onlaj / Piano-LED-Visualizer Public

Code Issues 21 Pull requests 1 Discussions Actions Projects ···

6a732caa81

Piano-LED-Visualizer / webinterface / views_api.py / Jump to

SirLefti showing cover state in webinterface ... History

3 contributors

1463 lines (1125 sloc) | 67.4 KB

```python
from webinterface import webinterface
from flask import render_template, send_file, redirect, request, url_for, jsonify
from lib.functions import find_between, theaterChase, theaterChaseRainbow, sound_of_da_police, sca
    rainbow, rainbowCycle, fastColorWipe, play_midi, clamp
import psutil
import threading
from neopixel import *
import webcolors as wc
import mido
from xml.dom import minidom
from subprocess import call
import subprocess
import datetime
import os
import math
from zipfile import ZipFile
import json
import ast
import time
import RPi.GPIO as GPIO

SENSECOVER = 12
GPIO.setmode(GPIO.BCM)
GPIO.setup(SENSECOVER, GPIO.IN, GPIO.PUD_UP)

@webinterface.route('/api/start_animation', methods=['GET'])
def start_animation():
    choice = request.args.get('name')
    speed = request.args.get('speed')
    if choice == "theaterchase":
        webinterface.menu.t = threading.Thread(target=theaterChase, args=(webinterface.ledstrip.st
                                                                          Color(127, 127, 127),
                                                                          webinterface.ledsettings
                                                                          webinterface.menu))
        webinterface.menu.t.start()

    if choice == "theaterchaserainbow":
        webinterface.t = threading.Thread(target=theaterChaseRainbow, args=(webinterface.ledstrip.
                                                                            webinterface.ledsettin
                                                                            webinterface.menu, 5))
        webinterface.t.start()

    if choice == "soundofdapolice":
        webinterface.t = threading.Thread(target=sound_of_da_police, args=(webinterface.ledstrip.s
                                                                           webinterface.ledsetting
                                                                           webinterface.menu, 1))
        webinterface.t.start()

    if choice == "scanner":
        webinterface.t = threading.Thread(target=scanner, args=(webinterface.ledstrip.strip,
                                                                webinterface.ledsettings,
                                                                webinterface.menu, 1))
        webinterface.t.start()

    if choice == "breathing":
        if speed == "fast":
            webinterface.t = threading.Thread(target=breathing, args=(webinterface.ledstrip.strip,
                                                                      webinterface.ledsettings,
                                                                      webinterface.menu, 5))
            webinterface.t.start()
        if speed == "medium":
            webinterface.t = threading.Thread(target=breathing, args=(webinterface.ledstrip.strip,
                                                                      webinterface.ledsettings,
                                                                      webinterface.menu, 10))
            webinterface.t.start()
        if speed == "slow":
            webinterface.t = threading.Thread(target=breathing, args=(webinterface.ledstrip.strip,
                                                                      webinterface.ledsettings,
                                                                      webinterface.menu, 25))
            webinterface.t.start()

    if choice == "rainbow":
        if speed == "fast":
            webinterface.t = threading.Thread(target=rainbow, args=(webinterface.ledstrip.strip,
                                                                    webinterface.ledsettings,
                                                                    webinterface.menu, 2))
            webinterface.t.start()
        if speed == "medium":
            webinterface.t = threading.Thread(target=rainbow, args=(webinterface.ledstrip.strip,
                                                                     webinterface.ledsettings,
                                                                     webinterface.menu, 20))
            webinterface.t.start()
        if speed == "slow":
            webinterface.t = threading.Thread(target=rainbow, args=(webinterface.ledstrip.strip,
                                                                     webinterface.ledsettings,
                                                                     webinterface.menu, 50))
            webinterface.t.start()

    if choice == "rainbowcycle":
        if speed == "fast":
            webinterface.t = threading.Thread(target=rainbowCycle, args=(webinterface.ledstrip.str
                                                                          webinterface.ledsettings,
                                                                          webinterface.menu, 1))
            webinterface.t.start()
        if speed == "medium":
            webinterface.t = threading.Thread(target=rainbowCycle, args=(webinterface.ledstrip.str
                                                                          webinterface.ledsettings,
                                                                          webinterface.menu, 20))
            webinterface.t.start()
        if speed == "slow":
            webinterface.t = threading.Thread(target=rainbowCycle, args=(webinterface.ledstrip.str
                                                                          webinterface.ledsettings,
                                                                          webinterface.menu, 50))
            webinterface.t.start()

    if choice == "stop":
        webinterface.menu.screensaver_is_running = False

    return jsonify(success=True)


@webinterface.route('/api/get_homepage_data')
def get_homepage_data():
    try:
        temp = find_between(str(psutil.sensors_temperatures()["cpu_thermal"]), "current=", ",")
    except:
        temp = find_between(str(psutil.sensors_temperatures()["cpu-thermal"]), "current=", ",")

    temp = round(float(temp), 1)

    upload = psutil.net_io_counters().bytes_sent
    download = psutil.net_io_counters().bytes_recv

    card_space = psutil.disk_usage('/')

    cover_opened = GPIO.input(SENSECOVER)

    homepage_data = {
        'cpu_usage': psutil.cpu_percent(interval=0.1),
        'memory_usage_percent': psutil.virtual_memory()[2],
        'memory_usage_total': psutil.virtual_memory()[0],
        'memory_usage_used': psutil.virtual_memory()[3],
        'cpu_temp': temp,
        'upload': upload,
        'download': download,
        'card_space_used': card_space.used,
        'card_space_total': card_space.total,
        'card_space_percent': card_space.percent,
        'cover_state': 'Opened' if cover_opened else 'Closed'
    }
    return jsonify(homepage_data)


@webinterface.route('/api/change_setting', methods=['GET'])
def change_setting():
    setting_name = request.args.get('setting_name')
    value = request.args.get('value')
    second_value = request.args.get('second_value')
    disable_sequence = request.args.get('disable_sequence')

    reload_sequence = True
    if (second_value == "no_reload"):
        reload_sequence = False

    if (disable_sequence == "true"):
        webinterface.ledsettings.__init__(webinterface.usersettings)
        webinterface.ledsettings.sequence_active = False

    if setting_name == "clean_ledstrip":
        fastColorWipe(webinterface.ledstrip.strip, True, webinterface.ledsettings)

    if setting_name == "led_color":
        rgb = wc.hex_to_rgb("#" + value)

        webinterface.ledsettings.color_mode = "Single"

        webinterface.ledsettings.red = rgb[0]
        webinterface.ledsettings.green = rgb[1]
        webinterface.ledsettings.blue = rgb[2]

        webinterface.usersettings.change_setting_value("color_mode", webinterface.ledsettings.colo
        webinterface.usersettings.change_setting_value("red", rgb[0])
        webinterface.usersettings.change_setting_value("green", rgb[1])
        webinterface.usersettings.change_setting_value("blue", rgb[2])

