

# **Project Report**

Data Storage Paradigms, IV1351

**Lana Ryzhova**

lana\_lana67@ukr.net

Ångermannagatan 1B lgh 1003

162 64 Vällingby

2024-12-11

## Data Storage Paradigms, IV1351 Project Report

### Task 4, Programmatic Access

1. Introduction	3
2. Literature study	3
3. Method	4
4. Result	9

**Code modified according to comments from Leif Lindbäck from 16 Dec at 15:19 Task A** **10**

**Remark from Leif Lindbäck from 16 Dec at 15:19 Task A** **15**

It's a waste of time to prepare the prepared statements every time they're executed. You could do that once when the program starts.

**Remark from Leif Lindbäck from 16 Dec at 15:19 Task A** **16**

Autocommit should be set to false once and for all when the program starts, not each time the database is called.

**Remark from Leif Lindbäck from 16 Dec at 15:19 Task A** **17**

FOR UPDATE shall be used by the SELECT statement reading information used when checking if a rental can be created (checkLimitQuery).

**Code modified according to comments from Leif Lindbäck from 16 Dec at 15:19 Task B** **19**

**Remark from Leif Lindbäck from 16 Dec at 15:19 Task B** **29**

It's a waste of time to prepare the prepared statements every time they're executed. You could do that once when the program starts.

**Remark from Leif Lindbäck from 16 Dec at 15:19 Task B** **30**

Autocommit should be set to false once and for all when the program starts, not each time the database is called.

**Remark from Leif Lindbäck from 16 Dec at 15:19 Task B** **30**

Task B is not accepted since it's not fully solved. It's not possible to list instruments or terminate rentals.

5. Discussion	33
6. Comments About the Course	34

## 1. Introduction

**The purpose of this report is:**

1.1 A program is written that can access the database and how the program can gain such access is described.

1.2 Higher Grade Part, Task A.

**The task was solved independently.**

## 2. Literature Study

1. 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.
2. A program is written that can access the database and how the program can gain such access is described.
3. The report and solution to this problem have been posted in my Git repository.

<https://github.com/Lana-1167/KTX-1167.git>

All necessary data is in the folder

<https://github.com/Lana-1167/KTX-1167 /seminar4/>

4. The document was read and understood tips-and-tricks-task4.pdf.

### 3. Method

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

**Solution:**

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:**

1. **Install the database driver:** For PostgreSQL — psycopg2 (installed via pip install psycopg2).
2. **Connect to the database:** Specify the parameters: host name, port, database name, user name and password.
3. **Execute queries:** Use SQL queries to select, insert, update or delete data.
4. **Process results:** Read the query results and process them for display or further use.
5. **Close the connection:** After performing the operations, you need to close the connection to the database.

**Python program:**

```
import psycopg2

def connect_to_db():
    # Database connection settings
    db_settings = {
        "dbname": "postgres",
        "user": "postgres",
        "password": "postgres",
        "host": "localhost",
        "port": "5432" # PostgreSQL Standard Port
```

```
}

try:
    # Establishing a connection to the database
    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_students(connection):
    try:
        # Create a cursor to execute queries
        cursor = connection.cursor()

        # We execute a SQL query
        query = "SELECT student_num, firstname, surname FROM student ORDER BY
student_num;"
        cursor.execute(query)

        # We get results
        students = cursor.fetchall()
        print("List of students:")
        for student in students:
            print(f"ID: {student[0]}, Name: {student[1]}, Surname:
{student[2]}")

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

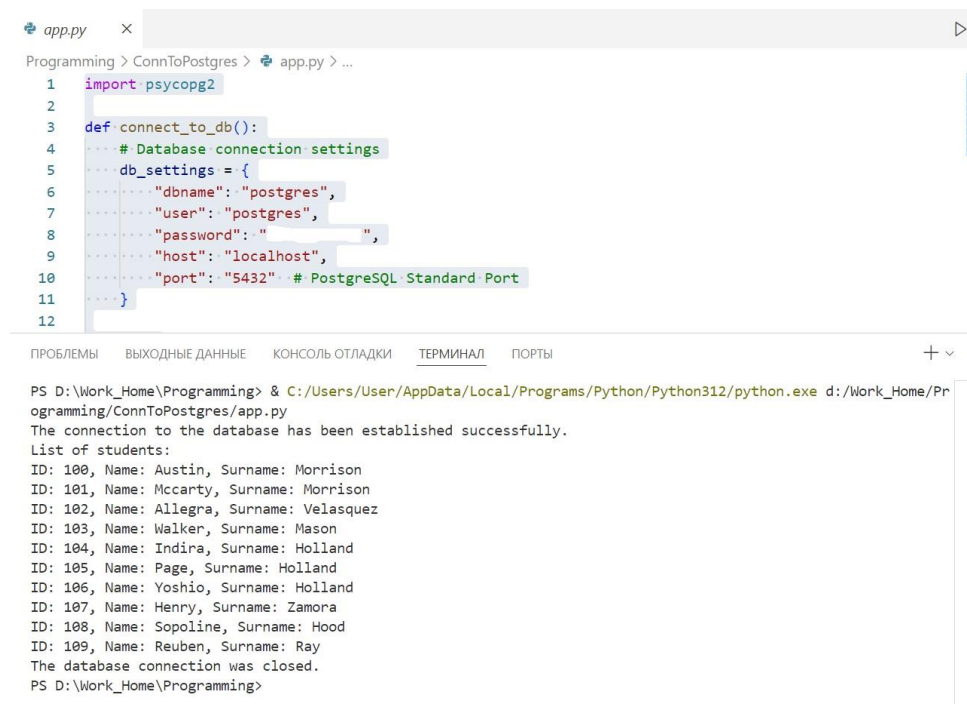
def main():
    # Connecting to the database
    connection = connect_to_db()
    if connection:
```

```
# We execute requests
fetch_students(connection)

# Closing the connection
connection.close()
print("The database connection was closed.")

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

**Result:** Python with the psycopg2 library for PostgreSQL database



The screenshot shows a code editor with a file named `app.py`. The code defines a `connect_to_db()` function that uses `psycopg2` to connect to a PostgreSQL database. The database settings are: dbname: 'postgres', user: 'postgres', password: 'postgres', host: 'localhost', and port: '5432'. Below the code, the terminal output shows the command to run the script, successful connection message, a list of 10 students, and the connection closing message.

