

PAYXPERT PAYROLL MANAGEMENT SYSTEM - CASE STUDY REPORT

Title: PayXpert Payroll Management System

Submitted by: Esaq A & Mahalakshmi R

Abstract

The PayXpert Payroll Management System is a comprehensive, modular software solution designed to automate and streamline payroll processes for organizations. It simplifies the management of employee records, salary calculations, tax deductions, and financial reporting through a menu-driven, user-friendly interface.

PayXpert follows object-oriented design principles and implements a layered architecture to ensure clean code, scalability, and maintainability. The system integrates with a secure SQL database to store employee information, payroll records, tax details, and financial transactions.

Key features of the system include automated payroll generation, dynamic net salary calculations considering overtime, deductions, and taxes, as well as detailed financial reporting for management insights. The solution incorporates robust exception handling through custom exceptions to manage errors such as invalid inputs, missing records, and database connectivity issues.

PayXpert emphasizes accurate financial computations, secure data handling, and efficient record management, providing organizations with a reliable tool to ensure payroll accuracy, tax compliance, and streamlined HR operations.

Introduction

The PayXpert Payroll Management System is designed to automate and simplify payroll processing for organizations. This project focuses on developing a menu-driven, modular system that efficiently manages employee records, payroll calculations, tax details, and financial reports while ensuring database integrity and user-friendly interaction.

Objectives

- Automate payroll generation and record management.
- Maintain accurate employee records including personal, financial, and employment details.
- Calculate net salary considering overtime, deductions, and applicable taxes.
- Store all records in a secure MySQL database.
- Implement proper exception handling and input validations.
- Provide modular, scalable, and maintainable code structure.

System Architecture

Layer	Description
Entity/Model Layer	Contains simple classes representing real-world entities like Employee, Payroll, Tax, FinancialRecord. These classes hold only data with private properties, getters, setters, and constructors.
DAO/Service Layer	Defines interfaces and their implementations for performing business operations like employee management, payroll generation, tax calculation and financial record management. This layer interacts with DB
Exception Layer	Contains custom exception classes to handle application-specific errors, enhancing system reliability and user experience.
Utility Layer	Provides reusable components for database connectivity and configuration management. Ensures centralized handling of connections and properties.
Main Application Layer	The entry point of the system, presenting a menu-driven interface that allows users to access various functionalities interactively.

Database Design

Create following tables in SQL Schema with appropriate class and write the unit test case for the application. SQL Tables:

Employee Table

- EmployeeID (Primary Key, INT)
- FirstName, LastName (VARCHAR)
- DateOfBirth (DATE)
- Gender (VARCHAR)
- Email (VARCHAR)
- PhoneNumber (VARCHAR)
- Address (VARCHAR)
- Position (VARCHAR)
- JoiningDate (DATE)
- TerminationDate (DATE, Nullable)

Query:

```
create table employee (employeeid int primary key, firstname varchar(50) not null, lastname
varchar(50) not null, dateofbirth date not null, gender varchar(10) not null, email
varchar(100) not null, phonenumber varchar(15), address varchar(50), position varchar(50)
not null, joiningdate date not null, terminationdate date null);
```

```
mysql> create database PayXpert;
Query OK, 1 row affected (0.05 sec)

mysql> use payXpert;
Database changed
mysql> CREATE TABLE Employee (EmployeeID INT PRIMARY KEY, FirstName VARCHAR(50) NOT NULL, LastName VARCHAR(50) NOT NULL, DateOfBirth DATE NOT NULL, Gender VARCHAR(10) NOT NULL, Email VARCHAR(100) NOT NULL, PhoneNumber VARCHAR(15), Address VARCHAR(255), Position VARCHAR(50) NOT NULL, JoiningDate DATE NOT NULL, TerminationDate DATE NULL);
Query OK, 0 rows affected (0.09 sec)

mysql>
```

Payroll Table

- PayrollID (Primary Key, INT)
- EmployeeID (Foreign Key)
- PayPeriodStartDate, PayPeriodEndDate (DATE)
- BasicSalary, OvertimePay, Deductions, NetSalary (DECIMAL)

Query:

```
create table payroll (payrollid int primary key, employeeid int, payperiodstartdate date not
null, payperiodenddate date not null, basicsalary decimal(10,2) not null, overtimepay
decimal(10,2), deductions decimal(10,2), netsalary decimal(10,2), foreign key (employeeid)
references employee(employeeid));
```

```
mysql> create table payroll (payrollid int primary key, employeeid int, payperiodstartdate date not n
ull, basicsalary decimal(10,2) not null, overtimepay decimal(10,2), deductions decimal(10,2), netsalary decimal(10,2), foreign key (e
mployeeid) references employee(employeeid));
Query OK, 0 rows affected (0.16 sec)

mysql> |
```

Tax Table

- TaxID (Primary Key, INT)
- EmployeeID (Foreign Key)
- TaxAmount (DECIMAL)
- TaxYear (INT)

Query:

```
create table tax (taxid int primary key, employeeid int, taxyear int not null, taxableincome
decimal(10,2) not null, taxamount decimal(10,2) not null, foreign key (employeeid)
references employee(employeeid));
```

```
mysql> create table tax (taxid int primary key, employeeid int, taxyear int not null, taxableincome decimal(10,2) not null, taxamount decimal(10,2) not null, foreign key (employeeid) references employee(employeeid));
Query OK, 0 rows affected (0.12 sec)
```

Financial Report Table

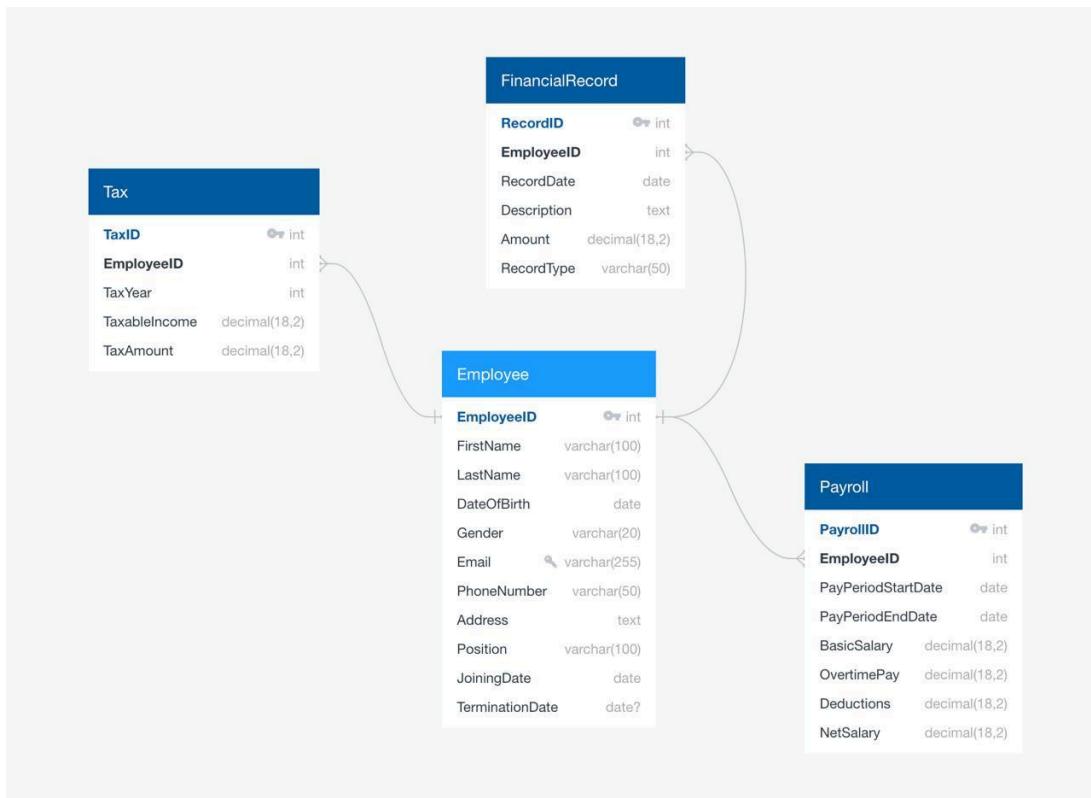
- RecordID.
- EmployeeID
- RecordDate:
- Description:
- Amount
- RecordType

Query:

```
create table financialrecord (recordid int primary key, employeeid int, recorddate date not null, description varchar(255), amount decimal(10,2) not null, recordtype varchar(50) not null, foreign key (employeeid) references employee(employeeid));
```

```
mysql> create table financialrecord (recordid int primary key, employeeid int, recorddate date not null, description varchar(255), amount decimal(10,2) not null, recordtype varchar(50) not null, foreign key (employeeid) references employee(employeeid));
ERROR 1050 (42S01): Table 'financialrecord' already exists
mysql>
```

ER Diagram



Create the model/entity classes corresponding to the schema within package entity with variables declared private, constructors (default and parametrized) and getters, setters)

Classes:

Employee:

- Properties: EmployeeID, FirstName, LastName, DateOfBirth, Gender, Email, PhoneNumber, Address, Position, JoiningDate, TerminationDate
- Methods: CalculateAge()