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "light_mode":
        webinterface.ledsettings.mode = value
        webinterface.usersettings.change_setting_value("mode", value)

    if setting_name == "fading_speed" or setting_name == "velocity_speed":
        webinterface.ledsettings.fadingspeed = int(value)
        webinterface.usersettings.change_setting_value("fadingspeed", webinterface.ledsettings.fad

    if setting_name == "brightness":
        webinterface.usersettings.change_setting_value("brightness_percent", int(value))
        webinterface.ledstrip.change_brightness(int(value), True)

    if setting_name == "backlight_brightness":
        webinterface.ledsettings.backlight_brightness_percent = int(value)
        webinterface.ledsettings.backlight_brightness = 255 * webinterface.ledsettings.backlight_b
        webinterface.usersettings.change_setting_value("backlight_brightness",
                                                       int(webinterface.ledsettings.backlight_brig
        webinterface.usersettings.change_setting_value("backlight_brightness_percent",
                                                       webinterface.ledsettings.backlight_brightne
        fastColorWipe(webinterface.ledstrip.strip, True, webinterface.ledsettings)

    if setting_name == "backlight_color":
        rgb = wc.hex_to_rgb("#" + value)

        webinterface.ledsettings.backlight_red = rgb[0]
        webinterface.ledsettings.backlight_green = rgb[1]
        webinterface.ledsettings.backlight_blue = rgb[2]

        webinterface.usersettings.change_setting_value("backlight_red", rgb[0])
        webinterface.usersettings.change_setting_value("backlight_green", rgb[1])
        webinterface.usersettings.change_setting_value("backlight_blue", rgb[2])

        fastColorWipe(webinterface.ledstrip.strip, True, webinterface.ledsettings)

    if setting_name == "sides_color":
        rgb = wc.hex_to_rgb("#" + value)

        webinterface.ledsettings.adjacent_red = rgb[0]
        webinterface.ledsettings.adjacent_green = rgb[1]
        webinterface.ledsettings.adjacent_blue = rgb[2]

        webinterface.usersettings.change_setting_value("adjacent_red", rgb[0])
        webinterface.usersettings.change_setting_value("adjacent_green", rgb[1])
        webinterface.usersettings.change_setting_value("adjacent_blue", rgb[2])

    if setting_name == "sides_color_mode":
        webinterface.ledsettings.adjacent_mode = value
        webinterface.usersettings.change_setting_value("adjacent_mode", value)

    if setting_name == "input_port":
        webinterface.usersettings.change_setting_value("input_port", value)
        webinterface.midiports.change_port("inport", value)

    if setting_name == "secondary_input_port":
        webinterface.usersettings.change_setting_value("secondary_input_port", value)

    if setting_name == "play_port":
        webinterface.usersettings.change_setting_value("play_port", value)
        webinterface.midiports.change_port("playport", value)

    if setting_name == "skipped_notes":
        webinterface.usersettings.change_setting_value("skipped_notes", value)
        webinterface.ledsettings.skipped_notes = value

    if setting_name == "add_note_offset":
        webinterface.ledsettings.add_note_offset()
        return jsonify(success=True, reload=True)

    if setting_name == "append_note_offset":
        webinterface.ledsettings.append_note_offset()
        return jsonify(success=True, reload=True)

    if setting_name == "remove_note_offset":
        webinterface.ledsettings.del_note_offset(int(value) + 1)
        return jsonify(success=True, reload=True)

    if setting_name == "note_offsets":
        webinterface.usersettings.change_setting_value("note_offsets", value)

    if setting_name == "update_note_offset":
        webinterface.ledsettings.update_note_offset(int(value) + 1, second_value)
        return jsonify(success=True, reload=True)

    if setting_name == "led_count":
        webinterface.usersettings.change_setting_value("led_count", int(value))
        webinterface.ledstrip.change_led_count(int(value), True)

    if setting_name == "shift":
        webinterface.usersettings.change_setting_value("shift", int(value))
        webinterface.ledstrip.change_shift(int(value), True)

    if setting_name == "reverse":
        webinterface.usersettings.change_setting_value("reverse", int(value))
        webinterface.ledstrip.change_reverse(int(value), True)

    if setting_name == "color_mode":
        reload_sequence = True
```

```python
            if (second_value == "no_reload"):
                reload_sequence = False

            webinterface.ledsettings.color_mode = value
            webinterface.usersettings.change_setting_value("color_mode", webinterface.ledsettings.colo
            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "add_multicolor":
            webinterface.ledsettings.addcolor()
            return jsonify(success=True, reload=True)

        if setting_name == "add_multicolor_and_set_value":
            settings = json.loads(value)

            webinterface.ledsettings.multicolor.clear()
            webinterface.ledsettings.multicolor_range.clear()

            for key, value in settings.items():
                rgb = wc.hex_to_rgb("#" + value["color"])

                webinterface.ledsettings.multicolor.append([int(rgb[0]), int(rgb[1]), int(rgb[2])])
                webinterface.ledsettings.multicolor_range.append([int(value["range"][0]), int(value["r

            webinterface.usersettings.change_setting_value("multicolor", webinterface.ledsettings.mult
            webinterface.usersettings.change_setting_value("multicolor_range",
                                                           webinterface.ledsettings.multicolor_range)

            return jsonify(success=True)

        if setting_name == "remove_multicolor":
            webinterface.ledsettings.deletecolor(int(value) + 1)
            return jsonify(success=True, reload=True)

        if setting_name == "multicolor":
            rgb = wc.hex_to_rgb("#" + value)
            webinterface.ledsettings.multicolor[int(second_value)][0] = rgb[0]
            webinterface.ledsettings.multicolor[int(second_value)][1] = rgb[1]
            webinterface.ledsettings.multicolor[int(second_value)][2] = rgb[2]

            webinterface.usersettings.change_setting_value("multicolor", webinterface.ledsettings.mult

            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "multicolor_range_left":
            webinterface.ledsettings.multicolor_range[int(second_value)][0] = int(value)
            webinterface.usersettings.change_setting_value("multicolor_range", webinterface.ledsetting

            return jsonify(success=True, reload_sequence=reload_sequence)

```

```
    if setting_name == "multicolor_range_right":
        webinterface.ledsettings.multicolor_range[int(second_value)][1] = int(value)
        webinterface.usersettings.change_setting_value("multicolor_range", webinterface.ledsetting

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "remove_all_multicolors":
        webinterface.ledsettings.multicolor.clear()
        webinterface.ledsettings.multicolor_range.clear()

        webinterface.usersettings.change_setting_value("multicolor", webinterface.ledsettings.mult
        webinterface.usersettings.change_setting_value("multicolor_range", webinterface.ledsetting
        return jsonify(success=True)

    if setting_name == "rainbow_offset":
        webinterface.ledsettings.rainbow_offset = int(value)
        webinterface.usersettings.change_setting_value("rainbow_offset",
                                                       int(webinterface.ledsettings.rainbow_offset
        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "rainbow_scale":
        webinterface.ledsettings.rainbow_scale = int(value)
        webinterface.usersettings.change_setting_value("rainbow_scale",
                                                       int(webinterface.ledsettings.rainbow_scale)
        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "rainbow_timeshift":
        webinterface.ledsettings.rainbow_timeshift = int(value)
        webinterface.usersettings.change_setting_value("rainbow_timeshift",
                                                       int(webinterface.ledsettings.rainbow_timesh
