Masaryk University
Faculty of Informatics

Web interface for music notation editing

Bachelor's Thesis

Miroslav Šerý

Brno, Spring 2021

Masaryk University
Faculty of Informatics

Web interface for music notation editing

Bachelor's Thesis

Miroslav Šerý

Brno, Spring 2021

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Miroslav Šerý

Advisor: doc. RNDr. Petr Sojka, Ph.D.

Acknowledgements

I want to express sincere thanks to my supervisor, doc. RNDr. Petr Sojka, Ph.D., for guidance and motivation throughout my work on this thesis. Moreover, I am grateful to the team of ProScholy.cz, especially to Filip Kratoš, for helpful insights, discussions, and feedback. My thanks also belong to my family and friends for exemplary support. And above all, to *Thee who art in Heaven*.

Abstract

This thesis deals with implementing a web interface for music notation editing based on the GNU LilyPond engraving software. The goal was to create a flexible notation input system with dynamic export features to be deployed within a public songbook Zpěvník.

I created a specific interface that can work with notation segments and produce the final score by their concatenation. This was motivated by a typical structure of popular songs with repeated sections. I integrated this interface within an internal content administration of the Zpěvník project, and it has served as a means of transcribing sheet music optimized for mobile devices.

The concatenation process is implemented directly within the LilyPond input format by creating a custom extension of a built-in template for choral music. I made an abstraction logic for LilyPond's CLI because the rendering needs to be executed server-side. Finally, I developed a front-end GUI and a back-end support logic that build upon the existing codebase of Zpěvník. The interface fulfills its goals within the internal administration. However, public export is currently limited.

The approach of segmented notation introduces extra structure in the input format, which gives more freedom to the output possibilities. I consider this to be a distinguishable feature of the interface in the context of other standard notation GUIs.

Keywords

sheet music, music notation, LilyPond, lead sheet, chords, web development, JavaScript, PHP, Vue.js, Scheme

Contents

Introduction **1**

1 Music-related software **3**
1.1 Sheet music documents 3
1.2 Notation encoding approaches 5
1.3 Web-based rendering 6
1.4 Sheet music on Zpěvník 7
1.5 Segmented input reasoning 7

2 GNU LilyPond **9**
2.1 History and context 9
2.2 Input syntax introduction 9
2.2.1 Notation basics 10
2.2.2 Score composition with variables 14
2.3 Compilation and rendering 15
2.3.1 SVG 15
2.3.2 MIDI 16
2.4 Projects related to LilyPond 16

3 Framework for segmented music notation **17**
3.1 The LilyPond template logic 17
3.1.1 Voices as variables 18
3.1.2 Short syntax of polyphony with shared rhythm 18
3.1.3 Segment concatenation 19
3.1.4 Key and time signatures 20
3.1.5 Keeping the staves alive 20
3.1.6 Fixing the positions of lyrics and chords 21
3.1.7 Output customizing 21
3.1.8 Known issues and drawbacks 22
3.2 Renderer server and client 23
3.2.1 Generating renderable code 23
3.2.2 Client-server communication 24
3.2.3 Output rendering and postprocessing 25
3.3 Summary 25

4 Web interface **27**

4.1 Requirements and design considerations 27
4.1.1 Representation of multiple renders 28
4.1.2 Queued job processing 28
4.1.3 Storage for rendered score files 28
4.2 Technological background 29
4.3 Front-end GUI 32
4.3.1 Score setup 33
4.3.2 Segment input 33
4.3.3 Concatenated preview 33
4.3.4 Technical aspects 35
4.4 Server side 35
4.4.1 Database and API extending 35
4.4.2 Implementing services 36
4.5 Deployment and testing 37
4.6 Current shortcomings 37
4.7 Summary 38

5 Conclusion **39**

Bibliography **41**

A Content of the attachment **45**

B Zpěvník's usage statistics **47**

C PHP Client library example workflow **49**

List of Figures

1.1 Development stages of music notation editing interface 8

2.1 Example of a basic C major scale 10
2.2 Example of a polyphonic input 10
2.3 Example of explicit time and key signatures 11
2.4 Example of basic notes and rests 12
2.5 Example of ties and dotted notation 12
2.6 Example of tuplets 13
2.7 Example of lyrics with melismata 13
2.8 Example of different barlines 14

3.1 Objective of the LilyPond framework 17
3.2 LilyPond template file structure 19
3.3 Repeated key and time signatures 20
3.4 Displacement of the choral bracket 21
3.5 A template issue – slurs in soloMale variable 23
3.6 Data flow diagram for renderer client and server 24

4.1 Application infrastructure diagram 30
4.2 Sequences of events after chosen user actions 31
4.3 UI layout wireframe 32
4.4 User inteface – segment input 34
4.5 ER Diagram for database extending 36

B.1 Report from the Google Analytics tool 47

Introduction

Music notation plays an essential role for most interpreters and performers. Years of software development within this area have brought about many different approaches to work with sheet music, be it using proprietary applications or leveraging open-source libraries.

The Czech-based volunteer initiative ProScholy aims to collect and share Christian church music. The first product is an online public songbook – Zpěvník – which focuses on bridging the gap between the church songs' authors and the interpreters. A convenient and flexible platform for distributing sheet music should become an invaluable contribution to this process.

Personal motivation

My work reflects my experience of being a member of the ProScholy team full of musically and technically educated people, some of whom are also practiced in sheet-music production. I am thankful for the possibility to contribute to this project and to be a part of a team where lengthy but fruitful discussions are not an exception.

Utilizing textual formats for music notation representation was a novelty for me, and I would like to continue to explore their possibilities. I hope this work can bring this attitude to more people interested in sheet music production.

Scope of this thesis

Firstly, this thesis focuses on leveraging GNU LilyPond as the back-end for creating a custom music notation template. On top of that, it concerns ways of its incorporation into Zpěvník's environment by creating an interactive user interface and a necessary back-end support logic.

I consider dividing sheet music into short reusable segments and implementing this idea within LilyPond as my essential contribution. I reckon that with some amount of preprocessing, this idea applies to more text-based music notation representations.

Overview of chapters

The first Chapter (1) introduces commonly used notation software and an overview of chosen open-source notation encoding formats. Lastly, it outlines the gradual process of sheet music support within the Zpěvník project and reasoning about segmenting the input. The thesis continues with Chapter 2 describing the GNU LilyPond software – mainly its basic syntax and principles, output formats, and related community projects. Chapter 3 discusses developing a LilyPond template for segmented music and a set of support tools to produce sheet music within this template. The template is "brought alive" by a user interface and a support infrastructure that make up the content of Chapter 4. Possibilities for further development of the interface are mentioned, too. The outtakes of the work are revised in conclusion.

1 Music-related software

Same as document preparation, ways of producing, storing, and interpreting digital music scores have gone through rich development. This chapter introduces some of the popular notation software and tools for musical analysis and packages for web notation rendering. Lastly, it discusses the context of music notation editing within the Zpěvník project.

1.1 Sheet music documents

To get an overview of common notation software principles, an analogy can be made showing a correspondence between the music-specific and generic electronic-document worlds:

- Sibelius, Finale — Microsoft Word

  Sibelius and Finale represent a group of proprietary Graphical User Interface (GUI) applications built upon the WYSIWYG principle, meaning the users do not operate with any form of source code. Every change happens in place and instantly. They are primarily targeted at the MS Windows operating system.

- MuseScore — LibreOffice

  MuseScore is an open-source multi-platform desktop application based on the same principles as above (GUI/WYSIWIG). Besides the desktop software, MuseScore also provides an online ecosystem [2] for sharing sheet music within its format.[1]

  Thanks to its free license and intuitive user interface, it has become a well-established piece of notation software among many amateur musicians, including myself.

1. Until 2020, free users could upload only up to five public scores, but the limit has nowadays been removed. [3]

- GNU LilyPond — LaTeX

  GNU LilyPond[2] is an extensive music engraving system conceptually different from the abovementioned applications. It consists of a markup language and a typesetting engine invokable via a Command Line Interface (CLI) – the workflow is almost identical to producing LaTeX documents.

- Frescobaldi — TeXstudio, TeXworks

  Frescobaldi is an open-source desktop application – a visual front-end that simplifies the workflow of using the LilyPond engine. It also comes with useful functions to create and adjust LilyPond's textual notation format. For the actual rendering, it needs a working installation of LilyPond on the local machine.

- MusicXML — XML

  MusicXML is a structured text format based on the XML syntax. It was made with the purpose to become sort of a *lingua franca* among different notation software. [4] However, from my personal experience, using MusicXML for conversion is often prone to errors and inconsistencies as the level of support varies among the notation software.

- OMR — OCR

  Optical Music Recognition is an extension of OCR that concerns transcribing scores into machine-readable formats such as MusicXML. This task is much more challenging from the technical perspective because of the two-dimensional relations characteristic of sheet music.[3]

2. LilyPond is released under the GNU license. In this thesis, the term "LilyPond" is used interchangeably to denote the software and its input code format.
3. See [5] for an overview of state-of-the-art approaches to OMR.

1.2 Notation encoding approaches

Below is a list of several approaches to represent and work with music notation in a computational way, which means that some programming skills are generally necessary to use up their possibilities. All of them have been well-maintained in the long term. Together with LilyPond and many other packages, they can be considered as a subset of a concept of *musical programming* [6], which itself also encompasses (live) music production and similar scenarios.

