Abbildung 1 HM Logo[1]

Seminararbeit

Integration von Datenbanken mit JDBC

Vorgelegt von: *Philip Thamm*
Semestergruppe: *IF6*

Eingereicht bei: *Dipl.-Inform. Michael Theis*
Abgabetermin: *15.05.2015*
Studienfach: *Aktuelle Technologien verteilter Java-Anwendungen*
Semester: *Sommersemester 2015*

[1] http://www.fiff.de/veranstaltungen/fiff-jahrestagungen/jt2011/HM_Deu_CMYK.jpg

Inhalt

Abbildungsverzeichnis: ... 3
Codeverzeichnis: ... 4
1. Einleitung ... 5
2. Die Architektur ... 5
2.1. Treibertypen ... 6
2.1.1. Typ 1 (JDBC-ODBC-Brücke) ... 6
2.1.2. Typ 2 (Native API Java Treiber) ... 7
2.1.3. Typ 3 (JDBC-Net-Treiber) ... 7
2.1.4. Typ 4 (Nativ-Protokoll-Treiber) ... 8
3. Programmierkonzept ... 8
3.1. Ein Client für eine MySQL Datenbank ... 9
Beispielcode: ... 9
Ausgabe ... 10
3.1.1. Schritt 1: ... 11
3.1.2. Schritt 2: ... 11
3.1.3. Schritt 3: ... 11
3.1.4. Schritt 4: ... 12
3.1.5. Schritt 5: ... 13
3.1.6. Schritt 6 ... 14
3.1.7. Schritt 7: ... 14
4. JDBC-Abstraktion von Spring ... 15
4.1. Beispiel: ... 16
4.1.1. Beans.xml ... 16
4.1.2. Actor.java ... 17
4.1.3. ActorMapper.java ... 17
4.1.4. JdbcTest.java ... 18
5. Einbindung von JDBC in Java EE Anwendungen ... 19
6. Fazit ... 22
Literaturverzeichnis ... 24

Abbildungsverzeichnis:

Abbildung 1 HM Logo 1
Abbildung 2 JDBC Architektur 5
Abbildung 3 Typ-1 Treiber 6
Abbildung 4 Typ-2 Treiber 7
Abbildung 5 Typ-3 Treiber 7
Abbildung 6 Typ-4 Treiber 8
Abbildung 7 Spring Zuständigkeiten 15
Abbildung 8 Wildfly DataSource Konfiguration 20

Codeverzeichnis:

Code 1 JDBC Beispiel 10
Code 2 Ausgabe JDBC Beispiel 10
Code 3 Spring Beans.xml 16
Code 4 Spring Actor.java 17
Code 5 Spring ActorMapper.java 18
Code 6 Spring JDBCTest.java 19
Code 7 Java EE DataSource Servlet 21

1. Einleitung

Das Zugreifen, Sammeln und Verwalten von Informationen ist im „Informationszeitalter“ eine der zentralen Säulen der Wirtschaft. Hierfür bietet die EDV Datenbankverwaltungssysteme (DBMS) an. Diese Systeme arbeiten auf einer Datenbasis. Die Programme, die die Datenbasis kontrollieren, besitzen einen großen Anteil der DBMS. Heutzutage sind die Datenmodelle durch relationale Modelle, kurz gesagt Tabellen, die miteinander in Beziehung stehen, dargestellt. Wenn man auf diese Tabellen zugreifen möchte, um damit die Datenbankausprägung zu erfahren, benötigt man Abfragemöglichkeiten. Java erlaubt mit JDBC den Zugriff auf relationale Datenbanken.[2]

JDBC (**J**ava **D**ata**b**ase **C**onnectivity) ist eine herstellerneutrale Programmierschnittstelle (Application Programming Interface, API). Sie erlaubt den Programmierern, Verbindung zu einer Datenbank herzustellen und mittels SQL die Datenbank abzufragen oder zu aktualisieren, ohne sich um die spezifischen Eigenheiten eines bestimmten Datenbanksystems zu kümmern.[3]
Die erste JDBC-Spezifikation gab es im Juni 1996. Die Schnittstellen und wenigen Klassen sind ab dem JDK 1.1 in Java integriert. Durch die JDBC API wird eine Abstraktion von relationalen Datenbanken erreicht. Um die Unterschiede kümmern sich sogenannte JDBC-Treiber, welche von den jeweiligen Datenbankherstellern bereitgestellt werden. Die JDBC-API ist vollständig in Java implementiert und ist somit auch plattformunabhängig.

2. Die Architektur

Die Architektur von JDBC besteht aus zwei Schichten. Die obere Schicht ist die eigentliche JDBC-API. Diese kommuniziert mit der unteren Schicht, der JDBC-Treibermanager-API und sendet dieser die verschiedenen SQL-Anweisungen. Der Treibermanager sollte mit den verschiedenen Treibern von Drittherstellern kommunizieren. Diese stellen die eigentliche Verbindung zur Datenbank her. Der Programmierer muss sich nicht um herstellerspezifische Eigenheiten von verschiedenen DBMS kümmern. Siehe Abbildung 2.

