Java Database Connectivity

Gracin Denis, IB 4 C

Agenda

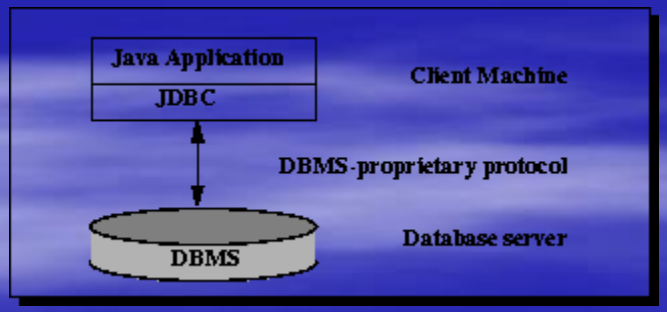
- 1. JDBC-Architektur
- 2. Treiber der JDBC
  - 2.1 Typ-1 Treiber
  - 2.2 Typ-2 Treiber
  - 2.3 Typ-3 Treiber
  - 2.4 Typ-4 Treiber
- 3. Verbindungsablauf
- 4. Connection Pool
- 5. Datenbankzugriff mit JDBC
- 6. Verbindungsaufbau
- 7. Übermittlung von Abfragen
- 8. Schwachstellen von Statement

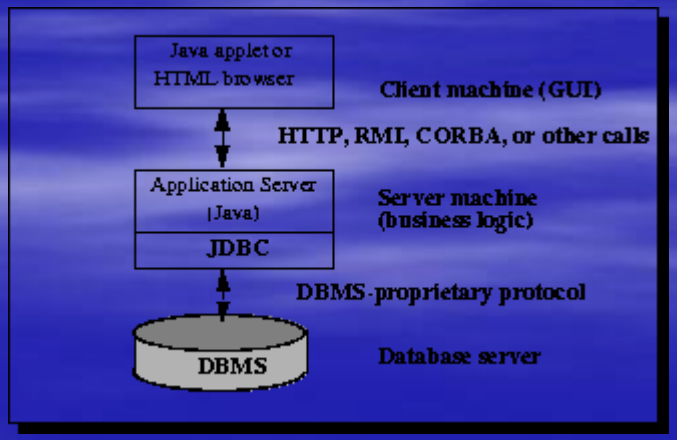
Agenda

- 9. Prepared Statement
- 10. Ergebnisverwaltung
- 11. Besonderheiten von ResultSet
- 12. Änderung am ResultSet
- 13. Verbindungen schließen
- 14. Spring Framework
- 15. Verbindungsaufbau mit Spring
- 16. RowMapper
- 17. Abfrage Übermittlung mit Spring
- 18. Update Methode
- 19. Sprint-Fazit

1. JDBC-Architektur

- Unterstützt das 2-Schichten- und 3-Schichten-Modell
- 2-Schichten-Modell: Direkte Verbindung zur Datenbank
- 3-Schichten-Modell: Verbindung über eine mittlere Schicht
- Die mittlere Schicht ist meist in C oder C++ geschrieben, da beste Perfomance
- Durch Optimierungen gewinnt Java immer mehr an Bedeutung