Code:

```
entity > 📁 employee.py > 📄 Employee
 1  from datetime import date
 2
 3  class Employee:
 4      def __init__(self, employee_id=None, first_name=None, last_name=None, date_of_birth=None, gender=None,
 5                   email=None, phone_number=None, address=None, position=None, joining_date=None,
 6                   termination_date=None):
 7          self.__employee_id = employee_id
 8          self.__first_name = first_name
 9          self.__last_name = last_name
10          self.__date_of_birth = date_of_birth
11          self.__gender = gender
12          self.__email = email
13          self.__phone_number = phone_number
14          self.__address = address
15          self.__position = position
16          self.__joining_date = joining_date
17          self.__termination_date = termination_date
18
19      def get_employee_id(self):
20          return self.__employee_id
21      def get_first_name(self):
22          return self.__first_name
23      def get_last_name(self):
24          return self.__last_name
25      def get_date_of_birth(self):
26          return self.__date_of_birth
27      def get_gender(self):
28          return self.__gender
29      def get_email(self):
30          return self.__email
31      def get_phone_number(self):
32          return self.__phone_number
33      def get_address(self):
34          return self.__address
35      def get_position(self):
36          return self.__position
37      def get_joining_date(self):
38          return self.__joining_date
39      def get_termination_date(self):
40          return self.__termination_date
```

```
entity > 🏠 employee.py > Employee
 3   class Employee:
41     def set_employee_id(self, employee_id):
42       self.__employee_id = employee_id
43     def set_first_name(self, first_name):
44       self.__first_name = first_name
45     def set_last_name(self, last_name):
46       self.__last_name = last_name
47     def set_date_of_birth(self, date_of_birth):
48       self.__date_of_birth = date_of_birth
49     def set_gender(self, gender):
50       self.__gender = gender
51     def set_email(self, email):
52       self.__email = email
53     def set_phone_number(self, phone_number):
54       self.__phone_number = phone_number
55     def set_address(self, address):
56       self.__address = address
57     def set_position(self, position):
58       self.__position = position
59     def set_joining_date(self, joining_date):
60       self.__joining_date = joining_date
61     def set_termination_date(self, termination_date):
62       self.__termination_date = termination_date
63     def calculate_age(self):
64       if self.__date_of_birth:
65         today = date.today()
66         dob = self.__date_of_birth
67         age = today.year - dob.year - ((today.month, today.day) < (dob.month, dob.day))
68         return age
69       return None
70
71     def __str__(self):
72       return (f"Employee ID: {self.__employee_id}\n"
73               f"Name: {self.__first_name} {self.__last_name}\n"
74               f"DOB: {self.__date_of_birth}\n"
75               f"Gender: {self.__gender}\n"
76               f"Email: {self.__email}\n"
77               f"Phone: {self.__phone_number}\n"
78               f"Address: {self.__address}\n"
79               f"Position: {self.__position}\n"
80               f"Joining Date: {self.__joining_date}\n"
81               f"Termination Date: {self.__termination_date if self.__termination_date else 'N/A'}")
```

Payroll:

- Properties: PayrollID, EmployeeID, PayPeriodStartDate, PayPeriodEndDate, BasicSalary, OvertimePay, Deductions, NetSalary

```
entity > payroll.py > Payroll
1  class Payroll:
2      def __init__(self, payroll_id=None, employee_id=None, pay_period_start_date=None, pay_period_end_date=None,
3                   basic_salary=None, overtime_pay=None, deductions=None, net_salary=None):
4          self.__payroll_id = payroll_id
5          self.__employee_id = employee_id
6          self.__pay_period_start_date = pay_period_start_date
7          self.__pay_period_end_date = pay_period_end_date
8          self.__basic_salary = basic_salary
9          self.__overtime_pay = overtime_pay
10         self.__deductions = deductions
11         self.__net_salary = net_salary
12     def get_payroll_id(self):
13         return self.__payroll_id
14     def get_employee_id(self):
15         return self.__employee_id
16     def get_pay_period_start_date(self):
17         return self.__pay_period_start_date
18     def get_pay_period_end_date(self):
19         return self.__pay_period_end_date
20     def get_basic_salary(self):
21         return self.__basic_salary
22     def get_overtime_pay(self):
23         return self.__overtime_pay
24     def get_deductions(self):
25         return self.__deductions
26     def get_net_salary(self):
27         return self.__net_salary
28     def set_payroll_id(self, payroll_id):
29         self.__payroll_id = payroll_id
30     def set_employee_id(self, employee_id):
31         self.__employee_id = employee_id
32     def set_pay_period_start_date(self, pay_period_start_date):
33         self.__pay_period_start_date = pay_period_start_date
34     def set_pay_period_end_date(self, pay_period_end_date):
35         self.__pay_period_end_date = pay_period_end_date
36     def set_basic_salary(self, basic_salary):
37         self.__basic_salary = basic_salary
38     def set_overtime_pay(self, overtime_pay):
39         self.__overtime_pay = overtime_pay
40     def set_deductions(self, deductions):
41         self.__deductions = deductions
42     def set_net_salary(self, net_salary):
43         self.__net_salary = net_salary
44     def __str__(self):
45         return (f"Payroll ID: {self.__payroll_id}\n"
46                 f"Employee ID: {self.__employee_id}\n"
47                 f"Pay Period: {self.__pay_period_start_date} to {self.__pay_period_end_date}\n"
48                 f"Basic Salary: {self.__basic_salary}\n"
49                 f"Overtime Pay: {self.__overtime_pay}\n"
50                 f"Deductions: {self.__deductions}\n"
51                 f"Net Salary: {self.__net_salary}")
```

Tax:

- Properties: TaxID, EmployeeID, TaxYear, TaxableIncome, TaxAmount

```
entity > 📁 tax.py > 📄 Tax
1  class Tax:
2      def __init__(self, tax_id=None, employee_id=None, tax_year=None, taxable_income=None, tax_amount=None):
3          self.__tax_id = tax_id
4          self.__employee_id = employee_id
5          self.__tax_year = tax_year
6          self.__taxable_income = taxable_income
7          self.__tax_amount = tax_amount
8
9      def get_tax_id(self):
10         return self.__tax_id
11
12     def get_employee_id(self):
13         return self.__employee_id
14
15     def get_tax_year(self):
16         return self.__tax_year
17
18     def get_taxable_income(self):
19         return self.__taxable_income
20
21     def get_tax_amount(self):
22         return self.__tax_amount
23
24     def set_tax_id(self, tax_id):
25         self.__tax_id = tax_id
26
27     def set_employee_id(self, employee_id):
28         self.__employee_id = employee_id
29
30     def set_tax_year(self, tax_year):
31         self.__tax_year = tax_year
32
33     def set_taxable_income(self, taxable_income):
34         self.__taxable_income = taxable_income
35
36     def set_tax_amount(self, tax_amount):
37         self.__tax_amount = tax_amount
38
39     def __str__(self):
40         return (f"Tax ID: {self.__tax_id}\n"
41                 f"Employee ID: {self.__employee_id}\n"
42                 f"Tax Year: {self.__tax_year}\n"
43                 f"Taxable Income: {self.__taxable_income}\n"
44                 f"Tax Amount: {self.__tax_amount}")
```

Financial Record:

- Properties: RecordID, EmployeeID, RecordDate, Description, Amount, RecordType

EmployeeService (implements IEmployeeService):

- Methods: GetEmployeeById, GetAllEmployees, AddEmployee, UpdateEmployee, RemoveEmployee

```
dao > ipy employee_service.py > ...
1  from abc import ABC, abstractmethod
2
3  class IEmployeeService(ABC):
4      @abstractmethod
5      def get_employee_by_id(self, employee_id):
6          pass
7
8      @abstractmethod
9      def get_all_employees(self):
10         pass
11
12     @abstractmethod
13     def add_employee(self, employee_data):
14         pass
15
16     @abstractmethod
17     def update_employee(self, employee_data):
18         pass
19
20     @abstractmethod
21     def remove_employee(self, employee_id):
22         pass
```

PayrollService (implements IPayrollService):

- Methods: GeneratePayroll, GetPayrollById, GetPayrollsForEmployee, GetPayrollsForPeriod

```
dao > ipy payroll_service.py > ...
1  from abc import ABC, abstractmethod
2  from datetime import date
3
4  class IPayrollService(ABC):
5      @abstractmethod
6      def generate_payroll(self, employee_id: int, start_date_obj: date, end_date_obj: date,
7                          basic_salary: float, overtime_pay: float = 0.0, deductions: float = 0.0):
8          pass
9
10     @abstractmethod
11     def get_payroll_by_id(self, payroll_id):
12         pass
13
14     @abstractmethod
15     def get_payrolls_for_employee(self, employee_id):
16         pass
17
18     @abstractmethod
19     def get_payrolls_for_period(self, start_date, end_date):
20         pass
```

TaxService (implements ITaxService):

- Methods: CalculateTax, GetTaxById, GetTaxesForEmployee, GetTaxesForYear

```
dao > itax_service.py > ...
1  from abc import ABC, abstractmethod
2
3  class ITaxService(ABC):
4      @abstractmethod
5      def calculate_tax(self, employee_id: int, tax_year: int, taxable_income: float, tax_rate: float = 0.15):
6          pass
7
8      @abstractmethod
9      def get_tax_by_id(self, tax_id):
10         pass
11
12     @abstractmethod
13     def get_taxes_for_employee(self, employee_id):
14         pass
15
16     @abstractmethod
17     def get_taxes_for_year(self, tax_year):
18         pass
```

FinancialRecordService (implements IFinancialRecordService):

- Methods: AddFinancialRecord, GetFinancialRecordById, GetFinancialRecordsForEmployee, GetFinancialRecordsForDate

```
dao> ✘ ifinancial_record_service.py > ...
1  from abc import ABC, abstractmethod
2  from datetime import date
3
4  class IFinancialRecordService(ABC):
5      @abstractmethod
6      def add_financial_record(self, employee_id: int, record_date_obj: date, description: str, amount: float, record_type: str):
7          pass
8
9      @abstractmethod
10     def get_financial_record_by_id(self, record_id):
11         pass
12
13     @abstractmethod
14     def get_financial_records_for_employee(self, employee_id):
15         pass
16
17     @abstractmethod
18     def get_financial_records_for_date(self, record_date):
19         pass
20
```

DatabaseContext:

- A class responsible for handling database connections and interactions.

```
1  import configparser
2  import os
3
4  class DBPropertyUtil:
5      @staticmethod
6      def get_connection_properties(config_file_name="db.properties"):
7          """
8              Reads database connection properties from a .properties file.
9              Expects a section like [mysql] with keys: host, user, password, database, port (optional).
10             Returns a dictionary of these properties.
11          """
12          config = configparser.ConfigParser()
13
14          if not os.path.exists(config_file_name):
15              raise FileNotFoundError(f"Configuration file '{config_file_name}' not found.")
16
17          config.read(config_file_name)
18
19          if 'mysql' in config:
20              mysql_config = config['mysql']
21
22              required_keys = ['host', 'user', 'password', 'database']
23              for key in required_keys:
24                  if key not in mysql_config:
25                      raise ValueError(f"Missing required key '{key}' in [mysql] section of '{config_file_name}'.")
26
27              connection_props = {
28                  'host': mysql_config['host'],
29                  'user': mysql_config['user'],
30                  'password': mysql_config['password'],
31                  'database': mysql_config['database']
32              }
33
34              if 'port' in mysql_config:
35                  try:
36                      connection_props['port'] = int(mysql_config['port'])
37                  except ValueError:
38                      raise ValueError(f"Invalid port number '{mysql_config['port']}' in [mysql] section. Port must be an integer.")
39
40              return connection_props
41          else:
42              raise ValueError(f"[mysql] section not found in '{config_file_name}'.")
```

