Tonebox

How it works

Basically there are number of model classes that are in charge of storing the data about songs-playlists in the database and also getting the stored data from the database.

Then we have a number of views that are in charge of getting the information from the models filtering and displaying them.

In order to run the app you need to have the pyside2 and tiny tag.

An overview of the application

Tonebox consists of three parts core, gui, tools.

Core

Core has the manager, manager model, media player, queue model modules.

Manager

Its responsible for connecting to the database adding and removing songs and playlists from the database also

Adding and removing songs from playlists and filtering songs and playlists.

Manager model

It inherits from manager so it does all the stuff that manager can, except it sends a signal to views so they can edit themselves.

Media player

Its responsible for playing the songs.

Queue model

Manages the song queues.

Gui

It contains the rename, info and settings dialogs and their implementations, Also the main ui and media player and queue widgets(that are part of the main ui) and finally the views module.

Views

it connects to the models and shows info to the user and if it gets a signal indicating a change in the info it edits its self. Basically its in charge of displaying information about the songs and playlists to the user and also filtering the information in a certain way.

Tools

It contains the setup module which has the default settings and checks if there is a settings file validates it (if the settings file doesn't exist it creates one) and its also in charge of updating, reading, writing from the settings file.

In the rest of the document we try to explain some parts of the application in more detail.

Manager

Its imports are sqlite3 and tinytag and it contains 3 classes song, playlist, Manager.

Playlist

Playlist has 3 instance variables.

Db_id : an integer given to it by the database

Name: a string set by the user

Songs : a list of songs that it contains given to it by the managers get _all_data method

Song

Song has 16 instance.

Db_id given by the database , path of where the song is, tag used by tiny tag to get the songs other instances that are title, album, artist, track total (the number of tracks in the album the song is in),

Duration, genre, year, composer, bitrate, sample rate, comment, image and file size.

Manager

Manager has a class variable called "SQLITE_SCHEMA" and 17 methods.

SQLITE_SCHEMA

It has 3 tables

Songs: it has 2 fields the song_id (generated by the database itself) and path.

Playlists: it has 2 fields the playlist_id (generated by the database itself) and name.

SongsPlaylistsGroups: it has 4 fields record_id (generated by the database itself), song_id(foreign key referencing the song_id field in songs), playlist_id(foreign key referencing the playlist_id field in playlists) and finally playlist_order which is the songs position in a playlist.

The __init__ method

Its arguments are the instance itself (like all the other methods) and db_path which uses it to set the instance variable db_path.

And then it sets the instance variables songs and playlists to an emty dictionary.

Then it calles the open_connection method and if that return true it calls the setup_database and get_all_data methods.

Open_connection

Tries to connect to the database and set a cursor and then it return 1.

It passes the exceptions to the show_errors_to_user.

Is_database_valid

…

Setup_database

If the database is not valid it calles the close_connection method and raises and exception.

Otherwise it sets up the database using SQLITE_SCHEMA(which contained the databases schema) and commits it and if there are any errors it passes it to show_errors_to_users plus the “setup_database” string which is where the error occurred.

Get_all_data

Gets all the song paths stored in the database plus their ids generated by the database.

Uses the path and id to make instances of songs and then populates the songs dictionary (instance variable initially set to an emty dictionary in the init method) which in it the keys are the ids given by the database and the values are instances of the song class.

Uses the name and id to make instances of playlists and then populates the playlists dictionary (instance variable initially set to an emty dictionary in the init method) which in it the keys are the ids given by the database and the values are instances of the playlist class and creates a list called playlist_ids containing all of the playlist ids in the database.

Iterates over the playlist ids in playlist_ids and gets all the song_ids of the songs it contains plus the order that they were add to the playlist and then populates each playlist instances songs variable(a list containing instances of the songs it has). If any errors occur it passes them to the show_errors_to_users method.

Add_playlist

Takes in the playlists name inserts it into the database gets the id generated by the database to populate the playlists dictionary(instance variable initially set to an emty dictionary in the init method) and commits the changes. And returns True.

If any errors occur it passes the error to the show_errors_to_users method and returns False.

Remove_playlist

It takes in either the name of the playlist the user wants to remove or its id.

If it takes the name it gets the playlists id from the playlists dictionary(instance variable initially set to an emty dictionary in the init method and then populated in the get_all_data method) and then using the id it deletes the key-value of that id in the playlists dictionary plus every record that contains the id in the playlists and songsplaylistsgroups tables in the database and commits the changes and returns True.

If any errors occur it passes the error to the show_errors_to_users method.

Add_song

Takes in the songs path inserts it into the database gets the id generated by the database to populate the songs dictionary(instance variable initially set to an emty dictionary in the init method and then populated in the get_all_data method) and commits the changes. And returns True.

If any errors occur it passes the error to the show_errors_to_users method and returns False.