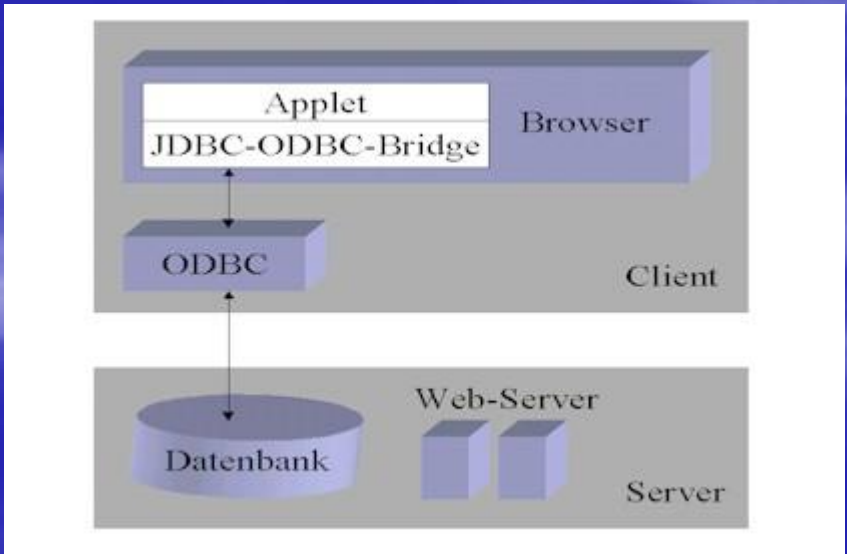




2. Treiber der JDBC

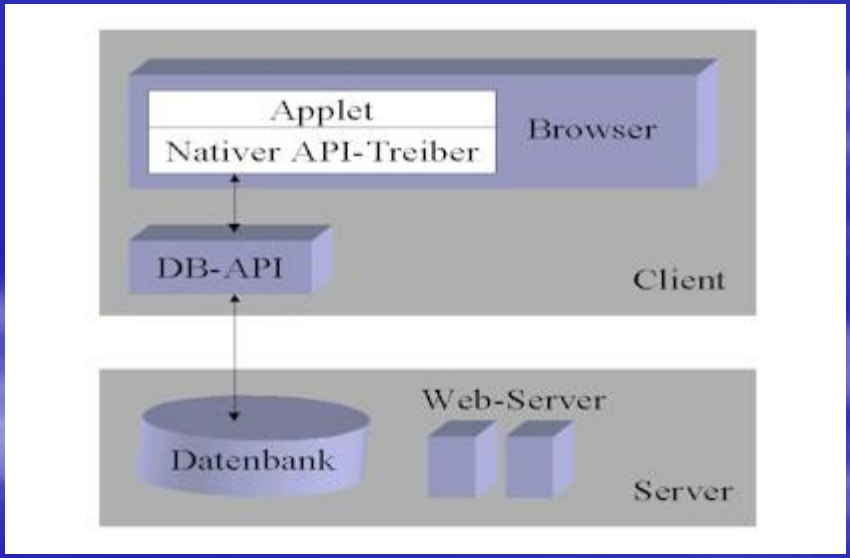
2.1 Typ-1 Treiber: JDBC-ODBC-Bridge

- Brücke zwischen Java Anwendung und ODBC Schnittstelle
- Wird nur noch verwendet wenn die Datenbank keine JDBC-Treiber installiert hat
- Nachteil: Client benötigt ODBC-Treiber, und Zugriff auf Native Methoden



2.2 Typ-2 Treiber

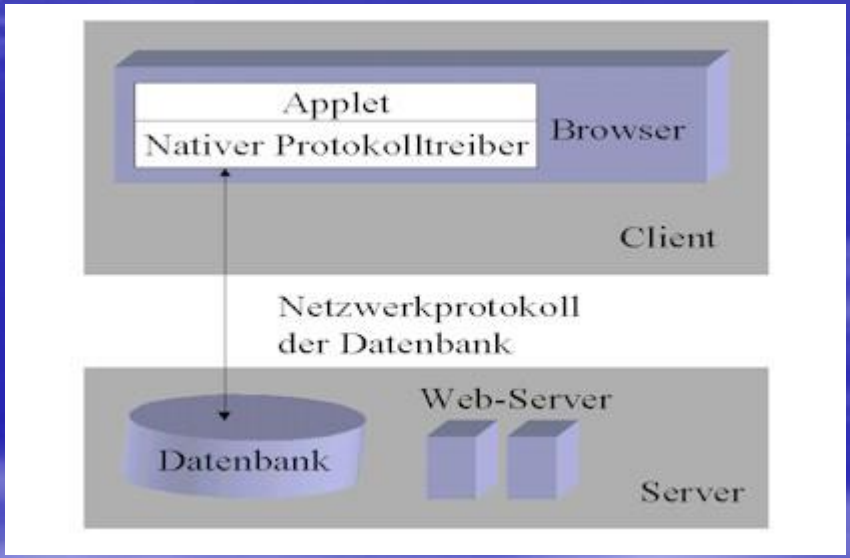
- Übersetzt die Aufrufe direkt in Datenbank-API aufrufe
- Nachteil: Auch hier benötigt der Client Treiber und den Zugriff auf Native Methoden



