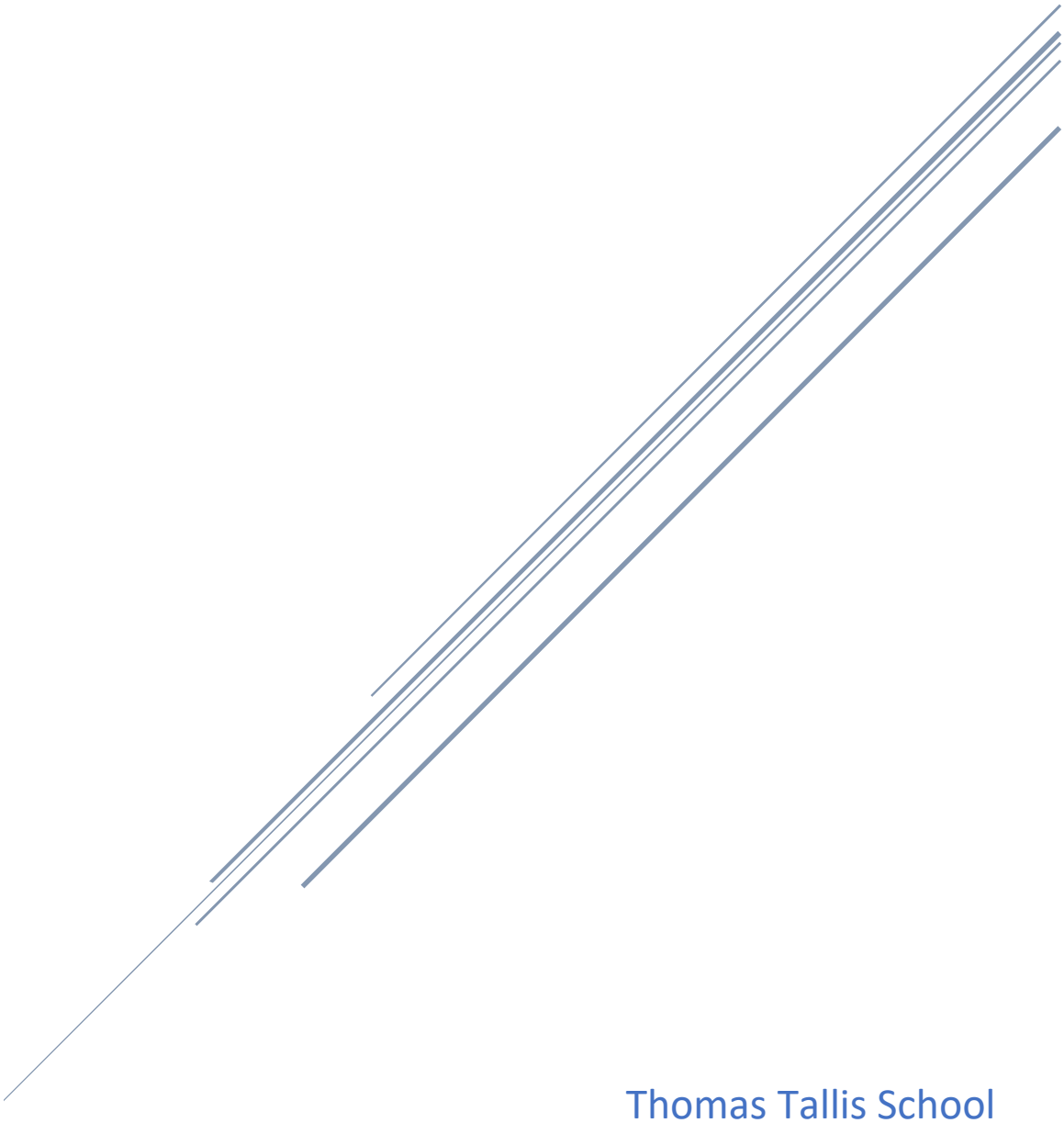
A LEVEL PROGRAMMING PROJECT

Music Composing Website - ScoreLily



Thomas Tallis School
Beauty Oluokun

Table of Contents

Design

Decomposition of Problem

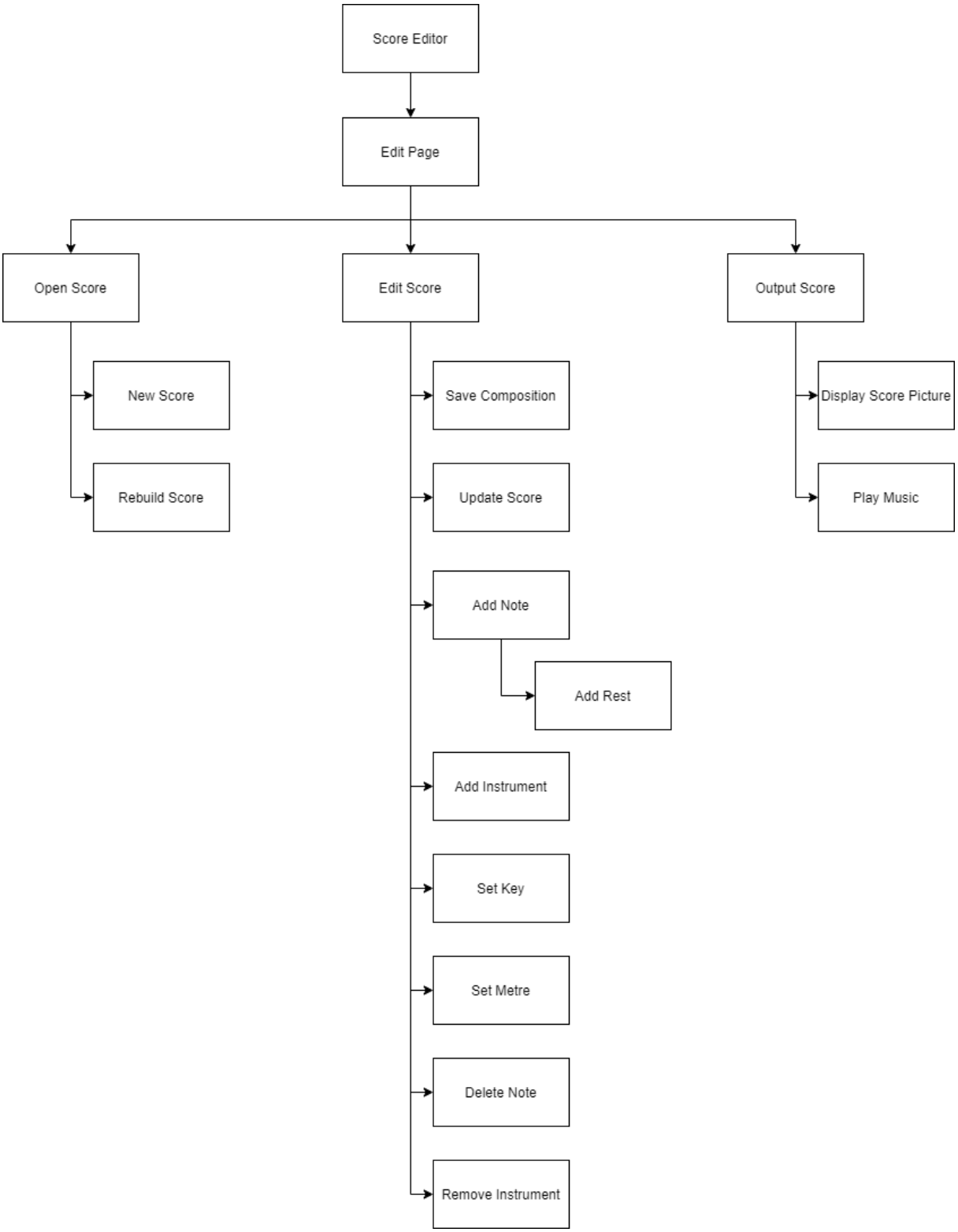


Fig. 11 – Score Editor top-down decomposition

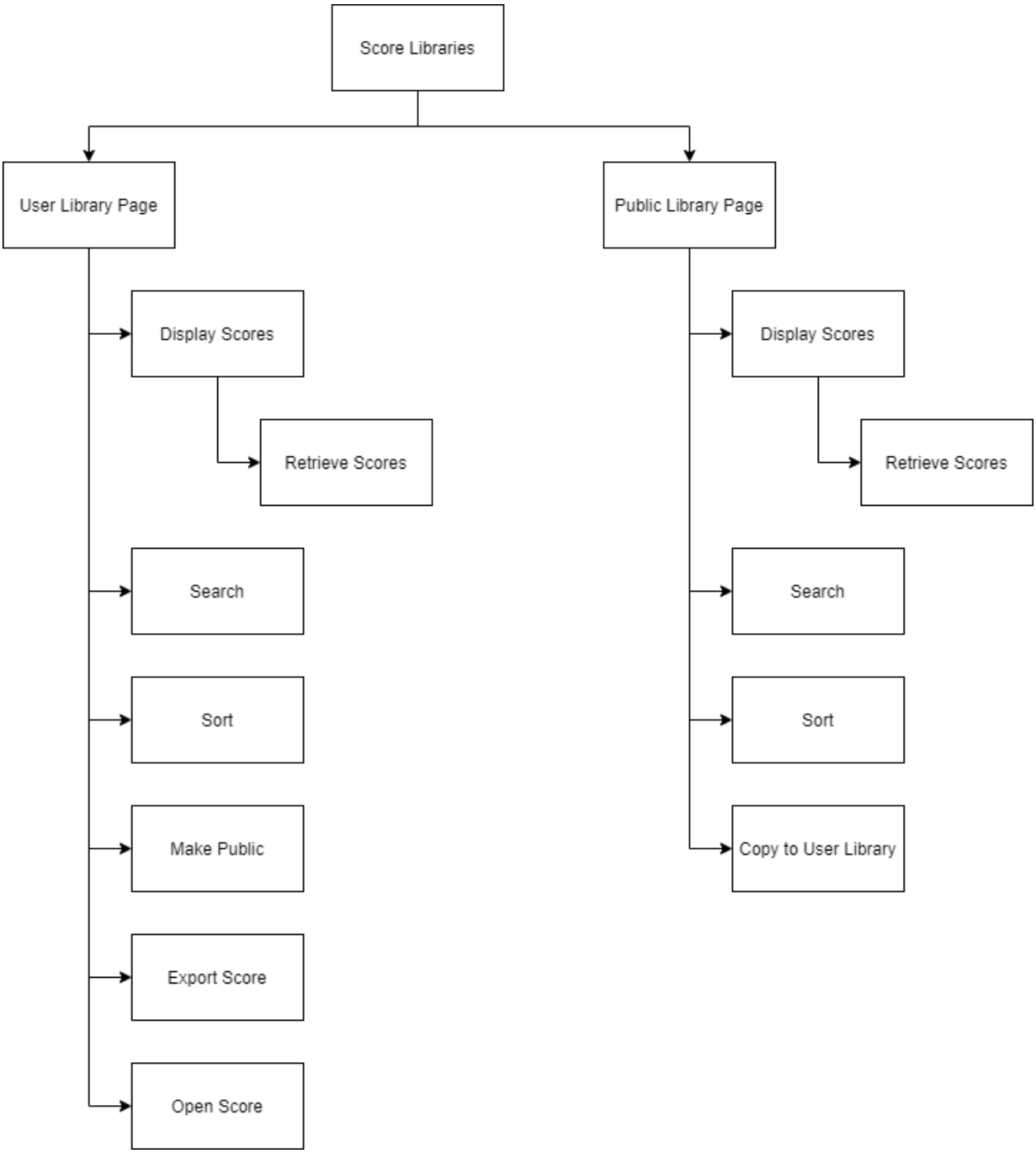


Fig. 12 – Score Libraries top-down decomposition

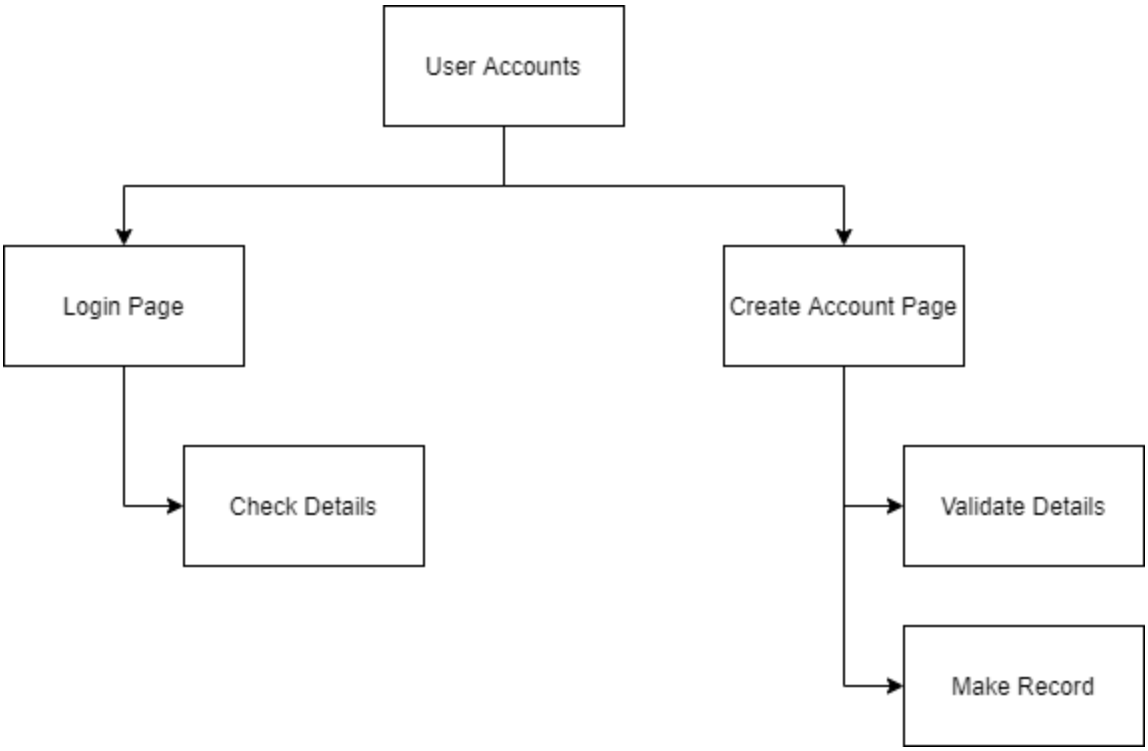


Fig. 13 – User Accounts top-down decomposition

The system is made of 3 main modules: Score Editor, User Account and Score Libraries. The score editor module handles the creation and modification of the music scores. The user accounts module handles how user accounts are created and how the user logins in and out of the system. The score libraries module handles how the scores are retrieved and displayed on the public library and user library pages.

Score Editor

The Score Editor module only contains one webpage, the 'Edit' webpage. That webpage has 3 main purposes: opening, editing and outputting the music score.

Open Score:

This module handles opening of new scores or previously made scores; it contains 2 sub-modules:

- **New score =** A new score record is made in the database and a new instance of Mingus' Composition object is created.
- **Rebuild score =** The desired score record is retrieved from the database and saved data is used to create an instance of Mingus' Composition that contains the needed instruments, notes etc.

Edit Score:

This module handles how the Composition object holding the musical data is edited and how the lilypond, MIDI, text and PDF files associated with the score are updated. It contains 8 sub-modules:

- **Save composition =** The contents of the Composition object used– the Tracks, Bar metres and keys and notes - are saved in a text file in a format that can easily read and converted back into a Composition object.
- **Update score =** If there are changes made to the music score, the music represented in the Composition object is converted into the lilypond format and written to the score's lilypond file. The data in the Composition object is also converted to the MIDI format and written to the score's MIDI file. The lilypond file is used to update the score's PDF file.
- **Add Note & Add rest =** Places a note, of a specified name and octave, or a rest of a specified duration in a chosen bar, at a chosen position within a chosen instrument's track.
- **Add instrument =** Adds a Track object to the Composition object with a specified instrument name.
- **Set key =** Changes the key of the Bar objects in every Track object in the Composition object from a specified bar number.
- **Set metre =** Changes the metre of the Bar objects in every Track object in the Composition object from a specified bar number.
- **Delete Note =** Removes a specified note from a chosen Bar object in a chosen Track object within the Composition object without affecting the rest of the bar.
- **Remove instrument =** Removes a specified Track object from the Composition object.

Output Score:

This module handles how the music score is presented to the user through visuals and audio. It has 2 sub-modules:

- **Display score picture =** Takes the score's updated PDF file and embeds it in the Edit page.
- **Play music =** Takes the score's updated MIDI file and embeds it in the Edit page – allowing the user to control the playback of the audio.

Score Libraries