        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "speed_slowest_color":
        rgb = wc.hex_to_rgb("#" + value)
        webinterface.ledsettings.speed_slowest["red"] = rgb[0]
        webinterface.ledsettings.speed_slowest["green"] = rgb[1]
        webinterface.ledsettings.speed_slowest["blue"] = rgb[2]

        webinterface.usersettings.change_setting_value("speed_slowest_red", rgb[0])
        webinterface.usersettings.change_setting_value("speed_slowest_green", rgb[1])
        webinterface.usersettings.change_setting_value("speed_slowest_blue", rgb[2])

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "speed_fastest_color":
        rgb = wc.hex_to_rgb("#" + value)
        webinterface.ledsettings.speed_fastest["red"] = rgb[0]
        webinterface.ledsettings.speed_fastest["green"] = rgb[1]
        webinterface.ledsettings.speed_fastest["blue"] = rgb[2]
```

```python

        webinterface.usersettings.change_setting_value("speed_fastest_red", rgb[0])
        webinterface.usersettings.change_setting_value("speed_fastest_green", rgb[1])
        webinterface.usersettings.change_setting_value("speed_fastest_blue", rgb[2])

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "gradient_start_color":
        rgb = wc.hex_to_rgb("#" + value)
        webinterface.ledsettings.gradient_start["red"] = rgb[0]
        webinterface.ledsettings.gradient_start["green"] = rgb[1]
        webinterface.ledsettings.gradient_start["blue"] = rgb[2]

        webinterface.usersettings.change_setting_value("gradient_start_red", rgb[0])
        webinterface.usersettings.change_setting_value("gradient_start_green", rgb[1])
        webinterface.usersettings.change_setting_value("gradient_start_blue", rgb[2])

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "gradient_end_color":
        rgb = wc.hex_to_rgb("#" + value)
        webinterface.ledsettings.gradient_end["red"] = rgb[0]
        webinterface.ledsettings.gradient_end["green"] = rgb[1]
        webinterface.ledsettings.gradient_end["blue"] = rgb[2]

        webinterface.usersettings.change_setting_value("gradient_end_red", rgb[0])
        webinterface.usersettings.change_setting_value("gradient_end_green", rgb[1])
        webinterface.usersettings.change_setting_value("gradient_end_blue", rgb[2])

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "speed_max_notes":
        webinterface.ledsettings.speed_max_notes = int(value)
        webinterface.usersettings.change_setting_value("speed_max_notes", int(value))

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "speed_period_in_seconds":
        webinterface.ledsettings.speed_period_in_seconds = float(value)
        webinterface.usersettings.change_setting_value("speed_period_in_seconds", float(value))

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "key_in_scale_color":
        rgb = wc.hex_to_rgb("#" + value)
        webinterface.ledsettings.key_in_scale["red"] = rgb[0]
        webinterface.ledsettings.key_in_scale["green"] = rgb[1]
        webinterface.ledsettings.key_in_scale["blue"] = rgb[2]

```

```python
            webinterface.usersettings.change_setting_value("key_in_scale_red", rgb[0])
            webinterface.usersettings.change_setting_value("key_in_scale_green", rgb[1])
            webinterface.usersettings.change_setting_value("key_in_scale_blue", rgb[2])

            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "key_not_in_scale_color":
            rgb = wc.hex_to_rgb("#" + value)
            webinterface.ledsettings.key_not_in_scale["red"] = rgb[0]
            webinterface.ledsettings.key_not_in_scale["green"] = rgb[1]
            webinterface.ledsettings.key_not_in_scale["blue"] = rgb[2]

            webinterface.usersettings.change_setting_value("key_not_in_scale_red", rgb[0])
            webinterface.usersettings.change_setting_value("key_not_in_scale_green", rgb[1])
            webinterface.usersettings.change_setting_value("key_not_in_scale_blue", rgb[2])

            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "scale_key":
            webinterface.ledsettings.scale_key = int(value)
            webinterface.usersettings.change_setting_value("scale_key", int(value))

            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "next_step":
            webinterface.ledsettings.set_sequence(0, 1, False)
            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "set_sequence":
            if (int(value) == 0):
                webinterface.ledsettings.__init__(webinterface.usersettings)
                webinterface.ledsettings.sequence_active = False
            else:
                webinterface.ledsettings.set_sequence(int(value) - 1, 0)
            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "change_sequence_name":
            sequences_tree = minidom.parse("sequences.xml")
            sequence_to_edit = "sequence_" + str(value)

            sequences_tree.getElementsByTagName(sequence_to_edit)[
                0].getElementsByTagName("settings")[
                0].getElementsByTagName("sequence_name")[0].firstChild.nodeValue = str(second_value)

            pretty_save("sequences.xml", sequences_tree)

            return jsonify(success=True, reload_sequence=reload_sequence)

        if setting_name == "change_step_value":
```

```
        sequences_tree = minidom.parse("sequences.xml")
        sequence_to_edit = "sequence_" + str(value)

        sequences_tree.getElementsByTagName(sequence_to_edit)[
            0].getElementsByTagName("settings")[
            0].getElementsByTagName("next_step")[0].firstChild.nodeValue = str(second_value)

        pretty_save("sequences.xml", sequences_tree)

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "change_step_activation_method":
        sequences_tree = minidom.parse("sequences.xml")
        sequence_to_edit = "sequence_" + str(value)

        sequences_tree.getElementsByTagName(sequence_to_edit)[
            0].getElementsByTagName("settings")[
            0].getElementsByTagName("control_number")[0].firstChild.nodeValue = str(second_value)

        pretty_save("sequences.xml", sequences_tree)

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "add_sequence":
        sequences_tree = minidom.parse("sequences.xml")

        sequences_amount = 1
        while True:
            if (len(sequences_tree.getElementsByTagName("sequence_" + str(sequences_amount))) == 0
                break
            sequences_amount += 1

        settings = sequences_tree.createElement("settings")

        control_number = sequences_tree.createElement("control_number")
        control_number.appendChild(sequences_tree.createTextNode("0"))
        settings.appendChild(control_number)

        next_step = sequences_tree.createElement("next_step")
        next_step.appendChild(sequences_tree.createTextNode("1"))
        settings.appendChild(next_step)

        sequence_name = sequences_tree.createElement("sequence_name")
        sequence_name.appendChild(sequences_tree.createTextNode("Sequence " + str(sequences_amount
        settings.appendChild(sequence_name)

        step = sequences_tree.createElement("step_1")

        color = sequences_tree.createElement("color")
        color.appendChild(sequences_tree.createTextNode("RGB"))
        step.appendChild(color)

        red = sequences_tree.createElement("Red")
        red.appendChild(sequences_tree.createTextNode("255"))
        step.appendChild(red)

        green = sequences_tree.createElement("Green")
        green.appendChild(sequences_tree.createTextNode("255"))
        step.appendChild(green)

        blue = sequences_tree.createElement("Blue")
        blue.appendChild(sequences_tree.createTextNode("255"))
        step.appendChild(blue)

        light_mode = sequences_tree.createElement("light_mode")
        light_mode.appendChild(sequences_tree.createTextNode("Normal"))
        step.appendChild(light_mode)

        element = sequences_tree.createElement("sequence_" + str(sequences_amount))
        element.appendChild(settings)
        element.appendChild(step)

        sequences_tree.getElementsByTagName("list")[0].appendChild(element)

        pretty_save("sequences.xml", sequences_tree)

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "remove_sequence":
        sequences_tree = minidom.parse("sequences.xml")

        # removing sequence node
        nodes = sequences_tree.getElementsByTagName("sequence_" + str(value))