Remove_song

It takes in either the path of the song the user wants to remove or its id.

If it takes the path it gets the playlists id from the songs dictionary(instance variable initially set to an emty dictionary in the init method and then populated in the get_all_data method) and then using the id it deletes the key-value of that id in the songs dictionary plus every record that contains the id in the songs and songsplaylistsgroups tables in the database and commits the changes and returns True.

If any errors occur it passes the error to the show_errors_to_users method.

Songs_dict_filter

makes a list of songs(the elements of the list are song ids) that have a certain attribute and then it returns that list.

Playlists_dict_filter

makes a list of playlists (the elements of the list are playlist ids) that have a certain attribute and then returns it.

Add_song_to_playlist

it either takes a songs path and the name of the playlist the user wants to add the song to or their ids.

If it takes the path and the name it gets the playlists-songs id from the playlists-songs dictionary(instance variable initially set to an emty dictionary in the init method and then populated in the get_all_data method)

Adds the instance of song to the instance of the playlists songs list.

Inserts the song-playlist ids and the length of the instance of the playlists songs list(which is the playlists order) in the songsplaylistsgroups table in the database and commits the changes and returns true.

If any errors occur it passes the error to the show_errors_to_users method.

Remove_song_from_playlist

it either takes a songs name and the playlist the user wants to remove the song from or their ids.

If it takes the path and the name it gets the playlists-songs id from the playlists-songs dictionary(instance variable initially set to an emty dictionary in the init method and then populated in the get_all_data method).

removes the instance of song from the instance of the playlists songs list.

Gets the first records(of the song being added to the playlist) playlist_order stores it in a flag deletes the record from the database containing the song-playlist id and flag and then updates the playlist_order field (decreases it by 1) of all the records that have a larger playlist_order and the commits the changes and returns true.

If any errors occur it passes them to the show_errors_to_users method and returns False.

Edit_playlist_name

It takes in the id of the playlist and its new name it changes the name of the instance of that playlist in the playlist dictionary(instance variable initially set to an emty dictionary in the init method and then populated in the get_all_data method) using the playlist id.

If any errors occur it passes them to the show_errors_to_users method.

Filter

It takes in a database query and tries to execute and then commit the changes.

If any errors occur it passes them to the show_errors_to_users method.

Close_connection

Closes the connection to the database.

Show_errors_to_users

It takes in the error message and a key-word argument place and prints the error message and the place it took place.

Manager model

its imports are QObejct and singl From pysdie2.Qtcore and QMessageBox from PySide2.QtWidgets and finally the manager class from Manager.

Manager model only has one class called model.

The model class

This class inherits from manager and QObject(so it can have signals as class variables) and has 7 class variables, 9 methods.

Class variables

All of the class variables are signals and are called songAdded, playlistAdded, playlistRemoved, songAddedToPlaylist, songRemovedFromPlaylist, modelUpdated.

Methods

the __init__ method

calls the Qobject and managers init methods.

show_errors_to_user

takes in string called err and a keyword argument called place and will pass them to managers show_errors_to_users method to get the output. Then it calls Qmessagebox.critical with None, “Error” string, output which will show the error message to the user.

rename_playlist

tries the edit the playlists name using managers edit_playlist_name method if it fails it calls the show_errors_to_user mehod otherwise it emits the modelUpdated signal.

add_songs

it takes in any number of song paths. It also keeps the number of songs successfully added to the database in a variable called added_songs_count.

It starts off by iterating through the song paths and tries to add them to the database by calling the managers add_song method if that is successful it will increase added_songs_count by 1 And if there are any errors it calls the show_errors_to_user method.

At the end, if added_songs_count is larger than zero (if at least 1 song was successfully added to the database) it emits the songAdded, modelUpdated signals.

remove_songs

it takes in any number of song ids. It also keeps the number of songs successfully removed from the database in a variable called removed_songs_count.

It goes through the song ids and tries to remove them from the database by calling the managers remove_song method if that is successful it will increase removed_songs_count by 1 And if there are any errors it calls the show_errors_to_user method.

Finally, if removed_songs_count is larger than zero (if at least 1 song was successfully removed from the database) it emits the songRemoved, modelUpdated signals.

add_playlists

it takes in any number of playlist names. It also keeps the number of playlists successfully added to the database in a variable called added_playlists_count.

It iterats through the playlist names and tries to add them to the database by calling the managers add_playlist method if that is successful it will increase added_playlists_count by 1 And if there are any errors it calls the show_errors_to_user method.

At the end, if added_playlists_count is larger than zero (if at least 1 playlist was successfully added to the database) it emits the playlistAdded, modelUpdated signals.

remove_playlists

it takes in any number of playlist ids. It also keeps the number of playlists successfully removed from the database in a variable called removed_playlists_count.

