```python
# This file specifies the logic to be executed when a user requests a known url from the webserver.
# It's arranged in functions that are called when the route parameter in the decorator of the function
# is requested from the client.

# The following imports are mostly tools flask provide to build a webserver which is capable of
# responding to users POST and GET requests and build custom html templates.
import os
import secrets
from datetime import datetime, timedelta
from collections import OrderedDict
from PIL import Image
from flask import (render_template, url_for, flash, redirect, request, abort,
                   Markup)
from projectCode import app, db, bcrypt, mail
from projectCode.forms import (RegistrationForm, LoginForm, UpdateAccountForm,
                               PostForm, RequestResetForm, ResetPasswordForm,
                               ClassForm, StudentForm, AddStudentToClass,
                               RemoveStudentFromClass, TopicForm, HomeworkForm,
                               TestForm, CommentForm, CourseForm, ExamForm,
                               HomeworkMarkForm, TestMarkForm, ExamMarkForm,
                               SearchForm)
from projectCode.models import (User, Post, Class, Student, Topic, Homework,
                               Test, Comment, Course, Exam, HomeworkMark,
                               TestMark, ExamMark)
from flask_login import login_user, current_user, logout_user, login_required
from flask_mail import Message


class GraphDataset():
    def __init__(self, label):
        self.label = label
        self.data = []


class Graph():
    def __init__(self, title, type):
        self.title = title
        self.type = type
        self.labels = []
        self.datasets = []


# DEPRECATED
def monthList(dates):
    start, end = [datetime.strptime(_, "%Y-%m-%d") for _ in dates]
    total_months = lambda dt: dt.month + 12 * dt.year
    mlist = []
    for tot_m in range(total_months(start)-1, total_months(end)):
        y, m = divmod(tot_m, 12)
        mlist.append(datetime(y, m+1, 1).strftime("%b-%y"))
    return mlist


def exMonthList(start, end):
    # start, end = [datetime.strptime(_, "%Y-%m-%d") for _ in dates]
    total_months = lambda dt: dt.month + 12 * dt.year
    mlist = []
    for tot_m in range(total_months(start)-1, total_months(end)):
        y, m = divmod(tot_m, 12)
        mlist.append(datetime(y, m+1, 1).strftime("%b-%y"))
    return mlist



# Any route which has a @login_required() decorator will check if the current user has a recognised
# session id, if so it will attempt to log them into the specified session. If the user isnt
# authenticated they will be redirected to a login page.

# The home route, which is also the default url when the site is accessed will fetch posts from users
# and display them. The program will organise the posts in pages with the ability to paginate between
# them. The home.html template is then built by flask using the posts passed as a parameter and the
# logic defined in the html page. Flask uses ********jinger to build custom templates in html, then
# serve them to the client, the logic for the custom templates is specified in the html page iself.
@app.route("/", methods=['GET', 'POST'])
@app.route("/home", methods=['GET', 'POST'])
def home():
    # Initializing the search and comment forms...
    s_form = SearchForm()
    c_form = CommentForm()
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.date_posted.desc())

    # If a post request is recieved then the following if statement will run...
    srch = 0
    if s_form.validate_on_submit():
        # Getting the search results...
        search_results = posts.filter(Post.title.like('%' + s_form.search_query.data + '%'))
        search_user_results = User.query.filter(User.username.like('%' + s_form.search_query.data + '%'))
        srch = 1 # Search has been performed and results will be shown.
    else:
        search_results = []
        search_user_results = []

    posts = posts.paginate(page=page, per_page=5)
    return render_template('home.html',
                    posts=posts,
                    form=c_form,
                    s_form=s_form,
                    srch=srch,
                    search_results=search_results,
                    search_user_results=search_user_results)

# The dash route contains all the calculation and response for the data displayed
# in the dashboard page.
@app.route("/dash")
def dash():

    courses = Course.query.filter_by(teacher=current_user)

    cur_tops = []
    for course in courses:
        for topic in course.topics:
            if datetime.utcnow() > topic.start_date and datetime.utcnow() < topic.end_date:
                cur_tops.append([course, topic])

    upcoming_homeworks, due_homeworks = [], []
    for cur_top in cur_tops:
        for homework in cur_top[1].homeworks:
            if homework.due_date > datetime.utcnow():
                upcoming_homeworks.append([homework, cur_top[1]])
            elif homework.due_date <= datetime.utcnow():
                due_homeworks.append([homework, cur_top[1]])


    upcoming_tests, due_tests = [], []
    for cur_top in cur_tops:
        for test in cur_top[1].tests:
            if test.date > datetime.utcnow():
                upcoming_tests.append([test, cur_top[1]])
            elif test.date <= datetime.utcnow():
                due_tests.append([test, cur_top[1]])
```

```
cur_top_classes = set()
for cur_top in cur_tops:

    cur_top_classes.update(cur_top[0].classes)


homework_marks = {}
for homework in due_homeworks:
    homework_marks[homework[0].id] = {}

    for xclass in cur_top_classes:

        for student in xclass.students:
            mark = HomeworkMark.query.filter_by(homework_id=homework[0].id).filter_by(student_id=student.id).all()
            homework_marks[homework[0].id][student.id] = mark


            # CHECK IF HANDED IN LATE
            try:
                if len(homework_marks[homework[0].id][student.id]):
                    hmwk = Homework.query.filter_by(id=mark[0].homework_id).first_or_404()
                    if homework_marks[homework[0].id][student.id][0].date_handed_in > hmwk.due_date:
                        print("")
                        homework_marks[homework[0].id][student.id][0].late = True
            except:
                pass



test_marks = {}
for test in due_tests:
    test_marks[test[0].id] = {}

    for xclass in cur_top_classes:

        for student in xclass.students:
            mark = TestMark.query.filter_by(test_id=test[0].id).filter_by(student_id=student.id).all()
            test_marks[test[0].id][student.id] = mark


            # CHECK IF HANDED IN LATE
            try:
                if len(test_marks[test[0].id][student.id]):
                    tst = Test.query.filter_by(id=mark[0].test_id).first_or_404()
                    if test_marks[test[0].id][student.id][0].date_completed > tst.date:
                        print("")
                        test_marks[test[0].id][student.id][0].late = True
            except Exception as e:
                print("PASSING DATE CHECK (PROBABLY HAND IN DATE IS NULL)", test_marks[test[0].id][student.id][0])
                pass


# Class performance in last test graph.
class_perf = Graph("Avg Class Performance On Last Test", "bar")
class_perf_data = {"label":"Avg Class Performance Last Test", "data":[]}

for xclass in cur_top_classes:
    class_perf.labels.append(xclass.class_name)

cls_avgs = []
mrks = []
for test in due_tests:
    for xclass in cur_top_classes:

        for student in xclass.students:
            try:
                mrks.append(TestMark.query.filter_by(test_id=test[0].id).filter_by(student_id=student.id).all()[0].mark)
            except IndexError:
                pass


        try:
            avg = sum(mrks) / len(mrks)
            class_perf_data["data"].append(avg)
        except ZeroDivisionError:
            class_perf_data["data"].append(0)
        mrks = []

class_perf.datasets.append(class_perf_data)


classes_perf_time = Graph("Avg Classes Performance Over Time", "line")

dates = ["2014-10-10", "2016-01-07"]
classes_perf_time.labels = monthList(dates)


courses = db.session.query(Course.id).filter_by(teacher=current_user)
classes = Class.query.filter(Class.course_id.in_(courses)).all()


# Getting date span for graph...
topics_start_asc = Topic.query.filter(Topic.course_id.in_(courses)).order_by(Topic.start_date.asc()).all()
topics_end_desc = Topic.query.filter(Topic.course_id.in_(courses)).order_by(Topic.end_date.desc()).all()

if len(topics_start_asc) and len(topics_end_desc):

    earliest_start = topics_start_asc[0].start_date
    latest_end = topics_end_desc[0].end_date

    dates = [earliest_start, latest_end]
    classes_perf_time.labels = exMonthList(earliest_start, latest_end)


    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses)).all()


    import random

    for xclass in classes:
        class_perf_data = GraphDataset(xclass.class_name)
        class_perf_data = {"label":xclass.class_name, "data":[]}

        for i in range(30):
            class_perf_data["data"].append(random.randint(0,100))

        classes_perf_time.datasets.append(class_perf_data)