        for node in nodes:
            parent = node.parentNode
            parent.removeChild(node)

        # changing nodes tag names
        i = 1
        for sequence in sequences_tree.getElementsByTagName("list")[0].childNodes:
            if (sequence.nodeType == 1):
                sequences_tree.getElementsByTagName(sequence.nodeName)[0].tagName = "sequence_" +
                i += 1

        pretty_save("sequences.xml", sequences_tree)

        return jsonify(success=True, reload_sequence=reload_sequence)

    if setting_name == "add_step":
        sequences_tree = minidom.parse("sequences.xml")

        step_amount = 1
        while True:
            if (len(sequences_tree.getElementsByTagName("sequence_" + str(value))[0].getElementsBy
                    "step_" + str(step_amount))) == 0):
                break
            step_amount += 1

        step = sequences_tree.createElement("step_" + str(step_amount))

        color = sequences_tree.createElement("color")

        color.appendChild(sequences_tree.createTextNode("RGB"))
        step.appendChild(color)

        red = sequences_tree.createElement("Red")
        red.appendChild(sequences_tree.createTextNode("255"))
        step.appendChild(red)

        green = sequences_tree.createElement("Green")
        green.appendChild(sequences_tree.createTextNode("255"))
        step.appendChild(green)

        blue = sequences_tree.createElement("Blue")
        blue.appendChild(sequences_tree.createTextNode("255"))
        step.appendChild(blue)

        light_mode = sequences_tree.createElement("light_mode")
        light_mode.appendChild(sequences_tree.createTextNode("Normal"))
        step.appendChild(light_mode)

        sequences_tree.getElementsByTagName("sequence_" + str(value))[0].appendChild(step)

        pretty_save("sequences.xml", sequences_tree)

        return jsonify(success=True, reload_sequence=reload_sequence, reload_steps_list=True)

    # remove node list with a tag name "step_" + str(value), and change tag names to maintain orde
    if setting_name == "remove_step":

        second_value = int(second_value)
        second_value += 1

        sequences_tree = minidom.parse("sequences.xml")

        # removing step node
        nodes = sequences_tree.getElementsByTagName("sequence_" + str(value))[0].getElementsByTagN
```

```
                "step_" + str(second_value))
            for node in nodes:
                parent = node.parentNode
                parent.removeChild(node)

            # changing nodes tag names
            i = 1
            for step in sequences_tree.getElementsByTagName("sequence_" + str(value))[0].childNodes:
                if (step.nodeType == 1 and step.tagName != "settings"):
                    sequences_tree.getElementsByTagName("sequence_" + str(value))[0].getElementsByTagN
                        0].tagName = "step_" + str(i)
                    i += 1

            pretty_save("sequences.xml", sequences_tree)

            return jsonify(success=True, reload_sequence=reload_sequence)

        # saving current led settings as sequence step
        if setting_name == "save_led_settings_to_step" and second_value != "":

            # remove node and child under "sequence_" + str(value) and "step_" + str(second_value)
            sequences_tree = minidom.parse("sequences.xml")

            second_value = int(second_value)
            second_value += 1

            nodes = sequences_tree.getElementsByTagName("sequence_" + str(value))[0].getElementsByTagN
                "step_" + str(second_value))
            for node in nodes:
                parent = node.parentNode
                parent.removeChild(node)

            # create new step node
            step = sequences_tree.createElement("step_" + str(second_value))

            # load color mode from webinterface.ledsettings and put it into step node
            color_mode = sequences_tree.createElement("color")
            color_mode.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.color_mo
            step.appendChild(color_mode)

            # load mode from webinterface.ledsettings and put it into step node
            mode = sequences_tree.createElement("light_mode")
            mode.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.mode)))
            step.appendChild(mode)

            # if mode is equal "Fading" or "Velocity" load mode from webinterface.ledsettings and put
            if (webinterface.ledsettings.mode == "Fading" or webinterface.ledsettings.mode == "Velocit
                fadingspeed = sequences_tree.createElement("fadingspeed")

                # depending on fadingspeed name set different fadingspeed value
                if (webinterface.ledsettings.fadingspeed == "Slow"):
                    fadingspeed.appendChild(sequences_tree.createTextNode("10"))
                elif (webinterface.ledsettings.fadingspeed == "Medium"):
                    fadingspeed.appendChild(sequences_tree.createTextNode("20"))
                elif (webinterface.ledsettings.fadingspeed == "Fast"):
                    fadingspeed.appendChild(sequences_tree.createTextNode("40"))
                elif (webinterface.ledsettings.fadingspeed == "Very fast"):
                    fadingspeed.appendChild(sequences_tree.createTextNode("50"))
                elif (webinterface.ledsettings.fadingspeed == "Instant"):
                    fadingspeed.appendChild(sequences_tree.createTextNode("1000"))
                elif (webinterface.ledsettings.fadingspeed == "Very slow"):
                    fadingspeed.appendChild(sequences_tree.createTextNode("2"))

                step.appendChild(fadingspeed)

            # if color_mode is equal to "Single" load color from webinterface.ledsettings and put it i
            if (webinterface.ledsettings.color_mode == "Single"):
                red = sequences_tree.createElement("Red")
                red.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.red)))
                step.appendChild(red)

                green = sequences_tree.createElement("Green")
                green.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.green)))
                step.appendChild(green)

                blue = sequences_tree.createElement("Blue")
                blue.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.blue)))
                step.appendChild(blue)

            # if color_mode is equal to "Multicolor" load colors from webinterface.ledsettings and put
            if (webinterface.ledsettings.color_mode == "Multicolor"):
                # load value from webinterface.ledsettings.multicolor
                multicolor = webinterface.ledsettings.multicolor

                # loop through multicolor object and add each color to step node under "sequence_"+str
                for i in range(len(multicolor)):
                    color = sequences_tree.createElement("color_" + str(i + 1))
                    new_multicolor = str(multicolor[i])
                    new_multicolor = new_multicolor.replace("[", "")
                    new_multicolor = new_multicolor.replace("]", "")

                    color.appendChild(sequences_tree.createTextNode(new_multicolor))
                    step.appendChild(color)

                # same as above but with multicolor_range and "color_range_"+str(i)
                multicolor_range = webinterface.ledsettings.multicolor_range
                for i in range(len(multicolor_range)):
                    color_range = sequences_tree.createElement("color_range_" + str(i + 1))
                    new_multicolor_range = str(multicolor_range[i])

                    new_multicolor_range = new_multicolor_range.replace("[", "")
                    new_multicolor_range = new_multicolor_range.replace("]", "")
                    color_range.appendChild(sequences_tree.createTextNode(new_multicolor_range))
                    step.appendChild(color_range)

            # if color_mode is equal to "Rainbow" load colors from webinterface.ledsettings and put it
            if (webinterface.ledsettings.color_mode == "Rainbow"):
                # load values rainbow_offset, rainbow_scale and rainbow_timeshift from webinterface.le
                rainbow_offset = sequences_tree.createElement("Offset")
                rainbow_offset.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.
