Hochschule München

Abbildung 1: JDBC[1]

Gracin Denis

Betreuer: Dipl.-Inform. Michael Theis

Fach: Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

[1] http://howtodoinjava.com/wp-content/uploads/JDBC-Icon.png

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Einleitung

Diese Arbeit soll dem Leser ein Grundverständnis für den Aufbau und die Nutzung von JDBC in Verbindung mit einer MySQL Datenbank vermitteln. Außerdem sollen die Vorzüge des Spring Frameworks näher erläutert werden. Hierzu wurde ein Lokaler MySQL Datenbankserver eingerichtet und zwei kleine Java-Programme geschrieben, welche auf den besagten Server zugreifen.

2 Begriffsklärung

2.1 JDBC

Seit JDK 1.1 ist JDBC Bestandteil der Java Bibliothek. JDBC steht für "Java Database Connectivity" und ist an Microsofts ODBC, "Open Database Connectivity", angelehnt. Da aber ODBC mit C-Sharp realisiert wurde, konnte es nicht komplett übernommen werden und musste an das Objektorientierte Design von Java angepasst werden.

JDBC stellt einen einfachen Mechanismus bereit, der einen leichten Zugriff auf Datenbanksysteme ermöglicht. Dabei nehmen Treiber die Schnittstellen zwischen dem Programm und der Datenbank ein. Da sich der Quellcode im Wesentlichen nicht ändert, kann man mit diesem, auf jede beliebige Datenbank zugreifen, sofern Treiber dafür existieren. Dies verdanken wir unter anderem auch der Standarddatenbanksprache SQL, "Structured Query Language".[2]

2.2 Datenbanksystem

Der Zugriff und die Verwaltung von Informationen spielen heutzutage eine immer größere Rolle, egal ob für Unternehmen, Wissenschaftler, etc. Um diese enorme

[2] http://www.java.seite.net/jdbc/index.html (aufgerufen am 05.05.2014)

elektronischen Datenmengen, die sich laut Statistiken alle 5 Jahre verdoppeln, in den Griff zu bekommen musste etwas entwickelt werden.
Deshalb wurde ein Datenbankverwaltungssystem, oder auch nur Datenbanksystem, entwickelt um auf diese Daten schnell zugreifen und diese verwalten zu können.[3]

Mittlerweile gibt es eine Vielzahl an verschiedenen Datenbanksystemen, welche die Daten auf verschiedene Art und Weise speichern und verwalten. Für diese Studienarbeit verwende ich eine relationale Datenbank.
Eine relationale Datenbank besteht aus beliebig vielen Tabellen, die sich auf beliebige Art und Weise verknüpfen lassen. Die Daten werden in Spalten, so-genannten Attributen, und Reihen als Datensätze gespeichert. Wichtig ist, dass die Daten richtig, also dem Datentyp entsprechend, gespeichert werden und keine Doppelungen vorkommen dürfen.[4]

3 MySQL Datenbankserver

Wie in der Einleitung bereits erwähnt, wurde für diese Studienarbeit ein lokaler Datenbank-Server eingerichtet.

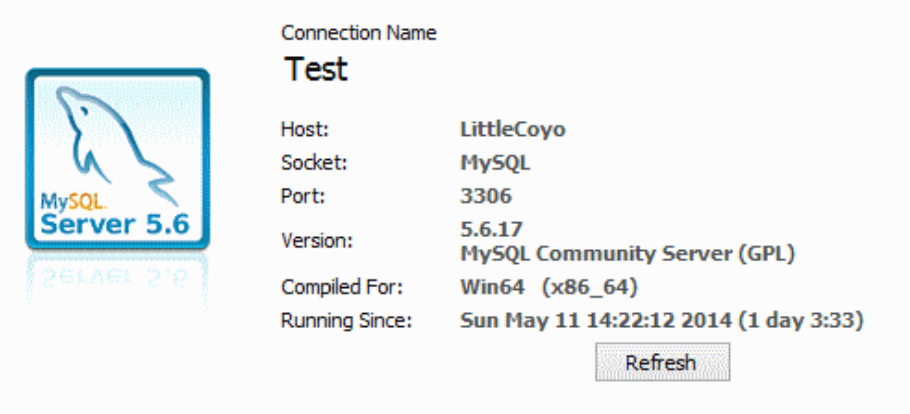

Abbildung 2: Lokaler Datenbankserver

Um den Server zu verwalten, wurde das Programm "MySQL Workbench 6.1" installiert. Dieses kann man kostenlos auf der Oracle Webseite herunterladen. Mit