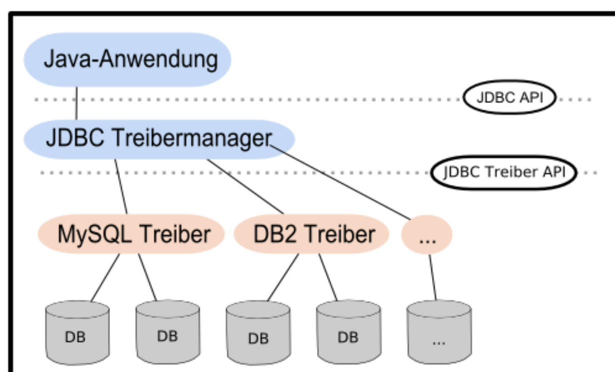

Abbildung 2 JDBC Architektur[4]

[2] Java SE 8 Standard Bibliothek
[3] http://www.inf.uni-konstanz.de/dbis/teaching/ss01/data-on-the-web/local/jdbcsqlj.pdf gesehen am 28.03.2015
[4] http://www.iwiki.de/wiki/index.php/JDBC

2.1. Treibertypen

Oracle definiert vier Treiberkategorien, die im Folgenden beschrieben werden. Die vier Treibertypen unterscheiden sich im Wesentlichen darin, ob sie über einen nativen Anteil verfügen oder nicht. Nativ bedeutet, dass die Aufrufe an eine datenbankspezifische Bibliothek weitergeleitet werden.

2.1.1. Typ 1 (JDBC-ODBC-Brücke)

Die ODBC-Schnittstelle (Open Database Connectivity Standard) ist ein Standard von Microsoft der den Zugriff auf Datenbanken ermöglicht. ODBC existiert weitestgehend nur in der Windows Welt. Da am Anfang der JDBC Entwicklung keine Treiber existierten, musste der Zugriff auf die Datenbank auf anderen Wegen umgesetzt werden. Es wurde eine Lösung mittels der JDBC-ODBC-Brücke gefunden. Die Aufrufe von JDBC werden in ODBC Aufrufe auf der Clientseite umgewandelt. Da die Performance nicht optimal war, stellte dies nur eine Notlösung dar. In Java 8 wurde diese Brücke entfernt und ist somit kein Bestandteil des JDK mehr. Allerdings ist dies auch keine Notwendigkeit mehr, da für jede bedeutende Datenbank ein direkter JDBC-Treiber existiert.

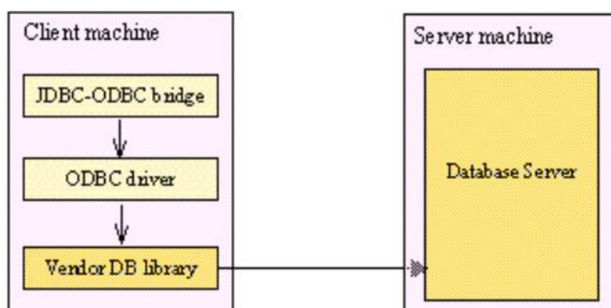

Abbildung 3 Typ-1 Treiber[5]

[5] http://www.javaworld.com/article/2076117/java-web-development/jdbc-drivers-in-the-wild.html

2.1.2. Typ 2 (Native API Java Treiber)

Ein Typ 2 Treiber übersetzt die JDBC-Aufrufe direkt in Aufrufe der Datenbank-API. Dazu enthält der Treiber einen Programmcode der typischerweise in C/C++ geschrieben ist und welcher die nativen Methoden aufruft. Treiber von Typ 2 sind nicht portabel, da sie für die ODBC Brücke auf die Plattformbibliothek (installierte Bibliothek) für ODBC zurückgreifen müssen. Einen Treiber auf ein anderes System zu portieren, funktioniert in der Regel nicht, da er fest mit der Datenbank verbunden ist. Möchte man z.B. einen Treiber auf ein Smartphone mit einem ARM Prozessor übertragen, würde es hier schon Probleme wegen der Hardwarearchitektur (x86 zu ARM) geben.

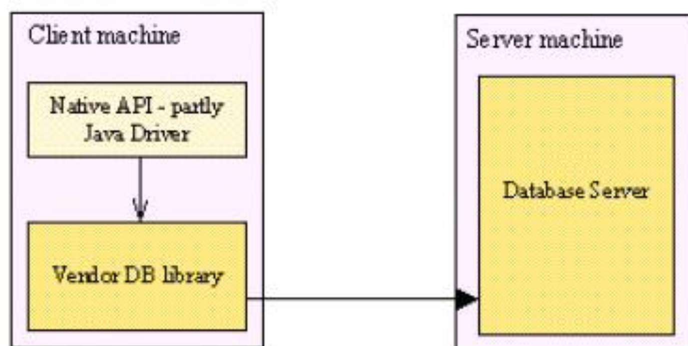

Abbildung 4 Typ-2 Treiber[6]