                step.appendChild(rainbow_offset)

                rainbow_scale = sequences_tree.createElement("Scale")
                rainbow_scale.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings.r
                step.appendChild(rainbow_scale)

                rainbow_timeshift = sequences_tree.createElement("Timeshift")
                rainbow_timeshift.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.rainbow_timeshift)))
                step.appendChild(rainbow_timeshift)

            # if color_mode is equal to "Speed" load colors from webinterface.ledsettings and put it i
            if (webinterface.ledsettings.color_mode == "Speed"):
                # load values speed_slowest["red"] etc from webinterface.ledsettings and put them unde
                speed_slowest_red = sequences_tree.createElement("speed_slowest_red")
                speed_slowest_red.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_slowest["red"])))
                step.appendChild(speed_slowest_red)

                speed_slowest_green = sequences_tree.createElement("speed_slowest_green")
                speed_slowest_green.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_slowest["green"])
                step.appendChild(speed_slowest_green)

                speed_slowest_blue = sequences_tree.createElement("speed_slowest_blue")
                speed_slowest_blue.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_slowest["blue"]))
                step.appendChild(speed_slowest_blue)

                # same as above but with "fastest"
                speed_fastest_red = sequences_tree.createElement("speed_fastest_red")
                speed_fastest_red.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_fastest["red"])))
                step.appendChild(speed_fastest_red)

                speed_fastest_green = sequences_tree.createElement("speed_fastest_green")
                speed_fastest_green.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_fastest["green"])
                step.appendChild(speed_fastest_green)

                speed_fastest_blue = sequences_tree.createElement("speed_fastest_blue")
                speed_fastest_blue.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_fastest["blue"]))
                step.appendChild(speed_fastest_blue)

                # load "speed_max_notes" and "speed_period_in_seconds" values from webinterface.ledset
                # and put them under speed_max_notes and speed_period_in_seconds

                speed_max_notes = sequences_tree.createElement("speed_max_notes")
                speed_max_notes.appendChild(sequences_tree.createTextNode(str(webinterface.ledsettings
                step.appendChild(speed_max_notes)

                speed_period_in_seconds = sequences_tree.createElement("speed_period_in_seconds")
                speed_period_in_seconds.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.speed_period_in_seconds
                step.appendChild(speed_period_in_seconds)

            # if color_mode is equal to "Gradient" load colors from webinterface.ledsettings and put i
            if (webinterface.ledsettings.color_mode == "Gradient"):
                # load values gradient_start_red etc from webinterface.ledsettings and put them under
                gradient_start_red = sequences_tree.createElement("gradient_start_red")
                gradient_start_red.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.gradient_start["red"]))
                step.appendChild(gradient_start_red)

                gradient_start_green = sequences_tree.createElement("gradient_start_green")
                gradient_start_green.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.gradient_start["green"]
                step.appendChild(gradient_start_green)

                gradient_start_blue = sequences_tree.createElement("gradient_start_blue")
                gradient_start_blue.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.gradient_start["blue"])
                step.appendChild(gradient_start_blue)

                # same as above but with gradient_end
                gradient_end_red = sequences_tree.createElement("gradient_end_red")
                gradient_end_red.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.gradient_end["red"])))
                step.appendChild(gradient_end_red)

                gradient_end_green = sequences_tree.createElement("gradient_end_green")
                gradient_end_green.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.gradient_end["green"]))
                step.appendChild(gradient_end_green)

                gradient_end_blue = sequences_tree.createElement("gradient_end_blue")
                gradient_end_blue.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.gradient_end["blue"])))
                step.appendChild(gradient_end_blue)

            # if color_mode is equal to "Scale" load colors from webinterface.ledsettings and put it i
            if (webinterface.ledsettings.color_mode == "Scale"):
                # load values key_in_scale_red etc from webinterface.ledsettings and put them under ke
                key_in_scale_red = sequences_tree.createElement("key_in_scale_red")
                key_in_scale_red.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.key_in_scale["red"])))
                step.appendChild(key_in_scale_red)

                key_in_scale_green = sequences_tree.createElement("key_in_scale_green")
                key_in_scale_green.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.key_in_scale["green"]))
                step.appendChild(key_in_scale_green)

                key_in_scale_blue = sequences_tree.createElement("key_in_scale_blue")
                key_in_scale_blue.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.key_in_scale["blue"])))
                step.appendChild(key_in_scale_blue)

                # same as above but with key_not_in_scale
                key_not_in_scale_red = sequences_tree.createElement("key_not_in_scale_red")
                key_not_in_scale_red.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.key_not_in_scale["red"]
                step.appendChild(key_not_in_scale_red)

                key_not_in_scale_green = sequences_tree.createElement("key_not_in_scale_green")
                key_not_in_scale_green.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.key_not_in_scale["green
                step.appendChild(key_not_in_scale_green)

                key_not_in_scale_blue = sequences_tree.createElement("key_not_in_scale_blue")
                key_not_in_scale_blue.appendChild(
                    sequences_tree.createTextNode(str(webinterface.ledsettings.key_not_in_scale["blue"
                step.appendChild(key_not_in_scale_blue)

            try:
                sequences_tree.getElementsByTagName("sequence_" + str(value))[
                    0].insertBefore(step,
                                    sequences_tree.getElementsByTagName("sequence_" + str(value))[
                                        0].getElementsByTagName("step_" + str(second_value + 1))[0])
            except:
                sequences_tree.getElementsByTagName("sequence_" + str(value))[0].appendChild(step)

            pretty_save("sequences.xml", sequences_tree)

```

```python
            return jsonify(success=True, reload_sequence=reload_sequence, reload_steps_list=True)

        if setting_name == "screen_on":
            if (int(value) == 0):
                webinterface.menu.disable_screen()
            else:
                webinterface.menu.enable_screen()

        if setting_name == "reset_to_default":
            webinterface.usersettings.reset_to_default()

        if setting_name == "restart_rpi":
            call("sudo /sbin/reboot now", shell=True)

        if setting_name == "turnoff_rpi":
            call("sudo /sbin/shutdown -h now", shell=True)

        if setting_name == "update_rpi":
            call("sudo git reset --hard HEAD", shell=True)
            call("sudo git checkout .", shell=True)
            call("sudo git clean -fdx", shell=True)
            call("sudo git pull origin master", shell=True)

        if setting_name == "connect_ports":
            webinterface.midiports.connectall()
            return jsonify(success=True, reload_ports=True)

        if setting_name == "disconnect_ports":
            call("sudo aconnect -x", shell=True)
            return jsonify(success=True, reload_ports=True)

        if setting_name == "restart_rtp":
            call("sudo systemctl restart rtpmidid", shell=True)

        if setting_name == "start_recording":
            webinterface.saving.start_recording()