# Class performance per assignment test.
classes_perf_per_test = Graph("Avg Classes Performance Over Time", "line")


courses = db.session.query(Course.id).filter_by(teacher=current_user)
topics = Topic.query.filter(Topic.course_id.in_(courses)).all()
classes = Class.query.filter(Class.course_id.in_(courses)).all()

temp_data = []
labels = []
```

```
    for xclass in classes:
        classes_perf_per_test_data = {"label":xclass.class_name, "data":[]}

        for topic in topics:

            for hmwk in topic.homeworks:
                labels.append("HMWK "+topic.name+" "+hmwk.name)

                # Get avg perf
                mrks = []
                for student in xclass.students:
                    try:
                        mrks.append(HomeworkMark.query.filter_by(homework_id=hmwk.id).filter_by(student_id=student.id).all()
[0].mark)
                    except IndexError:
                        pass

                c_sum = 0
                c_len = 0
                for mrk in mrks:
                    try:
                        c_sum += mrk
                        c_len += 1
                    except TypeError:
                        pass


                if c_len:
                    avg = c_sum // c_len
                    classes_perf_per_test_data["data"].append(avg)
                else:
                    classes_perf_per_test_data["data"].append(0)

                mrks = []


            for test in topic.tests:
                labels.append("TEST "+topic.name+" "+test.name)

                # Get avg perf
                mrks = []
                for student in xclass.students:
                    try:
                        mrks.append(TestMark.query.filter_by(test_id=test.id).filter_by(student_id=student.id).all()[0].mark)
                    except IndexError:
                        pass

                c_sum = 0
                c_len = 0
                for mrk in mrks:
                    try:
                        c_sum += mrk
                        c_len += 1
                    except TypeError:
                        pass


                if c_len:
                    avg = c_sum // c_len
                    classes_perf_per_test_data["data"].append(avg)
                else:
                    classes_perf_per_test_data["data"].append(0)

                mrks = []
        classes_perf_per_test.datasets.append(classes_perf_per_test_data)

    # Remove duplicates...
    n_labels = []
    for label in labels:
        if not label in n_labels:
            n_labels.append(label)

    classes_perf_per_test.labels = n_labels


    return render_template('dash.html', cur_tops=cur_tops,
                           upcoming_homeworks=upcoming_homeworks,
                           due_homeworks=due_homeworks,
                           due_tests=due_tests,
                           homework_marks=homework_marks,
                           test_marks=test_marks,
                           class_perf=class_perf,
                           classes_perf_time=classes_perf_time,
                           classes_perf_per_test=classes_perf_per_test,
                           cur_top_classes=cur_top_classes)



# Route to allow user to post the result of a homework via the dashboard.
@app.route('/submit_homework', methods=['POST'])
def submit_homework():
    mark = request.form['mark']
    date_handed_in = request.form['date_handed_in']
    homework_id = request.form['homework_id']
    student_id = request.form['student_id']

    try:
        int(mark)
    except ValueError:
        flash('Mark must be an int.', 'danger')
        return redirect(url_for('dash'))

    homework = Homework.query.first_or_404(homework_id)
    print(mark)
    print(homework.max_mark)
    if int(mark) > homework.max_mark:
        flash('Mark is more than the Homeworks maximum mark.', 'danger')
        return redirect(url_for('dash'))


    homework_mark = HomeworkMark(mark=mark,
                           homework_id=homework_id,
                           student_id=student_id,
                           date_handed_in=datetime.strptime(date_handed_in, '%Y-%m-%d'))

    db.session.add(homework_mark)
    db.session.commit()
    flash('Homework has been marked!', 'success')
    return redirect(url_for('dash'))


# Route to allow user to post the result of a test via the dashboard.
@app.route('/submit_test', methods=['POST'])
def submit_test():
    mark = request.form['mark']
    date_completed = request.form['date_handed_in']
    test_id = request.form['test_id']
    student_id = request.form['student_id']
    try:
        int(mark)
    except ValueError:
        flash('Mark must be an int.', 'danger')
        return redirect(url_for('dash'))

    test = Test.query.first_or_404(test_id)
    if int(mark) > test.max_mark:
        flash('Mark is more than the Tests maximum mark.', 'danger')
        return redirect(url_for('dash'))


    test_mark = TestMark(mark=mark,
                         test_id=test_id,
                         student_id=student_id,
                         date_completed=datetime.strptime(date_completed, '%Y-%m-%d'))

    db.session.add(test_mark)
    db.session.commit()
    flash('Test has been marked!', 'success')
    return redirect(url_for('dash'))

# The comment route is for users to sumbit a comment to a post.
@app.route('/post/<int:post_id>/comment', methods=['POST'])
@login_required
def add_comment(post_id):
    post = Post.query.get_or_404(post_id)
    comment_content = request.form['comment']

    if len(comment_content) > 250:
        flash('Comment is too long!', 'danger')
        return redirect(url_for('post', post_id=post_id))

    if not len(comment_content):
        flash('No comment found!', 'danger')
        return redirect(url_for('post', post_id=post_id))

    comment = Comment(content=comment_content, commenter=current_user, post=post)
    db.session.add(comment)
    db.session.commit()
    flash('Your comment has been posted!', 'success')
    return redirect(url_for('post', post_id=post_id))

# '/post/<int:post_id>/comment/<int:comment_id>/update'
@app.route('/comment/<int:comment_id>/update', methods=['POST'])
@login_required
def update_comment(comment_id):
    comment = Comment.query.get_or_404(comment_id)
    post_id = int(comment.post_id)
    if comment.commenter != current_user:
        abort(403)

    comment_content = request.form['comment']

    print(comment_content)

    if len(comment_content) > 500:
        return "Post is too long!", 201

    if len(comment_content) < 2:
        return "Post content too short!", 201


    comment.content = comment_content
    db.session.commit()
```

```python
    return "Your Post Was Updated Successfully!", 201


@app.route("/post/<int:post_id>/comment/<int:comment_id>/delete", methods=['GET'])
@login_required
def delete_comment(post_id, comment_id):
    comment = Comment.query.get_or_404(comment_id)
    if comment.commenter != current_user:
        abort(403)
    db.session.delete(comment)
    db.session.commit()
    flash('Your comment has been deleted!', 'success')
    return redirect(url_for('post', post_id=post_id))


# The about route simply returns a static about page which explains the system is some detail and
# the setup process.
@app.route("/about")
def about():
    return render_template('about.html', title='About')


# The register route can be called with multiple methods, post and get. It first checks if the user
# is already signed in, if so the user is redirected to the home page. If the user isn't signed in
# then a form is build using the specification in the forms.py file. "form.validate_on_submit()"
# basically checks if the method of requests is POST, it then validates the user input based upon
# the validation rules that are specified in the forms.py file. If the method of request is GET,
# then the system simply returns the register.html page with the necissary options for a user to
# register an account. If validation is successful the user is added to the database.
@app.route("/register", methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = RegistrationForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user = User(username=form.username.data, email=form.email.data, password=hashed_password)
        db.session.add(user)
        db.session.commit()
        flash('Your account has been created! You are now able to log in', 'success')
        return redirect(url_for('login'))
    return render_template('acc_register.html', title='Register', form=form)


# The login route works very simularly to the register route however there is a extra option
# for if the validation and verification of user input is failed. Messages are "flashed" to the
# user if login is successful, or unsuccussful.
@app.route("/login", methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user and bcrypt.check_password_hash(user.password, form.password.data):
            login_user(user, remember=form.remember.data)
            next_page = request.args.get('next')
            return redirect(next_page) if next_page else redirect(url_for('home'))
        else:
            flash('Login Unsuccessful. Please check email and password', 'danger')
    return render_template('acc_login.html', title='Login', form=form)


# Route calls to the "logout_user()" function which simply terminates the users session and
# ensures that their session id is not usable anymore.
@app.route("/logout")
```

```python
def logout():
    logout_user()
    return redirect(url_for('home'))


# This route is for saving a picture when the user wishes to update their profile picture.
def save_picture(form_picture):
    random_hex = secrets.token_hex(8)
    _, f_ext = os.path.splitext(form_picture.filename)
    picture_fn = random_hex + f_ext
    picture_path = os.path.join(app.root_path, 'static/profile_pics', picture_fn)

    output_size = (125, 125)
    i = Image.open(form_picture)
    i.thumbnail(output_size)
    i.save(picture_path)

    return picture_fn


