

종합설계 프로젝트 수행 보고서

프로젝트명	작곡 입문자를 위한 AI 작곡 어플리케이션
팀번호	S1-12
문서제목	수행계획서() 2차발표 중간보고서() 3차발표 중간보고서() 4차발표 보고서() 최종 보고서(O)

2020.12.04

팀원 : 하정현(팀장)

조현민

최정환

이동호

지도교수 :

서 대 영

지도교수 :

최 종 필

(인)

Seo, Hyeon

문서 수정 내역

작성일	대표작성자	버전(Revision)	수정내용	
2020.01.20	하정현(팀장)	1.0	진행보고서	최초작성
2020.02.24	하정현(팀장)	1.1	2차 진행보고서	설계서추가
2020.04.27	하정현(팀장)	1.2	3차 진행 보고서	프로토타입 내용 추가 및 디버깅 설계도 추가, 일부분 설계도 수정
2020.06.25	하정현(팀장)	1.3	4차 진행 보고서	AI 관련 기능 수정
2020.11.22	하정현(팀장)	1.4	최종 보고서	연구결과 및 소요 재료 서술

문서 구성

진행단계	프로젝트 계획서 발표	중간발표1 (2월)	중간발표2 (5월)	학기말발표 (6월)	최종발표 (11 월)
기본양식	계획서 양식	계획서 양식	계획서 양식	계획서 양식	계획서 양식
포함되는 내용	I. 서론 (1~6) II. 본론 (1~3) 참고자료	I. 서론 (1~6) II. 본론 (1~4) 참고자료	I. 서론 (1~6) II. 본론 (1~5) 참고자료	I. 서론 (1~6) II. 본론 (1~5) 참고자료	I II III

이 문서는 한국산업기술대학교 컴퓨터공학부의
 “종합설계”교과목에서 프로젝트“작곡 입문자를 위한 AI 작곡
 어플리케이션”을 수행하는
 S1-12, 하정현, 조현민, 최정환, 이동호 학생이 작성한 것으로
 사용하기 위해서는 팀원들의 허락이 필요합니다.

목 차

I. 서론

1. 작품선정 배경 및 필요성	4
2. 기존 연구/기술동향 분석	4
3. 개발 목표	4
4. 팀 역할 분담	4
5. 개발 일정	5
6. 개발 환경	6
6.1. 플랫폼 개발	
6.2. 디버깅 환경	
6.3. 외부 디바이스	
6.4. 외부 라이브러리 및 참고 오픈소스	

II. 본론

1. 개발 내용	7
1.0. 기능 개요	
1.1. 초기 설정	
1.2. 자동 작곡 수행	
1.3. 일반 작곡 수행	
1.4. 미디 디바이스 활용	
1.5. 저장 기능	
2. 문제 및 해결방안	8
3. 시험시나리오	9
4. 상세 설계	11
4.1. 시스템 전체 구성도(클라이언트)	11
4.2. 시스템 전체 구성도(서버)	13
4.3. 시스템 세부 구성도(클라이언트)	14
4.3.1. HarmoVisualSheet	14
4.3.2. HarmoDataModel	21
4.3.3. HarmoStatisticModule	24
4.3.4. HarmoControl	24
4.3.5. MidiDeviceModule	26
4.3.6. Save Module	29
4.3.7. AIClientTask	40
4.4. 시스템 세부 구성도(서버)	41
4.4.0 Config	41
4.4.1 ConnectionModule	41
4.4.2 DataTranslator	42

4.3.3 ClientManagement	42
4.3.4 DeepLearningManagement	43
4.3.5 DeepLearningChecker	44
4.3.6 DLProcess	45
4.5. Server-Client 통신 설계서	45
4.5.0. 개요	45
4.5.1. Redis	46
4.5.2. 전체 구성도	46
4.5.3. 프로토콜 및 DB 데이터	47
5. 프로토타입	52
III. 결과	
1. 연구결과	
2. 소요 재료	
참고 자료	56
Github Project	56

I. 서론

1. 작품선정 배경 및 필요성

작곡을 처음 배우는 입문자의 경우, 화음이 어떻게 배치되어야 하는지, 또는 화음이 왜 필요한지 이해하지 못하는 경우가 많아 듣기에 어색한 음악을 만드는 경우가 많다. 화성학 교재를 통해 화성학을 제대로 익힌 경우에는 문제가 되지 않으나, 이를 배우는 과정이 까다롭고 어려워 도중에 포기하는 입문자가 많다는 점을 고려해 작곡 입문자를 위한 애플리케이션을 개발하게 되었다. 이 애플리케이션은 화음 위주의 작곡 방식에 대한 학습을 목적으로 했으며, 화음을 중심으로 작곡을 수행시킨 후 사용자에게 적절한 화음을 제시해 보다 빠르고 친숙하게 사용자가 학습에 용이하도록 하는 것이 주 목적이다.

2. 기존 연구/기술동향 분석

Magenta Project

머신러닝을 이용한 예술 창작 학습 AI 알고리즘을 설계하는 오픈소스 프로젝트인 '마젠타 프로젝트'는 음악 분야일 경우, 1천여 가지 악기와 30여만 가지의 음이 담긴 데이터베이스를 구축하고 이를 AI에 학습시켜 새로운 소리, 음악을 만들어낸다. 알고리즘은 순환신경망 네트워크(RNN)를 사용. 그러나 이 프로젝트는 현재 음악에서의 기승전결 같은 요소는 만들지는 못한다.

마젠타 프로젝트는 RNN기법 중 Sketch-RNN을 사용하는데, 이 모델은 추상적인 개념을 일반화 시켜 그림으로 표현할 수 있다. 즉 그림일 경우, 사용자가 낙서를 그리면 이 모델은 기존에 그려진 낙서를 바탕으로 그림을 그린다. 이 기술은 현재 Google에서 연구 중이며, 초기 모델이다. 이 모델은 음악에서도 적용이 가능한데, 예를 들어 사용자가 앞에서 작곡한 내용을 바탕으로 다음의 곡을 작곡할 수 있다.

(Sketch-RNN github: https://github.com/tensorflow/magenta/tree/master/magenta/models/sketch_rnn)

3. 개발 목표

전문가가 아닌 입문자를 목표로 한 어플리케이션으로 화음 위주로 작곡 또는 교정을 수행한 후, 사용자에게 이 기능을 수행하는 것에 대한 정보를 사용자에게 상기시킴으로써, 학습이 가능하게 하는 것이 목적

4. 팀 역할 분담

01월 18일 이전

Front End	App의 UI 및 악보 인터페이스	최정환
Back End	Midi Module 및 기타 API를 이용한 기능 구현	이동호
	Midi Data를 조정하는 Module 개발 및 DB 설계/구축	조현민
Deep Learning Algorithm	LSTM을 이용한 작곡 AI 신경망 구축 및 화음 알고리즘 개발	하정현

문제점: 겨울방학에 4명 중 2명이 현장실습을 나가는 상황이 발생해, 역할 분담 밸런스를 맞추기 위해, Back End 분야에 현장실습을 나가는 2명을 배치함으로써, 졸업작품 진행에 부담이 가지 않게 하려고 했으나, 01월 04일 백엔드 관련 회의 결과, 백엔드 진행 난이도가 낮은 반면에, 프론트 엔드는 어플리케이션에 그래픽 툴을 사용해 악보를 그리는 기능을 구현하는 난이도가 너무 높아, 혼자서 감당하기 어렵다는 애로사항이 발생

해결방안: Front End에 2명을 배치, Back End에 2명을 배치, 단, Deep Learning/Algorithm은 따로 두지 않고 Back End에 흡수, 각 분야마다 현장실습을 나가지 않는 사람과 나가는 사람을 같이 배치함으로써, 현장실습 나가는 사람이 나가지 않는 사람을 보조하는 역할을 하여, 졸업작품에 부담이 가지 않게 유도. 18일 이후에 변경되는 역할분담은 아래와 같다.

현재 조현민 학생은 해외 현장실습으로 인하여 소통이 원활하게 되지 않는 상황이므로 3월부터 합류 예정

01월 18일 이후

Front End	악보그리는 API 및 알고리즘 개발	최정환
	악보를 그리는 알고리즘을 개발 및 악보를 제외한 나머지 UI 개발	이동호(현장실습)
Back End	DB 구축/설계, Midi Module개발 기능 개발	조현민(현장실습)
	AI 신경망 구축 및 화음 알고리즘 개발, Midi Device 통신 Module 개발, 미디데이터 분석	하정현

4월 현재

Front End	악보 인터페이스 개발 및 보완	최정환
	UI/UX 개발 및 관리	이동호
Back End	DB 설계 및 구현	조현민
	AI 신경망 구축 및 화음 알고리즘 개발, 미디 데이터 분석	하정현

5. 개발 일정



6. 개발 환경

6.1. 플랫폼 개발: 안드로이드 플랫폼으로 개발을 하며 스마트폰 해상도와, 태블릿 해상도로 구현을 한다. IDE는 Android-Studio를 사용하며, 언어는 Kotlin과 Java를 사용, Java는 이 프로젝트에서 사용할 외부 라이브러리의 호환을 위해 일부를 사용하고, 대부분의 어플리케이션 개발은 Kotlin을 사용한다.

6.2. 딥러닝 환경: 딥러닝 학습 코드는 Python을 사용하고 IDE는 Visual Code를 사용. 딥러닝 라이브러리는 Tensorflow 2.1과 Keras를 사용한다. 또한 원활한 학습 진행을 위해 GPU(GTX 960)을 사용 하려고 했으나.

2월 초에 Tensorflow 2.0을 사용하는 도중 최적화 작업을 무시하고 딥러닝을 수행하는 문제점을 발견, 코드상의 문제가 아닌 2.0버전 고유의 오류임을 확인하고 2.1로 업그레이드를 시도했지만.

이를 사용하는 도중 Batch Size를 줄여도 Memory Allocation Error가 발생하는 이유로, Google Platform을 사용하거나 다른 방법을 찾는 중

2월 23일 23시에 Google Platform의 GPU를 탑재한 VM 서버를 사용한 결과 아무런 문제 없이 학습을 하는 데 성공했으며 현재는 기존의 오픈소스를 어플리케이션에 맞게 다시 모델과 함수를 구현하고 데이터셋의 추가 확보와 손실 비율(val_loss)를 줄이는 데 집중하고 있음.

4월 현재 계속되는 비정상적인 Overfitting 현상으로 인해 오픈소스를 사용하지 않고 Magenta Melody_RNN API로 대신하고 있으며 딥러닝 서버(모듈)은 Melody_RNN으로 사용 중이고 동시에 오픈소스에 대한 Overfitting 문제를 해결 중에 있음

6.3. 외부 디바이스: 악보를 쉽게 컨트롤 하기위해 미디 키보드와 데이터 통신을 하는 모듈을 개발할 것이므로 Nektar LX25라는 미디 키보드를 사용해서 개발 또는 데모를 하게 된다. (악보 인터페이스의 상태에 따라서 구현이 될 지는 미지수이기 때문에 4차발표 때 데모 예정)

6.4. 외부 라이브러리 및 참고 오픈소스

미디 컨트롤 라이브러리 - <https://github.com/LeffelMania/android-midi-lib>

미디 드라이버 라이브러리 - <https://github.com/billthefarmer/mididriver>

안드로이드 차트 라이브러리 - <https://github.com/PhilJay/MPAndroidChart>

악보 오픈소스 - <https://github.com/letzteSeite/MidiSheetMusic-Android>

딥러닝 오픈소스 - <https://github.com/Re-Coma/Classical-Piano-Composer>

Magenta Melody RNN - <https://magenta.tensorflow.org/>

II. 본론

1. 개발 내용

1.0 기능 개요

- 사용자가 단순히 악보를 그리는 수준이 아니라 화음 배치 기능을 사용해서 상황에 맞는 화음을 배치를 할 수 있다.
- AI Random 기능을 사용하여 장르 및 스타일에 따른 곡을 전체적으로 작곡을 할 수

있다.

- 미디 디바이스를 사용하여 악보를 쉽게 컨트롤 할 수 있다.
- 곡 분석 테이블 기능을 이용하여 사용자가 어떻게 작곡을 하고 있는지 볼 수 있다.
- 시간이 남는다면 SNS 기능을 추가하는 것도 고려하고 있다.

1.1 초기 설정:

- 곡 제목, 빠르기, 박자, 코드, 코멘트를 입력받는다.
- 곡 제목은 128자 이하로 설정한다.
- 빠르기는 2/4, 3/4, 4/4 일 경우 4분의 4박자를 기준으로 bpm을 설정하고, 3/8, 6/8 은 점 4분음표를 기준으로 bpm을 설정한다. 그러나 실제 Midi Driver에서 악보 재생을 수행할 때는 4분음표 기준의 bpm을 사용하므로 점4분음표 기준의 bpm은 1.5를 곱해서 사용한다.
- 코드는 C Major, D Minor... 등을 의미한다.
- 코멘트는 곡에 대한 설명으로 최대 256자 까지 입력 가능하며 필요없으면 안해도 된다.

1.2 AI 작곡 수행

- AI 작곡 수행: 사용자가 작성한 멜로디와 딥러닝한 데이터를 바탕으로 해당하는 멜로디에 적합한 화음을 AI가 제공한다.
- 알고리즘 화음 배치: Deep Learning이 아닌 화성학에 기반한 알고리즘을 수행해서 화음 배치를 하는 기능이다.

1.3 일반 작곡 수행: 자동 작곡이 아닌 사용자가 직접 음표나 쉼표를 사용하여 악보에 데이터를 입력하는 기능이다.

1.4 미디 디바이스(키보드)를 사용한 작곡 수행: 미디 키보드를 활용하여 일반 작곡 수행을 할 수 있다.

1.5 저장 기능: 수동적으로 터치해서 저장하는 경우 불편할 수 있으므로 시간 Log DB를 새로 구축해서 일정 간격(Default: 30초) 마다 자동 저장을 수행한다. 수동저장도 지원.

2. 문제 및 해결방안

2.1 Back End

문제점: SQLite에서 원하는 데이터 형식을 지원하지 않는 문제 발생

--> 데이터 타입이 없기 때문에 안드로이드에서 처리시에 시간 관련된 처리가 어렵다.

해결방안: 예를들어 날짜의 경우에는 SQLite 내부 함수인 date 함수, time 함수, datetime 함수, julianday 함수, strftime 함수를 사용하면 현재 또는 지정된 날짜의 날짜와 시간을 구할 수 있다.

그러므로서 시간 테이블을 따로 만들어 해결이 가능하다.

2.2 Deep Learning

장르별로 딥러닝을 수행하게 되면 1차 발표 당시 4마디 딥러닝 작곡의 무의미하게 되므로 사용자 설정에 따라 곡 전체를 작곡하는 방향으로 추진, 그러나 작곡만 하면 어플리케이션의 목적에 어긋나므로 작곡을 한 후 곡에 대한 분석을 하여 디스플레이에 출력한다.

박자에 따라 딥러닝을 수행해야 함: Music21을 이용해 박자별로 분류하고 학습을 시행. 단, 이에 대한 데이터는 많아야 한다.

AI 기능을 어플리케이션 자체 내에서 수행하려고 했으나 용량이 큰 AI모델로 인한 용량 문제와 TensorflowLite의 적은 정보로 인해 서버에서 수행하는 것으로 결정. 현재 프로토타입 구현까지 마친 상태 3차 발표에서도 서버를 활용할 예정이다.

1차 발표 당시 제기된 문제

1. 작곡 딥러닝 알고리즘의 주제 정하기: 클래식(작곡가 별로), 발라드로 선정, 박자에 따라 결과도 달라야 하므로 2/4, 4/4와 3/8, 6/8, 3/4 박자 두 분류로 나눠서 딥러닝 수행 **4월 26일 현재 데이터셋 부족으로 박자에 따른 모델 생성은 취소 될 가능성이 있음**
2. 딥러닝 결과에 대한 평가(사용자 기준): 사용자가 좋아요를 누르면 BackPropogation을 수행하기. 그렇지 않으면 학습을 하지 않는 방식 (RNN은 분류가 아닌 회귀 방식이므로 CNN과 반대로 진행한다)
3. 딥러닝 결과에 대한 평가(데모): 장르별로 테스트를 해서 분위기가 얼마나 차이나는 지 확인

2차 발표 당시 제기된 문제

1. UI 추가 자료 필요: 2차 발표 당시 UI가 제대로 설계가 되지 않았으나. 이 부분은 프로토타입을 구현하면서 80% 이상 해결되었다.
2. **AI 작곡에 대한 평가 방법 및 평가 기준:** AI가 “정확한” 작곡을 수행하는 것에 대한 평가 방법은 지금도 논의 중이다. 차선택으로 **화음 진행도를 고려하고 있다.** AI에서 추출된 데이터들은 반주까지 포함이 아닌 멜로디만 출력되고 반주는 알고리즘으로 추가하기 때문에 화음이 불일치 되는 지는 측정하기 어렵고 화음이 추가되고 그 화음들의 진행이 정상적으로 진행되는 지 측정을 함으로써 평가를 한다. 단 이를 구현하는데 부가적인 알고리즘을 작성해야 하기 때문에 평가 기준은 아직 정하지 못했고 대부분 구현이 완료된 4차 발표쯤에 평가 기능 구현하고 평가 기준을 정해서 발표 예정이다. 그러나 **장르에 따라 작곡이 다르게 되는지에 대한 판정하는 방법 같은 경우. 이는 예술계열의 특성상 작품을 객관적으로 판단하는 것 자체가 난해하기 때문에 이 부분은 아직 결정하지 못한 상태.** 빅데이터로 작곡 성향을 판단하는 것도 고려한 적이 있었으나 이는 **데모에서는 시연이 불가능하기 때문에 취소되었다.**

3차 발표 당시 제기된 문제

1. 프로그램의 목적과 메인 기능이 명확하지 않음: 프로그램의 목적이 AI가 전체 멜로디를 작성하는 부분적인 멜로디를 작성하는지 혹은 그 외의 것인지 명확하지 않았다. 프로그램을 처음 만들 때의 목적을 다시 돌아봐서 작곡 초심자들이 어려워하는 화음배치를 AI가 도와

- 주기 위한 프로그램이라는 것을 상기시켜서 사용자가 작성한 멜로디를 바탕으로 어울리는 화음을 추천, 제공하는 기능을 메인으로 삼으려고 한다.
2. 결과물로 나온 곡을 평가하는 방법: 음악으로 수치적으로 또는 정량적으로 평가 하는 부분이 까다로워 프로그램 이용자들이 결과물로 나온 음악을 들을 수 있고 이 음악에 대한 평가를 할 수 있는 기능을 넣을 예정이다.