            return jsonify(success=True, reload_songs=True)

        if setting_name == "cancel_recording":
            webinterface.saving.cancel_recording()
            return jsonify(success=True, reload_songs=True)

        if setting_name == "save_recording":
            now = datetime.datetime.now()
            current_date = now.strftime("%Y-%m-%d %H:%M")
            webinterface.saving.save(current_date)
            return jsonify(success=True, reload_songs=True)

        if setting_name == "change_song_name":
```

```
        if os.path.exists("Songs/" + second_value):
            return jsonify(success=False, reload_songs=True, error=second_value + " already exists

        if "_main" in value:
            search_name = value.replace("_main.mid", "")
            for fname in os.listdir('Songs'):
                if search_name in fname:
                    new_name = second_value.replace(".mid", "") + fname.replace(search_name, "")
                    os.rename('Songs/' + fname, 'Songs/' + new_name)
        else:
            os.rename('Songs/' + value, 'Songs/' + second_value)
            os.rename('Songs/cache/' + value + ".p", 'Songs/cache/' + second_value + ".p")



        return jsonify(success=True, reload_songs=True)

    if setting_name == "remove_song":
        if "_main" in value:
            name_no_suffix = value.replace("_main.mid", "")
            for fname in os.listdir('Songs'):
                if name_no_suffix in fname:
                    os.remove("Songs/" + fname)
        else:
            os.remove("Songs/" + value)

            file_types = [".musicxml", ".xml", ".mxl", ".abc"]
            for file_type in file_types:
                try:
                    os.remove("Songs/" + value.replace(".mid", file_type))
                except:
                    pass

            try:
                os.remove("Songs/cache/" + value + ".p")
            except:
                print("No cache file for " + value)

        return jsonify(success=True, reload_songs=True)

    if setting_name == "download_song":
        if "_main" in value:
            zipObj = ZipFile("Songs/" + value.replace(".mid", "") + ".zip", 'w')
            name_no_suffix = value.replace("_main.mid", "")
            songs_count = 0
            for fname in os.listdir('Songs'):
                if name_no_suffix in fname and ".zip" not in fname:
                    songs_count += 1
                    zipObj.write("Songs/" + fname)
            zipObj.close()
            if songs_count == 1:
                os.remove("Songs/" + value.replace(".mid", "") + ".zip")
                return send_file("../Songs/" + value, mimetype='application/x-csv', attachment_fil
                                 as_attachment=True)
            else:
                return send_file("../Songs/" + value.replace(".mid", "") + ".zip", mimetype='appli
                                 attachment_filename=value.replace(".mid", "") + ".zip", as_attach
        else:
            return send_file("../Songs/" + value, mimetype='application/x-csv', attachment_filenam
                             as_attachment=True)

    if setting_name == "download_sheet_music":
        file_types = [".musicxml", ".xml", ".mxl", ".abc"]
        i = 0
        while i < len(file_types):
            try:
                new_name = value.replace(".mid", file_types[i])
                return send_file("../Songs/" + new_name, mimetype='application/x-csv', attachment_
                                 as_attachment=True)
            except:
                i += 1
        webinterface.learning.convert_midi_to_abc(value)
        try:
            return send_file("../Songs/" + value.replace(".mid", ".abc"), mimetype='application/x-
                             attachment_filename=value.replace(".mid", ".abc"), as_attachment=True
        except:
            print("Converting failed")


    if setting_name == "start_midi_play":
        webinterface.saving.t = threading.Thread(target=play_midi, args=(value, webinterface.midip
                                                                         webinterface.saving, webi
                                                                         webinterface.ledsettings,
                                                                         webinterface.ledstrip))
        webinterface.saving.t.start()

        return jsonify(success=True, reload_songs=True)

    if setting_name == "stop_midi_play":
        webinterface.saving.is_playing_midi.clear()
        fastColorWipe(webinterface.ledstrip.strip, True, webinterface.ledsettings)

        return jsonify(success=True, reload_songs=True)

    if setting_name == "learning_load_song":
        webinterface.learning.t = threading.Thread(target=webinterface.learning.load_midi, args=(v
        webinterface.learning.t.start()

```

```python
            return jsonify(success=True, reload_learning_settings=True)

        if setting_name == "start_learning_song":
            webinterface.learning.t = threading.Thread(target=webinterface.learning.learn_midi)
            webinterface.learning.t.start()

            return jsonify(success=True)

        if setting_name == "stop_learning_song":
            webinterface.learning.is_started_midi = False
            fastColorWipe(webinterface.ledstrip.strip, True, webinterface.ledsettings)

            return jsonify(success=True)

        if setting_name == "change_practice":
            value = int(value)
            webinterface.learning.practice = value
            webinterface.learning.practice = clamp(webinterface.learning.practice, 0, len(webinterface
            webinterface.usersettings.change_setting_value("practice", webinterface.learning.practice)

            return jsonify(success=True)

        if setting_name == "change_tempo":
            value = int(value)
            webinterface.learning.set_tempo = value
            webinterface.learning.set_tempo = clamp(webinterface.learning.set_tempo, 10, 200)
            webinterface.usersettings.change_setting_value("set_tempo", webinterface.learning.set_temp

            return jsonify(success=True)

        if setting_name == "change_hands":
            value = int(value)
            webinterface.learning.hands = value
            webinterface.learning.hands = clamp(webinterface.learning.hands, 0, len(webinterface.learn
            webinterface.usersettings.change_setting_value("hands", webinterface.learning.hands)

            return jsonify(success=True)

        if setting_name == "change_mute_hand":
            value = int(value)
            webinterface.learning.mute_hand = value
            webinterface.learning.mute_hand = clamp(webinterface.learning.mute_hand, 0, len(webinterfa
            webinterface.usersettings.change_setting_value("mute_hand", webinterface.learning.mute_han

            return jsonify(success=True)

        if setting_name == "learning_start_point":
            value = int(value)
            webinterface.learning.start_point = value
            webinterface.learning.start_point = clamp(webinterface.learning.start_point, 0, webinterfa
            webinterface.usersettings.change_setting_value("start_point", webinterface.learning.start_
            webinterface.learning.restart_learning()

            return jsonify(success=True)

        if setting_name == "learning_end_point":
            value = int(value)
            webinterface.learning.end_point = value
            webinterface.learning.end_point = clamp(webinterface.learning.end_point, webinterface.lear
            webinterface.usersettings.change_setting_value("end_point", webinterface.learning.end_poin
            webinterface.learning.restart_learning()

            return jsonify(success=True)

        if setting_name == "set_current_time_as_start_point":
            webinterface.learning.start_point = round(float(webinterface.learning.current_idx * 100 /
            webinterface.learning.start_point = clamp(webinterface.learning.start_point, 0, webinterfa
            webinterface.usersettings.change_setting_value("start_point", webinterface.learning.start_
            webinterface.learning.restart_learning()

            return jsonify(success=True, reload_learning_settings=True)

        if setting_name == "set_current_time_as_end_point":
            webinterface.learning.end_point = round(float(webinterface.learning.current_idx * 100 / fl
            webinterface.learning.end_point = clamp(webinterface.learning.end_point, webinterface.lear
            webinterface.usersettings.change_setting_value("end_point", webinterface.learning.end_poin
            webinterface.learning.restart_learning()