# Route checks if user is authenticated, then gets and displays their account information.
# This method also takes post requests and allows the user to update their account information.
@app.route("/account", methods=['GET', 'POST'])
@login_required
def account():
    form = UpdateAccountForm()
    if form.validate_on_submit():
        if form.picture.data:
            picture_file = save_picture(form.picture.data)
            current_user.image_file = picture_file
        current_user.username = form.username.data
        current_user.email = form.email.data
        db.session.commit()
        flash('Your account has been updated!', 'success')
        return redirect(url_for('account'))
    elif request.method == 'GET':
        form.username.data = current_user.username
        form.email.data = current_user.email
    image_file = url_for('static', filename='profile_pics/' + current_user.image_file)
    return render_template('account.html', title='Account',
                           image_file=image_file, form=form)


# Allows the user to make a new post and post to the server. The GET method, responds with
# a post enter page, and the POST method stores the information to the database.
@app.route("/post/new", methods=['GET', 'POST'])
@login_required
def new_post():
    form = PostForm()
    if form.validate_on_submit():
        post = Post(title=form.title.data, content=form.content.data, author=current_user)
        db.session.add(post)
        db.session.commit()
        flash('Your post has been created!', 'success')
        return redirect(url_for('home'))
    return render_template('new_post.html', title='New Post',
                           form=form, legend='New Post')


# Allows the user to view a specified post, if the user owns the post then they will have
# the option to edit the title or contence of the post while other users can only view the
# contence of the post.
@app.route("/post/<int:post_id>")
def post(post_id):
    form = CommentForm()
    post = Post.query.get_or_404(post_id)
    comments = Comment.query.filter_by(post=post)
    return render_template('single_post.html', title=post.title, form=form, post=post, comments=comments)


# Checks for the user to be authenticated and allows them to update the post if the post
# belongs to them.
@app.route("/post/<int:post_id>/update", methods=['GET', 'POST'])
@login_required
def update_post(post_id):
    post = Post.query.get_or_404(post_id)
    if post.author != current_user:
        abort(403)
    form = PostForm()
    if form.validate_on_submit():
        post.title = form.title.data
        post.content = form.content.data
        db.session.commit()
        flash('Your post has been updated!', 'success')
        return redirect(url_for('post', post_id=post.id))
    elif request.method == 'GET':
        form.title.data = post.title
        form.content.data = post.content
    return render_template('new_post.html', title='Update Post',
                           form=form, legend='Update Post')


# If the post author matches the current user then the post specified in
# the url will be deleted.
@app.route("/post/<int:post_id>/delete", methods=['POST'])
@login_required
def delete_post(post_id):
    post = Post.query.get_or_404(post_id)
    if post.author != current_user:
        abort(403)
    for comment in post.comments:
        db.session.delete(comment)
    db.session.delete(post)
    db.session.commit()
    flash('Your post has been deleted!', 'success')
    return redirect(url_for('home'))


# This route is here to allow any user to see all posts of a specified user. It
# will return all of the posts whos author corrisponds to the user specified.
@app.route("/user/<string:username>")
def user_posts(username):
    page = request.args.get('page', 1, type=int)
    user = User.query.filter_by(username=username).first_or_404()
    posts = Post.query.filter_by(author=user)\
        .order_by(Post.date_posted.desc())\
        .paginate(page=page, per_page=5)
    return render_template('all_user_posts.html', posts=posts, user=user)


# When this function is called by the server, a password reset email will be
# sent to the user specified in the parameter.
def send_reset_email(user):
    token = user.get_reset_token()
    msg = Message('Password Reset Request',
                  sender='noreply@demo.com',
                  recipients=[user.email])
    msg.body = f'''To reset your password, visit the following link:
{url_for('reset_token', token=token, _external=True)}

If you did not make this request then simply ignore this email and no changes will be made.
'''
```

```python
    mail.send(msg)


# Route is used by a user to request a password reset on their account. (Current user is
# to identify the user).
@app.route("/reset_password", methods=['GET', 'POST'])
def reset_request():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = RequestResetForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        send_reset_email(user)
        flash('An email has been sent with instructions to reset your password.', 'info')
        return redirect(url_for('login'))
    return render_template('reset_request.html', title='Reset Password', form=form)


# Checks for a token we can use to identify a user that has requested a password reset.
# If the user has requested a reset and is authenticated.
@app.route("/reset_password/<token>", methods=['GET', 'POST'])
def reset_token(token):
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    user = User.verify_reset_token(token)
    if user is None:
        flash('That is an invalid or expired token', 'warning')
        return redirect(url_for('reset_request'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user.password = hashed_password
        db.session.commit()
        flash('Your password has been updated! You are now able to log in', 'success')
        return redirect(url_for('login'))
    return render_template('reset_token.html', title='Reset Password', form=form)


# Route for creating a new class. If POST method is used then user input is validated and
# class is added to database if all validation rules are passed.
@app.route("/new_class", methods=['GET', 'POST'])
@login_required
def new_class():

    form = ClassForm()

    courses = Course.query.filter_by(teacher=current_user).all()
    form.course_id.choices = [(course.id, course.name) for num, course in enumerate(courses)]

    if not len(courses):
        flash(Markup('To add a class you must first <a href="' + url_for('new_course') + '">add a course</a>.'), 'danger')

    # If user doesn't request for a course, then enable the select box and set the result as the course_id...
    try:
        course_id = request.args['course_id']
        form.course_id.value = course_id
    except KeyError:
        form.course_id.enabled = True
        course_id = form.course_id.data

    if form.validate_on_submit():

        course = Course.query.get_or_404(course_id)
        if course.teacher != current_user:
            abort(403)
        new_class = Class(class_name=form.class_name.data,
                          class_starting_date=form.class_starting_date.data,
                          course=course)

        db.session.add(new_class)
        db.session.commit()
        flash(f'New class has been added to {course.name}!', 'success')

        next_page = request.args.get('next')
        return redirect(next_page) if next_page else redirect(url_for('classes'))

    return render_template('new_class.html', title='Create Class', form=form, legend="Add A Class To Manage")


# Route to allow the user to view all of their classes, the classes are queried from the database and the "teacher" parameter is used
# to ensure only classes belonging to the current user is returned.
@app.route("/classes")
@login_required
def classes():
    page = request.args.get('page', 1, type=int)
    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses)).paginate(page=page, per_page=5)
    return render_template('all_classes.html', title="My Classes", classes=classes, user=current_user)


# Allows the user to view all the details of a specific class who's id will be passed into the url. This is mostly done
# automatically by a button on the interface. If the user is trying to access a class that they do not teach then a 403
# FORBIDDEN responce is issued.
@app.route("/class/<int:class_id>")
@login_required
def xclass(class_id):
    xclass = Class.query.get_or_404(class_id)
    if xclass.course.teacher != current_user:
        abort(403)



    classes_perf_per_test = Graph("Avg Classes Performance Over Time", "line")


    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    topics = Topic.query.filter(Topic.course_id.in_(courses)).all()
    classes = Class.query.filter(Class.course_id.in_(courses)).all()

    temp_data = []
    labels = []



    print(xclass.class_name)
    classes_perf_per_test_data = {"label":xclass.class_name, "data":[]}

    for topic in topics:

        for hmwk in topic.homeworks:
            print(hmwk.name)
            labels.append("HMWK "+topic.name+" "+hmwk.name)

            # Get avg perf
            mrks = []
            for student in xclass.students:
                try:
                    mrks.append(HomeworkMark.query.filter_by(homework_id=hmwk.id).filter_by(student_id=student.id).all()[0].mark)
                except IndexError:
                    pass
            c_sum = 0
            c_len = 0
            for mrk in mrks:
                try:
                    c_sum += mrk
                    c_len += 1
                except TypeError:
                    pass


            if c_len:
                avg = c_sum // c_len
                classes_perf_per_test_data["data"].append(avg)
            else:
                classes_perf_per_test_data["data"].append(0)

            mrks = []


        for test in topic.tests:
            print(test.name)
            labels.append("TEST "+topic.name+" "+test.name)

            # Get avg perf
            mrks = []
            for student in xclass.students:
                try:
                    mrks.append(TestMark.query.filter_by(test_id=test.id).filter_by(student_id=student.id).all()[0].mark)
                except IndexError:
                    pass

            c_sum = 0
            c_len = 0
            for mrk in mrks:
                try:
                    c_sum += mrk
                    c_len += 1
                except TypeError:
                    pass


            if c_len:
                avg = c_sum // c_len
                classes_perf_per_test_data["data"].append(avg)
            else:
                classes_perf_per_test_data["data"].append(0)

            mrks = []



    print("\n\n", classes_perf_per_test_data["data"])



    classes_perf_per_test.datasets.append(classes_perf_per_test_data)

    # Get avg perf

    # Remove duplicates...
    n_labels = []
    for label in labels:
        if not label in n_labels:
            n_labels.append(label)

    classes_perf_per_test.labels = n_labels
```

```python
    print(n_labels)






    return render_template('single_class.html', title=xclass.class_name, xclass=xclass, class_perf=classes_perf_per_test)


