**Aufgaben „Datenbankanbindung“**

Zielsetzung der Aufgaben

Diese Aufgaben ermöglichen es, sich mit Datenbankanbindungen und den damit verbundenen Vorgängen in der Sprache Java und der IDE IntelliJ IDEA zu befassen, und darüber hinaus Handlungssicherheit im Umgang mit bereits Bekanntem zu gewinnen.

Alle Aufgaben werden in getrennten Projekten bearbeitet. Wiederholungen sollten nicht mittels Kopien gelöst werden, sondern jeweils einzeln geschrieben werden.

Vorbereitung

Legen Sie auf Ihrem lokalen PostgreSQL-Datenbankserver eine leere Datenbank namens "wetter" an. Diese Datenbank wird im Rahmen der Aufgabe 0 als "Spielwiese" dienen, um verschiedene Operationen durchzuführen.

Aufgabe 0-1: Den Datenbanktreiber für PostgreSQL einbinden

Finden Sie im Internet den PostgreSQL-JDBC-Treiber (bei mehreren verfügbaren Versionen nutzen Sie den mit der höchsten JDBC-Versionsnummer) und binden Sie den Treiber als Bibliothek in ein neues Projekt ein.

Aufgabe 0-2: Eine Verbindung herstellen und Metadaten abfragen (ohne Properties)

Schreiben Sie eine Klasse Connect mit einer main()-Methode.

Stellen Sie eine Verbindung mit der Datenbank "wetter" her. Die Fehlerbehandlung (try-with-resources) soll durch Ausgabe der Message und des SQLstate auf der Konsole erfolgen.

Lassen Sie sich von dieser Verbindung das java.sql.DatabaseMetaData-Objekt geben, fragen Sie bei diesem den DatabaseProductName und die DatabaseProductVersion ab und geben Sie beides auf der Konsole aus.

Fügen Sie nun nach der Ausgabe folgende Zeilen in Ihr Programm ein, um eine sicht- und steuerbare Pause zu erzwingen:

System.***out***.println(**"Enter drücken..."**);  
String s = **new** Scanner(System.***in***).nextLine();

Nun sollte das Programm in der ….nextLine()-Zeile auf eine Eingabe warten – mit einer offenen Datenbankverbindung.

Überzeugen Sie sich mit Hilfe des Programms *pgAdminIII* von der Existenz dieser Verbindung, indem Sie die SQL-Konsole von *pgAdminIII* benutzen, um folgenden Befehl auszuführen:

SELECT \* FROM pg\_stat\_activity;

Vergleichen Sie das Ergebnis mit und ohne laufendes Java-Programm. Bewerten Sie die Ergebnisse, insbesondere im Hinblick auf try-with-resources.

Aufgabe 0-3: Statement und Erzeugung von Tabellen

Richten Sie eine neue Datenquelle in IntelliJ ein, und verbinden Sie sie mit der Datenbank wetter, um zukünftig den "Umweg" über *pgAdminIII* vermeiden zu können.

Schreiben Sie eine Klasse Initialisierer mit einer main()-Methode, welche die Datei wetterdaten\_tabellen.sql einliest, die Zeilen der Datei zu Strings mit korrekten SQL-Befehlen zusammenfügt.

Die Befehle in wetterdaten\_tabellen.sql sind so gesetzt, wie sie von IntelliJ (und anderen SQL-Editoren) gesetzt werden – leider ist das ein Format, das gut für Menschen geeignet ist, aber schlecht für Maschinen.

Die SQL-Befehle müssen für die Verarbeitung mit JDBC

* aus einem String pro Befehl bestehen
* dürfen mit einem ";" am Ende abgeschlossen werden (bei PostgreSQL)

Schreiben Ergänzen Sie die Klasse Initialisierer um eine statische Methode

List<String> readSQLFile( String fileName )

welche die Textdatei, deren Name übergebenen wurde einliest, die Befehls-Strings zusammenbaut und in eine Liste schreibt, die zurückgegeben wird.

Lassen Sie sich nun von Ihrer Connection ein java.sql.Statement-Objekt geben, und lassen Sie alle Befehle aus der o.a. Liste jeweils mit execute(String sql) aus. Achten Sie darauf, dass das Statement im selben try-with-resources-block erzeugt wird, wie die Connection!

Überzeugen Sie sich davon, dass in der Datenbank die passenden Tabellen angelegt werden.

Aufgabe 0-3a: Statement im Batch-Modus

Um mehrere Befehle effizient zum DBMS zu kommunizieren, bietet das Statement einen Batch-Modus. Hier werden bekommt das Statement mehrere SQL-Befehle nacheinander übergeben, hält diese aber zurück bis es einen entsprechenden Ausführungsbefehl bekommt.

Dadurch kann die Kommunikation mit dem DMBS minimiert und Kommunikationsoverhead vermieden werden.