2.3 Typ-3 Treiber

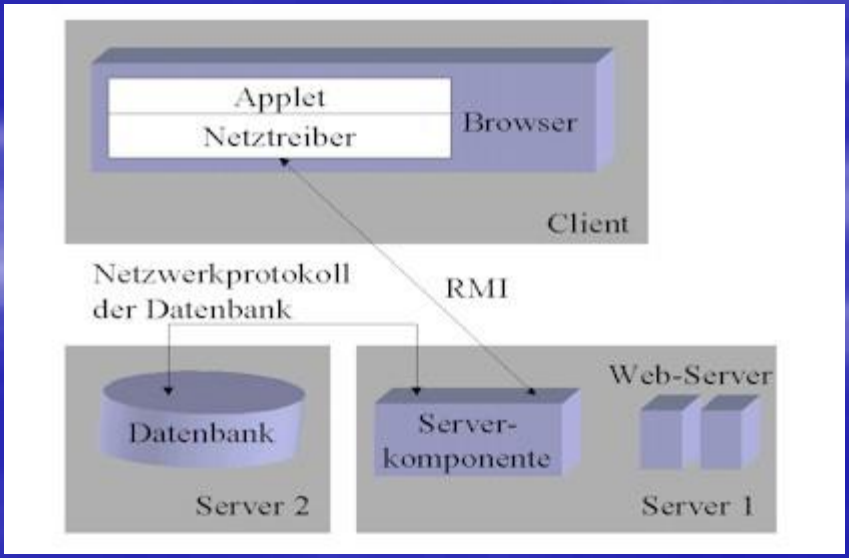
- Universeller Treiber, welche erst beim Datenbankzugriff auf den Client geladen werden.
- Kommunikation erfolgt über eine Software-Schnittstelle (Middleware)
- Vorteil: Keine zusätzlichen Treiber werden benötigt und schneller als Typ 1 und 2
- Besonderheit: Verbindung kann verschlüsselt werden
- Gut da die meisten Datenbanken keine verschlüsselten Datenbankverbindungen anbieten
- Typ-3 eignet sich im Zusammenhang mit Firewalls sehr gut für Internet-Protokolle

2.3 Typ-3 Treiber



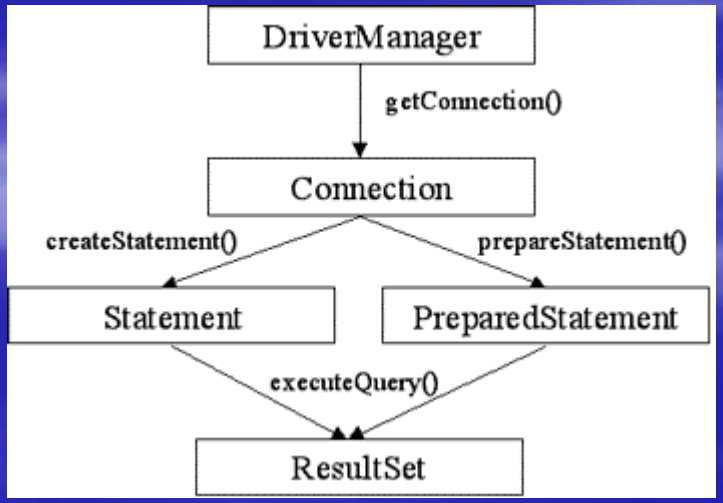
2.4 Typ-4 Treiber

- Treiber ist vollständig in Java geschrieben
- Kommunikation erfolgt direkt mit dem Datenbankserver
- Spricht über datenbankspezifische Protokolle direkt mit der Datenbank über einen offenen IP-Port
- Vorteil: keine zusätzlichen Treiber, schneller als alle anderen Treiber, Server übersetzt die Befehle



3. Verbindungsablauf

- Schritt 1: JDBC-Datenbanktreiber laden
- Schritt 2: Datenbankverbindung aufbauen
- Schritt 3: SQL-Anweisungsobjekt erzeugen
- Schritt 4: SQL-Anweisung ausführen
- Schritt 5: Ergebnisse auswerten
- Schritt 6: SQL-Anweisungsobjekt schließen
- Schritt 7: Datenbankverbindung schließen