# Allows the user to update the details of a class, it returns a form with all current values filled in.
# If the user is trying to update the details of a class that they do not teach then a 403 FORBIDDEN responce is issued.
@app.route("/class/<int:class_id>/update", methods=['GET', 'POST'])
@login_required
def update_class(class_id):
    xclass = Class.query.get_or_404(class_id)
    if xclass.course.teacher != current_user:
        abort(403)
    form = ClassForm()

    courses = Course.query.filter_by(teacher=current_user).all()
    form.course_id.choices = [(course.id, course.name) for num, course in enumerate(courses)]

    if form.validate_on_submit():
        xclass.class_name = form.class_name.data
        xclass.class_starting_date = form.class_starting_date.data
        xclass.course_id = form.course_id.data
        db.session.commit()
        flash('Your class has been updated!', 'success')
        return redirect(url_for('xclass', class_id=xclass.id))
    elif request.method == 'GET':
        form.class_name.data = xclass.class_name
        form.class_starting_date.data = xclass.class_starting_date
        form.course_id.data = xclass.course.id
        form.course_id.enabled = True
    return render_template('new_class.html', title='Update Class',
                           form=form, legend='Update Class')


# An endpoint that only takes a POST request, if the user is the teacher of the class then it will be deleted on request.
@app.route("/class/<int:class_id>/delete", methods=['POST'])
@login_required
def delete_class(class_id):
    xclass = Class.query.get_or_404(class_id)
    if xclass.course.teacher != current_user:
        abort(403)
    db.session.delete(xclass)
    db.session.commit()
    flash('Your class has been deleted!', 'success')
    return redirect(url_for('classes'))


@app.route("/class/<int:class_id>/report")
@login_required
def class_report(class_id):
    xclass = Class.query.get_or_404(class_id)
    if xclass.course.teacher != current_user:
        abort(403)

    pdf = FPDF('P', 'mm', 'A4')
    pdf.add_page()
    pdf.set_font("Arial", size=16)

    top = pdf.y
    pdf.multi_cell(158, 10, txt="Student", align="C")
    pdf.y = top
pdf.set_text_color(200, 0, 0)
pdf.multi_cell(193, 10, txt="Track ", align="C")
pdf.y = top
pdf.set_text_color(0, 0, 0)
pdf.multi_cell(225, 10, txt="Report", align="C")


pdf.set_font("Arial", "B", size=12)
pdf.ln()
pdf.multi_cell(0, 5, "Class Info")
pdf.set_font("Arial", size=10)
pdf.ln(3)
pdf.multi_cell(0, 5, ('Class Name: %s' % xclass.class_name))
pdf.ln(1)
pdf.multi_cell(0, 5, ('Assigned To: %s' % xclass.course.name))
pdf.ln()


pdf.set_font("Arial", "B", size=12)
pdf.ln()
pdf.multi_cell(0, 5, "Students")
pdf.set_font("Arial", size=10)
pdf.ln(3)

for student in xclass.students:
    pdf.multi_cell(0, 5, "- " + student.name)
    pdf.ln(1)
pdf.ln()


pdf.set_font("Arial", "B", size=12)
pdf.ln()
pdf.multi_cell(0, 5, "Class Avg Performance")

pdf.set_font("Arial", size=10)
pdf.ln(3)

temp_data = []
labels = []

data = []

for topic in xclass.course.topics:

    for hmwk in topic.homeworks:
        top = pdf.y
        pdf.multi_cell(0, 5, topic.name +" (Hmwk) "+hmwk.name)

        # Get avg perf
        mrks = []
        for student in xclass.students:
            try:
                mrks.append(HomeworkMark.query.filter_by(homework_id=hmwk.id).filter_by(student_id=student.id).all()[0].mark)
            except IndexError:
                pass

        c_sum = 0
        c_len = 0
        for mrk in mrks:
            try:
                c_sum += mrk
                c_len += 1
            except TypeError:
                pass
            pdf.y = top
            pdf.x = 100

            if c_len:
                avg = c_sum // c_len
                pdf.multi_cell(0, 5, str(avg) + "/" + str(hmwk.max_mark))
            else:
                pdf.multi_cell(0, 5, "No Mark Awarded")

            mrks = []


        for test in topic.tests:
            top = pdf.y
            pdf.multi_cell(0, 5, topic.name+" (Test) "+test.name)
            # labels.append("TEST "+topic.name+" "+test.name)

            # Get avg perf
            mrks = []
            for student in xclass.students:
                try:
                    mrks.append(TestMark.query.filter_by(test_id=test.id).filter_by(student_id=student.id).all()[0].mark)
                except IndexError:
                    pass

            c_sum = 0
            c_len = 0
            for mrk in mrks:
                try:
                    c_sum += mrk
                    c_len += 1
                except TypeError:
                    pass

            pdf.y = top
            pdf.x = 100

            if c_len:
                avg = c_sum // c_len
                pdf.multi_cell(0, 5, str(avg) + "/" + str(test.max_mark))
            else:
                pdf.multi_cell(0, 5, "No Mark Awarded")

            mrks = []


    pdf.output(app.config['PDF_FILE_DUMP'] + "student_report.pdf")

    return send_file("static/pdf_gen/student_report.pdf", attachment_filename='student_report.pdf')


# Route for adding a new student. If POST method is used then user input is validated and
# student is added to database if all validation rules are passed. All the current users classes are queried and
# passed into the form so that the teacher can select a class to add the student to.
@app.route("/new_student", methods=['GET', 'POST'])
@login_required
def new_student():
    form = StudentForm()
    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses))

    if not classes.first():
        flash(Markup('To add a student you must first <a href="' + url_for('new_class') + '">add a class</a>.'), 'danger')

    form.class_id.choices = [(xclass.id, xclass.class_name) for xclass in classes]
    if form.validate_on_submit():
```

```python
        new_student = Student(name=form.name.data,
                              email=form.email.data,
                              address=form.address.data,
                              parent_phone=form.parent_phone.data,
                              predicted_grade=form.predicted_grade.data)
        join_class = Class.query.filter_by(id=form.class_id.data).first_or_404()
        new_student.classes.append(join_class)
        db.session.add(new_student)
        db.session.commit()
        flash('New student has been added!', 'success')
        return redirect(url_for('students'))

    return render_template('new_student.html', title='Add Student', form=form, legend="Add Student")


# Route to allow the user to view all of their students, the students are queried from the database and the "teacher" parameter is used
# to ensure only students that are taught by the current user is returned.
@app.route("/students", methods=['GET', 'POST'])
@login_required
def students():
    page = request.args.get('page', 1, type=int)

    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses))

    students = set()
    for xclass in classes:
        students.update(xclass.students)

    s_form = SearchForm()
    srch = 0
    search_results = []
    if s_form.validate_on_submit():
        print("Search For:", s_form.search_query.data)

        student_ids = [student.id for student in students]

        search_results = Student.query.filter(Student.name.like('%' + s_form.search_query.data + '%'))
        search_results = search_results.filter(Student.id.in_(student_ids))
        srch = 1

    return render_template('all_students.html',
                           title="My Students",
                           students=students,
                           user=current_user,
                           s_form=s_form,
                           search_results=search_results,
                           srch=srch)


# Allows the user to view all the details of a specific student who's id will be passed into the url. This is mostly done