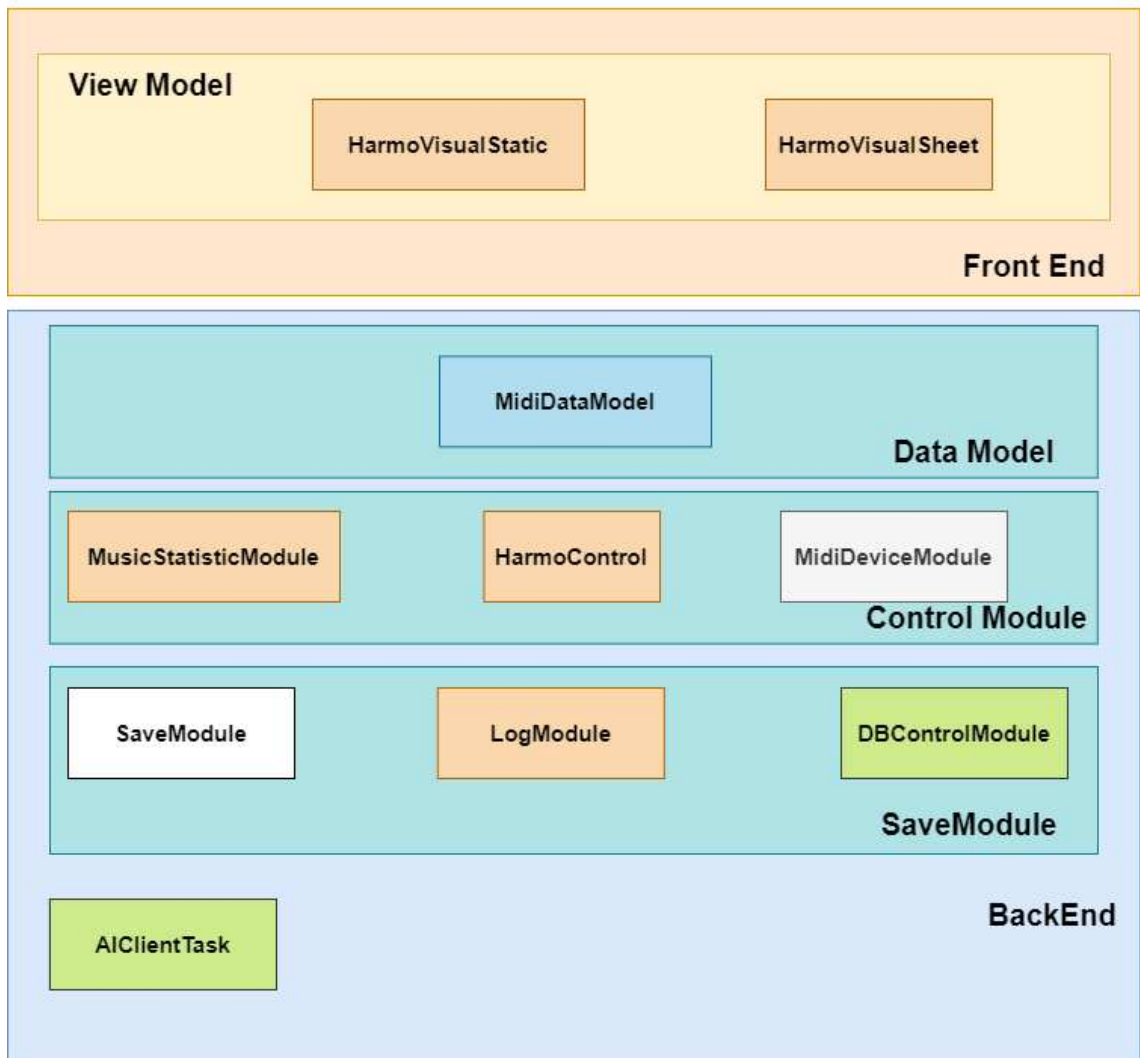
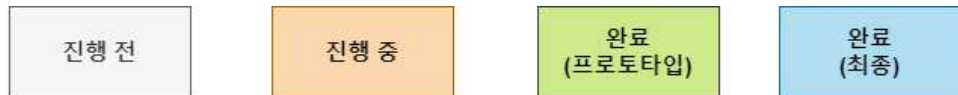
3. 시험시나리오

- 3.1 화음 배치 알고리즘을 이용한 작곡 시나리오: 해당 위치를 선택하고 항목을 선택해서 화음 배치 시도
- 3.2 AI 기능을 사용하여 해당 멜로디에 화음을 추천, 제공. 분위기를 선택하여 같은 위치에서 제공하는 화음이 달라 짐.
- 3.3 곡 분석 시나리오: 작업하고 있는 곡에 분석을 수행하여 해당 곡의 기본 정보와 음 이나 화음의 사용 여부를 통계를 낸다.
- 3.4 미디 디바이스를 이용한 음표 입력 시나리오: 음표가 선택되지 않았거나 음표가 선택이 되어도 수정 위치를 정하지 않았을 경우 해당 눌러진 키에 대한 소리가 나야 하고 위치까지 선택되었을 경우 입력까지 할 수 있게 한다.

4. 상세설계

4.1. 시스템 전체 구성도(클라이언트)

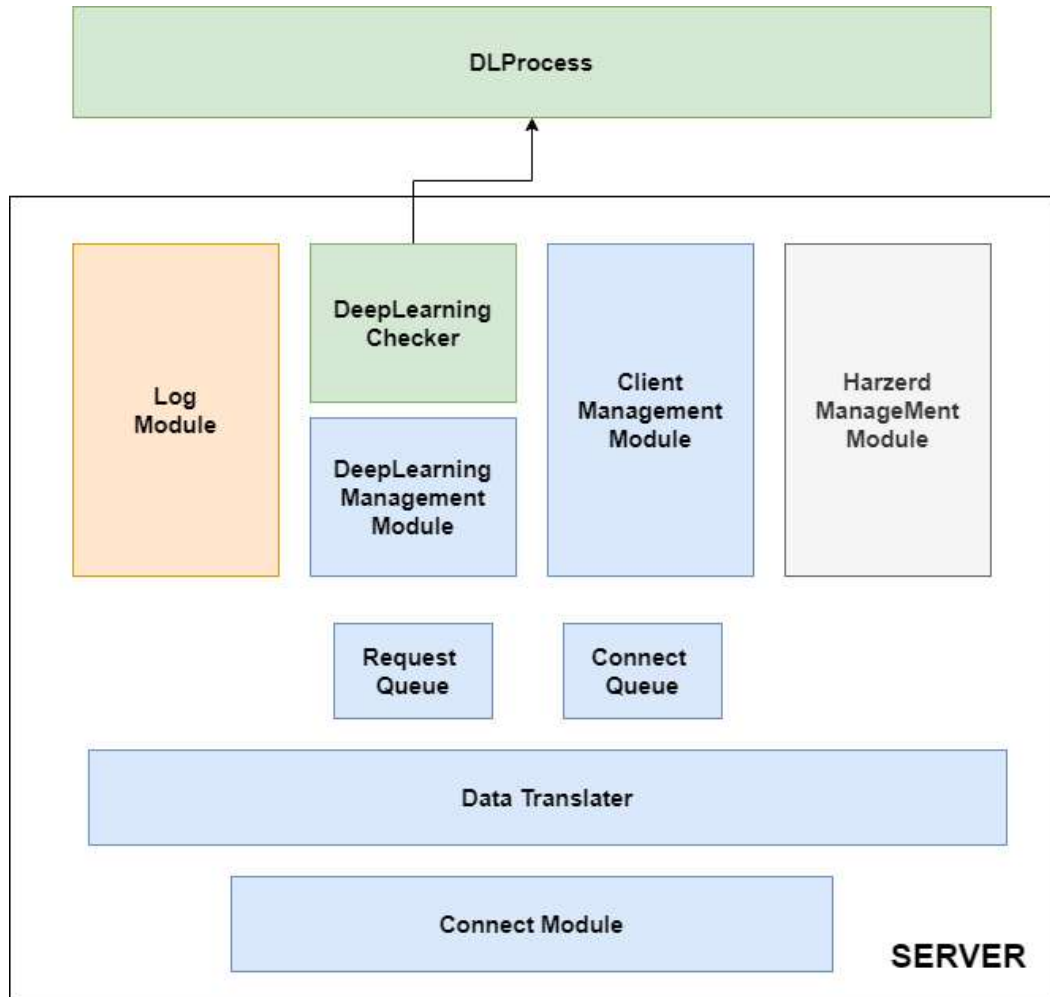
- Android Platform
- 언어: Kotlin
- 통신 프로토콜: TCP(Redis DB를 이용한 통신)



- **HarmoVisualStatic**: 곡을 분석한 결과를 출력할 때 사용되는 최상위 모듈로 Activity로부터 상속을 받는다.
- **HarmoVisualSheet**: 악보 인터페이스
- **MidiDataModel**: 미디 데이터 모델로 기본적인 연산이 가능하다. 기존의 다른 라이브러리에도 이러한 모델이 존재하나 (EX Android-Midi-Library) 어플리케이션의 목적에 맞는 모델을 따로 사용할 필요가 있기 때문에 따로 구현을 한 상태
- **MusicStatisticModule**: 곡을 분석하기 위해 사용되는 모듈로 MidiDataModel로부터 데이터를 수집하고 HarmoVisualStatic으로 결과를 제출한다.
- **HarmoControl**: 화음 배치 모듈
- **MidiDeviceModule**: 미디 키보드로부터 데이터를 받는 모듈, HarmoVisualSheet의 구현 결과에 따라서 미디 키보드 기능이 취소 될 가능성이 있기 때문에 현재는 구현이 중단된 상태
- **SaveModule**: 데이터를 저장하는 모듈로 DB를 기반으로 한다.
- **AIClientTask**: 서버와 통신하는 모듈이다.

4.2. 시스템 전체 구성도(딥러닝 서버)

- Linux Deamon Process
- 언어: Python
- 통신 프로토콜: TCP(Redis DB를 이용한 통신)
- 딥러닝 API(Magenta Melody RNN(Tensorflow))
- GPU 활용



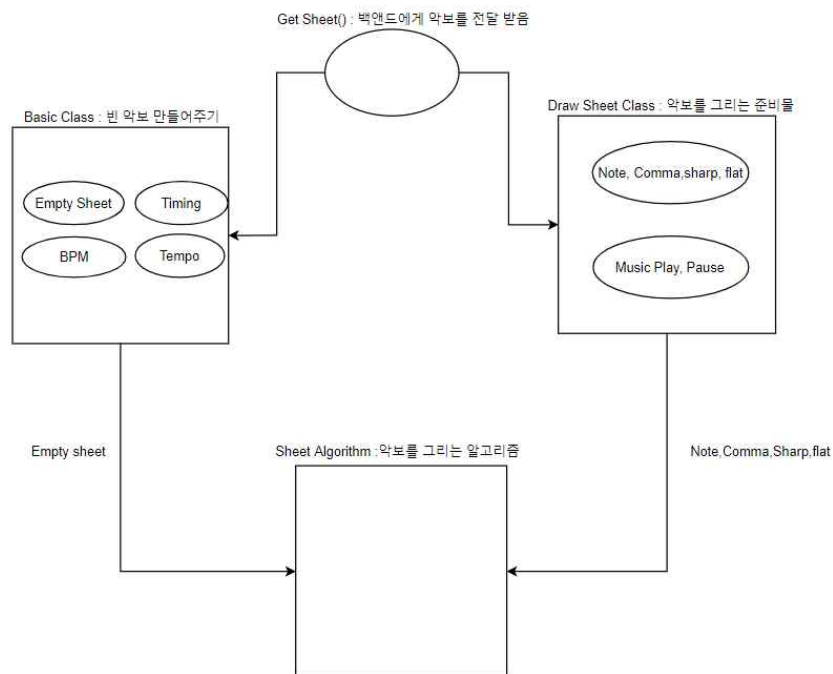
- **Connect Module:** 클라이언트(정확히는 Redis Server)로부터 데이터를 받는데 사용되는 모듈로 메인 쓰레드에서 작동한다.
- **Data Translator:** 서버는 데이터를 Json형식으로 받는다 그렇기 때문에 데이터가 정확한 프로토콜을 지켰는지 확인하기 위해 Data Translator모듈을 이용해서 판정하고 데이터가 올바르지 않은 데이터일 경우, 버린다.
- **RequestQueue, ConnectQueue:** Data Translator를 거친 데이터는 RequestData와 ConnectData로 가공되어지는 데 이를 저장하는 구조가 Queue이다.
- **ClientManagementModule:** 클라이언트의 접속 상태를 관리하는 모듈로 Thread로부터 상속을 받기 때문에 별도의 쓰레드로 작동한다.

ConnectQueue에서 데이터를 받아서 클라이언트의 연결/비연결 처리를 수행한다.

- **DeepLearningManagementModule**: 딥러닝 수행 상태를 관리하는 모듈로 Thread로부터 상속을 받기 때문에 이 또한 별도의 쓰레드로 작동한다.
- **DeepLearningChecker**: DLManagementModule에서 요청을 받으면 DeepLearningChecker로 별도의 쓰레드를 생성하면 이 모듈은 딥러닝을 수행하는 프로세스를 생성해서 끝날 때 까지 기다린 다음 결과물인 미디파일을 받아 클라이언트에게 보내고 DL Management에게 끝났다고 보고한다.
- **LogModule**: 서버가 수행 한 내용을 기록하는 모듈로 아직 설계중에 있다.
- **Harzard Module**: 서버에 이상이 생길 때 대처하는 모듈로 서버 보완이 어느정도 완료가 되면 구현 예정이다.

4.3. 세부 구성도(클라이언트)

4.3.1 HarmoVisualSheet: 악보를 그리는 모듈



- **Empty Sheet**: 빈 악보를 그린다. 악보 오선지 및 기초적인 엘리먼트 (음자리표, 박자 등) 을 찍운다.
- **Timing**: 악보 위에 올라갈 박자를 뜻한다. 엘리먼트의 일부분이다.
- **BPM**: 음악의 속도를 결정하는 엘리먼트이다. 악보 위에 별도로 출력을 한다.
- **Tempo**: 음악의 박자를 뜻하는 엘리먼트이다. Timing과 같이 사용된다.
- **Get Sheet()**: 백엔드로부터 악보를 전달받는다. 이때, 전달받는 악보의 확장자는 *.mid 파일로 Midi 파일을 뜻한다.
- **Note**: 음표를 뜻하는 엘리먼트이다. 악보 위에 올라갈 주요 엘리먼트이며 4 분 음표 외에도 8분 음표, 16분 음표, 32분 음표 등 다양한 바리에이션을 지원한다.
- **Comma**: 쉼표를 뜻하는 엘리먼트이다. 악보 위에 올라갈 주요 엘리먼트이며 4 분

쉼표 외에도 8분 쉼표, 16분 쉼표, 32분 쉼표 등 다양한 바리에이션을 지원한다.

- **Sharp**: 샵 (#) 을 뜻한다. 악보의 처음, 또는 음표의 뒤에 붙는 엘리먼트이다.
- **Flat**: 플랫을 뜻한다. 악보의 처음, 또는 음표의 뒤에 붙는 엘리먼트이다.
- **Sheet Algorithm**: 악보를 그리는 알고리즘을 뜻한다. 위에 서술된 엘리먼트들을 조합하여 최종적으로 사람이 인식할 수 있는 악보를 만들게 된다.

- 코드리뷰

에디터에서 생성된 노트가 재생될 수 있도록 한다.