3. Verbindungsablauf

- Alle Schritte sind in Try-Catch-Blocks zu implementiren um Fehler abzufangen
- Schritt 7 ist besonders wichtig, da Datenbankverbindungen relativ schwergewichtig sind und einige Systemressourcen verbrauchen
- Empfehlung: Nicht für jede, einzelne SQL-Abfrage eine Datenbankverbindung etablieren
- Größere Firmen, welche ständigen Zugriff benötigen, besitzen daher einen „Connection Pool“

4. „Connection Pool“

- Stellt einige Verbindungen mit der Datenbank her und verwaltet diese
- Programme welche auf die Datenbank zugreifen erhalten eine Verbindung
- Diese Verbindung wird im „Connection Pool“ auf besetzt markiert und später wieder freigegeben
- Dadurch lassen sich Overheads, welche durch ständiges Auf- und Abbauen von Verbindungen vermeiden
- „Connection Pools“ mussten früher von Hand realisiert werden
- Seit JDBC 2.0 wird dies von Klassen unterstützt

5. Datenbankzugriff mit JDBC

Verbindungsaufbau

- Seit JDBC 4.0 muss der Datenbanktreiber nicht mehr geladen werden
- Dies übernimmt der Drivermanager
- Mit der getConnection(string url) wird die Verbindung aufgebaut
- Wichtige URLs die für einen Verbindungsaufbau notwendig sind
- Je nach Datenbank wird die dementsprechende URL benötigt

DBMS	URL
Derby	jdbc:derby:net://servername:port/
HSQLDB	jdbc:hsqldb:hsql://servername:port/database
MySQL	jdbc:mysql://servername:port/database
Oracle	jdbc:oracle:thin:@host:port:database

6. Verbindungsaufbau

```
public class DatenbankAbfragen {
    static Connection con = null;
    static Statement stm = null;
    static ResultSet rs = null;

    public DatenbankAbfragen()
    {
        try
        {
            // Optional  seit JDBC 4
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch(ClassNotFoundException e)
        {
            e.printStackTrace();
            System.exit(1);
        }

        // Datenbankverbindungseinstellungen definieren
        final String url = "jdbc:mysql://localhost/filme";
        final String loginName = "gracin";
        final String password = "asdf";

        try
        {
            // Stellt die Verbindung zur Datenbank her.
            con = DriverManager.getConnection(url, loginName, password);
            // SQL-Befehlsobjekt
            stm = con.createStatement();
        }
        catch(SQLException e)
        {
            e.printStackTrace();
            System.exit(1);
        }
    }
```

7. Übermittlung von Abfragen

- Eine Möglichkeit ist das Interface Statement
- Mit der Methode „executeQuery(string query)“ können gewöhnliche Abfragen als String verschickt werden
- Statt Ist-Gleich-Zeichen wird hier ein „like“ verwendet, da Suchfunktion
- Das Prozentzeichen sorgt dafür, dass die Datenbank alle Werte, die den übergebenen String enthalten, ausgibt

```
try {
    rs = stm.executeQuery(
            "SELECT * " +
            "FROM Filme " +
            "WHERE title like '%"+title+"%'");

    // Auswertung der Ergebnisse
    while(rs.next()){
        System.out.print("Nr: " + rs.getString(1)+"\t");
        System.out.print("Titel: " + rs.getString(2)+"\t");
        System.out.println("Notiz: " + rs.getString(3));
    }
}
catch(SQLException e)
{
    e.printStackTrace();
    System.exit(1);
}
```

7. Übermittlung von Abfragen

- Um die Datenbank zu aktualisieren, benötigen wir die Methode „executeUpdate(string query)“
- Das „i“ steht für die Anzahl der aktualisierten Datensätze
- Nach der Aktualisierung kann dann überprüft werden ob der gewünschte Datensatz aktualisiert wurde oder nicht

```
int i = stm.executeUpdate(
        "UPDATE Filme " +
        "SET title = '"+title+"', notiz = '"+notiz+"'" +
        "WHERE nr =" + nr);
```