```
1 import psycopg2
2
3 def connect_to_db():
4     # Database connection settings
5     db_settings = {
6         "dbname": "postgres",
7         "user": "postgres",
8         "password": "postgres",
9         "host": "localhost",
10        "port": "5432" # PostgreSQL Standard Port
11    }
12
```

PROБЛЕМЫ    ВЫХОДНЫЕ ДАННЫЕ    КОНСОЛЬ ОТЛАДКИ    ТЕРМИНАЛ    ПОРТЫ

PS D:\Work\_Home\Programming> & C:/Users/User/AppData/Local/Programs/Python/Python312/python.exe d:/Work\_Home/Pr  
ogramming/ConnToPostgres/app.py  
The connection to the database has been established successfully.  
List of students:  
ID: 100, Name: Austin, Surname: Morrison  
ID: 101, Name: Mccarty, Surname: Morrison  
ID: 102, Name: Allegra, Surname: Velasquez  
ID: 103, Name: Walker, Surname: Mason  
ID: 104, Name: Indira, Surname: Holland  
ID: 105, Name: Page, Surname: Holland  
ID: 106, Name: Yoshio, Surname: Holland  
ID: 107, Name: Henry, Surname: Zamora  
ID: 108, Name: Sopoline, Surname: Hood  
ID: 109, Name: Reuben, Surname: Ray  
The database connection was closed.  
PS D:\Work\_Home\Programming>

**Explanation of the program:****1. Function connect\_to\_db:**

Establishes a connection to the database using the connection parameters.

**2. Function fetch\_students:**

Executes a query to retrieve a list of students from the student table.

Reads and displays information about students.

**3. main:**

Initiates a connection and calls functions to perform requests.

Closes the connection when finished.

**Higher Grade Part, Task A**

**In the Method chapter of your report, mention which IDE(s) and other tool(s) you used and explain how you proceeded and reasoned when writing the program. Do not explain the result of each step you took, only explain the steps themselves.**

**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.

## **Considerations when writing a program**

### **1. Separation into functions:**

The program has been broken down into logical parts: list of musical instruments, rental of musical instruments, and end of rental.

This makes it easier to add new features and test existing ones.

### **2. Database design:**

Tables such as mus\_instruments, renting\_instruments, student, and others were analyzed to understand their relationships.

SQL queries were written to efficiently utilize the existing data structure.

### **3. Error handling:**

Each transaction is wrapped in a try-catch block to ensure that the transaction is rolled back if it fails.

### **4. Transaction security:**

Disabled autocommit mode (`conn.setAutoCommit(false)`) to manually manage transactions.

Use `commit()` to commit successful operations and `rollback()` to roll back on error.

### **5. Accounting for business logic:**

For example, the check for a limit of two instruments per student during rental was taken into account.

When the rental ends, the data is not deleted, but the return date is simply updated.

### **6. User interface:**

A simple command line interface was chosen for user interaction.



### **Brief explanation of the program**

The program is a console application for managing the rental of musical instruments. It allows:

List the available instruments of a certain type for rent.

Complete the student's instrument rental, checking that the two-instrument limit has not been exceeded.

End the instrument rental by updating the database entry.

## **4. Result**

### **Higher Grade Part, Task A**

In the Result chapter of your report, briefly explain the program and in particular explain ACID transaction handling. Include links to your git repository, and make sure the repository is public. Also include a printout of a sample run. The git repository must also contain the scripts that create the database and insert data. It shall be possible to test your solution by executing first the script that creates the database, then the script that inserts data, and finally execute your program.

The assignment is to develop part of Soundgood's web site. You're however only required to develop a very limited set of functionalities, namely what's used when instruments are rented. Also, since focus here is on database access, you're not required to develop the web interface, but a command line user interface is sufficient.

The program is required to handle ACID transactions properly, which means autocommit must be turned off, instead the program must call commit and rollback as required. Handling transactions properly also means that `SELECT FOR UPDATE` must be used when required. Finally, you have to make sure the database contains sufficient data to check that all queries work as intended. If needed, update the script that inserts data, created in task two. You're allowed to change the database you created in task two if needed.

**Code modified according to comments from Leif Lindbäck from 16 Dec at 15:19:**