2.1.3. Typ 3 (JDBC-Net-Treiber)

Ein JDBC Treiber ist ein universeller in Java programmierter Treiber, der beim Datenbankzugriff auf den Client geladen wird. Dieser Treiber kommuniziert mit der Datenbank über eine Zwischenschicht. Typ 3 Treiber übernehmen somit eine Vermittlerrolle, denn erst der Middleware-Server leitet Anweisungen für die Datenbank an diese weiter. Ein Typ 3 Treiber ist vor allem für Applets und Internetdienste geeignet, da seine Klassendateien kleiner als bei einem Typ 4 Treiber sind. Sie können somit den Traffic einschränken. Über ein spezielles Protokoll zum Middleware-Server kann auch eine verschlüsselte Verbindung eingesetzt werden. Da zudem das Middleware-Protokoll unabhängig von der Datenbank ist, muss auf Clientseite für den Zugriff auf verschiedene Datenbanken nicht mehr alle Treibertypen installiert werden.

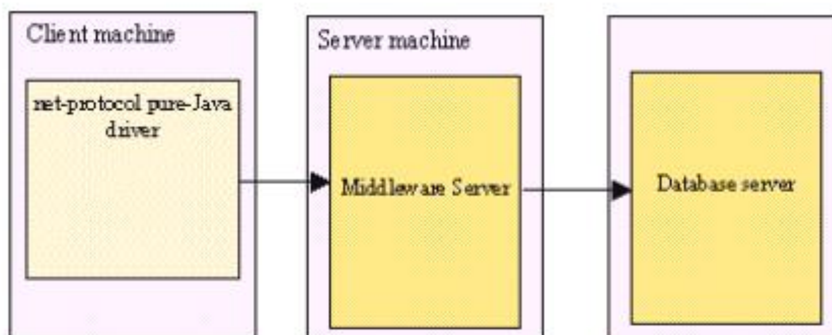

Abbildung 5 Typ-3 Treiber[7]

[6] http://www.javaworld.com/article/2076117/java-web-development/jdbc-drivers-in-the-wild.html

2.1.4. Typ 4 (Nativ-Protokoll-Treiber)

Typ 4 Treiber sind vollständig in Java programmiert. Diese Treiber kommunizieren direkt mit dem Datenbankserver. Dies geschieht mithilfe des datenbankspezifischen Protokolls über einen offenen Port. Dadurch sind diese Treiber im Gegenteil zu Java-Net-Treibern datenbankspezifisch. Da dies eine Direktverbindung zur Datenbank ist, ist es zugleich die performanteste Lösung.

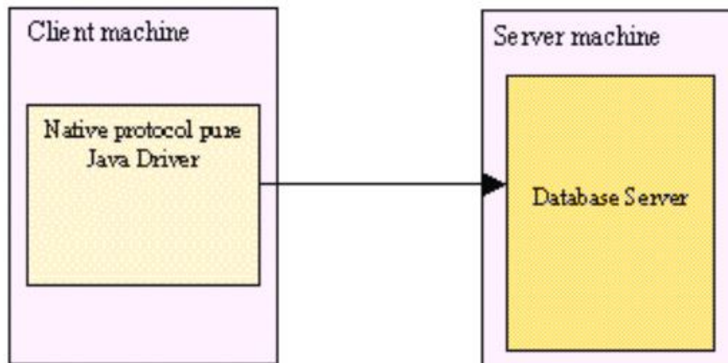

Abbildung 6 Typ-4 Treiber[8]

Die meisten Datenbankhersteller stellen ihren Datenbanken entweder einen Typ 3 oder Typ 4 Treiber zur Verfügung. Typ 1 und Typ 2 Treiber werden oft für das Testen verwendet oder in Unternehmensnetzwerken, in denen die zusätzliche Installation der notwendigen Bibliotheken kein großes Problem sein sollte. Häufig werden Typ 1 und 2 Treiber auch dort eingesetzt, wo ein höherwertiger Treiber noch nicht fertig gestellt ist.
Nach aktuellem Stand sind in Java 7 die JDBC Version 4.1 sowie in Java 8 die JDBC Version 4.2 im Einsatz. In Java 7 wurde das „try with resources" implementiert. In Java 8 waren die Änderungen nur minimal und nicht erwähnenswert.
Der Aufbau der JDBC-Treiber ist an die JDBC-Treiber-API spezifiziert. Diese API ist daher nur für die Hersteller von Datenbanken und deren Treibern von Bedeutung. Anwendungsentwickler arbeiten hauptsächlich mit der JDBC-API und sind somit unabhängig von der eigentlichen Implementierung der Treiber.

3. Programmierkonzept

Die für die JDBC-Programmierung verwendeten Klassen sind im Package **java.sql** enthalten.

Schritte die für einen Zugriff auf eine relationale Datenbank notwendig sind.

