Python Coding Challenge -2

Career Hub , The Job Board

Submitted by : Mridul Bhardwaj

1.Create and implement the mentioned class and the structure in your application.

JobListing Class:

Attributes:

• JobID (int): A unique identifier for each job listing.

• CompanyID (int): A reference to the company offering the job.

• JobTitle (string): The title of the job.

• JobDescription (string): A detailed description of the job.

• JobLocation (string): The location of the job.

• Salary (decimal): The salary offered for the job.

• JobType (string): The type of job (e.g., Full-time, Part-time, Contract).

• PostedDate (DateTime): The date when the job was posted.

Methods:

• Apply(applicantID: int, coverLetter: string): Allows applicants to apply for the job by providing their ID and a cover letter.

• GetApplicants(): List: Retrieves a list of applicants who have applied for the job.

```python
from datetime import datetime
from Exception.Exception import NegativeSalaryException
class JobListing:
    def __init__(self, JobID, CompanyID, JobTitle, JobDescription, JobLocation, Salary, JobType,PostedDate):
        self._JobID = JobID
        self._CompanyID = CompanyID
        self._JobTitle = JobTitle
        self._JobDescription = JobDescription
        self._JobLocation = JobLocation
        self._Salary = Salary
        self._JobType = JobType
        self._PostedDate =PostedDate
```

```python
@property
def JobID(self):
    return self._JobID

@JobID.setter
def JobID(self,new_JobID):
    if isinstance(new_JobID, int) and new_JobID > 0:
        self._JobID = new_JobID
    else:
        raise ValueError("JobID must be a positive integer.")
```

1 usage

```python
@property
def CompanyID(self):
    return self._CompanyID

@CompanyID.setter
def CompanyID(self,new_CompanyID):
    if isinstance(new_CompanyID, int) and new_CompanyID > 0:
        self._CompanyID = new_CompanyID
    else:
        raise ValueError("CompanyID must be a positive integer.")
```

1 usage

```python
@property
def JobTitle(self):
    return self._JobTitle
```

```python
@JobTitle.setter
def JobTitle(self, new_JobTitle):
    if isinstance(new_JobTitle, str) and len(new_JobTitle) > 0:
        self._JobTitle = new_JobTitle
    else:
        raise ValueError("JobTitle must be a non-empty string.")

1 usage
@property
def JobDescription(self):
    return self._JobDescription

@JobDescription.setter
def JobDescription(self, new_JobDescription):
    if isinstance(new_JobDescription, str) and len(new_JobDescription) > 0:
        self._JobDescription = new_JobDescription
    else:
        raise ValueError("JobDescription must be a non-empty string.")
1 usage
@property
def JobLocation(self):
    return self._JobLocation

@JobLocation.setter
def JobLocation(self, new_JobLocation):
    if isinstance(new_JobLocation, str) and len(new_JobLocation) > 0:
        self._JobLocation = new_JobLocation
    else:
        raise ValueError("JobLocation must be a non-empty string.")
```

```python
@property
def Salary(self):
    return self._Salary

@Salary.setter
def Salary(self, new_Salary):
    if isinstance(new_Salary, float) and new_Salary >= 0:
        self._Salary = new_Salary
    else:
        raise NegativeSalaryException
```

1 usage

```python
@property
def JobType(self):
    return self._JobType

@JobType.setter
def JobType(self, new_JobType):
    if isinstance(new_JobType, str) and len(new_JobType) > 0:
        self._JobType = new_JobType
    else:
        raise ValueError("JobType must be a non-empty string.")
```

1 usage

```python
@property
def PostedDate(self):
    return self._PostedDate
```

```python
@PostedDate.setter
def PostedDate(self, new_PostedDate):
    if isinstance(new_PostedDate, datetime):
        self._PostedDate =new_PostedDate
    else:
        raise ValueError("PostedDate must be a datetime object.")
```

