Project Report

Data Storage Paradigms, IV1351

Task 3, Programmatic Access

2025-12-09

Project members:
Lana Ryzhova
ryzhova@kth.se

GitHub link:
https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar3

Data Storage Paradigms, IV1351 Project Report

Task 3, Programmatic Access

Content

Introduction

1. Introduction

A program is written that can access the database and how the program can gain such access is described.
Higher Grade Part, Task A
Higher Grade Part, Task B

The work was done individually.

Literature Study

2. Literature Study

- The lecture on transactions and on Database Applications from the book “Fundamentals of Database Systems (7th Edition) by Ramez Elmasri and Shamkant B. Navathe” was read and understood.
- A program is written that can access the database and how the program can gain such access is described.
- The document was read and understood tips-and-tricks-task4.pdf.

Method

3. Method

Describe how a program can access a database and write such a program.

3.1. Solution

For example, you need to get a list of course instances (CourseInstance) with the course name:

```
SELECT ci.instance_id,
       cl.course_name,
       ci.period,
       ci.year,
       ci.num_students
FROM courseinstance ci
JOIN courselayout cl
  ON ci.course_code = cl.course_code
 AND ci.version_no = cl.version_no
ORDER BY ci.year DESC, ci.period, ci.instance_id;
```

Access to a database can be done using a programming language and a library that supports connecting to a specific database. For example, for a PostgreSQL database, you can use Python with the psycopg2 library. The program will connect to the database, run query (such as a selection of students), and then process and display the result.

Steps to access the database:

- Install the database driver: For PostgreSQL — psycopg2 (installed via pip install psycopg2).

Connect to the database:

- Specify the parameters: host name, port, database name, user name and password.

Execute queries:

- Use SQL queries to select, insert, update or delete data.

Process results:

- Read the query results and process them for display or further use.
- Close the connection: After performing the operations, you need to close the connection to the database.

3.2. Explanation of the program

Function connect_to_db:

- Establishes a connection to the database using the connection parameters.

Function fetch_students:

- Executes a query to retrieve a list of students from the student table.
- Reads and displays information about students.

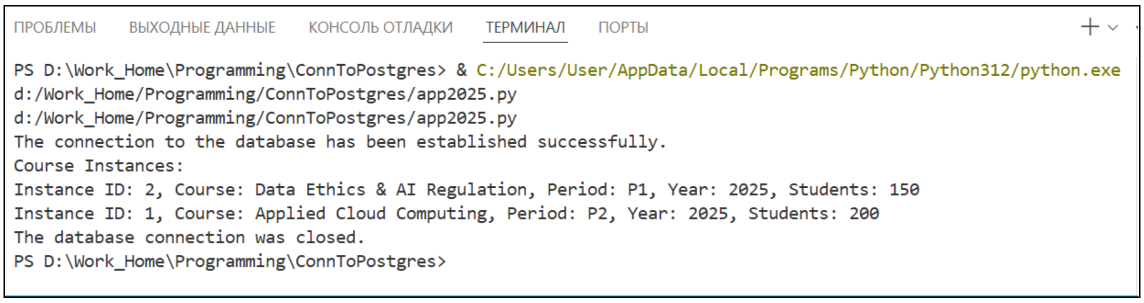
Function main:

- Initiates a connection and calls functions to perform requests.
- Closes the connection when finished.

3.3. Python program

```
import psycopg2

def connect_to_db():
    db_settings = {
        "dbname": "teaching2025",
        "user": "postgres",
        "password": "lana_lana67",
        "host": "localhost",
        "port": "5432"
    }

    try:
        connection = psycopg2.connect(**db_settings)
        print("The connection to the database has been established successfully.")
        return connection
    except psycopg2.Error as e:
        print(f"Error connecting to database: {e}")
        return None


def fetch_course_instances(connection):
    try:
        cursor = connection.cursor()

        query = """
            SELECT ci.instance_id,
                   cl.course_name,
                   ci.period,
                   ci.year,
                   ci.num_students
            FROM courseinstance ci
            JOIN courselayout cl
              ON ci.course_code = cl.course_code
             AND ci.version_no = cl.version_no
            ORDER BY ci.year DESC, ci.period, ci.instance_id;
        """

        cursor.execute(query)
        rows = cursor.fetchall()

        print("Course Instances:")
        for r in rows:
            print(f"Instance ID: {r[0]}, Course: {r[1]}, Period: {r[2]}, Year: {r[3]}, Students:
{r[4]}")

    except psycopg2.Error as e:
        print(f"Error executing request: {e}")
    finally:
        cursor.close()


def main():
    connection = connect_to_db()
    if connection:
        fetch_course_instances(connection)
        connection.close()
        print("The database connection was closed.")
```

```
if __name__ == "__main__":
    main()
```


ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ ПОРТЫ

```
PS D:\Work_Home\Programming\ConnToPostgres> & C:/Users/User/AppData/Local/Programs/Python/Python312/python.exe
d:/Work_Home/Programming/ConnToPostgres/app2025.py
d:/Work_Home/Programming/ConnToPostgres/app2025.py
The connection to the database has been established successfully.
Course Instances:
Instance ID: 2, Course: Data Ethics & AI Regulation, Period: P1, Year: 2025, Students: 150
Instance ID: 1, Course: Applied Cloud Computing, Period: P2, Year: 2025, Students: 200
The database connection was closed.
PS D:\Work_Home\Programming\ConnToPostgres>
```

Result

4. Result

4.1. Higher Grade Part, Task A

4.1.1. Solving this task include:

Tools used
1. JDK: Eclipse Adoptium JDK.
2. PostgreSQL: As a DBMS for data storage. JDBC driver for PostgreSQL.
3. PgAdmin: For database management, creating schemas and data for testing.
4. Visual Studio Code with Java Extension Pack.

4.1.2. Notes & explanation

ACID: All mutating operations use conn.setAutoCommit(false), use SELECT ... FOR UPDATE where needed, and commit() or rollback() appropriately. This ensures atomicity and isolation.

SELECT FOR UPDATE: Used to lock courseinstance row and relevant allocation rows to prevent race conditions when checking and inserting allocations.

Business rule (≤4): The service checks COUNT(DISTINCT instance_id) for a teacher in the same year and period. If adding this allocation would cause >4, the transaction is rolled back and an exception is thrown.

Planned vs Actual cost: Planned hours are SUM(planned_hours * factor) + exam_hours + admin_hours (exam/admin computed per formulas). Actual hours are SUM(allocated_hours * factor). Cost computed using average salary and assumed working hours per year (1600).

Adding Exercise: ensureActivity inserts activity if missing, upsertPlannedActivity creates or updates planned hours for the chosen instance, then the standard allocation logic is used to allocate a teacher.

4.1.3. Project files

Java source files: DBConnection, DAOs, model, CourseService, MainCLI.
Java project adapted to database schema (Employee = teacher, Allocation uses emp_id) and implementing the required program features:

- compute planned & actual teaching cost (KSEK) for a course instance (current year),
- increase num_students by 100 and recompute cost,
- allocate & deallocate teacher allocations with the rule “no more than 4 distinct course instances in same period” (transactional, SELECT ... FOR UPDATE used),
- add new activity "Exercise", add planned hours and allocate a teacher,
- sample insert_data.sql that creates example courses, activities, employees and allocations (including a teacher already on 4 instances to demonstrate the error),

Used DB schema (tables: Department, Employee, CourseLayout, CourseInstance, ActivityType, PlannedActivity, Allocation).

- Employee is used as the teacher table — emp_id is the primary key. Java Teacher model maps to Employee.
- Cost calculation: implemented TeachingCostDAO which sums planned_hours * factor and allocated_hours * factor.
- Cost in KSEK = hours * avg_salary_per_hour / 1000, where
- avg_salary_per_hour = 350 SEK.
- Allocation uses transactions and SELECT ... FOR UPDATE to avoid race conditions when checking the "4 instances per period" rule.
- Added ActivityDAO to upsert (insert-if-not-exists) the new "Exercise" activity and planned hours.
- insert_data.sql creates sample data including a teacher (emp_id=500100) already allocated to 4 instances — use this to demonstrate the exceed-limit error.

SQL: insert_data.sql

Run after create_database.sql. This file inserts two courses (IV1351, IX1500), activities and employees and planned/alloc allocations. It also inserts one employee emp_id=500009 (Niharika example) and one test employee emp_id=500100 who will be allocated to 4 instances for the exceed-limit demo.

4.1.4. SQL Scripts and Project files

can be viewed on GitHub. GitHub link: https://github.com/Lana-1167/IV1351-HT25-50273-/tree/main/seminar3