[3] Datenbanksysteme - Alfons Kemper, André Eickler S.21
[4] http://www.itwissen.info/definition/lexikon/Relationale-Datenbank-database-relational.html (aufgerufen am 05.05.2014)

diesem Programm lassen sich einfache Schemen, also Datenbanken, erstellen und bearbeiten. Wie sich herausstellte war die Einrichtung des Datenbanksystems schnell erledigt und nach einer kurzen Einarbeitung konnte man die wichtigsten Funktionen aneignen.
Für die geschriebenen Java Programme wurden zwei Datenbanken mit den Namen "filme" und "student" erstellt. Die Datenkbank "filme" enthält eine Tabelle "Filme". Das zweite Programm greift auf die Datenbank "student" zu, verhält sich aber vom Ablauf anders. Es stellt eine Verbindung her, erstellt eine Tabelle "Student", befüllt diese und löscht sie dann wieder.
Neue Benutzer können schnell erstellt werden und die Rechteverwaltung ist auch benutzerfreundlich angelegt, so kann man z.B. die maximale Anzahl der Queries, die ein Benutzer pro Stunde abgeben darf, limitieren. Auch kann man einstellen, ob der Benutzer über Lese- oder auch Schreibrechte verfügen soll.

4 Grundlagen der JDBC API

4.1 Architektur der JDBC API

Die JDBC Architektur unterstützt das 2-Schichten- und 3-Schichten-Modell.

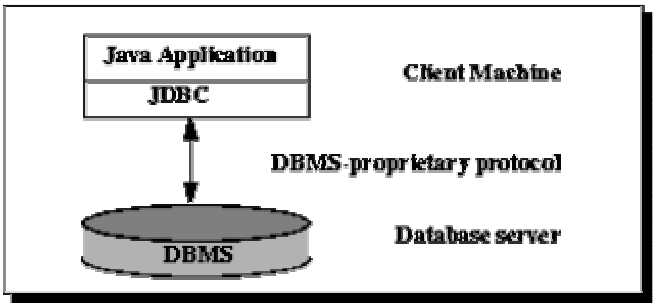

Abbildung 3: 2-Schichten-Modell [5]

Im 2-Schichten-Modell greift der Benutzer direkt auf die Datenbank zu, hierfür wird ein JDBC-Treiber benötigt, um eine Verbindung aufzubauen. Sobald eine Verbindung aufgebaut wurde, können die Datenbankbefehle direkt an die Datenbank

[5] http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html (aufgerufen am 22.04.2014)

gesendet werden und der Benutzer erhält die Ergebnisse der ausgeführten Abfrage. Normalerweise ist die Zielquelle auf einem entferntem Server, welcher mit einem Netzwerk, sei es Internet oder Intranet, verbunden ist. Hier fungiert der Benutzer als Client und der Server als Host.[6]

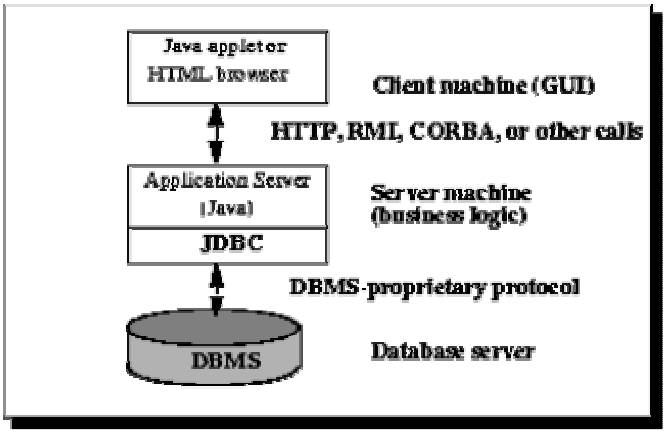

Abbildung 4: 3-Schichten-Modell [7]

Im 3-Schichten-Modell werden die Datenbankbefehle an eine mittlere Schicht übermittelt, die diese Abfrage dann weiter an die Datenbank übergibt. Die Datenbank verarbeitet die erhaltenen Befehle und sendet die Rückgabe zurück an die mittlere Schicht, diese wiederum zurück zum Benutzer. Ein Vorteil dieses Verfahrens ist, dass die Programme einfacher gestaltet sind, da man sich nicht mehr um die Datenbank Verbindung kümmern muss.
Die mittlere Schicht ist meist in C oder C++ geschrieben, da diese Programmiersprache die beste Perfomance liefern. Mit der Optimierung der Compiler, welche Java Bytecode in effizienten Maschinen-Code kompiliert, gewinnt die Java Plattform für die mittlere Schicht immer mehr an Bedeutung.

Da die Unternehmen immer mehr Java als Serversprache verwenden, bekommt JDBC als mittlere Schicht eine immer größere Bedeutung. Einige der Funktionen, wie "connection pooling", "distributed transactions" und "disconnected rowsets", machen JDBC zu einer guten Server Technologie. [8]

[6] http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html (aufgerufen am 22.04.2014)
[7] http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html (aufgerufen am 22.04.2014)
[8] http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html (aufgerufen am 22.04.2014)

