systems fileChooser, allowing users to choose a file path of where the MusicXML data should be exported to. Once a file location is selected the MusicXML data is exported to the file. If a user does not have permission export to the location that they selected the textArea displays an error.

4.2 Future Work

Several things could be done to enhance this project and continue it moving forward. First, a simple advancement to the project could be to implement an advanced search. Currently when a user types in the search field both the songTitle and the composer field of the database are searched with the same text string that user has input. An advanced search mode could be added to the project that would let users search each field independently.

Another feature that may or may not be wanted, is the ability to pretty-print the exported XML file. Currently the WoodStox API exports the XML data to a file with no line breaks in it. While this does create smaller XML files, it also makes the XML very difficult to read. However, there is no need for users to be reading straight XML data. The only issue of the exported file currently not being pretty-printed is that I had use an application to pretty-print the file for me before I could use the diff command while testing.

An additional feature that could be added to the project could be the added support for MusicXML’s compressed file format. Currently the application only supports plain text MusicXML files, i.e. *.xml files. As of MusicXML 2.0 a compressed MusicXML format was introduced. This compressed format reduces the size of MusicXML files and uses a *.mxl file

extension. As described on MusicXML's website the compression is based on a zip XML format, as well this zip format is compatible with the java.util.zip package [Mak16]. Compressed MXL files have a specific file structure, each MXL file must have a META-INF/container.xml file. This container file defines where the MusicXML file is located in the zip archive, as well it defines if there are any other files in archive, where these files are located and the media-type of the file. The MXL zip archive may also hold PDFs, music files, and images files [Mak16].

In order for the MXL format to be supported some thought will have to be put in on how to handle the files. Such as Will the MXL file be unzipped to a temporary location in order to get and read the MusicXML file, or is it possible to read the MusicXML file while it is still archived? As well, will exporting MusicXML data to the MXL format be supported? For complete details and specifications of the MXL format see MusicXML's website at `http://www.musicxml.com/tutorial/compressed-mxl-files/`.

The most useful feature that could added to this project would be the ability to display the parsed MusicXML data as sheet music. I believe a good place to start with this would be to use JavaFX's Canvas API to draw the sheet music. The parsed MusicXML data contains a lot of information about where items such as notes should be displayed on a canvas using x and y coordinates. Starting with a MusicXML's header section, the defaults element contains sub-elements that have display information about the system, the page, and the staff layouts. As well, the defaults section contains information about font families and how other aspects of the music should be displayed document wide. In the part/measure section of a MusicXML file each measure element (for partwise-scores), or each part (for timewise-scores) can contain a print element that defines general display parameters. On a more fine tuned scale the attributes of elements such as notes define specific or relative coordinates of where they should be placed as well. Anyone that plans on implementing displaying of MusicXML files should become familiar with MusicXMLs XSD as it contains all of the information about which elements and attributes define how things should be displayed. A readable and searchable version of the XSD is located in MusicXMLs user manual found at `http://usermanuals.musicxml.com/MusicXML/MusicXML.htm`.

4.3 Conclusion

The MusicXML format has become very popular with musicians for creating sheet music digitally. This project created a way to manage MusicXML files by storing their locations in an easy to use database. This project also looked at different ways to parse and store MusicXML data in java objects, ultimately implementing the best solution of each based on the current needs. Hopefully this project is continued and expanded upon in the future.

Bibliography

[Bes16] BestSolution. e(fx)clipse - javafx tooling and runtime for eclipse and osgi, 2016.

[Con16] SQLite Consortium. Sql database engine. https://www.sqlite.org, 2016.

[Dat16] Refsnes Data. W3schools: Xml tutorial. http://www.w3schools.com/xml/default.asp, 2016.

[Fas09] FasterXML. Woodstox home. http://wiki.fasterxml.com/WoodstoxHome, 2009.

[Fas16] FasterXML. Github: Woodstox, the gold standard stax xml api implementation. https://github.com/FasterXML/woodstox, 2016.

[Git16] GitHub. Github, how people build software. https://github.com, 2016.

[Jet16] JetBrains. Jetbrains s.r.o: Development tools for professionals and teams. https://www.jetbrains.com/idea/, 2016.

[Lim16] Writelatex Limited. Overleaf: Real-time collaborative writing and publishing tools with integrated pdf preview. https://www.overleaf.com, 2016.