The Score Libraries module contains two webpages: 'User Library' and 'Public Library'. These two pages have very similar purposes but the main difference between them is that 'Public Library' is the same for all users and 'User Library' is different for each user. The two webpages have four submodules in common: 'Display Scores', 'Retrieve Scores', 'Search' and 'Sort'.

Display Scores handles displaying the list of score files in each library in a table. The table should have sections like title, author, last modified etc. Display Scores has one submodule called **Retrieve Scores**. Retrieves scores goes through all the score records in the database and returns all records that meet a certain criterion – for 'Public Library' the score must be set as public and for 'User Library' the score must be linked to specific user.

Search handles how the search bar on each webpage filters through the scores displayed to execute the user's search request. It should do this by comparing the titles of scores to the text entered.

Sort handles changing the order that the scores are displayed in. The user can choose different ways to order the scores and then a sorting algorithm is used to sort the scores based on the criterion given.

The two webpages both have submodules specific to themselves:

Public Library:

- **Copy to User Library =** A score in the public library that was not made by the user is added to the user's score library so they can access it quickly.

User Library:

- **Make Public =** Flags a score, made by the user, in their library as public so it can be displayed on the 'Public Library' webpage.
- **Export Score =** If the user wants the export the music file, it downloads the chosen score's MIDI file to the user's computer. If the user wants to export the pdf file, it the downloads the chosen score's PDF file to the user's computer.
- **Open Score =** Sends the data needed to Score Editor's Open Score module so that the score can be opened on the 'Edit' webpage.

User Accounts

The User Accounts module contains two webpages: 'Login' and 'Create Account'.

Create Account:

A new user would use this webpage to create an account, this then creates a new user record allowing the user to access the rest of the website features. This webpage contains 2 submodules:

- **Validate Details =** Checks that the username, email and password entered are is a valid format to be processed. Also checks the strength of the password given criteria. If the details given pass the validation checks, an account can be made and if not, a message is sent back to the user to try again.

- **Make Record =** A new user record is made using the validated username, email and password details. Then the user is redirected to their 'User Library' page.

Login:

A returning user would use this webpage to log back into their previously made account so they can access the other features of the website and their scores. It only contains one submodule:

- **Check Details** = The username entered is checked against the username fields of the records in the database. If a record is found with that username, the password entered is compared with that record's password field to see if they are equal. If they are, the user is redirected to the 'User Library' webpage and if not, the user is sent a message that the password entered was incorrect. If a record wasn't found with a matching username, the user is sent a message that the username entered doesn't exist.