# automatically by a button on the interface. If the user is trying to access the details of a student that they do not
# teach then a 403 FORBIDDEN responce is issued.
@app.route("/student/<int:student_id>")
@login_required
def student(student_id):
    student = Student.query.get_or_404(student_id)
    if student.classes[0].course.teacher != current_user:
        abort(403)


    classes_perf_per_test = Graph("Avg Performance Per Test", "line")


    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    topics = Topic.query.filter(Topic.course_id.in_(courses)).all()
    classes = Class.query.filter(Class.course_id.in_(courses)).all()

    temp_data = []
    labels = []

    classes_perf_per_test_data = {"label":student.name, "data":[]}

    for topic in topics:

        for hmwk in topic.homeworks:
            print(hmwk.name)
            labels.append("HMWK "+topic.name+" "+hmwk.name)

            try:
                mark = HomeworkMark.query.filter_by(homework_id=hmwk.id).filter_by(student_id=student.id).all()[0].mark

                try:
                    int(mark)
                    classes_perf_per_test_data["data"].append(mark)
                except ValueError:
                    classes_perf_per_test_data["data"].append(0)

            except IndexError:
                pass



        for test in topic.tests:
            print(test.name)
            labels.append("TEST "+topic.name+" "+test.name)

            try:
                mark = TestMark.query.filter_by(test_id=test.id).filter_by(student_id=student.id).all()[0].mark

                try:
                    int(mark)
                    classes_perf_per_test_data["data"].append(mark)
                except ValueError:
                    classes_perf_per_test_data["data"].append(0)
            except IndexError:
                pass



    print("\n\n", classes_perf_per_test_data["data"])


    classes_perf_per_test.datasets.append(classes_perf_per_test_data)

    # Remove duplicates...
    n_labels = []
    for label in labels:
        if not label in n_labels:
            n_labels.append(label)

    classes_perf_per_test.labels = n_labels

    print(n_labels)


    return render_template('single_student.html', title=student.name, student=student, class_perf=classes_perf_per_test)


# Allows the user to update the details of a student, it returns a form with all current values filled in.
# If the user is trying to update the details of a student that they do not teach then a 403 FORBIDDEN responce is issued.
```

```python
@app.route("/student/<int:student_id>/update", methods=['GET', 'POST'])
@login_required
def update_student(student_id):
    student = Student.query.get_or_404(student_id)
    if student.classes[0].course.teacher != current_user:
        abort(403)
    form = StudentForm()

    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses))
    form.class_id.choices = [(xclass.id, xclass.class_name) for xclass in classes]
    if form.validate_on_submit():
        join_class = Class.query.filter_by(id=form.class_id.data).first_or_404()
        student.classes[0] = join_class # Only the first class
        student.name = form.name.data
        student.email = form.email.data
        student.address = form.address.data
        student.parent_phone = form.parent_phone.data
        student.predicted_grade = form.predicted_grade.data
        db.session.commit()
        flash('Your class has been updated!', 'success')
        return redirect(url_for('student', student_id=student.id))
    elif request.method == 'GET':
        form.class_id.data = student.classes[0].id # Only the first class
        form.name.data = student.name
        form.email.data = student.email
        form.address.data = student.address
        form.parent_phone.data = student.parent_phone
        form.predicted_grade.data = student.predicted_grade
    return render_template('new_student.html', title='Update Student',
                           form=form, legend='Update Student')


# An endpoint that only takes a POST request, if the user is the teacher of the student then it will be deleted on request.
@app.route("/student/<int:student_id>/delete", methods=['POST'])
@login_required
def delete_student(student_id):
    student = Student.query.get_or_404(student_id)
    if student.classes[0].course.teacher != current_user:
        abort(403)
    db.session.delete(student)
    db.session.commit()
    flash('Your student has been deleted!', 'success')
    return redirect(url_for('students'))


# This endpoint takes the id of a student and allows the user to add them to a class. This is usually used from the student
# page from a button called add to class.
@app.route("/student/<int:student_id>/add_to_class", methods=['GET', 'POST'])
@login_required
def add_to_class(student_id):
    student = Student.query.get_or_404(student_id)
    if student.classes[0].course.teacher != current_user:
        abort(403)
    form = AddStudentToClass()
    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses))
    form.class_id.choices = [(xclass.id, xclass.class_name) for xclass in classes]
    # form.class_id.choices = [(xclass.id, xclass.class_name) for xclass in Class.query.filter_by(teacher=current_user)]
    if form.validate_on_submit():
        join_class = Class.query.filter_by(id=form.class_id.data).first_or_404()
        student.classes.append(join_class)
        db.session.commit()
        flash('Your class has been updated!', 'success')
        return redirect(url_for('student', student_id=student.id))
    elif request.method == 'GET':
        current_classes = student.classes
        for j in range(len(form.class_id.choices)):
            for i, xclass in enumerate(form.class_id.choices):
                for curclass in current_classes:
                    if xclass[0] == curclass.id:
                        del form.class_id.choices[i]

    return render_template('add_student_to_class.html', title='Add Student To Class',
                          form=form, legend='Add Student To Class')


# This endpoint takes the id of a student and allows the user to remove them from a class. This is usually used from the student
# page from a button called remove from class.
@app.route("/student/<int:student_id>/remove_from_class", methods=['GET', 'POST'])
@login_required
def remove_from_class(student_id):
    student = Student.query.get_or_404(student_id)
    if student.classes[0].course.teacher != current_user:
        abort(403)
    form = RemoveStudentFromClass()
    courses = db.session.query(Course.id).filter_by(teacher=current_user)
    classes = Class.query.filter(Class.course_id.in_(courses))
    form.class_id.choices = [(xclass.id, xclass.class_name) for xclass in classes if xclass in student.classes]
    if form.validate_on_submit():
        remove_class = Class.query.filter_by(id=form.class_id.data).first_or_404()

        if not (len(student.classes) > 1):
            flash('You cannot delete the only class a student is assigned to.', 'danger')
            return redirect(url_for('student', student_id=student.id))

        for i, xclass in enumerate(student.classes):
            if xclass.id == remove_class.id:
                del student.classes[i]

        db.session.commit()
        flash('Your class has been updated!', 'success')
        return redirect(url_for('student', student_id=student.id))
    elif request.method == 'GET':
        pass

    return render_template('add_student_to_class.html', title='Remove Student From Class',
                          form=form, legend='Remove Student From Class')



from flask import send_file
from fpdf import FPDF

@app.route("/student/<int:student_id>/report")
@login_required
def student_report(student_id):
    student = Student.query.get_or_404(student_id)
    if student.classes[0].course.teacher != current_user:
        abort(403)

    pdf = FPDF('P', 'mm', 'A4')
    pdf.add_page()
    pdf.set_font("Arial", size=16)

    top = pdf.y
    pdf.multi_cell(158, 10, txt="Student", align="C")
    pdf.y = top
    pdf.set_text_color(200, 0, 0)
    pdf.multi_cell(193, 10, txt="Track ", align="C")
    pdf.y = top
    pdf.set_text_color(0, 0, 0)
    pdf.multi_cell(225, 10, txt="Report", align="C")


    pdf.set_font("Arial", "B", size=12)
    pdf.ln()
    pdf.multi_cell(0, 5, "Student Info")
    pdf.set_font("Arial", size=10)
    pdf.ln(3)
    pdf.multi_cell(0, 5, ('Name: %s' % student.name))
    pdf.ln(1)
    pdf.multi_cell(0, 5, ('Email: %s' % student.email))
    pdf.ln(1)
    pdf.multi_cell(0, 5, ('Address: %s' % student.address))
    pdf.ln(1)
    pdf.multi_cell(0, 5, ('Parent Phone Number: %s' % student.parent_phone))
    pdf.ln(1)
    pdf.multi_cell(0, 5, ('Predicted Grade: %s' % student.predicted_grade))
    pdf.ln()


    pdf.set_font("Arial", "B", size=12)
    pdf.ln()
    pdf.multi_cell(0, 5, "Students Classes")
    pdf.set_font("Arial", size=10)
    pdf.ln(3)

    for xclass in student.classes:
        pdf.multi_cell(0, 5, "- " + xclass.class_name)
        pdf.ln(1)
    pdf.ln()


    pdf.set_font("Arial", "B", size=12)
    pdf.ln()
    pdf.multi_cell(0, 5, "Student Performance")

    pdf.set_font("Arial", "B", size=10)
    pdf.ln()
    pdf.multi_cell(0, 1, "Homework")
    pdf.set_font("Arial", size=10)
    pdf.ln(3)

    # FOR HOMEWORKS
    marks = HomeworkMark.query.filter_by(student_id=student.id).all()

    for mark in marks:
        hmwk = mark.homework

        top = pdf.y
        pdf.multi_cell(0, 5, "("+hmwk.topic.name +") "+hmwk.name)
        pdf.y = top
        pdf.x = 95

        try:
            int(mark.mark)
            if mark.mark != "":