            return jsonify(success=True, reload_learning_settings=True)

        if setting_name == "change_handL_color":
            value = int(value)
            webinterface.learning.hand_colorL += value
            webinterface.learning.hand_colorL = clamp(webinterface.learning.hand_colorL, 0, len(webint
            webinterface.usersettings.change_setting_value("hand_colorL", webinterface.learning.hand_c

            return jsonify(success=True, reload_learning_settings=True)

        if setting_name == "change_handR_color":
            value = int(value)
            webinterface.learning.hand_colorR += value
            webinterface.learning.hand_colorR = clamp(webinterface.learning.hand_colorR, 0, len(webint
            webinterface.usersettings.change_setting_value("hand_colorR", webinterface.learning.hand_c

            return jsonify(success=True, reload_learning_settings=True)

        if setting_name == "change_learning_loop":
            value = int(value == 'true')
            webinterface.learning.is_loop_active = value
            webinterface.usersettings.change_setting_value("is_loop_active", webinterface.learning.is_

            return jsonify(success=True)


        return jsonify(success=True)


    @webinterface.route('/api/get_sequence_setting', methods=['GET'])
    def get_sequence_setting():
        response = {}

        color_mode = webinterface.ledsettings.color_mode

        light_mode = webinterface.ledsettings.mode

        fading_speed = webinterface.ledsettings.fadingspeed

        red = webinterface.ledsettings.red
        green = webinterface.ledsettings.green
        blue = webinterface.ledsettings.blue
        led_color = wc.rgb_to_hex((int(red), int(green), int(blue)))

        multicolor = webinterface.ledsettings.multicolor
        multicolor_range = webinterface.ledsettings.multicolor_range

        rainbow_scale = webinterface.ledsettings.rainbow_scale
        rainbow_offset = webinterface.ledsettings.rainbow_offset
        rainbow_timeshift = webinterface.ledsettings.rainbow_timeshift

        speed_slowest_red = webinterface.ledsettings.speed_slowest["red"]
        speed_slowest_green = webinterface.ledsettings.speed_slowest["green"]
        speed_slowest_blue = webinterface.ledsettings.speed_slowest["blue"]
        speed_slowest_color = wc.rgb_to_hex((int(speed_slowest_red), int(speed_slowest_green), int(spe
        response["speed_slowest_color"] = speed_slowest_color

        speed_fastest_red = webinterface.ledsettings.speed_fastest["red"]
        speed_fastest_green = webinterface.ledsettings.speed_fastest["green"]
        speed_fastest_blue = webinterface.ledsettings.speed_fastest["blue"]
        speed_fastest_color = wc.rgb_to_hex((int(speed_fastest_red), int(speed_fastest_green), int(spe
        response["speed_fastest_color"] = speed_fastest_color

        gradient_start_red = webinterface.ledsettings.gradient_start["red"]
        gradient_start_green = webinterface.ledsettings.gradient_start["green"]
        gradient_start_blue = webinterface.ledsettings.gradient_start["blue"]
        gradient_start_color = wc.rgb_to_hex((int(gradient_start_red), int(gradient_start_green), int(
        response["gradient_start_color"] = gradient_start_color

        gradient_end_red = webinterface.ledsettings.gradient_end["red"]
        gradient_end_green = webinterface.ledsettings.gradient_end["green"]
        gradient_end_blue = webinterface.ledsettings.gradient_end["blue"]
        gradient_end_color = wc.rgb_to_hex((int(gradient_end_red), int(gradient_end_green), int(gradie
        response["gradient_end_color"] = gradient_end_color

        key_in_scale_red = webinterface.ledsettings.key_in_scale["red"]
        key_in_scale_green = webinterface.ledsettings.key_in_scale["green"]
        key_in_scale_blue = webinterface.ledsettings.key_in_scale["blue"]
        key_in_scale_color = wc.rgb_to_hex((int(key_in_scale_red), int(key_in_scale_green), int(key_in
        response["key_in_scale_color"] = key_in_scale_color

        key_not_in_scale_red = webinterface.ledsettings.key_not_in_scale["red"]
        key_not_in_scale_green = webinterface.ledsettings.key_not_in_scale["green"]
        key_not_in_scale_blue = webinterface.ledsettings.key_not_in_scale["blue"]
        key_not_in_scale_color = wc.rgb_to_hex(
            (int(key_not_in_scale_red), int(key_not_in_scale_green), int(key_not_in_scale_blue)))
        response["key_not_in_scale_color"] = key_not_in_scale_color

        response["scale_key"] = webinterface.ledsettings.scale_key

        response["led_color"] = led_color
        response["color_mode"] = color_mode
        response["light_mode"] = light_mode
        response["fading_speed"] = fading_speed
        response["multicolor"] = multicolor
        response["multicolor_range"] = multicolor_range
        response["rainbow_scale"] = rainbow_scale
        response["rainbow_offset"] = rainbow_offset
        response["rainbow_timeshift"] = rainbow_timeshift
        return jsonify(response)


    @webinterface.route('/api/get_settings', methods=['GET'])
    def get_settings():
        response = {}

        red = webinterface.usersettings.get_setting_value("red")
        green = webinterface.usersettings.get_setting_value("green")
        blue = webinterface.usersettings.get_setting_value("blue")
        led_color = wc.rgb_to_hex((int(red), int(green), int(blue)))

        backlight_red = webinterface.usersettings.get_setting_value("backlight_red")
        backlight_green = webinterface.usersettings.get_setting_value("backlight_green")
        backlight_blue = webinterface.usersettings.get_setting_value("backlight_blue")
        backlight_color = wc.rgb_to_hex((int(backlight_red), int(backlight_green), int(backlight_blue)

        sides_red = webinterface.usersettings.get_setting_value("adjacent_red")
        sides_green = webinterface.usersettings.get_setting_value("adjacent_green")
        sides_blue = webinterface.usersettings.get_setting_value("adjacent_blue")
        sides_color = wc.rgb_to_hex((int(sides_red), int(sides_green), int(sides_blue)))

        light_mode = webinterface.usersettings.get_setting_value("mode")
        fading_speed = webinterface.usersettings.get_setting_value("fadingspeed")

        brightness = webinterface.usersettings.get_setting_value("brightness_percent")
        backlight_brightness = webinterface.usersettings.get_setting_value("backlight_brightness_perce

        response["led_color"] = led_color
        response["light_mode"] = light_mode
        response["fading_speed"] = fading_speed

        response["brightness"] = brightness
        response["backlight_brightness"] = backlight_brightness
        response["backlight_color"] = backlight_color

        response["sides_color_mode"] = webinterface.usersettings.get_setting_value("adjacent_mode")
        response["sides_color"] = sides_color

        response["input_port"] = webinterface.usersettings.get_setting_value("input_port")
        response["play_port"] = webinterface.usersettings.get_setting_value("play_port")

        response["skipped_notes"] = webinterface.usersettings.get_setting_value("skipped_notes")
        response["note_offsets"] = webinterface.usersettings.get_setting_value("note_offsets")
        response["led_count"] = webinterface.usersettings.get_setting_value("led_count")
        response["led_shift"] = webinterface.usersettings.get_setting_value("shift")
        response["led_reverse"] = webinterface.usersettings.get_setting_value("reverse")

        response["color_mode"] = webinterface.usersettings.get_setting_value("color_mode")

        response["multicolor"] = webinterface.usersettings.get_setting_value("multicolor")
        response["multicolor_range"] = webinterface.usersettings.get_setting_value("multicolor_range")

        response["rainbow_offset"] = webinterface.usersettings.get_setting_value("rainbow_offset")
        response["rainbow_scale"] = webinterface.usersettings.get_setting_value("rainbow_scale")
        response["rainbow_timeshift"] = webinterface.usersettings.get_setting_value("rainbow_timeshift

        speed_slowest_red = webinterface.usersettings.get_setting_value("speed_slowest_red")
        speed_slowest_green = webinterface.usersettings.get_setting_value("speed_slowest_green")
        speed_slowest_blue = webinterface.usersettings.get_setting_value("speed_slowest_blue")
        speed_slowest_color = wc.rgb_to_hex((int(speed_slowest_red), int(speed_slowest_green), int(spe
        response["speed_slowest_color"] = speed_slowest_color

        speed_fastest_red = webinterface.usersettings.get_setting_value("speed_fastest_red")