Interface Design

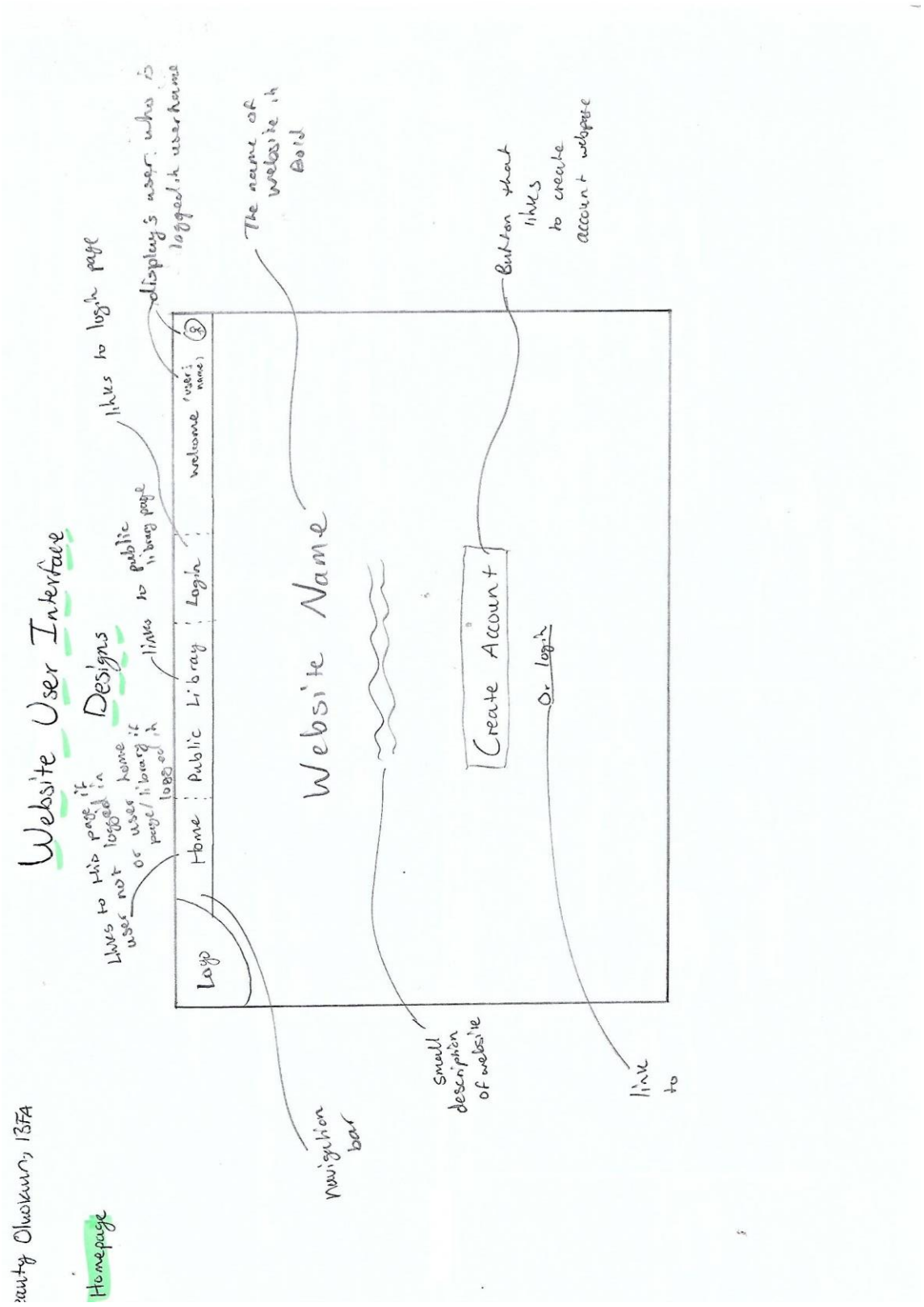


Fig. 14 – Sketch of 'Home' webpage

Fig. 15 – Sketches of 'Create Account' and 'Login' webpages

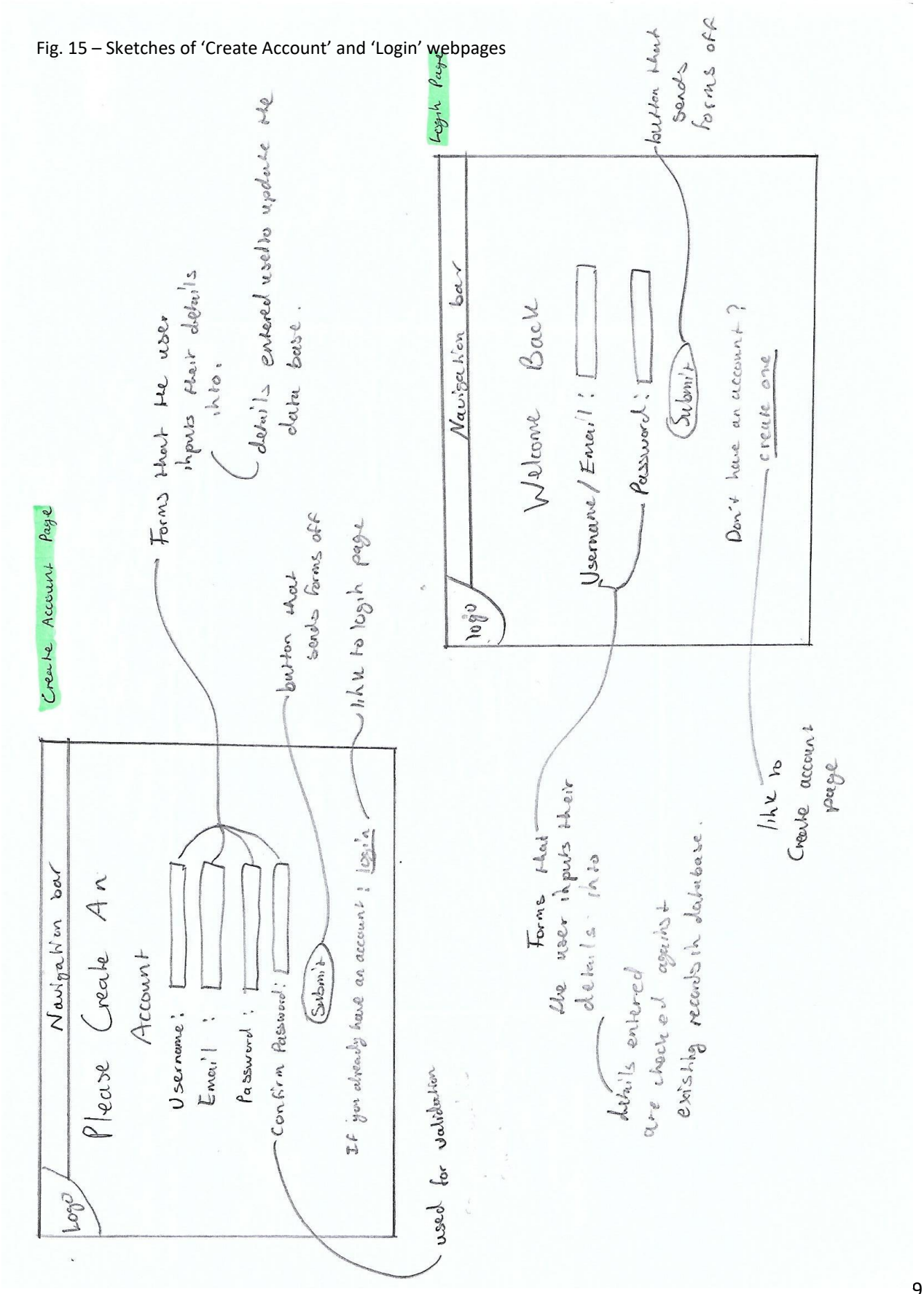
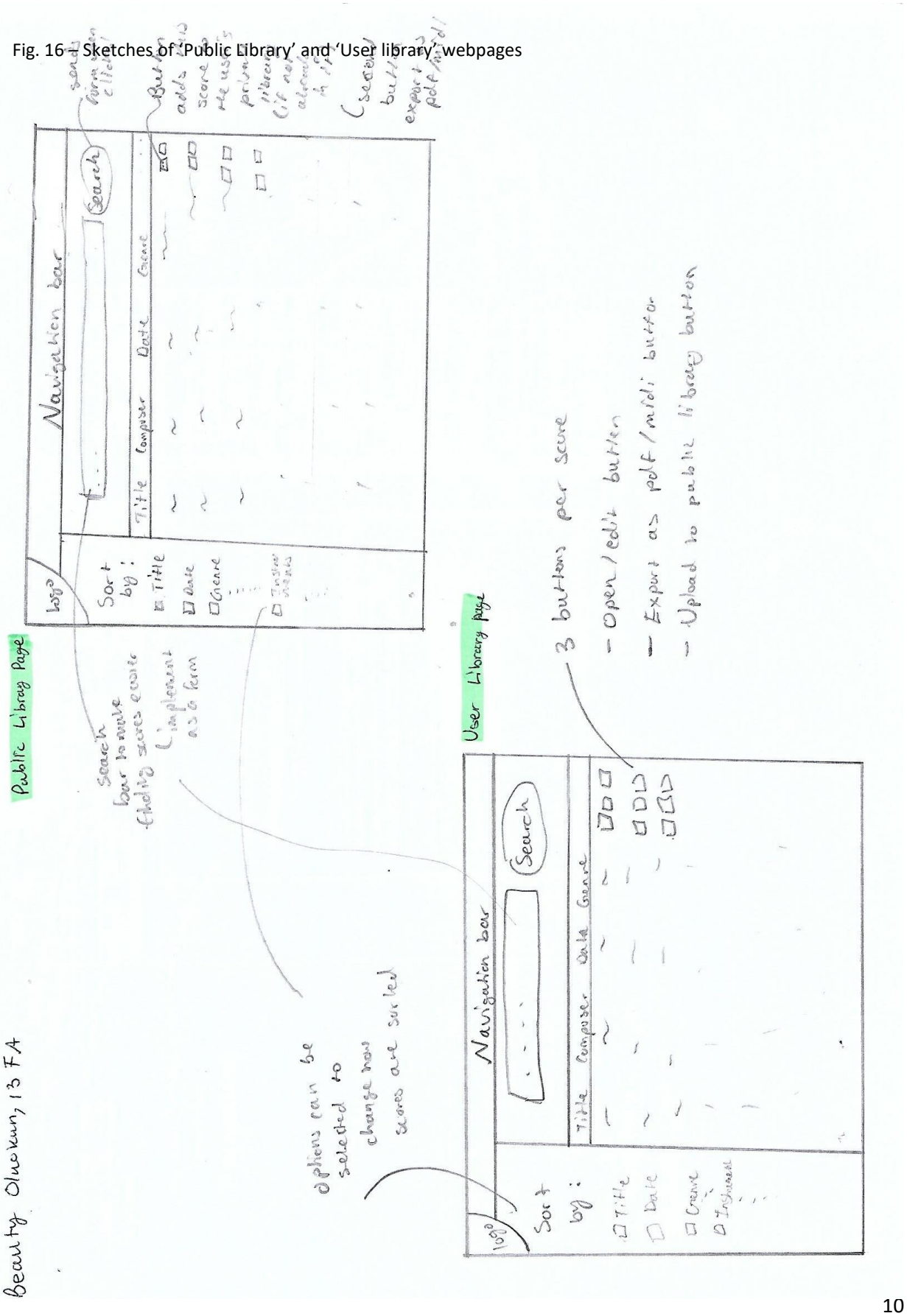