8. Schwachstellen von Statements

- Allerdings besitzt das Interface „Statement“ ein paar Schwachstellen
- 1. Schwachstelle:
  - Bei Aktualisierung einer Liste mit Objekten, wird Abfrage immer vom neuem Verarbeitet: schlechte Performance
- 2. Schwachstelle:
  - Probleme mit der SQL-Syntax, diese erkennt die Apostrophe als String-Vergleichsfunktion
  - Beispiel: Suchen wir eine Person namens O'Connor
  - Abfrage hierfür: "SELECT * FROM Person WHERE Name = 'O'Connor' “
  - Datenbank erkennt den String als „O“
  - Programm wirft Exception: „java.sql.SQLException: unexpected token: CONNOR"
- 3. Schwachstelle:
  - Leichtes einschleusen Schadhafter SQL-Anweisungen (SQL-Injection)
- Schutz bietet Prepared Statement

9. Prepared Statement

- Prepared Statement verarbeitet den String etwas anders
- Es erstellt eine Art Schablone, welche Parameter erwartet und beliebig oft wiederholt werden kann
- Die Stellen, wo Parameter erwartet werden, werden mit einem Fragezeichen gekennzeichnet und sieht wie folgt aus:

```
String sqlUpdate = "UPDATE Filme " +
        "SET title = ?, notiz = ?" +
        "WHERE nr = ?";

stm = con.prepareStatement(sqlUpdate);
stm.setString(1, title);
stm.setString(2, notiz);
stm.setInt(3, nr);
int i = stm.executeUpdate();
```

10. Ergebnisverwaltung

- Die Ergebnisse aus der Datenbank werde in einem ResultSet-Objekt gespeichert
- Mit Hilfe einer while-Schleife kann dieses ausgewertet werden
- Bevor die Daten ausgelesen werden können, muss die Methode „next()“ mindestens einmal ausgeführt werden
- Dies ist Notwendig da der Datensatzzeiger (Cursor) immer vor den Datensätzen steht
- Die Folge davon ist, dass der Index ungewohnter weise bei 1 beginnt
- Standardmäßig kann sich im ResultSet nur vorwärts bewegen und einmal angesprochen werden

```
// Auswertung der Ergebnisse
while(rs.next()){
    System.out.print("Nr: " + rs.getString(1)+"\t");
    System.out.print("Titel: " + rs.getString(2)+"\t");
    System.out.println("Notiz: " + rs.getString(3));
}
```

11. Besonderheiten von ResultSet

- ResultSets können im Statement den Bedürfnissen nach konfiguriert werden
- Das ResultSet bietet hiefür vordefinierte Methoden an

```
Statement stm(int resultSetType, int resultSetConcurrency) throws SQLException;
```

- Mit dem Parameter „resultSetType“ lässt sich folgendes einstellen:
  - ResultSet.TYPE_FORWARD_ONLY: Laufrichtung nur Vorwärts, einmaliges Ansprechen
  - ResultSet.TYPE_SCROLL_INSENSITIVE: Laufrichtung beliebig, mehrmaliges Ansprechen, Änderungen in der Datenbank werden nicht übernommen
  - ResultSet.TYPE_SCROLL_SENSITIVE: Laufrichtung beliebig, mehrmaliges Ansprechen, Änderungen in der Datenbank werden übernommen

- Mit dem Parameter „resultSetConcurrency“ lässt sich folgendes einstellen:
  - Resulst.Set.CONCUR_READ_ONLY: Sandard, erlaubt keine Änderung auf die Datenbank
  - Resulst.Set.CONCUR_UPDATABLE: Änderungen auf die Datenbank sind erlaubt, jedoch nur wenn die Abfrage keine Joins oder Gruppierfunktionen enthält, da diese nicht mehr zugeordnet werden können

12. Änderung am ResultSet

- Sofern eingestellt, kann ein ResultSet Datenbankeinträge verändern
- Mit der „updateString()“ Methode lässt sich der aktuell angezeigte Datensatz ändern und mit „updateRow()“ in die Datenbank übernehmen

```
restultSet.updateString(columnName, newValue);
restultSet.updateRow();
```

- Falls jedoch fälschlicherweise ein Datensatz verändert wurde, kann die Datenbank mit „cancelRowUpdate()“ in den Ursprungszustand versetzt werden
- Des Weiteren gibt es noch die Methoden „refreshRow()“, „insertRow()“ und „deleteRow()“