1 usage (1 dynamic)

```python
def retrieve_job_listings(self):
    try:
        self._db_connector.open_connection()
        query = ("SELECT C.CompanyName,J.JobTitle,J.JobDescription,J.JobLocation,"
                 "J.Salary,J.JobType FROM companies C JOIN Jobs J on J.CompanyID=C.CompanyID ")

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query)
            job_listings = cursor.fetchall()
            for job in job_listings:
                company_name,job_title,job_description,job_location,salary,job_type = job
                print(f"Company Name: {company_name},Job Title: {job_title},JobDescription :{job_description} , "
                      f"JobLocation:{job_location},Salary: {salary},JobType:{job_type}")

    except Exception as e:
        print(f"Error retrieving job listings: {e}")

    finally:
        self._db_connector.close_connection()
```

1 usage (1 dynamic)

```python
def search_job_listings_by_salary_range(self, min_salary, max_salary):
    try:
        self._db_connector.open_connection()
        query = ("SELECT J.JobTitle, C.CompanyName, J.Salary FROM Jobs J "
                 "JOIN Companies C on J.CompanyID=C.CompanyID "
                 "WHERE J.Salary BETWEEN %s AND %s")
        values = (min_salary, max_salary)

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query, values)
            job_listings = cursor.fetchall()
            for job_title,  company_name, salary in job_listings:
                print(f"Job Title: {job_title},Company Name: {company_name},Salary :{salary}")

    except Exception as e:
        print(f"Error searching job listings: {e}")

    finally:
        self._db_connector.close_connection()
def Apply(self, applicant_id, cover_letter):
    #similiar method created in create_applicant_profile in Application.py
   pass


def GetApplicants(self):
    # similiar method created in get_applicants in Application.py
    pass
```

Company Class:

Attributes:

• CompanyID (int): A unique identifier for each company.

• CompanyName (string): The name of the hiring company.

• Location (string): The location of the company.

Methods:

• PostJob(jobTitle: string, jobDescription: string, jobLocation: string, salary: decimal, jobType: string): Allows a company to post a new job listing.

• GetJobs(): List: Retrieves a list of job listings posted by the company.

```
class Company:
    def __init__(self, CompanyID, CompanyName, Location):
        self._CompanyID = CompanyID
        self._CompanyName = CompanyName
        self._Location = Location
    1 usage
    @property
    def CompanyID(self):
        return self._CompanyID

    @CompanyID.setter
    def CompanyID(self, new_CompanyID):
        if isinstance(new_CompanyID, int) and new_CompanyID > 0:
            self._CompanyID = new_CompanyID
        else:
            raise ValueError("CompanyID must be a positive integer.")

    1 usage
    @property
    def CompanyName(self):
        return self._CompanyName

    @CompanyName.setter
    def CompanyName(self, new_CompanyName):
        if isinstance(new_CompanyName, str) and len(new_CompanyName) > 0:
            self._CompanyName = new_CompanyName
        else:
            raise ValueError("CompanyName must be a non-empty string.")

    1 usage
    @property
    def Location(self):
        return self._Location
```

```
    @Location.setter
    def Location(self, new_Location):
        if isinstance(new_Location, str) and len(new_Location) > 0:
            self._Location =  new_Location
        else:
            raise ValueError("Location must be a non-empty string.")
    def Post_job(self):
        #similiar method done in joblisting as company_post_job_listing
        pass
```

Applicant Class:

Attributes:

• ApplicantID (int): A unique identifier for each applicant.

• FirstName (string): The first name of the applicant.

• LastName (string): The last name of the applicant.

• Email (string): The email address of the applicant.

• Phone (string): The phone number of the applicant.

• Resume (string): The applicant's resume or a reference to the resume file.

Methods:

• CreateProfile(email: string, firstName: string, lastName: string, phone: string): Allows applicants to create a profile with their contact information.