                pdf.multi_cell(0, 5, str(mark.mark) + "/" + str(hmwk.max_mark))
            else:
                pdf.multi_cell(0, 5, "No Mark Awarded")
        except ValueError:
            pdf.multi_cell(0, 5, "No Mark Awarded")
        pdf.ln(1)


    pdf.set_font("Arial", "B", size=10)
    pdf.ln()
```

```
    pdf.multi_cell(0, 1, "Tests")
    pdf.set_font("Arial", size=10)
    pdf.ln(3)

    # FOR TESTS
    marks = TestMark.query.filter_by(student_id=student.id).all()

    for mark in marks:
        test = mark.test

        top = pdf.y
        pdf.multi_cell(0, 5, "("+test.topic.name +") "+test.name)
        pdf.y = top
        pdf.x = 95

        try:
            int(mark.mark)
            if mark.mark != "":
                pdf.multi_cell(0, 5, str(mark.mark) + "/" + str(test.max_mark))
            else:
                pdf.multi_cell(0, 5, "No Mark Awarded")
        except ValueError:
            pdf.multi_cell(0, 5, "No Mark Awarded")
        pdf.ln(1)

    pdf.ln()

    pdf.output(app.config['PDF_FILE_DUMP'] + "student_report.pdf")

    return send_file("static/pdf_gen/student_report.pdf", attachment_filename='student_report.pdf')



# HOMEWORKS

@app.route("/new_homework/<int:topic_id>", methods=['GET', 'POST'])
@login_required
def new_homework(topic_id):
    form = HomeworkForm()
    topic = Topic.query.first_or_404(topic_id)
    if topic.course.teacher != current_user:
        abort(403)
    if form.validate_on_submit():

        topic = Topic.query.first_or_404(topic_id)

        if form.due_date.data > topic.start_date.date() and form.due_date.data < topic.end_date.date():
            new_homework = Homework(name=form.name.data,
                                    due_date=form.due_date.data,
                                    max_mark=form.max_mark.data,
                                    topic_id=topic_id)

            db.session.add(new_homework)
            db.session.commit()
            flash('New homework has been added!', 'success')
            return redirect(url_for('topics'))

        else:
            flash("Homework must be due between "+topic.start_date.strftime("%d %b %Y")+" to "+topic.end_date.strftime("%d %b
%Y") + " for it to be a part of the topic '"+topic.name+"'.", "danger")


    return render_template('new_homework.html', title='New Homework', form=form, legend="New Homework")



@app.route("/homework/<int:homework_id>")
```

```python
@login_required
def homework(homework_id):
    homework = Homework.query.get_or_404(homework_id)
    if homework.topic.course.teacher != current_user:
        abort(403)

    cur_top_classes = homework.topic.course.classes

    class_perf = Graph("Avg Class Performance On " + homework.name, "bar")
    class_perf_data = {"label":"Avg Class Performance On " + homework.name, "data":[]}

    for xclass in cur_top_classes:
        class_perf.labels.append(xclass.class_name)

    cls_avgs = []
    mrks = []
    for xclass in cur_top_classes:
        for student in xclass.students:
            try:
                mark = HomeworkMark.query.filter_by(homework_id=homework.id).filter_by(student_id=student.id).all()[0].mark
            except IndexError:
                mark = 0
            try:
                mrks.append(int(mark))
            except:
                mrks.append(0)

        try:
            avg = sum(mrks) / len(mrks)
        except ZeroDivisionError:
            avg = 0
        class_perf_data["data"].append(avg)
        mrks = []


    if sum(class_perf_data["data"]) != 0:
        class_perf.datasets.append(class_perf_data)

    return render_template('single_homework.html', title=homework.name, homework=homework, class_perf=class_perf)


@app.route("/homework/<int:homework_id>/update", methods=['GET', 'POST'])
@login_required
def update_homework(homework_id):
    homework = Homework.query.get_or_404(homework_id)
    if homework.topic.course.teacher != current_user:
        abort(403)
    form = HomeworkForm()

    courses = Course.query.filter_by(teacher=current_user).all()

    if form.validate_on_submit():

        if form.due_date.data > homework.topic.start_date.date() and form.due_date.data < homework.topic.end_date.date():
            homework.name = form.name.data
            homework.due_date = form.due_date.data
            homework.max_mark = form.max_mark.data
            db.session.commit()
            flash('Your homework has been updated!', 'success')
            return redirect(url_for('topics'))
        else:
            flash("Homework must be due between "+homework.topic.start_date.strftime("%d %b %Y")+" to 
"+homework.topic.end_date.strftime("%d %b %Y") + " for it to be a part of the topic '"+homework.topic.name+"'.", "danger")


    elif request.method == 'GET':
        form.name.data = homework.name
        form.due_date.data = homework.due_date
        form.max_mark.data = homework.max_mark
    return render_template('new_homework.html', title='Update Homework',
                           form=form, legend='Update Homework')


@app.route("/homework/<int:homework_id>/delete", methods=['POST'])
@login_required
def delete_homework(homework_id):
    homework = Homework.query.get_or_404(homework_id)
    if homework.topic.course.teacher != current_user:
        abort(403)

    for mark in homework.homework_marks:
        db.session.delete(mark)

    db.session.delete(homework)
    db.session.commit()
    flash('Your homework has been deleted!', 'success')
    return redirect(url_for('topics'))

from flask_wtf import FlaskForm
from wtforms import FieldList, StringField

@app.route("/homework/<int:homework_id>/mark", methods=['GET', 'POST'])
@login_required
def mark_homework(homework_id):

    homework = Homework.query.get_or_404(homework_id)
    classes = homework.topic.course.classes

    student_number = 0
    for xclass in classes:
        student_number += len(xclass.students)

    class LocalForm(HomeworkMarkForm):pass
    LocalForm.marks = FieldList(StringField('Mark'), min_entries=student_number)
    form = LocalForm()

    mark_fields = {}
    c = 0
    for xclass in classes:
        mark_fields[xclass.id] = []
        students = xclass.students

        print(xclass.class_name)
        for student in xclass.students:
            print(student.name)
            form.marks[c].label.text = student.name + "'s Mark"
            mark_fields[xclass.id].append([student, form.marks[c]])
            c += 1

    if form.validate_on_submit():
        students = []
        for xclass in classes:
            students = students + xclass.students

        marks = []
        for i, mark in enumerate(form.marks.data):

            if mark:
                marks.append((students[i], mark))

        print(marks)
```

```python
        for mark in marks:

            try:

                if int(mark[1]) > homework.max_mark:
                    flash("The maximum mark for this homework is "+str(homework.max_mark)+". All marks must be bellow this.",
"danger")
                    return render_template('mark_homework.html', title='Mark Homework',
                                           form=form, classes=classes, mark_fields=mark_fields, legend='Mark Homework')

                homework_mark = HomeworkMark(mark=mark[1],
                                         homework_id=homework_id,
                                         student_id=mark[0].id)

                print(homework_mark)
                db.session.add(homework_mark)

            except ValueError:
                flash("All marks must be ints!", "danger")
                return render_template('mark_homework.html', title='Mark Homework',
                                       form=form, classes=classes, mark_fields=mark_fields, legend='Mark Homework')

        db.session.commit()
        flash('Homework has been marked!', 'success')
        return redirect(url_for('dash'))

    else:
        print(form.errors)

    if homework.topic.course.teacher != current_user:
        abort(403)

    return render_template('mark_homework.html', title='Mark Homework',
                           form=form, classes=classes, mark_fields=mark_fields, legend='Mark Homework')


# TESTS

@app.route("/new_test/<int:topic_id>", methods=['GET', 'POST'])
@login_required
def new_test(topic_id):
    form = TestForm()
    topic = Topic.query.first_or_404(topic_id)
    if topic.course.teacher != current_user:
        abort(403)

    if form.validate_on_submit():


        if form.date.data > topic.start_date.date() and form.date.data < topic.end_date.date():
            new_test = Test(name=form.name.data,
                            date=form.date.data,
                            max_mark=form.max_mark.data,
                            topic_id=topic_id)

            db.session.add(new_test)
            db.session.commit()
            flash('New test has been added!', 'success')
            return redirect(url_for('topics'))

        else:
            flash("Test must be due between "+topic.start_date.strftime("%d %b %Y")+" to "+topic.end_date.strftime("%d %b %Y") +
" for it to be a part of the topic '"+topic.name+"'.", "danger")


    return render_template('new_test.html', title='New Test', form=form, legend="New Test")
```

```
@app.route("/test/<int:test_id>")
@login_required
def test(test_id):
    test = Test.query.get_or_404(test_id)
    if test.topic.course.teacher != current_user:
        abort(403)


    cur_top_classes = test.topic.course.classes

    class_perf = Graph("Avg Class Performance On " + test.name, "bar")
    class_perf_data = {"label":"Avg Class Performance On " + test.name, "data":[]}

    for xclass in cur_top_classes:
        class_perf.labels.append(xclass.class_name)