4.2 Treiber von JDBC API

4.2.1 Typ-1 Treiber

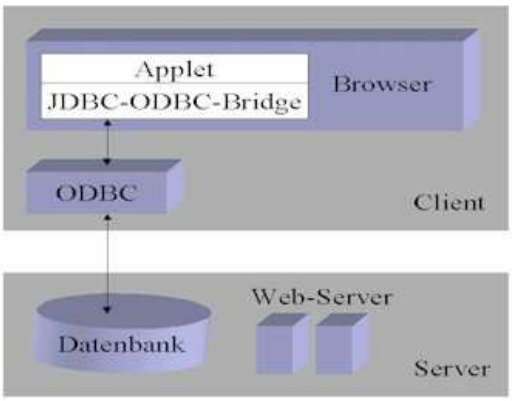

Abbildung 5: Typ -1 Treiber

Typ-1, auch bekannt als JDBC-ODBC-Bridge, fungierte anfangs, wie der Name schon sagt, als Brücke zwischen einer Java Anwendung und der damals genormten ODBC Schnittstelle von Microsoft. Diese wandelte die clientseitigen Aufrufe von JDBC in ODBC-Aufrufe um. Damit diese Umwandlung funktioniert, ist die Installation eines ODBC-Treibers notwendig und daher ungeeigenet für den Einsatz im Internet. Außerdem ist zu erwähnen, dass der Typ-1-Treiber native Methoden besitzt und diese Methoden ebenfalls den Einsatz über das Internet erschweren. Native Methoden sind Methoden, die in einer anderen Sprache beginnen und so auf systemspezifische Funktionen zugreifen, die in der Java-API nicht vorhanden sind .[9] Diese Methode wird heutzutage dann verwendet, wenn die Datenbank nur ODBC-Treiber verwendet und keine JDBC-Treiber installiert hat.[10]

[9] http://pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzaha%2Fnmjni.htm
[10] http://www.itwissen.info/definition/lexikon/Applet-applet.html,
http://www.schuenemann.name/studium/sim3D/sim3d-kap2.html (jeweils am aufgerufen 07.05.2014)

4.2.2 Typ-2 Treiber

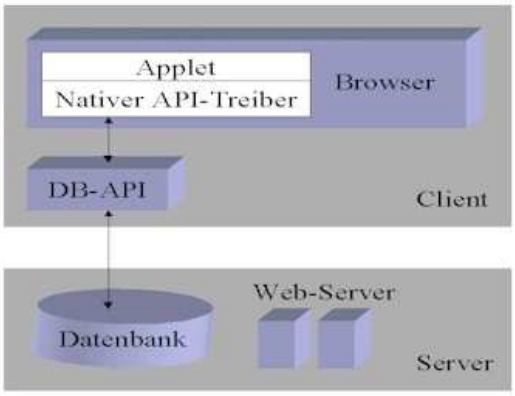

Abbildung 6: Typ -2 Treiber

Typ-2 Treiber übersetzen die JDBC-Aufrufe direkt in Aufrufe der Datenbank-API. Wie Typ-1 enthält auch Typ-2 native Methoden. Somit ist der Client abermals aufgefordert, die zusätzlichen Treiber auf dem System zu installieren. Da die Treiber von den Typen 1 und 2 auf native Methoden zurückgreifen müssen, sind diese nicht portabel, da sie auf plattformspezifische Zugriffsmöglichkeiten für die Datenbank angewiesen sind. Der Nachteil ist, dass Applets (ein Computerprogramm, das nicht als eigenständige Applikation betrieben wird, siehe Widget)[11] mit diesen Treibern nicht umgehen können. Einem Applet ist es nicht erlaubt, auf nativen Code zuzugreifen und kann deshalb keine Verbindung zu einer Datenbank herstellen.

[11] http://www.itwissen.info/definition/lexikon/Applet-applet.html,
http://www.schuenemann.name/studium/sim3D/sim3d-kap2.html (jeweils am aufgerufen 07.05.2014)

4.2.3 Typ-3 Treiber

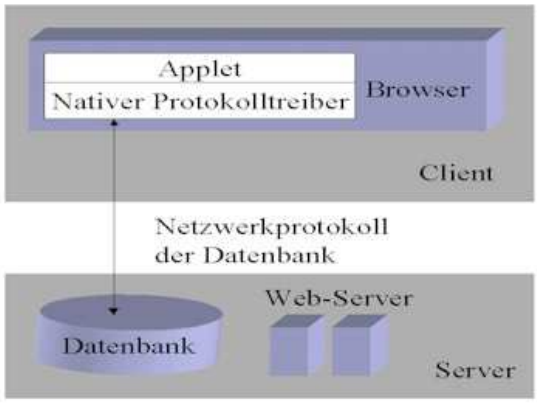

Abbildung 7: Typ -3 Treiber

Mittels des Typ-3 Treibers werden die JDBC-API-Befehle in ein generisches Datenbankmanagementsystem (DBMS) übersetzt und an einen Middleware-Treiber auf einen Anwendungsserver übertragen. Erst der Anwendungsserver transformiert die Befehle für den jeweiligen Datenbankserver um und leitet diese weiter. Typ-3 eignet sich im Zusammenhang mit Firewalls sehr gut für Internet-Protokolle. Das besondere an Typ-3 ist, dass der Client keine spezifische Bibliothek installieren muss und er keine Informationen über den Datenbankserver benötigt.[12]