1. Einbinden der JDBC-Datenbanktreiber in den Klassenpfad
2. Unter Umständen Anmelden der Treiberklassen
3. Verbindung zur Datenbank aufbauen
4. Eine SQL Anweisung erzeugen
5. Die SQL Anweisung ausführen
6. Das Ergebnis/die Ergebnismenge der Anweisung holen und verarbeiten
7. Die Datenbankverbindung schließen

[7] http://www.javaworld.com/article/2076117/java-web-development/jdbc-drivers-in-the-wild.html
[8] http://www.javaworld.com/article/2076117/java-web-development/jdbc-drivers-in-the-wild.html

3.1. Ein Client für eine MySQL Datenbank.

Ein Beispiel soll das Programmierkonzept für JDBC veranschaulichen bevor das Programm im Einzelnen seziert wird. Das Programm MySQLAccess nutzt eine MySQL Datenbank um Daten abzurufen. Die Testdatenbank „sakila“ von MySQL kann unter folgendem Link heruntergeladen werden. http://dev.mysql.com/doc/index-other.html

Beispielcode:

```
//STEP 1 : Einbinden der JDBC API
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class MySQLAccess {
	// Database and credentials
	static final String DB_URL = "jdbc:mysql://localhost/sakila";
	static final String USER = "root";
	static final String PASS = "";

	public static void main(String[] args) throws ClassNotFoundException {
		// Step 2 : Anmelden der Treiberklasse
		try {
			Class.forName("com.mysql.jdbc.Driver");
		} catch (ClassNotFoundException e){
			e.printStackTrace();
		}

		Connection conn = null;
		Statement stmt = null;
		ResultSet rs = null;
		try {
			// STEP 3 : Open a connection
			System.out.println("Connecting to database...");
			conn = DriverManager.getConnection(DB_URL, USER, PASS);

			// STEP 4 : Eine SQL Anweisung erzeugen
			System.out.println("Creating statement...");

			String sql;
			sql = "SELECT actor_id, last_name, last_update from actor ORDER
BY last_name";

			// STEP 5 : Die SQL Anweisung ausführen
			stmt = conn.createStatement();
			rs = stmt.executeQuery(sql);

			// STEP 6 : Das Ergebnis holen und Verarbeiten
			while (rs.next()) {
				// Retrieve by column name
				int id = rs.getInt("actor_id");
				String Name = rs.getString("last_name");
				Date dob = rs.getDate("last_update");
```

```
                // Display values
                System.out.print("actor_id: " + id);
                System.out.print(", last_name: " + Name);
                System.out.print(", last_update: " + dob + "\n");
            }
            // STEP 7 : Verbindungen auflösen und aufräumen
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException se) {
            // Handle errors for JDBC
            se.printStackTrace();
        } // end try
        // Im Fehlerfall alles abbauen
        finally {
            if (rs != null)
                try {
                    rs.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            if (stmt != null)
                try {
                    stmt.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            if (conn != null)
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }

        }
        System.out.println("Goodbye!");
    }
}
```

Code 1 JDBC Beispiel

Ausgabe

```
Connecting to database...
Creating statement...
actor_id: 58, last_name: AKROYD, last_update: 2006-02-15
actor_id: 158, last_name: BASINGER, last_update: 2006-02-15
actor_id: 124, last_name: BENING, last_update: 2006-02-15
Goodbye!
```

Code 2 Ausgabe JDBC Beispiel

Im Package **java.sql.*** sind wichtige Klassen enthalten, welche in den folgenden Schritten erläutert werden.

Connection : Verbindung zu einer Datenbank
Statement : Abarbeiten von SQL Anweisungen
PreparedStatement : vorkompilierte SQL Anweisung
CallableStatement : vorkompilierte Stored-Procedure Anweisung
ResultSet : Abfragen von Ergebnissen

Bevor losgelegt werden kann, muss sich die JDBC-Treiber-Jar der verwendeten Datenbank im Klassenpfad der JVM befinden. Die Jar muss natürlich auch als Bibliothek dem Projekt hinzugefügt werden.

3.1.1. Schritt 1:

Die JDBC API von Java kann mittels eines Imports in den Quelltext eingebunden werden. Hier können gezielt einzelne Klassen oder das gesamte Package eingebunden werden.

```
import java.sql.*;
```

3.1.2. Schritt 2:

Das Anmelden der Treiberklasse geschieht seit Java 6 (JDBC 4.0) soweit vom Treiberproduzenten vorbereitet automatisch. Der Programmierer muss somit den Namen der Treiberklasse nicht mehr kennen. Für das manuelle Laden der Treiberklasse wird folgende Zeile im Code angegeben. Da die Klasse nur geladen werden muss, wird lediglich ein Aufruf benötigt und nicht der return-Wert. `Class.forName("com.mysql.jdbc.Driver");`

3.1.3. Schritt 3:

Nach dem erfolgreichen Laden eines Treibers kann eine Verbindung mittels eines **Connection**-Objektes hergestellt werden. Dies geschieht über eine Factory Methode der Klasse **DriverManager**. Dem Methodenaufruf wird eine Datenbank-URL und optional ein Benutzername sowie ein Passwort übergeben. Als erfolgreichen Methodenaufruf wird ein Connection-Objekt zurück geliefert. Die **DriverManager** Klasse hält ebenfalls sämtliche Datenbanktreiber über die das System verfügt. Alle Methoden, die die Klasse zur Verfügung stellt, sind statische Methoden. Es können unter anderem alle angemeldeten Treiber abgefragt werden, bzw. ein neuer Treiber angemeldet werden. Sämtliche Methoden sind in der Oracle Dokumentation unter https://docs.oracle.com/javase/8/docs/api/java/sql/DriverManager.html zu finden.

```
Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
```

Wie wird ein passender Treiber gefunden?