ValidationService:

- A class for input validation and business rule enforcement.

```
util > ➜ validation_service.py > ...
1  import re
2  from datetime import date, datetime
3
4  class ValidationService:
5
6      @staticmethod
7      def is_valid_email(email: str) -> bool:
8          if not email or not isinstance(email, str):
9              return False
10         pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
11         return re.match(pattern, email) is not None
12
13     @staticmethod
14     def is_valid_phone_number(phone_number: str) -> bool:
15         if not phone_number or not isinstance(phone_number, str):
16             return False
17         pattern = r"^\+?[\\d\\s()]-{7,15}$"
18
19         return re.match(pattern, phone_number) is not None
20
21     @staticmethod
22     def is_valid_date_string(date_str: str, date_format: str = "%Y-%m-%d") -> bool:
23         if not date_str or not isinstance(date_str, str):
24             return False
25         try:
26             datetime.strptime(date_str, date_format)
27             return True
28         except ValueError:
29             return False
30
31     @staticmethod
32     def is_valid_future_date(date_obj: date) -> bool:
33         if not isinstance(date_obj, date):
34             return False
35         return date_obj > date.today()
36
37     @staticmethod
38     def is_valid_past_date(date_obj: date) -> bool:
39         if not isinstance(date_obj, date):
40             return False
41         return date_obj < date.today()
42
43     @staticmethod
44     def is_non_empty_string(value: str) -> bool:
45         return isinstance(value, str) and bool(value.strip())
46
47     @staticmethod
48     def is_positive_integer(value) -> bool:
49         return isinstance(value, int) and value > 0
50
51     @staticmethod
52     def is_positive_float(value) -> bool:
53         return isinstance(value, (int, float)) and value > 0
54
55     @staticmethod
56     def is_non_negative_float(value) -> bool:
57         return isinstance(value, (int, float)) and value >= 0
58
59     @staticmethod
60     def is_within_length_limit(text: str, max_length: int, min_length: int = 0) -> bool:
61         if not isinstance(text, str) or not isinstance(max_length, int) or not isinstance(min_length, int):
62             return False
63         return min_length <= len(text) <= max_length
64
65     @staticmethod
66     def is_valid_gender(gender: str, accepted_genders=None) -> bool:
67         if accepted_genders is None:
68             accepted_genders = ["male", "female", "other", "prefer not to say"]
69         return isinstance(gender, str) and gender.lower() in accepted_genders
70
71     @staticmethod
72     def validate_date_range(start_date: date, end_date: date) -> bool:
73         if not (isinstance(start_date, date) and isinstance(end_date, date)):
74             return False
75         return start_date <= end_date
76
```

ReportGenerator:

- A class for generating various reports based on payroll, tax, and financial record data.

```
util > ⚡ report_generator.py > ...
1  from datetime import date
2  from dao.payroll_service import PayrollService
3  from dao.tax_service import TaxService
4  from dao.financial_record_service import FinancialRecordService
5  from dao.employee_service import EmployeeService
6  from entity.employee import Employee
7  from entity.payroll import Payroll
8  from entity.tax import Tax
9  from entity.financial_record import FinancialRecord
10 from exception.custom_exceptions import EmployeeNotFoundException, DatabaseConnectionException, PayrollNotFoundException
11
12
13 class ReportGenerator:
14     def __init__(self, db_file_name="db.properties"):
15         """
16             Initializes the ReportGenerator with necessary service instances.
17         """
18         try:
19             self.employee_service = EmployeeService(db_file_name)
20             self.payroll_service = PayrollService(db_file_name)
21             self.tax_service = TaxService(db_file_name)
22             self.financial_record_service = FinancialRecordService(db_file_name)
23         except DatabaseConnectionException as e:
24             print(f"Failed to initialize ReportGenerator due to DB connection issue: {e}")
25             raise
26
27     def generate_employee_payslip(self, employee_id: int, payroll_id: int, output_format: str = "text"):
28         """
29             Generates a payslip for a specific employee and payroll period.
30             output_format can be "text" or "pdf".
31             output_path is used if format is "pdf".
32         """
33         try:
34             employee = self.employee_service.get_employee_by_id(employee_id)
35             payroll = self.payroll_service.get_payroll_by_id(payroll_id)
36
37             if not employee:
38                 raise EmployeeNotFoundException(f"Employee with ID {employee_id} not found (unexpectedly not raised by service).")
39             if not payroll:
40                 raise PayrollNotFoundException(f"Payroll with ID {payroll_id} not found (unexpectedly not raised by service).")
41             if payroll.get_employee_id() != employee_id:
42                 raise Exception(f"Data integrity issue: Payroll ID {payroll_id} does not belong to Employee ID {employee_id}.")
43
44             payslip_data = {
45                 "Employee Name": f"{employee.get_first_name()} {employee.get_last_name()}",
46                 "Employee ID": employee.get_employee_id(),
47                 "Position": employee.get_position(),
48                 "Pay Period Start": payroll.get_pay_period_start_date().isoformat(),
49                 "Pay Period End": payroll.get_pay_period_end_date().isoformat(),
50                 "Basic Salary": f"{payroll.get_basic_salary():.2f}",
51                 "Overtime Pay": f"{payroll.get_overtime_pay():.2f}",
52                 "Gross Salary": f"({payroll.get_basic_salary() + payroll.get_overtime_pay():.2f})",
53                 "Deductions": f"{payroll.get_deductions():.2f}",
54                 "Net Salary": f"{payroll.get_net_salary():.2f}",
55                 "Generation Date": date.today().isoformat()
56             }
57
58             if output_format.lower() == "text":
59                 report_str = f"--- PAYSPLIT ---\n"
60                 for key, value in payslip_data.items():
61                     report_str += f'{key}: {value}\n'
62                 report_str += "--- END OF PAYSPLIT ---"
63                 return report_str
64             else:
65                 return "Unsupported output format."
66
67         except EmployeeNotFoundException as enfe:
68             return str(enfe)
69         except Exception as e:
70             return f"Error generating payslip: {e}"
71
72     def generate_employee_tax_report(self, employee_id: int, tax_year: int, output_format: str = "text"):
73         """Generates a tax report for a specific employee and tax year.""""
```

```

util > ℹ report_generator.py > ...
13   class ReportGenerator:
14     def generate_employee_tax_report(self, employee_id: int, tax_year: int, output_format: str = "text"):
15       try:
16         employee = self.employee_service.get_employee_by_id(employee_id)
17         if not employee:
18           raise EmployeeNotFoundException(f"Employee with ID {employee_id} not found.")
19
20         taxes = self.tax_service.get_taxes_for_employee(employee_id)
21         year_specific_taxes = [t for t in taxes if t.get_tax_year() == tax_year]
22
23         if not year_specific_taxes:
24           return f"No tax records found for Employee ID {employee_id} for tax year {tax_year}."
25
26         report_title = f"Tax Report for {employee.get_first_name()} {employee.get_last_name()} (ID: {employee_id}) - Year {tax_year}"
27         tax_data_list = []
28         total_taxable_income = 0
29         total_tax_amount = 0
30
31         for tax_record in year_specific_taxes:
32           tax_data_list.append({
33             "Tax ID": tax_record.get_tax_id(),
34             "Taxable Income": f"{tax_record.get_taxable_income():.2f}",
35             "Tax Amount": f"{tax_record.get_tax_amount():.2f}"
36           })
37         total_taxable_income += tax_record.get_taxable_income()
38         total_tax_amount += tax_record.get_tax_amount()
39
40         summary = {
41           "Total Taxable Income for Year": f"{total_taxable_income:.2f}",
42           "Total Tax Paid for Year": f"{total_tax_amount:.2f}"
43         }
44
45         if output_format.lower() == "text":
46           report_str = f"--- {report_title} ---\n"
47           for record_dict in tax_data_list:
48             for key, value in record_dict.items():
49               report_str += f" {key}: {value}\n"
50             report_str += " ---\n"
51           report_str += "Summary:\n"
52           for key, value in summary.items():
53             report_str += f" {key}: {value}\n"
54           report_str += "--- END OF TAX REPORT ---"
55           return report_str
56         else:
57           return "Unsupported output format for tax report."
58
59     except EmployeeNotFoundException as enfe:
60       return str(enfe)
61     except Exception as e:
62       return f"Error generating tax report: {e}"
63
64   def generate_financial_summary_report(self, employee_id: int, start_date: date, end_date: date, output_format: str = "text"):
65     """Generates a financial summary report for an employee over a period."""
66     try:
67       employee = self.employee_service.get_employee_by_id(employee_id)
68       if not employee:
69         raise EmployeeNotFoundException(f"Employee with ID {employee_id} not found.")
70
71       all_records = self.financial_record_service.get_financial_records_for_employee(employee_id)
72
73       period_records = [
74         r for r in all_records
75         if r.get_record_date() and start_date <= r.get_record_date() <= end_date
76       ]
77
78       if not period_records:
79         return f"No financial records found for Employee ID {employee_id} between {start_date} and {end_date}."
80
81       report_title = (f"Financial Summary for {employee.get_first_name()} {employee.get_last_name()} (ID: {employee_id})\n"
82                     f"Period: {start_date.isoformat()} to {end_date.isoformat()}")
83
84       records_details = []
85       total_income = 0.0
86       total_expenses = 0.0
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
```

Business Logic Implementation

The PayXpert system incorporates well-defined business logic to ensure accurate payroll processing and financial management. Key business rules applied across modules include:

Employee Module Logic

- Unique Employee ID validation before adding new employees.
 - Age calculation derived from Date of Birth to ensure eligibility for employment.
 - Restriction: Employees marked with Termination Date are excluded from active payroll processing.

```
dao > EmployeeService.py > EmployeeService > _map_row_to_employee
1  from datetime import date
2  from dao.employee_service import IEmployeeService
3  from entity.employee import Employee
4  from util_db_conn_util import DBConnUtil
5  from exception.custom_exceptions import EmployeeNotFoundException, DatabaseConnectionException, InvalidInputException
6  import mysql.connector
7
8  class EmployeeService(IEmployeeService):
9      def __init__(self, db_file_name=db.properties):
10          self.db_file_name = db_file_name
11
12      def _get_cursor(self):
13          connection = DBConnUtil.get_connection(self.db_file_name)
14          if not connection:
15              raise DatabaseConnectionException("Failed to get database connection.")
16          return connection, connection.cursor(dictionary=True)
17
18      def _close_cursor_connection(self, connection, cursor):
19          if cursor:
20              cursor.close()
21
22      def _map_row_to_employee(self, row):
23          if row:
24              return Employee(employee_id=row.get("EmployeeID"), first_name=row.get("FirstName"), last_name=row.get("LastName"),
25                               date_of_birth=row.get("DateOfBirth"), gender=row.get("Gender"), email=row.get("Email"), phone_number=row.get("PhoneNumber"),
26                               address=row.get("Address"), position=row.get("Position"), joining_date=row.get("JoiningDate"),
27                               termination_date=row.get("TerminationDate"))
28          ]
29
30      return None
31
32  def get_employee_by_id(self, employee_id):
33      if not isinstance(employee_id, int) or employee_id <= 0:
34          raise InvalidInputException("Employee ID must be a positive integer.")
35      connection, cursor = None, None
36      try:
37          connection, cursor = self._get_cursor()
38          cursor.execute("SELECT * FROM Employee WHERE EmployeeID = %s", (employee_id,))
39          row = cursor.fetchone()
40          if not row:
41              raise EmployeeNotFoundException(f"Employee with ID {employee_id} not found.")
42          return self._map_row_to_employee(row)
43      except EmployeeNotFoundException:
44          raise
45      except mysql.connector.Error as err:
46          raise DatabaseConnectionException(f"Database error fetching employee by ID {employee_id}: {err}")
47      except Exception as e:
48          raise DatabaseConnectionException(f"Unexpected error fetching employee by ID {employee_id}: {e}")
49      finally:
50          self._close_cursor_connection(connection, cursor)
51
52  def get_all_employees(self):
53      connection, cursor = None, None
54      try:
55          connection, cursor = self._get_cursor()
56          cursor.execute("SELECT * FROM Employee")
57          rows = cursor.fetchall()
58          employees = [self._map_row_to_employee(row) for row in rows]
59          return employees
60      except mysql.connector.Error as err:
61          raise DatabaseConnectionException(f"Database error fetching all employees: {err}")
62      except Exception as e:
63          raise DatabaseConnectionException(f"Unexpected error fetching all employees: {e}")
64      finally:
65          self._close_cursor_connection(connection, cursor)
66
67  def add_employee(self, employee: Employee):
68      if not isinstance(employee, Employee):
69          raise InvalidInputException("Invalid employee data provided.")
70      connection, cursor = None, None
71      try:
72          connection, cursor = self._get_cursor()
73          sql = """
74              INSERT INTO Employee
75                  (FirstName, LastName, DateOfBirth, Gender, Email, PhoneNumber, Address, Position, JoiningDate, TerminationDate)
76                  VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
77          """
78
```

```

dao > EmployeeService > _map_row_to_employee
8   class EmployeeService(IEmployeeService):
66     def add_employee(self, employee: Employee):
77       params = (
78         employee.get_first_name(), employee.get_last_name(), employee.get_date_of_birth(),
79         employee.get_gender(), employee.get_email(), employee.get_phone_number(),
80         employee.get_address(), employee.get_position(), employee.get_joining_date(),
81         employee.get_termination_date()
82       )
83       cursor.execute(sql, params)
84       connection.commit()
85
86       last_id = cursor.lastrowid
87       if last_id:
88         employee.set_employee_id(last_id)
89       else:
90         cursor.execute("SELECT LAST_INSERT_ID()")
91         last_id = cursor.fetchone()['LAST_INSERT_ID']
92         employee.set_employee_id(last_id)
93
94       return employee
95     except mysql.connector.Error as err:
96       if connection: connection.rollback()
97       raise DatabaseConnectionException(f"Database error adding employee: {err}")
98     except Exception as e:
99       if connection: connection.rollback()
100      raise DatabaseConnectionException(f"Unexpected error adding employee: {e}")
101    finally:
102      self._close_cursor_connection(connection, cursor)
103
104  def update_employee(self, employee: Employee):
105    if not isinstance(employee, Employee) or not employee.get_employee_id():
106      raise InvalidInputException("Invalid employee data or missing Employee ID for update.")
107    connection, cursor = None, None
108    try:
109      connection, cursor = self._get_cursor()
110      cursor.execute("SELECT EmployeeID FROM Employee WHERE EmployeeID = %s", (employee.get_employee_id(),))
111      if not cursor.fetchone():
112        raise EmployeeNotFoundException(f"Employee with ID {employee.get_employee_id()} not found for update.")
113
114      sql = """
115        UPDATE Employee SET
116          FirstName = %s, LastName = %s, DateOfBirth = %s, Gender = %s, Email = %s,
117          PhoneNumber = %s, Address = %s, Position = %s, JoiningDate = %s, TerminationDate = %s
118        WHERE EmployeeID = %s
119        """
120
121      params = (
122        employee.get_first_name(), employee.get_last_name(), employee.get_date_of_birth(),
123        employee.get_gender(), employee.get_email(), employee.get_phone_number(),
124        employee.get_address(), employee.get_position(), employee.get_joining_date(),
125        employee.get_termination_date(), employee.get_employee_id()
126      )
127      cursor.execute(sql, params)
128      connection.commit()
129      if cursor.rowcount == 0:
130        raise EmployeeNotFoundException(f"Employee with ID {employee.get_employee_id()} was not updated (possibly deleted after check).")
131      return employee
132    except EmployeeNotFoundException:
133      if connection: connection.rollback()
134    except mysql.connector.Error as err:
135      if connection: connection.rollback()
136      raise DatabaseConnectionException(f"Database error updating employee ID {employee.get_employee_id()}: {err}")
137    except Exception as e:
138      if connection: connection.rollback()
139      raise DatabaseConnectionException(f"Unexpected error updating employee ID {employee.get_employee_id()}: {e}")
140    finally:
141      self._close_cursor_connection(connection, cursor)
142
143  def remove_employee(self, employee_id):
144    if not isinstance(employee_id, int) or employee_id <= 0:
145      raise InvalidInputException("Employee ID must be a positive integer.")
146    connection, cursor = None, None