4.2.4 Typ-4 Treiber

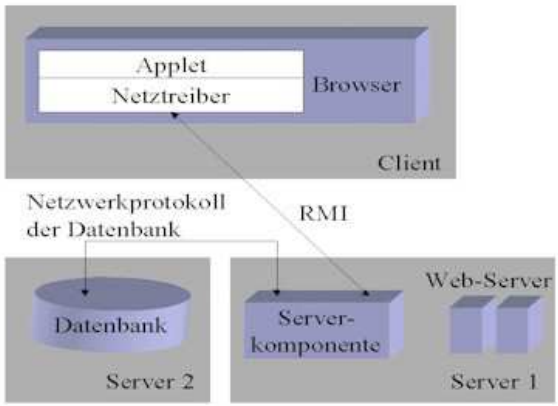

Abbildung 8: Typ -4 Treiber[13]

[12] openbook.galileocomputing.de/javainsel9/javainsel_24_003.htm, http://www.schuenemann.name/studium/sim3D/sim3d-kap2.html (jeweils am aufgerufen 07.05.2014)
[13] Abbildung 5-8 http://www.schuenemann.name/studium/sim3D/sim3d-kap2.html#Kapitel21

Der Typ-4 Treiber ist vollständig in Java programmiert und kommuniziert auf direktem Wege mit dem Datenbankserver. Hier werden die JDBC-API Befehle direkt an das DMBS gesendet und anschließend für den jeweilige Datenbankserver übersetzt und übertragen. Durch die direkte Verbindung mit der Datenbank ist Typ-4 schneller als Typ-3. Dafür ist Typ-4 weniger flexibel. Da der Server die Befehle übersetzt, muss auch hier keine Bibliothek vorhanden sein. Wegen der schnellen Verbindung eignet sich Typ-4 gut für das Intranet.[14]

4.3 Verbindungsablauf

In diesem Abschnitt werden die Schritte eines Verbindungsablaufs erläutert. Für die Verbindung bietet JDBC-API verschiedene Interfaces und Klassen, die in den Packages java.sql und javax.sql definiert sind. Die folgende Übersicht liefert eine stichpunktartige Übersicht der einzelnen Schritte:

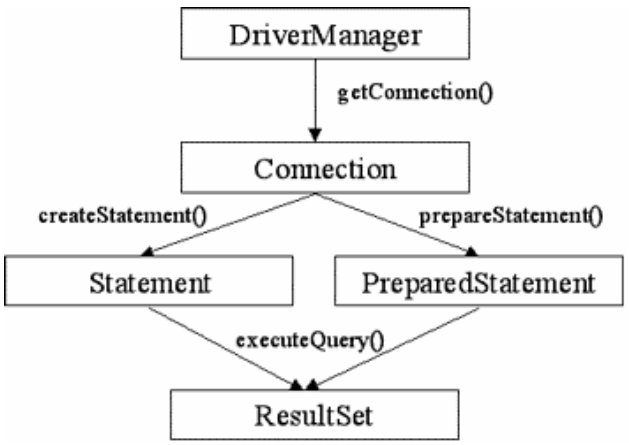

Abbildung 9: Verbindungsablauf

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

[14] openbook.galileocomputing.de/javainsel9/javainsel_24_003.htm, http://www.schuenemann.name/studium/sim3D/sim3d-kap2.html (jeweils am aufgerufen 07.05.2014)

Schritt 6: SQL-Anweisungsobjekt schließen
Schritt 7: Datenbankverbindung schließen

Schritt 7 ist besonders wichtig, da Datenbankverbindungen relativ schwergewichtig sind und einige Systemressourcen verbrauchen. Aus diesem Grund, sollte man zur Ausführung von SQL-Anweisungen, nicht jedes Mal wieder eine eigene Datenbankverbindung aufbauen, sondern, wenn möglich, eine bereits bestehende Verbindung nutzen. Daher werden die Schritte 4 und 5 in der Regel mehrfach durchlaufen.[15]
Damit nicht jedes mal eine aufwendige Verbindung zur Datenbank hergestellt werden muss und diese wegen z.B. nur einer Abfrage gleich wieder geschlossen wird, soll ein Connection-Pool Abhilfe schaffen.
Der Connection-Pool öffnet einige Verbindungen zur Datenbank und verwaltet die Verbindungen zentral. Benötigt z.B. ein Programm Zugriff auf die Datenbank, wird diese durch den Pool bereitgestellt und als besetzt markiert. Sobald das Programm fertig ist, wird die Verbindung wieder auf "verfügbar" gesetzt. Dadurch lässt sich der Overhead des Auf- und Abbaus von Verbindungen vermeiden. Gerade in größeren Anwendungen finden oftmals diverse Zugriffe durch verschiedene Benutzer in kurzem zeitlichem Abstand statt.
Connection Pooling musste früher selbst realisiert werden. Seit JDBC 2.0 wird es durch Klassen des JDKs unterstützt.[16]

Sehr wichtig ist noch zu erwähnen, dass alle Schritte in einem Try-Catch Block zu implementieren sind, um mögliche Fehler abzufangen und auszuwerten.