Sämtliche Treiber verwaltet die Klasse **DriverManager** in einem privaten Objekt DriverInfo. Dieses Objekt enthält wiederum ein Driver-Objekt, ein SecurityContext-Objekt und den Klassennamen className.
Nach dem Aufruf der Methode getConnection() iteriert der DriverManager die Liste aller DriverInfo-Objekte und versucht sich über die connect() Methode am Datenbankserver anzumelden. Bemerkt der Treiber, dass der Aufruf fehl schlägt und Null zurück liefert,

versucht er das nächste DriverInfo-Objekt aufzurufen. Sollten alle DriverInfo-Objekte fehlschlagen und keine Verbindung zu Stande kommen, so wirft der Treiber eine Fehlermeldung (SQLException "No suitable driver", "08001") aus. Dies ist auch der Grund dafür, warum ab Java 6 kein explizierter Treiber mehr angemeldet werden muss. Es werden einfach alle Möglichkeiten durchgespielt.

3.1.4. Schritt 4:

Ist es gelungen ein Connection-Objekt zu erzeugen, können SQL Kommandos abgesetzt und an die Datenbank übergeben werden. Diese Anweisungen können einfach als String verpackt werden. Zu achten ist lediglich auf die korrekte Syntax der jeweiligen anzufragenden Datenbank. Es können sich minimale Unterschiede bei verschiedenen Datenbanktypen ergeben.

Für sämtliche Anfragen, die an die Datenbank gestellt werden, sind so genannte Statement Objekte von der Connection zu erfragen. Hier sind drei verschiedene Typen zu Unterscheiden.

- Anweisungen vom Typ **Statement**
- Anweisungen vom Typ **PreparedStatement**
- Anweisungen vom Typ **CallableStatement**

Statement:

Repräsentiert das normale Statement Interface. Was die Performance betrifft sollte immer auf Statements zurückgegriffen werden. Statements sollten nur verwendet werden, wenn bereits erkennbar ist, dass die Anweisung lediglich einmal durchgeführt wird. Im Gegenteil zu PreparedStatements ermöglicht das Statement keine parametrisierten SQL Anfragen. Falls Parameter übergeben werden sollen, muss der String dementsprechend zusammengebaut werden. Bei dieser Art des Zusammenbauens, können dem String auch weitere SQL Fragmente übergeben werden. Dies wird vor Ausführung nicht überprüft. Es kann somit zu SQL Injections kommen.[9]

```
sql = "SELECT last_name, last_update from actor where actor_id =“ + actorId +
„ ORDER BY last_name";
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

[9] http://crunchify.com/what-are-the-difference-between-jdbcs-statement-preparedstatement-and-callablestatement/

PreparedStatement:

PreparedStatment ist eine abgeleitete Klasse von Statement. In den meisten Fällen ist es effizienter im Hinblick auf mehrfache Ausführung der SQL-Anweisung. Die Anweisung wird kompiliert an das DBMS übergeben und es müssen im Nachhinein lediglich Werte über „setter“ eingefügt werden. Durch das Vorkompilieren wird die Anweisung nur ein einziges Mal erzeugt. Da beim Einfügen der Datentyp überprüft wird, sind keine SQL-Injections möglich.

```
PreparedStatement pstmt = con.prepareStatement("UPDATE PRODUCTS SET PRICE = ?
WHERE ID = ?");
pstmt.setFloat(1, 546.00f);
pstmt.setInt(2, 7889);
```

CallableStatement:

CallableStatement ist eine abgeleitete Klasse von PreparedStatement. Zusätzlich zu der parametrisierten SQL Anweisung aus den PreparedStatements kann ein Objekt vom Typ CallableStatement noch mit StoredProcedures arbeiten. Diese sind so etwas wie Methodenaufrufe, die der SQL Anweisung mit übergeben werden. CalculateStatistics ist im unten aufgeführtem Beispiel eine definierte Funktion im Datenbanksystem.

```
CallableStatement callableStatement = connection.prepareCall("{call
calculateStatistics(?, ?)}");
callableStatement.setString(1, "param1");
callableStatement.setInt   (2, 123);
```

3.1.5. Schritt 5:

Bei SQL-Queries muss zwischen Abfragen und anderen Anweisungen unterschieden werden. Andere Anweisungen könnten eine Datenmanipulation bzw. eine Einfüge-Operation sein. Bei reinen Abfragen (SELECT) an die Datenbank wird das Ergebnis in Form eines ResultSet zurück geliefert. Die Auswertung eines ResultSet kann erst beginnen, nachdem der Treiber die Informationen von der Datenbank bekommen hat. Der Zeiger ist zu Beginn auf eine Position vor der ersten Zeile gesetzt. Der erstmalige Aufruf der **next**() Methode setzt den Zeiger auf die erste Zeile der geladenen Ergebnisse. Dies wird normalerweise mittels einer while-Schleife umgesetzt.

