**Technologies Used**

* **Flask** (for the web framework)
* **Flask-JWT-Extended** (for JWT-based authentication)
* **Flask-SQLAlchemy** (for database management)
* **bcrypt** (for password hashing)

**Step-by-Step Guide**

**1. Setting Up the Environment**

Start by setting up your Flask environment. You can install the required packages using pip:

bash

pip install Flask Flask-JWT-Extended Flask-SQLAlchemy Flask-Bcrypt

**2. Project Structure**

plaintext

project/

│

├── app.py # Main file where Flask app is created

├── models.py # Database models (Users, Roles, Permissions)

├── routes.py # Routes for authentication and resource access

├── utils.py # Helper functions

└── config.py # Configuration (JWT secret key, database URL)

**3. Configuration (config.py)**

In this file, you will store important configurations like your database URL and JWT secret key.

python

import os

class Config:

SECRET\_KEY = os.getenv('SECRET\_KEY', 'your\_secret\_key')

SQLALCHEMY\_DATABASE\_URI = 'sqlite:///site.db'

JWT\_SECRET\_KEY = os.getenv('JWT\_SECRET\_KEY', 'super\_secret')

SQLALCHEMY\_TRACK\_MODIFICATIONS = False

**4. Database Models (models.py)**

You will create three main models: **User**, **Role**, and **Permission**.

python

from app import db

from flask\_bcrypt import Bcrypt

bcrypt = Bcrypt()

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(150), unique=True, nullable=False)

email = db.Column(db.String(150), unique=True, nullable=False)

password = db.Column(db.String(256), nullable=False)

role\_id = db.Column(db.Integer, db.ForeignKey('role.id'))

def set\_password(self, password):

self.password = bcrypt.generate\_password\_hash(password).decode('utf-8')

def check\_password(self, password):

return bcrypt.check\_password\_hash(self.password, password)

class Role(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(100), unique=True, nullable=False)

permissions = db.relationship('Permission', backref='role', lazy=True)

class Permission(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(100), nullable=False)

role\_id = db.Column(db.Integer, db.ForeignKey('role.id'))

**5. JWT Authentication (routes.py)**

For authentication, users will be able to register, log in, and obtain a JWT token that will be used to authenticate future requests.

python

from flask import Flask, request, jsonify

from flask\_jwt\_extended import JWTManager, create\_access\_token, jwt\_required, get\_jwt\_identity

from app import app, db

from models import User, Role

jwt = JWTManager(app)

@app.route('/register', methods=['POST'])

def register():

data = request.get\_json()

user = User(username=data['username'], email=data['email'])

user.set\_password(data['password'])

db.session.add(user)

db.session.commit()

return jsonify(message="User created successfully"), 201

@app.route('/login', methods=['POST'])

def login():

data = request.get\_json()

user = User.query.filter\_by(email=data['email']).first()

if user and user.check\_password(data['password']):

token = create\_access\_token(identity=user.id)

return jsonify(access\_token=token), 200

return jsonify(message="Invalid credentials"), 401

@app.route('/logout', methods=['POST'])

@jwt\_required()

def logout():

return jsonify(message="Logged out successfully"), 200

**6. Authorization and Role-Based Access Control (routes.py)**

Here, we will implement RBAC, ensuring only certain roles can access specific routes.

python

from functools import wraps

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity

from models import User, Role

def role\_required(role\_name):

def wrapper(fn):

@wraps(fn)

def decorator(\*args, \*\*kwargs):

user\_id = get\_jwt\_identity()

user = User.query.get(user\_id)

if user.role.name != role\_name:

return jsonify(message="Access denied"), 403

return fn(\*args, \*\*kwargs)

return decorator

return wrapper

@app.route('/admin', methods=['GET'])

@jwt\_required()

@role\_required('Admin')

def admin\_route():

return jsonify(message="Welcome, Admin!"), 200

@app.route('/user', methods=['GET'])

@jwt\_required()

@role\_required('User')

def user\_route():

return jsonify(message="Welcome, User!"), 200

**7. Password Hashing**

In the user registration, the password is hashed using bcrypt:

python

from flask\_bcrypt import Bcrypt

bcrypt = Bcrypt()

def set\_password(password):

return bcrypt.generate\_password\_hash(password).decode('utf-8')

def check\_password(hashed\_password, password):

return bcrypt.check\_password\_hash(hashed\_password, password)

**8. Testing Your System**

You can now test your RBAC system by:

* Registering users with different roles (Admin, User).
* Logging in to get JWT tokens.
* Trying to access role-protected routes (/admin, /user).

Ensure users can only access resources based on their role, as defined in the role\_required decorator.

**Additional Features (Optional)**

* **Password Reset**: Implement password reset functionality using secure tokens.
* **Account Locking**: Lock user accounts after several failed login attempts.
* **Audit Logs**: Track access logs for sensitive resources.

**Conclusion**

This solution offers a flexible way to implement Authentication, Authorization, and Role-Based Access Control in a secure Flask-based web system. You can extend it with more roles and permissions depending on your needs. Let me know if you need further details on any specific section!