[15] Der Weg zum Java-Profi Seite 780
[16] Der Weg zum Java-Profi Seite 786

5 Datenbankzugriff mit JDBC

5.1 Verbindungsaufbau

Der folgende Ablauf bleibt in der Regel immer gleich. Vor der Einführung von JDBC 4 musste man als erstes den Datenbanktreiber laden. Dieser Vorgang ist jetzt nicht mehr notwendig, wurde aber der Vollständigkeit halber gelassen. Diese Aufgabe übernimmt jetzt der DriverManager. Als nächstes baut man mit Hilfe des DriverManagers eine Verbindung zur Datenbank auf und führt die Methode "getConnection()" mit den erforderlichen Parametern aus. Sobald eine Verbindung zur Datenbank hergestellt ist, kann man die gewünschten SQL-Befehle an die Datenbank senden. Wichtig zu erwähnen ist, dass verschiedene Datenbanken verschiedene URL Formate benutzen.

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

Tabelle 1: Datenbank-URLs[17]

[17] Der Weg zum Java-Profi Seite 781

```
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);
        }
    }
```

Listing 1: Verbindungsaufbau zur Datenbank

Da man in Listing 1 einen MySQL-Server nutzt, benötigt man den MySQL URL-Typen verwenden.

5.2 Übermitteln von Abfragen

Eine Möglichkeit für das übermitteln von Abfragen ist das Interface "Statement". Man schickt mit der Methode "executeQuery()" eine gewöhnliche Abfrage als String, wie z.B. "SELECT * FROM Filme WHERE Title = 'K-Pax' ", und erhält den gewünschten Wert bzw. Datensatz.

```
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);
}
```

Listing 2: Übermittlung der Abfrage

Wie Listing 2 zeigt, wird statt einem Ist-Gleich-Zeichen ein "like ' %" title+"%' " verwendet. Das Prozentzeichen sorgt dafür, dass die Datenbank alle Werte, die den übergebenen String enthalten, ausgibt.
Wenn man allerdings die Datenbank verändern möchte, muss man die Methode "executeUpdate()" verwenden. Anders als in Listing 2 dargestellt, kann das erhaltene Ergebnis auch entsprechende Objekte speichern und wiedergeben. [18]

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

Listing 3: Update-Methode

Allerdings besitzt das Interface "Statement" ein paar Schwachstellen. Die erste ist erstmal, wenn wir eine Liste mit Objekten haben und die gleiche Abfrage immer und immer wiederholen wird die gesamte Abfrage jedes Mal an die Datenbank übermittelt und diese muss den String von neuem auswertet und ausführt. Dies kann sich negativ auf die Performance ausüben.
Das nächste Problem ist die Syntax von SQL, diese erkennt die Apostroph als String-Vergleichsfunktion, was in Java bekanntlich die Anführungsstriche sind. Angenommen wir haben eine Tabelle Personen und wir wollen alle Personen mit dem Namen O'Connor einsehen, sähe die Abfrage als String wie folgt aus:

[18] http://clieber.de/index.php?id=4 (aufgerufen am 20.04.2014)

"SELECT * FROM Person WHERE Name = 'O'Connor' ".

Die Datenbank erkennt zwar den Befehl, hört aber nach dem O auf, da laut SQL-Syntax der String-Begrenzer aufhört. Das Programm würde eine "SQLException", mit dem Fehler "java.sql.SQLException: unexpected token: CONNOR", werfen.
Außerdem besteht die Gefahr von SQL-Injection. Mit SQL-Injection wird das Einschleusen potenziell schadhafter SQL-Anweisungen in die Datenbankkommandos einer Applikation genannt und stellt eine weit verbreitete Sicherheitslücke dar. Wenn man diese Gefahr jedoch kennt, kann man sich natürlich auch davor schützen. Um sich vor solchen Angriffen zu schützen sollte man konsequent mit "PreparedStatements" arbeiten.[19]

Um eine allgemeine Abfrage, die Parameter erwartet, zu erstellen wurde das Interface "PreparedStatement" eingeführt. Genau wie bei Statements wird als erstes ein String mit der SQL-Abfrage an die Datenbank gesendet, welcher wie folgt aussieht: "SELECT * FROM Person WHERE = ?". Allerdings verarbeitet die Datenbank den eingehenden String ein wenig anders. Wie einem gleich auffällt enthält die WHERE-Bedingung ein Fragezeichen, welche als Platzhalter für eingehende Parameter dient. Nun hat die Datenbank quasi eine Vorlage, die sie solange das PreparedStatement nicht geschlossen wurde, beliebig oft aufrufen kann und sie die Anweisung nicht immer neu auswerten muss sondern einfach nur die Parameter einsetzen muss.

```
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();
```

Listing 4: Update mit PreparedStatement

Man kann also sagen dass die "Statement"-Möglichkeit für einzelne primitive Anwendungen und Abfragen, welche nur einmalig vorkommen und sich nicht

[19] Der Weg zum Java-Profi Seite 784ff

wiederholen. Das Interface "PreparedStatement" ist eher für größere Anwendungen, die eine Mehrfachausführung der gleichen SQL-Anweisungen nutzen, besser geeignet. Man kann natürlich auch nur die "PreparedStatement"-Methode aus Listing 4 nutzen, da sie keine erkennbaren Nachteile aufweist und nur von der Implementierung ein wenig abweicht.