```
import java.sql.*;
import java.util.Scanner;

public class RentalSystem {

    private static final String DB_URL =
"jdbc:postgresql://localhost:5432/postgres";
    private static final String DB_USER = "postgres";
    private static final String DB_PASSWORD = "lana_lana67";

    private Connection conn;
    private PreparedStatement listAvailableStmt;
    private PreparedStatement checkLimitStmt;
    private PreparedStatement rentStmt;
    private PreparedStatement endRentalStmt;

    public RentalSystem() {
        try {
            // Establishing a connection to the database
            conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
            conn.setAutoCommit(false); // Set AutoCommit to false once
            prepareStatements();
        } catch (SQLException e) {
            e.printStackTrace();
            closeConnection();
        }
    }

    private void prepareStatements() throws SQLException {
        // Prepare queries once at program startup
        String listAvailableQuery = "SELECT i.instrument_id, i.brand_id, c.cost,
p.price_service " +
            "FROM mus_instruments i " +
            "JOIN conditions c ON i.instrument_id = c.instrument_id " +
            "JOIN price p ON p.price_id = 1 " +
            "WHERE p.price_start <= CURRENT_DATE " +
```

```

        "AND p.price_finish >= CURRENT_DATE " +
        "AND i.type_id = ? " +
        "AND NOT EXISTS (" +
        "    SELECT 1 " +
        "    FROM renting_instruments r " +
        "    WHERE r.instrum_id = i.instrument_id " +
        "    AND r.period_end >= CURRENT_DATE " +
        "    AND r.period_start <= CURRENT_DATE" +
        ")";
listAvailableStmt = conn.prepareStatement(listAvailableQuery);

String checkLimitQuery = "SELECT renting_id FROM renting_instruments " +
    "WHERE student_num = ? AND period_end >= CURRENT_DATE FOR
UPDATE";
checkLimitStmt = conn.prepareStatement(checkLimitQuery);

String rentQuery = "INSERT INTO renting_instruments (student_num,
instrum_id, period_start, period_end, quantity_instruments, price_id) "
    +
    "VALUES (?, ?, ?, ?, 1, (SELECT price_id FROM price WHERE
price_id = 1 AND price_start <= CURRENT_DATE AND price_finish >=
CURRENT_DATE))";
rentStmt = conn.prepareStatement(rentQuery);

String endRentalQuery = "UPDATE renting_instruments SET period_end =
CURRENT_DATE WHERE renting_id = ?";
endRentalStmt = conn.prepareStatement(endRentalQuery);
}

public void listAvailableInstruments(String instrumentType) {
    try {
        listAvailableStmt.setInt(1, Integer.parseInt(instrumentType));
        ResultSet rs = listAvailableStmt.executeQuery();

        while (rs.next()) {
            int instrumentId = rs.getInt("instrument_id");
            String brand = rs.getString("brand_id");
            double price = rs.getDouble("price_service");

```

```
        System.out.println("Instrument ID: " + instrumentId + ", Brand: " + brand + ", Price: " + price);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
}

public void rentInstrument(int studentId, int instrumentId, String periodStart, String periodEnd) {
    try {
        // Checking the limit using FOR UPDATE
        checkLimitStmt.setInt(1, studentId);
        ResultSet rs = checkLimitStmt.executeQuery();

        int rentedCount = 0;
        while (rs.next()) {
            rentedCount++;
        }

        if (rentedCount >= 2) {
            System.out.println("Student has already rented the maximum allowed number of instruments. To rent music instruments again, you need to terminate the current rent. Run the program again and select the required action.");
            return;
        }

        // Performing a rental operation
        rentStmt.setInt(1, studentId);
        rentStmt.setInt(2, instrumentId);
        rentStmt.setDate(3, Date.valueOf(periodStart));
        rentStmt.setDate(4, Date.valueOf(periodEnd));
        try {
            rentStmt.executeUpdate();
            conn.commit(); // Committing a transaction
            System.out.println("Instrument rented successfully.");
        } catch (SQLException e) {
            if (e.getSQLState().equals("23505")) {
```

```
// Check for uniqueness violation
        System.out.println(
            "Error: Renting ID already exists. Please enter a
new renting ID or end the rental.");
    } else {
        throw e; // Skip the remaining errors
    }
}

} catch (SQLException e) {
    try {
        conn.rollback(); // Rollback in case of error
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    e.printStackTrace();
}

}

public void endRental(int rentingId) {
    try {
        endRentalStmt.setInt(1, rentingId);
        endRentalStmt.executeUpdate();
        conn.commit(); // Committing a transaction
        System.out.println("Rental ended successfully.");
    } catch (SQLException e) {
        try {
            conn.rollback(); // Rollback transaction
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
        e.printStackTrace();
    }
}

private void closeConnection() {
    try {
        if (conn != null && !conn.isClosed())
            conn.close();
    }
}
```

```
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    RentalSystem system = new RentalSystem();
    Scanner scanner = new Scanner(System.in);

    try {
        // Offering the user a choice of action
        System.out.println("What would you like to do?");
        System.out.println("1. Rent an instrument");
        System.out.println("2. End a rental");
        int choice = scanner.nextInt();

        if (choice == 1) {
            // The rental process
            System.out.println("Enter instrument type to list available
instruments:");
            scanner.nextLine();
            // Absorb the remaining newline character
            String instrumentType = scanner.nextLine();
            system.listAvailableInstruments(instrumentType);