• ApplyForJob(jobID: int, coverLetter: string): Enables applicants to apply for a specific job listing.

```
from Exception.Exception import InvalidEmailFormat

class Applicant:
    def __init__(self, ApplicantID, FirstName, LastName, Email, Phone, Resume):
        self._ApplicantID = ApplicantID
        self._FirstName = FirstName
        self._LastName = LastName
        self._Email = Email
        self._Phone = Phone
        self._Resume = Resume
```

```
1 usage
@property
def ApplicantID(self):
    return self._ApplicantID

@ApplicantID.setter
def ApplicantID(self, new_ApplicantID):
    if isinstance(new_ApplicantID, int) and new_ApplicantID > 0:
        self._ApplicantID = new_ApplicantID
    else:
        raise ValueError("ApplicantID must be a positive integer.")

1 usage
@property
def FirstName(self):
    return self._FirstName

@FirstName.setter
def FirstName(self, new_FirstName):
    if isinstance(new_FirstName, str) and len(new_FirstName) > 0:
        self._FirstName = new_FirstName
    else:
        raise ValueError("FirstName must be a non-empty string.")

1 usage
@property
def LastName(self):
    return self._LastName
```

```
@LastName.setter
def LastName(self, new_LastName):
    if isinstance( new_LastName, str) and len( new_LastName) > 0:
        self._LastName = new_LastName
    else:
        raise ValueError("LastName must be a non-empty string.")

1 usage
@property
def Email(self):
    return self._Email

@Email.setter
def Email(self, new_Email):
    if "@" in new_Email and "." in new_Email:
        self._Email = new_Email
    else:
        raise InvalidEmailFormat
1 usage
@property
def Phone(self):
    return self._Phone

@Phone.setter
def Phone(self, new_Phone):
    if len(new_Phone) == 10 and new_Phone.isdigit():
        self._Phone = new_Phone
    else:
        raise ValueError("Invalid phone number format.")
```

JobApplication Class:

Attributes:

• ApplicationID (int): A unique identifier for each job application.

• JobID (int): A reference to the job listing.

• ApplicantID (int): A reference to the applicant.

• ApplicationDate (DateTime): The date and time when the application was submitted.

• CoverLetter (string): The cover letter submitted with the application.

```python
from datetime import datetime

class JobApplication:
    def __init__(self, ApplicationID, JobID, ApplicantID, ApplicationDate, CoverLetter):
        self._ApplicationID = ApplicationID
        self._JobID = JobID
        self._ApplicantID = ApplicantID
        self._ApplicationDate = ApplicationDate
        self._CoverLetter = CoverLetter

    @property
    def ApplicationID(self):
        return self._ApplicationID

    @ApplicationID.setter
    def ApplicationID(self, new_ApplicationID):
        if isinstance(new_ApplicationID, int) and new_ApplicationID > 0:
            self._ApplicationID = new_ApplicationID
        else:
            raise ValueError("ApplicationID must be a positive integer.")

    1 usage
    @property
    def JobID(self):
        return self._JobID

    @JobID.setter
    def JobID(self, new_JobID):
        if isinstance(new_JobID, int) and new_JobID > 0:
            self._JobID = new_JobID
        else:
            raise ValueError("JobID must be a positive integer.")

    1 usage
    @property
    def ApplicantID(self):
        return self._ApplicantID
```

```python
@ApplicantID.setter
def ApplicantID(self, new_ApplicantID):
    if isinstance(new_ApplicantID, int) and new_ApplicantID > 0:
        self._ApplicantID = new_ApplicantID
    else:
        raise ValueError("ApplicantID must be a positive integer.")

1 usage
@property
def ApplicationDate(self):
    return self._ApplicationDate

@ApplicationDate.setter
def ApplicationDate(self, new_ApplicationDate):

    if isinstance(new_ApplicationDate, datetime):
        self._ApplicationDate = new_ApplicationDate
    else:
        raise ValueError("Invalid datetime format.")

1 usage
@property
def CoverLetter(self):
    return self._CoverLetter
```

```python
@CoverLetter.setter
def CoverLetter(self, new_CoverLetter):
    if isinstance(new_CoverLetter, str) and len(new_CoverLetter) > 0:
        self._CoverLetter = new_CoverLetter
    else:
        raise ValueError("CoverLetter must be a non-empty string.")

1 usage (1 dynamic)
def apply_for_job(self,ApplicationID, JobID, ApplicantID, CoverLetter):
    try:
        self._db_connector.open_connection()
        ApplicationDate = datetime.now()


        query = ("INSERT INTO Applications (ApplicationID, JobID,ApplicantID,ApplicationDate, CoverLetter) "
                 "VALUES (%s, %s, %s, %s,%s)")
        values = (ApplicationID, JobID, ApplicantID, ApplicationDate, CoverLetter)

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query, values)

        self._db_connector.connection.commit()
        print("Application submitted successfully.")

    except Exception as e:
        print(f"Error applying for the job: {e}")

    finally:
        self._db_connector.close_connection()
```

3.Exceptions handling

Create and implement the following exceptions in your application.