5.3 Ergebnisverwaltung

Nachdem wir die SQL-Anweisungen an die Datenbank übermittelt haben, erhalten wir natürlich, sofern keine Exception geworfen wurde, ein Ergebnis welches erstmal als ResultSet-Objekt gespeichert wird. Dieses kann mit Hilfe einer While-Schleife (siehe Listing 2) ausgewertet werden. Bevor man aber die Daten verarbeiten kann, muss die Methode next() mindestens einmal aufgerufen werden, egal wie viele Datensätze das Result enthält. Hier stellt sich nun die Frage "Warum muss man immer zuerst die Methode next() aufrufen?". Die Antwort ist ganz einfach. Ein ResultSet enthält einen Datensatzzeiger (auch Cursor genant) welcher den ausgewählten Datensatz referenziert. Da der Zeiger immer vor den Datensätzen steht, muss dieser einmal bewegt werden. Das hat zur Folge dass der Index nicht bei 0 anfängt, sondern ungewohnter weise bei 1.
Das besondere an ResultSet ist nicht das speichern der Datensätze, sondern eher das man die gespeicherten Daten modifizieren kann.[20]

5.3.1 Konfiguration eines ResultSet

Standardmäßig erfolgt die Verarbeitung in sukzessiver Vorwärtsrichtung und kann nicht bearbeitet werden. Die Einstellungen erfolgen nicht über das ResultSet sondern werden im Statement festgelegt, welches wiederum das ResultSet erzeugt:

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

Listing 5: ResultSet einstellen

[20] Der Weg zum Java-Profi Seite 792

Durch setzen einer entsprechenden Konstante für "resultSetType" wird einem nicht nur next() angeboten, sondern auch eine freie Navigierbarkeit. Dadurch können Datensätze direkt angesprungen werden, aber auch eine rückwärts Navigation wird angeboten.
"resultSetConcurrency" ermöglicht einem das verarbeiten der Datensätze. Hier werden einem updateXXX()- und getXXX()-Methoden angeboten, welche immer aufrufbar sind.

Was am ResultSet besonders interessant ist, ist die Sensitivität für Änderungen. Während wir die erhaltenen Daten verarbeiten, können andere Benutzer auf die Datenbank zugreifen und Veränderungen vornehmen, die auch unsere Daten betreffen. Also kann es passieren, dass wir in der Zeit wo wir die Daten verarbeiten sich diese ändern könnten und wir so mit alten Datensätzen arbeiten. Wenn das ResultSet sensitiv ist, kann es passieren dass die getXXX()-Methoden andere Werte liefern, verglichen mit den Werten zum Konstruktionszeitpunkt, da sie jegliche Änderungen widerspiegelt.
Ist das ResultSet allerdings nicht sensitiv, so bekommt man keine Änderungen mit und arbeitet nur mit den Daten die während des Konstruktionszeitpunktes erstellt wurden. Möglicherweise erhält man veraltete Werte.[21]

Für die Wahl der Einstellungen stellt ResultSet vorgefertigte Methoden bereit. Für Parameter "resultTypeSet" sind folgende Werte definiert:

- ResultSet.TYPE_FORWARD_ONLY: Standardeinstellung und erlaubt nur die Vorwärtsrichtung. Jeder Datensatz kann genau einmal verarbeitet werden.
- ResultSet.TYPE_SCROLL_INSENSITIVE: Erlaubt einem beliebige Datensätze anzuspringen, bei Bedarf auch mehrfach. Wenn in der Zwischenzeit in der Datenbank Datensätze geändert wurden, werden diese nicht im ResultSet aktualisiert.
- ResultSet.TYPE_SCROLL_SENSITIVE: Bietet ebenfalls eine dynamische Einsicht in die Daten, jedoch werden mögliche Änderungen übernommen.

[21] Der Weg zum Java-Profi Seite 793

Für den Parameter "resultSetConcurrency" sind folgende Werte zulässig:

- Resulst.Set.CONCUR_READ_ONLY: Standardeinstellung und erlaubt dem ResultSet keine Änderungen auf die Datenbank.
- Resulst.Set.CONCUR_UPDATABLE: Dem ResultSet wird erlaubt Änderungen zu übernehmen, jedoch mit gewissen Einschränkungen. Sobald die Abfrage Joins oder Aggregats- bzw. Gruppierfunktionen nutzt können die Datensätze nicht mehr zugeordnet werden. Der Grund dafür ist, dass die Ergebnismenge nicht 1:1 auf die Datensätze der Datenbank abbildbar ist.[22]

5.3.2 Änderung am ResultSet

In diesem Abschnitt gehen wir davon aus, dass das ResultSet so eingestellt ist, dass man die Datensätze ändern, welche auch in der Datenbank geändert werden, und zu jeder belieben Position springen können.