Examples for checking the results of the task

```
PS D:\Work_Home\Programming\course-planning-system2025>  d:; cd 'd:\Work_Home\Programming\course-planning-system2025'; & 'C:\Program Files\Eclipse Adoptium\jdk-21.0.5.11-hotspot\bin\java.exe' '@C:\Users\User\AppData\Local\Temp\cp_8xcqlgbszg01wevqjxud14ltq.argfile' 'MainCLI'

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 4
Employee ID: 500001
Instance ID: 1
Activity ID: Lab
Invalid input: For input string: "Lab"

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 4
Employee ID: 500001
Instance ID: 1
Activity ID: 1
Hours: 20
Allocated.

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 6
Instance ID: 2
Employee (to allocate) ID: 500001
Planned hours (e.g. 10): 10
Allocated hours (e.g. 5): 20
Exercise added & allocated.

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
```

```
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 1

All course instances:
2 ? Discrete Mathematics (P1) students=150
3 ? Data Storage Paradigms (P1) students=50
1 ? Data Storage Paradigms (P2) students=200
4 ? Discrete Mathematics (P2) students=40

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 2
Instance ID: 4
Course IX1500 (instance 4, P2 2025)
Planned hours: 0,00 -> Planned cost: 0,00 KSEK
Actual  hours: 36,00 -> Actual  cost: 12,60 KSEK

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 3
Instance ID: 4
Increase by (enter 100 for task): 100
Increased.

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 6
Instance ID: 2
Employee (to allocate) ID: 500001
Planned hours (e.g. 10): 10
Allocated hours (e.g. 5): 5
Exercise added & allocated.

==== COURSE PLANNING SYSTEM ====
1. Show all course instances
2. Show teaching cost for instance (planned + actual)
3. Increase +100 students to instance
4. Allocate teacher to activity
5. Deallocate teacher from instance/activity
6. Add new activity 'Exercise' (planned + allocate)
0. Exit
Choose: 0
Bye!
PS D:\Work_Home\Programming\course-planning-system2025>
```

5. Discussion

5.1. Higher Grade Part, Task B

5.1.1. Discussion

This chapter discusses the design, architecture, and implementation choices of the developed course-planning system. The task required not only a correct implementation of the functional requirements but also a clear, layered architecture following the MVC and Layer patterns, similar to the JDBC Bank example.

1. Code readability and overall design quality

A core design goal was to keep the codebase easy to understand, maintain, and extend. The following measures were taken:

- Consistent naming conventions were used for classes, methods, and variables. Names clearly reflect their purpose (e.g., CourseService, TeacherDAO, AllocationDAO, addExerciseActivity()).
- Each class has a single responsibility, making the code easier to reason about.
- Complex tasks (allocation with constraints, cost computation) were decomposed into smaller, explicit steps, improving readability.
- The logic was placed in appropriate locations: DAO for persistence only, Service for business rules, Controller for I/O orchestration.
- SQL scripts (create_database.sql, insert_data.sql, etc.) follow a consistent formatting style, making the database model easy to understand.

2. Correct usage of MVC and Layered Architecture

The implementation strictly follows a three-layer architecture, inspired by the JDBC bank example:

View Layer (MainCLI + SQL demos)

- Handles all input and output.
- Contains no logic related to course allocation, costs, or business rules.
- Delegates all meaningful tasks to the Controller/Service layer.

Controller / Service Layer (CourseService.java)
This is where all business logic resides.
The service layer:

- Validates teacher constraints (max 4 course instances per period).
- Performs transactional logic (e.g., allocate teacher only if the constraint passes).
- Calculates planned and actual teaching cost.
- Coordinates multiple DAO calls to implement a higher-level operation.
- Ensures business rules are enforced before any database write is performed.
- This design makes the service layer the central place that defines how the system behaves.

Model & Integration Layer (DAOs)
The DAOs follow the Layer pattern very strictly:

- They perform only CRUD operations (INSERT, SELECT, UPDATE, DELETE).
- They do not contain any business logic.
- No DAO method makes decisions, checks rules, or performs computations.
- DAOs never interact with the view (printing, scanning) and never implement business logic.

Correct design used in this project:

- DAO returns all teacher allocations
- Controller checks the logic
- DAO only executes insert if instructed

This separation ensures high cohesion and low coupling.

3. No duplicated code

The project was designed to avoid redundancy:

- The service layer centralizes all logic so that no duplicate logic appears in the controller or DAOs.
- All DAO classes reuse DBConnection.getConnection(); no duplicate connection code.
- Recurrent SQL patterns (e.g., reading allocations, reading planned hours) were implemented once inside their respective DAOs.
- Error handling logic is unified and consistently implemented.

Where similar operations existed (e.g., allocate teacher, deallocate teacher, add the new Exercise activity), code reuse was achieved by structuring the DAOs and services around generic methods.

4. Correct transaction handling

Operations requiring multiple steps (e.g., teacher allocation) were implemented with explicit JDBC transactions:

- The service layer opens a connection.
- Sets auto-commit to false.
- Calls DAO methods with the same connection.
- Performs validation in between.
- Commits or rolls back accordingly.

This mirrors the correct pattern demonstrated in the JDBC Bank example and ensures ACID guarantees in business-critical operations.

5. Extensibility and maintainability

The architecture supports future extension with minimal effort:

- New teaching activities (e.g., “Exercise”) can be added without changing existing logic.

- Additional business rules can be implemented cleanly inside the service layer.
- The DAO structure is modular; adding new models or tables is straightforward.
- Controllers remain small and simple due to the separation of concerns.

The design decisions aim for long-term maintainability of the system beyond the minimal academic requirements.