• Invalid Email Format Handling:

o In the Job Board application, during the applicant registration process, users are required to enter their email addresses. Write a program that prompts the user to input an email address and implement exception handling to ensure that the email address follows a valid format (e.g., contains "@" and a valid domain). If the input is not valid, catch the exception and display an error message. If it is valid, proceed with registration.

• Salary Calculation Handling:

o Create a program that calculates the average salary offered by companies for job listings. Implement exception handling to ensure that the salary values are non-negative when computing the average. If any salary is negative or invalid, catch the exception and display an error message, indicating the problematic job listings.

• File Upload Exception Handling:

o In the Job Board application, applicants can upload their resumes as files. Write a program that handles file uploads and implements exception handling to catch and handle potential errors, such as file not found, file size exceeded, or file format not supported. Provide appropriate error messages in each case.

• Application Deadline Handling: o Develop a program that checks whether a job application is submitted before the application deadline. Implement exception handling to catch situations where an applicant tries to submit an application after the deadline has passed. Display a message indicating that the application is no longer accepted.

• Database Connection Handling:

o In the Job Board application, database connectivity is crucial. Create a program that establishes a connection to the database to retrieve job listings. Implement exception handling to catch database-related exceptions, such as connection errors or SQL query errors. Display appropriate error messages and ensure graceful handling of these exceptions.

```
class InvalidEmailFormat(Exception):
    def __init__(self, message="Invalid email format. Please enter a valid email address."):
        self.message = message
        super().__init__(self.message)

2 usages
class NegativeSalaryException(Exception):
    def __init__(self, message="Salary cannot be negative. Please enter a valid salary."):
        self.message = message
        super().__init__(self.message)

class FileUploadException(Exception):
    def __init__(self, message="Error during file upload."):
        self.message = message
        super().__init__(self.message)

class DeadlineExceededException(Exception):
    def __init__(self, message="Application deadline has passed. Applications are no longer accepted."):
        self.message = message
        super().__init__(self.message)

class DatabaseConnectionException(Exception):
    def __init__(self, message="Error connecting to the database. Please try again later."):
        self.message = message
        super().__init__(self.message)
```

4.Database Connectivity

```
import mysql.connector

class DBConnection :
    def __init__(self):
        self.con = mysql.connector.connect(
            host="localhost",
            port = "3306",
            user = "root",
            password = "9927559686",
            database = "careerhub"
        )

    def getDBConnection(self):
        return self.con.cursor()
```

Create and implement the following tasks in your application.

- Job Listing Retrieval:

Write a program that connects to the database and retrieves all job listings from the "Jobs" table. Implement database connectivity using Entity Framework and display the job titles, company names, and salaries.

```python
@PostedDate.setter
def PostedDate(self, new_PostedDate):
    if isinstance(new_PostedDate, datetime):
        self._PostedDate =new_PostedDate
    else:
        raise ValueError("PostedDate must be a datetime object.")
```

```python
1 usage (1 dynamic)
def retrieve_job_listings(self):
    try:
        self._db_connector.open_connection()
        query = ("SELECT C.CompanyName,J.JobTitle,J.JobDescription,J.JobLocation,"
                 "J.Salary,J.JobType FROM companies C JOIN Jobs J on J.CompanyID=C.CompanyID ")

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query)
            job_listings = cursor.fetchall()
            for job in job_listings:
                company_name,job_title,job_description,job_location,salary,job_type = job
                print(f"Company Name: {company_name},Job Title: {job_title},JobDescription :{job_description} , "
                      f"JobLocation:{job_location},Salary: {salary},JobType:{job_type}")

    except Exception as e:
        print(f"Error retrieving job listings: {e}")

    finally:
        self._db_connector.close_connection()
```

• Applicant Profile Creation:

Create a program that allows applicants to create a profile by entering their information. Implement database connectivity to insert the applicant's data into the "Applicants" table. Handle potential database-related exceptions.

```python
    1 usage (1 dynamic)
    def create_applicant_profile(self, ApplicantID, FirstName, LastName, Email, Phone, Resume):
        try:
            self._db_connector.open_connection()

            query = ("INSERT INTO Applicants (ApplicantID, FirstName, LastName, Email, Phone, Resume) VALUES"
                     " (%s, %s, %s, %s, %s, %s)")
            values = (ApplicantID, FirstName, LastName, Email, Phone, Resume)

            with self._db_connector.connection.cursor() as cursor:
                cursor.execute(query, values)

            self._db_connector.connection.commit()
            print("Applicant profile created successfully.")

        except Exception as e:
            print(f"Error creating applicant profile: {e}")

        finally:
            self._db_connector.close_connection()
```

• Job Application Submission:

Develop a program that allows applicants to apply for a specific job listing. Implement database connectivity to insert the job application details into the "Applications" table, including the applicant's ID and the job ID. Ensure that the program handles database connectivity and insertion exceptions.

```python
@CoverLetter.setter
def CoverLetter(self, new_CoverLetter):
    if isinstance(new_CoverLetter, str) and len(new_CoverLetter) > 0:
        self._CoverLetter = new_CoverLetter
    else:
        raise ValueError("CoverLetter must be a non-empty string.")

1 usage (1 dynamic)
def apply_for_job(self,ApplicationID, JobID, ApplicantID, CoverLetter):
    try:
        self._db_connector.open_connection()
        ApplicationDate = datetime.now()


        query = ("INSERT INTO Applications (ApplicationID, JobID,ApplicantID,ApplicationDate, CoverLetter) "
                 "VALUES (%s, %s, %s, %s,%s)")
        values = (ApplicationID, JobID, ApplicantID, ApplicationDate, CoverLetter)

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query, values)

        self._db_connector.connection.commit()
        print("Application submitted successfully.")

    except Exception as e:
        print(f"Error applying for the job: {e}")

    finally:
        self._db_connector.close_connection()
```

• Company Job Posting:

Write a program that enables companies to post new job listings. Implement database connectivity to insert job listings into the "Jobs" table, including the company's ID. Handle database-related exceptions and ensure the job posting is successful.

```python
1 usage (1 dynamic)
def company_post_job_listing(self,JobID,CompanyID,JobTitle, JobDescription, JobLocation, Salary, JobType):
    try:
        self._db_connector.open_connection()

        query = ("INSERT INTO Jobs (JobID,CompanyID, JobTitle, JobDescription, JobLocation, Salary, JobType, PostedDate) "
                 "VALUES (%s, %s, %s, %s, %s, %s, %s,%s)")
        values = (JobID,CompanyID, JobTitle, JobDescription,
                  JobLocation, Salary, JobType, datetime.now(),)

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query, values)

        self._db_connector.connection.commit()
        print("Job listing posted successfully.")

    except Exception as e:
        print(f"Error posting job listing: {e}")

    finally:
        self._db_connector.close_connection()
```

• Salary Range Query:

Create a program that allows users to search for job listings within a specified salary range. Implement database connectivity to retrieve job listings that match the user's criteria, including job titles, company names, and salaries. Ensure the program handles database connectivity and query exceptions.

```
1 usage (1 dynamic)
def search_job_listings_by_salary_range(self, min_salary, max_salary):
    try:
        self._db_connector.open_connection()
        query = ("SELECT J.JobTitle, C.CompanyName, J.Salary FROM Jobs J "
                 "JOIN Companies C on J.CompanyID=C.CompanyID "
                 "WHERE J.Salary BETWEEN %s AND %s")
        values = (min_salary, max_salary)

        with self._db_connector.connection.cursor() as cursor:
            cursor.execute(query, values)
            job_listings = cursor.fetchall()
            for job_title,  company_name, salary in job_listings:
                print(f"Job Title: {job_title},Company Name: {company_name},Salary :{salary}")

    except Exception as e:
        print(f"Error searching job listings: {e}")

    finally:
        self._db_connector.close_connection()
def Apply(self, applicant_id, cover_letter):
    #similiar method created in create_applicant_profile in Application.py
   pass


def GetApplicants(self):
    # similiar method created in get_applicants in Application.py
    pass
```