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

Listing 6: Aktualisierung eines ResultSet

Mit der updateString() Methode lässt sich im momentan angezeigtem Datensatz in der angegebenen Spalte, den Wert ändern. Diese Änderung ist erstmals nur im ResultSet, kann aber mit der Methode updateRow() in die Datenbank übernommen werden.
In der Praxis ist die Verarbeitung der Daten, zumindest meistens, wesentlich komplexer und kann zu Fehlern führen oder man die geänderten Daten doch nicht in die Datenbank zu übernehmen. Hierfür hat ResultSet eine Methode, welche die vorgenommenen Änderungen wieder rückgängig macht und den Datenbanksatz wieder in den Ursprungszustand zurücksetzt: cancelRowUpdate().

[22] Der Weg zum Java-Profi Seite 793ff

Da wir in der Regel nicht die einzigen sind, die Zugriff auf die Datenbank haben und andere Benutzer in der Zwischenzeit auch Änderungen vornehmen könnten, gibt es die Methode refreshRow(). Wie der Name bereits vermuten lässt, aktualisiert das ResultSet und holt sich erneut die Daten aus der Datenbank. [23]
Neben Änderungen einzelner Werte im ResultSet, sind auch andere Möglichkeiten offen. Wie z.B. das hinzufügen (insertRow()) oder auch das löschen (delteRow()) von Datensätzen.[24]

5.4 Schließen der Verbindungen

Um den Datenbankserver zu entlasten und Ressourcen für andere Benutzer freizugeben muss jede erstellte Verbindung auch wieder geschlossen werden. Mithilfe der Methode close() lässt sich dies schnell umsetzen. Normalerweise wird jedes close() in einem eigenem try-catch-block abgearbeitet werden. Dies hat den Vorteil, dass wenn eine close() Methode eine Exception wirft die anderen trotzdem ausgeführt wird. Stellen wir uns vor wir hätten alle drei einem Block und die zweite Methode wirft eine Exception aus, würde die Aufwendige Verbindung zur Datenbank nicht geschlossen werden und würde unnötig Ressourcen belegen.[25]

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

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

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

Listing 7: Try-Catch Block für jede Verbindung

[23] Der Weg zum Java-Profi Seite 795ff
[24] Der Weg zum Java-Profi Seite 796
[25] Der Weg zum Java-Profi Seite 786

6 JDBC mit Spring Framework

Im letzten Abschnitt wird gezeigt, wie einfach der Zugriff auf JDBC-Abstraktionen mit dem Springframework gestaltet ist. Zunächst müssen wir die Bibliotheken mysql-connector-java.jar, org.springframework.jdbc.jar und org.springframework.transaction.jar[26] in das Projekt einbinden. Diese kann man kostenlos im Internet herunterladen.

Für diesen Abschnitt wurde ebenfalls ein kleines Programm geschrieben, welches Schülerdaten (Name und Alter) verwaltet. Die Verbindung zur Datenbank erfolgt wie folgt:

Als erstes erstellen wir eine xml-Datei mit dem Namen "Beans", diese enthält alle relevanten Daten um sich mit dem SQL Server zu verbinden. "Beans" wird von den Entwicklern standardmäßig gewählt. Hierfür werden weitere Bibliotheken benötigt: antlr-runtime-3.0.1, org.springframework.beans-3.1.0.M2, org.springframework.context-3.1.0.M2, org.springframework.core-3.1.0.M2[27]

```
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>
```

Listing 8: Beans.xml

[26] http://www.java2s.com/Code/Jar/o/Downloadorgspringframeworkjdbcjar.htm
[27] http://www.tutorialspoint.com/spring/spring_jdbc_example.htm (aufgerufen am 07.05.2014)

Das Hauptprogramm holt sich mit folgendem Befehl die Werte und stellt dann eine Verbindung her:

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

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

Listing 9: Verbindungsaufbau zur Datenbank

Einmal eine Vorlage kann diese mit ein paar Änderungen stets überall übernommen werden. Mit dieser Variante lässt sich schnell und einfach eine Verbindung herstellen, ohne sich über Exceptions Gedanken machen zu müssen.