MidiEvent class형식을 가진 note 여러 개가 모여 하나의 MidiTrack 클래스인 track이 되고 여러 개의 track이 모여 tracks가 되는 구조. allevents는 모든 트랙의 노트정보를 담고 있다.

```
1757 +     public void recalculateEvents() {  
1758 +         this.allevents = this.tracks.stream().map(MidiTrack::recalculateMidiEvent)  
1759 +             .collect(Collectors.toCollection(ArrayList::new));  
1760 +     }
```

recalculateEvents 함수는 java의 stream 모듈을 이용하여 각 트랙에 recalculateMidiEvent 함수를 실행시키서 MidiEvent를 다시 계산해 낸다.

```

158 +     public ArrayList<MidiEvent> recalculateMidiEvent() {
159 +         ArrayList<MidiEvent> events = new ArrayList<>();
160 +
161 +         this.getNotes().sort((n1, n2) -> n1.getStartTime() - n2.getStartTime());
162 +
163 +         int prevEndTime = 0;
164 +         for (MidiNote note : this.getNotes()) {
165 +             int delta = prevEndTime - note.getStartTime();
166 +             MidiEvent noteOnEvent = new MidiEvent();
167 +             noteOnEvent.DeltaTime = delta;
168 +             noteOnEvent.StartTime = note.getStartTime();
169 +             noteOnEvent.HasEventflag = true;
170 +             noteOnEvent.EventFlag = (byte) 0x90;
171 +             noteOnEvent.Channel = (byte) note.getChannel();
172 +             noteOnEvent.NoteNumber = (byte) note.getNumber();
173 +             noteOnEvent.Velocity = 127;
174 +             noteOnEvent.Instrument = 0;
175 +             noteOnEvent.KeyPressure = 64;
176 +             noteOnEvent.ChanPressure = 64;
177 +             noteOnEvent.ControlNum = 0;
178 +             noteOnEvent.ControlValue = 0;
179 +             noteOnEvent.PitchBend = 0;
180 +             noteOnEvent.Numerator = 0;
181 +             noteOnEvent.Denominator = 0;
182 +             noteOnEvent.Tempo = 0;
183 +             noteOnEvent.Metaevent = 0;
184 +             noteOnEvent.Metalength = 0;
185 +             noteOnEvent.Value = new byte[] {};
186 +
187 +             events.add(noteOnEvent);

```

즉, 위 함수는 트랙 하나에 있는 모든 노트를 for문으로 돌려서 각 노트의 값들을 다시 세팅해주는 함수이다.

- 노트를 드래그하여 높낮이를 조절할 수 있게 한다.

```

18 +     public void moveNote(int downX, int downY, int upY);

```

movenote 함수를 호출한다.


```

1467 +     @Override
1468 +     public void moveNote(int downX, int downY, int upY) {
1469 +         if (isEditMode) {
1470 +             // FIXME: 잘못된 액세스 방식.
1471 +             // FIXME: 1번 트랙의 노트만 삭제됨.
1472 +             int pulseTime = player.sheet.PulseTimeForPoint(new Point(scrollX + downX, scrollY + downY));
1473 +             MidiNote note = this.player.midifile.getTracks().get(0).findNoteByPulse(pulseTime);
1474 +
1475 +             if (note != null) {
1476 +                 Log.d("SheetMusic", "note found: " + note.toString());
1477 +                 int delta = upY - downY;
1478 +                 note.setNumber(note.getNumber() - (delta / 10));
1479 +                 this.sheetMusicRequestListener.onRefreshRequest();
1480 +             }
1481 +         }
1482 +     }

```

클릭된 포인트 값을 입력 받아서 pulseTime값을 계산한다.

그 다음 pulseTime값을 가지고 findNoteByPulse함수를 실행하여 해당 노트를 찾는다. 해당 노트를 찾았으면(note != null) 드래그 전 후 차이 값을(delta)를 구하고 노트의 키음을 변경된 키음으로 다시 설정한다.(note.setNumber) 반영된 키음으로 악보도 재생성.(sheetMusicRequestListener)

● 노트를 삭제하는 기능을 추가한다.

```

221 +     MidiNote findNoteByPulse(int pulse) {
222 +         for (MidiNote note : this.getNotes()) {
223 +             if (note.getStartTime() == pulse) {
224 +                 return note;
225 +             }
226 +         }
227 +         return null;
228 +     }

```

```

72 +     public interface SheetMusicRequestListener {
73         void onNoteAddRequest(int trackNum, MidiNote midiNote);
74 +     void onRefreshRequest();

```

```

111 +     private SheetMusicRequestListener sheetMusicRequestListener = null;

```

```

135 +     public void init(MidiFile file, MidiOptions options, SheetMusicRequestListener sheetMusicRequestListener) {

```

```

140 +         this.sheetMusicRequestListener = sheetMusicRequestListener;

```

```

1448 +         if (isEditMode) {
1449 +             // FIXME: 잘못된 액세스 방식.
1450 +             // FIXME: 1번 트랙의 노트만 삭제됨.
1451 +             int pulseTime = player.sheet.PulseTimeForPoint(new Point(scrollX + x, scrollY + y));
1452 +             MidiNote note = this.player.midifile.getTracks().get(0).findNoteByPulse(pulseTime);
1453 +
1454 +             if (note != null) {
1455 +                 Log.d("SheetMusic", "note found: " + note.toString());
1456 +                 Log.d("SheetMusic", "removing note from sheet");
1457 +                 this.player.midifile.getTracks().get(0).getNotes().remove(note);
1458 +                 this.sheetMusicRequestListener.onRefreshRequest();
1459 +             }
1460 +             return;
1461 +         }

```

```

62 + public class SheetMusicActivity extends MidiHandlingActivity implements SheetMusic.SheetMusicRequestListener {

```

```

221 +         this.midifile.recalculateEvents();

```

```

525 +     @Override
526 +     public void onRefreshRequest() {
527 +         createSheetMusic(options);
528 +         this.midifile.recalculateEvents();
529 +     }

```

moveNote 함수와 같은 방식으로 해당 위치의 노트를 찾아서 그 노트를 삭제한다 (remove)

마찬가지로 변경을 반영하기 위해 악보를 refresh. (onRefreshRequest)

onRefreshRequest는 첫 장에 나와있는 recalculateEvents를 사용하여 노트 정보를 다시 할당하여 변경된 노트 정보가 반영된 새로운 악보를 생성한다.

```

210 +     public void reinit(MidiOptions options) {
211 +         MidiFile file = player.midifile;
212 +         zoom = 1.0f;
213 +
214 +         paint = new Paint();
215 +         paint.setTextSize(12.0f);
216 +         Typeface typeface = Typeface.create(paint.getTypeface(), Typeface.NORMAL);
217 +         paint.setTypeface(typeface);
218 +         paint.setColor(Color.BLACK);
219 +
220 +         ArrayList<MidiTrack> tracks = file.ChangeMidiNotes(options);
221 +         scrollVert = options.scrollVert;
222 +         showNoteLetters = options.showNoteLetters;
223 +         TimeSignature time = file.getTime();
224 +         if (options.time != null) {
225 +             time = options.time;
226 +         }
227 +         if (options.key == -1) {
228 +             mainkey = GetKeySignature(tracks);
229 +         }
230 +         else {
231 +             mainkey = new KeySignature(options.key);
232 +         }
233 +         numtracks = tracks.size();
234 +
235 +         int lastStart = file.EndTime() + options.shifttime;

```

GUI 다시 설정하고, 옵션에서 선택한 트랙, scrollVert, showNoteLetters으로 변경
 옵션에서 키설정을 안 했을 경우엔 트랙에서 메인 키 값을 찾고, 했을 경우엔 설정한
 값을 메인 키 값으로 한다. numtracks는 트랙의 개수, lastStart는 이전에 끝난 시
 간 값에 옵션값을 더한 값.

```

243 +     ArrayList<ArrayList<MusicSymbol>> allsymbols =
244 +         new ArrayList<ArrayList<MusicSymbol>> (numtracks);
245 +
246 +     for (int tracknum = 0; tracknum < numtracks; tracknum++) {
247 +         MidiTrack track = tracks.get(tracknum);
248 +         ClefMeasures clefs = new ClefMeasures(track.getNotes(), time.getMeasure());
249 +         ArrayList<ChordSymbol> chords = CreateChords(track.getNotes(), mainkey, time, clefs);
250 +         allsymbols.add(CreateSymbols(chords, clefs, time, lastStart));
251 +     }
252 +
253 +     ArrayList<ArrayList<LyricSymbol>> lyrics = null;
254 +     if (options.showLyrics) {
255 +         lyrics = GetLyrics(tracks);
256 +     }

```

트랙마다 악보에 표시될 오브젝트들(clefs, chords)를 트랙의 모든 노트의 정보와 맞게 생성하고 추가한다.

```

259 +     SymbolWidths widths = new SymbolWidths(allsymbols, lyrics);
260 +     AlignSymbols(allsymbols, widths, options);
261 +
262 +     staffs = CreateStaffs(allsymbols, mainkey, options, time.getMeasure());
263 +     CreateAllBeamedChords(allsymbols, time);
264 +     if (lyrics != null) {
265 +         AddLyricsToStaffs(staffs, lyrics);
266 +     }

```

```

271 +     for (Staff staff : staffs) {
272 +         staff.CalculateHeight();
273 +     }
274 +     zoom = 1.0f;
275 +
276 +     scrollAnimation = new ScrollAnimation(this, scrollVert);
277 +
278 + }

```

모든 표시할 오브젝트가 담긴 allsymbols와 lyrics를 가지고 크기를 계산한다. 그 다음 크기와 설정값(option)에 맞게 align한다. MusicSymbols, key, lines는 하나의 staff를 이룬다. 그리고 beamed할 건반도 생성한다.

그리고 각 staff마다의 높이를 계산하고, scrollVert값에 해당하는 scroll 애니메이션도 생성한다.

```

1529 +     } else {
1530 +         Log.d("SheetMusic", "adding new note");
1531 +         MidiNote newNote = new MidiNote(pulseTime, 0, 60, 400);
1532 +         this.player.midifile.getTracks().get(0).getNotes().add(newNote);
1533 +         this.sheetMusicRequestListener.onRefreshRequest();

```

주어진 pulsetime으로 새 노트를 하나 만들어 추가한다.

```

246 +     private void
247 +     updateSheetMusic(MidiOptions options) {
248 +         sheet.reinit(options);
249 +         player.updateToolbarButtons();
250 +         layout.requestLayout();
251 +         sheet.draw();
252 +     }

```

위 함수는 악보를 갱신하는 함수로, reinit을 통해 리셋을 하고 각종 버튼과 레이어

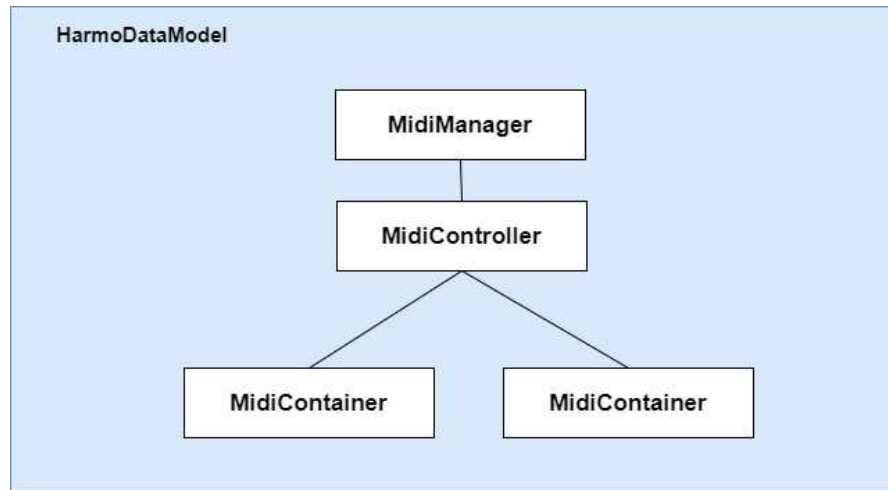
웃을 갱신한 후 악보를 새로 그린다.

4.3.2 HarmoDataModel(2차 보고서: Control API)

(Simple Reference:

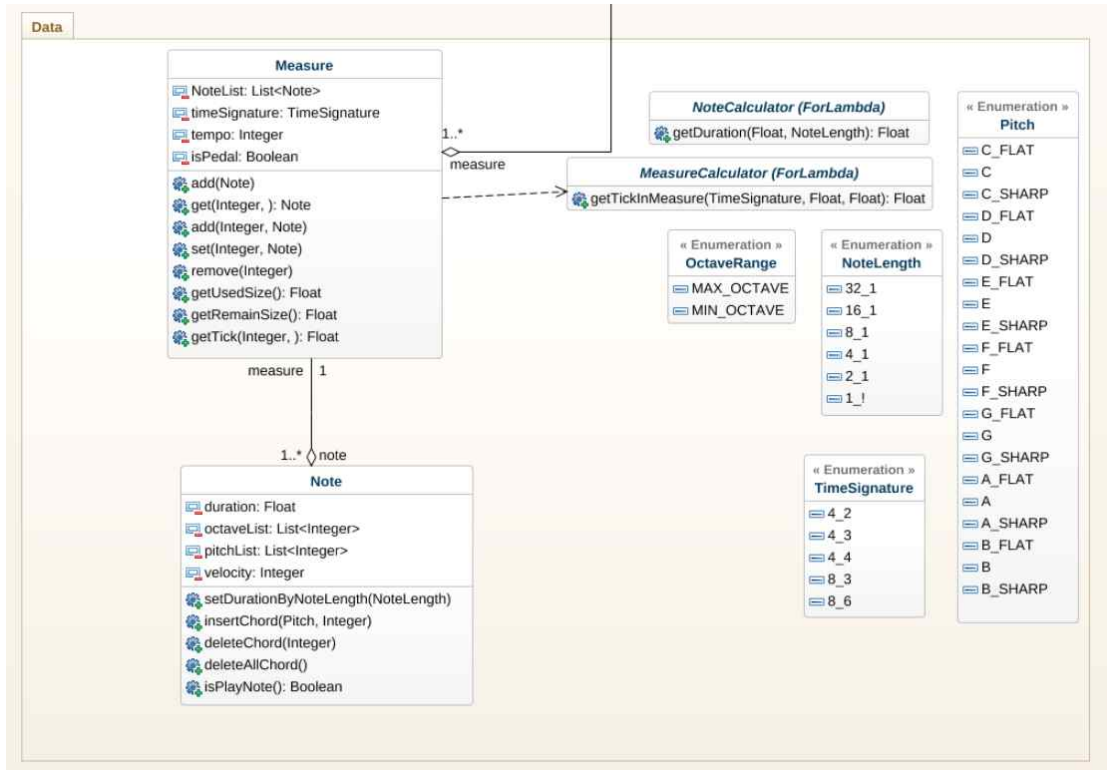
https://github.com/Re-Coma/Android-Midi-Control-Library/blob/master/API_Information.md)

- MidiPlayer는 HarmoVisualSheet에 의해 작동이 되므로 설계에서 제외



- Midi Control Module은 원활한 미디 재생을 위해 외부 라이브러리인 android-midi-lib를 바탕으로 해서 작성이 된다.
- **Midi Manager**: 이 어플리케이션의 모든 미디 데이터들을 총괄 관리하는 모듈로 하위 모듈들을 컨트롤 한다.
- **Midi Controller**: 미디 재생을 제외한 미디 데이터 조작을 하기 위해 설계된 클래스이다.
- **Midi Container**: 악보 데이터가 들어있는 클래스로 이 어플리케이션은 두 개의 트랙(멜로디, 반주)를 사용하기 때문에 Mid Controller에는 두 개의 Midi Container가 들어있다.
- **Cursor Control**: 위의 구성도에서는 시각화 하지 않았지만 사용자가 선택한 음표의 위치를 가리키며 사용자가 음표를 선택할 때마다 Cursor Control는 사용자가 선택한 위치의 음표로 이동하여 악보 인터페이스와 동기화를 한다. 악보를 재생할 때도 마디 단위로 Cursor Control이 움직인다.

● 악보 데이터 및 미디 데이터 구조



- 악보 데이터 구조는 계층적으로 되어 있으며 MidiList -> Measure -> Note 순이다.
- **Midi List**: 마디의 모음을 의미하며 악보로 따지면 오선지 하나(트랙 한 개)가 된다.
- **Measure**: 마디 하나를 의미하며 이 안에 음표(Note)와 박자(Time Signature), 빠르기(Tempo), 페달 사용(isPedal)를 설정할 수 있다. 이 어플리케이션에서는 박자와 빠르기를 악보 하나에 하나만 설정할 수 있음에도 불구하고 Midi Container의 멤버변수가 아닌 Measure의 멤버 변수로 들어간 이유는 두 가지가 있는데, 이는 차후에 어플리케이션을 완성하고 난 후에 확장 기능으로 중간의 박자나 빠르기 변형을 구현하기 위해서다.
- **Note**: 음표 및 침표를 나타내며 변수가 들어가야 하는 데 노트의 길이가 포함된다. 음표 하나에 여러 음이 들어갈 수 있으므로 pitch와 octave를 리스트로 선언하며 반대로 이 두 리스트의 원소가 없으면 침표로 인식한다.

1) 데이터 관리 알고리즘

- 데이터 관리는 주로 Measure에서 이루어 지는데 Measure 안에서 Note가 List로 저장되어 있지만 List에서의 Note들의 Duration의 합이 Measure의 최대 길이와 같아야 한다.
- **Measure의 최대 길이의 계산**: $\text{time_signature.numerator} * \text{time_signature.demoninator}$ (numerator는 박자(4박자, 3박자)를

의미하고 demoniator는 박자의 기준을 의미한다. 4분음표를 1로 설정하고 8분음표를 0.5로 잡는다.)