Kopieren Sie die Klasse Initialisierer und benennen Sie sie um in InitialisiererBatch (Rechtsklick → Refactor → Copy).

Ersetzen Sie den execute(String sql)-Teil durch eine geeignete Kombination aus addBatch(String sql)-Aufrufen und genau einem executeBatch()-Aufruf.

Bei diesem kurzen Beispiel werden Sie wahrscheinlich keinen Laufzeitunterschied feststellen können.

Aufgabe 0-4: PreparedStatement und Einfügen von Daten

Das Einfügen von Daten (INSERT) erfordert oft die wiederholte Ausführung desselben SQL-Statements mit unterschiedlichen Parametern (=Daten). Prepared Statements können vom DBMS gecacht werden, um das wiederholte Compilieren des Statements zu vermeiden.

Weiterhin schützen Prepared Statements bis zu einem gewissen Grad vor SQL-Injection-Attacken, da die eingefügten Daten automatisch *vor* der Ausführung des Statements maskiert/escaped und typgeprüft werden.

Schreiben Sie eine Klasse Betanker, deren main()-Methode eine Befüllung der Datenbank-Tabellen mittels PreparedStatements vornimmt. Analog zu der Methode readSQLFile() aus Aufgabe 0-3 benötigen Sie eine Methode readCSVFile(), die einen Dateinamen übernimmt und eine List<String> mit den einzelnen Zeilen (=Datensätzen) zurückgibt. Achten Sie darauf, die Kopfzeile (in der sich die Spaltennamen befinden, aber kein Datensatz) nicht mit zu übernehmen!

Achtung: die Schlüssel sind hier bereits vorhanden und werden mit in die Datenbank eingefügt!

Für das Befüllen der einzelnen Tabellen bietet es sich wieder an, Methoden zu schreiben.

Beginnen Sie mit der Tabelle wetterstation (aus der Datei wetterdaten\_Wetterstation.csv) und der zugehörigen Methode fillStations(). Als Trennzeichen wird in der Datei ein Semikolon (';') genutzt.

Nach der Aufspaltung einer Zeile in die einzelnen Spalten müssen dem PreparedStatement die Parameter mitgeteilt werden – nachdem sie in den richtigen Typ umgewandelt wurden. Bei Kommazahlen gibt es hier das Problem, dass in der Datei ein ',' als Dezimaltrennzeichen genutzt wurde, Java aber für Double.parseDouble() einen '.' erwartet.

Spalten, für die keine Daten existieren, müssen auf (SQL-) NULL gesetzt werden. Dies geschieht mit der Methode setNull(int parameterIndex, int sqlType) des PreparedStatement-Objekts. Der SQL-Datentyp (zweiter Parameter) muss über den JDBC-Treiber ermittelt werden:

JDBCType.XYZ.getVendorTypeNumber()

z.B. JDBCType.INTEGER.getVendorTypeNumber() für den SQL-Datentyp INTEGER.

Die Daten der Wetterstation *Berlin-Dahlem* sind unvollständig, daher ist dieser SQL-NULL-Trick erforderlich (das ist aber nicht zu ändern).

Alle Parametersätze sollen zunächst mittels addBatch() dem Ausführungsstapel des PreparedStatements hinzugefügt werden, danach wird der Ausführungsstapel mit executeBatch() einmalig ausgeführt.

Füllen Sie dann die Tabelle wettermessung (aus der Datei wetterdaten\_Wettermessung.csv) und der zugehörigen Methode fillSamples(). In dieser Datei gibt es keine unvollständigen Daten.

Um das Datum in den richtigen Datentyp (java.sql.Date)[[1]](#footnote-1) zu parsen gibt es die statische Methode java.sql.Date.valueOf(String s), die einen String in der Form yyyy-mm-dd in ein Datum verwandelt, das in den SQL-Datentyp DATE umgewandelt werden kann.

Aufgabe 0-5: Abfragen und ResultSets

Nachdem die Datenbank nun angelegt und befüllt ist, sollen von einem Java-Progamm aus Abfragen gestellt werden und die Ergebnisse der Abfragen ausgewertet werden.

Als Beispiel soll herausgefunden werden, wann und wo die längste Sonnenscheindauer stattfand:

**SELECT standort**, **datum**, **sonnenscheindauer FROM** wetterstation  
 **INNER JOIN** wettermessung **ON** wetterstation.**s\_id** = wettermessung.**fk\_s\_id  
WHERE sonnenscheindauer** = (**SELECT** *max*(**sonnenscheindauer**) **FROM** wettermessung);

Beantworten Sie diese Frage zunächst mit einem SQL-Client (damit Sie vorher wissen, was das Java-Programm antworten sollte).