- Music Encoding Initiative (MEI)

  MEI is a music researchers' take on creating a representation format for various musical documents. Same as MusicXML, it is specified by XML standards. Its primary target is to capture the abstract semantics of music independently on the engraving style (opposed to LilyPond) and notation software requirements (op. to MusicXML). [7, 8]

- Humdrum

  Humdrum is a textual format for encoding music notation and a set of tools for musical analysis of documents stored within this format. Humdrum tools are platform-agnostic, which means they can be embraced by various programming environments.

- music21

  Music21 is a Python library focused on the field of computer-aided musicology. It provides tools for musical analysis together with native interfaces for importing and exporting other notation formats, such as MusicXML, MEI, or Humdrum. LilyPond can be utilized to perform the output rendering process.

- abjad

  Abjad is a Python package for building complex LilyPond scores in an incremental way by providing an object-model abstraction layer over many LilyPond constructs and the CLI functionality.

1.3 Web-based rendering

There are two distinct approaches regarding processing sheet music on the web – client-side and server-side. The former means the actual rendering happens directly in the client browser. Current browser engines (such as V8) support executing the JavaScript and WebAssembly languages, [9] which means that any client-side program needs to be written in or compiled into one of them.

Below is a selection of remarkable packages for client-side sheet music rendering that have been maintained. I consider Verovio as one of the major leaps to universal web engraving of music notation based on a well-adopted format (MEI).

- VexFlow

  VexFlow is a JavaScript library; it renders sheet music based on a custom object-model representation. The OSMD project uses it as a backend for rendering MusicXML files.

- abcjs

  Abcjs is a JavaScript library; it renders sheet music in a popular and simple ABC format.

- Verovio

  Verovio is written in C++, but also compiled to JavaScript. It renders scores natively in the MEI music format. It can work with ABC, MusicXML, Humdrum and other formats, too.

So far, there is no client-side implementation of LilyPond available.[4] Consequently, sheet music written in LilyPond can be rendered only server-side and delivered as an image to the client. Therefore, a server running a LilyPond instance is necessary. For the rendering to happen on-the-fly, as in the case of online LilyPond editors Hacklily or LilyBin, the renderer server must also implement an API accessible by the front-end.

4. It requires a complete Scheme interpreter to execute a fully functional LilyPond renderer.

1.4 Sheet music on Zpěvník

Since its beginnings, Zpěvník has supported storing sheet music documents (PDF) within its song database. As of 5/2021, a collection of over 800 such files have been uploaded by content editors. Nonetheless, the majority of Zpěvník users is coming from mobile devices[5] and not surprisingly, sheet music with the primary purpose of being printed on (A4) paper is not optimal for viewing on narrow mobile screens.

Hence, there was a call for a platform for storing unified sheet music. It should fulfill Zpěvník's use-cases and requirements, mainly that the outputs should be optimized for mobile devices and easily editable within its web administration.

Preferring LilyPond as a backend for sheet music rendering over other abovementioned solutions was motivated by several facts. Firstly, and maybe most importantly, it is its popularity among the Zpěvník content editors and generally among Czech Christian musicians.[6]

Due to a lack of a client-side rendering ability, LilyPond might not seem ideal for the web. Nonetheless, it is still one of the most extensible engraving systems with an intuitive textual input.

1.5 Segmented input reasoning

Apart from smaller layout width, the optimization for mobile devices should consider the infinite scrollable space, which expects an opposite approach as with printed scores, where the notation is usually as concise as possible, often by ignoring repeated sections. This is where the idea of segmented input comes from – one input can be used to produce both the succinct notation and an extended one just by defining the desirable order of segments. This approach has other advantages and also limitations, which are discussed in relevant parts of this thesis.

5. See the detailed statistics in Figure B.1
6. See e.g. `https://github.com/jirihon/lilypond-scores`, `https://github.com/igneus/In-adiutorium` or `https://kancional.cz`

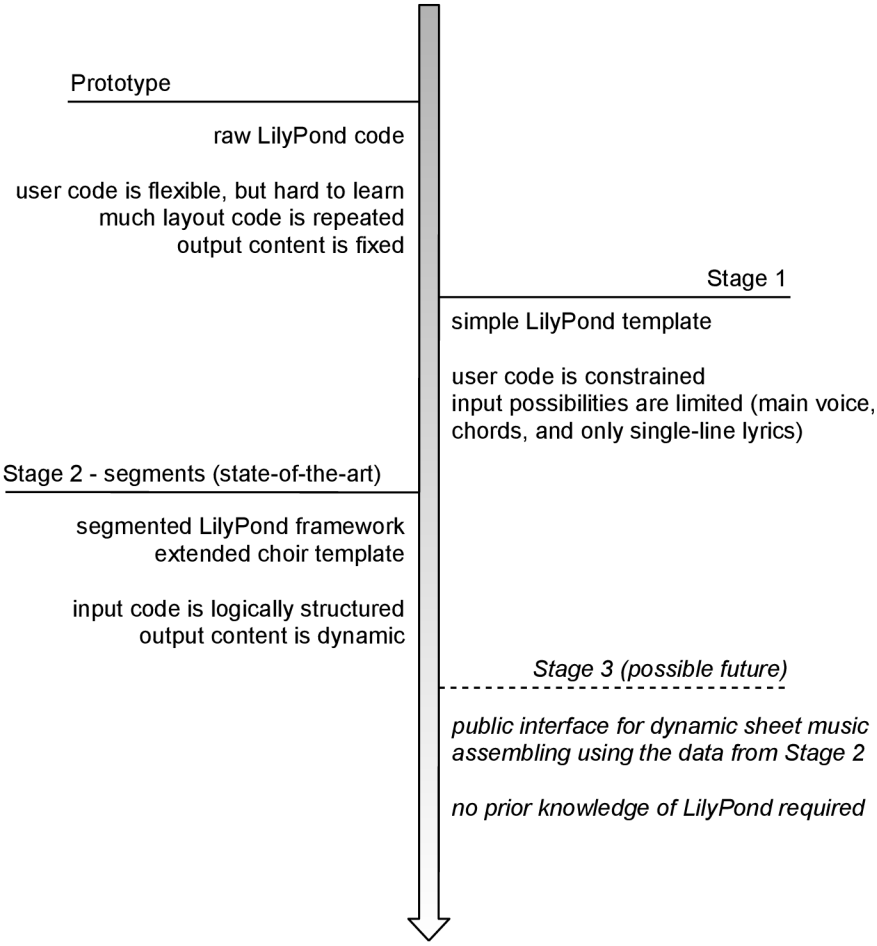

Figure 1.1: Development of the user interface features and underlying LilyPond support code went through several stages. It can be characterized as moving towards easier user adoption and more output freedom at the cost of greater implementation complexity.

2 GNU LilyPond

Using GNU LilyPond as the rendering engine assumes working with its input language syntax. Thus, apart from overview of LilyPond's history and related projects, this chapter presents LilyPond's input principles within a relevant scope.

2.1 History and context

GNU LilyPond is a long-running project started in 1996 by Han-Wen Nienhuys and Jan Nieuwenhuizen out of dissatisfaction with then-current engraving systems.[1] [12]

The primary orientation on engraving qualities is important as it explains the design decisions of the LilyPond input format. [13, pp. 1–11] The format is optimized for comfortable and human-readable music notation *typesetting*, as opposed to e.g., MEI aiming at general music representation.

Both LilyPond renderer and the input format deeply integrate the Scheme programming language. [14, p. 1] Scheme functions can be embedded directly alongside the notation, which makes LilyPond a computationally powerful and extendable system. On the other hand, this brings an inherent complexity regarding parsing, converting to other formats, etc., compared to stricter formats.

One of LilyPond's major advantages is its extensive documentation available either as a web page [15] or as compiled PDFs [14, 16, 17, 18]. However, this amount of resources also makes up a difficult learning curve, especially for non-technical users.

2.2 Input syntax introduction

While LilyPond supports many different notation styles [16, pp. 448–487], the following text concerns basics of so-called *common western music notation*. It is meant to introduce LilyPond syntax constructs and related musical terminology.

1. These systems, such as MusiXTeX [10] continued to be further developed, see [11] for overview.

2.2.1 Notation basics

The note's *pitch* is denoted by letters *a* through *g* as usual in the music terminology.[2] They can be adjoined by an alternation (sharp, flat) written as a suffix (-is, -es). Another well-known musical term is the *octave*. In LilyPond, the note's octave can be expressed as zero or more single quotation mark characters or commas (e.g. `c''` or `d,`).

```
{ c' d' e' f' | g' a' b' c'' }
```

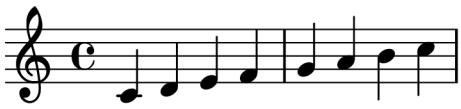

Figure 2.1: Construction of the basic C major scale starting from `c'`

The `\relative` directive is (routinely) used to save keystrokes, as most of neighbor note pairs happen to be pitch-wise near from each other [19] – their *interval* is small. This directive makes use of such property and recursively expects the next note to lie within an interval *less than a fifth* with the last one. Should the interval be bigger, the resulting octave is to be changed by extra octave marks.[3]

Angle brackets in LilyPond have unified semantics – they can be read as *in parallel*. There are more variations[4] on using them; the basic one is to use single angle brackets to make a chord out of multiple notes.

```
\relative { d' f a <d, f a> | d a' c <d, a' c> }
```

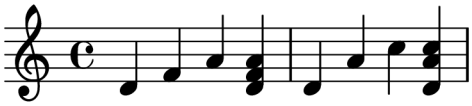

Figure 2.2: Two chords are written as consecutive notes and then as polyphony. Correct writing of octave marks is a matter of practice.