- **마디 초기화:** 마디의 초기화는 애초에 비어있는 상태가 아닌 Measure의 최대 길이를 가진 쉼표 하나로 시작한다. 아무것도 없으면 Measure 안에 있는 음표의 길이 합과 Measure의 최대 길이가 맞지 않기 때문이다.
- **노트 수정:** 노트 수정의 기준은 해당 위치의 음표나 쉼표에 위치와 길이를 수정하지 않고 그 안의 데이터만 변경을 하는 것으로 정한다. 음표를 쉼표로 바꾸거나, 음표의 pitch 및 octave를 변경 도는 음 추가가 그 예다.
- **노트 삭제:** 노트 삭제는 존재하지 않고 노트 병합 및 분할로 대신한다.
- **노트 분할과 병합:** 노트 분할은 하나의 길이를 가진 노트를 두 개 이상의 노트로 분할을 하는 것을 의미하고 노트 병합은 여러 개의 노트를 하나로 합치는 것을 의미한다. 이때 분할 전과 분할 후의 길이는 일정해야 하며 분할과 병합의 구별 방법은 Midi Cursor가 특정 노트를 선택했을 때 새로 입력해야 하는 노트의 길이가 가리키는 노트의 길이보다 작으면 분할이고 그 반대일 경우 병합으로 판별한다. 음표 삭제는 쉼표 변경을 의미하므로 쉼표로 변경하고 주위에 쉼표가 또 있으면 하나로 합쳐야 하기 때문에 병합에 포함된다. 쉼표 삭제도 주변에 쉼표가 존재할 때 가능한데 이 부분도 병합에 포함된다.

2) 미디 재생 원리(위 모듈에서는 기능이 제외되었으나 다른 모듈에서 사용될 가능성이 있기 때문이 기술)

- 처음 설계 당시 미디를 재생하기 위해 Android-midi-lib에서 지원하는 Midi Event Interface를 사용해 구현하려고 했으나 이를 구현하려면 미디파일을 따로 생성을 해야 할뿐더러 자료형 변경이 자주 생기는 이유로 따로 사용하지 않고 따로 하드코딩으로 진행한다. 물론 billthefarmer-Midi-Driver를 사용해서 구현을 한다.
- 위의 라이브러리를 활용하여 미디를 재생하는 가장 기본적인 코드는 다음과 같다.

```
// kotlin code
1. var midiDriver : MidiDriver = MidiDriver()
2. midiDriver.setOnMidiStartListener(listener)
3. var event : toByteArray( (0x90).or(0x00).toByte(),
4.                          0x3c.toByte(),
5.                          0x7f.toByte())
6. midiDriver.write(event)
```

- line 1에서 midiDriver를 초기화 한다.
- line 2에서 listener를 추가하는 데 이는 미디 재생을 시작할 때 인터페이스를 조작하는 것이므로 생략한다.
- line 3~5에서 미디음을 내기 위한 데이터를 준비하는데 길이가 3인 ByteArray가 들어간다. 첫 번째 데이터는 노트 재생 여부와 채널을

의미하는 데 0x90은 노트 재생, 0x80은 노트 재생 정지를 의미한다. 이 시그널로 음 재생을 정하기 때문에 해당 음 길이를 간격을 정해서 음 재생을 할 수 있다. 0x00는 미디 채널을 의미하는데 이 어플리케이션은 1채널만 사용하므로 0x00 으로 고정한다.

- 두 번 째 데이터는 pitch를 의미한다. 0x3c면 가장 중앙에 있는 C를 의미한다.
- line 6에서 write 함수를 사용하면 기기에서 미디 음이 나오게 된다.
- 이를 활용해서 위에 기술한 페달 효과를 구현할 수 있는데 마디 시작점부터 0x90을 사용하여 마디 끝까지 오기 전 까지 마디 안에 있는 음들을 순차적으로 소리를 내다가. 마지막 부분에 0x80을 사용하여 Midi List에 있는 데이터들을 전부 재생 종료를 하면 된다.

4.3.3 MusicStatisticModule

1) 개요: 2월 초에 제안된 기능으로 지난 1차 발표 당시 장르별 AI 작곡이 필요하다는 지적이 나온 후로, 4마디 작곡 기능을 없애는 대신, 처음부터 끝까지 작곡을 하는 것으로 결정을 하였다. 그러나 작곡만 하면 사실상 이 어플리케이션의 의미가 없으므로 AI로 작곡이 되어진 것에 대해 분석 및 통계를 내어 해당 곡의 속성을 파악하는 기능을 구현할 필요가 생겼다. 물론 이 기능은 단순히 AI작곡 뿐만 아니라 사용자가 직접 작업을 하는 악보에도 해당이 되는 기능이다.

2) 지금까지 설계 및 제안된 기능들

- **사용된 음 통계:** 말 그대로 사용된 음들을 수집해서 차트로 나타낸다.
- **Key Signiture 통계:** Krumhansl과 Schmukler이 제안한 Key Finding Algorithm을 사용해 사용된 화음 코드를 수집한다. 이 때 화음은 알고리즘 특성상 Major와 Minor만 측정이 가능하다. 화음 코드를 수집해 “사용된 음 통계”처럼 차트로 나타내고 마디 마다 어떤 성향의 코드가 있는 지 bar 형태로 나타낸다.

3) 앞으로 설계할 기능들

- **멜로디 조회:** 화음이 아닌 멜로디를 통계로 잡는 다. 이 때 x축을 시간, y축을 pitch로 설정해서 화면에 출력을 한다. 단순히 출력만 하는 것이 아닌 반복되는 부분이나 특정 부분을 파악함으로써 멜로디가 어떻게 진행되는 지에 대해 판단을 한다. 이 기능은 회귀분석과 시계열 분석을 어느정도 한 다음 설계를 할 예정이다.
- **화음 진행상황 조회:** 각 마디에 설정되어 있는 화음들을 추출하여 화음이 어떻게 진행되는 지 판단하고나 이상한 부분을 지적하는 기능이다. 이 기능도 화성학에 대해 연구를 좀더 깊게 한 다음 설계를 할 예정이다.

4.3.4 HarmoControl: 화음 배치를 수행할 때 사용되는 알고리즘으로 화성학 교재에 기술되어 있는 화음 배치를 사용한다. 이 어플리케이션에서 사용할 화음 배치 기능은

다음과 같다.

- **자율 선택:** 자율 선택은 사용자가 수동으로 화음을 직접 선택하는 기능이다. 또한 해당 사용자가 선택된 루트 음에 대한 메이저 다이어 토닉 또는 마이너 다이어토닉을 선택할 수 있다. (다이어토닉이란 해당 루트 음의 메이저 및 마이너 스케일에 대한 3화음 또는 4화음을 의미하며 마이너 다이어토닉은 자연 단음계, 화성 단음계, 가락 단음계로 나뉘어 진다)
- **주요 3화음 생성:** 해당 루트 음에 대한 토닉(I), 서브 도미넌트(IV), 도미넌트(V)를 생성한다. 그리고 이 세가지는 일반과 대리로 나뉘는 데 대리는 일반 토닉/도미넌트/서브도미넌트를 대신하는 화음을 의미하며 대리 코드를 사용하면 표현할 수 있는 음악의 범위가 확장된다. 단, 대리 토닉 또는 대리 도미넌트로 끝을 마무리 할 때 III^m7 코드 같은 경우 토닉의 주요 특징인 루트 음이 빠져 있고, 도미넌트의 주요 특징인 트라이톤(루트음과 증 4도 차이 나는 도수 예를 들어 루트음이 도이면 트라이톤은 파가 된다.) 없기 때문에 사용할 수 없다.
- **노트 커넥션:** 노트 간의 자연스러운 접속을 하기 위한 기술이며 홀드, 슬라이드, 접속이 있다.

1) **홀드:** 2개의 화음끼리 배음에 공통음을 가지고 있는 것을 의미한다.



위의 그림에서 Cm의 5배음과 Em의 루트음이 동일하므로 홀드가 되고 있다. 5배음 위치에서 루트음 위치로 내려갔으므로 차수 하강이 되고 (C major 기준)루트음(C)과 5배음(E)의 도수 차는 장 3도 이므로 장3도 상행이 있다. 이와 같은 방식으로 홀드-차수상행에서는 장3도 상행/완전4도 하행/단3도 하행 이 있고, 차수상승에서는 단3도상행/장3도 하행이 있다. 둘 중에 차수하강은 루트음에 가까워지므로 상대적으로 접속을 매끄럽게 한다.

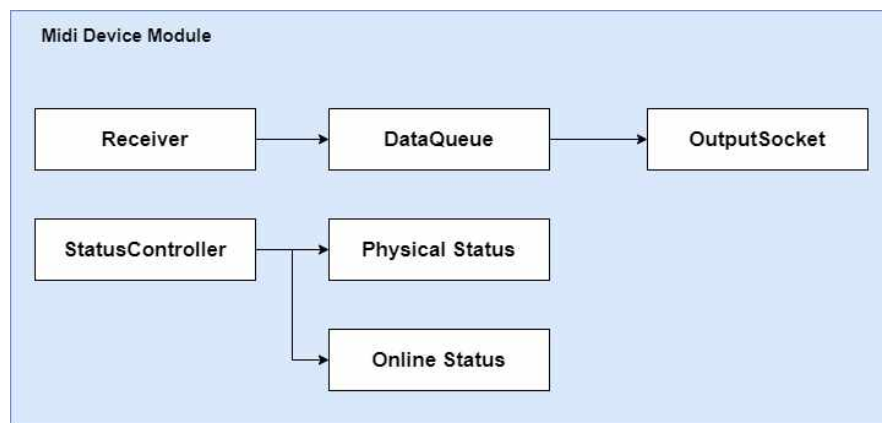
2) **슬라이드:** 슬라이드는 1온음 또는 1반음 씩 상승하는 것을 의미한다. 즉 반음/온음 상행, 반음/온음 하행을 의미한다.

- **도미넌트 베리에이션:** 도미넌트(V)를 변형(Vsus4)하여 토닉과 도미넌트 사이의 연결을 더욱 매끄럽게 하는 기능으로 화음 루트는 V7sus4 - V7 - I과 V7sus4 - I를 사용한다. 이것 말고도 다른 베리에이션 버전이 있으나 이 어플리케이션에서는 표현 범위 밖이므로 가능하다면 테스트 과정에 기능을 더 추가할 예정이다.
- **세컨드리 도미넌트:** 토닉 이외의 다이어토닉 코드를 임시 토닉으로 보고 종지 느낌으로 접속할 때의 사용하는 임시 도미넌트를 의미한다. 세컨드리 도미넌트의 조건으로 루트음은 완전 4도 상행으로 진행해야 하고, 트라이톤이 단 3도로 해결되어야 한다. 세컨드리 도미넌트 앞에 II-V화

를 사용해서 표현을 확장할 수 있다.

- **디미니시**: 디미니시 코드는 도미넌트의 특징인 트라이톤을 갖고 있기 때문에 도미넌트의 대리 코드로 사용이 가능하다. 이 어플리케이션은 **패싱 디미니시**를 사용하는 데, 패싱 디미니시는 두 개의 화음 사이에 앞의 화음의 루트에 반음을 올려서 연결을 하는 기능이다.
- 위의 기능 말고도 **텐션이나, 마이너 코드 사용 등 다른 기능들도** 추가적으로 연구가 계속되고 있다.

4.3.5. MidiDeviceModule: 미디 키보드와 통신을 하기 위한 모듈이다.

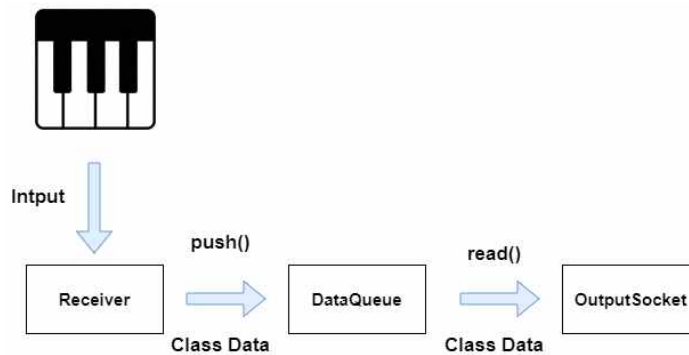


- **Receiver**: 실제 미디 디바이스로부터 미디 데이터를 받는 클래스이다, 이 클래스는 직접 작성한 것이 아닌 안드로이드 Midi Device Library에서 지원하는 기능이다.
- **DataQueue**: Receiver로부터 받은 데이터를 형식에 맞게 가공하여 사용자부터 데이터를 꺼내 쓸 수 있게 저장하는 클래스이다.
- **OutputSocket**: 사용자가 미디 디바이스로부터 데이터를 쉽게 사용하기 위한 클래스
- **Status Controller**: 안드로이드와 미디 키보드 사이의 연결 상태를 확인하는 클래스로, 실제 물리적 연결을 확인하는 Physical Status와 안드로이드에서 사용할 수 있게 포트가 열려져 있는지 확인하는 Online Status를 활용한다.

1) 데이터 구조

- 미디 디바이스로부터 Recerver를 통해 받는 데이터는 3가지가 된다.
- **Signal**: 디바이스의 어느 부분(건반, 밴드(휠) 등...)에서 입력이 들어왔는지, 그리고 건반에서 왔을 경우 건반을 눌렀을 때와 떼었을 때의 Signal도 구별이 된다.
- **Pitch** 말 그래도 Pitch를 의미한다. 건반을 누르면 음 위치에 따라 다르게 출력이 되고 밴드에서는 위로 올리면 127, 중간은 64, 밑이면 0이 출력된다.
- **Velocity**: 음의 세기를 의미하지만 이 어플리케이션에서는 사용하지 않는다.

2) 미디 디바이스 통신 원리



- 디바이스 통신 원리는 중간에 DataQueue를 추가했는데 그 이유는 경우에 따라 데이터를 기다렸다가 한꺼번에 받는 상황도 있기 때문에 TCP/IP API를 모방해서 설계했다.
- Midi Device로부터 입력을 받으면 아까 서술했다시피 위 3개의 데이터를 받는다. Receiver에서 이를 받으면 이 어플리케이션에 맞게 데이터를 가공해서 Class로 저장하고 DataQueue로 저장을 하게 되는 데, 이 DataQueue도 최대 적재할 수 있는 크기가 제한되기 때문에 한계가 넘어가면 가장 맨 앞에 있는 데이터가 삭제된다.
- DataQueue에 데이터가 저장되어 있다가, 사용자가 OutputSocket에서 read() 함수를 사용하게 되면 DataQueue에서 맨 위에 있는 데이터가 밖으로 나오게 된다.

3) 미디 디바이스와 Midi Cursor 간의 프로토콜

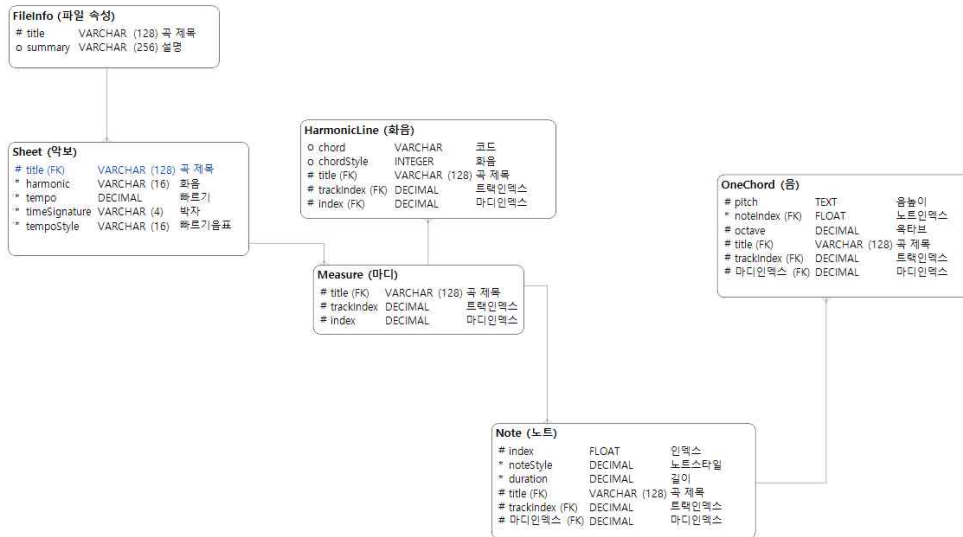
- 미디 디바이스의 사용 목적은 터치 없이 음표 입력의 가능과 커서 이동이다. 미디 디바이스를 활용한 기능은 아래와 같다.
- **음표 및 쉼표 입력, 음표 및 마디 사이 이동**
- 음표 및 쉼표 입력은 Midi Device Module에서 Midi Control Module로 바로 데이터를 보내서 처리하면 되지만, 음표 및 마디 사이 이동은 어느정도의 알고리즘이 필요하다.
- **음표 및 마디 사이 이동:** 음표 이동은 Pitch Band를 사용한다. 위로 올리면 앞으로 이동하고 밑으로 내리면 뒤로 이동한다. Pitch Band는 탄성력이 있기 때문에 올리거나 내렸다 떼면 다시 원상태로 돌아가기 때문에 이동을 멈추는 것이 가능하다. 그리고 단순히 이동이 아닌 휠을 돌리는 세기에 따라 속도도 달라야 하므로 이 어플리케이션은 속도를 3단계 지원하고 이에 충족하는 값은 다음과 같다.