[Mak16] MakeMusic. Musicxml for exchanging digital sheet music. http://www.musicxml.com, 2016.

[Mic05] Sun Microsystems. Why stax? https://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/tutorial/doc/SJSXP2.html, 2005.

[Ora14a] Oracle. Getting started with javafx. http://docs.oracle.com/javase/8/javafx/get-started-tutorial/index.html, 2014.

[Ora14b] Oracle. Implementing javafx best practices. http://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.htm, 2014.

[Ora16] Oracle. Javafx faq. http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html, 2016.

[Qui16] Liam Quin. W3c: Extensible markup language. https://www.w3.org/XML/, 2016.

Appendix A

Setup & Install

A.1 Running the Project

The project was created using the following software:

- macOS Sierra, v10.12.2
- SQLite3 v3.14.0
- IntelliJ IDEA 2016.2.5
  - JRE: 1.8.0_112-release-287-b2 x86_64
  - JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

Setup:

- Import the project into IntelliJ:
  - Select ”Import Project”
  - Select the folder where the project is located, click ok.
  - Select ”Create project from existing sources” click next
  - Select a project name, click next. It is fine to override the existing .idea folder.
  - Ensure the src folder is selected, click next.
  - Ensure the libraries are selected, click next.

  - Ensure the modules are selected, click next.
  - Select a JDK (I am using 1.8)
  - Click finish.

- Once the project opens, if the existing .idea folder was overwritten, you must also:
  - From the Run drop-down menu select ”Edit Configurations...”
  - Select the ”+” to create a new configuration
  - Select ”Application”
  - At the very top of the window Name the new configuration ”Main”
  - Select the ”...” button beside the ”Main Class:” field
  - Choose the only option, should be ”Main (MusicXML)”
  - Click OK, click OK again

- Build the project.

- run the project.

Windows Compatibility:

- This project has NOT been tested on Windows. However it should still be windows compatible. All file paths in the code use relative file paths and File.separator, which gets the character used by the operating system to separate directories. I am not aware of any other Java dependencies that are operating system specific.

- However the current sample database that is supplied with the project may not work properly as all of the file paths for the sample *.xml files use ”/” to separate directories. This can be easily corrected though.

If a developer would like to create a similar project using some or all of the same tools, the instructions on how to setup these tools are covered in the sections below.

A.2 Creating an SQLite Database

Use the instructions here to create a new database or to reset the projects current database back to its default.
It is being assumed that SQLite is installed, if it is not installed go to `https://sqlite.org/download.html` to download it.

- In terminal or command prompt navigate to the folder where you would like the database to be located. Or if resetting the projects database navigate to the folder where the musicXML.db file is located.
- Run the command "sqlite3 musicXML.db" this will create or open the file musicXML.db with sqlite3.
- Run the command ".read <file path>/createDB-Mac.sql" this will run the sql script. The script drops the musicXMLFiles table if it exists and then recreates it, and then populates it with some sample *.xml files. This script can be edited to suite your needs.
- Finally run the command ".exit" to close the connection to the database.

NOTE: The file paths for the sample *.xml files in the createDB-Mac.sql file are relative file paths to the projects directory, and are currently formatted to work with macOS.

A.3 Connecting a SQLite DB to a Java Project

In order to work with and use an sqlite database inside of a java project the SQLiteJDBC library is required. To connect the library to a project:

- Download the sqlite-jdbc jar file from: `https://bitbucket.org/xerial/sqlite-jdbc/downloads`. For this project I am using version 3.8.11.2.
- Place the downloaded sqlite-jdbc jar file into the lib sub-directory of the projects directory, if a lib directory does not exist you can create it.
- Add the sqlite-jdbc jar to the classpath.

- In the project view, right click on the project and select ”Open Module Settings” from the drop-down menu.
- Ensure that you are in the Modules section, then select the dependencies tab.
- Click the ”+” and select JARs or directories...
- Find and select the sqlite-jdbc jar file that was downloaded earlier.
- Click ”OK” to apply the changes and close all of the dialog boxes.