Schreiben Sie eine Klasse Abfragen, deren main()-Methode eine Datenbank-Verbindung herstellt und mittels eines PreparedStatement-Objekts und dessen executeQuery()-Methode die o.a. Abfrage stellt. Als Ergebnis-Objekt erhalten Sie ein ResultSet, dessen Inhalt zeilenweise auf der Konsole auszugeben ist.

Aufgabe 1: FakePersonenGenerator an der Datenbank

Legen Sie ein neues Projekt an. Binden Sie die aktuellen PostGreSQL-Treiber ein, und legen Sie mittels *pgAdminIII* eine Datenbank personen an. Legen Sie in IntelliJ eine Datenquelle für diese Datenbank an.

Kopieren Sie die Klassen Person, Names und eventuelle Generator-Klassen aus dem FakePersonGenerator-Projekt kopieren, so dass hier auch Personen-Objekte erzeugt werden können.

Versehen Sie die Person-Klasse mit Gettern für alle Attribute und die Kombination strasse + " "+hausnummer.

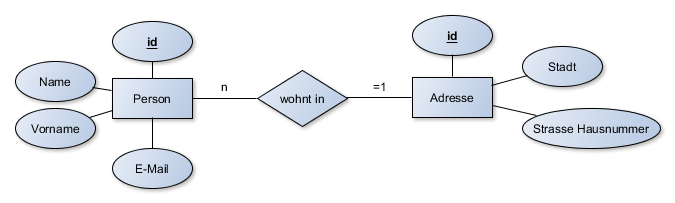
Schreiben Sie eine Klasse PersonDBConnector mit folgenden Methoden

* public PersonDBConnector(String url, String user, String pass) : Konstruktor, stellt die DB-Verbindung her (Fehlerbehandlung ausserhalb)
* public void dbAnlegen() : Anlegen der o.a. Tabellen in der Datenbank persondb mit vorherigem DROP IF EXISTS (Fehlerbehandlung ausserhalb)
* public void speichern( Person p ): das übergebene Person-Objekt p mittels PreparedStatement-Objektes in die Datenbank schreiben. Achten Sie auf die nötige Einfügereihenfolge! (Fehlerbehandlung selbst)
* private int adresseIdHolen(Person p) : holt die als Fremdschlüssel benötigte id der Adresse mittels SELECT (Fehlerbehandlung selbst)
* private int adresseSpeichern(Person p) : Speichern der Adresse in einer separaten, privaten Methode passieren, die id der Adresse zurückgibt – diese Methode muss entscheiden, ob sie eine Adresse neu einfügen muss, oder ob die Adresse bereits vorhanden ist. (Fehlerbehandlung selbst)
* public void disconnect() schlißt die DB-Verbindung (Fehlerbehandlung ausserhalb)

Der Konstruktor von PersonDBConnector soll die Datenbankverbindung herstellen (mit URL und Zugangsdaten aus Konstruktor-Parametern). Das entstehende Connection-Objekt muss in einem Attribut gespeichert werden.

Schreiben Sie ein Programm, das sich mittels eines PersonDBConnector-Objekts und mit dem Datenbankserver verbindet und die Tabellen gem. u.a. ER-Modell anlegt (dieses Modell ist nicht vollständig normalisiert, das soll aber so sein, damit die DB-Struktur nicht zu komplex wird).

Die ids sind dabei vom Typ SERIAL (vgl. PostGreSQL-Dokumentation, Kap. 8.1.4), alles andere vom Typ TEXT (ebd. Kap. 8.3.). Straße und Hausnummer werden vor dem Einfügen zu einem einzigen String verkettet.



Achten Sie darauf, die Fehlerbehandlung in der main()-Methode durchzuführen. Nutzen Sie finally (vgl. Exceptions-Kapitel) um sicherzustellen, dass die DB-Verbindung auf jeden Fall geschlossen wird.

Fügen Sie 10 000 Personen in die Datenbank ein und überzeugen Sie sich davon, dass die Datensätze tatsächlich in der Datenbank ankommen.

Aufgabe 2 : Das Properties-Objekt

Erforschen Sie die Klasse java.util.Properties und die zugehörigen Properties-Dateien. Wiederholen Sie Aufgabe 0-2 und nutzen Sie statt der Benutzername- und Passwort-Strings ein entsprechendes Properties-Objekt, das die nötigen Informationen aus einer Properties-Datei einliest (später können Sie das gleich für Aufgabe 1 und 3 machen).

Als zusätzliches Schlüssel-Wert-Paar verpacken Sie die Datenbank-URL in die Properties-Datei. Das wird zwar vom DriverManager nicht direkt ausgewertet, aber Sie können diese Auswertung selbst durchführen - und bekommen "zur Belohnung" einen zentralen Ort, an dem Ihre gesamte DB-Konfiguration gespeichert ist.

1. **Achtung**: es gibt auch noch java.util.Date und möglicherweise noch diverse Dinge in java.time.\* - das ist alles *nicht* das richtige! [↑](#footnote-ref-1)