방향/속도	Pitch 값
왼쪽/3	0 ~ 20
왼쪽/2	21 ~ 41
왼쪽/1	42 ~ 63
정지	64
오른쪽/1	65 ~ 75
오른쪽/2	76 ~ 96
오른쪽/3	97 ~ 127

- 마디 사이 이동은 Pitch Band와 건반을 사용한다. 건반 아무거나 하나를 누른 상태로 Pitch Band를 사용하면 마디 단위로 커서가 움직인다. 단, 속도차는 없다.

4.3.6 Save Module: 데이터를 영구적으로 저장하기 위한 모듈이다.

1) 설계



■ **FileInfo:** 악보의 이름과 설명을 저장한다.

■ **Sheet:** 악보의 코드, 빠르기, 박자를 정한다. 빠르기 음표는 4분음표(2/4, 3/4, 4/4) 또는 점 4분음표(3/8, 6/8)를 의미한다.

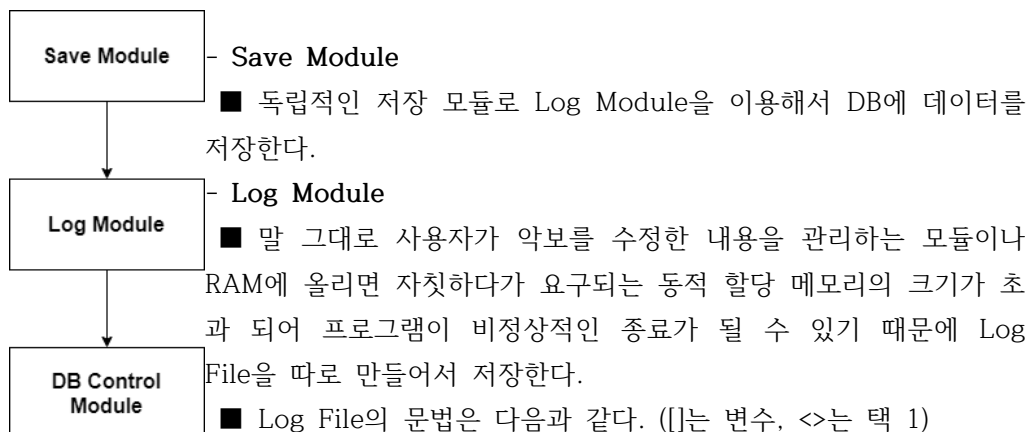
■ **Measure:** Midi Control API에서 설명한 Measure과 노트를 저장하는 방식은 유사하나 트랙 두 개를 관리하게 되는데 하나는 멜로디(Note, OneChord)이고 다른 하나는 멜로디+반주(HarmonicLine)이다.

■ **HarmonicLine:** 반주를 의미하며 화음 종류(chord, chordStyle)와, 반주 스타일(harmonicStyle)이 들어간다.

■ **Note:** Midi Control API의 Note와의 개념과는 동일하나 첼로와 음표의 테이블이 동일하고 그대신 음 테이블인 OneChord의 개수로 판단한다.

그림 : (DB 테이블)

2) Module 구성도 및 Log 문법



마디 추가: Measure [idx] is inserted
마디 삭제: Measure [idx] is deleted
노트 분할: Track [tidx] Measure [midx] Note [nidx] is Seperated To duration [startidx], [lastidx]
노트 합병: Track [tidx] Measure [midx] Note [idx_a] to [idx_b] is Merged
음표 음 추가: Track [tidx] Measure [midx] Note [idx] add Chord [octave] [pitch]
음표 음 삭제: Track [tidx] Measure [idx] Note [idx] Chord [octave] [pitch] is deleted
화음 생성: Track [tidx] Measure [midx] Harmonic is Generated to [chord] [chordStyle]
화음 수정: Track [tidx] Measure [midx] Harmonic is Changed to [chord] [chordStyle]

3) 클래스 다이어그램 및 레퍼런스

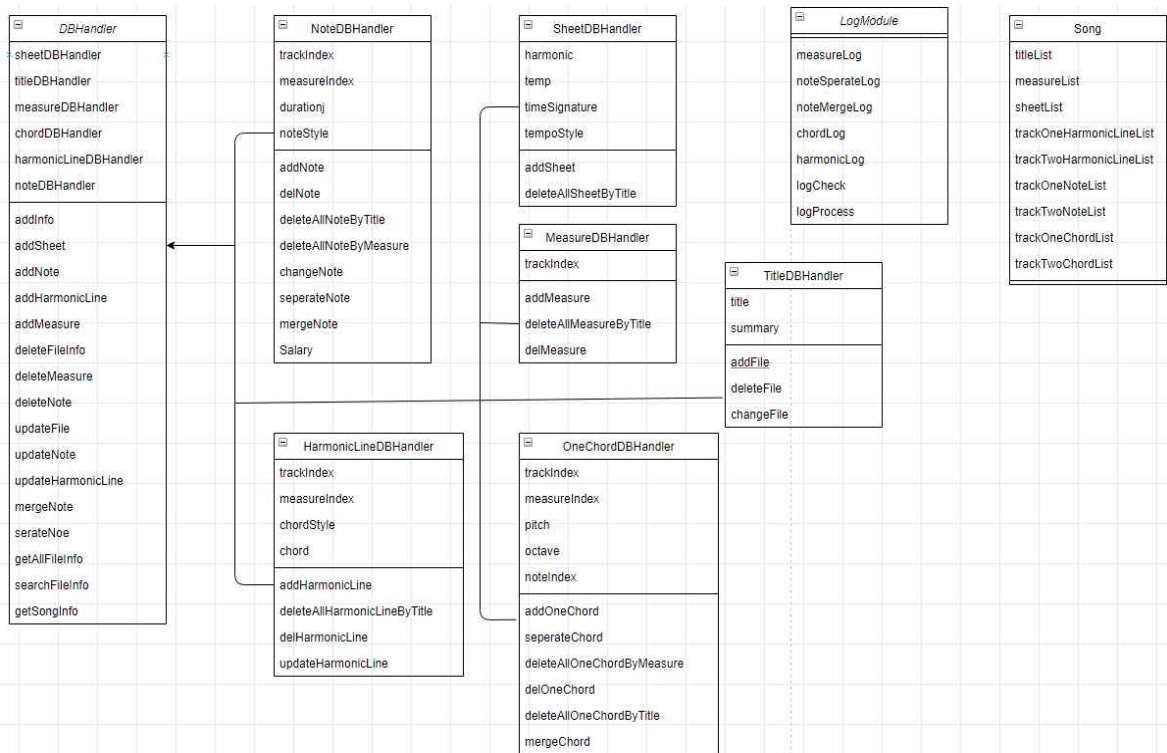
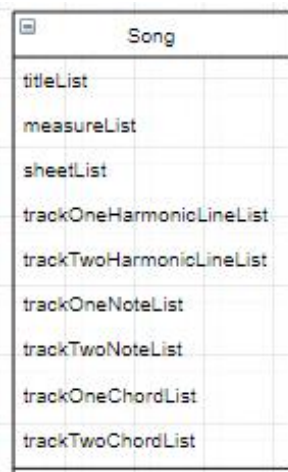


그림 : (클래스)

Song.kt (곡 정보들을 저장하는 클래스)

Song의 멤버변수



Song	
titleList	
measureList	
sheetList	
trackOneHarmonicLineList	
trackTwoHarmonicLineList	
trackOneNoteList	
trackTwoNoteList	
trackOneChordList	
trackTwoChordList	

titleList : 곡 제목과 설명을 저장하는 리스트

measureList : 마디 개수를 저장하는 변수

sheetList : 악보 정보를 저장하는 변수

trackOneharmonicLineList : 트랙1의 하모니 정보를 저장하는 리스트

trackTwoharmonicLineList : 트랙2의 하모니 정보를 저장하는 리스트

trackOneNoteList : 트랙1의 노트 정보를 저장하는 리스트

trackTwoNoteList : 트랙2의 노트 정보를 저장하는 리스트

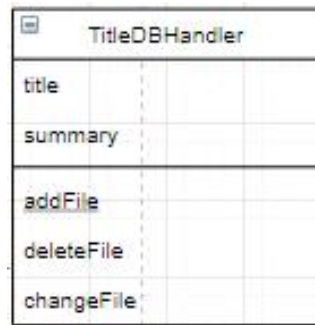
trackOneChordList : 트랙1의 음 정보를 저장하는 리스트

trackTwoChordList : 트랙2의 음정보를 저장하는 리스트

DBhandler.kt (DB 만들때 이름 만드는 기반이름들 선언)

DBHandler	DBHandler의 멤버함수
sheetDBHandler	oncreate() : 테이블의 생성을 명령 하여 DB 파일을 만드는 기능
titleDBHandler	getAllFileInfo() : 파일목록 불러오기 기능 (title과 summary 리스트)
measureDBHandler	searchFileInfo(keyWord) : 매개변수인 Keyword를 목록중 검색하는 기능
chordDBHandler	getSongInfo(titleName) : titlename을 입력받아서 곡에대한 정보를 얻는 기능
harmonicLineDBHandler	addInfo(title, summary) : 파일 정보를 추가하는 함수
noteDBHandler	addSheet(title, harmonic, tempo, timeSignature, tempoStyle) : 악보 정보를 DB에 추가하는 함수를 실행
addInfo	addNote(title, trackIndex, measureIndex, noteIndex, octave, pitch, duration, noteStyle) : 음표 정보를 DB에 추가하는 함수를 실행
addSheet	addHarmonicLine(title, trackIndex, measureIndex, chord, chordStyle) : 화음 정보를 DB에 추가하는 함수를 실행
addNote	addMeasure(title) : 마디 정보를 DB에 추가하는 함수를 실행
addHarmonicLine	deleteFileInfo(title) : 파일 정보를 DB에서 삭제하는 함수를 실행
addMeasure	deleteNote(title, trackIndex, measureIndex, deleteNoteIndex) : 노트와 음 정보를 DB에서 삭제하는 함수를 실행
deleteFileInfo	deleteMeasure(title, trackIndex, measureIndex) : 마디 정보를 DB에서 삭제하는 함수를 실행
deleteMeasure	updateFile(title, summary) : 파일 정보를 DB에서 수정하는 함수를 실행
deleteNote	updateNote(title, trackIndex, measureIndex, noteIndex, duration, noteStyle) : 노트 정보를 DB에서 수정하는 함수를 실행
updateFile	updateHarmonicLine(title, trackIndex, measureIndex, noteIndex, chord, chordStyle) : 화음 정보를 DB에서 수정하는 함수를 실행
updateNote	mergeNote(title, trackIndex, measureIndex, mergeNoteIndex, merge) : 노트를 병합 했을 때 정보를 DB에서 수정하는 함수를 실행
updateHarmonicLine	separateNote(title, trackIndex, measureIndex, separateNoteIndex, separate) : 노트를 분할 했을 때 정보를 DB에서 수정하는 함수를 실행
mergeNote	
separateNote	
getAllFileInfo	
searchFileInfo	
getSongInfo	

TitleDBHandler.kt (파일 속성 테이블을 사용하기 위한 클래스)



```
classDiagram
    class TitleDBHandler {
        title
        summary
        addFile()
        deleteFile()
        changeFile()
    }
```

The diagram shows a class named TitleDBHandler. It has two attributes: title and summary. It has three methods: addFile, deleteFile, and changeFile.

TitleDBHandler의 멤버함수

addFile(db, title, summary) : DB에 파일 정보를 추가하는 함수

1. 매개변수 db의 Title 테이블에서 가지고 가야할 Column은 values 변수를 통하여 TITLE, SUMMARY를 저장합니다.

deletefile(delTitle, db) : DB에서 해당하는 파일 이름을 가진 정보들을 삭제하는 함수

1. 매개변수 db의 title과 매개변수의 delTitle을 비교하여 테이블의 정보를 삭제합니다.

changeFile(db, title, summary) : DB에서 해당하는 파일 이름을 가진 정보들을 수정하는

1. 매개변수 db의 Title 테이블에서 가지고 가야할 Column은 values 변수를 통하여 TITLE, SUMMARY를 수정합니다.

HarmonicLineDBHandler.kt (HarmonicLine 테이블을 사용하기 위한 클래스)



HarmonicLineDBHandler	
trackIndex	
measureIndex	
chordStyle	
chord	
addHarmonicLine	
deleteAllHarmonicLineByTitle	
delHarmonicLine	
updateHarmonicLine	

HarmonicLineDBHandler의 멤버함수

`addHarmonicLine(db, title, trackIndex, measureIndex, chord, chordStyle)`

: DB에 HarmonicLine 정보를 추가하는 함수

1. 매개변수 db의 title, trackIndex, measureIndex, chord, chordStyle을 입력받아 테이블에 저장합니다.

`deleteAllHarmonicLineByTitle(delTitle, measureIndex, db)`

: HarmonicLine 테이블에서 하나의 정보를 삭제하는 함수

1. 매개변수 db의 title과 매개변수의 delTitle을 비교하여 삭제합니다.

`delHarmonicLine(db, title, trackIndex, deleteMeasureIndex)`

: DB에서 HarmonicLine 정보를 삭제하는 함수

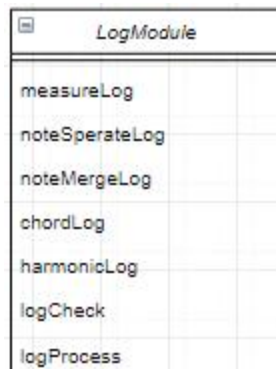
1. 매개변수 db의 title, trackIndex, measureIndex을 입력받아 테이블을 삭제합니다.

`updateHarmonicLine(db, title, trackIndex, measureIndex, chord, chordStyle)`

: DB에 HarmonicLine 정보를 수정하는 함수

1. 매개변수 db의 title, trackIndex, measureIndex, chord, chordStyle을 입력받아 테이블을 수정합니다.

LogModule.kt (event 발생시 로그데이터를 관리하는 테이블을 사용하는 클래스)



LogModule	
measureLog	
noteSperateLog	
noteMergeLog	
chordLog	
harmonicLog	
logCheck	
logProcess	

LogModuleDBHandler의 멤버함수

`measureLog (db, event, title, trackIndex, measureIndex)`

: 마디 추가 삭제 로그를 DB에 저장하는 함수

1. 매개변수 db의 새로 추가할 인덱스를 알기위해서 마지막 로그 인덱스를 불러옵니다.
2. 커서를 통해서 sql문으로 불러온 마지막 인덱스를 저장합니다.
3. values 변수에 저장하여 로그 테이블에 저장합니다.
이때 마지막 인덱스 이후에 추가하기 때문에 +1을 해줍니다.

`noteSeperateLog (db, event, title,
trackIndex,measureIndex,noteIndex,lastIndex,num)`

: 노트 분할 로그를 DB에 저장하는 함수

1. 매개변수 db의 새로 추가할 인덱스를 알기위해서 마지막 로그 인덱스를 불러옵니다.
2. 커서를 통해서 sql문으로 불러온 마지막 인덱스를 저장합니다.
3. values 변수에 저장하여 로그 테이블에 저장합니다.
이때 마지막 인덱스 이후에 추가하기 때문에 +1을 해줍니다.

`noteMergeLog (db, event, title,
trackIndex,measureIndex,noteIndex,lastIndex,num)`

: 노트 합병 로그를 DB에 저장하는 함수

1. 매개변수 db의 새로 추가할 인덱스를 알기위해서 마지막 로그 인덱스를 불러옵니다.
2. 커서를 통해서 sql문으로 불러온 마지막 인덱스를 저장합니다.
3. values 변수에 저장하여 로그 테이블에 저장합니다.
이때 마지막 인덱스 이후에 추가하기 때문에 +1을 해줍니다.

`chordLog (db, event, title, trackIndex, measureIndex, noteIndex, octave,
pitch, duration, noteStyle)`

: 음표 추가 삭제 로그를 DB에 저장하는 함수

1. 매개변수 db의 새로 추가할 인덱스를 알기위해서 마지막 로그 인덱스를 불러옵니다.
2. 커서를 통해서 sql문으로 불러온 마지막 인덱스를 저장합니다.
3. values 변수에 저장하여 로그 테이블에 저장합니다.
이때 마지막 인덱스 이후에 추가하기 때문에 +1을 해줍니다.

`harmonicLog (db, event, title, trackIndex, measureIndex, chord,
chordStyle)`

: 화음 생성 수정 로그를 DB에 저장하는 함수

1. 매개변수 db의 새로 추가할 인덱스를 알기위해서 마지막 로그 인덱스를 불러옵니다.
2. 커서를 통해서 sql문으로 불러온 마지막 인덱스를 저장합니다.
3. values 변수에 저장하여 로그 테이블에 저장합니다.
이때 마지막 인덱스 이후에 추가하기 때문에 +1을 해줍니다.

`logCheck (db, handler)`

: DB에 저장되어 있는 로그를 불러들어오는 함수

1. 로그테이블로부터 로그를 하나씩 불러오는 기능을 수행합니다.
2. Column 들은 “%”로 구분합니다.
3. 이후 커서에 데이터가 없을때까지 DB에 저장된 로그를 불러들이며
logProcess에 정보를 전달합니다.

logProcess (db, event, detail, handler)

- : 로그에 등록된 이벤트에 따라서 해당하는 DB에 대한 동작을 수행하는 함수
1. IF 문을 통해서 event에 해당하는 기능을 함수 호출을 통해서 수행합니다.