2. LilyPond supports also different language-specific codes, such as Italian *do-re-mi*. [16, p. 8]
3. For more details about the absolute and relative octave syntax, see [16, pp. 2–5].
4. E.g., double angle brackets for music expressions, or generally some complex objects.

A *key signature* at the beginning of a score section indicates which of the follow-up pitches are altered by default[5]. In the input syntax, this is expressed by the `\key` directive, followed by a scale name – the base note together with either `\major`, `\minor`, or another mode.

Next signature is the *time signature* or simply *time*. The basic time is 4/4[6] which is also the default LilyPond setting. This can be overridden by a `\time` directive followed by a fraction number. Generally, time influences the division of the score into *measures*[7].

```
\relative {
 \time 3/4
 \key e \major

 e' gis b | fis a cis | gis b dis
}
```

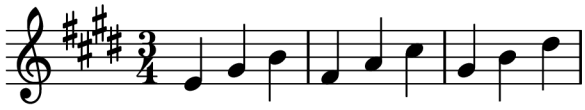

Figure 2.3: Time and key signatures are explicitly defined at the beginning of the score. Notice that LilyPond automatically renders them correctly, regardless of their source code order.

An important part of rhythm is a "musical pause" or rather a *rest*. Entering a rest as simple as entering a hypothetical note named `r`.

Both notes and rests have their specific notation symbols based on their *duration*. Duration is more often than not expressed fractionally – the basic unit is the *whole note* (*semibreve*) spanning a full measure in the 4/4 time. In LilyPond, the fractional duration is expressed as a number suffix.[8]

```
\relative { g'1 | g2 g4 g8 g16 g | r1 | r2 r4 r8 r16 r }
```

5. In the graphical representation. One still needs to use the alternation suffixes in the input code.
6. Commonly abbreviated as **C**
7. Also known as *bars*. It is a good practice to include bar-check operators '|' to make the source code clearer and instruct the compiler to check for rhythmic inconsistencies by printing eventual warnings.
8. Multiplicative durations, that is those longer than the semibreve, have their special directives.

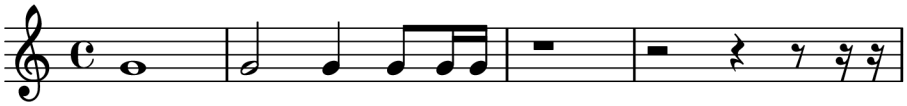

Figure 2.4: A whole note followed by *half note* (*minim*), *quarter note* (*crotchet*), *eight note* (*quaver*) and two *sixteenth notes* (*semiquaver*). The same pattern is repeated using rests.

When the number suffix is missing, LilyPond recursively uses the duration of the previous note.[9] If the duration is explicit, it can be extended by appending one or more dots to form a *dotted notation* where each dot adds up 50%.

A *tie* is used to join two consecutive notes of the same pitch to form a compound duration.[10] In LilyPond, tie is expressed by the *tilde* character ~.

```
\relative { c''4. g8 ~ g4. e8 | c'4. g2 e8  }
```

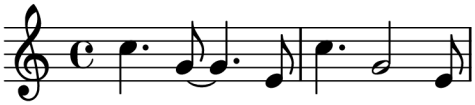

Figure 2.5: Sometimes it can be unclear, whether to use tie or not. These two bars have equivalent melody, but the first one is preferable as phrasing patterns are evident.

```
\relative {
  \tuplet 3/2 { f'2 f f }
  \time 3/4
  \tuplet 4/3 { f4 f f f }
}
```

To typeset music lyrics, LilyPond offers a simple directive `\addlyrics` that is ideal for short examples and simple melodies, but using it in a more elaborate way (such as in a multiple-voice setup) may lead to

9. By default, it applies quarter note duration, such as in the first three examples. However, it is recommended to always start with an explicit duration for writing complex scores with variables.
10. A common use-case for the ties is, e.g., when the note should last over an end of a measure.

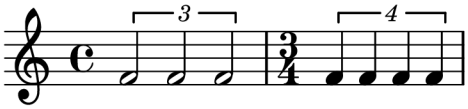

Figure 2.6: Example of two *tuplets* dividing rhythm differently than normally allowed by the time signature.

unexpected results. LilyPond also provides another way to add lyrics via variables, as used in Section 2.2.2.

A *slur* is delimited by an opening bracket '(' *after* the first slurred note, and a closing bracket ')' after the last slurred note. Slurs have a specific purpose when lyrics is present – they cause a *melisma*.

```
\relative {
   \partial 2
   b'4.( c8) | c1 ~ | c2 e |
   d4.( c8 b4. a8 g2) r
}
\addlyrics {
  A -- bba A -- bba __
}
```

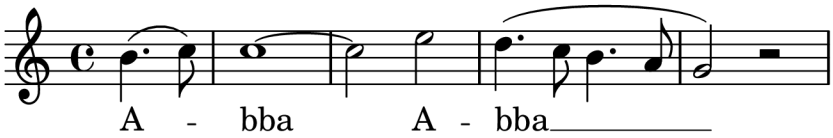

Figure 2.7: Melismata are formed under slurred groups of notes (1st and 4th measure) – they are characterized by singing one syllable over multiple notes. For the sake of readability, they can be marked with an extender.

Usually, the score of a song is visually broken down into sections by using specific barlines. In LilyPond, this can be achieved by inserting e.g. `\bar "||"` for separating inner sections or `\bar "\."` for the ending barline.

Barlines can also be altered implicitly by some directives, such as *long repeats*[11]. There are multiple variants of repeats [16, pp. 152–159] whereas the common one for songs with lyrics is shown in Figure 2.8 and adjacent code example.

11. This term comes from LilyPond documentation.

```
\relative {
  \repeat volta 2 { c' f a c }
  \alternative {{ a f a c } | { a d a f }}
  \bar "||" c f a c \bar "|."
}
```

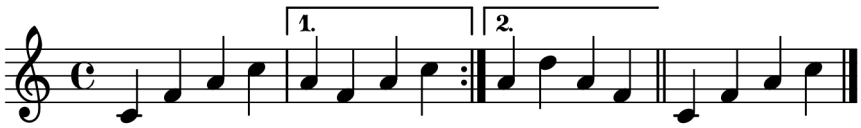

Figure 2.8: Score with four different barlines (single, repeat, double and ending). The latter two are set-up explicitly by `\bar` directives.

2.2.2 Score composition with variables

LilyPond is fairly benevolent in the means of structuring the notation data. A common practice is to define music in *variables* that can be reused later. Thus, a more elaborate LilyPond input file usually looks like this:

1. Header – Specification of the version, paper and layout configuration is accomplished via specific directives.

```
\version "2.22.0"
\paper {
    indent = 1\mm
}
\header {
    composer = "author"
}
\layout {
  \context {
    \Voice
    \override NoteHead.font-size = #4
  }
}
```

2. Body – Musical expressions are assigned to variables. By convention, variable names consist only of alphabetic letters.

```
melody = \relative { c'4 d2. }
text = \lyricmode { He -- llo }
```

3. Footer – So-called *contexts* (Voice, Lyrics, Staff,...) are defined within a `\score` block and filled with variables' content.

```
\score {
  <<
    \new Voice = "one" { \melody }
    \new Lyrics \lyricsto "one" { \text }
  >>
}
```

For some use-cases, it is convenient to put some common parts of LilyPond code into separate files and include them via the `\include` directive when necessary. The score block can also be defined dynamically based on available variables – this is the principle of the Chapter 3 LilyPond template.

2.3 Compilation and rendering

A minimal workflow of compiling LilyPond files consists of executing `lilypond file_name.ly`. The CLI includes multiple options, such as specifying the output format, which can be SVG, PDF, PS, EPS, PNG along with MIDI. [18, pp. 2–3]

Regarding use-cases for web and mobile applications, especially useful are the SVG and MIDI exports. Following is a brief description of these formats in the context of the LilyPond render process.

2.3.1 SVG

Scalable Vector Graphics file format (SVG) is de facto a standard for vector images on the web as all major web browsers support it. [20] It is based on XML and developed by Word Wide Web Consortium (W3C).

In LilyPond, SVG files are rendered by a specific backend that needs to be triggered via `--svg` option. By default, a fixed (A4) paper size is assumed and page breaks are enforced for any output format. It is desirable to disable fixed page height and enforce "one-page breaking" while rendering SVG images. [16, p. 567]

2.3.2 MIDI

Musical Instrument Digital Interface (MIDI) is a binary standard for exchanging musical data. Notation software, electronic or virtual music instruments or visualizers, and modern web browsers[12] support the MIDI interface. A so-called sound font is required to playback a MIDI file because the raw binary data don't encode any sounds, just metadata. A special `\midi` block needs to be included in the `\score` section to produce a MIDI file during the LilyPond rendering.

2.4 Projects related to LilyPond

Below is a list of a couple of community projects related to LilyPond that I reckon may be useful, even though I did not use them within the scope of this thesis. The Frescobaldi desktop application and the abjad Python library were already mentioned in the introductory Section 1.2.

- LyLuaTex

  *ly*LuaTEX is a LuaLATEX package that manages inclusion of LilyPond snippets within text documents. It is supposed to be a replacement for the native lilypond-book preprocessor included with LilyPond. [22]

- OpenLilyLib

  OpenLilyLib is a robust extension framework for LilyPond. However, many of its packages are still considered to be underdocumented or incomplete. Most notably, it provides an extensive snippet repository.