        speed_fastest_green = webinterface.usersettings.get_setting_value("speed_fastest_green")
        speed_fastest_blue = webinterface.usersettings.get_setting_value("speed_fastest_blue")
        speed_fastest_color = wc.rgb_to_hex((int(speed_fastest_red), int(speed_fastest_green), int(spe
        response["speed_fastest_color"] = speed_fastest_color
```

```python

    gradient_start_red = webinterface.usersettings.get_setting_value("gradient_start_red")
    gradient_start_green = webinterface.usersettings.get_setting_value("gradient_start_green")
    gradient_start_blue = webinterface.usersettings.get_setting_value("gradient_start_blue")
    gradient_start_color = wc.rgb_to_hex((int(gradient_start_red), int(gradient_start_green), int(
    response["gradient_start_color"] = gradient_start_color

    gradient_end_red = webinterface.usersettings.get_setting_value("gradient_end_red")
    gradient_end_green = webinterface.usersettings.get_setting_value("gradient_end_green")
    gradient_end_blue = webinterface.usersettings.get_setting_value("gradient_end_blue")
    gradient_end_color = wc.rgb_to_hex((int(gradient_end_red), int(gradient_end_green), int(gradie
    response["gradient_end_color"] = gradient_end_color

    key_in_scale_red = webinterface.usersettings.get_setting_value("key_in_scale_red")
    key_in_scale_green = webinterface.usersettings.get_setting_value("key_in_scale_green")
    key_in_scale_blue = webinterface.usersettings.get_setting_value("key_in_scale_blue")
    key_in_scale_color = wc.rgb_to_hex((int(key_in_scale_red), int(key_in_scale_green), int(key_in
    response["key_in_scale_color"] = key_in_scale_color

    key_not_in_scale_red = webinterface.usersettings.get_setting_value("key_not_in_scale_red")
    key_not_in_scale_green = webinterface.usersettings.get_setting_value("key_not_in_scale_green")
    key_not_in_scale_blue = webinterface.usersettings.get_setting_value("key_not_in_scale_blue")
    key_not_in_scale_color = wc.rgb_to_hex(
        (int(key_not_in_scale_red), int(key_not_in_scale_green), int(key_not_in_scale_blue)))
    response["key_not_in_scale_color"] = key_not_in_scale_color

    response["scale_key"] = webinterface.usersettings.get_setting_value("scale_key")

    response["speed_max_notes"] = webinterface.usersettings.get_setting_value("speed_max_notes")
    response["speed_period_in_seconds"] = webinterface.usersettings.get_setting_value("speed_perio

    return jsonify(response)


@webinterface.route('/api/get_recording_status', methods=['GET'])
def get_recording_status():
    response = {}
    response["input_port"] = webinterface.usersettings.get_setting_value("input_port")
    response["play_port"] = webinterface.usersettings.get_setting_value("play_port")

    response["isrecording"] = webinterface.saving.isrecording

    response["isplaying"] = webinterface.saving.is_playing_midi

    return jsonify(response)

@webinterface.route('/api/get_learning_status', methods=['GET'])
def get_learning_status():
    response = {}
    response["loading"] = webinterface.learning.loading
    response["practice"] = webinterface.usersettings.get_setting_value("practice")
    response["hands"] = webinterface.usersettings.get_setting_value("hands")
    response["mute_hand"] = webinterface.usersettings.get_setting_value("mute_hand")
    response["start_point"] = webinterface.usersettings.get_setting_value("start_point")
    response["end_point"] = webinterface.usersettings.get_setting_value("end_point")
    response["set_tempo"] = webinterface.usersettings.get_setting_value("set_tempo")
    response["hand_colorR"] = webinterface.usersettings.get_setting_value("hand_colorR")
    response["hand_colorL"] = webinterface.usersettings.get_setting_value("hand_colorL")
    response["hand_colorList"] = ast.literal_eval(webinterface.usersettings.get_setting_value("han
    response["is_loop_active"] = ast.literal_eval(webinterface.usersettings.get_setting_value("is_

    return jsonify(response)


@webinterface.route('/api/get_songs', methods=['GET'])
def get_songs():
    page = request.args.get('page')
    page = int(page) - 1
    length = request.args.get('length')
    sortby = request.args.get('sortby')
    search = request.args.get('search')

    start = int(page) * int(length)

    songs_list_dict = {}

    path = 'Songs/'
    songs_list = os.listdir(path)
    songs_list = [os.path.join(path, i) for i in songs_list]

    songs_list = sorted(songs_list, key=os.path.getmtime)

    if sortby == "dateAsc":
        songs_list.reverse()

    if sortby == "nameAsc":
        songs_list.sort()

    if sortby == "nameDesc":
        songs_list.sort(reverse=True)

    i = 0
    total_songs = 0

    for song in songs_list:
        if "_#" in song or not song.endswith('.mid'):
            continue
        if search:
            if search.lower() not in song.lower():
                continue
        total_songs += 1

    max_page = int(math.ceil(total_songs / int(length)))

    for song in songs_list:
        song = song.replace("Songs/", "")
        date = os.path.getmtime("Songs/" + song)
        if "_#" in song or not song.endswith('.mid'):
            continue

        if search:
            if search.lower() not in song.lower():
                continue

        i += 1
        if (i > int(start)):
            songs_list_dict[song] = date

        if len(songs_list_dict) >= int(length):
            break

    return render_template('songs_list.html', len=len(songs_list_dict), songs_list_dict=songs_list
                           max_page=max_page, total_songs=total_songs)


@webinterface.route('/api/get_ports', methods=['GET'])
def get_ports():
    ports = mido.get_input_names()
    ports = list(dict.fromkeys(ports))
    response = {}
    response["ports_list"] = ports
    response["input_port"] = webinterface.usersettings.get_setting_value("input_port")
    response["secondary_input_port"] = webinterface.usersettings.get_setting_value("secondary_inpu
    response["play_port"] = webinterface.usersettings.get_setting_value("play_port")
    response["connected_ports"] = str(subprocess.check_output(["aconnect", "-i", "-l"]))

    return jsonify(response)


@webinterface.route('/api/switch_ports', methods=['GET'])
def switch_ports():
    active_input = webinterface.usersettings.get_setting_value("input_port")
    secondary_input = webinterface.usersettings.get_setting_value("secondary_input_port")
    webinterface.midiports.change_port("inport", secondary_input)
    webinterface.usersettings.change_setting_value("secondary_input_port", active_input)
    webinterface.usersettings.change_setting_value("input_port", secondary_input)

```

```python
    fastColorWipe(webinterface.ledstrip.strip, True, webinterface.ledsettings)

    return jsonify(success=True)


@webinterface.route('/api/get_sequences', methods=['GET'])
def get_sequences():
    response = {}
    sequences_list = []
    sequences_tree = minidom.parse("sequences.xml")
    i = 0
    while True:
        try:
            i += 1
            sequences_list.append(
                sequences_tree.getElementsByTagName("sequence_" + str(i))[0].getElementsByTagName(
                    "sequence_name")[
                    0].firstChild.nodeValue)
        except:
            break
    response["sequences_list"] = sequences_list
    response["sequence_number"] = webinterface.ledsettings.sequence_number

    return jsonify(response)


@webinterface.route('/api/get_steps_list', methods=['GET'])
def get_steps_list():
    response = {}
    sequence = request.args.get('sequence')
    sequences_tree = minidom.parse("sequences.xml")
    steps_list = []
    i = 0

    for step in sequences_tree.getElementsByTagName("sequence_" + str(sequence))[0].childNodes:
        if (step.nodeType == 1):
            if (step.nodeName == "settings"):
                response["control_number"] = step.getElementsByTagName("control_number")[0].firstC
                response["next_step"] = step.getElementsByTagName("next_step")[0].firstChild.nodeV
            else:
                steps_list.append(step.nodeName)

    response["steps_list"] = steps_list
    return jsonify(response)


@webinterface.route('/api/set_step_properties', methods=['GET'])
def set_step_properties():
    sequence = request.args.get('sequence')
    step = request.args.get('step')
    webinterface.ledsettings.set_sequence(sequence, step, True)

    return jsonify(success=True)


def pretty_print(dom):
    return '\n'.join([line for line in dom.toprettyxml(indent=' ' * 4).split('\n') if line.strip()


def pretty_save(file_path, sequences_tree):
    with open(file_path, "w", encoding="utf8") as outfile:
        outfile.write(pretty_print(sequences_tree))
```