```

Payroll Module Logic

- Net Salary Calculation:
Net Salary = Basic Salary + Overtime Pay - Deductions - Applicable Taxes
 - Automatic calculation of Net Salary based on entered salary components.
 - Enforcement of unique Payroll ID for each record to prevent duplication.
 - Ensures payroll is only processed for active employees

```
dao > payroll_service.py > PayrollService > _map_row_to_payroll
1  from datetime import date
2  from dao.ipayroll_service import IPayrollService
3  from entity.payroll import Payroll
4  from util.db_conn_util import DBConnUtil
5  from exception.custom_exceptions import PayrollGenerationException, EmployeeNotFoundException, DatabaseConnectionException, InvalidInputException, PayrollNotFoundException
6  from dao.employee_service import EmployeeService
7  import mysql.connector
8
9  class PayrollService(IPayrollService):
10     def __init__(self, db_file_name=db.properties):
11         self.db_file_name = db_file_name
12         self.employee_service = EmployeeService(db_file_name)
13
14     def _get_cursor(self):
15         connection = DBConnUtil.get_connection(self.db_file_name)
16         if not connection:
17             raise DatabaseConnectionException("Failed to get database connection for PayrollService.")
18         return connection, connection.cursor(dictionary=True)
19
20     def _close_cursor_connection(self, connection, cursor):
21         if cursor:
22             cursor.close()
23
24     def _map_row_to_payroll(self, row):
25         if row:
26
27             return Payroll([
28                 payroll_id=row.get("PayrollID"),
29                 employee_id=row.get("EmployeeID"),
30                 pay_period_start_date=row.get("PayPeriodStartDate"),
31                 pay_period_end_date=row.get("PayPeriodEndDate"),
32                 basic_salary=float(row.get("BasicSalary", 0.0)) if row.get("BasicSalary") is not None else 0.0,
33                 overtime_pay=float(row.get("OvertimePay", 0.0)) if row.get("OvertimePay") is not None else 0.0,
34                 deductions=float(row.get("Deductions", 0.0)) if row.get("Deductions") is not None else 0.0,
35                 net_salary=float(row.get("NetSalary", 0.0)) if row.get("NetSalary") is not None else 0.0
36             ])
37         return None
38
39     def generate_payroll(self, employee_id: int, start_date_obj: date, end_date_obj: date,
40                         basic_salary: float, overtime_pay: float = 0.0, deductions: float = 0.0):
41         if not all((isinstance(employee_id, int), employee_id > 0,
42                    isinstance(start_date_obj, date), isinstance(end_date_obj, date),
43                    isinstance(basic_salary, (int, float)) and basic_salary >= 0,
44                    isinstance(overtime_pay, (int, float)) and overtime_pay >= 0,
45                    isinstance(deductions, (int, float)) and deductions >= 0)):
46             raise InvalidInputException("Invalid input for generating payroll. Check types and values.")
47
48         if start_date_obj > end_date_obj:
49             raise InvalidInputException("Pay period start date cannot be after end date.")
50
51         connection, cursor = None, None
52         try:
53             self.employee_service.get_employee_by_id(employee_id)
54
55             net_salary = (basic_salary + overtime_pay) - deductions
56             if net_salary < 0:
57                 raise PayrollGenerationException("Net salary cannot be negative. Adjust salary components or deductions.")
58
59             connection, cursor = self._get_cursor()
60             sql = """
61             INSERT INTO Payroll
62             (EmployeeID, PayPeriodStartDate, PayPeriodEndDate, BasicSalary, OvertimePay, Deductions, NetSalary)
63             VALUES (%s, %s, %s, %s, %s, %s, %s)
64             """
65
66             params = (
67                 employee_id, start_date_obj, end_date_obj,
68                 basic_salary, overtime_pay, deductions, net_salary
69             )
70             cursor.execute(sql, params)
71             connection.commit()
72
73             last_id = cursor.lastrowid
74             if not last_id:
75                 cursor.execute("SELECT LAST_INSERT_ID()")
76                 last_id = cursor.fetchone()["LAST_INSERT_ID()"]
```

```

dao > 🐍 payroll_service.py > 📈 PayrollService > ⚡ _map_row_to_payroll
  9   class PayrollService(IPayrollsService):
10     def generate_payroll(self, employee_id: int, start_date_obj: date, end_date_obj: date,
11       if not last_id:
12         |   raise PayrollGenerationException("Failed to retrieve ID for newly generated payroll.")
13 
14       return self.get_payroll_by_id(last_id)
15 
16     except EmployeeNotFoundException:
17       raise PayrollGenerationException(f"Cannot generate payroll: Employee with ID {employee_id} not found.")
18     except mysql.connector.Error as err:
19       if connection: connection.rollback()
20       raise PayrollGenerationException(f"Database error generating payroll for employee ID {employee_id}: {err}")
21     except Exception as e:
22       if connection: connection.rollback()
23       if isinstance(e, PayrollGenerationException):
24         |   raise
25       raise PayrollGenerationException(f"Unexpected error generating payroll for employee ID {employee_id}: {e}")
26     finally:
27       self._close_cursor_connection(connection, cursor)
28 
29     def get_payroll_by_id(self, payroll_id):
30       if not isinstance(payroll_id, int) or payroll_id <= 0:
31         raise InvalidInputException("Payroll ID must be a positive integer.")
32     connection, cursor = None, None
33     try:
34       connection, cursor = self._get_cursor()
35       cursor.execute("SELECT * FROM Payroll WHERE PayrollID = %s", (payroll_id,))
36       row = cursor.fetchone()
37       if not row:
38         |   raise PayrollNotFoundException(f"Payroll record with ID {payroll_id} not found.")
39       return self._map_row_to_payroll(row)
40     except PayrollNotFoundException:
41       raise
42     except mysql.connector.Error as err:
43       raise DatabaseConnectionException(f"Database error fetching payroll by ID {payroll_id}: {err}")
44     except Exception as e:
45       raise DatabaseConnectionException(f"Unexpected error fetching payroll by ID {payroll_id}: {e}")
46     finally:
47       self._close_cursor_connection(connection, cursor)
48 
49     def get_payrolls_for_employee(self, employee_id):
50       if not isinstance(employee_id, int) or employee_id <= 0:
51         raise InvalidInputException("Employee ID must be a positive integer.")
52 
53     connection, cursor = None, None
54     try:
55       connection, cursor = self._get_cursor()
56       cursor.execute("SELECT * FROM Payroll WHERE EmployeeID = %s ORDER BY PayPeriodStartDate DESC", (employee_id,))
57       rows = cursor.fetchall()
58       return [self._map_row_to_payroll(row) for row in rows]
59     except mysql.connector.Error as err:
60       raise DatabaseConnectionException(f"Database error fetching payrolls for employee ID {employee_id}: {err}")
61     except Exception as e:
62       raise DatabaseConnectionException(f"Unexpected error fetching payrolls for employee ID {employee_id}: {e}")
63     finally:
64       self._close_cursor_connection(connection, cursor)
65 
66     def get_payrolls_for_period(self, start_date_obj: date, end_date_obj: date):
67       if not (isinstance(start_date_obj, date) and isinstance(end_date_obj, date)):
68         raise InvalidInputException("Start date and end date must be valid date objects.")
69       if start_date_obj > end_date_obj:
70         raise InvalidInputException("Period start date cannot be after end date.")
71 
72     connection, cursor = None, None
73     try:
74       connection, cursor = self._get_cursor()
75       sql = """
76       SELECT *
77       FROM Payroll
78       WHERE PayPeriodStartDate <= %s AND PayPeriodEndDate >= %
79       ORDER BY PayPeriodStartDate
80       """
81 
82       cursor.execute(sql, (end_date_obj, start_date_obj))
83       rows = cursor.fetchall()
84       return [self._map_row_to_payroll(row) for row in rows]
85     except mysql.connector.Error as err:
86       raise DatabaseConnectionException(f"Database error fetching payrolls for period: {err}")

```

Tax Module Logic

- Tax deductions based on defined salary brackets.
- Each tax record linked to a valid Employee ID.
- Prevents duplicate tax entries for the same employee and tax year.
- Integrates calculated tax amounts into the Net Salary deduction process.

```
dao > ⚡ tax_service.py > ...
1  from datetime import date
2  from dao.itax_service import ITaxService
3  from entity.tax import Tax
4  from util.db_conn_util import DBConnUtil
5  from exception.custom_exceptions import TaxCalculationException, EmployeeNotFoundException, DatabaseConnectionException, InvalidInputException, TaxRecordNotFoundException
6  from dao.employee_service import EmployeeService
7  import mysql.connector
8
9  class TaxService(ITaxService):
10     def __init__(self, db_file_name='db.properties'):
11         self.db_file_name = db_file_name
12         self.employee_service = EmployeeService(db_file_name)
13
14     def _get_cursor(self):
15         connection = DBConnUtil.get_connection(self.db_file_name)
16         if not connection:
17             raise DatabaseConnectionException("Failed to get database connection for TaxService.")
18         return connection, connection.cursor(dictionary=True)
19
20     def _close_cursor_connection(self, connection, cursor):
21         if cursor:
22             cursor.close()
23
24     def _map_row_to_tax(self, row):
25         if row:
26             return Tax(
27                 tax_id=row.get("TaxID"),
28                 employee_id=row.get("EmployeeID"),
29                 tax_year=int(row.get("TaxYear")) if row.get("TaxYear") is not None else None,
30                 taxable_income=float(row.get("TaxableIncome", 0.0)) if row.get("TaxableIncome") is not None else 0.0,
31                 tax_amount=float(row.get("TaxAmount", 0.0)) if row.get("TaxAmount") is not None else 0.0
32             )
33         return None
34
35     def calculate_tax(self, employee_id: int, tax_year: int, taxable_income: float, tax_rate: float = 0.15):
36         if not all([isinstance(employee_id, int), employee_id > 0,
37                    isinstance(tax_year, int), tax_year > 1980,
38                    isinstance(taxable_income, (int, float)) and taxable_income >= 0,
39                    isinstance(tax_rate, float) and 0 <= tax_rate <= 1]):
40             raise InvalidInputException("Invalid input for calculating tax. Check types and values.")
41
42         connection, cursor = None, None
43         try:
44
45             self.employee_service.get_employee_by_id(employee_id)
46
47             tax_amount = taxable_income * tax_rate
48             connection, cursor = self._get_cursor()
49
50             cursor.execute(
51                 "SELECT TaxID FROM Tax WHERE EmployeeID = %s AND TaxYear = %s",
52                 (employee_id, tax_year)
53             )
54             existing_tax_record_row = cursor.fetchone()
55             tax_id_to_return = None
56
57             if existing_tax_record_row:
58                 tax_id_to_return = existing_tax_record_row['TaxID']
59                 sql_update = """
59                  UPDATE Tax SET TaxableIncome = %s, TaxAmount = %s
60                  WHERE TaxID = %s
61                  """
62
63                 params_update = (taxable_income, tax_amount, tax_id_to_return)
64                 cursor.execute(sql_update, params_update)
65             else:
66                 sql_insert = """
67                   INSERT INTO Tax (EmployeeID, TaxYear, TaxableIncome, TaxAmount)
68                   VALUES (%s, %s, %s, %s)
69                   """
69
70                 params_insert = (employee_id, tax_year, taxable_income, tax_amount)
71                 cursor.execute(sql_insert, params_insert)
72                 tax_id_to_return = cursor.lastrowid
73
74                 if not tax_id_to_return:
75                     cursor.execute("SELECT LAST_INSERT_ID()")
76                     tax_id_to_return = cursor.fetchone()['LAST_INSERT_ID()']
```

```

dao > ⚡ tax_service.py > ...
9   class TaxService(ITaxService):
10     def calculate_tax(self, employee_id: int, tax_year: int, taxable_income: float, tax_rate: float = 0.15):
11       tax_id_to_return = cursor.lastrowid
12       if not tax_id_to_return:
13         cursor.execute("SELECT LAST_INSERT_ID()")
14         tax_id_to_return = cursor.fetchone()['LAST_INSERT_ID()']
15
16       connection.commit()
17       if not tax_id_to_return:
18         raise TaxCalculationException(f"Failed to get TaxID after upsert operation for EmpID {employee_id}, Year {tax_year}.")
19
20     return self.get_tax_by_id(tax_id_to_return)
21
22   except EmployeeNotFoundException:
23     raise TaxCalculationException(f"Cannot calculate tax: Employee with ID {employee_id} not found.")
24   except mysql.connector.Error as err:
25     if connection: connection.rollback()
26     raise TaxCalculationException(f"Database error calculating tax for EmpID {employee_id}, Year {tax_year}: {err}")
27   except Exception as e:
28     if connection: connection.rollback()
29     if isinstance(e, TaxCalculationException): raise
30     raise TaxCalculationException(f"Unexpected error calculating tax for EmpID {employee_id}, Year {tax_year}: {e}")
31   finally:
32     self._close_cursor_connection(connection, cursor)
33
34
35   def get_tax_by_id(self, tax_id):
36     if not isinstance(tax_id, int) or tax_id <= 0:
37       raise InvalidInputException("Tax ID must be a positive integer.")
38     connection, cursor = None, None
39     try:
40       connection, cursor = self._get_cursor()
41       cursor.execute("SELECT * FROM Tax WHERE TaxID = %s", (tax_id,))
42       row = cursor.fetchone()
43       if not row:
44         raise TaxRecordNotFoundException(f"Tax record with ID {tax_id} not found.")
45       return self._map_row_to_tax(row)
46     except TaxRecordNotFoundException:
47       raise
48     except mysql.connector.Error as err:
49       raise DatabaseConnectionException(f"Database error fetching tax by ID {tax_id}: {err}")
50     except Exception as e:
51       raise DatabaseConnectionException(f"Unexpected error fetching tax by ID {tax_id}: {e}")
52     finally:
53       self._close_cursor_connection(connection, cursor)
54
55
56   def get_taxes_for_employee(self, employee_id):
57     if not isinstance(employee_id, int) or employee_id <= 0:
58       raise InvalidInputException("Employee ID must be a positive integer.")
59     connection, cursor = None, None
60     try:
61       connection, cursor = self._get_cursor()
62       cursor.execute("SELECT * FROM Tax WHERE EmployeeID = %s ORDER BY TaxYear DESC", (employee_id,))
63       rows = cursor.fetchall()
64       return [self._map_row_to_tax(row) for row in rows]
65     except mysql.connector.Error as err:
66       raise DatabaseConnectionException(f"Database error fetching taxes for employee ID {employee_id}: {err}")
67     except Exception as e:
68       raise DatabaseConnectionException(f"Unexpected error fetching taxes for employee ID {employee_id}: {e}")
69     finally:
70       self._close_cursor_connection(connection, cursor)
71
72
73   def get_taxes_for_year(self, tax_year):
74     if not isinstance(tax_year, int) or tax_year <= 1900:
75       raise InvalidInputException("Tax year must be a valid year (e.g., greater than 1900).")
76     connection, cursor = None, None
77     try:
78       connection, cursor = self._get_cursor()
79       cursor.execute("SELECT * FROM Tax WHERE TaxYear = %s ORDER BY EmployeeID", (tax_year,))
80       rows = cursor.fetchall()
81       return [self._map_row_to_tax(row) for row in rows]
82     except mysql.connector.Error as err:
83       raise DatabaseConnectionException(f"Database error fetching taxes for year {tax_year}: {err}")
84     except Exception as e:
85       raise DatabaseConnectionException(f"Unexpected error fetching taxes for year {tax_year}: {e}")
86     finally:
87       self._close_cursor_connection(connection, cursor)
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
```