- Mutopia

  Mutopia is a collection of sheet music typeset with LilyPond by community volunteers. It can search and filter by various categories, and the scores are available to be downloaded in multiple formats, including the LilyPond source.

12. The standard for browsers is named WebMIDI. [21]

3 Framework for segmented music notation

The LilyPond input format is, in fact, a programming language, which makes it a highly extendable system. This chapter starts with a description of a custom LilyPond template. This template, together with user input, make up *renderable code*[1]. The text follows with a set of tools that serve as wrappers for the LilyPond-specific functionality. Namely, they implement renderable code assembling, executing the CLI, and postprocessing the output files. Another way to characterize this chapter is that it manages the process of transformation of the user input into rendered scores, as depicted in Figure 3.1.

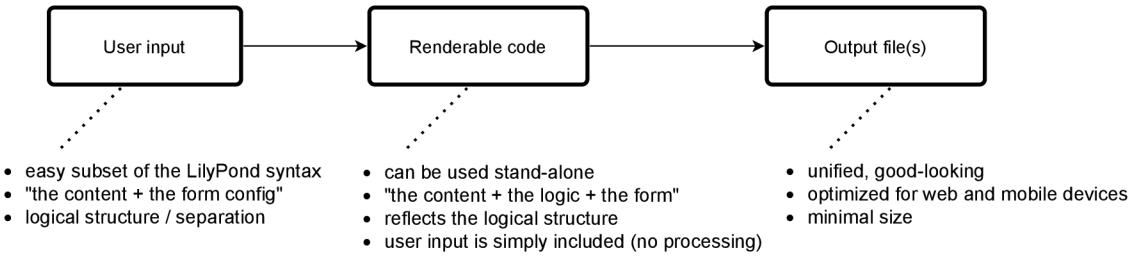

Figure 3.1: The objective of the framework is to make this illustrated transformation process possible and straightforward. The "form config" is mapped to pieces of LilyPond code that influence the form of the output.

3.1 The LilyPond template logic

Section 2.2.2 introduced an arbitrary example of renderable code structured into three parts – header, body, and footer. Thus, a template should provide the functionality of the header and footer so that the user code contained within the body can be reasonably simple.

One such template – `ly/satb.ly` is defined directly in the GNU LilyPond repository. After my first attempts at extending raw LilyPond, I chose to build upon this elaborate template.[2] The changes

1. Renderable code in this text denotes such LilyPond input that the musical content is stored in variables and can be successfully rendered.
2. This helped to solve some beginner's issues with extending LilyPond.

include renaming the expected voice variables, adding extra functionality to ease some notation practices, and implementing correct segment concatenation. These changes are presented in detail in the following text.

3.1.1 Voices as variables

The prerequisite of separating the content is using variables within the body. The original `ly/satb.ly` template checks for presence of variables with various names like `SopranoMusic` or `SopranoLyrics` in the user input. It can also switch between four-stave and two-stave layout based on a definition of a special variable `TwoVoicesPerStaff`. [17, pp. 152–156]

```
solo = \relative { c' d e f g }
soloText = \lyricmode { La -- la -- la Lay -- la }
soloTextI = \lyricmode { Se -- cond line }
```

Listing 3.1: The expected property names could be changed to meet the local language. In contrast, the `\relative` and `\lyricmode` directives are built-in LilyPond constructs and thus had to stay unchanged to avoid overriding the base syntax.

3.1.2 Short syntax of polyphony with shared rhythm

When the secondary voices share rhythm with the leading voice, it is typical to use the chorded polyphony syntax where the leading voice is emphasized by making the other voices' notes smaller. With the help of a custom Scheme function, it is possible to enter such small notes in a concise syntax. A variable named `soloMale` is used for this purpose.

```
solo = \relative { d'4 d8 e16 d \tuplet 3/2 { c4 d c } }
soloMale = \relative { a' b a a s a b }

% alternative = \relative { <d' a'>4 <d b'>8 <e a>16 <d a'>
    \tuplet 3/2 { c4 <d a'> <c b'> } }
```

Listing 3.2: Short polyphony syntax makes the source code clearer than the standard polyphony input shown in the commented part of the example.

3.1.3 Segment concatenation

The idea of segmented music notation follows the fact that most popular songs are divisible into sections, such as verses or a chorus. While music *movements* in a classical music are visually separated[3], shorter sections in popular songs usually follow directly one after another.

The concatenation process should also respect the continuity and not enforce visual breaks. Moreover, it should be possibly implemented within the LilyPond template logic itself so that this template can be used independently.

Given that the user input is defined in variables, the first idea is to merge equally named variables from different segments. For simple music with one melody variable per segment, this is indeed reasonable. However, when the variable structure gets rich or differs among segments, especially when working with lyrics, this approach is inefficient and error-prone.

The solution to an effective segment "gluing" within LilyPond is mentioned in the documentation of the abjad library [23], with the specific idea being referred to as *Lilypond Context Concatenation*[4]. One of the key insights is using `\context` instead of `\new` directive while creating notation contexts.

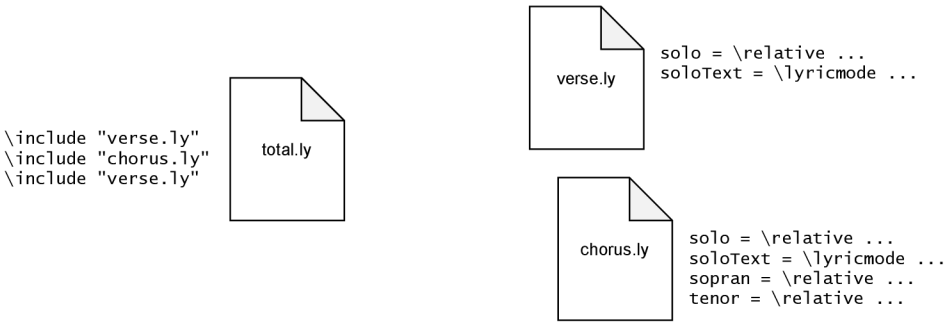

Figure 3.2: The structure of variables stays the same for each segment, but a different set of variables may be used. The total score is composed by simply including the segment files.

3. This can be achieved by creating multiple `\score` blocks in the LilyPond input.
4. Contexts are generally somewhat of a complicated concept in LilyPond. Interestingly, this use case is neither mentioned in the official documentation nor is evident

3.1.4 Key and time signatures

Separating the score into segments requires custom handling of time and key signatures because each segment must exist (and thus be rendered) independently of the rest. For that, it needs time and key to be explicitly defined. By default, this results in redundancy displayed in Figure 3.3.

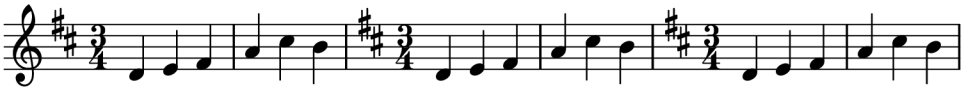

Figure 3.3: This piece of music was composed of three (equal) segments with explicit signatures, which results in rendering useless signature "changes". This is obviously not a satisfactory outcome.

LilyPond offers native ways to change this behavior, but they do not cover this specific scenario. As a solution, the leading key signature in each segment is supposed to be stored in a separate variable. Therefore the template can include it conditionally, based on the preceding musical content. The same logic is applied to time signatures.[5]

3.1.5 Keeping the staves alive

A different number of voices among the segments may result in a suboptimal behavior after the concatenation. This is caused by the fact that in the original `satb.ly` template, undefined voice variables are by default left empty.

This issue can be solved generally by filling up all empty voice variables with rests, which plays well together with a specific LilyPond setting to render staves only when necessary. [16, p. 207] There are two forms of rests data, the "normal" rests and so-called *spacer rests*[6] and two respective internal functions `mmrest-of-length` and `skip-of-length` for transforming music data into rests of equal duration.

Using ordinary rests for filling up empty bars is visually preferable. However, it doesn't function well for segments that start or end with a

5. Key and time need to be handled a bit differently concerning notation conventions and settings available in LilyPond. See [16, pp. 646–648] for more info.
6. Spacer rests do not produce any graphical objects and work simply as placeholders.

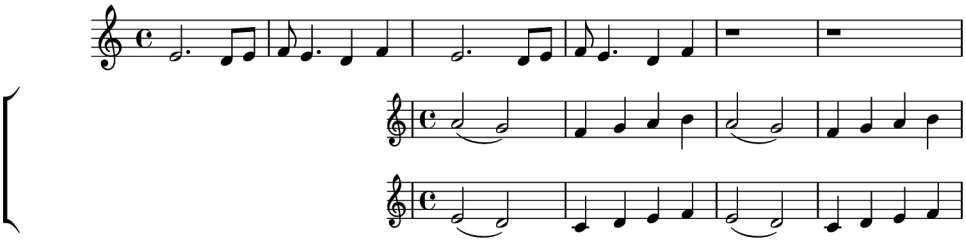

Figure 3.4: The first segment consisting of the first two bars includes only a solo voice, whereas the next segment includes three voices. Most notably, the choral bracket is displaced and a blank space is left.

partial measure. Therefore, spacer rests are used by default for their better flexibility.

3.1.6 Fixing the positions of lyrics and chords

The process of debugging issues with lyrics during segment concatenation became the most cumbersome part of my experience with LilyPond. I experimented with the `\addlyrics` command [17, p. 31], which only led to many unexpected results. So I chose to stick to represent lyrics with variables. However, this did not solve all concatenation-related issues. Therefore I had to create specific and rather ad-hoc functions copying the voice content into lyric variables to fix the renderer behavior. The final solution uses a rather under-documented internal function `extract-named-music` to filter out problematic data types, which seems to be a stable solution.