            System.out.println("Enter student ID to rent an instrument:");
            int studentId = scanner.nextInt();
            System.out.println("Enter instrument ID to rent:");
            int instrumentId = scanner.nextInt();
            System.out.println("Enter rental start date (yyyy-mm-dd):");
            String startDate = scanner.next();
            System.out.println("Enter rental end date (yyyy-mm-dd):");
            String endDate = scanner.next();
            system.rentInstrument(studentId, instrumentId, startDate,
endDate);

        } else if (choice == 2) {
            // Lease Termination Process
            System.out.println("Enter rental ID to end rental:");
```

```
        int rentingId = scanner.nextInt();
        system.endRental(rentingId);

    } else {
        System.out.println("Invalid choice. Please select 1 or 2.");
    }

} finally {
    scanner.close(); // Closing the Scanner resource
    system.closeConnection(); // Closing the database connection
}
}
```

**Remark from Leif Lindbäck from 16 Dec at 15:19**

**1. It's a waste of time to prepare the prepared statements every time they're executed. You could do that once when the program starts.**

In the code, the PreparedStatement is prepared once at program startup in the prepareStatements() method. This method is called from the RentalSystem class constructor, and all prepared statements are stored in the class fields (listAvailableStmt, checkLimitStmt, rentStmt, endRentalStmt).

**Code:**

```
private void prepareStatements() throws SQLException {
    // Create a prepared query
    String listAvailableQuery = "SELECT i.instrument_id, i.brand_id, c.cost, p.price_service " +
        "FROM mus_instruments i " +
        "JOIN conditions c ON i.instrument_id = c.instrument_id " +
        "JOIN price p ON p.price_id = 1 " +
        "WHERE p.price_start <= CURRENT_DATE " +
        "AND p.price_finish >= CURRENT_DATE " +
        "AND i.type_id = ? " +
```

```
"AND NOT EXISTS (" +  
"  SELECT 1 " +  
"  FROM renting_instruments r " +  
"  WHERE r.instrum_id = i.instrument_id " +  
"  AND r.period_end >= CURRENT_DATE " +  
"  AND r.period_start <= CURRENT_DATE" +  
");  
listAvailableStmt = conn.prepareStatement(listAvailableQuery);  
...  
}
```

**Remark from Leif Lindbäck from 16 Dec at 15:19**

**2. Autocommit should be set to false once and for all when the program starts, not each time the database is called.**

**AutoCommit is set to false once when the database connection is created in the RentalSystem class constructor.**

**Code:**