Fig. 16 – Sketches of 'Public Library' and 'User library' webpages



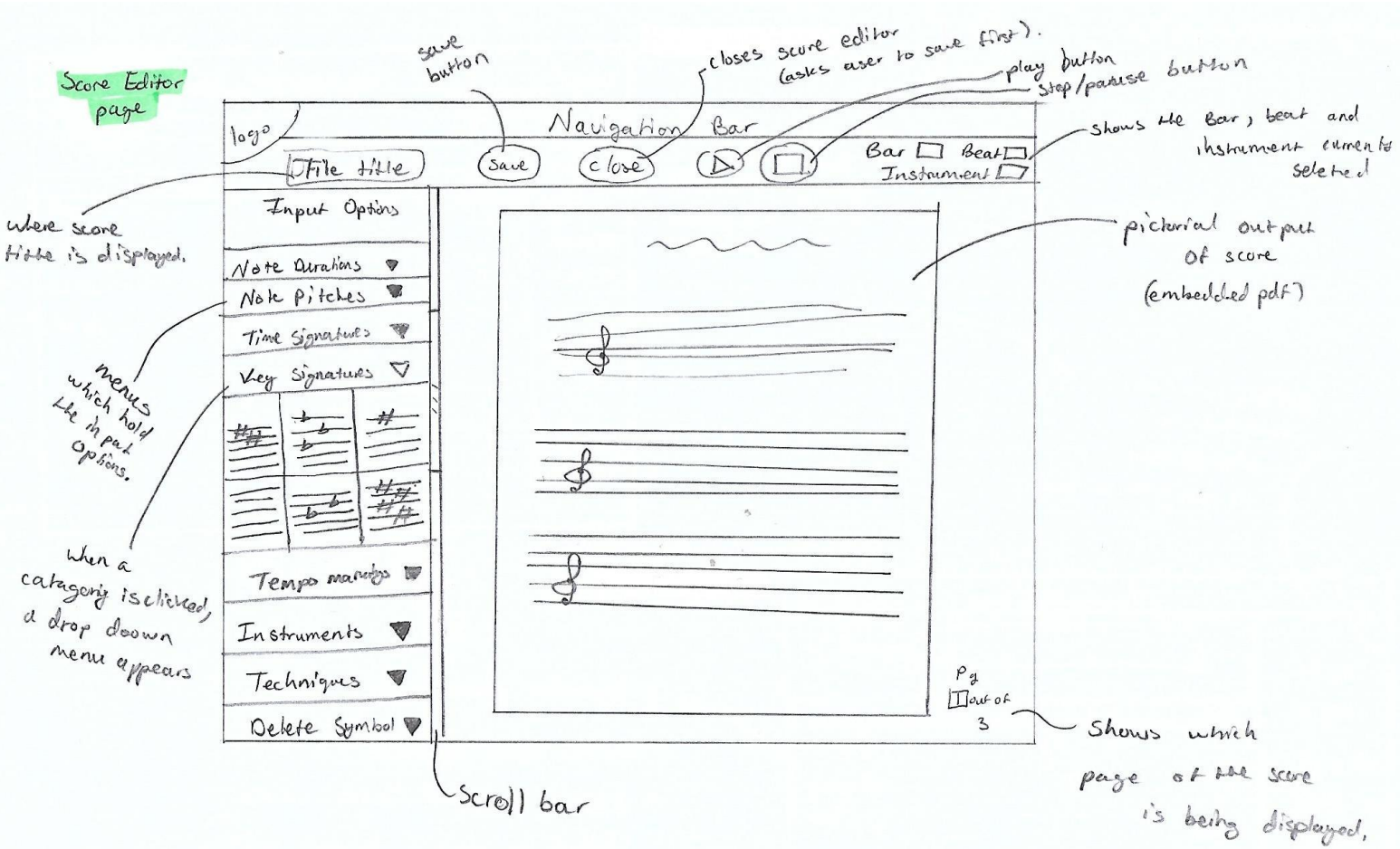


Fig. 17 – Sketch of 'Editor' webpage

Name of Webpage	Purpose	Widgets, Events or Procedures Used
Home	First page a user meets when they go to the website. It briefly explains what the website is and directs the user to either make an account or login if they already have one.	• Button that acts as a link to the 'Create Account' webpage. • A text link to the 'Login' webpage.
Create Account	Allows the user to enter in the username, email and password they want to make an account. It verifies and validates the data entered and then sends this data to the backend to be processed.	• A form with four fields and a submit button. Data sent via POST method. • A text link to the 'Login' webpage.
Login	Allows a returning user to enter their username and password so they can log back into their account. If the details entered match a record in the database, then the user is redirected to the 'User Library' webpage. If not, they stay on the 'Login' webpage and are given an error message.	• A form with 2 fields and a submit button. Data sent via POST method. • A text link to the 'Create Account' webpage.
Editor	Opens a selected score by accessing its record from the database and allowing the user to modify this score. The user can see the visual representation of the score. The user can play, pause and rewind the aural representation of the score. They use menus to select how they want to edit the score.	• An embedded PDF which acts as the visual representation of the score. • An embedded MIDI file which acts as the aural representation of the score. • Button that calls 'save composition' procedure when clicked. • Button that acts as a link to the 'User Library' webpage. • A menu of dropdown forms: a list of buttons that each reveal a form when clicked. • A form per dropdown button, each having several fields and a submit button. All data sent via POST method.
User Library	Acts as the homepage when the user is logged in. It displays all the user's scores in a table-like format: for each score the title, author, date it was last	• A search widget. • A form with a radio field and a submit button. Data sent via POST method.

	modified etc. is shown. The user can select a score to either edit, export or upload to the public library. There is a search bar to make it easier to look for scores. There are also options to change which order the scores are displayed in.	• For each score row, a button that acts a link to the 'Editor' webpage. • For each score row, a button that when clicked calls the 'export' procedure. • For each score row, a button that when clicked calls the 'upload to public' procedure.
Public Library	Has a similar format to the 'User Library' webpage. It displays all the scores that have been uploaded to the public library from different users in a table-like structure. Like the 'User Library' webpage, there is a search bar and options to change the way the scores are ordered. The user can select a score to either add to their private library or export.	• A search widget. • A form with a radio field and a submit button. Data sent via POST method. • For each score row, a button that when clicked calls the 'export' procedure. • For each score row, a button that when clicked calls the 'upload to private' procedure.

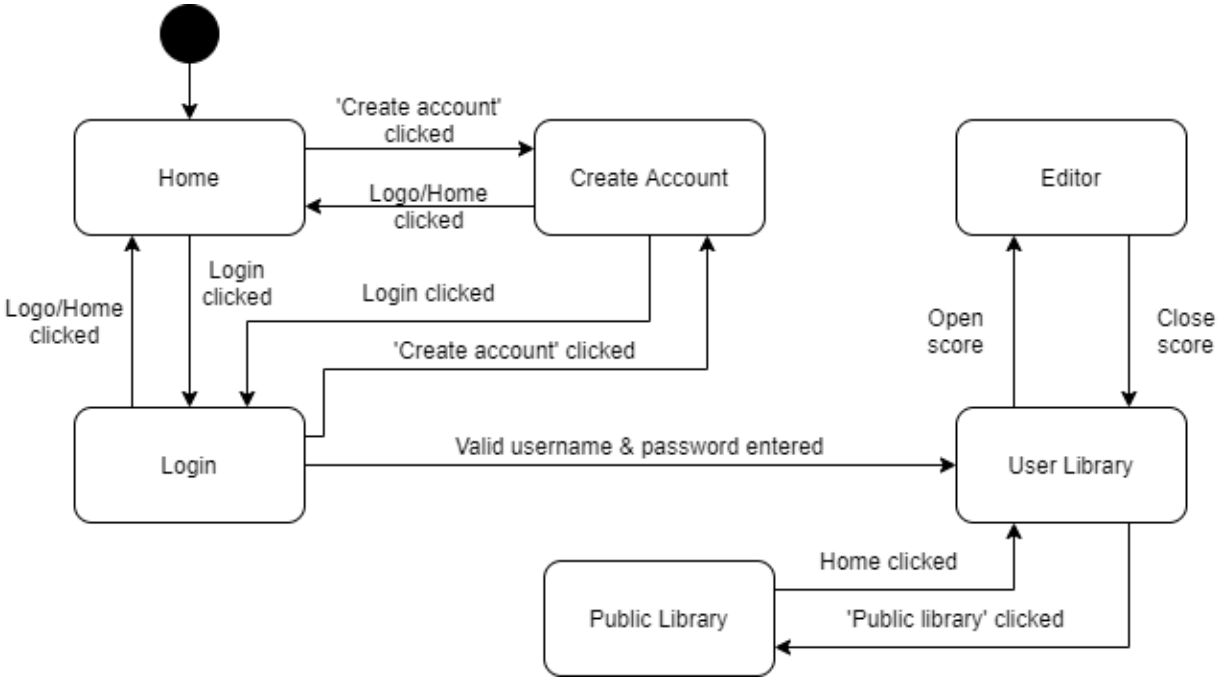


Fig. 18 – State model diagram for GUI

Justifying Usability Features

The static top navigation bar found on all webpages allows the user to always be able to turn to a different, familiar webpage when done with the one they're currently on. Without this feature, the user could easily feel trapped on certain webpages like the 'editor' page where there is not a clear 'back' button. Also, the fact that the same navigation bar is found on every page brings a sense of uniformity to the website, making it seem more professional.