Das Spring Framework bietet eine Vielzahl an vorgefertigten Klassen und Interfaces die die Arbeit mit JDBC erleichtern soll. Wie wir in Abschnitt 5 gelernt haben, müssen wir der Datenbank eine SQL Abfrage schicken, erhalten ein Ergebnis, speichern diese in einem ResultSet und werten sie anschließend aus. Außerdem müssen wir jede Abfrage in einem Try-Catch Block festhalten, was den Code in die Länge zieht.
Anders beim Spring Framework. Zuerst erstellen wir eine Hilfeklasse namens "StudentMapper" die wie folgt aussieht.[28]

```
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;
    }
}
```

Listing 10: Datensatz in ein Student-Objekt speichern

Diese soll uns beim speichern der Datensatzzeile in ein Student-Objekt helfen.
So nun wollen wir alle Schüler aus der Datenbank holen und diese in eine Liste mit Student-Objekt speichern. Nach Listing 2 müssten wir erstmal noch ein Statement erstellen bzw. verbinden, im ResultSet speichern und mit einer While-Schleife in das

[28] http://www.javabeat.net/introduction-to-spring-jdbc-framework/ (aufgerufen am 07.05.2014)

Objekt speichern. Nicht im Spring Framework, diese bietet eine Methode, die alle diese Aufgaben übernimmt und was noch wichtiger ist, nachdem wir das Ergebnis erhalten wird die Verbindung automatisch wieder getrennt, um Ressourcen für andere Benutzer wieder freizugeben. Das hat den Vorteil, dass man beim Programmieren sich keine Gedanken mehr machen muss, wann und ob die Verbindung getrennt werden sollte und dass man nicht den Try-Catch-Block vergisst.

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

Listing 11: Speichert die Datenbanktabelle

Auch das Aktualisieren oder das Abrufen einzelner Datensätze aus der Datenbank ist nun ein Kinderspiel:

```
public void create(String name, Integer age)
{
    String SQL = "insert into Student (name, age) values(?, ?)";

    jdbcTemplateObject.update(SQL, name, age);
    System.out.println("Created Record Name = " + name + " Age = "+ age);
}

public Student getStudent(Integer id)
{
    String SQL = "select * "
            + "from Student "
            + "where id = ?";
    Student student = jdbcTemplateObject.queryForObject(SQL, new Object[] {id}, new StudentMapper());
    return student;
}
```

Listing 12: Datenbank aktualisieren

Man kann wie beim PreparedStatement Fragezeichen als Platzhalter nutzen, nur muss man jetzt nicht mehr jeden Platzhalter einzeln ansprechen und den Update-Befehl starten, es erfolgt nun alles in einer Zeile.
Was die Sache auch angenehm macht, ist dass man das Objekt direkt speichern kann und keine While-Schleife hierfür benötigt.

Das Spring Framework ist eine super Erfindung welches einem das Leben sehr erleichtert, dennoch sollte man sich für ein besseres Verständnis die Grundzüge von JDBC aneignen und verstehen.

7 Fazit

Wie beim Mitkommilitonen Herr Topalovic war am Anfang die Freude groß, endlich wieder mit Java zu programmieren und für eine Zeit die Microsoft C-Sharp Welt zu vergessen. Zwar heißt es dass JPA mittlerweile Standard geworden ist, aber wie ich feststellen musste nutzen viele Firmen immer noch JDBC, unter anderem auch mein Arbeitgeber, die Firma Adesso.
Zu beginn habe ich erstmal ein kleines Programm geschrieben, welches auf eine Datenbank zugreift, ihren Inhalt ausliest und verändert. Ich war stolz darauf und dachte mir dass es so passt und gut zu erlernen ist. Anfangs habe ich das Spring Framework skeptisch betrachtet, ich denke dass die Redensart "**Was der Bauer nicht kennt, frisst er nicht** " hier zutreffend ist, und dachte mir dass ich es nicht hernehmen werde. Falsch gedacht! Ich hätte nie gedacht, dass das Spring Framework so viele Erleichterungen mit sich bringt. Ich habe zwar nur an der Oberfläche gekratzt.
Durch das Fach Software Engineering, wo wir mit C-Sharp programmieren, hatte ich etwas Erfahrung mit dem verwalten und auslesen von Datenbankdaten sammeln können. Ich muss sagen dass mir die Variante von Java um einiges besser gefällt und wenn ich in Zukunft vor der Wahl stünde, würde ich Java JDBC mit dem Spring Framework benutzen.

8 Literaturverzeichnis

Michael Inden (2010). Der Weg zum Java-Profi - Konzepte und Techniken für die professionelle Java-Entwicklung, 2. Auflage. Heidelberg: dpunkt.verlag GmbH

Alfons Kemper, André Eickler (2013). Datenbanksystem - Eine Einführung, 9. Auflage. Oldenbourg: Wissenschaftsverlag GmbH

Christof Lieber (2005). Abgerufen am 20.04.2014 von
http://clieber.de/index.php?id=4

Anonymus (2014). Abgerufen am 22.04.2014 von
http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html
Christoph Bergmann (1998). Abgerufen am 05.05.2014 von
http://www.java.seite.net/jdbc/index.html

Anonymus (2014). Abgerufen am 07.05.2014 von
http://www.tutorialspoint.com/spring/spring_jdbc_example.htm

Anonymus (2013). Abgerufen am 07.05.2014 von
http://www.javabeat.net/introduction-to-spring-jdbc-framework/