Another displacement caused by concatenating would occur to chord marks. By default, LilyPond located them *below* the staff. An expicit setting of the `alignAboveContext` [16, p. 780] property helped to reliably tackle this issue.

3.1.7 Output customizing

Customizations can conditionally be present as short fragments of LilyPond code. Some settings are to be activated by including specific variables:

- Hiding voices

  Voices represented by variables (such as `sopran` or `bas`) can be hidden by defining special variables `sopranHide`, `basHide` etc. with a `##t` value that accounts for true in LilyPond.

- Global and local transposition

  Assigning a note to a variable `globalTransposeRelativeC` makes LilyPond transpose the whole score. Similarly, each segment accepts a variable `partTranspose` with similar effect.

These and other modifications such as font, paper, and layout settings are appended as fragments (or so-called *stubs*) of LilyPond code by the client library. For example, the `merge_rests` config adds this excerpt to the resulting code:

```
\layout {
    \context { \Staff \consists "Merge_rests_engraver" }
}
```

3.1.8 Known issues and drawbacks

An inherent drawback of using a custom template in LilyPond is its worse interoperability. Firstly, the template has limited support of score types. In particular, it supports only SATB voicing and lead sheets. Moreover, dividing a song into sections requires manual intervention even when converting from an existing LilyPond source.

Another issue is using the global transposition setting that can produce overly complicated key signatures in some cases. A further examination of the `\transpose` function should make it possible to overcome this issue.

The custom logic for small chorded notes with the example usage in Listing 3.2 has an issue when adding slurs within the `soloMale` variable, which is shown in Figure 3.5.

Lastly, the principle described in Section 3.1.5 doesn't work well with the page breaking mechanism for printed scores. Specifically, it causes the breaking to happen too early, leaving some unused space at the bottom of each sheet. This is probably a LilyPond bug, and the behavior should be further examined or consulted with the community.

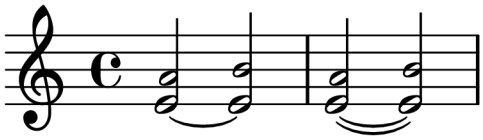

Figure 3.5: Adding a slur (the second measure) inside the `soloMale` variable causes an incorrect slur placement.

3.2 Renderer server and client

Besides the LilyPond template itself, the framework consists of a toolkit for its utilization. In particular, it is a renderer server and a client library. The server is a support layer over LilyPond's CLI with additional software for image postprocessing. The responsibility of the client library is to provide an API for assembling renderable code from fragments of user input and the LilyPond template and for communicating with the server.

The client library is implemented in the PHP language and can also be included as a package in a Laravel project. The server is supposed to be executed as a containerized image with Docker. This was motivated by technologies used at Zpěvník, which are discussed in Section 4.2.

3.2.1 Generating renderable code

One of the client library's functionalities is producing renderable code by inserting user code and LilyPond stubs according to the configuration into the prepared template. The data flow is visualized in Figure 3.6, where the user input is divided into two types of data:

- segments of LilyPond code containing the music content, and
- (an associative array of) layout configuration properties to override default library settings.

These data are passed by creating instances of specific PHP classes defined by the client library. A practical example workflow is shown in Listing C.1. Font files required for successful compilation aren't by default included in the renderable code because they can be built into the server to save redundant data transfer.

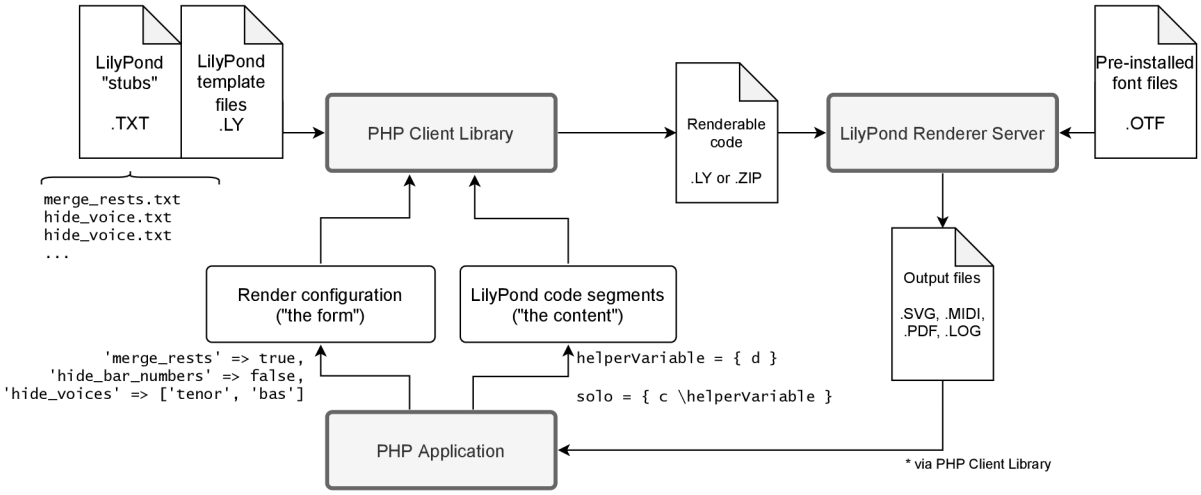

Figure 3.6: Data flow diagram capturing the process of creating and rendering a score. User code with render configuration is passed to the client library, which produces the renderable code. This code is sent to the renderer server, and as a final step, the application retrieves output files.

3.2.2 Client-server communication

The server exposes HTTP-accessible endpoints via the shell2http package that is configured in the container build specification. There are three endpoints available – namely for executing the rendering, retrieving the rendered files, and their subsequent deletion. The client library provides methods to call these endpoints using a GuzzleHTTP library for performing HTTP requests. Should the LilyPond code consist of multiple files, it is transferred in the form of a zipped file, which is internally handled by both the client library and the renderer server.

Currently, the renderer server is not supposed to be exposed as a public API, as it doesn't concern security or performance issues. These relaxed conditions are not critical since the communication can be realized within a virtual network, and a queue mechanism can distribute the workload.

3.2.3 Output rendering and postprocessing

The renderer server includes a Makefile with targets that execute the LilyPond CLI optionally with some postprocessing tasks. All the targets are invokable by the exposed API, so this file may serve as part of the specification. Postprocessing consists of simple commands to optimize or crop the rendered images. SVG postprocessing currently executes the svgo tool to optimize and minify the output file.

3.3 Summary

This chapter described creating a LilyPond template that supports segment-wise representation and producing a uniform output, which is its distinguishing feature. The concatenation functionality and emergent issues are solved within the LilyPond template itself to function independently. The template is included in a PHP client library that produces the final renderable code and manages the execution of LilyPond CLI via a custom renderer server.

4 Web interface

The online project Zpěvník has been continually developed since 2019, and the source code of the ecosystem is published open-source. [24] While the web application and its content are available publicly, it also includes an administration part with restricted access that serves as a Content Management System (CMS) – in this case for songs, interpreters, media, etc.

Extending this CMS with an interface for music notation editing can be considered one of the gradual improvements where I continue on a previous work of the Zpěvník team, including myself. Apart from creating a LilyPond framework discussed in the last chapter, a front-end user interface and back-end support for storing segmented sheet music are brought up in the following text.

4.1 Requirements and design considerations

During the development stages outlined in Section 1.4, the LilyPond template together with the user interface were implemented gradually, as new ideas stemmed from having experience with LilyPond along with user feedback. Also, the emerging requirements can be summarized into the following points:

- attaching the sheet music to an existing song database,
- fast live preview of rendered sheet music,
- generating differently formatted outputs from a single source,
- accessibility for the public frontends,
- bulk (re-)rendering of saved scores,
- downloading the generated LilyPond template, and
- file postprocessing and optimization.

The realization of these features went hand in hand with building a more sophisticated notation interface than just a simple web renderer

for raw LilyPond input, which would only mirror existing solutions. While some of the LilyPond-specific aspects are covered in Chapter 3, other design issues need to be considered concerning the web interface implementation:

4.1.1 Representation of multiple renders

The data model should reflect the principle of rendering multiple outputs from a single source. Moreover, these renders should be distinguishable from each other based on their render-specific configurations. This distinction is essential mainly because of the possible caching of prerendered scores. The idea is to use two separate sets of configuration options – a *score config* that is directly manageable by content editors and applies for all rendered outputs, and *render configs* that are generated semi-automatically. These configurations are supposed to be applied on top of a preconfigured global setup during the actual rendering.

4.1.2 Queued job processing

A queue mechanism needs to be deployed to the stack to effectively deal with rendering and postprocessing tasks while executing synchronous web requests. Each such job can take up orders of seconds, which could cause very long response times if handled synchronously. On top of that, it mitigates the possibility of causing a high load on the renderer server, which concurrent requests or bulk rendering tasks could cause.

4.1.3 Storage for rendered score files

In the prototypes, the SVG data rendered by LilyPond were directly stored as strings within the database rows. This was a suboptimal solution due to a lack of caching mechanisms for dynamic API responses, performance issues when fetching a lot of database models at once, and weak scaling possibilities for generating multiple output files for the same render. Therefore, using up the filesystem with the database containing references to the files instead of their actual content was considered a more optimal solution.

4.2 Technological background

As this part of work is an extension of an existing codebase, I mostly used up technologies that were already part of the deployed stack and I had previous experience with. They are all open-source licensed and can be considered as being part of industry standards.