```
public RentalSystem() {  
    try {  
        // Establishing a connection to the database  
        conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);  
        conn.setAutoCommit(false); // Set AutoCommit to false once  
        prepareStatements();  
    } catch (SQLException e) {  
        e.printStackTrace();  
        closeConnection();  
    }  
}
```



Remark from Leif Lindbäck from 16 Dec at 15:19

3. FOR UPDATE shall be used by the SELECT statement reading information used when checking if a rental can be created (checkLimitQuery).

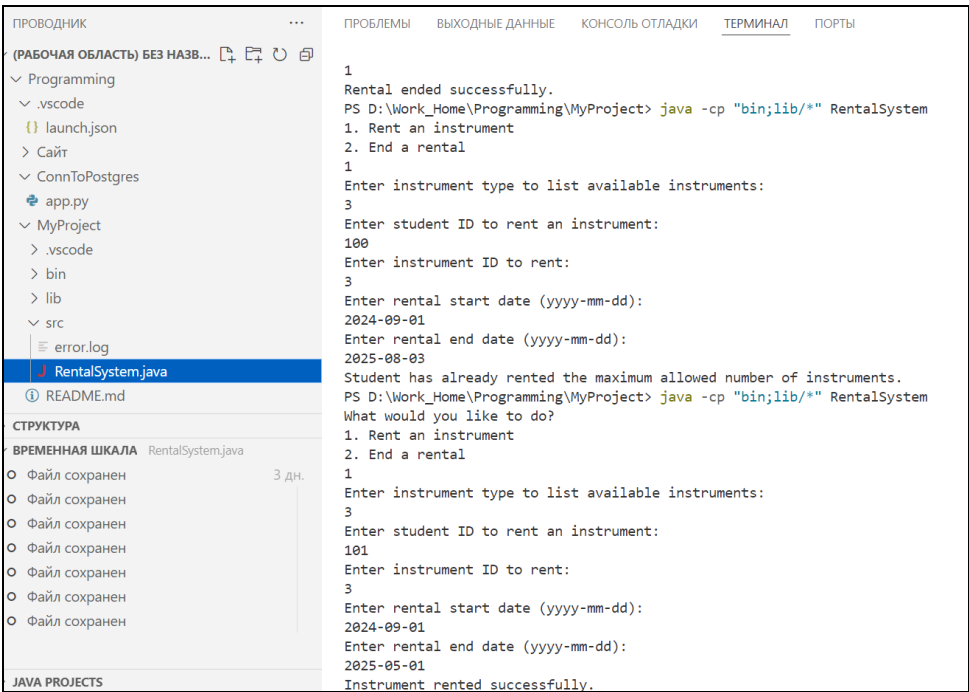
The checkLimitQuery used to check whether a student can rent an instrument includes the FOR UPDATE keyword to lock the rows for modification by other transactions until the current transaction completes.

String checkLimitQuery = "SELECT renting\_id FROM renting\_instruments " +

"WHERE student\_num = ? AND period\_end >= CURRENT\_DATE FOR UPDATE";

checkLimitStmt = conn.prepareStatement(checkLimitQuery);

Result:



1

2

select\* from public.renting\_instruments;

Data Output

Сообщения

Notifications

SQL

	renting_id [PK] Integer	student_num Integer	instrum_id Integer	period_start date	period_end date	quantity_instruments Integer	price_id Integer
1	3	100	2	2024-09-01	2025-08-20	2	1
2	1	100	1	2024-09-01	2024-12-24	1	1
3	4	101	3	2024-09-01	2025-05-01	1	1

**Higher Grade Part, Task B (gives 10p, since solving this task means also task A is solved)**

This task is identical to task A, but the program being developed must also be well designed and have a properly layered architecture. The required level of design and architecture is that of the JDBC bank example at the page Database Applications. The following must be showed in the discussion chapter of the report.

The code must be easy to understand. This is of course subjective, what is required is to show that you have tried sufficiently to make the code easy to understand.

The MVC and Layer patterns must be used correctly. There must be enough layers, packages and classes. Neither controller nor model are allowed contain any code related to the view (input or output). Also, the integration layer (the DAO) is only allowed to contain methods that create, read, update or delete rows in the database. There must not be any logic at all in the integration layer. As an example, this means you're not allowed to have a method in a DAO that checks if an instrument is available to rent, then checks if the student is allowed to rent, and finally creates a rental. Instead, to handle this scenario, the controller must first call a method in a DAO that reads rentals, then the controller checks if the instrument is available to rent, then calls another DAO method that reads student data, then checks if the student is allowed to rent, and then finally calls a DAO method that creates a rental.

There must not be any duplicated code.

To meet the requirements for **Task B**, we need to redesign the program to follow a **layered architecture** with proper **separation of concerns**. This will involve organizing the code into three layers using the **Model-View-Controller (MVC)** pattern and adhering to the **DAO (Data Access Object)** pattern for database operations.

## 1. Architecture Overview

- **Model Layer:** Represents the business logic and data of the application.

Entities (e.g., Instrument).

DAO classes for database operations (CRUD methods only).

- **View Layer:** Handles input/output with the user.

No direct interaction with the database or business logic.

- **Controller Layer:** Acts as a mediator between the View and Model layers.

Handles user requests, interacts with DAOs, and performs any necessary business logic.

## 2. Layered Code Design

### Packages

- **model:** Contains entity classes and business logic.
- **integration:** Contains DAOs for database access.
- **controller:** Contains the main logic and coordination.
- **view:** Handles user interaction.

Code modified according to comments from Leif Lindbäck from 16 Dec at 15:19:

### Entity Class (Model Layer)

```
package model;

public class Instrument {
    private int id;
    private String brand;
    private double price;

    public Instrument(int id, String brand, double price) {
        this.id = id;
        this.brand = brand;
        this.price = price;
    }

    public int getId() {
        return id;
    }
}
```

```
public String getBrand() {  
    return brand;  
}  
  
public double getPrice() {  
    return price;  
}  
  
@Override  
public String toString() {  
    return "Instrument ID: " + id + ", Brand: " + brand + ", Price: " +  
price;  
}  
}
```

### DAO Class (Integration Layer)

```
package integration;  
  
import java.sql.*;  
import java.util.ArrayList;  
import java.util.List;  
  
public class InstrumentDAO {  
    private static final String DB_URL =  
"jdbc:postgresql://localhost:5432/postgres";  
    private static final String DB_USER = "postgres";  
    private static final String DB_PASSWORD = "postgres";  
  
    private Connection conn;  
    private PreparedStatement listAvailableStmt;  
    private PreparedStatement getActiveRentalCountStmt;  
    private PreparedStatement createRentalStmt;  
    private PreparedStatement endRentalStmt;
```

```

public InstrumentDAO() {
    try {
        conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
        conn.setAutoCommit(false); // Disable auto-commit once
        prepareStatements();
    } catch (SQLException e) {
        throw new RuntimeException("Error initializing database connection:
" + e.getMessage());
    }
}

private void prepareStatements() throws SQLException {
    String listAvailableQuery = "SELECT mi.instrument_id, ti.types_instrum,
mi.brand_id, c.cost " +
        "FROM mus_instruments mi " +
        "JOIN conditions c ON mi.instrument_id = c.instrument_id " +
        "JOIN types_instruments ti ON mi.type_id = ti.type_id " +
        "WHERE mi.type_id = ? " +
        "AND NOT EXISTS ( " +
        "    SELECT 1 FROM renting_instruments ri " +
        "    WHERE ri.instrum_id = mi.instrument_id AND ri.period_end >=
CURRENT_DATE)";
    listAvailableStmt = conn.prepareStatement(listAvailableQuery);

    String getActiveRentalCountQuery = "SELECT COUNT(*) AS rental_count FROM
renting_instruments " +
        "WHERE student_num = ? AND period_end >= CURRENT_DATE";
    getActiveRentalCountStmt =
conn.prepareStatement(getActiveRentalCountQuery);

