Name: Hazrat Hanzala

Student ID: B00957159

Title: Code File

APP.py

```
from flask import Flask, jsonify

from routes.admin_routes import admin_bp

from routes.users_routes import users_bp

from routes.guest_routes import guest_bp

from routes.auth_routes import auth_bp

from routes.quizzes_routes import quizzes_bp


app = Flask(__name__)

app.config["SECRET_KEY"] = "mysecret@1"
app.config["JSONIFY_PRETTYPRINT_REGULAR"] = True

# Register Blueprints
app.register_blueprint(auth_bp, url_prefix="/auth")
app.register_blueprint(users_bp, url_prefix="/users")
app.register_blueprint(admin_bp, url_prefix="/admin")
app.register_blueprint(guest_bp, url_prefix="/guest")
app.register_blueprint(quizzes_bp, url_prefix="/quizzes")

@app.route('/', methods=['GET'])
def home():
    return jsonify({"message": "Welcome to Quiz API - COM661 Project"})

if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

db_config.py

```
# MongoDB Configuration
from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017")
```

```
db = client.Quiz
quizzes = db.quizzes
blacklist = db.blacklist
```

admin_routes.py

```
# ----------------------------------
# Admin Routes – Quiz Management and Analytics
# ---------------------------

from flask import Blueprint, jsonify, request, make_response
from config.db_config import db
from utils.decorators import token_required

# Define bp for admin routes------------------
admin_bp = Blueprint('admin_bp', __name__)
quizzes = db.quizzes

# ------- Create new Quiz
@admin_bp.route('/quizzes', methods=['POST'])
@token_required(role='admin')
def create_quiz():
    data = request.form if request.form else request.get_json()

    if not data or not data.get("title"):
        return make_response(jsonify({"error": "Quiz title is
required"}), 400)

    try:
        new_quiz = {
            "quizId": f"QZ-{db.quizzes.count_documents({}) + 1:03}",
            "title": data.get("title"),
            "difficulty": data.get("difficulty", "Medium"),
            "creator": data.get("creator", {"name": "Admin"}),
            "questions": data.get("questions", []),
            "attempts": []
        }

        quizzes.insert_one(new_quiz)
        return make_response(jsonify({"message": "Quiz created
successfully"}), 201)

    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)

# - update existing quiz
```

```
@admin_bp.route('/quizzes/<string:quizId>', methods=['PUT'])
@token_required(role='admin')
def update_quiz(quizId):



   data = request.form if request.form else request.get_json()
   update_fields = {k: v for k, v in data.items() if v}

   if not update_fields:
      return make_response(jsonify({"error": "No fields provided to
update"}), 400)

   result = quizzes.update_one({"quizId": quizId}, {"$set":
update_fields})

   if result.matched_count == 0:
      return make_response(jsonify({"error": "Quiz not found"}),
404)

   return make_response(jsonify({"message": "Quiz updated
successfully"}), 200)

# ------ delete quiz -----------
@admin_bp.route('/quizzes/<string:quizId>', methods=['DELETE'])
@token_required(role='admin')
```

```
def delete_quiz(quizId):
   try:
      result = quizzes.delete_one({"quizId": quizId})
      if result.deleted_count == 0:
         return make_response(jsonify({"error": "Quiz not found"}),
404)
      return make_response(jsonify({"message": "Quiz deleted
successfully"}), 200)
   except Exception as e:
      return make_response(jsonify({"error": str(e)}), 500)

#  view all quizzes ----------------
@admin_bp.route('/quizzes', methods=['GET'])
@token_required(role='admin')
def view_all_quizzes():

   try:
      all_quizzes = list(quizzes.find({}, {"_id": 0}))
      return make_response(jsonify(all_quizzes), 200)
   except Exception as e:
      return make_response(jsonify({"error": str(e)}), 500)

# - view quiz attempts ----------------
@admin_bp.route('/quizzes/<string:quizId>/attempts',
methods=['GET'])
```

```
@token_required(role='admin')
def view_quiz_attempts(quizId):

    quiz = quizzes.find_one({"quizId": quizId}, {"_id": 0, "attempts": 1})

    if not quiz:
        return make_response(jsonify({"error": "Quiz not found"}), 404)

    return make_response(jsonify(quiz.get("attempts", [])), 200)