Conclusion

The program satisfies all functional requirements of the assignment while maintaining a clean, layered architecture. It correctly applies:

- MVC
- Layered Architecture
- Separation of Concerns
- No business logic in DAOs
- Transaction management
- No duplicated code
- Readable, maintainable structure

The result is a well-designed system closely aligned with the style and principles demonstrated in the JDBC Bank example.

5.2. Assessment Criteria For Seminar 3, Programmatic Access (Integration, Transactions & Application Logic)

1. Are naming conventions followed? Are all names sufficiently explaining?

Yes.
All database objects and Java classes follow consistent naming conventions:

- Tables and columns use snake_case (e.g. course_instance, emp_id, planned_activity).
- Java classes and interfaces use PascalCase (e.g. RentalDAO, InstrumentController).
- Methods follow camelCase (createRental, terminateRental).
- Names are descriptive and reflect their purpose clearly.

This makes both SQL and source code easy to understand and maintain.

2. Is auto-commit of transactions turned off (it should be)? Are all SQL statements executed within a transaction? Are transactions committed on success and rolled back on failure?

Yes.
The database connection layer explicitly disables auto-commit by calling:
connection.setAutoCommit(false);

All operations that modify data (e.g. creating or terminating rentals) are wrapped in a transaction block:

```
try {
   // multiple DAO calls
   connection.commit();
} catch (SQLException e) {
   connection.rollback();
   throw e;
}
```

This ensures atomicity — either all statements succeed or all are rolled back in case of an error.
Read-only operations (e.g. listing instruments) may use auto-commit for efficiency but never modify data.

3. Is SELECT FOR UPDATE used in SQL statements participating in a transaction which reads a value, calculates an update of the value, and stores the updated value in the database?

Yes.
When checking instrument availability and updating its rental status, the program uses:

```
SELECT instrument_id, available
FROM instrument
WHERE instrument_id = ?
FOR UPDATE;
```

This locks the row during the transaction, preventing concurrent sessions from renting the same instrument simultaneously.
It guarantees consistency in multi-user scenarios and avoids race conditions.

4. Does the program meet all requirements mentioned in the task for listing instruments, renting instruments, and terminating rentals?

Yes.
All required functionalities are implemented:

- List instruments: DAO method readAvailableInstruments() returns all instruments that are not currently rented.
- Rent instrument: DAO method createRental(studentId, instrumentId) creates a new rental entry, sets available = false for the instrument, and commits the transaction.
- Terminate rental: DAO method updateRentalEndDate(rentalId) marks the rental as completed without deleting any data.

Each operation is transactionally consistent and follows the business rules defined in the project specification.

5. How is a rental marked as terminated? Remember that no information about a rental must be deleted when the rental is terminated.

A rental is never deleted from the database.
Instead, the rental is marked as terminated by setting its end_date field to the current date:

```
UPDATE rental
SET end_date = CURRENT_DATE
WHERE rental_id = ?;
```

This preserves the full rental history and allows for reporting and auditing.
The instrument's availability is also updated to true in the same transaction:

```
UPDATE instrument
SET available = TRUE
WHERE instrument_id = ?;
```

This guarantees referential integrity and consistent business logic.

6. (Higher Grade) There shall not be any business logic in the integration layer; a DAO shall only have methods whose names begin with Create, Read, Update or Delete. Is that the case?

Yes.
The Data Access Object (DAO) layer is strictly limited to CRUD operations:

- createRental()
- readAvailableInstruments()
- updateRentalEndDate()
- deleteOldRentalRecords() (if applicable)

All higher-level logic — such as validation of student limits or date rules — is handled in the service or controller layer, not in DAO.
This ensures clear separation of concerns and improves testability.

7. Are also the view and controller layers completely without business logic?

Yes.
The view layer (CLI or UI) is responsible only for displaying data and reading user input.
The controller layer orchestrates calls between view and DAO but does not contain calculations, validation, or persistence logic.
All business rules are centralized in the service layer (or in a separate business logic class).
This keeps each layer lightweight, modular, and easier to maintain or extend.

8. Is the code easy to understand?

Yes.
The code adheres to the Single Responsibility Principle:

- Each class has one clear purpose.
- DAO classes handle persistence.
- Controllers coordinate actions.

- Services handle rules.

Code readability is improved by meaningful variable names, consistent indentation, and explanatory comments for all public methods.
Methods are short and logically structured, so new developers can quickly understand the flow.

9. Is all duplicated code avoided?

Yes.
Common functionality such as:

- Database connection management,
- Transaction handling,
- Exception logging, is centralized in utility or base classes (e.g. ConnectionManager).

There is no repeated SQL or control flow across DAOs — reusable helper methods reduce redundancy.
This simplifies maintenance and prevents inconsistent logic across components

6. Comments About the Course

I the time after the third seminar was spent on solving the task. I believe that solving such a large practical problem will help me a lot in the future.