MeasureDBHandler.kt(DB Measure 테이블을 사용하기 위한 클래스)

MeasureDBHandler		
trackIndex		
addMeasure		
deleteAllMeasureByTitle		
delMeasure		

MeasureDBHandler의 멤버함수

addMeasure (db, title, trackIndex)

: DB에 measure 정보를 추가하는 함수

1. 가장 큰 마디의 인덱스를 찾고 새로 추가할때는 그 다음의 인덱스를 줍니다.
2. cursor 변수를 사용해 sql 문으로 불러온 마지막 인덱스를 index 변수에 저장합니다.
3. Measure 테이블에서 가지고 가야할 Column은 values 변수를 통하여
TRACK_INDEX, MEASURE_INDEX, TITLE을 저장합니다.

deleteAllMeasureByTitle (delTitle, measureIndex, trackIndex,db)

: Measure 테이블에서 해당하는 파일 이름을 가진 정보를 삭제하는 함수

1. Title 테이블에서의 정보를 삭제할 때 Measure 테이블에서도
해당 title에 대한 정보를 삭제 하여 줍니다.

delMeasure (db, title, trackIndex, deleteMeasureIndex)

: Measure 테이블에서 하나의 정보를 삭제하는 함수

1. 매개변수 db의 가장 큰 마디 인덱스를 불러오는 sql 문을 사용합니다.
2. sql 문으로 불러온 가장 큰 마디의 인덱스를 변수에 저장합니다.
3. 삭제를 원하는 마디의 정보를 삭제합니다.
4. 삭제된 인덱스가 비어있지 않도록 마디 인덱스들을 수정합니다.

NoteDBHandler.kt(Note 테이블을 사용하기 위한 클래스)

NoteDBHandler의 멤버함수

addNote (db, title, trackIndex, measureIndex, duration, noteStyle)

: DB에 Note 정보를 추가하는 함수

1. 매개변수 db의 새로 추가할 인덱스를 알기위해서
매개변수 값(title,Index)들을 이용하여 마지막 노트 인덱스를 불러옵니다.

NoteDBHandler		
trackIndex		
measureIndex		
durationj		
noteStyle		
addNote		
delNote		
deleteAllNoteByTitle		
deleteAllNoteByMeasure		
changeNote		
seperateNote		
mergeNote		
Salary		

2. 커서를 통해서 sql문으로 불러온 마지막 인덱스를 저장합니다.

3. values 변수에 저장하여 노트 테이블에 저장합니다.

delNote (db, title, trackIndex, measureIndex, deleteNoteIndex)

: DB에 Note 정보를 삭제하는 함수

1. 매개변수 db의 가장 큰 노트 인덱스를 불러오는 sql 문을 사용합니다.
2. sql 문으로 불러온 가장 큰 노트의 인덱스를 변수에 저장합니다.
3. 삭제를 원하는 노트의 정보를 삭제합니다.
4. 삭제된 인덱스가 비어있지 않도록 노트 인덱스들을 수정합니다.

deleteAllNoteByTitle(delTitle, measureIndex, trackIndex, noteIndex, db)

: DB에서 파일을 삭제했을 때 해당 파일에 관련된 Note 정보를 삭제하는 함수

1. 매개변수 db의 title과 매개변수의 delTitle을 비교하여 테이블의 정보를 삭제합니다.

changeNote(db, title, trackIndex, measureIndex, noteIndex, duration, noteStyle)

: DB에 Note 정보를 변경하는 함수

1. 매개변수 db의 노트 테이블을 입력받은 매개변수의 값들로 수정합니다.

seperateNote (db, title, trackIndex, measureIndex, mergeNoteIndex, merge)

: 노트를 분할하는 함수

1. 분할하는 개수만큼의 인덱스를 비워주고 공간만큼의 최대 인덱스는 늘려줍니다.
2. 해당 노트의 Duration을 분할하는 개수만큼 나눠줍니다.
3. 원래있던 노트는 지우고 분할하는 개수만큼 데이터를 똑같이 집어넣습니다.
4. 세팅된 값들을 Note 테이블에 저장합니다.

mergeNote (db, title, trackIndex, measureIndex, subNoteIndex, mergeNum)

: 노트를 병합하는 함수

1. 해당 노트들을 병합해줍니다.
2. 병합 후 빈 인덱스만큼 인덱스를 앞으로 땡겨줍니다.
3. 세팅된 값들을 Note 테이블에 저장합니다.

■ separate 와의 차이점은 병합을 먼저해준다는 점입니다.

OneChordDBHandler.kt(OneChord 테이블을 사용하기 위한 클래스)

OneChordDBHandler				
trackIndex				
measureIndex				
pitch				
octave				
noteIndex				
addOneChord				
seperateChord				
deleteAllOneChordByMeasure				
delOneChord				
deleteAllOneChordByTitle				
mergeChord				

OneChordDBHandler의 멤버함수

addOneChord (db, title, trackIndex, measureIndex, pitch, noteIndex, octave)

: DB에 Onechord 정보를 추가하는 함수

1. values 변수에 저장하여 코드 테이블에 저장합니다.

deleteAllOneChordByTitle (delTitle, db)

: OneChord 테이블에서 해당하는 파일 이름을 가진 정보를 삭제하는 함수

1. 매개변수 db의 title과 매개변수의 delTitle을 비교하여 테이블의 정보를 삭제합니다.

deleteAllOneChordByMeasure (db, delTitle, measureIndex)

: OneChord 테이블에서 해당하는 마디의 정보를 삭제하는 함수

1. 매개변수 db에서 해당하는 title과 MeasureIndex에 해당하는 테이블의 정보를 삭제합니다.

delOneChord (db, title, trackIndex, measureIndex, deleteNoteIndex)

: OneChord 테이블에서 하나의 정보를 삭제하는 함수

1. 매개변수 db의 가장 큰 코드 인덱스를 불러오는 sql 문을 사용합니다.
2. sql 문으로 불러온 가장 큰 코드의 인덱스를 변수에 저장합니다.
3. 삭제를 원하는 코드의 정보를 삭제합니다.
4. 삭제된 인덱스가 비어있지 않도록 코드 인덱스들을 수정합니다.

sperateChord (db, title, trackIndex, measureIndex, mergeNoteIndex, merge)

: Note에서 분할을 했을 때 OneChord 테이블에서도 해당하는 음표의 동기화를 위한 함수

1. 분할하는 개수만큼의 인덱스를 비워주고 공간만큼의 최대 인덱스는 올려줍니다.
2. 해당 Onechord 정보와 동일한 정보를 가진 데이터를 OneChord테이블에 분할 개수만큼 저장합니다

mergeChord (db, title, trackIndex, measureIndex, mergeNoteIndex, merge)

: Note에서 병합을 했을 때 OneChord 테이블에서도 해당하는 음표의 동기화를 위한 함수

1. 해당 OneChord 정보들을 병합해줍니다.
2. 병합 후 빈 인덱스만큼 인덱스를 앞으로 땡겨줍니다.
3. 세팅된 값들을 OneChord 테이블에 저장합니다.

SheetDBHandler.kt(Sheet 테이블을 사용하기 위한 클래스)

SheetDBHandler			
harmonic			
temp			
timeSignature			
tempoStyle			
addSheet			
deleteAllSheetByTitle			

SheetDBHandler의 멤버함수

addSheet (db, title, harmonic, tempo, timeSignature, tempoStyle)

: DB에 Sheet 정보를 추가하는 함수

1. values 변수에 저장하여 코드 테이블에 저장합니다.

deleteAllSheetByTitle (delTitle, db)

: Sheet 테이블에서 해당하는 파일 이름을 가진 정보를 삭제하는 함수

1. 매개변수 db의 title과 매개변수의 delTitle을 비교하여 테이블의 정보를 삭제합니다.

정리

해당 DB 구조의 흐름은

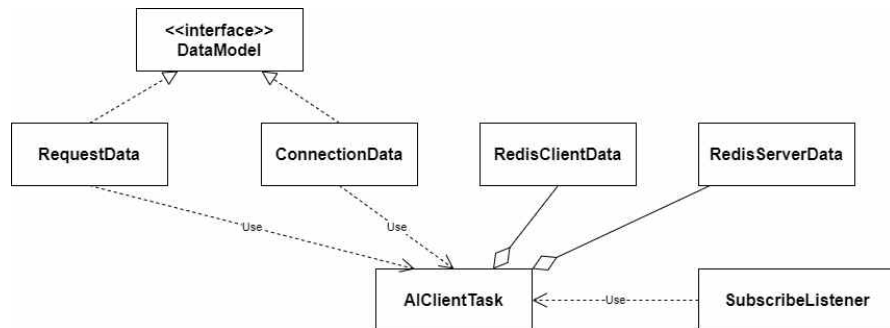
프론트에서 따라서 실행한 내용 바탕으로

로그정보를 저장하는 흐름이며

저장된 로그를 바탕으로 순서대로 실행(변경)하는 것 DB 데이터를 변경합니다.

4.3.7 AIClientTask:

- 서버로부터 딥러닝 요청을 하고 이에 대한 결과 데이터(미디어파일)을 받는 데이터다. 클래스 구성은 다음과 같다.



- **DataModel:** RequestData와 ConnectionData가 인터페이스로 상속을 하는 객체로 Json형식으로 서버를 보낼 때 이 인터페이스를 상속받는다. 그렇기 때문에 가상함수로 makeToJson()이 있다. 클래스에 존재하는 데이터들을 Json으로 묶은 다음에 문자열로 처리하는 함수다.
- **ConnectionData:** 서버에게 연결/연결 해제를 요청할 데이터들을 묶은 클래스, 매개변수로 자신의IP, 시리얼 넘버, 연결/비연결 여부가 있다.
- **RequestData:** 서버에게 AI 요청을 하기 위한 데이터들을 묶은 클래스, 매개변수로 자신의IP, 장르, 박자, 요청 노트크기가 있다.
- **RedisClientData:** 서버와 통신하기 위한 클라이언트 위치에 있는 데이터를 저장하는 클래스로 자신의 ip가 있다.
- **RedisServerData:** Redis Server에 대한 데이터를 저장하는 클래스로, 서버 호스트, 서버 패스워드, 시리얼 넘버가 있다.
- **SubscribeListener:** Redis Server로부터 데이터를 받는다.

4.4. 세부 구성도(서버)

4.4.0. Config

- 서버 프로세스는 실행하기 전에 Config.json으로부터 데이터를 입력 받아 이를 프로세스에 적용한다 Config의 내용은 다음과 같다.

```
{
  "host": "sweetcase.tk",
  "port": 7890,
  "pswd": "4680",
  "max_queue_size": 100,
  "max_note_size": 300,
  "use_gpu_value": 3,
  "serial": "avbk2#$@skd#%",
  "tmp_dir": "tmp",

  "models" : {
    "CHOPIN" : "models/basic_rnn.mag",
    "BACH" : "models/basic_rnn.mag",
    "BEETHOVEN" : "models/basic_rnn.mag",
    "SCARLATTI" : "models/basic_rnn.mag",
    "BALAD" : "models/basic_rnn.mag",
    "NEW_AGE" : "models/basic_rnn.mag"
  }
}
```

- ▶ host: Redis Server의 주소다
- ▶ port: Redis Server로 연결하기 위한 포트 번호
- ▶ pswd: Redis Server의 Password
- ▶ max_queue_size: 이 프로세스에서의 모든 큐가 데이터를 채울 수 있는 최대 개수
- ▶ max_note_size: 클라이언트가 요청할 수 있는 최대 노트 개수
- ▶ serial: 어플리케이션 클라이언트로부터 요청이 들어왔다는 것을 확인하기 위한 암호
- ▶ tmp_dir: AI 수행이 끝나고

임시적으로 파일을 보관하기 위한 디렉터리

- ▶ models: AI Model

4.4.1. ConnectionModule

- Redis Server와의 연결을 담당하는 모듈

ServerConnection
- redisConnector: redis.NewClient - requestQueue : *RequestQueue - connectionQueue : *ConnectionQueue - userMap : *map[string]int
+ GENERATOR (config : Config): void + open(queue : *RequestQueue) : void + close() : void + read() : string + send(msg : string, key : int) : void

1) 매개변수

- redisConnector: Redis 클라이언트 객체로 Redis Server와 연결을 하는데 사용
- requestQueue: 클라이언트로부터 AI 요청 데이터를 받고 임시로 저장하는 데 사용한다.
- connectionQueue: 클라이언트로부터 연결관련 요청 데이터를 받고 임시로 저장하는 데 사용한다.

2) 함수

- open: Redis Server와의 접속을 시도하고 초기 세팅을 한다.

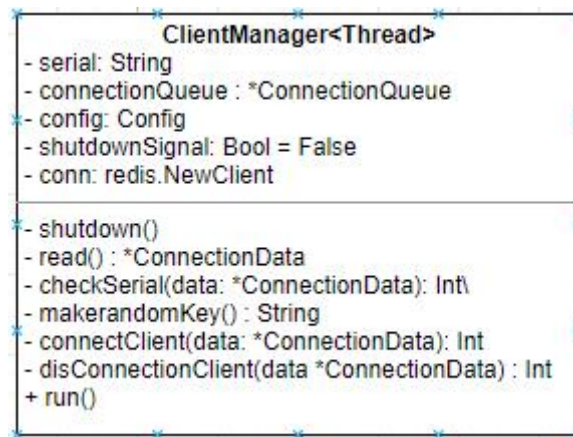
- close(): Redis Server와의 연결을 해제한다.
- read(): Redis Server로부터 데이터를 읽어들인다.
- send(): 해당 클라이언트에게 데이터를 보낸다.

4.4.2. DataTranslator

- ▶ DataTranslator는 Redis Server로부터 데이터를 읽어들이는 때 이 데이터가 클라이언트로부터 오는 데이터가 맞는 지 검사하는 클래스로 이를 검토하는 translateData()함수 하나만 존재한다. Json데이터를 불러오기 때문에 Json형식에 맞는지 1차적으로 검토 한 다음, 올바른 type값이 저장되어 있으면 type값에 따라 ConnectionData나 RequestData를 만들어서 Queue에 저장한다.

4.4.3. ClientManagement

- ▶ 서버에 접속 또는 접속을 시도하려는 클라이언트를 관리하는 객체로 스레드 객체로부터 상속을 받았기 때문에 이 객체 자체가 스레드로 작동이 된다. 또한 Client 정보(Client Host, Client Key)는 Client측에서도 데이터를 읽어야 하기 때문에 RAM에 저장하는 것이 아닌 Redis Server에 저장된다.



1) 매개변수

- serial: DataTranslator를 통해 들어온 데이터가 클라이언트로부터 직접 들어온 데이터인 지 검사하기 위한 일련의 코드이다. Config로부터 값을 불러들여 저장한다
- connectionQueue: 연결 데이터를 저장하는 큐로 초기화를 할 때 외부의 큐의 주소값을 저장해서 활용한다.
- config: 설정 객체
- shutdownSignal: (파이썬 기준)Thread클래스로부터 상속을 받는다. 따라서 스레드 수행을 원하는 시기에 멈추게 하기 위해 이 변수를 사용하고 shutdown()를 이용해 값을 변경한다.
- conn: Redis Client 객체

2) 함수

- shutdown(): 스레드를 종료하는 함수
- read(): ConnectionQueue로부터 데이터를 읽어들인다 없으면 None을 호출한다
- checkSerial(): read를 통해 들어온 데이터가 올바른 데이터인지 판단한다.

- `makerandomKey()`: User의 고유 키를 생성한다. 이 고유키는 차후에 클라이언트에게 데이터를 송신하기 위해 사용된다.
- `connectClient()`: `ConnectionQueue`로부터 받은 데이터를 이용해 클라이언트와의 연결을 설정한다. 이때 연결된 클라이언트에 대한 데이터는 RAM에서 저장하는 것이 아니라 Redis Server에 `IPMap`이라는 hash형 구조체에 `<host, userKey>` 저장한다.
- `disConnection()`: 클라이언트와의 연결을 해제하는 데 사용한다.

4.4.4. DeepLearningManagement

- ▶ 딥러닝 프로세스를 “간접적”으로 관리하는 객체로 딥러닝 수행, 상태 등을 관리한다. 이도 역시 Thread로부터 상속을 받았기 때문에 클래스 자체가 쓰레드로 작동된다.

DeepLearningManagement
<pre> - requestQueue : *RequestQueue - config: Config - conn: Redis.NewClient - shutdownSignal: Bool - DLProcessList: List<DeepLearningChecker> - listMutex: mutex </pre>
<pre> + GENERATOR(requestQueue: *RequestDataQueue, config: *Config, conn: Redis.NewClient) + shutdown() + read() + generate(data: *RequestData) + manageProcessList() + run() </pre>

1) 매개변수

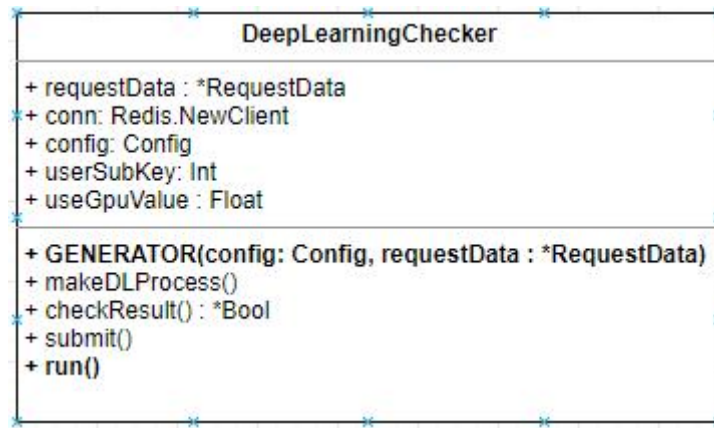
- `requestQueue`: AI요청 데이터를 저장하는 큐이다.
- `config`: 설정 객체
- `conn`: Redis Client 객체
- `shutdownSignal`: 쓰레드 종료 시그널
- `DLProcessList`: 진행중인 AI 프로세스를 저장하는 리스트로 정확히는 Thread List다 왜냐하면 DL 프로세스를 관리하는 쓰레드(DL Checker)를 따로 실행하기 때문이다.
- `listMutex`: `DLProcessList`를 수정할 때 이 `mutex`를 사용한다.