```
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
// do something }
```

Für eine Datenmanipulation bzw. eine Einfüge-Operation muss die Methode **executeUpdate**(String) verwendet werden. Diese Methode liefert bei erfolgreicher Ausführung die Anzahl der veränderten Zeilen als „**Integer“** zurück. Falls keine Änderungen durchgeführt wurden, würde der Wert 0 zurück geliefert werden. Ein Aufruf könnte z.B. folgendermaßen lauten.

```
Statement stmt = conn.createStatement();
String cmd = "UPDATE Angestellte SET Gehalt = Gehalt * 1.1 WHERE Angehörigkeit >
3";
int rows = stmt.executeUpdate(cmd);
```

3.1.6. Schritt 6

Wie bereits erwähnt werden Abfrageergebnisse in Form eines ResultSet zurück geliefert. Ein ResultSet besitzt einen internen Zeiger, welcher einen Zugriff auf einzelne Zeilen ermöglicht, ähnlich einem **java.util.Iterator**. Mit verschiedenen Methoden können die unterschiedlichen Spalten und Zeilen ausgewertet werden. Um weitere Zeilen zu bekommen, muss der Zeiger mit der Methode **next()** eine Position weiter geschoben werden. Falls alle Zeilen verbraucht sind, liefert die Methode next() den Wert false zurück.
Für den Zugriff auf Inhalte verschiedener Spalten gibt es verschiedene Methoden. Hier ist auf den Datentyp zu achten. Für jeden Datentyp gibt es eine eigene Methode. Jede Methode existiert in zwei verschiedenen Varianten. Ein Aufruf mit einem numerischen Parameter und ein Aufruf mit einer Zeichenkette. Der numerische Aufruf bezieht sich auf die Spalte mit dieser Nummer. Der Aufruf mit einer Zeichenkette bezieht sich auf die Spalte, welchen diesen Namen besitzt.
Besonders hervorzuheben ist die Methode resultset.getString(), welche auf sämtliche Datentypen angewendet werden kann, ohne eine Typ-Konvertierung beachten zu müssen.

```
// Retrieve by column name
int id = rs.getInt("actor_id");
String Name = rs.getString("last_name");
Date dob = rs.getDate("last_update");
```

SQL und Java Datentypen sind nicht identisch. Bei den get-Methoden sollte auf eine sinnvolle Typumwandlung geachtet werden. Hervorzuheben sind besondert die Typen DATE und TIME. Diese werden nicht wie gewohnt in java.util.Date bzw. java.Time umgewandelt, sondern in Typen des packages java.sql.Date bzw. java.sql.Time.

3.1.7. Schritt 7:

Objekte die nach erfolgreicher Durchführung nicht mehr benötigt werden sollten mit der close() Methode für den Garbagecollector freigegeben werden. Ein ResultSet wird normalerweise nach dem erfolgreichen Durchlaufen der while-Schleife geschlossen und freigegeben. Statement Objekte werden üblicherweise am Ende der Methode geschlossen. Connection Objekte allerdings erst bei Beenden der Anwendung. Die Verbindung wird für die gesamte Benutzungsdauer für zusätzliche Anfragen an die Datenbank offen gehalten.

4. JDBC-Abstraktion von Spring

Auch wenn der Hauptanwendungsbereich des Spring-Frameworks in der Java EE Entwicklung liegt, so können doch bestimmte Bereiche in jeder anderen Anwendung genutzt werden.[10]
Ein solcher Bereich ist unter anderem JDBC in Spring. Im Java-JDBC sind vor allem das Exception-Handling sowie das Handling von Verbindungen, Statements und ResultSets sehr kompliziert. Genau solchen oder ähnlichen Problemstellungen möchte sich das Spring-Framework widmen. Anhand eines einheitlichen Exception-Modells ist es möglich zwischen verschiedenen Datenbanken auf einheitliche Fehlermeldungen zurück zugreifen, so dass aus Sicht der Anwendung z.B. das Erstellen einer Tabelle eine einheitliche Fehlermeldung generiert wird. Datenbankverbindungen sowie Abfrageergebnisse werden von Spring selbstständig geschlossen, sofern sie nicht mehr verwendet werden müssen. Es können somit keine Ressourcenlecke entstehen. Außerdem versucht Spring auch ein einheitliches Modell für den Zugriff auf die Persistenzen zu bieten. Das bedeutet, anhand einer Datenquelle (der so genannten DataSource) kann zwischen den verschiedenen Persistenztypen auf dieselbe Datenbankverbindung zurückgegriffen werden. Das Hauptkonzept von Spring in Verbindung mit JDBC ist das sogenannte **JDBCTemplate**. Es ist die Hauptklasse von Spring-JDBC. Die Klasse führt SQL Anweisungen aus, liest Ergebnisse ein, transformiert sie in Objekte und ummantelt SQL-Exceptions in DataAccessExceptions. **JDBCTemplate** wird dabei von diversen Hilfsklassen unterstützt. Allen voran steht dabei der RowMapper. Dieser ermöglicht es Resultate aus den Anfragen zu ermitteln, so dass die Spalten und deren Werte auf entsprechende Objekte abgebildet werden können.
Wie in allen diesen Dingen „**Spring**" dem Entwickler hilft die Komplexität in den Griff zu bekommen, wird im Folgenden gezeigt.

Die folgende Tabelle zeigt die Zuständigkeiten.

Task	Spring	You
Connection Management	√	
SQL		√
Statement Management	√	
ResultSet Management	√	
Row Data Retrieval		√
Parameter Declaration		√
Parameter Setting	√	
Transaction Management	√	
Exception Handling	√	

Abbildung 7 Spring Zuständigkeiten[11]

Wie deutlich zu erkennen ist, muss sich der Programmierer lediglich um die für ihn wichtigen Dinge kümmern. Punkte wie Exception Handling wird vom Spring-Framework übernommen.

[10] http://www.itblogging.de/java/spring/spring-mysql-jdbc-tutorial/
[11] http://cdn.oreillystatic.com/en/assets/1/event/12/SimpleJDBC%20Development%20with%20Spring%202_5%20and%20MySQL%20Presentation.pdf

4.1. Beispiel:

Allen voran steht das Bean Konfigurations-XML. Hier wird das **Bean** für die Datenbank festgelegt. Man könnte allerdings auch die DataSource direkt in der Java-Klasse erstellen. Der Quelltext hierfür ist in der JdbcTest.java weiter unten auskommentiert.

4.1.1. Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

      <context:component-scan base-package="jdbc"/>

   <!-- Initialization for data source -->
   <bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/sakila" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>
</beans>
```
Code 3 Spring Beans.xml

