**Documentation**

This documentation provides an overview of the Python project for managing employee information, including a database and functionalities for adding, retrieving, updating, and calculating salaries.

Project Structure:

* src/models.py: Contains Python classes defining the database models for Employees, Teams, and Bonuses.
* src/database\_manager.py: Implements the DatabaseManager class responsible for interacting with the database using SQLAlchemy.
* tests/database\_manager.py: Contains unit tests for the DatabaseManager class.

Database Models (src/models.py):

Employees:

* Attributes:
* id (Integer, primary key)
* first\_name (String)
* last\_name (String)
* base\_salary (Integer)
* birth\_date (Date)
* hire\_date (Date)

Teams:

* Attributes:
* id (Integer, primary key)
* leader\_id (Integer, ForeignKey referencing Employees.id)
* employee\_id (Integer, ForeignKey referencing Employees.id)

Bonuses:

* Attributes:
* id (Integer, primary key)
* type (String)
* value (Integer)

Database Manager (src/database\_manager.py):

The DatabaseManager class provides methods for interacting with the database:

Initialization:

Establishes a connection to the database using the provided URL (defaults to sqlite:///employees.db).

Creates the database schema if it doesn't exist.

Adding Data:

Methods to add new employee, team, and bonus entries to the database.

Retrieving Data:

Methods to retrieve data from the database, including:

All employees

Specific employees based on various criteria (e.g., name, birthdate)

Team members of a specific leader

Information about bonuses (yearly bonus, leader bonus per member)

Methods to check if an employee is a team leader.

Salary Calculation:

Calculates the total salary for an employee considering:

Base salary

Years with the company (based on hire date)

Yearly bonus

Potential leader bonus (if the employee is a leader)

Provides options to:

Simply calculate the salary

Calculate the salary and print a simulated email message

Calculate the salary and return a tuple containing salary, employee name, and email message

Other Methods:

Removing and updating employee data

Populating the database with sample data (for testing purposes)

Unit Tests (database\_manager\_test.py):

These tests verify the functionalities of the DatabaseManager class, focusing on employee and team management aspects in an employee management system. They utilize the Employees, Teams, and DatabaseManager classes from the src.models and src.database\_manager modules, respectively.

Individual Test Descriptions:

1. test\_john\_doe\_is\_team\_leader:

*Purpose*: Checks if an employee named John Doe with a specific birthdate exists and is a team leader.

*Steps*:

Retrieves all employees using db\_manager.get\_all\_employees().

Asserts that the retrieved employees list is not None.

Filters the employees list to find John Doe based on first and last name.

Asserts that exactly one John Doe is found.

Verifies that the John Doe's birthdate matches the expected date.

Calls db\_manager.is\_leader to check if John Doe is a team leader and asserts the result.

1. test\_john\_doe\_team\_members:

*Purpose*: Validates that John Doe's team members are the expected individuals.

*Steps*:

Follows similar steps as test\_john\_doe\_is\_team\_leader to retrieve and filter employees for John Doe.

Retrieves John Doe's team members using db\_manager.get\_team\_members.

Asserts that the team member list has two members.

Converts team members' names to a list of formatted strings (first and last name combined).

Asserts that both expected team member names ("Myrta Torkelson" and "Jettie Lynch") are present in the list.

1. test\_tomas\_andre\_not\_in\_john\_doe\_team:

*Purpose*: Confirms that Tomas Andre is not part of John Doe's team.

*Steps*:

Mirrors the initial steps of test\_john\_doe\_team\_members to retrieve and filter employees for John Doe.

Retrieves John Doe's team members using db\_manager.get\_team\_members.

Converts team members' names to a list of formatted strings.

Asserts that "Tomas Andre" is not present in the list of team member names.

1. test\_gretchen\_watford\_base\_salary:

*Purpose*: Verifies that Gretchen Walford's base salary matches the expected value.

*Steps*:

Retrieves all employees using db\_manager.get\_all\_employees().

Asserts that the retrieved employees list is not None.

Filters the employee list to find Gretchen Walford based on first and last name.

Asserts that exactly one Gretchen Walford is found.

Asserts that Gretchen Walford's base salary is equal to 4000.

1. test\_tomas\_andre\_not\_team\_leader:

*Purpose*: Ensures that Tomas Andre is not a team leader and attempts to retrieve his non-existent team members (should not raise errors).

*Steps*:

Retrieves all employees using db\_manager.get\_all\_employees().

Asserts that the retrieved employees list is not None.

Filters the employee list to find Tomas Andre based on first and last name.

Asserts that exactly one Tomas Andre is found.

Calls db\_manager.is\_leader to check if Tomas Andre is a team leader and asserts that he is not.