2) 함수

- `shutdown()`: 쓰레드 종료
- `read()`: `requestQueue`로부터 데이터를 읽어들인다.
- `generate()`: `requestData`를 이용해 AI 프로세스 및 Checker 쓰레드를 실행한다.
- `manageProcessList()`: 프로세스 상태를 확인한다.

4.4.5. DeepLearningChecker

- ▶ 딥러닝 프로세스를 “직접적”으로 관리한다. 프로세스를 직접 실행하며 결과물인 MidiFile을 저장까지 한다. Thread로 상속되어있으나. 이는 프로세스가 종료될 때 가지 기다린 다음 데이터를 정리하고 종료하는 일시적 쓰레드이기 때문에 반복문을 사용하지 않고 shutdownSignal도 없다.



1) 매개변수

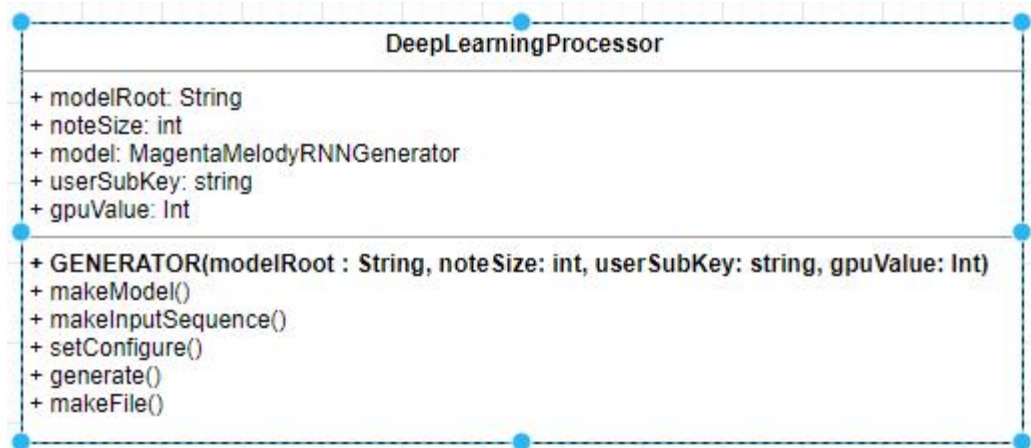
- `requestData`: 클라이언트로부터 받은 요청 데이터
- `conn`: Redis Server 연결부
- `config`: 설정 개개체
- `userSubKey`: User 고유 키(Subscribe Key)
- `useGpuValue`: Tensorflow GPU를 사용할 때 Session에 아무것도 설정하지 않은 경우 하나의 프로세스가 그래픽 카드의 전체 램 용량을 사용한다. 그렇게 되면 다른 프로세스가 그래픽 카드를 활용을 할 수없기 때문에 Value값에 따라서 RAM 용량을 조절한다. 예를 들어 GpuValue가 4이면 총 4개의 프로세스가 그래픽 카드를 사용해야 하므로 1개의 프로세스가 사용하는 RAM 비율은 0.25가 된다.

2) 함수

- `makeDLProcess()`: DL Process 객체를 생성한다. 실행하는 것이 아니다.
- `checkResult()`: 결과물이 만들어졌는 지 확인하는 함수로 미디파일이 만들어졌는지 확인하고 이에 따라 bool 타입의 값을 추출한다.
- `submit()`: `conn`를 이용하여 클라이언트에게 midi파일을 전송한다.

4.4.6. DeepLearningProcessor

- ▶ 딥러닝을 수행하는 단독 프로세스다. Magenta RNN으로 대신하고 있지만, 기존 오픈소스이 Overfitting 문제가 해결되면 오픈소스로 전환할 예정이다.



1) 매개변수

- `modelRoot`: 해당 AI 모델이 저장된 주소
- `noteSize`: 클라이언트로부터 요구된 노트 개수
- `model`: 작곡을 수행할 AI 모델
- `userSubKey`: 클라이언트의 고유 키
- `gpuValue`: GPU 값

2) 함수

- `makeModel()`: 모델을 생성하여 `model` 변수에 저장한다.
- `makeInputSequence()`: RNN 모델이기 때문에 시작 미디 데이터를 지정해야 하는 데 이 함수에서 시작 데이터를 생성한다.
- `setConfigure()`: `model`과 `sequence`를 이용해 AI를 설정한다.
- `generate()`: AI작곡을 수행하는 함수
- `makeFile()`: `generate()`에 의해 작곡된 데이터를 midi파일로 저장한다.

4.5. Server-Client 통신 설계서

4.5.0. 개요

- 이전에는 AI 수행을 어플리케이션 내에서 하려고 설계했으나, Tensorflow Lite 활용에 대한 레퍼런스의 부족과 여러개를 저장해야 하는 모델의 용량 문제로 인해 3월에 서버에서 수행하는 것으로 결정이 되었고 프로토타입 까지 구현이 완료된 상태이며 3차 발표에도 이를 이용하여 프로토타입 시연을 할 예정이다.
- 이 프로젝트에서는 Web을 사용할 일이 없기 때문에 WebServer를 사용하지 않고 Linux에서 동작하는 Daemon Process로 작동이 된다.
- 설계 당시, 프로세스 속도와 난이도를 고려해 C++, Python, Go 중에 Go언어를 선택했었으나, 해당 언어의 숙련도 부족과 Module Import 문제로 인해 현재를 Python으로 대체했으며 Go언어로 포팅될 지는 아직 미지수다.
- Socket 하드코딩을 사용하지 않는다. 그 이유는 보안이나 데이터 전송과

관련하여 많은 문제가 생길 것으로 예측을 하였기 때문이다, 따라서 중간에 Redis Server를 구축함으로써 Redis DB를 통하여 데이터를 송수신한다.

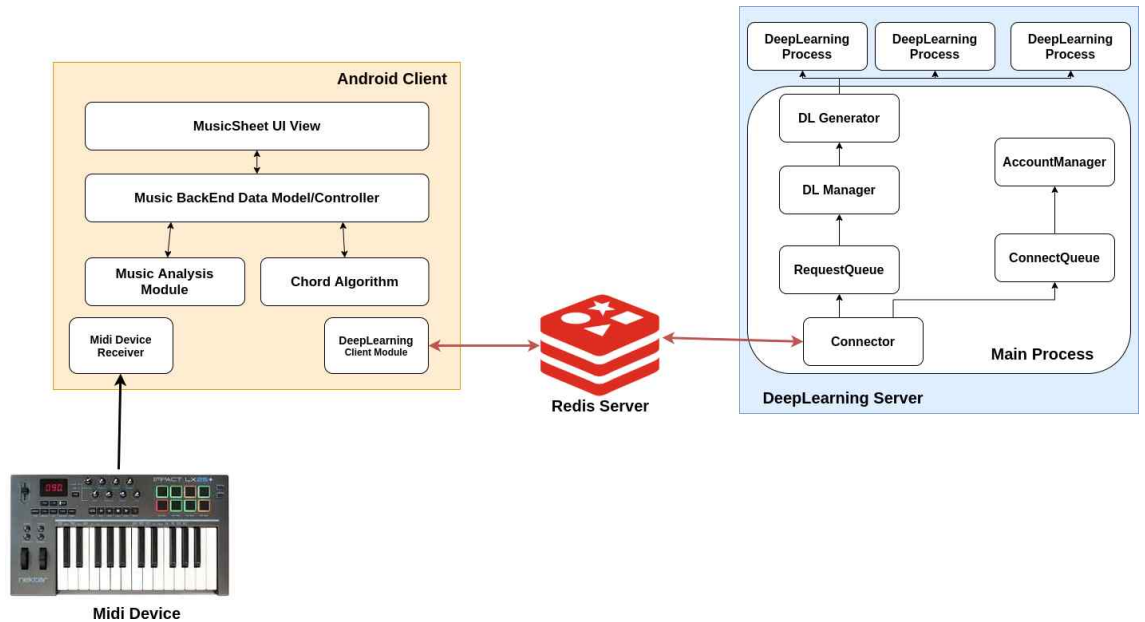
- 하지만 지금 구현된 서버도 아직은 프로토타입이기 때문에 외부의 위험으로부터 자유롭지는 않다. 이에 대한 보완은 3차 발표 이후에 진행 할 예정이다.

4.5.1. Redis



- Redis는 Remote Dictionary Server의 약자로서 Key-Value 구조의 비정형 데이터를 저장하고 관리한다.
- Key는 문자열로 한정되어있지만, Value는 문자열 뿐만 아니라 List/Set/Sorted Set/Hash도 포함이 된다. 이 프로젝트에서는 클라이언트 IP를 이용해 고유 키를 찾아야 하기 때문에 Hash를 사용한다.
- 비휘발성 데이터베이스이므로 일반 RDBMS나 MongoDB처럼 파일에 데이터를 저장하는 것이 아니라 RAM에 저장한다. 따라서 속도는 다른 데이터 보다 훨씬 빠르지만, Server가 종료되면 데이터도 같이 날아간다. 그러나 항상 날라가는 것은 아니고 중간에 디스크에 저장을 하는 복구기능이 따로 있다.
- Redis의 대표적인 기능은 위의 Key-Value관리도 있지만 Publish/Subscribe(줄여서 Pub/Sub)기능이 있다. 클라이언트 간의 통신에 주로 사용이 되는데 예를 들어 클라이언트1에서 “01”이라는 채널을 Subscribe하라는 명령을 내렸을 때 다른 클라이언트에서 “01”채널로 “aaa”란 데이터를 publish하라는 명령을 내리면 01이라는 채널을 구독하고 있는 클라이언트 측에 “aaa”라는 데이터가 날라온다. 즉 Socket역할을 한다고 보면 된다. 이 프로젝트에서도 클라이언트가 요청 데이터를 보내는 것과 서버에서 결과물을 클라이언트에게 보낼 때 이 기능을 주로 사용한다.

4.5.2. 전체 구성도



4.5.3. 프로토콜 및 DB 데이터

- Type: 클라이언트가 서버로 보낼 때 데이터 타입은 주로 3가지 인데 다음과 같다.
 - ⊙ CONNECT: 서버로 연결할 때 사용하는 type으로 정수값은 0이다.
 - ⊙ DISCONNECT: 연결해제 요청을 할 때 사용하는 type으로 정수값은 -1이다.
 - ⊙ REQUEST: ai요청을 할 때 사용하는 type으로 정수값은 1이다.
- ConnectData
 - ⊙ IP: 클라이언트의 ip.
 - ⊙ type: 연결관련 요청이므로 0 아니면 -1이 된다.
 - ⊙ serial: 암호와 비슷한 것으로 서버가 이를 받을 때 serial을 검사해서 일치하면 해당 데이터를 저장한다.
- RequestData
 - ⊙ IP: 클라이언트의 ip.
 - ⊙ genre: 장르
 - ⊙ timeSignature: 박자(차후에 삭제 가능성 있음)
 - ⊙ noteSize: 요청 노트 크기(차후에 변경 가능성 있음)
- IPMap: 프로토콜이 아닌 연결된 클라이언트를 파악하기 위해 DB내에 저장된 hash형 구조 Key값은 ip가 되고 value값은 랜덤으로 형성된 클라이언트 고유 키가 된다.

4.5.3. 작동 원리(알고리즘)

- 서버 가동


```

#values
CONFIG_ROOT="config.json"

# get string of timestamp
get_timestamp() {
    local stamp=$(date +%Y/%m/%d/%H:%M:%S:%6N)
    echo $stamp
}

# ConfigFile Check
check_config() {
    if [ -f $CONFIG_ROOT ]; then
        local timestamp=`get_timestamp`
        echo "[${timestamp}] check config file: $CONFIG_ROOT"
    else
        echo "config file: $CONFIG_ROOT is not exist"
        exit 1
    fi
}

# test redis server

# main process
check_config

# Finally Run python code
start_timestamp=`get_timestamp`
echo "[${start_timestamp}] starting process..."
python harmoisserver.py $CONFIG_ROOT &

```

- ◎ sh run.sh를 입력하면 쉘 스크립트가 실행이 되는 데 처음 서버를 실행할 때 필수적으로 사용하는 config.json이 존재하는 지 검토하고 있으면 실행한다. 프로토타입이기 때문에 추가적으로 json파일의 상태를 검토하는 함수 도 추가할 예정이다. Daemon Process로 돌아가기 때문에 python을 실행할 때 background로 실행한다.

```

# get config info
configRoot = sys.argv[1]
config = Config(configRoot)
conn = ServerConnection(config)
conn.open()

```

- ◎ 프로세스를 실행하면 Config.json으로부터 설정 데이터를 수집하여 Config객체를 생성한 다음 Redis Server에 연결을 시도한다. 실패할 경우 Exception를 호출하고 프로그램은 종료된다.


```
# 기존의 IMap이 존재하는지 확인(서버가 갑작스럽게 종료되는 경우)
if self.conn.exists(ServerConnectionKeyLabels.IPMAP.value) == True:
    # TODO 데이터 갖고오기
    print("already exist")
else:
    # 만들기
    self.conn.hset(ServerConnectionKeyLabels.IPMAP.value, AdminIPMap.IP.value, AdminIPMap.VALUE.value)
    # subscribe
    self.rawReader = self.conn.psubsub()
    self.rawReader.subscribe(ServerConnectionKeyLabels.MAIN_PIPE.value)
```

- ⊙ ServerConnection을 생성하고 open()함수를 이용해 연결을 시도하는 과정은 단순히 연결만 하는 것이 아니라 클라이언트의 고유키를 생성/삭제를 하기 위해 hash테이블을 생성한다. 단 Redis에서의 hash는 데이터가 비어있는 순간 hash테이블은 사라지기 때문에 admin용 데이터를 예시로 생성한다. (hset 명령어로 테이블 생성) 테이블을 생성했다면 subscribe를 사용하여 main pipe 채널을 이용해 클라이언트로부터 데이터를 받을 준비를 한다.

```
# run Threads
try:
    clientManagementThread.start()
    dlThread.start()
except Exception as e:
    # 에러 처리
    print(e)
    sys.exit()
```

- ⊙ 서버와 연결이 완료되었으면 클라이언트를 관리하는 Thread와 AI를 관리하는 Thread를 실행한다.

```
signal.signal(signal.SIGTERM, sighandler)
```

- ⊙ SIGTERMINAL에 대한 핸들러를 설정함으로써 만일이 사태에 대비한다(서버를 종료할 때 사용하기도 한다.)

▪ 서버 종료

```
echo "stop"
CONFIG_ROOT="config.json"

# get pid
python_pid=$(ps -au | grep "python harmoiserver.py $CONFIG_ROOT$" | awk '{print $2}')

# TODO 전처리 필요

# 프로세스 종료
kill -15 $python_pid
```

- ⊙ 서버를 가동할 때처럼 sh stop.sh를 이용하여 서버를 종료한다. 파이썬이 실행되고 있는 pid를 찾은 다음에(python_pid) kill 명령어를 이용하여

sigterm를 보낸다. 그렇게되면 서버는 이를 받고 handler를 이용하여 서버를 종료하는 데 코드는 다음과 같다.

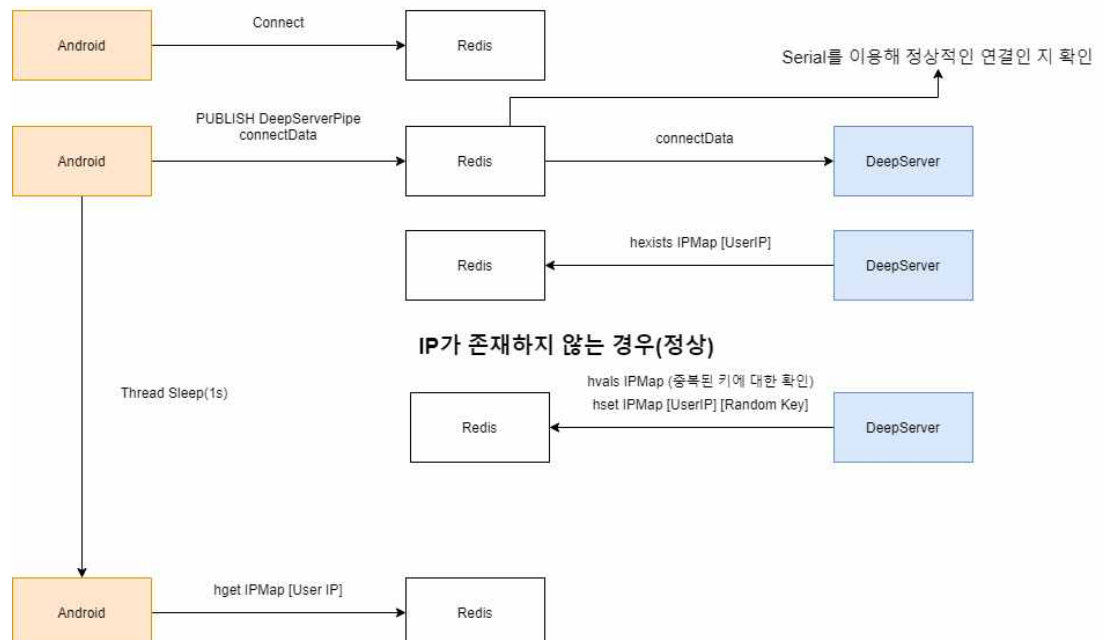
```
# 연결 닫기
def close(self, alreadyDisconnected=False):
    # ...

    # IpMAP 무효화
    if alreadyDisconnected is False:
        self.conn.delete(ServerConnectionKeyLabels.IPMAP.value)
        self.conn.connection_pool.disconnect()
```

- ⊙ 사용하고있는 Redis DB에 저장되어 있는 IpMap를 삭제하여 Redis를 빈 상태로 만든다. 단 Redis 종료는 Server관리자가 직접 해결한다.