Financial Report Module Logic

- Aggregates total salary expenses, tax deductions, and net expenditures.
- Provides summarized reports for management decision-making.
- Ensures financial reports only include active and valid payroll records.

```
dao > ⌘ financial_record_service.py > ...
1  from datetime import date
2  from dao.ifinancial_record_service import IFinancialRecordService
3  from entity.financial_record import FinancialRecord
4  from util_db_conn_util import DBConnUtil
5  from exception.custom_exceptions import FinancialRecordException, EmployeeNotFoundException, DatabaseConnectionException, InvalidInputException, FinancialRecordNotFoundException
6  from dao.employee_service import EmployeeService
7  import mysql.connector
8
9  class FinancialRecordService(IFinancialRecordService):
10     def __init__(self, db_file_name="db.properties"):
11         self.db_file_name = db_file_name
12         self.employee_service = EmployeeService(db_file_name)
13
14     def _get_cursor(self):
15         connection = DBConnUtil.get_connection(self.db_file_name)
16         if not connection:
17             raise DatabaseConnectionException("Failed to get database connection for FinancialRecordService.")
18         return connection, connection.cursor(dictionary=True)
19
20     def _close_cursor_connection(self, connection, cursor):
21         if cursor:
22             cursor.close()
23
24     def _map_row_to_financial_record(self, row):
25         if row:
26             return FinancialRecord(
27                 record_id=row.get("RecordID"),
28                 employee_id=row.get("EmployeeID"),
29                 record_date=row.get("RecordDate"),
30                 description=row.get("Description"),
31                 amount=float(row.get("Amount", 0.0)) if row.get("Amount") is not None else 0.0,
32                 record_type=row.get("RecordType")
33             )
34         return None
35
36     def add_financial_record(self, employee_id: int, record_date_obj: date, description: str, amount: float, record_type: str):
37         if not all([isinstance(employee_id, int), employee_id > 0,
38                    isinstance(record_date_obj, date),
39                    isinstance(description, str) and description.strip(),
40                    isinstance(amount, (int, float)),
41                    isinstance(record_type, str) and record_type.strip()]):
42             raise InvalidInputException("Invalid input for adding financial record. Check types and values.")
43
44         valid_record_types = ["income", "expense", "tax payment", "bonus", "reimbursement", "deduction"]
45         if record_type.lower() not in valid_record_types:
46             raise InvalidInputException(f"Invalid record type: {record_type}. Must be one of {valid_record_types}.")
47
48         connection, cursor = None, None
49         try:
50             self.employee_service.get_employee_by_id(employee_id)
51
52             connection, cursor = self._get_cursor()
53             sql = """
54             INSERT INTO FinancialRecord (EmployeeID, RecordDate, Description, Amount, RecordType)
55             VALUES (%s, %s, %s, %s, %s)
56             """
57             params = (employee_id, record_date_obj, description, amount, record_type)
58             cursor.execute(sql, params)
59             connection.commit()
60
61             last_id = cursor.lastrowid
62             if not last_id:
63                 cursor.execute("SELECT LAST_INSERT_ID()")
64                 last_id = cursor.fetchone()[LAST_INSERT_ID()]
65
66             if not last_id:
67                 raise FinancialRecordException("Failed to retrieve ID for newly added financial record.")
68
69         return self.get_financial_record_by_id(last_id)
70
```

```

dao > 🐍 financial_record_service.py > ...
  9   class FinancialRecordService(IFinancialRecordService):
 10     def add_financial_record(self, employee_id: int, record_date_obj: date, description: str, amount: float, record_type: str):
 11       ...
 12       except EmployeeNotFoundException:
 13         raise FinancialRecordException(f"Cannot add financial record: Employee with ID {employee_id} not found.")
 14       except mysql.connector.Error as err:
 15         if connection: connection.rollback()
 16         raise FinancialRecordException(f"Database error adding financial record for EmpID {employee_id}: {err}")
 17       except Exception as e:
 18         if connection: connection.rollback()
 19         if isinstance(e, FinancialRecordException) or isinstance(e, InvalidInputException): raise
 20         raise FinancialRecordException(f"Unexpected error adding financial record for EmpID {employee_id}: {e}")
 21       finally:
 22         self._close_cursor_connection(connection, cursor)
 23
 24     def get_financial_record_by_id(self, record_id):
 25       if not isinstance(record_id, int) or record_id <= 0:
 26         raise InvalidInputException("Record ID must be a positive integer.")
 27       connection, cursor = None, None
 28       try:
 29         connection, cursor = self._get_cursor()
 30         cursor.execute("SELECT * FROM FinancialRecord WHERE RecordID = %s", (record_id,))
 31         row = cursor.fetchone()
 32         if not row:
 33           raise FinancialRecordNotFoundException(f"Financial record with ID {record_id} not found.")
 34         return self._map_row_to_financial_record(row)
 35       except FinancialRecordNotFoundException:
 36         raise
 37       except mysql.connector.Error as err:
 38         raise DatabaseConnectionException(f"Database error fetching financial record by ID {record_id}: {err}")
 39       except Exception as e:
 40         raise DatabaseConnectionException(f"Unexpected error fetching financial record by ID {record_id}: {e}")
 41       finally:
 42         self._close_cursor_connection(connection, cursor)
 43
 44     def get_financial_records_for_employee(self, employee_id):
 45       if not isinstance(employee_id, int) or employee_id <= 0:
 46         raise InvalidInputException("Employee ID must be a positive integer.")
 47       connection, cursor = None, None
 48       try:
 49         connection, cursor = self._get_cursor()
 50         cursor.execute("SELECT * FROM FinancialRecord WHERE EmployeeID = %s ORDER BY RecordDate DESC", (employee_id,))
 51         rows = cursor.fetchall()
 52         return [self._map_row_to_financial_record(row) for row in rows]
 53       except mysql.connector.Error as err:
 54         raise DatabaseConnectionException(f"Database error fetching financial records for employee ID {employee_id}: {err}")
 55       except Exception as e:
 56         raise DatabaseConnectionException(f"Unexpected error fetching financial records for employee ID {employee_id}: {e}")
 57       finally:
 58         self._close_cursor_connection(connection, cursor)
 59
 60     def get_financial_records_for_date(self, record_date_obj: date):
 61       if not isinstance(record_date_obj, date):
 62         raise InvalidInputException("Record date must be a valid date object.")
 63       connection, cursor = None, None
 64       try:
 65         connection, cursor = self._get_cursor()
 66         cursor.execute("SELECT * FROM FinancialRecord WHERE RecordDate = %s ORDER BY EmployeeID", (record_date_obj,))
 67         rows = cursor.fetchall()
 68         return [self._map_row_to_financial_record(row) for row in rows]
 69       except mysql.connector.Error as err:
 70         raise DatabaseConnectionException(f"Database error fetching financial records for date {record_date_obj}: {err}")
 71       except Exception as e:
 72         raise DatabaseConnectionException(f"Unexpected error fetching financial records for date {record_date_obj}: {e}")
 73       finally:
 74         self._close_cursor_connection(connection, cursor)
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
```

Data Integrity and Exception Handling

- Prevents insertion of invalid or incomplete data through input validation.
- Catches database connectivity failures and handles gracefully.
- Provides user-friendly error messages for invalid operations.
- Modular code structure ensures separation of logic, data handling, and presentation.

```
exception > custom_exceptions.py > ...
1  class EmployeeNotFoundException(Exception):
2      """Custom exception for non-existing employee."""
3      def __init__(self, message="Employee not found."):
4          self.message = message
5          super().__init__(self.message)
6
7  class PayrollNotFoundException(Exception):
8      """Custom exception for non-existing payroll record."""
9      def __init__(self, message="Payroll record not found."):
10         self.message = message
11         super().__init__(self.message)
12
13 class PayrollGenerationException(Exception):
14     """Custom exception for issues during payroll generation."""
15     def __init__(self, message="Error generating payroll."):
16         self.message = message
17         super().__init__(self.message)
18
19 class TaxRecordNotFoundException(Exception):
20     """Custom exception for non-existing tax record."""
21     def __init__(self, message="Tax record not found."):
22         self.message = message
23         super().__init__(self.message)
24
25 class TaxCalculationException(Exception):
26     """Custom exception for errors during tax calculation."""
27     def __init__(self, message="Error calculating tax."):
28         self.message = message
29         super().__init__(self.message)
30
31 class FinancialRecordNotFoundException(Exception):
32     """Custom exception for non-existing financial record."""
33     def __init__(self, message="Financial record not found."):
34         self.message = message
35         super().__init__(self.message)
36
37 class FinancialRecordException(Exception):
38     """Custom exception for issues with financial record management."""
39     def __init__(self, message="Error with financial record management."):
40         self.message = message
41         super().__init__(self.message)
42
43 class InvalidInputException(Exception):
44     """Custom exception for input data that doesn't meet criteria."""
45     def __init__(self, message="Invalid input provided."):
46         self.message = message
47         super().__init__(self.message)
48
49 class DatabaseConnectionException(Exception):
50     """Custom exception for problems establishing or maintaining database connection."""
51     def __init__(self, message="Database connection error."):
52         self.message = message
53         super().__init__(self.message)
```

Unit Testing

Unit testing ensures the correctness, reliability, and robustness of the PayXpert Payroll Management System. The following test scenarios outline essential test cases applied during development:

Test Case: CalculateGrossSalaryForEmployee

Objective: Verify the system correctly calculates an employee's gross salary.