    String createRentalQuery = "INSERT INTO renting_instruments " +
        "(student_num, instrum_id, period_start, period_end,
quantity_instruments, price_id) " +
        "VALUES (?, ?, ?, ?, ?, ?)";
    createRentalStmt = conn.prepareStatement(createRentalQuery);

    String endRentalQuery = "UPDATE renting_instruments SET period_end =
CURRENT_DATE WHERE renting_id = ?";

```

```
        endRentalStmt = conn.prepareStatement(endRentalQuery);
    }

    public void listAvailableInstruments(int instrumentType) {
        try {
            listAvailableStmt.setInt(1, instrumentType);
            ResultSet rs = listAvailableStmt.executeQuery();

            System.out.println("Available instruments:");
            boolean found = false;
            while (rs.next()) {
                found = true;
                int instrumentId = rs.getInt("instrument_id");
                String type = rs.getString("types_instrum");
                String brand = rs.getString("brand_id");
                double price = rs.getDouble("cost");
                System.out.println("Instrument ID: " + instrumentId + ", Type: "
+ type + ", Brand: " + brand
                                + ", Price: " + price);
            }
            if (!found) {
                System.out.println("No available instruments found for this
type.");
            }
        } catch (SQLException e) {
            throw new RuntimeException("Error listing available instruments: " +
e.getMessage());
        }
    }

    public void endRental(int rentingId) {
        try {
            // Ensure the rental exists before attempting to end it
            String checkQuery = "SELECT renting_id, period_end FROM
renting_instruments WHERE renting_id = ?";
            try (PreparedStatement checkStmt =
conn.prepareStatement(checkQuery)) {
                checkStmt.setInt(1, rentingId);
                ResultSet rs = checkStmt.executeQuery();
            }
        }
    }
}
```

```

        if (!rs.next()) {
            throw new RuntimeException("Rental with ID " + rentingId + "
not found.");
        }
    }

    // Proceed with the rental end update
    endRentalStmt.setInt(1, rentingId);
    int rowsAffected = endRentalStmt.executeUpdate();
    if (rowsAffected > 0) {
        System.out.println("Rental with ID " + rentingId + " has been
successfully ended.");
    } else {
        System.out.println("No rental found with ID " + rentingId +
".");
    }

    // Commit the transaction
    conn.commit();
} catch (SQLException e) {
    try {
        conn.rollback();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    throw new RuntimeException("Error ending rental: " +
e.getMessage());
}

public int getActiveRentalCountForStudent(int studentId) {
    try {

// Ensure the rentals considered are still active (i.e., not ended)
        getActiveRentalCountStmt.setInt(1, studentId);
        ResultSet rs = getActiveRentalCountStmt.executeQuery();
        if (rs.next()) {
            return rs.getInt("rental_count");
        }
    }
}

```

```
        } catch (SQLException e) {
            throw new RuntimeException("Error retrieving active rental count: "
+ e.getMessage());
        }
        return 0;
    }

    public void createRental(int studentId, int instrumentId, Date startDate,
Date endDate, int quantity) {
        try {
            // Get the active rental count for the student (exclude rentals that
            // have already ended)
            int activeRentals = getActiveRentalCountForStudent(studentId);
            if (activeRentals >= 2) {
                throw new IllegalArgumentException(
                    "Student has already rented the maximum allowed number
of instruments (" + activeRentals
                        + ").");
            }

            // Proceed to create the rental
            createRentalStmt.setInt(1, studentId);
            createRentalStmt.setInt(2, instrumentId);
            createRentalStmt.setDate(3, startDate);
            createRentalStmt.setDate(4, endDate);
            createRentalStmt.setInt(5, quantity);
            createRentalStmt.setInt(6, getValidPriceId(instrumentId));
            createRentalStmt.executeUpdate();
            conn.commit();
            System.out.println("Rental created successfully.");
        } catch (SQLException e) {
            try {
                conn.rollback();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
            throw new RuntimeException("Error creating rental: " +
e.getMessage());
        }
    }
}
```



```
    }

    private int getValidPriceId(int instrumentId) {
        String query = "SELECT price_id FROM price WHERE type_service = 'Rent  
instruments' " +
            "AND price_start <= CURRENT_DATE AND price_finish >=  
CURRENT_DATE";
        try (PreparedStatement stmt = conn.prepareStatement(query)) {
            ResultSet rs = stmt.executeQuery();
            if (rs.next()) {
                return rs.getInt("price_id");
            }
        } catch (SQLException e) {
            throw new RuntimeException("Error retrieving price ID: " +
e.getMessage());
        }
        throw new RuntimeException("No valid price ID found.");
    }

    public List<Integer> getCurrentRentalsForInstrument(int instrumentId) {
        List<Integer> rentals = new ArrayList<>();
        String query = "SELECT renting_id FROM renting_instruments WHERE  
instrum_id = ? AND period_end >= CURRENT_DATE";
        try (PreparedStatement stmt = conn.prepareStatement(query)) {
            stmt.setInt(1, instrumentId);
            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
                rentals.add(rs.getInt("renting_id"));
            }
        } catch (SQLException e) {
            throw new RuntimeException("Error retrieving current rentals for  
instrument: " + e.getMessage());
        }
        return rentals;
    }
}
```

## Controller Class