If the user is on the 'home', 'create account' or 'login' page, there are text and links guiding them to either login or create an account allowing them to understand that you need an account to interact with the features of the website. This makes the website more usable as it makes it very apparent to a new user the account system of website – they won't waste time clicking on random links trying to create an account.

Styling

The font used throughout the website should be Bahnschrift. I have chosen this font as it is sans-serif and it is easier to read text in a san-serif font on a computer screen than a serif font. I decided against a monospace font as they can take up too much space on the computer screen. If the Bahnschrift font is not available on the user's browser the Verdana font should be used and if that is also not available, the Arial font should be used.

Fig. 19 – Chosen fonts

Each webpage will use the same colour scheme. The main background will be light blue (#b3e6ff) and the navigation bar and vertical side menus will have a darker blue background (#33bbff). The colour of buttons will either a pinky-coral (#ff704d) or a more orangey-coral (#ff884d). I have chosen to use shades of blue because it is the most liked colour in the world and is unisex, so it is likely to appeal to most of my target audience. Blue is also seen as a calm, relaxing colour so using blue emphasizes that writing music can be relaxing. Furthermore, blue is associated with intelligence and wisdom so using blue highlights that website can also help young people to learn more about music. I have chosen to use shades of coral because I feel it complements the blue without standing out too much onscreen and it matches the colour of the logo. The colours I've chosen are not dull, so they appeal to a young audience, but they are not childish. Only the light blue background will have black text on it, the other colours will have white text.

Fig. 20 – Chosen colours

Fig. 21 – Website logo

I have created a logo for my website using Microsoft PowerPoint. I got the image of the bass and treble clef heart online (website: https://www.teepublic.com/t-shirt/972031-treble-clef-and-bass-clef-heart). I also created 3 versions of backgrounds inspired by the logo and color scheme that could be used in some of the webpages (e.g. public library page).



ree versions of logo-inspired background images

Data Design

Database

My website uses a very simple relational database consisting of 2 tables: one representing the Score entity and the other representing the User entity. To create the database, Django models will be used. Each entity will be coded in Python as a class that inherits from Django's Model class. Each attribute of the class will correspond to field in that entity's table; the attributes themselves are instances of Django's Field classes. Once the models are migrated into the database, Django handles the SQL behind the transactions involving the database while I interact with records within the database as Python objects.

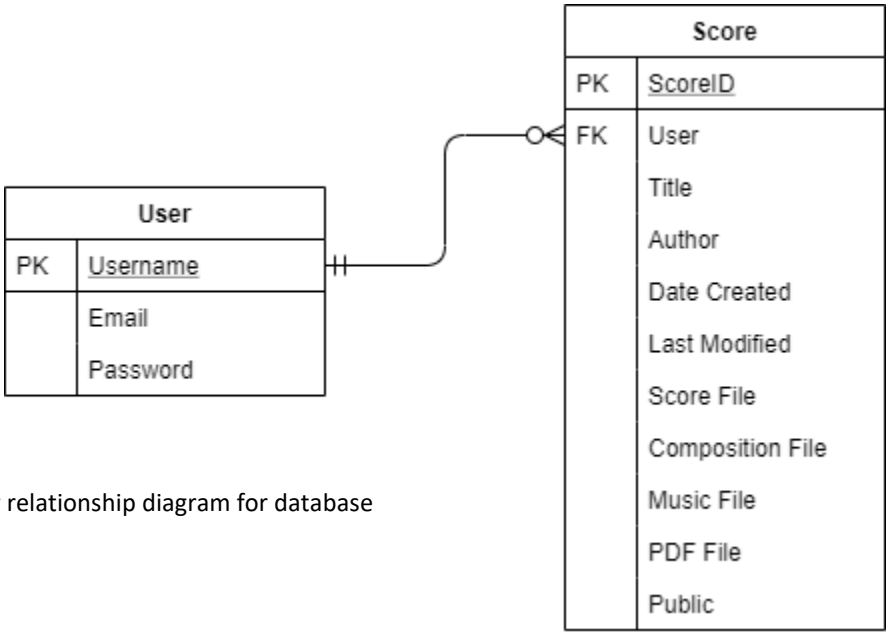


Fig. 23 – Entity relationship diagram for database

The User and Score entities have a one-to-many relationship as each user can have multiple scores associated with them while each score is made by one user. The tables are linked directly by 'User' field in the Score table that acts a foreign key. This relational database is in second normal form because it is already in first normal form (neither table contains a repeated attribute or groups of attributes) and neither table contains any partial dependencies. However, it is not in third normal form as the Score table's 'Author' field is dependent on the foreign key 'User' not the primary key 'ScoreID'. I have decided to leave the database in second normal form because the records are processed as objects in the Python functions and it easier

to use the ‘Author’ field to access the name of the user that score belongs to than the ‘User’ field. This is because the ‘User’ attribute is stored as the address to the corresponding User object not the primary key, ‘username’, of the User table.

User Class:

- **Username =** CharField object. This field holds the usernames of all the users; it acts as this table’s primary key.
- **Email =** EmailField object. This field holds the email addresses of all the users; each record should have a unique email address.
- **Password =** CharField object. This field holds the passwords of all the users; this field doesn’t have to be unique.

Score Class:

- **ScoreID =** AutoField object. This field holds the identification numbers of all the scores. It should be automatically generated when a new record is made, and it acts as this table’s primary key.
- **Title =** CharField object. This field holds the names of all the scores; this field doesn’t have to be unique.
- **Author =** CharField object. This field holds the name (username) of the User record linked to each score; this field doesn’t need to be unique.
- **Date Created =** DateTimeField object. This field holds the date and time that the score record was created. It doesn’t have to be unique, but it should be non-editable.
- **Last Modified =** DateTimeField object. This field holds the date and time that the score record files were last changed; this field doesn’t need to be unique.
- **Score File =** FileField object. This field holds the lilypond files for each score; this field must be unique.
- **Composition File =** FileField object. This field holds the text files used to save the Composition object information for each score; this field must be unique.
- **Music File =** FileField object. This field holds the MIDI files for each score; this field must be unique.
- **PDF File =** FileField object. This field holds the PDF files used as the pictorial representation for each score; this field must be unique.
- **User =** ForeignKey object. This field acts as the foreign key linking each score record to their User; this field doesn’t have to be unique.
- **Public =** BooleanField object. This field acts as the flag for whether the score should be added to the public library. Its default value is false.

Key Variables & Data Structures

In the Score Editor Module, many of the data structures I'll be using are classes from the Python Mingus library. These include the classes Composition, Track, Bar, Key, Note and MidiInstrument.

Module	Sub-Module/Function	Name	Data Type	Explanation
Score Editor	Open Score	score_id	Integer	Holds the ScoreID of the score record that is needing to be retrieved at a given time.
Score Editor	Open Score	username	String	Holds the Username of the User record the score record should be linked to.
Score Editor	Open Score	instrument_list	Array of strings	Holds all the available instrument names the user can chose from.
Score Editor	Open Score	metre_list	Array of strings	Holds all the available metres the user can choose from.
Score Editor	Open Score	key_list	Array of strings	Holds all the available keys the user can chose from.
Score Editor	Open Score	duration_list	Array of strings	Holds all the available note durations the user can chose from.
Score Editor	Open Score	note_list	Array of strings	Holds all the available note names the user can chose from.
Score Editor	Open Score	user	User Object	Allows the specified user record to be accessed through the User object.
Score Editor	Open Score, Edit Score, Output Score	score	Score Object	Allows the specified score record to be accessed through the Score object.
Score Editor	Open Score, Edit Score	composition	Composition Object	Hold all the music data for the score in a format that can be easily manipulated.
Score Editor	Open Score	title	String	Holds the title for a new score to be made.
Score Editor	Edit Score	metre	Tuple of integers	Holds the chosen metre in a valid form to be passed into the set_metre function.

Score Editor	Edit Score	key_sig	String	Holds the chosen key signature in a valid form to be passed into the set_key function.
Score Editor	Edit Score	instrument_name	String	Holds the chosen instrument in a valid form to be passed into the add_instrument function.
Score Editor	Edit Score	instrument_track	String	Holds the chosen instrument out of the selection of instrument tracks already in the composition.
Score Editor	Edit Score	duration	String	Holds the duration name of the note to be added in a valid form to be passed add_note function.
Score Editor	Edit Score	note_name	String	Holds the name of the note to be added in a valid form to be passed add_note function.
Score Editor	Edit Score	octave	Integer	Holds the octave of the note to be added in a valid form to be passed add_note function.
Score Editor	Edit Score	bar_num	Integer	Holds an identifier of which bar should be affected in the specified track.
Score Editor	Edit Score	position	Real	Holds the relative beat that should be affected in the specified bar.
Score Libraries	Display Scores, Sort, Search	scores	Array of Score objects	Holds the scores that have been selected to be shown in a library webpage.
User Accounts	Check details, Make Record, Validate Details	username	String	Holds the username entered by the user.
User Accounts	Make Record, Validate Details	email	String	Holds the email entered by the user.
User Accounts	Check details, Make Record, Validate Details	password	String	Holds the password entered by the user.

Validation & Verification

With form on the ‘Create Account’ webpage, double entry verification will be used when receiving data to send to the ‘Password’ field of the record being created. The form should contain two password input field and the data entered to both fields should be compared to see if they are identical; if they are processing continues, if not an error message is sent.

Presence checks should also be used on all input fields on the ‘Create Account’ and ‘Login’ pages to ensure no user records are made with empty fields and records are not searched for with incomplete information.

Range and format checks should be used on the password input field in the form on the ‘Create Account’ page. That data inputted should be checked if it is at least 8 characters, contains at least 1 lowercase letter, 1 uppercase letter, 1 digit and 1 special character.

Format checks should be used on the email input field in the form on the ‘Create Account’ page. As the ‘Email’ field is an instance of Django’s EmailField class, the email that tries to be entered into the database will automatically be checked to see if it is a valid email address.

Lookup validation should be used with the menus on the ‘Editor’, ‘User library’ and ‘Public library’ pages. Giving the user a list of discrete data to choose from eliminates the possibility of invalid data being passed through the system. All input fields on the ‘Editor’ page should be select menus other than ‘position’ fields. The menu of score characteristics on the library pages should be a list of radio options.