The back-end is powered by a PHP framework Laravel based on an MVC architecture not unusual among server-side web frameworks. [25] It is accompanied by MySQL database server along with Redis in-memory data storage system for simple operations. Contrary to MySQL, Redis was first deployed within the scope of this work to be a provider for Laravel's built-in queueing mechanism. While there are various supported queue providers, Redis was preferred due to its possible future use as cache storage.

The project makes use of the Docker container platform. This setup causes minimal overhead and provides an isolated configurable environment for each abovementioned piece of software. Moreover, it eases automation of the deployment tasks by defining specific build steps for each service.

The client-side is based on a JavaScript framework for creating interactive user interfaces – Vue.js together with a material-design[1] visual component library Vuetify.

The API is defined via GraphQL specification, which is an alternative to well-known RESTful endpoints. Both back-end and front-end support it by leveraging appropriate third-party libraries.

This stack augmented by custom LilyPond packages from Section 3.2 is involved in the process of handling music notation – editing, rendering, and persisting. Figure 4.1 shows a diagram of the deployment architecture. Currently, all server nodes are separated only virtually by running as separated Docker images, which is managed by the `docker-compose` tool. This architecture allows porting of any service (LilyPond, database, cache) onto a separate cloud-hosted[2] instance should that become necessary out of scaling requirements. This setup has not been tested yet.

1. Visual design system for web and mobile applications developed by Google.
2. Docker hosting is supported by most of the prominent cloud providers.

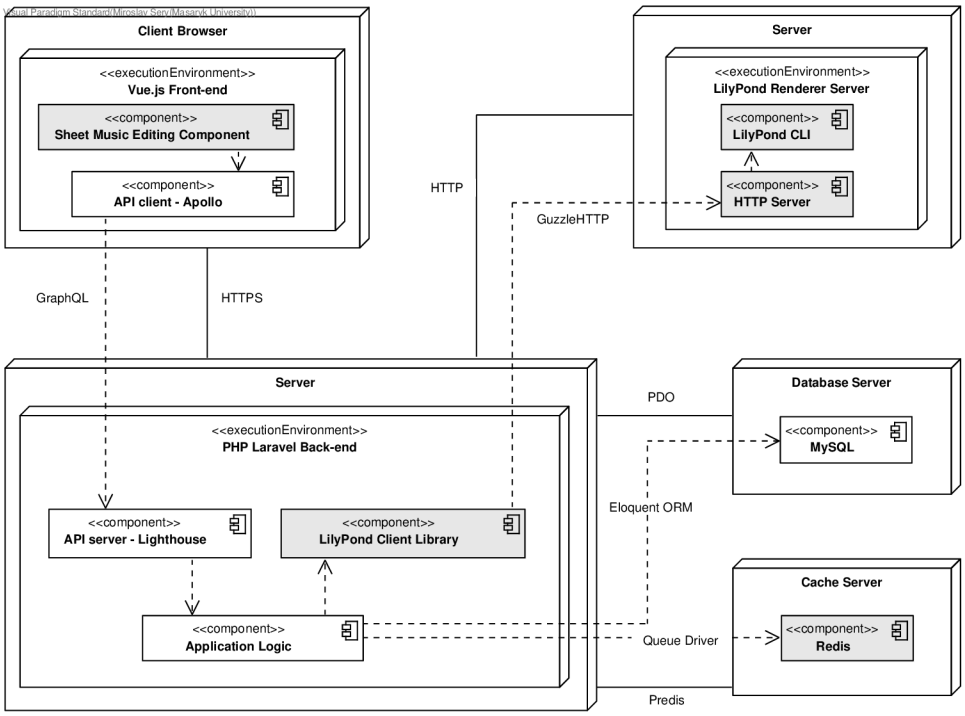

Figure 4.1: Overview of the application's infrastructure. Dashed arrows are to be interpreted as dependency relations, which means the application logic depends on a LilyPond client library along with database and cache providers. The diagram shows only a relevant subpart of the entire Zpěvník infrastructure. Grayed boxes denote components that were newly implemented and deployed within the scope of this thesis.

Concrete examples of event sequences concerning the architecture parts are presented in Figure 4.2. This offers a detailed look at two use-cases: live rendering of sheet music during music notation editing and saving the data. The former (A) starts when an editor interacts with the notation interface, e.g., by typing in the LilyPond input field. A debounce filter is used to merge quick changes into one API request that is sent to the server, which returns the rendered result. The latter (B) extends the older codebase for saving the song's data. It mainly deals with dispatching asynchronous jobs to produce multiple renders of the same score and subsequent saving into the database.

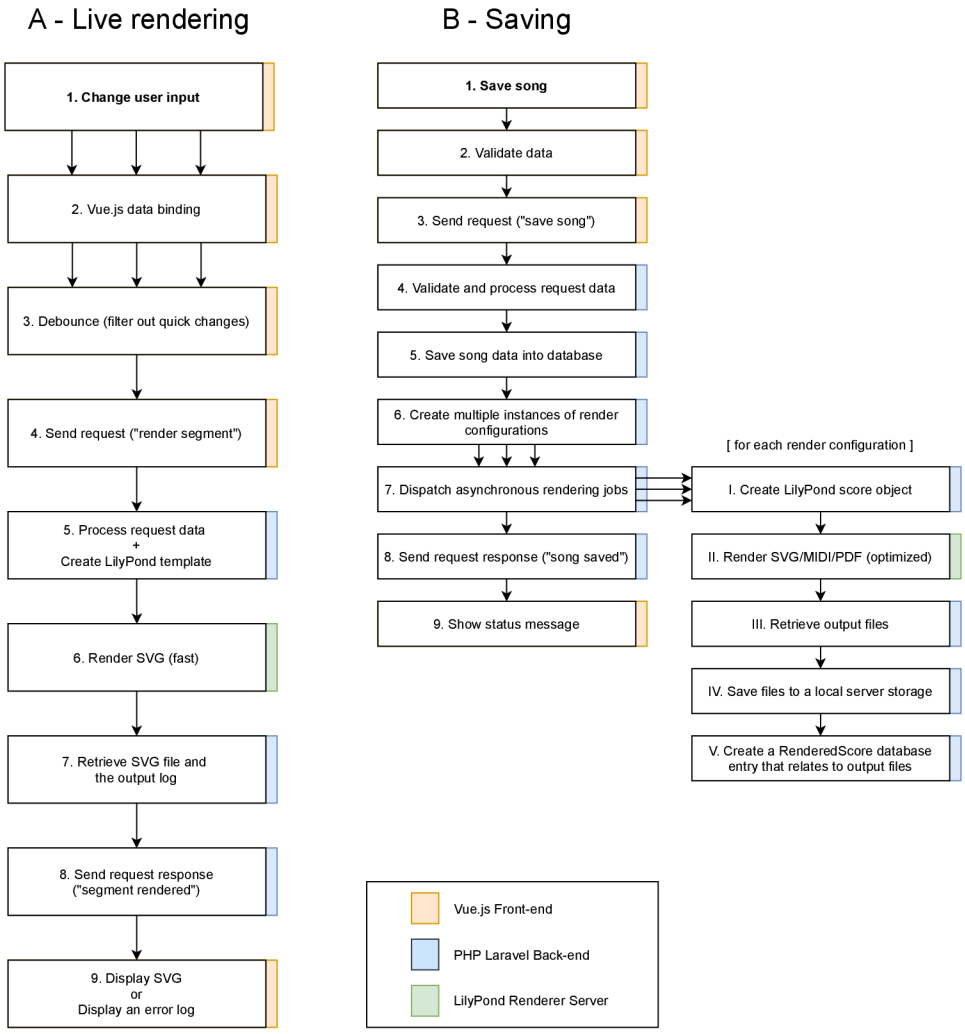

Figure 4.2: These two diagrams present sequences of events happening during live rendering and saving of the content. Execution environments are distinguished by colors.

4.3 Front-end GUI

Figure 4.3 shows a logical layout of sections of a segment-wise notation interface that resulted from previous experience with a simple UI for a uniform LilyPond input. It is specific by working with segments and incorporating form controls that influence particular properties of the generated LilyPond template.

The score-level configuration is supposed to be in the header, whereas the footer should contain a preview of the concatenated score render and export tools. In between, there is a space for the actual notation content. Following text deals with describing these three parts and technical aspects of the interactive functionality.

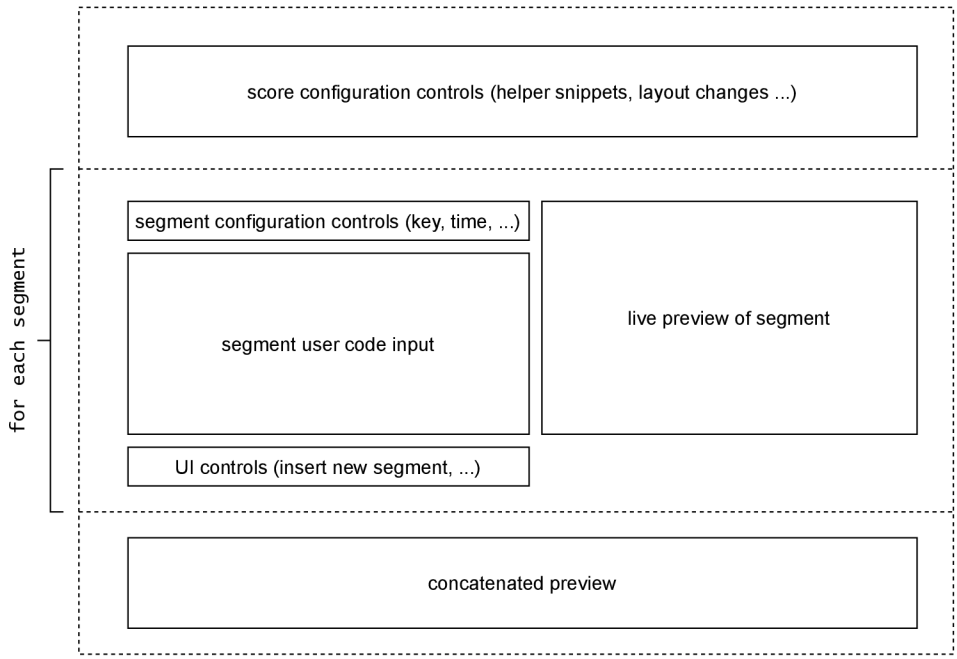

Figure 4.3: Simplified layout wireframe for segmented notation input separated into three sections – the header with score-level configuration, the body consisting of one or more sections with input/preview controls and the footer with a preview of the concatenated score.