# Quiz Leaderboard ----------------
@admin_bp.route('/quizzes/<string:quizId>/leaderboard', methods=['GET'])
@token_required(role='admin')
def leaderboard(quizId):

    quiz = quizzes.find_one({"quizId": quizId})

    if not quiz:
        return make_response(jsonify({"error": "Quiz not found"}), 404)

    attempts = quiz.get("attempts", [])
    sorted_attempts = sorted(attempts, key=lambda x:
x.get("score", 0), reverse=True)[:5]
    return make_response(jsonify(sorted_attempts), 200)


# quiz statistics ----------------
@admin_bp.route('/stats', methods=['GET'])
@token_required(role='admin')
def stats():
    try:
        pipeline = [
            {"$group": {"_id": "$difficulty", "count": {"$sum": 1}}},
            {"$sort": {"count": -1}}
        ]
        data = list(quizzes.aggregate(pipeline))
        return make_response(jsonify(data), 200)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)


#   view single quiz ----------------


@admin_bp.route('/quizzes/<string:quizId>', methods=['GET'])
@token_required(role='admin')
def view_quiz(quizId):
    quiz = quizzes.find_one({"quizId": quizId}, {"_id": 0})
    if not quiz:
        return make_response(jsonify({"error": "Quiz not found"}), 404)
    return make_response(jsonify(quiz), 200)
```

auth_routes.py

```
# Auth Routes – Registration, Login and Logout

from flask import Blueprint, request, jsonify, make_response
from config.db_config import db
from utils.decorators import token_required
import bcrypt, jwt, datetime, base64

# Blueprint setup
auth_bp = Blueprint('auth_bp', __name__)
users = db.users
blacklist = db.blacklist
SECRET_KEY = "mysecret@1"

# -----register new user
@auth_bp.route('/register', methods=['POST'])
def register_user():
    data = request.form if request.form else request.get_json()

    if not data.get('email') or not data.get('password'):
        return make_response(jsonify({"error": "Email and password
required"}), 400)
    if users.find_one({"email": data.get('email')}):
        return make_response(jsonify({"error": "Email already
registered"}), 409)

    hashed_pw = bcrypt.hashpw(data.get('password').encode('utf-
8'), bcrypt.gensalt())
    new_user = {
        "name": data.get('name', 'Anonymous'),
        "email": data.get('email'),
        "password": hashed_pw,
        "role": data.get('role', 'user')
    }

    users.insert_one(new_user)
    return make_response(jsonify({"message": "User registered
successfully"}), 201)

# --login existing user bauth + json
@auth_bp.route('/login', methods=['POST'])
def login_user():
    auth_header = request.headers.get('Authorization')
    data = None

    #   Basic Auth
    if auth_header and auth_header.startswith('Basic '):
        try:
            encoded = auth_header.split(" ")[1]
            decoded = base64.b64decode(encoded).decode("utf-8")
            email, password = decoded.split(":", 1)
            data = {"email": email, "password": password}
        except Exception as e:
            return make_response(jsonify({"error": f"Invalid Basic Auth
format: {str(e)}"}), 400)

    #  JSON body
    if not data:
        data = request.form if request.form else request.get_json()

    if not data or not data.get("email") or not data.get("password"):
        return make_response(jsonify({"error": "Email and password
required"}), 400)

    #  Validate user
    user = users.find_one({"email": data.get("email")})
    if not user or not
bcrypt.checkpw(data.get("password").encode("utf-8"),
user["password"]):
        return make_response(jsonify({"error": "Invalid credentials"}),
401)


    #  generate JWT token
    token = jwt.encode({
        "user": user["email"],
        "role": user.get("role", "user"),
        "exp": datetime.datetime.utcnow() +
datetime.timedelta(hours=2)
    }, SECRET_KEY, algorithm="HS256")



    if isinstance(token, bytes):
        token = token.decode("utf-8")


    return make_response(jsonify({
        "token": token,
        "role": user.get("role", "user")
    }), 200)


# ----- logout user
@auth_bp.route('/logout', methods=['POST'])
```

```
@token_required()
def logout_user():

   token = request.headers.get('x-access-token')
   blacklist.insert_one({"token": token})
   return make_response(jsonify({"message": "Logged out
successfully"}), 200)
```

guest_routes.py