The ‘position’ field of some forms on the ‘Editor’ page is receives free data input from the user, so the data entered should be checked if it is a number and a range check should be made to ensure the number is between 1 and maximum number of beats that specific bar.

Main Algorithms

Score Editor

Open Score:

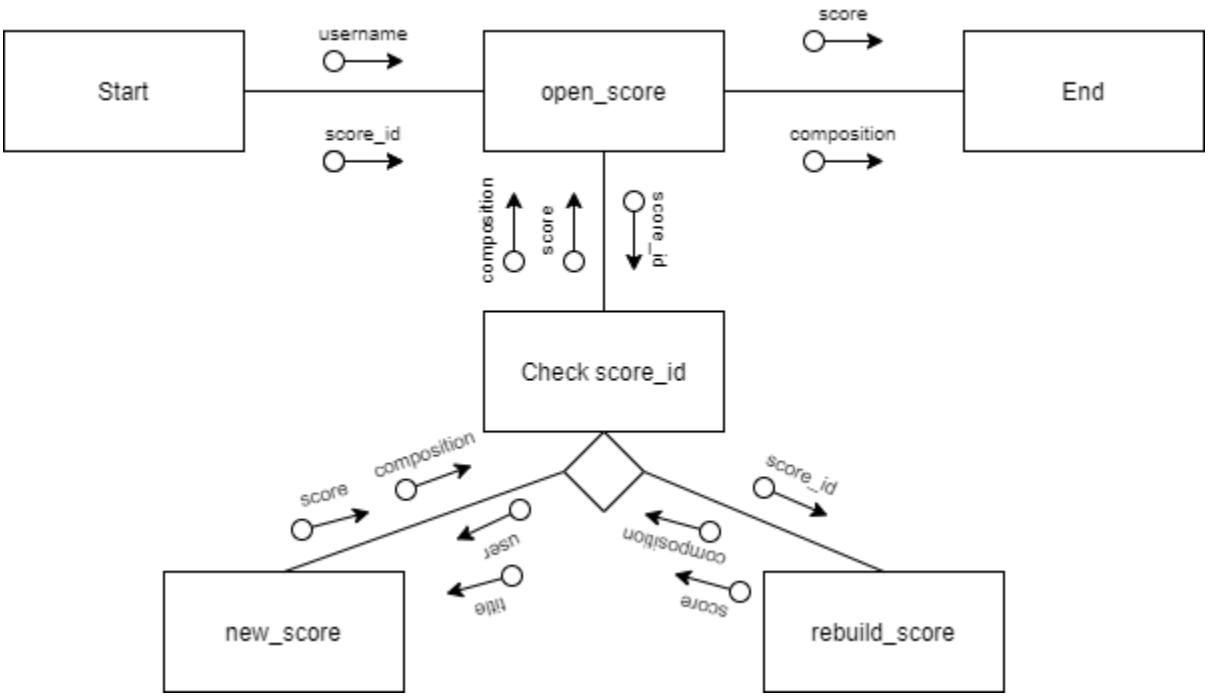


Fig. 24 – Structure chart for open_score module

Two variables, username and score_id, are inputted to the open_score procedure. Username is used to retrieve the required User record. Then a selection statement is used to check if the inputted score_id exists in the database – this is done to check if this score already exists or if a new score must be created. If score_id is found, the rebuild_score sub-routine is called and if not found the new_score sub-routine is called. Both procedures return 2 variables: score and composition.

Pseudocode for new_score:

```
FUNCTION new_score(title, user)
        score ← new Score(title = title, author = user.username, score_file = title + '.ly',
music_file = title + '.midi', composition_file = title + '.txt', pdf_file = title + '.pdf', user = user)
        save(score)
        composition ← new Composition()
        return score, composition
ENDFUNCTION
```

Pseudocode for rebuild_score:

```
FUNCTION rebuild_score(score_id)
        score ← Score.objects.get(primary_key = score_id)
        file ← score.composition_file
        open(file, mode = 'read')
        composition_string ← read(file)
        close(file)
        composition ← new Composition()
        tracks_list ← composition_string.separate('?')

        FOR t ← element IN tracks_list DO
                t_list ← t.separate(';')
                instrument_name ← t_list[0]
                instrument ← new MidiInstrument(instrument_name)
                track ← Track(instrument)

                FOR b ← element in t_list[1 : end] DO
                        b_list ← b.separate('~')
                        new_bar ← new Bar(b_list[0], (int(b_list[1]), int(b_list[2])))
                        new_bar.bar ← list(b_list[3])
                        track.add_bar(new_bar)
                ENDFOR

                composition.add_track(track)
        ENDFOR

        return score, composition
ENDFUNCTION
```

Edit Score:

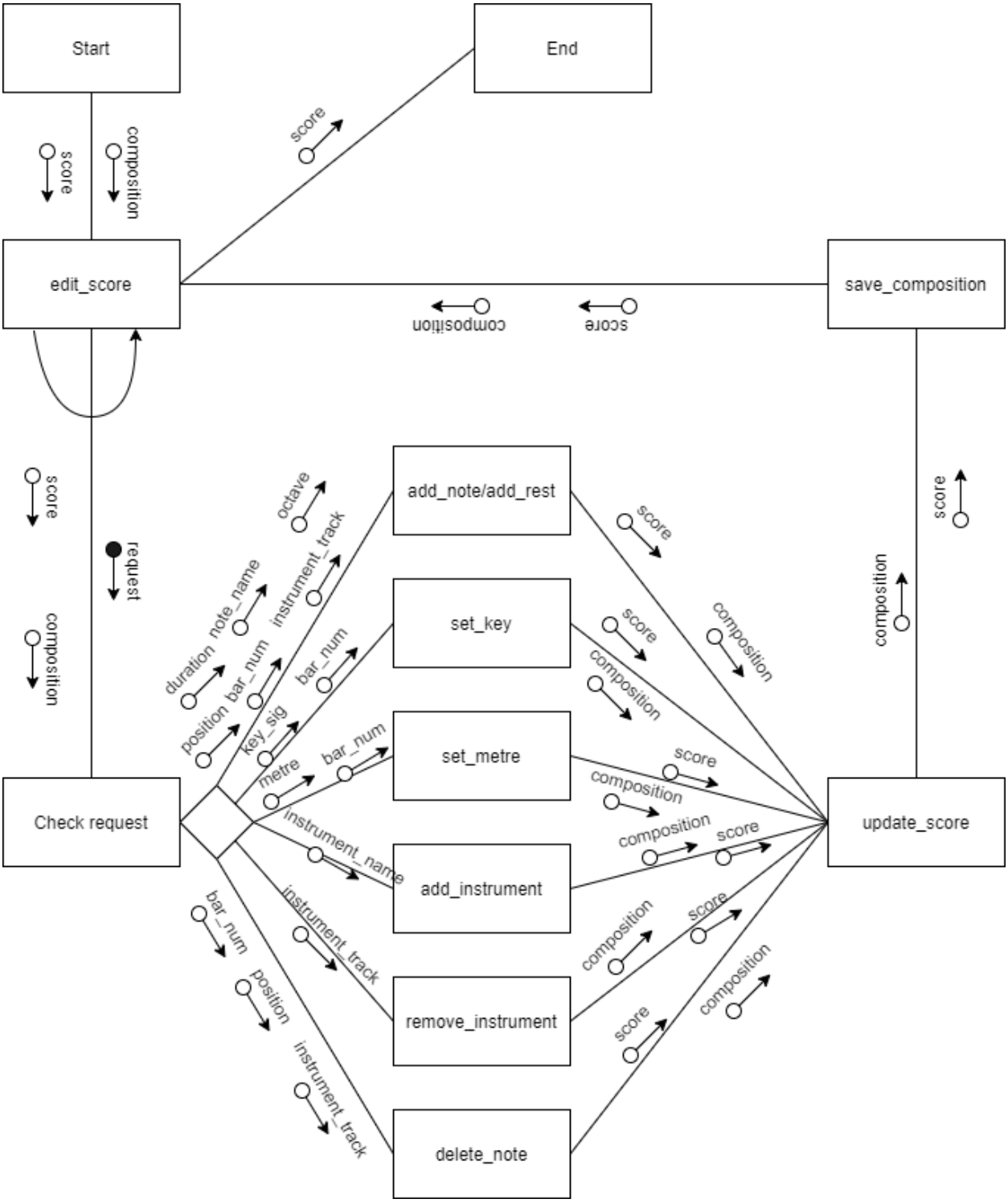


Fig. 25 – Structure chart for edit_score module

Two variables from the open_score module, score and composition, are inputted to the edit_score module. When the user submits a form on the 'edit' webpage a 'request' is sent which carries the form data and an identifier for which sub-rountine should carry out the request. Then either the add_instrument, add_note/add_rest, set_key, set_metre, delete_note or remove_instrument sub-routine is called - all these procedures change the variable composition. Then the procedures update_score and save_composition are called that change the variable score.

Pseudocode for add_note/add_rest:

```
PROCEDURE add_note(composition, duration, instrument_track, note_name, octave, bar_num,
position)
        t ← select_track(instrument_track, composition)

        IF note_name = NULL AND octave = NULL THEN
                new_note ← NULL
        ELSE
                new_note ← new Note(note_name, octave)
        ENDIF

        IF duration = "semibreve" THEN
                length ← value.semibreve
        ELSEIF duration = "minim" THEN
                length ← value.minim
        ELSEIF duration = "crotchet" THEN
                length ← value.crotchet
        …
        ELSEIF duration = "triplet crotchet" THEN
                length ← value.triplet(value.crotchet)
        …
        ELSEIF duration = "dotted semibreve" THEN
                length ← value.dots(value.semibreve)
        …
        ENDIF

        chosen_bar ← t[bar_num – 1]
        beat ← (position – 1) / (chosen_bar.length * 4)
        space_left ← 1 / (chosen_bar.length – beat)
        temp1 ← chosen_bar.current_beat
        chosen_bar.current_beat ← beat

        IF space_left < length THEN
                chosen_bar.place_notes(new_note, space_left)
                next_bar ← t[bar_num]
```

'select_track' is a function I will code myself. Its purpose is to return the Track object within the Composition object that's name attribute is equal to the string held in the variable 'instrument_track'.