    cls_avgs = []
    mrks = []
    for xclass in cur_top_classes:
        for student in xclass.students:
            try:
                mark = TestMark.query.filter_by(homework_id=homework.id).filter_by(student_id=student.id).all()[0].mark
            except IndexError:
                mark = 0
            try:
                mrks.append(int(mark))
            except:
                mrks.append(0)

        try:
            avg = sum(mrks) / len(mrks)
        except ZeroDivisionError:
            avg = 0
        class_perf_data["data"].append(avg)
        mrks = []

    if sum(class_perf_data["data"]) != 0:
        class_perf.datasets.append(class_perf_data)

    return render_template('single_test.html', title=test.name, test=test, class_perf=class_perf)


@app.route("/test/<int:test_id>/update", methods=['GET', 'POST'])
@login_required
def update_test(test_id):
    test = Test.query.get_or_404(test_id)
    if test.topic.course.teacher != current_user:
        abort(403)
    form = TestForm()

    if form.validate_on_submit():

        if form.date.data > test.topic.start_date.date() and form.date.data < test.topic.end_date.date():
            test.name = form.name.data
            test.max_mark = form.max_mark.data
            test.date = form.date.data
            db.session.commit()
            flash('Your test has been updated!', 'success')
            return redirect(url_for('topics'))

        else:
            flash("Test must be due between "+test.topic.start_date.strftime("%d %b %Y")+" to "+test.topic.end_date.strftime("%d
%b %Y") + " for it to be a part of the topic '"+test.topic.name+"'.", "danger")
```

```python
    elif request.method == 'GET':
        form.name.data = test.name
        form.max_mark.data = test.max_mark
        form.date.data = test.date

    return render_template('new_test.html', title='Update Test',
                           form=form, legend='Update Test')


@app.route("/test/<int:test_id>/delete", methods=['POST'])
@login_required
def delete_test(test_id):
    test = Test.query.get_or_404(test_id)
    if test.topic.course.teacher != current_user:
        abort(403)

    marks = TestMark.query.filter_by(test_id=test.id)

    for mark in marks:
        db.session.delete(mark)

    db.session.delete(test)
    db.session.commit()
    flash('Your test has been deleted!', 'success')
    return redirect(url_for('topics'))


@app.route("/test/<int:test_id>/mark", methods=['GET', 'POST'])
@login_required
def mark_test(test_id):

    test = Test.query.get_or_404(test_id)
    classes = test.topic.course.classes

    student_number = 0
    for xclass in classes:
        student_number += len(xclass.students)

    class LocalForm(TestMarkForm):pass
    LocalForm.marks = FieldList(StringField('Mark'), min_entries=student_number)
    form = LocalForm()

    mark_fields = {}
    c = 0
    for xclass in classes:
        mark_fields[xclass.id] = []
        students = xclass.students

        print(xclass.class_name)
        for student in xclass.students:
            print(student.name)
            form.marks[c].label.text = student.name + "'s Mark"
            mark_fields[xclass.id].append([student, form.marks[c]])
            c += 1

    if form.validate_on_submit():
        students = []
        for xclass in classes:
            students = students + xclass.students

        marks = []
        for i, mark in enumerate(form.marks.data):

            if mark:
                marks.append((students[i], mark))
        print(marks)

        for mark in marks:

            try:

                if int(mark[1]) > test.max_mark:
                    flash("The maximum mark for this test is "+str(test.max_mark)+". All marks must be bellow this.", "danger")
                    return render_template('mark_test.html', title='Mark Test',
                                           form=form, classes=classes, mark_fields=mark_fields, legend='Mark Test')

                test_mark = TestMark(mark=mark[1],
                                     test_id=test_id,
                                     student_id=mark[0].id)

                print(test_mark)
                db.session.add(test_mark)


            except ValueError:
                flash("All marks must be ints!", "danger")
                return render_template('mark_test.html', title='Mark Test',
                                       form=form, classes=classes, mark_fields=mark_fields, legend='Mark Test')

        db.session.commit()
        flash('Test has been marked!', 'success')
        return redirect(url_for('dash'))

    else:
        print(form.errors)

    if test.topic.course.teacher != current_user:
        abort(403)

    return render_template('mark_test.html', title='Mark Test',
                           form=form, classes=classes, mark_fields=mark_fields, legend='Mark Test')


# EXAMS

@app.route("/new_exam/<int:course_id>", methods=['GET', 'POST'])
@login_required
def new_exam(course_id):
    form = ExamForm()
    if form.validate_on_submit():
        new_exam = Exam(name=form.name.data,
                        date=form.date.data,
                        max_mark=form.max_mark.data,
                        course_id=course_id)

        db.session.add(new_exam)
        db.session.commit()
        flash('New exam has been added!', 'success')
        return redirect(url_for('courses'))
    return render_template('new_exam.html', title='New Exam', form=form, legend="New Exam")



@app.route("/exam/<int:exam_id>")
@login_required
def exam(exam_id):
    exam = Homework.query.get_or_404(exam_id)
    if exam.course.teacher != current_user:
        abort(403)
    return render_template('single_exam.html', title=exam.name, exam=exam)
```

```python
@app.route("/exam/<int:exam_id>/update", methods=['GET', 'POST'])
@login_required
def update_exam(exam_id):
    exam = Exam.query.get_or_404(exam_id)
    if exam.course.teacher != current_user:
        abort(403)
    form = ExamForm()

    if form.validate_on_submit():
        exam.name = form.name.data
        exam.max_mark = form.max_mark.data
        exam.date = form.date.data
        db.session.commit()
        flash('Your exam has been updated!', 'success')
        return redirect(url_for('courses'))
    elif request.method == 'GET':
        form.name.data = exam.name
        form.max_mark.data = exam.max_mark
        form.date.data = exam.date
    return render_template('new_exam.html', title='Update Exam',
                           form=form, legend='Update Exam')


@app.route("/exam/<int:exam_id>/delete", methods=['POST'])
@login_required
def delete_exam(exam_id):
    exam = Exam.query.get_or_404(exam_id)
    if exam.course.teacher != current_user:
        abort(403)
    db.session.delete(exam)
    db.session.commit()
    flash('Your exam has been deleted!', 'success')
    return redirect(url_for('courses'))


@app.route("/exam/<int:exam_id>/mark", methods=['GET', 'POST'])
@login_required
def mark_exam(exam_id):

    exam = Exam.query.get_or_404(exam_id)
    classes = exam.course.classes

    student_number = 0
    for xclass in classes:
        student_number += len(xclass.students)

    class LocalForm(ExamMarkForm):pass
    LocalForm.marks = FieldList(StringField('Mark'), min_entries=student_number)
    form = LocalForm()

    mark_fields = {}
    c = 0
    for xclass in classes:
        mark_fields[xclass.id] = []
        students = xclass.students

        print(xclass.class_name)
        for student in xclass.students:
            print(student.name)
            form.marks[c].label.text = student.name + "'s Mark"
            mark_fields[xclass.id].append([student, form.marks[c]])
            c += 1

    if form.validate_on_submit():
        students = []
        for xclass in classes:
            students = students + xclass.students

        marks = []
        for i, mark in enumerate(form.marks.data):

            if mark:
                marks.append((students[i], mark))

        print(marks)

        for mark in marks:
            exam_mark = ExamMark(mark=mark[1],
                                 exam_id=exam_id,
                                 student_id=mark[0].id)

            print(exam_mark)
            db.session.add(exam_mark)

        db.session.commit()
        flash('Exam has been marked!', 'success')
        return redirect(url_for('dash'))

    else:
        print(form.errors)

    if exam.course.teacher != current_user:
        abort(403)

    return render_template('mark_exam.html', title='Mark Exam',
                           form=form, classes=classes, mark_fields=mark_fields, legend='Mark Exam')




# COURSES

@app.route("/new_course", methods=['GET', 'POST'])
@login_required
def new_course():
    form = CourseForm()
    form.grade_system.choices = [(0, "A*-U"),
                                 (1, "A-U"),
                                 (2, "A-E"),
                                 (3, "9-1"),
                                 (4, "A-F"),
                                 (5, "A+-F")]
    if form.validate_on_submit():
        new_course = Course(name=form.name.data,
                            start_date=form.start_date.data,
                            year_num=form.year_num.data,
                            grading_system="".join([str(form.grade_system.data), form.grade_system.choices[form.grade_system.data]
[1]]),
                            teacher=current_user)
        db.session.add(new_course)
        db.session.commit()
        flash('New course has been added!', 'success')
        return redirect(url_for('courses'))

    return render_template('new_course.html', title='New Course', form=form, legend="New Course")