Attempts to retrieve Tomas Andre's team members using db\_manager.get\_team\_members (expected to return an empty list, not an error).

1. test\_jude\_overcash\_not\_in\_database:

*Purpose*: Confirms that Jude Overcash is not present in the employee database.

*Steps*:a

Retrieves all employees using db\_manager.get\_all\_employees().

Asserts that the retrieved employees list is not None.

Filters the employee list to find Jude Overcash based on first and last name.

Asserts that the filtered list is empty (no Jude Overcash found).

1. test\_employee\_salary\_without\_leadership\_bonus (failing due to incorrect date):

*Purpose*: Calculates the salary for an employee who is not a team leader and validates the calculated value

SQLAlchemy Database Models Documentation (models.py, database\_manager.py)

This documentation provides an overview of the SQLAlchemy database models used in the project for managing employee information.

Project Structure

The project consists of the following database models defined using SQLAlchemy:

* Employees: Represents individual employees in the organization.
* Teams: Defines the relationships between employees, including team leaders and team members.
* Bonuses: Stores information about various types of bonuses provided to employees.

Database Models

Employees

* id (Integer, Primary Key): Unique identifier for each employee.
* first\_name (String): First name of the employee.
* last\_name (String): Last name of the employee.
* base\_salary (Integer): Base salary of the employee.
* birth\_date (Date): Date of birth of the employee.
* hire\_date (Date): Date when the employee was hired.

Teams

* id (Integer, Primary Key): Unique identifier for each team.
* leader\_id (Integer, Foreign Key referencing Employees.id): ID of the team leader.
* employee\_id (Integer, Foreign Key referencing Employees.id): ID of a team member.

Bonuses

* id (Integer, Primary Key): Unique identifier for each bonus.
* type (String): Type of the bonus (e.g., yearly bonus, leader bonus per member).
* value (Integer): Value of the bonus.

Database Manager

The DatabaseManager class interacts with the database using SQLAlchemy. It provides methods for adding, retrieving, updating, and deleting data from the database.

Key Features of SQLAlchemy

* Declarative Base: Defines database models using Python classes with SQLAlchemy's declarative base.
* SQL Expression Language: Constructs SQL queries using Python constructs for database interactions.
* ORM: Maps database tables to Python classes and enables seamless interaction between the database and application code.
* Session Management: Provides transactional scope for database operations using SQLAlchemy's session management.
* Usage: To interact with the database, instantiate the DatabaseManager class and use its methods to perform CRUD operations on employee data, manage teams, and handle bonuses.

**CI/CD Documentation: Python Application Test with GitHub Actions**

Workflow Configuration

The CI/CD workflow is defined in a YAML file named main.yml located in the .github/workflows directory of the repository. The workflow is triggered on pushes to the main branch and pull requests targeting the main branch.

name: Python application test with Github Actions

on:

push:

branches: [main]

pull\_request:

branches: [main]

Jobs: The workflow consists of a single job named build, which runs on an Ubuntu latest environment.

jobs:

build:

runs-on: ubuntu-latest

Steps: The job comprises several steps to set up the Python environment, install dependencies, perform linting with Flake8, and run tests with pytest.

steps:

Checkout Repository: Uses the actions/checkout@v2 action to fetch the repository's source code.

- uses: [actions/checkout@v2](mailto:actions/checkout@v2)

Set up Python 3.8: Utilizes the actions/setup-python@v2 action to set up Python version 3.8 for the environment.

- name: Set up Python 3.8

uses: actions/setup-python@v2

with:

python-version: 3.8

Install Dependencies: Upgrades pip and installs necessary dependencies including flake8, pytest, and sqlalchemy.

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install flake8 pytest sqlalchemy

Lint with Flake8: Runs Flake8 to perform linting checks on the Python code. Flags syntax errors, undefined names, and enforces coding standards. Sets maximum line length and complexity thresholds.

- name: Lint with flake8

run: |

# stop the build if there are Python syntax errors or undefined names

flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics

# exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide

flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 –statistics

Test with Pytest: Executes pytest to run the test suite for the Python application.

- name: Test with pytest

run: |

pytest

Triggering the Workflow: The workflow automatically triggers on pushes to the main branch and pull requests targeting the main branch.

CI/CD Process: Upon triggering, the workflow sets up the Python environment, installs dependencies, performs linting checks, and runs tests.

Reviewing Results: The workflow provides feedback on code quality and test results directly in the GitHub Actions interface.

Integration with Pull Requests: Pull requests are automatically checked for code quality and test coverage before being merged into the main branch.

Conclusion: This CI/CD setup using GitHub Actions ensures code consistency, quality, and reliability by automating the testing process for the Python application. It helps maintain a high standard of code hygiene and facilitates collaboration among developers contributing to the project.