'value' is a class from the Mingus library.

```
            temp2 ← next_bar.current_beat
            next_bar.current_beat ← 0
            next_bar.place_notes(new_note, (length – space_left))
            next_bar.current_beat ← temp2
        ELSE
            chosen_bar.place_notes(new_note, length)
        ENDIF

        chosen_bar.current_beat ← temp1
ENDPROCEDURE
```

Pseudocode for set_key:

```
PROCEDURE set_key(composition, key_sig, bar_num)
        FOR t ← element IN composition.tracks DO
            FOR i ← (bar_num – 1) TO length(t) DO
                t[i].key ← key_sig
            ENDFOR
        ENDFOR
ENDPROCEDURE
```

Pseudocode for set_metre:

```
PROCEDURE set_metre(composition, metre, bar_num)
        FOR t ← element IN composition.tracks DO
            FOR i ← (bar_num – 1) TO length(t) DO
                t[i].set_meter(metre)
            ENDFOR
        ENDFOR
ENDPROCEDURE
```

Pseudocode for add_instrument:

```
PROCEDURE add_instrument(composition, instrument_name)
        instrument ← new MidiInstrument(instrument_name)
        new_track ← new Track(instrument)
        new_track.name ← instrument.name
        composition.add_track(new_track)
ENDPROCEDURE
```

Pseudocode for remove_instrument:

```
PROCEDURE remove_instrument(composition, instrument_track)
        track ← select_track(instrument_track, composition)
        composition.tracks.remove(track)
ENDPROCEDURE
```

Pseudocode for delete_note:

```
PROCEDURE delete_note(composition, instrument_track, bar_num, position)
        t ← select_track(instrument_track, composititon)
        chosen_bar ← t[bar_num – 1]
        beat ← (position – 1) / (chosen_bar.length *4)
        temp ← chosen_bar.current_beat
        chosen_bar.current_beat ← beat

        FOR n ← element IN chosen_bar DO
                IF n[0] = beat THEN
                        duration ← n[1]
                        chosen_bar.place_rest(duration)
                ENDIF
        ENDFOR

ENDPROCEDURE
```

lilypond and midi are both modules from the Mingus library.

Pseudocode for update_score:

```
PROCEDURE update_score(composition, score)
        lilypond_string ← lilypond.from_Composition(composition)
        lily_file ← score.score_file
        lilypond.to_pdf(lilypond_string, lily_file.name)
        midi_file ← score.music_file
        midi.midi_file_out.write_Composition(midi_file.name, composition)
        score.save()
ENDPROCEDURE
```

Pseudocode for save_composition:

```
PROCEDURE save_composition(composition, score)
        file ← score.composition_file
        open(file, mode = 'write')
        composition_string ← ''

        FOR t ← element IN composition.tracks DO
                composition_string ← composition_string + t.name + ';'

                FOR b ← element IN t.bars DO
                        composition_string ← composition_string + b.key + '~'
                        composition_string ← composition_string + str(b.meter[0]) + '~'
                        composition_string ← composition_string + str(b.meter[1]) + '~'
                        composition_string ← composition_string + str(b.bar) + ';'
                ENDFOR
```

```
            compositon_string ← composition_string + '?'
        ENDFOR
        write(composition_string, file)
        close(file)
        score.save()
ENDPROCEDURE
```

Score Libraries

Pseudocode for retrieve_scores:

```
FUNCTION retrieve_scores(library, username)
        scores ← []

        IF library = "public" THEN
                FOR s ← element IN Score.objects.all() DO
                        IF s.public = True THEN
                                scores.add_element(s)
                        ENDIF
                ENDFOR

        ELSEIF library = "user" THEN
                FOR s ← element IN Score.objects.all() DO
                        IF s.user.username = username THEN
                                scores.add_element(s)
                        ENDIF
                ENDFOR

        ENDIF
        return scores
ENDFUNCTION
```

Two variables, 'library' and 'username' are passed into retrieve_score. If 'library' contains "public", only the scores where the public attribute is set to True are retrieved from the database and added to the list 'scores'. If 'library' contains "user", only scores where the username attribute of the user object in the user attribute is equal to the string in 'username' are retrieved from the database and added to the list 'scores'.

Sort:

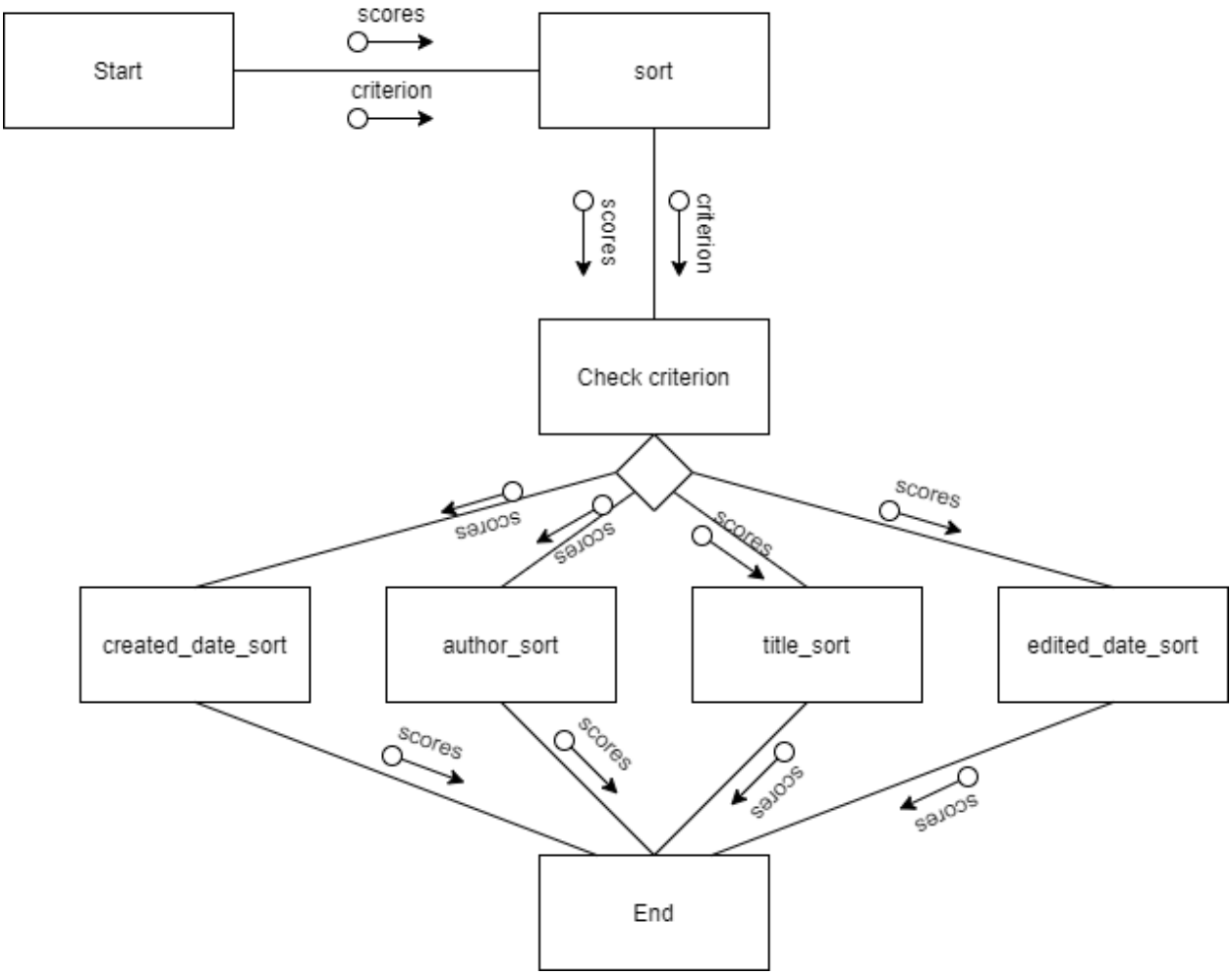


Fig. 26 – Structure chart for sort module

Two variables, 'scores' and 'criterion', are inputted to the sort submodule. 'scores' is the list of score objects to be sorted and 'criterion' is the what should be used to determine the order – either the title, author, last modified or date created attribute. Depending on the value of 'criterion', the correct sorting function is chosen. Each of which are a version of the merge sort algorithm; I chose to use merge sort as the 'scores' list is likely to be very large and merge sort has a good time complexity – O(nlog n). Then the sorted 'scores' list is returned.

Pseudocode for title_sort:

```
PROCEDURE title_sort(scores)
        IF length(scores) > 1 THEN
                mid ← length(scores) DIV 2
                left_half ← scores[0 : mid]
                right_half ← scores[mid : END]
```

```
            title_sort(left_half)
            title_sort(right_half)
            i ← 0
            j ← 0
            k ← 0

            WHILE i < length(left_half) AND j < length(right_half) DO
                IF left_half[i].title < right_half[j].title THEN
                    scores[k] ← left_half[i]
                    i ← i + 1
                ELSE
                    scores[k] ← right_half[j]
                    j ← j + 1
                ENDIF
                k ← k + 1
            ENDWHILE

            WHILE i < length(left_half) DO
                scores[k] ← left_half[i]
                i ← i + 1
                k ← k + 1
            ENDWHILE

            WHILE j < length(right_half) DO
                scores[k] ← right_half[j]
                j ← j + 1
                k ← k + 1
            ENDWHILE

        ENDIF
ENDPROCEDURE
```

The other 3 sorting functions will take a similar structure.