```
package controller;

import integration.InstrumentDAO;
import java.sql.Date;

public class RentalController {
    private final InstrumentDAO instrumentDAO;

    public RentalController(InstrumentDAO instrumentDAO) {
        this.instrumentDAO = instrumentDAO;
    }

    public void rentInstrument(int studentId, int instrumentId, String
startDateStr, String endDateStr, int quantity) {
        try {
            Date startDate = Date.valueOf(startDateStr);
            Date endDate = Date.valueOf(endDateStr);

            // Checking limits for an instrument
            if
(instrumentDAO.getCurrentRentalsForInstrument(instrumentId).size() >= 2) {
                throw new IllegalArgumentException("This instrument has reached
its rental limit.");
            }

            // Checking student limits
            int activeRentals =
instrumentDAO.getActiveRentalCountForStudent(studentId);
            if (activeRentals >= 2) { // Active rentals limit - 2
                throw new IllegalArgumentException("Student has reached the
rental limit.");
            }
            // Creating a lease
            instrumentDAO.createRental(studentId, instrumentId, startDate,
endDate, quantity);
            System.out.println("Rental created successfully.");
        } catch (Exception e) {
```

```
        throw new RuntimeException("Error creating rental: " +
e.getMessage());
    }
}

public void endRental(int rentalId) {
    try {
        instrumentDAO.endRental(rentalId);
        System.out.println("Rental ended successfully.");
    } catch (Exception e) {
        throw new RuntimeException("Error ending rental: " +
e.getMessage());
    }
}
}
```

### View Class

```
package view;

import controller.RentalController;
import integration.InstrumentDAO;

import java.util.Scanner;

public class RentalView {
    public static void main(String[] args) {
        InstrumentDAO dao = new InstrumentDAO();
        RentalController rentalController = new RentalController(dao);
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.println("What would you like to do?");
            System.out.println("1. Rent an instrument");
            System.out.println("2. End a rental");
        }
```

```
System.out.println("3. List of instruments");
int choice = scanner.nextInt();

switch (choice) {
    case 1:
        System.out.print("Enter student ID: ");
        int studentId = scanner.nextInt();

        System.out.print("Enter instrument ID: ");
        int instrumentId = scanner.nextInt();

        System.out.print("Enter rental start date (YYYY-MM-DD): ");
        String startDateStr = scanner.next();

        System.out.print("Enter rental end date (YYYY-MM-DD): ");
        String endDateStr = scanner.next();

        System.out.print("Enter quantity of instruments: ");
        int quantity = scanner.nextInt();

        rentalController.rentInstrument(studentId, instrumentId,
startDateStr, endDateStr, quantity);
        break;

    case 2:
        System.out.print("Enter rental ID to end: ");
        int rentalId = scanner.nextInt();
        rentalController.endRental(rentalId);
        break;

    case 3:
        System.out.print("Enter instrument type: ");
        int instrumentType = scanner.nextInt();
        dao.listAvailableInstruments(instrumentType);
        break;

    default:
        System.out.println("Invalid choice. Please try again.");
}
```

```
        } finally {  
            scanner.close();  
        }  
    }  
}
```

**Remark from Leif Lindbäck from 16 Dec at 15:19**

**1. It's a waste of time to prepare the prepared statements every time they're executed. You could do that once when the program starts.**

**How it is implemented in code: In the `prepareStatements()` method, a `PreparedStatement` object is created once when the `InstrumentDAO` object is initialized. These prepared statements are used multiple times without the need to be re-prepared.**

```
public InstrumentDAO() {  
  
    try {  
  
        conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);  
  
        conn.setAutoCommit(false); // Устанавливаем autocommit один раз  
  
        prepareStatements(); // Подготовка всех необходимых PreparedStatement  
  
    } catch (SQLException e) {  
  
        throw new RuntimeException("Error initializing database connection: " +  
            e.getMessage());  
  
    }  
  
}
```

**Remark from Leif Lindbäck from 16 Dec at 15:19**

**2. Autocommit should be set to false once and for all when the program starts, not each time the database is called.**

**How it is implemented in code: Setting the autocommit mode is done once when creating the connection in the constructor of the InstrumentDAO class. This ensures that all operations performed through this Connection object require an explicit commit() call to complete the transaction, rather than being performed automatically.**

```
conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
```

```
conn.setAutoCommit(false); // Disabling autocommit
```

**Remark from Leif Lindbäck from 16 Dec at 15:19**

**3. Task B is not accepted since it's not fully solved. It's not possible to list instruments or terminate rentals.**

**How it's fixed in code:**

**List of instruments: The listAvailableInstruments() method correctly selects available instruments based on the passed instrumentType. Output is sent to the console if instruments are found, or a message is displayed that instruments are missing.**

```
public void listAvailableInstruments(int instrumentType) {  
    try {  
        listAvailableStmt.setInt(1, instrumentType);  
        ResultSet rs = listAvailableStmt.executeQuery();  
  
        System.out.println("Available instruments:");  
        boolean found = false;  
        while (rs.next()) {  
            found = true;  
            int instrumentId = rs.getInt("instrument_id");  
            String type = rs.getString("types_instrum");  
            String brand = rs.getString("brand_id");
```