Test Steps:

- Provide sample inputs for basic salary and overtime pay.
- Validate that the gross salary equals the sum of basic salary and overtime pay.

Expected Result: Gross Salary = Basic Salary + Overtime Pay

```
def test_get_payrolls_for_employee(self):
    print("\nRunning test_get_payrolls_for_employee for employee ID: " + str(self.test_employee_id))
    p1 = self.payroll_service.generate_payroll(self.test_employee_id, date(2024, 4, 1), date(2024, 4, 30), 5100, 100, 50)
    p2 = self.payroll_service.generate_payroll(self.test_employee_id, date(2024, 5, 1), date(2024, 5, 31), 5200, 150, 60)

    payrolls = self.payroll_service.get_payrolls_for_employee(self.test_employee_id)
    self.assertTrue(len(payrolls) == 2)

    payroll_ids = [p.get_payroll_id() for p in payrolls]
    self.assertEqual(p1.get_payroll_id(), payroll_ids[0])
    self.assertEqual(p2.get_payroll_id(), payroll_ids[1])
    print("test_get_payrolls_for_employee PASSED.")
```

Test Case: CalculateNetSalaryAfterDeductions

Objective: Ensure the system accurately calculates net salary after applying deductions (taxes, insurance, etc.).

Test Steps:

- Provide inputs for basic salary, overtime pay, and applicable deductions.
- Validate that net salary is correctly computed.

Expected Result:

Net Salary = Basic Salary + Overtime Pay - Deductions - Tax

```
def test_get_taxes_for_employee(self):
    print(f"\nRunning test_get_taxes_for_employee for EID: {self.test_employee_id}...")
    tax1 = self.tax_service.calculate_tax(self.test_employee_id, self.current_year -1, 60000)
    tax2 = self.tax_service.calculate_tax(self.test_employee_id, self.current_year, 65000)

    taxes = self.tax_service.get_taxes_for_employee(self.test_employee_id)
    self.assertTrue(len(taxes) >= 2)
    tax_ids_retrieved = [t.get_tax_id() for t in taxes]
    self.assertIn(tax1.get_tax_id(), tax_ids_retrieved)
    self.assertIn(tax2.get_tax_id(), tax_ids_retrieved)
    print("test_get_taxes_for_employee PASSED.")
```

Test Case: VerifyTaxCalculationForHighIncomeEmployee

Objective: Test tax calculation logic for high-income employees to ensure proper deduction according to tax brackets.

Test Steps:

- Simulate a high-income employee with salary exceeding the tax threshold.
- Validate the tax deduction is accurate as per defined tax rates.

Expected Result:

Tax calculated based on appropriate tax bracket rules.

```
def test_get_taxes_for_employee(self):
    print(f"\nRunning test_get_taxes_for_employee for EID: {self.test_employee_id}...")
    tax1 = self.tax_service.calculate_tax(self.test_employee_id, self.current_year -1, 60000)
    tax2 = self.tax_service.calculate_tax(self.test_employee_id, self.current_year, 65000)

    taxes = self.tax_service.get_taxes_for_employee(self.test_employee_id)
    self.assertTrue(len(taxes) >= 2)
    tax_ids_retrieved = [t.get_tax_id() for t in taxes]
    self.assertIn(tax1.get_tax_id(), tax_ids_retrieved)
    self.assertIn(tax2.get_tax_id(), tax_ids_retrieved)
    print("test_get_taxes_for_employee PASSED.")
```

Test Case: ProcessPayrollForMultipleEmployees

Objective: Test payroll processing for multiple employees to ensure system handles batch operations.

Test Steps:

- Provide payroll inputs for several employees.
- Run payroll processing module.
- Validate correct payroll records generated for each employee.

Expected Result:

Payroll processed accurately for all valid employees.

```
def test_get_payrolls_for_employee(self):
    print(f"\nRunning test_get_payrolls_for_employee for employee ID: {self.test_employee_id}...")
    p1 = self.payroll_service.generate_payroll(self.test_employee_id, date(2024,4,1), date(2024,4,30), 5100,100,50)
    p2 = self.payroll_service.generate_payroll(self.test_employee_id, date(2024,5,1), date(2024,5,31), 5200,150,60)

    payrolls = self.payroll_service.get_payrolls_for_employee(self.test_employee_id)
    self.assertTrue(len(payrolls) >= 2)

    payroll_ids = [p.get_payroll_id() for p in payrolls]
    self.assertIn(p1.get_payroll_id(), payroll_ids)
    self.assertIn(p2.get_payroll_id(), payroll_ids)
    print("test_get_payrolls_for_employee PASSED.")
```

Test Case: VerifyErrorHandlerForInvalidEmployeeData

Objective: Ensure system handles invalid employee data gracefully with proper error messages.

Test Steps:

- Enter incomplete or invalid employee details (e.g., missing name, invalid date).
 - Attempt to add employee record.
- Expected Result: System rejects invalid input with clear error messages, preventing data corruption.

```
def test_add_employee_invalid_input_type(self):
    print("\nRunning test_add_employee_invalid_input_type (Unit Test)...")
    with self.assertRaises(InvalidInputException):
        self.employee_service.add_employee("not an employee object")
    print("test_add_employee_invalid_input_type (Unit Test) PASSED.")
```

Output Screenshots

- Employee Add/Update/View with age calculation.

```
iamesaq@Esaqs-MacBook-Pro:~/Project-feature-initial-dev-and-testing-setup$ python3 -m main.main_module
PayXpert System Initialized Successfully.

===== PayXpert Main Menu =====
1. Employee Management
2. Payroll Management
3. Tax Management
4. Financial Record Management
5. Reporting
0. Exit
Enter your choice: 1

--- Employee Management ---
1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
0. Back to Main Menu
Enter choice: 1

--- Add New Employee ---
First Name: Esaq
Last Name: A
Date of Birth (YYYY-MM-DD): 2003-03-09
Gender (Male, Female, Other): Male
Email: esaq@gmail.com
Phone Number: 8667638590
Address: Chennai
Position: Developer
Joining Date (YYYY-MM-DD): 2025-07-01
Termination Date (optional, leave blank if N/A) (YYYY-MM-DD):

Employee added successfully with ID: 5
Employee ID: 5
Name: Esaq A
DOB: 2003-03-09
Gender: male
Email: esaq@gmail.com
Phone: 8667638590
Address: Chennai
Position: Developer
Joining Date: 2025-07-01
Termination Date: N/A
```

```
--- Employee Management ---
1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
0. Back to Main Menu
Enter choice: 2

--- Get Employee by ID ---
Enter Employee ID: 2
Employee ID: 2
Name: Bob Johnson
DOB: 1985-11-20
Gender: Male
Email: bob.johnson@example.com
Phone: 555-0102
Address: 456 Oak Street
Position: Project Manager
Joining Date: 2018-03-15
Termination Date: N/A
Age: 39
```

```
--- Employee Management ---
```

1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
6. Back to Main Menu

```
Enter choice: 3
```

```
--- All Employees ---
```

```
Employee ID: 1  
Name: Alice Smith  
DOB: 1990-05-15  
Gender: Female  
Email: alice.smith@example.com  
Phone: 555-0101  
Address: 123 Willow Lane  
Position: Software Engineer  
Joining Date: 2020-08-01  
Termination Date: N/A  
Age: 35
```

```
-----
```

```
Employee ID: 2  
Name: Bob Johnson  
DOB: 1985-11-20  
Gender: Male  
Email: bob.johnson@example.com  
Phone: 555-0102  
Address: 456 Oak Street  
Position: Project Manager  
Joining Date: 2018-03-15  
Termination Date: N/A  
Age: 39
```

```
-----
```

```
Employee ID: 3  
Name: Carol Williams  
DOB: 1995-02-10  
Gender: Female  
Email: carol.williams@example.com  
Phone: 555-0103  
Address: 21st Avenue  
Position: Data Analyst  
Joining Date: 2022-01-10  
Termination Date: N/A  
Age: 30
```

```
-----
```

```
Employee ID: 4  
Name: Esaq A  
DOB: 2003-03-09  
Gender: male  
Email: esaq@gmail.com  
Phone: 8667638598  
Address: Chennai  
Position: Developer  
Joining Date: 2025-07-01  
Termination Date: N/A  
Age: 22
```

```
--- Employee Management ---
```