Verwendet werden hier die Klasse DriverManagerDataSource und deren zugehörige Zugangsdaten zur Datenbank mit den jeweiligen Attributen. MySql ist unter dem Port 3306 zu erreichen. Die Datenbank ist wie im vorherigen Beispiel wieder die MySql Demodatenbank sakila.

4.1.2. Actor.java

Da im Spring-Framework SQL-Select-Abfragen auf Objekte gemapt werden, wird hierfür eine Klasse Actor erstellt. Diese liefert Schauspielerobjekte, welche ganz einfache Attribute wie ID oder Nachnamen besitzen. Es wird für eine bessere Ausgabe im Beispiel die toString()-Methode überschrieben.

```
package jdbc;

import java.sql.Date;

// Ein Schauspieler
public class Actor {

	private int actorId;
	private String lastname;
	private Date lastUpdate;

	public Actor(int actorId, String lastname, Date lastUpdate) {
		this.actorId = actorId;
		this.lastname = lastname;
		this.lastUpdate = lastUpdate;
	}

	@Override
	public String toString() {
		return "Actor: " + actorId + ", lastname: "
				+ lastname + ", lastUpdate: " + lastUpdate;
	}
}
```

Code 4 Spring Actor.java

4.1.3. ActorMapper.java

Als nächstes wird eine Klasse benötigt, die Datensätze einzelner Abfragen auf einzelne Objekte mapt. Dieses Mapping geschieht im Hintergrund des JdbcTemplates. Hierzu gibt es ein Interface in Spring namens **RowMapper<T>.**
Die Methode **mapRow**() wird intern aufgerufen und liefert Stück für Stück alle Zeilen der SQL-Query. Es werden dann alle Attribute gemäße des Codes auf Objekte gemapt. Man erhält nach erfolgreichem Durchlaufen der Methode eine Liste aller erzeugten Objekte zur geliefert.

```
import java.sql.ResultSet;
import java.sql.SQLException;

import jdbc.Actor;

import org.springframework.jdbc.core.RowMapper;

// Implementiert das Interface RowMapper
public class ActorMapper implements RowMapper<Actor> {

	@Override
	// Erhält vom Spring-Framework einen Eintrag und mapt diesen auf ein Objekt
	public Actor mapRow(ResultSet rs, int rowNum) throws SQLException  {
```

```
            return new Actor(rs.getInt("actor_id"), rs.getString("last_name"),
                        rs.getDate("last_update"));
      }

}
```