It goes through the playlist ids and tries to remove them from the database by calling the managers remove_playlist method if that is successful it will increase removed_playlists_count by 1 And if there are any errors it calls the show_errors_to_user method.

if removed_playlists_count is larger than zero (if at least 1 playlist was successfully removed from the database) it emits the playlistRemoved, modelUpdated signals.

add_songs_to_playlist

its inputs are a playlist id and any number of song ids. It also keeps the number of songs successfully added to the playlist in a variable called added_songs_count.

It goes through the song ids and tries to add them to the database(to songsplaylists table) by calling the managers add_song_to_playlist method and also to the playlists songs list, if that is successful it will increase added_songss_count by 1 And if there are any errors it calls the show_errors_to_user method.

if added_songs_count is larger than zero (if at least 1 song was successfully added to the playlist) it emits the songAddedToPlaylist, modelUpdated signals.

remove_songs_from_playlist

its inputs are a playlist id and any number of song ids. It also keeps the number of songs successfully removed from the playlist in a variable called removed_songs_count.

It goes through the song ids and tries to remove them from the database(the record containing the song id and the playlist id from the songsplaylists table) by calling the managers remove_song_from_playlist method and also from the playlists songs list, if that is successful it will increase removed_songss_count by 1 And if there are any errors it calls the show_errors_to_user method.

if removed_songs_count is larger than zero (if at least 1 song was successfully removed from the playlist) it emits the songRemovedFromPlaylist, modelUpdated signals.

Setup

Its imports are QObject, Signal from PySide2.QtCore and json and os.

It contains a function called validate_songs_view_headers and 2 classes called Settings and SettingsModel.

Validate_songs_view_headers function

It takes a dictionary in and checks if its keys are the same as the keys in the

SONGS_VIEW_HEADERS_TRANSLATIONS (a class variable in SettingsModel class) dictionary

If they are it returns true otherwise it returns false.

Settings class

This class has 4 class variables and 11 methods.

Class variables

DEFAULT_SETTINGS_PATH : contains the settings files path.

DEFAULT_JSON_FIELDS : a dictionary containing default settings.

DEFAULT_JSON_FIELDS_VALIDATORS: a dictionary which in it the keys are settings and the values are validators(all of them are lambda functions except for the SongsViewHeaders key, SongsViewHeaders validator is the validate_songs_view_headers function).

SUPPORTED_AUDIO_FILES : a list containing the supported audio formats.

methods

The __init__ method

Sets json_dict(instance variable) to an empty dictionary and calles the setup_settings method.

Setup_settings

it starts off by checking if a settings file exists it does this by calling the settings_file_exists method if the file does not exist it creates one by calling the create_settings_file method otherwise it calls the read_settings_file method (which sets the content of the file to the json_dict variable).

then it checks if the file is valid by calling the is_settings_file_valid method and if its not valid it passes the "Settings file not recognized. Making new one!" string to the show_messages_to_user method and then creates a new settings file by calling the create_settings_file method.

is_settings_file_valid

it starts off by checking if the Jason_dict keys are identical to the keys in the DEFAULT_JSON_FIELDS(a class variable in Settings class) dictionary If they are not it returns false.

However if they are it validates each keys values using the values in the DEFAULT_JSON_FIELDS_VALIDATORS(a class variable in Settings class) dictionary and if all values are valid it returns true otherwise it will return false.

settings_file_exists

using os.path checks if the settings file exists (the file path was stored in DEFAULT_SETTINGS_PATH)

if the file doesn't exist it returns False otherwise it tries to load the file and return true

if any errors occur it passes the error to the show_errors_to_user method and returns False.

Update

It takes a key and a value and it sets them in the Jason_dict dictionary and then it calls the write_settings_file method.

Get

It takes in a key returns the value of that key in the Jason_dict dictionary.

create_settings_file

it sets a copy of DEFAULT_JSON_FIELDS to json_dict and then it calles the write_settings_file method.

write_settings_file

opens the settings file (using DEFAULT_SETTINGS_PATH) in write mode and dumps the content of the json_dict in it.

read_settings_file

opens the settings file (using DEFAULT_SETTINGS_PATH) in read mode and loads the content of the file to json_dict.

show_messages_to_user

takes in a message and prints it.

show_errors_to_user

takes in a error and prints it.

SettingsModel class

The settingsmodel inherits from Qobject (so it can have signal as aclass variable) and settings.

It also has 2 class variables and 3 methods.

Class variables

SONGS_VIEW_HEADERS_TRANSLATIONS: the keys are the songs header in upper case and the values are the same thing in lower case

settingsUpdated: a signal

methods

The __init__ method

Calls the init methods of QObject and settings.

write_settings_file

calls the settings write_settings_file and then emits the settingsUpdated signal

read_settings_file

calls the settings read_settings_file and then emits the settingsUpdated signal