```
from flask import Blueprint, jsonify, make_response
from config.db_config import db

guest_bp = Blueprint('guest_bp', __name__)
quizzes = db.quizzes

# -------public quizzes list ----
@guest_bp.route('/public', methods=['GET'])
def guest_quizzes():

   try:
      data = list(quizzes.find({}, {"_id": 0, "quizId": 1, "title": 1,
"difficulty": 1}))
```

```
        return make_response(jsonify(data), 200)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)


# -------quiz preview ----
@guest_bp.route('/<string:quizId>/preview', methods=['GET'])
def preview_quiz(quizId):


    try:
        quiz = quizzes.find_one({"quizId": quizId}, {"_id": 0, "title": 1,
"difficulty": 1, "questions": 1})
        if not quiz:
            return make_response(jsonify({"error": "Quiz not found"}),
404)


        summary = {
            "quizId": quizId,
            "title": quiz["title"],
            "difficulty": quiz["difficulty"],
            "question_count": len(quiz.get("questions", []))
        }
        return make_response(jsonify(summary), 200)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)
```

```
# ------------public leaderboard ----
@guest_bp.route('/quizzes/<string:quizId>/leaderboard',
methods=['GET'])
def public_leaderboard(quizId):
   quiz = quizzes.find_one({"quizId": quizId})


   if not quiz:
      return make_response(jsonify({"error": "Quiz not found"}),
404)



   attempts = quiz.get("attempts", [])
   if not isinstance(attempts, list):
      return make_response(jsonify({"error": "Invalid attempts
format"}), 500)


   sorted_attempts = sorted(attempts, key=lambda x:
x.get("score", 0), reverse=True)[:5]
   return make_response(jsonify(sorted_attempts), 200)
```

<u>**quizzes_routes.py**</u>

```
from flask import Blueprint, jsonify, request, make_response
from config.db_config import db
from utils.decorators import token_required
from bson import ObjectId
from collections import OrderedDict
import jwt

# Define blueprint for all quiz related routes
quizzes_bp = Blueprint('quizzes_bp', __name__)
quizzes = db.quizzes
SECRET_KEY = "mysecret@1"

# ------------get all quizzes (admin access) ----
@quizzes_bp.route('/', methods=['GET'])
def get_all_quizzes():

    data_to_return = []
    page_num = request.args.get('pn', default=1, type=int)
    page_size = request.args.get('ps', default=10, type=int)
    page_start = (page_num - 1) * page_size

    try:
        cursor = quizzes.find().skip(page_start).limit(page_size)

        for quiz in cursor:
            quiz["_id"] = str(quiz["_id"])
            ordered = OrderedDict()



            for key in ["quizId", "title", "difficulty", "creator", "questions",
"attempts", "_id"]:
                if key in quiz:
                    ordered[key] = quiz[key]

            data_to_return.append(ordered)

        return make_response(jsonify(data_to_return), 200)

    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)

# --guest routes -
@quizzes_bp.route('/public', methods=['GET'])
def guest_quizzes():

    try:
        data = list(quizzes.find({}, {"_id": 0, "quizId": 1, "title": 1,
"difficulty": 1}))
        return make_response(jsonify(data), 200)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)


@quizzes_bp.route('/<string:quizId>/preview', methods=['GET'])
def preview_quiz(quizId):

    try:
        quiz = quizzes.find_one(
            {"quizId": quizId},
            {"_id": 0, "title": 1, "difficulty": 1, "questions": 1}
        )

        if not quiz:
            return make_response(jsonify({"error": "Quiz not found"}),
404)

        summary = {
            "quizId": quizId,
            "title": quiz["title"],
            "difficulty": quiz["difficulty"],
            "question_count": len(quiz.get("questions", []))
        }

        return make_response(jsonify(summary), 200)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)

# ---------get quiz details
@quizzes_bp.route('/<string:quizId>', methods=['GET'])
@token_required(role='user')
def get_quiz_details(quizId):

    try:
        quiz = quizzes.find_one({"quizId": quizId}, {"_id": 0})
        if not quiz:
            return make_response(jsonify({"error": "Quiz not found"}),
404)



        if "questions" in quiz:
            if isinstance(quiz["questions"], list):
                for q in quiz["questions"]:
                    q.pop("correct_answer", None)
            elif isinstance(quiz["questions"], dict):
                quiz["questions"].pop("correct_answer", None)

        return make_response(jsonify(quiz), 200)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)