1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
6. Back to Main Menu

```
Enter choice: 4
```

```
--- Update Employee ---
```

```
Enter Employee ID to update: 5  
Existing data:
```

```
Employee ID: 5  
Name: Esaq A  
DOB: 2003-03-09  
Gender: male  
Email: esaq@gmail.com  
Phone: 8667638598  
Address: Chennai  
Position: Developer  
Joining Date: 2025-07-01  
Termination Date: N/A  
Enter new data (leave blank to keep current value):  
First Name [Esaq]:  
Last Name [A]:  
Date of Birth [2003-03-09] (YYYY-MM-DD):  
Gender [male]:  
Email [esaq@gmail.com]: mynameisesaq@gmail.com  
Phone Number [8667638598]: 8667637598  
Address [Chennai]:  
Position [Developer]:  
Joining Date [2025-07-01] (YYYY-MM-DD):  
Termination Date [N/A] (YYYY-MM-DD, or empty to clear):
```

```
Employee updated successfully:
```

```
Employee ID: 5  
Name: Esaq A  
DOB: 2003-03-09  
Gender: male  
Email: mynameisesaq@gmail.com  
Phone: 8667637598  
Address: Chennai  
Position: Developer  
Joining Date: 2025-07-01  
Termination Date: N/A
```

```
--- Employee Management ---
1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
0. Back to Main Menu
Enter choice: 5

--- Remove Employee ---
Enter Employee ID to remove: 5
Are you sure you want to remove Esaq A (ID: 5)? (yes/no): yes
Employee ID 5 removed successfully.

--- Employee Management ---
1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
0. Back to Main Menu
Enter choice: 2

--- Get Employee by ID ---
Enter Employee ID: 5
Error: Employee with ID 5 not found.
```

- Payroll Add/View with total record count.

```
--- Payroll Management ---
1. Generate Payroll for Employee
2. Get Payroll by ID
3. Get Payrolls for Employee
4. Get Payrolls for Period
0. Back to Main Menu
Enter choice: 1

--- Generate Payroll ---
Enter Employee ID: 3
Pay Period Start Date (YYYY-MM-DD): 2025-07-01
Pay Period End Date (YYYY-MM-DD): 2025-07-31
Basic Salary: 25000
Overtime Pay (optional, default 0): 0
Deductions (optional, default 0): 0

Payroll generated successfully:
Payroll ID: 5
Employee ID: 3
Pay Period: 2025-07-01 to 2025-07-31
Basic Salary: 25000.0
Overtime Pay: 0.0
Deductions: 0.0
Net Salary: 25000.0
```

```
--- Payroll Management ---
1. Generate Payroll for Employee
2. Get Payroll by ID
3. Get Payrolls for Employee
4. Get Payrolls for Period
0. Back to Main Menu
Enter choice: 2

--- Get Payroll by ID ---
Enter Payroll ID: 2
Payroll ID: 2
Employee ID: 1
Pay Period: 2024-04-01 to 2024-04-30
Basic Salary: 6000.0
Overtime Pay: 250.0
Deductions: 700.0
Net Salary: 5550.0
```

```
--- Payroll Management ---
1. Generate Payroll for Employee
2. Get Payroll by ID
3. Get Payrolls for Employee
4. Get Payrolls for Period
0. Back to Main Menu
Enter choice: 3

--- Get Payrolls for Employee ---
Enter Employee ID: 3
Payroll ID: 5
Employee ID: 3
Pay Period: 2025-07-01 to 2025-07-31
Basic Salary: 25000.0
Overtime Pay: 0.0
Deductions: 0.0
Net Salary: 25000.0
```

```
--- Payroll Management ---
1. Generate Payroll for Employee
2. Get Payroll by ID
3. Get Payrolls for Employee
4. Get Payrolls for Period
0. Back to Main Menu
Enter choice: 4

--- Get Payrolls for Period ---
Search Period Start Date (YYYY-MM-DD): 2024-03-01
Search Period End Date (YYYY-MM-DD): 2024-03-31
Payroll ID: 4
Employee ID: 1
Pay Period: 2024-03-01 to 2024-03-31
Basic Salary: 10000.0
Overtime Pay: 0.0
Deductions: 0.0
Net Salary: 10000.0

Payroll ID: 1
Employee ID: 1
Pay Period: 2024-03-01 to 2024-03-31
Basic Salary: 6000.0
Overtime Pay: 700.0
Deductions: 700.0
Net Salary: 5800.0

Payroll ID: 3
Employee ID: 2
Pay Period: 2024-03-01 to 2024-03-31
Basic Salary: 7500.0
Overtime Pay: 0.0
Deductions: 1000.0
Net Salary: 6500.0
```

- Tax data entry and deduction display.

```
--- Tax Management ---
1. Calculate/Record Tax for Employee
2. Get Tax Record by ID
3. Get Taxes for Employee
4. Get Taxes for Year
0. Back to Main Menu
Enter choice: 1

--- Calculate/Record Tax ---
Enter Employee ID: 3
Tax Year (e.g., 2023): 2024
Taxable Income for the year: 250000
Tax Rate (e.g., 0.15 for 15%, default 0.15): 0.15

Tax calculated and recorded successfully:
Tax ID: 3
Employee ID: 3
Tax Year: 2024
Taxable Income: 250000.0
Tax Amount: 37500.0
```

```
--- Tax Management ---
1. Calculate/Record Tax for Employee
2. Get Tax Record by ID
3. Get Taxes for Employee
4. Get Taxes for Year
0. Back to Main Menu
Enter choice: 2

--- Get Tax Record by ID ---
Enter Tax ID: 1
Tax ID: 1
Employee ID: 1
Tax Year: 2023
Taxable Income: 72000.0
Tax Amount: 10800.0
```

```
--- Tax Management ---
1. Calculate/Record Tax for Employee
2. Get Tax Record by ID
3. Get Taxes for Employee
4. Get Taxes for Year
0. Back to Main Menu
Enter choice: 3
```

```
--- Get Taxes for Employee ---
Enter Employee ID: 1
Tax ID: 1
Employee ID: 1
Tax Year: 2023
Taxable Income: 72000.0
Tax Amount: 10800.0
-----
```

```
--- Tax Management ---
1. Calculate/Record Tax for Employee
2. Get Tax Record by ID
3. Get Taxes for Employee
4. Get Taxes for Year
0. Back to Main Menu
Enter choice: 4
```

```
--- Get Taxes for Year ---
Enter Tax Year: 2024
Tax ID: 3
Employee ID: 3
Tax Year: 2024
Taxable Income: 250000.0
Tax Amount: 37500.0
-----
```

- Financial Reports showing salary and tax summaries.

```
--- Financial Record Management ---
1. Add Financial Record
2. Get Financial Record by ID
3. Get Financial Records for Employee
4. Get Financial Records for Date
0. Back to Main Menu
Enter choice: 1
```

```
--- Add Financial Record ---
Enter Employee ID: 6
Record Date (YYYY-MM-DD): 2025-07-01
Description: Bonus
Amount (can be negative for debits if applicable by your logic): 250000
Record Type (e.g., income, expense, bonus, tax payment): bonus
```

```
Financial record added successfully:
Record ID: 5
Employee ID: 6
Record Date: 2025-07-01
Description: Bonus
Amount: 250000.0
Record Type: bonus
```

```
--- Financial Record Management ---
1. Add Financial Record
2. Get Financial Record by ID
3. Get Financial Records for Employee
4. Get Financial Records for Date
0. Back to Main Menu
Enter choice: 2
```

```
--- Get Financial Record by ID ---
Enter Record ID: 1
Record ID: 1
Employee ID: 1
Record Date: 2024-03-15
Description: Travel Expense Reimbursement
Amount: 150.0
Record Type: reimbursement
```

```

--- Financial Record Management ---
1. Add Financial Record
2. Get Financial Record by ID
3. Get Financial Records for Employee
4. Get Financial Records for Date
0. Back to Main Menu
Enter choice: 3

--- Get Financial Records for Employee ---
Enter Employee ID: 6
Record ID: 5
Employee ID: 6
Record Date: 2025-07-01
Description: Bonus
Amount: 250000.0
Record Type: bonus
-----
```

```

--- Financial Record Management ---
1. Add Financial Record
2. Get Financial Record by ID
3. Get Financial Records for Employee
4. Get Financial Records for Date
0. Back to Main Menu
Enter choice: 4

--- Get Financial Records for Date ---
Enter Record Date (YYYY-MM-DD): 2025-07-01
Record ID: 5
Employee ID: 6
Record Date: 2025-07-01
Description: Bonus
Amount: 250000.0
Record Type: bonus
-----
```

- Sample error handling (e.g., invalid input, connection issues).

```

===== PayXpert Main Menu =====
1. Employee Management
2. Payroll Management
3. Reporting
4. Financial Record Management
5. Reporting
0. Exit
Enter your choice: 1

--- Employee Management ---
1. Add New Employee
2. Get Employee by ID
3. Get All Employees
4. Update Employee
5. Remove Employee
0. Back to Main Menu
Enter choice: 2

--- Get Employee by ID ---
Enter Employee ID: 19
Error: Employee with ID 19 not found.
```

```

--- Payroll Management ---
1. Generate Payroll for Employee
2. Get Payroll by ID
3. Get Payrolls for Employee
4. Get Payrolls for Period
0. Back to Main Menu
Enter choice: 1

--- Generate Payroll ---
Enter Employee ID: 9
Pay Period Start Date (YYYY-MM-DD): 2008-09-09
Invalid date format. Please use YYYY-MM-DD.
Pay Period Start Date (YYYY-MM-DD): 2008-09-09
Pay Period End Date (YYYY-MM-DD): 2008-10-09
Basic Salary: abcd
Invalid input. Please enter a number.
Basic Salary: 20000
Overtime Pay (optional, default 0): a
Invalid input. Please enter a number.
Overtime Pay (optional, default 0): 0
Deductions (optional, default 0): 23
Error: Cannot generate payroll: Employee with ID 9 not found.
```

Challenges Faced

- Handling date formats during input and database storage.
- Ensuring encapsulation with getters and setters.
- Database column mismatches and corrections.
- Implementing specific attribute update using interactive menu.
- Validating optional fields like Termination Date.

- Integrating tax deductions into salary calculations.
- Generating consolidated financial reports.

Conclusion

The PayXpert system successfully demonstrates core concepts of OOP, database management, and exception handling. It offers scalable architecture that integrates Employee, Payroll, Tax, and Financial Reporting modules. The system emphasizes clean, modular code, efficient record management, automated financial calculations, and meaningful report generation.

Future Enhancements

- GUI development for better user experience.
- Enhanced validations and security.
- Multi-user access and concurrency control.
- Role-based authentication.