@app.route("/courses")
@login_required
def courses():
    page = request.args.get('page', 1, type=int)
    courses = Course.query.filter_by(teacher=current_user)

    courses = courses.paginate(page=page, per_page=5)

    return render_template('all_courses.html', title="My Courses", courses=courses, user=current_user)


@app.route("/course/<int:course_id>/update", methods=['GET', 'POST'])
@login_required
def update_course(course_id):
    course = Course.query.get_or_404(course_id)
    if course.teacher != current_user:
        abort(403)
    form = CourseForm()
    form.grade_system.choices = [(0, "A*-U"),
                                (1, "A-U"),
                                (2, "A-E"),
                                (3, "9-1"),
                                (4, "A-F"),
                                (5, "A+-F")]
    if form.validate_on_submit():
        course.name = form.name.data
        course.start_date = form.start_date.data
        course.year_num = form.year_num.data
        course.grading_system = "".join([str(form.grade_system.data), form.grade_system.choices[form.grade_system.data][1]])
        db.session.commit()
        flash('Your course has been updated!', 'success')
        return redirect(url_for('courses'))
    elif request.method == 'GET':
        form.name.data = course.name
        form.start_date.data = course.start_date
        form.year_num.data = course.year_num
        form.grade_system.data = int(course.grading_system[0])
    return render_template('new_course.html', title='Update Course',
                          form=form, legend='Update Course')


@app.route("/course/<int:course_id>/delete", methods=['POST'])
@login_required
def delete_course(course_id):
    course = Course.query.get_or_404(course_id)
    if course.teacher != current_user:
        abort(403)

    for topic in course.topics:
        db.session.delete(topic)

    for exam in course.exams:
        db.session.delete(exam)

    for xclass in course.classes:
        db.session.delete(xclass)

    db.session.delete(course)
    db.session.commit()
    flash('Your course has been deleted!', 'success')
    return redirect(url_for('courses'))

# TOPICS

@app.route("/new_topic", methods=['GET', 'POST'])
@login_required
def new_topic():
    form = TopicForm()

    courses = Course.query.filter_by(teacher=current_user).all()
    form.course_id.choices = [(course.id, course.name) for num, course in enumerate(courses)]

    if not len(courses):
        flash(Markup('To add a topic you must first <a href="' + url_for('new_course') + '">add a course</a>.'), 'danger')

    # If user doesn't request for a course, then enable the select box and set the result as the course_id...
    from datetime import date, timedelta

    try:
        course_id = request.args['course_id']
        form.course_id.value = course_id

        course = Course.query.get_or_404(course_id)
        minDate = course.start_date.strftime("%Y-%m-%d")

        all_block_dates = []
        for topic in course.topics:
            sdate = topic.start_date   # start date
            edate = topic.end_date   # end date
            delta = edate - sdate
            block_dates = [(sdate + timedelta(days=i)).strftime("%Y-%m-%d") for i in range(delta.days + 1)]
            all_block_dates = all_block_dates + block_dates

        all_block_dates = sorted(list(set(all_block_dates)))

        print(all_block_dates)


    except KeyError:
        form.course_id.enabled = True
        course_id = form.course_id.data

    if form.validate_on_submit():

        course = Course.query.get_or_404(course_id)
        if course.teacher != current_user:
            abort(403)


        if form.begin_date.data < form.end_date.data:

            conflicts = 0
            for topic in course.topics:
                if (form.begin_date.data < topic.start_date.date() and form.end_date.data < topic.start_date.date()) or
(form.begin_date.data > topic.end_date.date() and form.end_date.data > topic.end_date.date()):
                    pass
                else:
                    conflicts += 1
                    flash("Topic dates conflict with the active period of another topic! Change BOTH dates to before
"+topic.start_date.strftime("%d %b %Y")+" or after "+topic.end_date.strftime("%d %b %Y") + " to resolve the conflict.", "danger")

            if not conflicts:
                if form.begin_date.data < course.start_date.date():
                    flash("Topic cannot begin before the course it's a part of does!", "danger")
                else:

                    new_topic = Topic(name=form.name.data,
                                      start_date=form.begin_date.data,
                                      end_date=form.end_date.data,
                                     course=course)

                    db.session.add(new_topic)
                    db.session.commit()
                    flash(f'New topic has been added to {course.name}!', 'success')

                    next_page = request.args.get('next')
```

```python
                return redirect(next_page) if next_page else redirect(url_for('topics'))
        else:
            flash("The topic cannot end before it begins!", "danger")

    return render_template('new_topic.html',
                        title='New Topic',
                        form=form,
                        legend="New Topic",
                        min_date=minDate,
                        blocked_dates=all_block_dates)


@app.route("/topics", methods=['GET', 'POST'])
@login_required
def topics():
    page = request.args.get('page', 1, type=int)
    courses = Course.query.filter_by(teacher=current_user)


    s_form = SearchForm()
    srch = 0
    search_results = []
    if s_form.validate_on_submit():
        print("Search For:", s_form.search_query.data)

        course_ids = [course.id for course in courses.all()]

        search_results = Topic.query.filter(Topic.name.like('%' + s_form.search_query.data + '%'))
        search_results = search_results.filter(Topic.course_id.in_(course_ids))
        srch = 1


    if not len(courses.all()):
        flash(Markup('To add a topic you must first <a href="' + url_for('new_course') + '">add a course</a>.'), 'danger')

    courses = courses.paginate(page=page, per_page=5)


    return render_template('all_topics.html',
                    title="My Topics",
                    courses=courses,
                    user=current_user,
                    today=datetime.now(),
                    s_form=s_form,
                    search_results=search_results,
                    srch=srch)


@app.route("/topic/<int:topic_id>")
@login_required
def topic(topic_id):
    topic = Topic.query.get_or_404(topic_id)
    if topic.course.teacher != current_user:
        abort(403)
    return render_template('single_topic.html', title=topic.name, topic=topic)


@app.route("/topic/<int:topic_id>/update", methods=['GET', 'POST'])
@login_required
def update_topic(topic_id):
    topic = Topic.query.get_or_404(topic_id)
    if topic.course.teacher != current_user:
        abort(403)
    form = TopicForm()
    form.course_id.enabled = True
    courses = Course.query.filter_by(teacher=current_user).all()
    form.course_id.choices = [(course.id, course.name) for num, course in enumerate(courses)]

    if form.validate_on_submit():
        course = Course.query.get_or_404(form.course_id.data)

        if form.begin_date.data < form.end_date.data:

            conflicts = 0
            for t in course.topics:
                if t.id == topic.id:
                    pass
                elif (form.begin_date.data < t.start_date.date() and form.end_date.data < t.start_date.date()) or
(form.begin_date.data > t.end_date.date() and form.end_date.data > t.end_date.date()):
                    pass
                else:
                    conflicts += 1
                    flash("Topic dates conflict with the active period of another topic! Change BOTH dates to before
"+t.start_date.strftime("%d %b %Y")+" or after "+t.end_date.strftime("%d %b %Y") + " to resolve the conflict.", "danger")

            if not conflicts:
                if form.begin_date.data < course.start_date.date():
                    flash("Topic cannot begin before the course it's a part of does!", "danger")
                else:
                    topic.name = form.name.data
                    topic.start_date = form.begin_date.data
                    topic.end_date = form.end_date.data
                    topic.course_id = form.course_id.data
                    db.session.commit()
                    flash('Your topic has been updated!', 'success')
                    return redirect(url_for('topics'))
        else:
            flash("The topic cannot end before it begins!", "danger")

    elif request.method == 'GET':
        form.name.data = topic.name
        print()
        form.begin_date.data = topic.start_date
        form.end_date.data = topic.end_date
        form.course_id.data = topic.course.id
    return render_template('new_topic.html', title='Update Topic',
                           form=form, legend='Update Topic')


@app.route("/topic/<int:topic_id>/delete", methods=['POST'])
@login_required
def delete_topic(topic_id):
    topic = Topic.query.get_or_404(topic_id)
    if topic.course.teacher != current_user:
        abort(403)

    for test in topic.tests:
        db.session.delete(test)

    for hmwk in topic.homeworks:
        db.session.delete(hmwk)

    db.session.delete(topic)
    db.session.commit()
    flash('Your topic has been deleted!', 'success')
    return redirect(url_for('courses'))
```