# ---------submit quiz attempt ----
@quizzes_bp.route('/<string:quizId>/attempts', 
methods=['POST'])
@token_required(role='user')
def submit_quiz_attempt(quizId):

    data = request.get_json()

    user_email = data.get('email')
    answers = data.get('answers', [])

    # Fetch quiz data
    quiz = quizzes.find_one({"quizId": quizId})
    if not quiz:
        return make_response(jsonify({"error": "Quiz not found"}), 
404)

    questions = quiz.get("questions", [])
    score = 0

    # Calculate score
    if isinstance(questions, list):
        for idx, question in enumerate(questions):
            if idx < len(answers) and answers[idx] ==
question.get("correct_answer"):
                score += question.get("marks", 0)
    elif isinstance(questions, dict):
        # Single question quiz
        if answers and answers[0] ==
questions.get("correct_answer"):
            score = questions.get("marks", 0)

    attempt_record = {
        "userId": data.get('userId'),
        "name": data.get('name'),
        "email": user_email,
        "score": score,
        "completed": True
    }


    if not isinstance(quiz.get("attempts"), list):
        quizzes.update_one({"quizId": quizId}, {"$set": {"attempts":
[]}})


    quizzes.update_one(
        {"quizId": quizId},
        {"$push": {"attempts": attempt_record}}
    )

    return make_response(jsonify({
        "message": "Attempt submitted successfully",
        "score": score
    }), 201)

# -------public leaderboard ----
@quizzes_bp.route('/<string:quizId>/leaderboard',
methods=['GET'])
def public_leaderboard(quizId):
    try:
        quiz = quizzes.find_one({"quizId": quizId})
        if not quiz:
            return make_response(jsonify({"error": "Quiz not found"}),
404)

        attempts = quiz.get("attempts", [])
        sorted_attempts = sorted(attempts, key=lambda x:
x.get("score", 0), reverse=True)[:5]

        return make_response(jsonify(sorted_attempts), 200)
    except Exception as e:
```

```
    return make_response(jsonify({"error": str(e)}), 500)
```

users_routes.py

```
from flask import Blueprint, request, jsonify, make_response
from config.db_config import db
from utils.decorators import token_required as jwt_required
import bcrypt, jwt, datetime

# Blueprint setup
users_bp = Blueprint('users_bp', __name__)
users = db.users
quizzes = db.quizzes
blacklist = db.blacklist

SECRET_KEY = "mysecret@1"

# --------------reg new user ----------------
@users_bp.route('/register', methods=['POST'])
def register_user():

  data = request.form if request.form else request.get_json()

  # Validate essential fields
```

```
    if not data.get('email') or not data.get('password'):
        return make_response(jsonify({"error": "Email and password 
are required"}), 400)

    # Prevent duplicate registration
    if users.find_one({"email": data.get('email')}):
        return make_response(jsonify({"error": "Email already 
registered"}), 409)

    hashed_pw = bcrypt.hashpw(data.get('password').encode('utf-
8'), bcrypt.gensalt())

    new_user = {
        "name": data.get('name', 'Anonymous'),
        "email": data.get('email'),
        "password": hashed_pw,
        "role": data.get('role', 'user')
    }

    users.insert_one(new_user)
    return make_response(jsonify({"message": "User registered 
successfully"}), 201)

# ----------login user
@users_bp.route('/login', methods=['POST'])
```

```
def login_user():
    data = request.form if request.form else request.get_json()
    user = users.find_one({"email": data.get('email')})

    if not user or not
bcrypt.checkpw(data.get('password').encode('utf-8'),
user['password']):
        return make_response(jsonify({"error": "Invalid credentials"}),
401)

    token = jwt.encode({
        'user': user['email'],
        'role': user['role'],
        'exp': datetime.datetime.utcnow() +
datetime.timedelta(hours=2)
    }, SECRET_KEY, algorithm='HS256')

    return make_response(jsonify({'token': token}), 200)

# ----------view profile
@users_bp.route('/profile', methods=['GET'])
@jwt_required()
def user_profile():
    try:
        token = request.headers.get('x-access-token')
        data = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
        user = users.find_one({"email": data.get('user')}, {"_id": 0, "password": 0})

        if not user:
            return make_response(jsonify({"error": "User not found"}), 404)

        return make_response(jsonify(user), 200)

    except jwt.ExpiredSignatureError:
        return make_response(jsonify({"error": "Token expired"}), 401)
    except Exception:
        return make_response(jsonify({"error": "Invalid token"}), 401)