13. Verbindungen schließen

- Um den Datenbankserver zu entlasten sollten nicht mehr benötigte Verbindungen, mit der Methode „close()“, geschlossen werden
- Mit Verbindungen werden Connections, Statements und ResultSets gemeint
- Jede sollte in einem eigenem try-catch-block aufgefangen werden, damit diese trotz evtl. auftretender Fehler geschlossen werden

```
try
{
    rs.close();
}
catch (SQLException e)
{
    e.printStackTrace();
}
```

```
try
{
    stm.close();
}
catch (SQLException e)
{
    e.printStackTrace();
}
```

```
try
{
    con.close();
}
catch (SQLException e)
{
    e.printStackTrace();
}
```

14. Spring Framework

- In einer xml-Datei können die relevanten Daten für die Verbindung zur Datenbank gespeichert werden
- Vorteil hier, durch ledigliches austauschen der Datei kann eine andere, gleich aufgebaute, Datenbank angesprochen werden

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">
6
7     <!-- Initialization for data source -->
8     <bean id="dataSource"
9        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10       <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
11       <property name="url" value="jdbc:mysql://localhost/test"/>
12       <property name="username" value="gracin"/>
13       <property name="password" value="asdf"/>
14    </bean>
15
16    <!-- Definition for studentJDBCTemplate bean -->
17    <bean id="studentJDBCTemplate"
18       class="com.tutorialspoint.StudentJDBCTemplate">
19       <property name="dataSource"  ref="dataSource" />
20    </bean>
21
22 </beans>
```

15. Verbindungsaufbau mit Spring

- Das Hauptprogramm holt sich die xml-Datei und stellt mit den gespeicherten Daten eine Verbindung her
- Hierfür wird kein try-catch-block mehr benötigt, da Spring dies automatisch übernimmt
- Im folgendem Beispiel stellen wir eine Verbindung zu einer Schülerdatenbank her

```
ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

StudentJDBCTemplate studentJDBCTemplate =
        (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");
```

16. RowMapper

- Zuerst muss man eine Hilfsklasse erstellen, namens „StudentMapper“
- Diese soll die erhaltenen Datensätze in ein Student-Objekt speichern

```
public class StudentMapper implements RowMapper<Student> {
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        Student student = new Student();
        student.setId(rs.getInt("id"));
        student.setName(rs.getString("name"));
        student.setAge(rs.getInt("age"));

        return student;
    }
}
```

17. Abfrage Übermittlung mit Spring

- Wie zuvor wird die Abfrage in einem String gespeichert und an die Datenbank gesendet
- Durch vorgefertigten Methoden übergeben wir die Abfrage und Parameter, in dem Fall als Student-Objekt
- Für einzelne Objekte wird die Methode „queryForObject()“ benötigt
- Verbinungsaufbau, -abbau und try-catch übernimmt die das „jdbcTemplateObject“

```
try {
    rs = stm.executeQuery(
            "SELECT * " +
            "FROM Filme " +
            "WHERE title like '%"+title+"%'");

    // Auswertung der Ergebnisse
    while(rs.next()){
        System.out.print("Nr: " + rs.getString(1)+"\t");
        System.out.print("Titel: " + rs.getString(2)+"\t");
        System.out.println("Notiz: " + rs.getString(3));
    }
}
catch(SQLException e)
{
    e.printStackTrace();
    System.exit(1);
}
```

```
public List<Student> listStudents()
{
    String SQL = "select *"
            + "from Student";
    List<Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
    return students;
}
```

18. Update Methode

- Wie in Prepared Statement werden Fragezeichen als Platzhalter genutzt
- Bei Spring muss allerdings nicht die Position des Fragezeichen bestimmt werden und kann einfach in der Update-Methode übergeben werden

```
String sqlUpdate = "UPDATE Filme " +
        "SET title = ?, notiz = ?" +
        "WHERE nr = ?";

stm = con.prepareStatement(sqlUpdate);
stm.setString(1, title);
stm.setString(2, notiz);
stm.setInt(3, nr);
int i = stm.executeUpdate();
```

```
// Aktualisiert den Datensatz
public void update(Integer id, Integer age)
{
    String SQL = "update Student set age = ? where id = ?";
    jdbcTemplateObject.update(SQL, age, id);
    System.out.println("Update Record with ID = " +id);
}
```

19. Spring-Fazit

- Super Framework welches das arbeiten mit JDBC sehr erleichtert und übersichtlicht gestaltet
- Jedoch sollte für ein besseres Verständnis, die Grundzüge von JDBC erlernt und verstanden werden

Noch fragen?