User Accounts

Pseudocode for validate_details:

```
FUNCTION validate_details(email, password)
        valid_password ← False
        valid_email ← False
        special_charcters ← ['@', '!', '£', '%', '^', '&', '*', '(', ')', '-', '_', '=', '+', '?', '[', ']', '{', '}']
        lower ← 0
        upper ← 0
        digit ← 0
        special ← 0
```

```
        FOR char ← element IN password DO
                IF char in special_characters THEN
                        special ← special + 1

                ELSEIF char.is_alpha() THEN
                        IF char.is_lower() THEN
                                lower ← lower + 1
                        ELSEIF char.is_upper() THEN
                                upper ← upper + 1
                        ENDIF

                ELSEIF char.is_number() THEN
                        digit ← digit + 1
                ENDIF
        ENDFOR

        IF special > 0 AND lower > 0 AND upper > 0 AND digit > 0 AND length(password) >= 8
THEN
                valid_password ← True
        ENDIF

        IF '@' IN email AND '.' IN email AND email[-1] != '@' AND email[-1] != '.' THEN
                valid_email ← True
        ENDIF

        IF valid_password AND valid_email THEN
                return True
        ELSE
                return False
        ENDIF
ENDFUNCTION
```

Two variables, 'email' and 'password', are passed into the validate_details function. A for loop is used the access each character in the string 'password' and check if it's an uppercase letter, a lowercase letter, a digit or a special character and increment the appropriate counter. If all counters are more than 0 and the password is at least 8 characters long, the 'valid_password' variable is switched to True. If the string 'email' contains the characters '@' and '.' and the last character is not either of those characters, the 'valid_email' password is switched to True. If both flag variables are True, the function returns True and if not the function returns False.

Development Plan

My system will be developed in 3 iterations: the first is coding score editor module, the second is coding the user accounts and libraries modules of the system and the third is refining the user interface and functionality of the entire system.

During the first cycle, I will code the main skeleton of the score editor module of system. This consists of the setting up the relational database, coding all the main functions and procedures used to create and modify a score and creating a basic editor webpage with little styling. I will then test the functionality of the module against the system requirements.

During the second cycle, I will code the remaining user accounts and libraries modules. This consists of coding all the functions and procedures used to allow a user to create an account or login using the database, coding all the functions and procedures used to retrieve score records from the database and list the requested scores in specific orders and creating the home, create account, login, public library and user library webpages with little styling. Then the 2 new modules will be tested against the system requirements.

During the third cycle, the feedback from the testing in the first and second cycles will be used to modify and improve all 3 modules. Then the user interface will be improved by adding styling and ensuring all webpages are uniform. Then the entire website will be tested against the system requirements again.

Deliverables

First Cycle

- Basic Score Editor module
- Documented tests and solution suggestions

Second Cycle

- Basic User Accounts module
- Basic Score Libraries module
- Documented tests and solution suggestions

Third Cycle

- Styling Finished
- Fully functioning website
- Documented tests

Test Plan

Alpha testing will be used during the first two cycles as I will be the person carrying all the tests. In the first cycle, I will first carry out a round of black box testing by checking if the score editor module meets a selection of system objectives from the analysis stage. In the table below I have listed the inputs I will use and the expected outputs, when carrying out the test I will state whether the correct output was given. For all tests that failed, I will then carry white box tests to work through the code line by line to find the errors and bugs. I will then document the steps I plan to take to solve the problem. I will take this same approach at the end of the second cycle. Then, at end of the third cycle, black box testing will be again to test if improvements have been made from the previous cycles. Beta testing will also be carried out; I will get 2 people who have never seen the website to test it and give me feedback.

First Cycle

Test No.	Objective No.	Purpose	Input Data	Type of Data	Expected Output
1	5	Test whether the input menus on the edit webpage can be navigated.	Test carried out by clicking and scrolling through the menus.	n/a	When a button is clicked to open each menu, the menu appears.
2	8, 12	Test whether the visual representation (embedded pdf) appears on the edit page	n/a	n/a	That it appears and changes when score is updated.
3	9	Test whether a sound that matches the pitch of the note and voice of the instrument when a note is added.	Input to add note form: 1) Instrument = 'violin', note = 'C', octave = 4, bar = 1, position = 1, duration = 'minim' 2) Instrument = 'clarinet', note = 'E', octave = 4, bar = 1, position = 1, duration = 'minim'	Normal	Sound should be heard when both notes are added. The first sound should sound like a violin and second like a clarinet. The first sound should be lower in pitch than the second.
4	10, 19	Test whether the score can be played within the edit page.	Click play button. The score should already contain one track, in the	Normal	The sound should be played. 4 identical sounds in the

			grand piano voice, with 4 c-4 crotchet notes.		voice of a grand piano should be heard.
5	10, 19	Test whether an empty score will not play sound but will not cause an error.	Click play button. The score should contain one track, in the grand piano voice, with no notes.	Boundary	No sound should be heard because audio is zero seconds long.
6	13	Test whether a note can be successfully added to the score.	Input to add note form: 1) Instrument = 'grand piano', note = 'C', octave = 4, bar = 1, position = 1, duration = 'minum' 2) Instrument = 'acoustic guitar', note = 'E', octave = 5, bar = 1, position = 3, duration = 'dotted quaver' 3) Instrument = 'viola', note = 'A', octave = 4, bar = 1, position = 1.5, duration = 'triplet crotchet'	Normal	At the end of adding all three notes, the composition object should look like: [[Acoustic Grand Piano ['F-0' – 'B-8'], [[[0.0, 2, ['C-4']]]]], [Acoustic Guitar ['E-3' – 'E-7'], [[[0.5, 5.3333333, ['E-5']]]]], [Viola['C-3' – 'E-6'], [[[0.125, 6.0, ['A-4']]]]]]
7	15	Test whether the key can be successfully changed.	Input to key form: 1) Key = 'A minor', bar = 2 2) Key = 'D major', bar 2	Normal	First time, the value of composition[0][0].key should be 'C' and composition[0][1].key should be 'a'. Second time, the value of composition[0][0].meter should be 'C' and the value of composition[0][1].key should be 'D'.
8	16	Test whether the time signature (metre) can be successfully changed.	Input to metre form: 1) Metre = 3/2, bar = 2 2) Metre = 6/8, bar = 2	Normal	First time, the value of composition[0][0].meter should be (4, 4) and the value of composition[0][1].meter should be (3, 2). Second

					time, the value of composition[0][0].meter should be (4, 4) and the value of composition[0][1].meter should be (6, 8).
9	17	Test whether a new instrument can be added successfully.	Input to instrument form: 1) Instrument = 'Piccolo' 2) Instrument = 'Tuba'	Normal	The composition object should look like: [[Piccolo ['D-4' – 'C-7'], []], [Tuba ['D-1' – 'F-4'], []]]
10	20	Test whether a note can be deleted successfully.	(The score should already contain one track, in the grand piano voice, with 4 c-4 crotchets.) Then input to delete note form: 1) Bar = 1, position = 1 2) Bar = 1, position = 3	Normal	The composition object should look like: [[Acoustic Grand Piano ['F-0' – 'B-8'], [[[0.0, 4, None], [0.25, 4, ['C-4']], [0.5, 4, None], [0.75, 4, ['C-4']]]]]]
11	n/a	Test whether an instrument can be deleted from the score successfully.	(The score should contain 3 tracks – one with the violin voice, one with the Oboe voice and one with the Timpani voice.) Input to form: 1) Track = 'Oboe' 2) Track = 'Violin'	Valid	The composition object should look like: [[Timpani ['D-2' – 'C-4'], []]]

Second Cycle

Test No.	Objective No.	Purpose	Input Data	Type of Data	Expected Output
1	1	Test whether the create account page allows you to make an account when details are valid successfully.	Input to form: 1) Username = 'ben10', email = 'benny@aol.com', password = 'Abc56Ht&%'	Normal	The details are accepted without an error message and we're redirected to the user library page.

2	2, 3	Test whether the create account page can detect invalid details.	Input to form: 1) Username = 'Rona2020', email = 'ronacov19gmail.co.uk', password1 = 'GiveYusefAnA*Please01', password2 = 'GiveYusefAnA*Please01' 2) Username = 'Grace26', email = 'gg45678@yahoo.com', password1 = '123', password2 = '123' 3) Username = 'jeremy', email = '456@hotmail.co.uk', password1 = 'google456', password2 = 'google123'	Erroneous	All three times we should be redirected back to the create account page and a message should appear on the page, explaining the problem.
3	4	Test whether the form on the login page allows you to login with valid details.	(This account must be made beforehand and can be done through admin site.) Input to form: 1) Username = 'ben10', password = 'Abc56Ht&%'	Normal	The details should be accepted without giving an error message and we should be redirected to the user library page.
4	4	Test whether the login page will reject details that don't exist within the database.	(This account should not exist.) Input to form: 1) Username = 'abi', password = 'password987'	Erroneous	We should be redirected to the login page and a message should appear on screen explaining the problem.
5	6	Test whether the search bar on the public library page successfully decreases the number of scores displayed according to texted typed in.	(Scores with titles 'sunny day', 'Sunday' and 'sun's out' should be already made. Scores with author 'ben10' and 'Rona2020' should be already made.) Input into form: 1) Search text = 'sun' 2) Search text = '20' 3) Search text = 'apple'	Normal, Erroneous	Only scores which have titles or authors which contain the search text should remain.
6	11	Test whether a file is successfully downloaded when the export button	One score should be chosen on each page and the export button should be pressed to export both as a MIDI and PDF file.	Normal	For each score, two non-empty files should be downloaded – one

		on the user library and public library pages.			pdf and one MIDI audio.
7	n/a	Test whether the scores displayed on the public library page can be sorted based on different criteria successfully.	(At least 4 scores must be displayed.) Input to form: 1) Radio option = 'title' 2) Radio option = 'author' 3) Radio option = 'last modified' 4) Radio option = 'date created'	Normal	Each time the order of the scores must change. First in alphabetical order by title, second in alphabetical order by author, third order by last modified date (most recent first) and lastly order by date created (most recent first).
8	n/a	Test whether all links on all pages are functioning properly.	Go to each page and click on all the links.	Normal	All links should lead to their intended destination webpages.
9	n/a	Test whether a score can successfully be uploaded to the public library on the user library page.	Select one score and click the upload button.	Normal	The page should reload and the public column of the chosen score's row have changed from "No" to "Yes".
10	n/a	Test whether a score in the score table can be opened successfully from the user library page.	Select one score and click the open button.	Normal	We should be redirected to the editor page and the correct score should appear on the page.