# ---------update user role
@users_bp.route('/update-role', methods=['PUT'])
@jwt_required(role='admin')
def update_user_role():
    data = request.form if request.form else request.get_json()
    email = data.get('email')
    new_role = data.get('role')

    if not email or not new_role:
        return make_response(jsonify({"error": "Email and new role
are required"}), 400)

    result = users.update_one({"email": email}, {"$set": {"role":
new_role}})

    if result.matched_count == 0:
        return make_response(jsonify({"error": "User not found"}),
404)

    return make_response(jsonify({"message": f"User role updated
to {new_role}"}), 200)

# ---view user attempts
@users_bp.route('/<string:userId>/attempts', methods=['GET'])
@jwt_required(role='user')
def view_user_attempts(userId):

    try:
        user_attempts = []

        for quiz in quizzes.find({}, {"_id": 0, "quizId": 1, "title": 1,
"attempts": 1}):
            for attempt in quiz.get("attempts", []):
                if attempt.get("userId") == userId:
                    user_attempts.append({
                        "quizId": quiz["quizId"],
                        "quizTitle": quiz["title"],
                        "score": attempt["score"],
                        "completed": attempt["completed"]
                    })

        if not user_attempts:
            return make_response(jsonify({"message": "No attempts found for this user"}), 200)

        return make_response(jsonify(user_attempts), 200)

    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)

# ------------delete user account
@users_bp.route('/delete', methods=['DELETE'])
@jwt_required(role='user')
def delete_user_account():
    try:
        token = request.headers.get('x-access-token')
        data = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
        email = data.get('user')
        result = users.delete_one({"email": email})

        if result.deleted_count == 0:
            return make_response(jsonify({"error": "Account not
found"}), 404)

        return make_response(jsonify({"message": "Account deleted
successfully"}), 200)

    except jwt.ExpiredSignatureError:
        return make_response(jsonify({"error": "Token expired"}), 401)
    except Exception as e:
        return make_response(jsonify({"error": str(e)}), 500)

# ----logout user
@users_bp.route('/logout', methods=['POST'])
@jwt_required()
def logout_user():
    token = request.headers.get('x-access-token')

    # Check if already blacklisted
    if blacklist.find_one({"token": token}):
        return make_response(jsonify({"message": "Token already
invalidated"}), 200)
```

```
    blacklist.insert_one({"token": token})
    return make_response(jsonify({"message": "User logged out successfully"}), 200)
```

blacklist.py

```
blacklist = set()

def add_to_blacklist(token):
    blacklist.add(token)

def is_token_blacklisted(token):
    return token in blacklist
```

decorators.py

```
from flask import request, jsonify, make_response
from config.db_config import db
from functools import wraps
import jwt

# Secret key
SECRET_KEY = "mysecret@1"
```

```
blacklist = db.blacklist

# -----------token required decorator -----------
def token_required(role=None):

   def decorator(func):
      @wraps(func)
      def wrapper(*args, **kwargs):
         token = request.headers.get("x-access-token")

         # If no token provided
         if not token:
            return make_response(jsonify({"error": "Token missing"}),
401)

         # Check if the token exists in blacklist (revoked token)
         if blacklist.find_one({"token": token}):
            return make_response(jsonify({"error": "Token has been
cancelled"}), 401)

         try:
            # Decode the JWT token using secret key
            data = jwt.decode(token, SECRET_KEY,
algorithms=["HS256"])
```

```
            # If role restriction is applied (e.g., admin-only route)
            if role and data.get("role") != role:
                return make_response(jsonify({"error": "Access
denied"}), 403)

        except jwt.ExpiredSignatureError:
            return make_response(jsonify({"error": "Token expired"}),
401)
        except jwt.InvalidTokenError:
            return make_response(jsonify({"error": "Invalid token"}),
401)
        except Exception as e:
            return make_response(jsonify({"error": f"Token validation
failed: {str(e)}"}), 401)

        # Token valid → continue with the route
        return func(*args, **kwargs)

      return wrapper
   return decorator
```