4.3.1 Score setup

Score setup includes switches for settings available in the LilyPond template that are reasonable to be applied on the whole score regardless of particular render-specific configurations. This configuration thus contributes to the total score layout, and it is manageable by notation editors.

It comprises three boolean switches that alter the visual output, such as whether pairs of choral voices should be merged onto one staff. This section still serves as somewhat of a conceptual prototype, so new settings are expected to be requested by active users.

Besides the configuration controls, an input for custom LilyPond helper code is present. It is hidden by default as it is supposed to be used by more experienced LilyPond editors.[3]

4.3.2 Segment input

The segment-wise input section contains a couple of components displayed in Figure 4.4. Setting the segment's name in (A) is necessary only for the final concatenation string, while the other controls influence the visual output. During the notation input, it may be useful to render only relevant voices for clarity, which the switch in (E) can do. If the user-code doesn't contain any of the pre-defined voice variable names, a warning is shown. No other deeper check that would require thorough code analysis is performed.

The text input is a simple textarea from the Vuetify library. It overrides the default `tab` key functionality to insert a tab space instead of moving focus to another HTML element, which is the standard browser behavior.

4.3.3 Concatenated preview

At the bottom of the page, there is a section to create and preview the concatenated output. All segments are automatically concatenated according to their order. This default sequence can be overridden by

3. So far, this code is included multiple times in the final code, which may cause unexpected results for inexperienced LilyPond users. A warning with a similar message is shown in the UI.

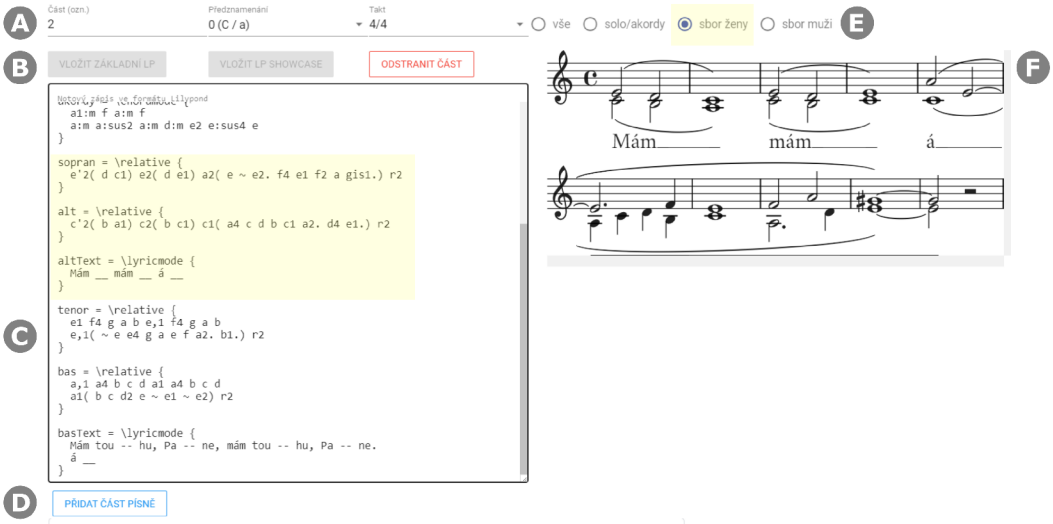

Figure 4.4: Screenshot of the segment input notation. Only a subset of the notation is set-up for preview, which is marked by yellow boxes. A – segment name, time and key settings, B – segment actions, C – LilyPond code input, D – insert new segment action, E – filter visible voices, F – live render output.

a *concatenation string* to rearrange and visually separate the segments. Apart from segment names, the concatenation string may include:

- newlines to enforce line breaking,
- vertical bar symbols (|) to insert a section-dividing bar line, and
- a dot to insert an ending barline.

With this syntax, explicit breaks and usual barline changes can be extracted from the very LilyPond code, which gives rise to more dynamical content in the means of composability. Should the concatenation string input field contain illegal tokens (such as incorrect segment name), a warning is shown to the user.

Furthermore, downloading the renderable LilyPond source code and a rendered PDF export is possible via clickable buttons. The source code can be opened e.g., in the Frescobaldi desktop application to get an advanced notation experience. The PDF export is currently

suboptimal due to an issue in the LilyPond template described in the last paragraph of Section 3.1.8.

4.3.4 Technical aspects

The interactivity of the interface is based on Vue.js data-binding functionality that can be usefully combined with a concept of smart query from the Vue Apollo library. This allows to dynamically query the server and update the data model when any input parameter changes. It is necessary to set up the *debounce* property that enforces a minimal request interval to prevent too frequent server requests, for example, during text typing.

Another characteristic of Vue.js is its component-first architecture, which was applied to implement the notation interface as separate components.

4.4 Server side

The work on the back-end side of the Zpěvník application written in the Laravel PHP framework consisted of extending the database, the internal and public API, incorporating the LilyPond Renderer Client library, and deploying an asynchronous task-processing mechanism. These main contributions are described more in-depth in the rest of this section.

4.4.1 Database and API extending

As introduced earlier, working with the database and defining an API is mediated via helper libraries. In Laravel, database entities are supposed to be mapped to *Model* classes reflecting the MVC pattern style. These Model classes don't need to contain much logic, as the most common operations, such as accessing the properties, are implemented via magic methods in PHP.[4]

4. Laravel aims to be as easy-to-use as possible. However, some of its patterns rather discourage well-established concepts, such as static analysis or SOLID principles. Working with Model classes and their magic methods is one of the cases.

Both Models and the API definitions thus reflect the database structure mirrored by Figure 4.5. Consider that there is no specific database table for notation segments. Instead, they are encoded within a generic JSON[5] type because no relational functionality is required, and new subfields can be added without changing the database schema.

The `external` entity captures the logic of uploaded files on Zpěvník. Some of them are expected to be renderable in the future. Therefore an optional relation to rendered scores is defined. As the naming suggests, "rendered scores" aren't supposed to be bound only to LilyPond.

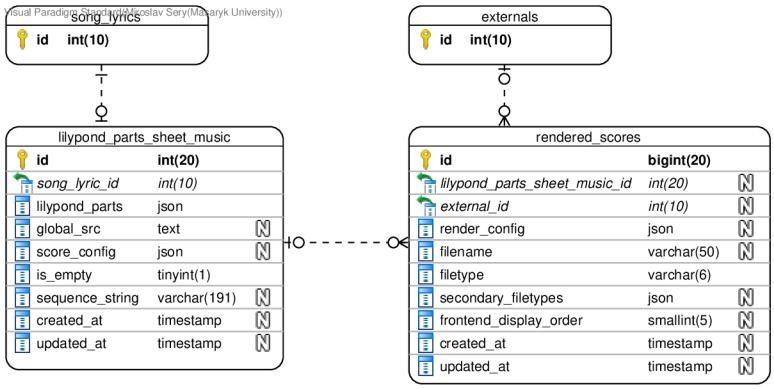

Figure 4.5: Entity-Relation diagram of added database entities with relations to the rest of the database (entities at the top). N encodes a nullable field.

4.4.2 Implementing services

Apart from the basic MVC pattern propagating separation of data-transforming logic into a Controller layer, web frameworks such as Laravel encourage delegating complex responsibilities to specific classes generally referred to as *Services*. These classes can be instantiated via internal framework mechanisms that leverage the *dependency injection* pattern. Generally, these practices are supposed to make the codebase better structured and decoupled from unnecessary instantiation logic.

I implemented four distinct service classes that encapsulate the logic of invoking the LilyPond client library, creating the segmented-

5. MySQL includes native support for the JSON datatype since version 5.7.8. [26]

notation data model, dispatching rendering jobs, and synchronization of rendered outputs with the *rendered scores* database table entries.

4.5 Deployment and testing

The user interface and its functionality have been upgraded incrementally and tested by users, out of whom some are familiar with the LilyPond syntax, while others only have experience with writing sheet music in general. Features and fixes that were implemented based on the feedback include:

- issues with LilyPond segment concatenation,
- concatenation string functionality,
- live rendering bugs, and
- issues with source code export.

The interface is subject to further testing and improvement based on the real-world usage, which already amounts to 250 sheet music documents written in (any of) its versions. Overall, the user reports have been positive and helped shape ideas for further development.

4.6 Current shortcomings

Firstly, this interface is missing some standard features of LilyPond GUIs. These include live playback, the ability to use MIDI devices for notation input, syntax highlighting, code completion, or leveraging the LilyPond's *Point and Click* functionality [18, p. 43] for quick navigation.