```

        double price = rs.getDouble("cost");
        System.out.println("Instrument ID: " + instrumentId + ", Type: " + type + ", Brand: " +
brand
        + ", Price: " + price);
    }
    if (!found) {
        System.out.println("No available instruments found for this type.");
    }
} catch (SQLException e) {
    throw new RuntimeException("Error listing available instruments: " + e.getMessage());
}
}

```

**Ending a Lease: The `endRental()` method gracefully ends a lease by updating the lease end date to the current date (`CURRENT_DATE`). It also checks for the existence of the specified lease before performing the update.**

```

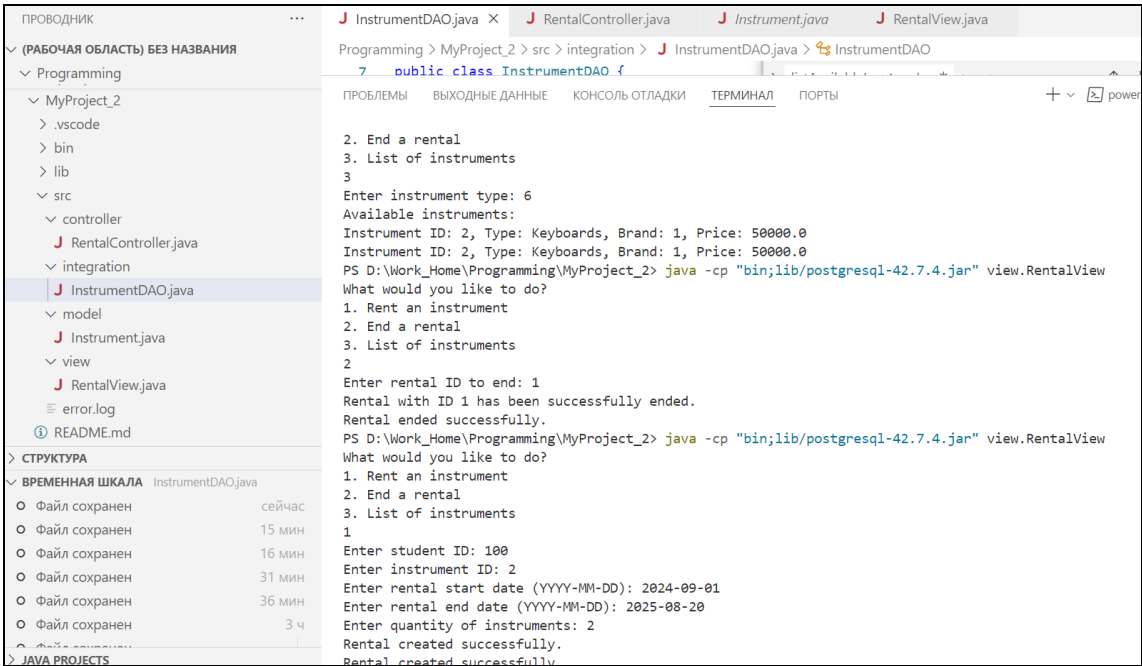
public void endRental(int rentingId) {
    try {
        String checkQuery = "SELECT renting_id, period_end FROM renting_instruments WHERE
renting_id = ?";
        try (PreparedStatement checkStmt = conn.prepareStatement(checkQuery)) {
            checkStmt.setInt(1, rentingId);
            ResultSet rs = checkStmt.executeQuery();
            if (!rs.next()) {
                throw new RuntimeException("Rental with ID " + rentingId + " not found.");
            }
        }

        endRentalStmt.setInt(1, rentingId);
        int rowsAffected = endRentalStmt.executeUpdate();
        if (rowsAffected > 0) {
            System.out.println("Rental with ID " + rentingId + " has been successfully ended.");
        } else {
            System.out.println("No rental found with ID " + rentingId + ".");
        }
        conn.commit();
    } catch (SQLException e) {
        try {
            conn.rollback();

```

```
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
    throw new RuntimeException("Error ending rental: " + e.getMessage());
}
}
```

5. Result



Запрос История запросов

```
1 select* from public.renting_instruments;
2
```

Data Output Сообщения Notifications

	renting_id [PK] integer	student_num integer	instrum_id integer	period_start date	period_end date	quantity_instruments integer	price_id integer
1		1	100	1	2024-09-01	2024-12-23	1
2		3	100	2	2024-09-01	2025-08-20	2



## 6. Discussion

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

Naming conventions are followed. All names are fairly self-explanatory.

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

Yes. Automatic transaction commit is disabled. All SQL statements are executed within a transaction. Transactions are committed on success and rolled back on failure.

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

SELECT FOR UPDATE is used in SQL statements that participate in a transaction that reads a value from the database, computes an update to the value, and stores the updated value in the database.

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

The program meets all the requirements specified in the task for listing tools, renting tools and terminating a rental.

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

The rental is marked as terminated. No information about the rental is deleted when the rental is terminated.

**6. The bullets below applies only to the higher grade task.**

**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?**

**Are also the view and cotroller layers completely without business logic?**

**Is the code easy to understand?**

**Is all duplicated code avoided?**

There is no business logic in the integration layer, DAO has only methods whose names start with Create, Read, Update or Delete. The presentation and controller layers are also completely devoid of business logic. The code is easy to understand. Code duplication is avoided.

**7. Comments About the Course**

All 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.