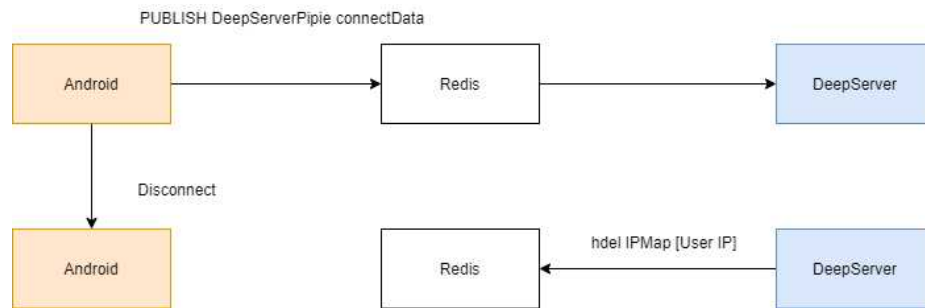
■ 연결 요청

처음 연결 시



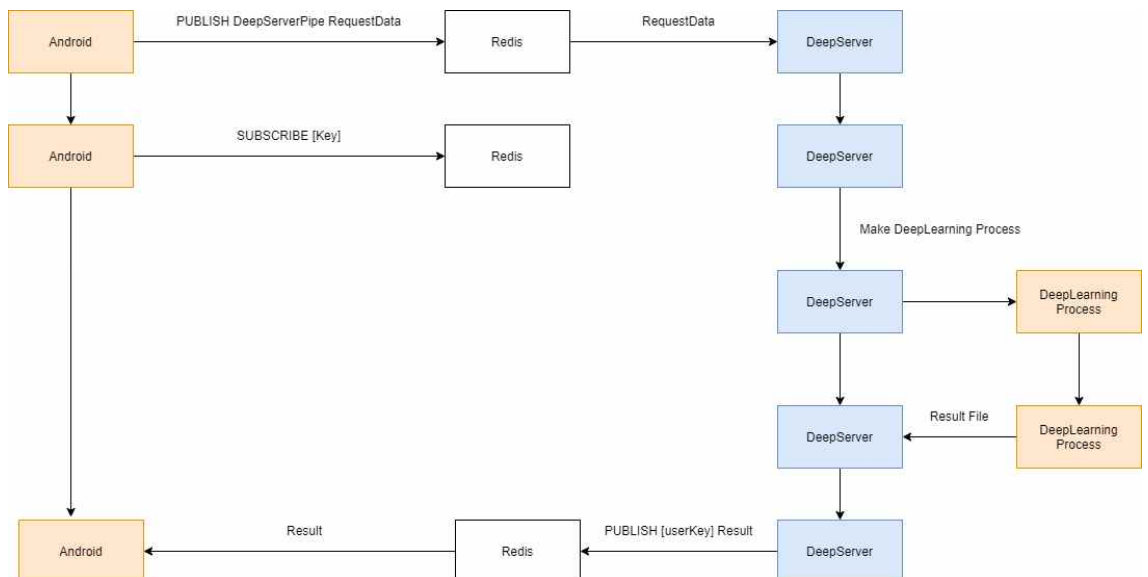
- ⊙ 클라이언트(안드로이드)측은 우선 Redis에 접속하는 것이 우선이므로 연결을 시도한다.
- ⊙ 연결에 성공됐으면 ConnectionData를 Json형식으로 만든 다음 문자열로 변환하여 Server가 구독하고 있는 채널로 데이터를 전송한다.
- ⊙ 데이터를 받은 서버는 문자열 데이터를 다시 Json으로 변환하고 serial를 확인하여 일치한 지 판별한 다음 일치하면 랜덤으로 해당 IP에 대한 10자리 고유키(숫자, 영어)를 생성하여 IPMap에 추가한다.
- ⊙ 클라이언트는 1초 동안 기다린 다음 hget을 이용하여 IPMap에 접근하여 자신의 IP에 해당하는 고유키를 받고 이 고유키로 Redis DB에 채널을 만들고 구독한다. 차후에 서버는 이 고유키로 결과물을 보낼 것이다.

▪ 연결 해제



- ◎ 연결 종료는 요청보다 훨씬 간단하다. 클라이언트는 바로 연결을 끊으면 되고 서버는 IMap에 연결해제를 요청한 IP 필드만 삭제하면 되기 때문이다.

▪ AI 수행 요청

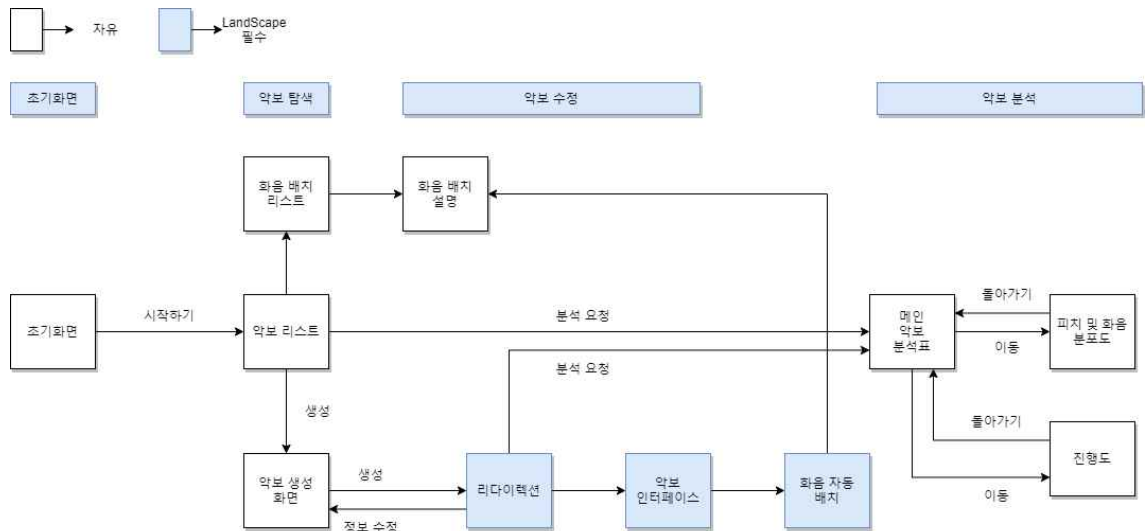


- ◎ AI는 Server에 AI 요청을 하기 위해 딥러닝 서버가 구독하고 있는 채널로 RequestData를 전송하면 서버는 이를 받는다.
- ◎ 클라이언트로부터 받은 데이터를 이용하여 딥러닝 프로세스를 생성하여 실행한다. 실행이 되는 동안 클라이언트는 자신의 고유 키로 채널을 생성하여 데이터가 올 때 까지 기다린다.
- ◎ AI 수행이 끝나면 파일로 생성하게 되고 메인 서버는 이를 byte배열로 전환하여 요청한 클라이언트가 구독하고 있는 채널로 전송한다.
- ◎ 클라이언트는 이를 받고 미디파일로 저장한 후 MidiFile 클래스로 변환하여 악보에 적용한다.

5. 프로토타입

5.1. 개요: 원래는 모든 기능에 대한 mockup을 제작하려고 했으나, 악보 인터페이스가 아직 다 완성이 안 된 문제로 일부분만 기술하였습니다.

5.2. 흐름도



5.3. 악보 리스트



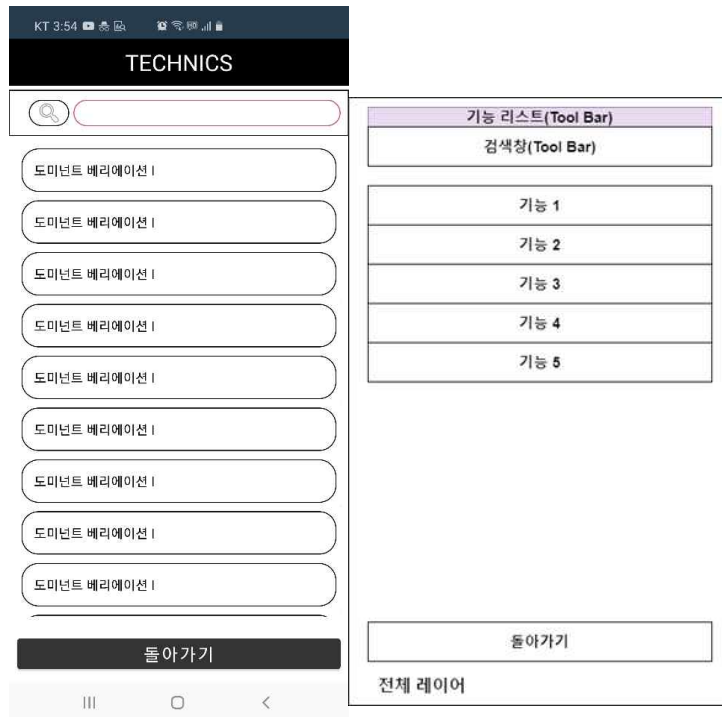
- ⊙ 사용자가 작곡을 한 악보들을 리스트로 나열한 액티비티로 검색란을 통해 원하는 악보를 찾을 수가 있다.

5.4. 악보 리스트(선택 시)



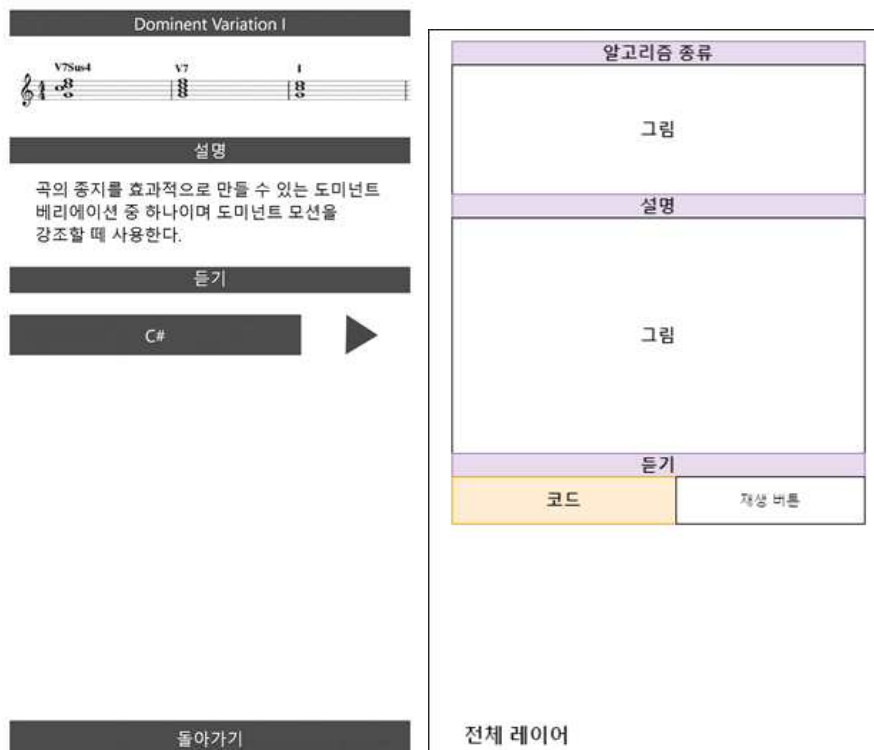
- ⊙ 해당 악보를 선택하면 위와 같은 창이 뜨는 데 코멘트 보기를 이용하여 곡에 대한 설명을 볼 수 있고 분석하기 버튼을 이용하여 미디 데이터를 불러와 분석을 할 수 있으며 열기/취소/삭제 버튼으로 악보를 관리할 수 있다.

5.5. 화음 리스트



- ◎ 화음 배치에 대한 테크닉들을 검색할 수 있게 제작되었으며 해당 항목을 누르면 이에 대한 설명이 나타난다.

5.6. 화음 배치 설명



- ◎ 화음 배치에 대한 설명이 들어가며 듣기란에는 해당 코드에 대한 화음

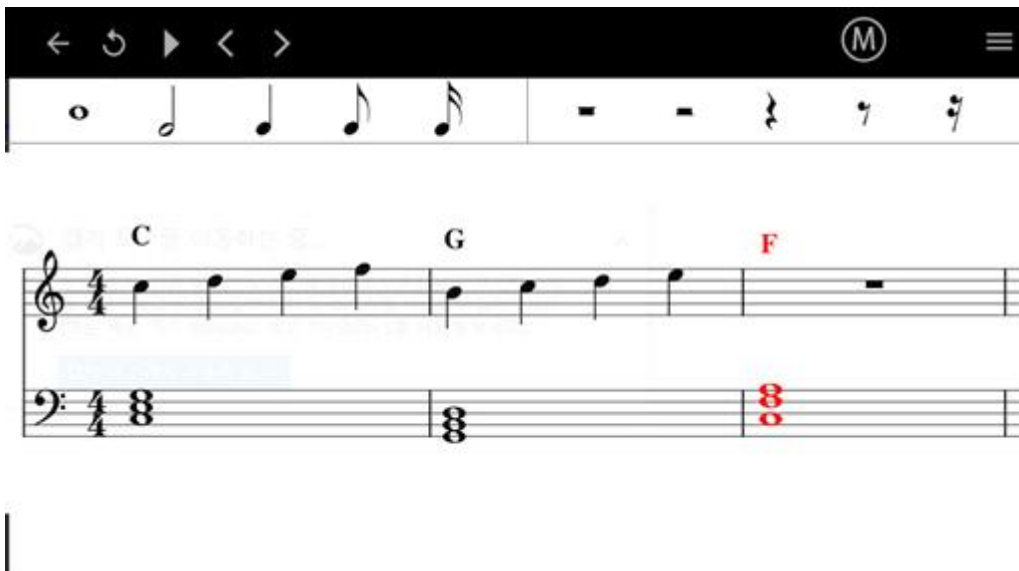
배치를 직접 들을 수 있다.

5.7. 악보 생성 화면



⊙ 악보를 생성할 때 사용하는 화면

5.8. 악보 화면 및 기능 실행



- ⊙ 악보를 생성하면 다음과 같은 창이 나타난다.
- ⊙ 기본적인 음표, 쉬표 생성 및 삭제가 가능하며 스크롤을 이용해 악보를 이동하고 화음배치 기능을 실행하면 반주부분과 코드부분에 생성을 한다.

5.9 AI를 이용해 분위기에 따른 작곡 수행

- ⊙ 분위기가 좋은 경우와 슬픈 경우 두 가지로 나뉘어서 기능을 수행.
- ⊙ 결과의 외관은 5.8과 비슷하나 멜로디도 추가로 작곡됨.

III. 결론

1. 연구 결과

- LSTM을 이용하여 기본적인 AI작곡이 가능하며 차후 데이터셋만 잘 세팅한다면 단순히 Major Minor가 아닌 다양한 스타일의 작곡이 가능할 수 있다.
- AI 엔진은 하드웨어적 성능이 만족하지 않으면 반드시 서버에서 진행해야 한다.

2. 소요 재료

- Google Platform GUI Server

참고자료

Android - Midi Device 통신 방법:

<https://developer.android.com/reference/android/media/midi/package-summary>

Key Finding Algorithm:

<http://rnhart.net/articles/key-finding/>

Key Finding Algorithm(논문) - What's Key for Key? The Krumhansl-Schmuckler

Key-Finding Algorithm Reconsidered

<http://davidtemperley.com/wp-content/uploads/2015/11/temperley-mp99.pdf>

파이썬 Music21 라이브러리를 활용한 미디 데이터 분석

<https://inspiringpeople.github.io/data%20analysis/midi-music-data-extraction-using->

Magenta Project: <https://magenta.tensorflow.org/>

Github Project

메인 프로젝트

최종 프로젝트: <https://github.com/Re-Coma/HarmoAssist>

Server: <https://github.com/Re-Coma/HarmoAIComposorModule>

병합 이후에 삭제될 예정인 프로젝트

화음 배치 모듈: <https://github.com/Re-Coma/HarmoControl>

미디 데이터 컨트롤 모듈: <https://github.com/Re-Coma/Android-Midi-Control-Library>