The concatenation string syntax doesn't provide an implicit way of dealing with segments that require slight modification to be used in the final score, e.g., sometimes it is necessary to "fill up" a segment ending with an unfinished measure with additional rests. So far, editors can overcome this by creating an extra segment containing sole rests. Nonetheless, this functionality could be incorporated directly into the concatenation string syntax.

Dynamic export possibilities are currently available only for content editors. In the forseeable future, a chosen subset (such as transposition) should be made accessible to public frontends to let visitors create their own customized outputs. For this case, load testing has to be executed to assess the server capabilities.

So far, a single concatenation sequence defines every rendered output. It could be advantageous to define render-specific concatenation sequences, e.g., for printed scores. However, to fully use up the potential of dynamic segmenting, the syntax should be further extended. Should the syntax become too complex, an interesting idea would be to employ the Blockly visual code composing tool.

4.7 Summary

This chapter presented an analysis and principles of realization of a user interface for music-notation editing with segmented input, server-rendered live preview, and export capabilities. It covered specific aspects of extending an existing architecture of the Zpěvník codebase with this interface. Each iteration of upgrades was subject to testing by a closed group of content editors. They provided feedback for further improvements and bug fixes, some of which I already implemented.

5 Conclusion

The goal of this work was to develop a web interface for music notation editing and publishing within an online songbook application Zpěvník based on the open-source GNU LilyPond typesetting software.

I opened up this thesis by introducing LilyPond in the context of standard music typesetting software, along with other approaches to textual notation representation and several libraries for web notation rendering. I also discussed the relations to the Zpěvník web project.

After an overview of basic LilyPond input syntax, I described specific issues that arose while creating a technique for segment-wise music typesetting. This part of the work consisted of extending a built-in LilyPond template for chorals and lead sheets. I incorporated the resulting template into a PHP library that also deals with engraving and postprocessing, which is provided by a custom LilyPond CLI wrapper – a renderer server. Both the PHP library and the renderer server were published as stand-alone repositories.

Further, I dealt with creating a custom web user interface and its integration into Zpěvník's codebase. I continuously deployed several versions of this interface and received feedback from the community of content editors. There are currently rather limited export integrations with the public site and the PDF generation should be further improved, but the interface is adequately functional for internal use.

Nonetheless, I developed a straightforward way to create flexible notation material. This was fulfilled by embracing the concept of music segments along with a means of their concatenation, which I consider to be a key feature of my work distinguishable from common notation GUIs. This approach lays the basis for a sophisticated public interface for assembling custom sheet music regardless of experience with the underlying notation software.

Bibliography

1. NIENHUYS, Han-Wen; NIEUWENHUIZEN, Jan. LilyPond, a system for automated music engraving. In: *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*. 2003, vol. 1, pp. 167–171.
2. *About MuseScore* [online] [visited on 2021-02-16]. Available from: `https://musescore.com/about`.
3. MANDELSTAM, David. *Unlimited score uploads for everyone – we've removed the 5-scores upload limit* [online]. 2020-02-19 [visited on 2021-02-16]. Available from: `https://musescore.com/groups/improving-musescore-com/discuss/5058097`.
4. *What is MusicXML* [online] [visited on 2021-02-16]. Available from: `http://www.musicxml.com`.
5. ELONA, Shatri; GYÖRGY, Fazekas. *Optical Music Recognition: State of the Art and Major Challenges*. 2020. Available from arXiv: `2006.07885 [cs.CV]`.
6. DOBRIAN, Chris. *Music Programming* [online]. 1988 [visited on 2021-04-25]. Available from: `https://music.arts.uci.edu/dobrian/CD.MusicProgramming.htm`.
7. *What is MEI* [online] [visited on 2021-02-16]. Available from: `https://music-encoding.org/about/`.
8. PUGIN, Laurent. Interaction with Music Encoding. In: *„Ei, dem alten Herrn zoll' ich Achtung gern'"*. [N.d.], pp. 617–630.
9. *V8 Javascript Engine* [online] [visited on 2021-02-16]. Available from: `https://v8.dev`.
10. TAUPIN, Daniel; MITCHELL, Ross; EGLER, Andreas. MusiXTEX: Using TEX to Write Polyphonic or Instrumental Music, Version T.114. 2006, p. 10. Available also from: `http://icking-music-archive.org/software/musixtex/mxdoc110d.pdf`.
11. HUFFLEN, Jean-Michel. Requirements for a Music Engraving Program: a Composer's Point of View. 2017. Available also from: `https://guitex.org/home/images/ArsTeXnica/AT024/hufflen-venice.pdf`.

12. NIENHUYS, Han-Wen; NIEUWENHUIZEN, Jan. *Preface* [online]. 2002-07 [visited on 2021-02-12]. Available from: http://lilypond.org/doc/v2.12/Documentation/user/lilypond-learning/Preface.

13. LILYPOND DEVELOPMENT TEAM. LilyPond - Essay on automated music engraving [online]. [N.d.] [visited on 2021-02-15]. Available from: https://lilypond.org/doc/v2.22/Documentation/essay.pdf.

14. THE LILYPOND DEVELOPMENT TEAM. *Extending* [online] [visited on 2021-03-30]. Available from: https://lilypond.org/doc/v2.22/Documentation/extending.pdf.

15. LILYPOND DEVELOPMENT TEAM. *LilyPond – Music notation for everyone* [online] [visited on 2021-02-11]. Available from: http://lilypond.org/doc/v2.22/Documentation/web/index.html.

16. THE LILYPOND DEVELOPMENT TEAM. *Notation reference* [online] [visited on 2021-03-22]. Available from: http://lilypond.org/doc/v2.22/Documentation/notation.pdf.

17. THE LILYPOND DEVELOPMENT TEAM. *Learning manual* [online] [visited on 2021-03-22]. Available from: https://lilypond.org/doc/v2.22/Documentation/learning.pdf.

18. THE LILYPOND DEVELOPMENT TEAM. *Usage* [online] [visited on 2021-03-30]. Available from: https://lilypond.org/doc/v2.22/Documentation/usage.pdf.

19. BOWLING, Daniel; SUNDARARAJAN, Janani; HAN, Shui'Er; PURVES, Dale. Expression of Emotion in Eastern and Western Music Mirrors Vocalization. *PloS one*. 2012, vol. 7, e31942. Available from DOI: 10.1371/journal.pone.0031942.

20. *HTML SVG Graphics* [online] [visited on 2021-05-04]. Available from: https://www.w3schools.com/Html/html5_svg.asp.

21. *Web MIDI API* [online]. The World Wide Web Consortium (W3C) [visited on 2021-05-14]. Available from: https://www.w3.org/TR/webmidi/.

22. PERON, Fr. Jacques; LISKA, Urs; SPRINGUEL, Br. Samuel. *lyLuaTEX* [online]. 2019-05-27 [visited on 2021-02-11]. Available from: http://ftp.cvut.cz/tex-archive/support/lyluatex/lyluatex.pdf.

23. BAČA, Trevor. *Abjad documentation: LilyPond context concatenation* [online] [visited on 2021-03-22]. Available from: https://abjad.github.io/appendices/lilypond-context-concatenation.html.

24. THE PROSCHOLY.CZ DEVELOPMENT TEAM. *ProScholy.cz GitHub* [online] [visited on 2021-05-14]. Available from: https://github.com/proscholy.

25. *Server-side web frameworks* [online]. MDN Web Docs [visited on 2021-05-19]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

26. THE MYSQL DEVELOPMENT TEAM. *Changes in MySQL 5.7.8* [online] [visited on 2021-05-17]. Available from: https://dev.mysql.com/doc/relnotes/mysql/5.7/en/news-5-7-8.html.

A Content of the attachment

The attachment contains a PDF user manual for content editors and three folders with source code. The folders `lp-client` and `lp-server` contain the functionality described in Chapter 3 and each can be executed stand-alone. The folder `zpevnik-added` contains *only* files that have been added to the existing Zpěvník's codebase and that refer to the content of Chapter 4.

All of this software is published open-source on GitHub under the AGPL 3.0 license. The attachment code state corresponds to a version fixed under git tag `thesis` for each repository. This tagged version is also referenced within this thesis, wherever a link is followed by symbol.

The AGPL license was selected because GNU LilyPond is released under the GPL 3.0, which enforces its use on derived projects and the AGPL variant is commonly used for code running server-side.

B Zpěvník's usage statistics

Figure B.1: Google Analytics export for the Zpěvník project – most users use the application from mobile devices. Results show an aggregated monthly sum of visits from both the website and an accompanying Android/iOS application.

C PHP Client library example workflow

```
<?php

use ProScholy\LilypondRenderer\Client;
use ProScholy\LilypondRenderer\LilypondPartsTemplate;
use ProScholy\LilypondRenderer\LilypondPartsRenderConfig;

// 1. instantiate a client with the address
// of a running server instance
$client = new Client('localhost', '3100');

// 2. create a render layout configuration
$config = new LilypondPartsRenderConfig([
    'merge_rests' => true,
    'hide_bar_numbers' => false,
    'hide_voices' => ['tenor', 'bas']
]);

// 3. instantiate a templated LilyPond source
$src = new LilypondPartsTemplate('helperVariable = { d }',
   $config)
    ->withPart('chorus', 'solo = { c \helperVariable }');

// 4. do the rendering process and retrieve the output file
$res = $client->renderSvg($src);
$svg = $client->getResultOutputFile($res);
```

Listing C.1: Practical example of working with the PHP client library that provides a convenient way of utilizing the LilyPond template.