- To use the sqlite-jdbc library to connect to an sqlite database use the following lines:

```
Class.forName("org.sqlite.JDBC");
Connection database =
  DriverManager.getConnection("jdbc:sqlite:<database location>");
```

A.4 Adding the Woodstox API to a Java Project

Information about the Woodstock API can be found on their Github page `https://github.com/FasterXML/woodstox`

- Woodstox used to be available from the Codehaus website, however Codehaus is no more. Woodstox can now be downloaded using Maven directly inside of IntelliJ.
- If you do not have Maven installed (”mvn −−version” to check):
  - If you are on mac and have Homebrew installed (`http://brew.sh`), you can simply run ”brew install maven” to install it.
  - Otherwise, go to `https://maven.apache.org/download.cgi` to download Maven.
  - Installation instructions can be found at `https://maven.apache.org/install.html` the instructions are also reproduced here:
  - Unpack the archive that was downloaded above where you would like to store the binaries.
  - A directory called ”apache-maven-3.x.y” will be created.
  - Add the bin directory to your PATH, eg:

    * Unix-based operating systems (Linux, Solaris and Mac OS X)

      ```
      export PATH=/usr/local/apache-maven-3.x.y/bin:$PATH
      ```

    * Windows

      ```
      set PATH="c:\program files\apache-maven-3.x.y\bin";%PATH%
      ```

  - Make sure `JAVA_HOME` is set to the location of your JDK
  - Run "mvn --version" to verify that it is correctly installed.

- Now that Maven is installed, inside of IntelliJ, in the project view, right click on the project and select "Open Module Settings" from the drop-down menu.
- Ensure that you are in the Modules section, then select the dependencies tab.
- Click the "+" and select Library... -> From Maven...
- Search for "com.fasterxml.woodstox"
- Select the version of Woodstox wanted. I am using "com.fasterxml.woodstox:woodstox-core:5.0.3" for this project.
- To download the JAR files, check the "Download to:" box and select a location to download the files to. I suggest that the files are download to the lib directory of the project.
- Click "OK" to apply the changes and close all of the dialog boxes.
- A basic example of using the Woodstock API can be found at `http://www.studytrails.com/java/xml/woodstox/java-xml-stax-woodstox-basic-parsing/`

A.5 Using JavaFX Scene Builder

- JavaFX Scene Builder 2.0 for Windows, Mac, and Linux can be downloaded from `http://www.oracle.com/technetwork/java/javafxscenebuilder-1x-archive-2199384.html`.
- Installing Scene Builder is as simple as running the downloaded file.

- Once Scene Builder has been installed it can be accessed on it own or through IntelliJ.
- To access Scene Builder from inside IntelliJ, create a new JavaFX project or open an existing JavaFX project.
- To edit a scene using Scene Builder right click on the *.FXML file and select Open In Scene builder. This will open Scene Builder and load the FXML file.
- If the JavaFX project does not have a *.FXML file, Scene Builder can be opened on its own and used to create a new one. The created FXML file can then be added to an IntelliJ project.

Appendix B

Get Elements Method

Located here is the getElements() method, which is the main driver for parsing a MusicXML document. It is being included as a reference to help readers better understand how the application parses files.

```
1  public static void getElements(XMLStreamReader2 xmlStreamReader,
2                                 Interface startMethod, Interface endMethod) {
3    try {
4      wLoop: while(xmlStreamReader.hasNext()){
5        int eventType = xmlStreamReader.next();
6        switch (eventType) {
7          case XMLEvent.START_ELEMENT:
8            if (startMethod.interfaceMethod()) {
9              break wLoop;
10           }
11           break;
12         case XMLEvent.END_ELEMENT:
13           if (endMethod.interfaceMethod()) {
14             break wLoop;
15           }
16           break;
17         default:
```

```
18                 break;
19             }
20         }
21     } catch (XMLStreamException e) {
22         //e.printStackTrace();
23         System.out.println("ERROR: Streaming Error");
24     }
25 }
```

Appendix C

Acronyms

Many acronyms are used in this document, for a quick reference they are listed here:

API - Application Program Interface

DOM - Document Object Model

DTD - Document Type Definition

GUI - Graphic User Interface

HTML - HyperText Markup Language

IDE - Integrated Development Environment

JAXP - Java API for XML Processing

MVC - ModelViewController

RIA - Rich Internet Application

SAX - Simple API for XML

StAX - Streaming API for XML

TrAX - Transformation API for XML

VCS - Version Control System

WYSIWYG - What You See Is What You Get

XML - EXtensible Markup Language

XSD - XML Schema Definition

XSLT - EXtensible Stylesheet Language Transformations