Code 5 Spring ActorMapper.java

4.1.4. JdbcTest.java

In der Main Methode wird dann zum Schluss nur die Konfigurationsdatei inklusive des festgelegten Bean geladen. Die zugehörigen Einstellungen werden aus dem Bean ausgelesen, damit eine Verbindung zur Datenbank erzeugt werden kann. Es kann mittels dieses Kontextes eine DataSource erzeugt werden, welche wiederum an das JdbcTemplate übergeben wird. Mittels des erzeugten JdbcTemplates sowie einem RowMapper wird eine SQL Query abgesetzt und man erhält als Rückgabewert eine Liste von Objekten.
Diese Liste kann dann z.B. in einer foreach Schleife durchlaufen werden und ausgegeben werden.

```
import java.util.List;

import javax.sql.DataSource;

import jdbc.Actor;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcTest {

// Liefert die in der Beans.xml konfigurierte DataSource
private static DataSource getDataSource() {
      // Pfad zu der Konfigurationsdatei
      ApplicationContext ac = new ClassPathXmlApplicationContext("Beans.xml");
      // Bean mit der ID 'mysqlDataSource' (MySQL Konfiguration) laden
      DataSource dataSource = (DataSource) ac.getBean("dataSource");

      // Beispiel einer hardcodierten DataSource
      // DriverManagerDataSource dataSource2 = new DriverManagerDataSource();
      // dataSource2.setDriverClassName("com.mysql.jdbc.Driver");
      // dataSource2.setUrl("jdbc:mysql://localhost:3306/sakila");
      // dataSource2.setUsername("root");
      // dataSource2.setPassword("");

      return dataSource;
}

@SuppressWarnings({ "unchecked", "rawtypes" })
public static void main(String[] args) {
      // Liefert ein mit der vorkonfigurierten DataSource gefülltes
      // JDBCTemplate
      DataSource dataSource = getDataSource();
      JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

      // SQL Query
      String sql = "SELECT actor_id, last_name, last_update from actor ORDER BY 
last_name";
```

```
		List<Actor> actorList = jdbcTemplate.query(sql, new ActorMapper());

		// Ausgeben aller Schauspieler
		for (Actor actor : actorList) {
			System.out.println(actor);
		}

	}
}
```
Code 6 Spring JDBCTest.java

5. Einbindung von JDBC in Java EE Anwendungen

Um eine Datenbank aus einer Java EE Applikation ansprechen zu können, gibt es zu den obigen zwei Möglichkeiten noch eine Einfachere. Da man beim Entwickeln einer Java EE Applikation ohnehin einen Applikationsserver benötigt, um die Anwendung im späteren Verlauf auf diesem Applikationsserver zu deployen, kann eine Datenquelle (**DataSource**) im Applikationsserver eingerichtet werden. Hierzu muss im ersten Schritt der Datenbanktreiber im Server eingebunden werden und danach die Datenquelle für diesen Treiber eingerichtet werden. Im unten aufgeführten Beispiel wird der Typ 4 Treiber von MYSQL genommen und dem Wildfly Applikationsserver übergeben.
Im Anschluss kann der Treiber dann über eine **JNDI** (**J**ava **N**aming and **D**irectory **I**nterface) API in der Java Applikation aufgerufen werden.

Schritt 1 – Den Treiber am Applikationsserver einrichten.
Hier exemplarisch dargestellt für das Wildfly CLI Eingabefenster. Zu finden unter C:\{Wildflyverzeichnis}\wildfly-8.2.0.Final\bin\jboss-cli.bat.
Nach starten der Batchdatei können die folgenden Kommandozeilen-Befehle abgesetzt werden.

Installieren des MySql Treibers:
Der erste Schritt besteht darin den MySQL Treiber auf dem Server zu installieren. Hierzu muss der Treiber sich lokal auf dem PC in einem Verzeichnis befinden.

```
deploy ~/Lokales Verzeichnis/mysql-connector-java-5.1.35-bin.jar
```

Um zu überprüfen, ob der Treiber sachgemäß installiert wurde, wird folgendes Kommando verwendet.

```
/subsystem=datasources:installed-drivers-list
```

Eine Ausgabe könnte folgendermaßen aussehen:

```
"driver-name" => "mysql-connector-java-5.1.35-bin.jar_com.mysql.jdbc.Driver_5_1",
"deployment-name" => "mysql-connector-java-5.1.35-
bin.jar_com.mysql.jdbc.Driver_5_1",
```

Danach muss die so genannte DataSource erzeugt werden. Um alle bereits vorhandenen Datenquellen anzuzeigen, kann folgendes Kommando verwendet werden.

```
/subsystem=datasources:read-resource(recursive=true)
```

Das Erzeugen einer neuen Datenquelle hier exemplarisch für meine Datenbank wird folgendermaßen abgesetzt.

```
data-source add --name=fwpFach --jndi-name=java:/jdbc/fwpFach --driver-name=mysql-connector-java-5.1.35-bin.jar_com.mysql.jdbc.Driver_5_1 --connection-url=jdbc:mysql://127.0.0.1:3306/sakila2 --user-name=root --password=
```

Im Anschluss kann die Datenverbindung überprüft werden.

```
/subsystem=datasources/data-source=fwpFach:test-connection-in-pool
```

Die Antwort sollte so lauten und ein erfolgreiches verbinden bestätigen. .

```
{
    "outcome" => "success",
    "result" => [true]
}
```
[12]

Dies kann natürlich auch über das Webinterface des Servers direkt geschehen. Zu sehen im folgendem Screenshot.

Abbildung 8 Wildfly DataSource Konfiguration

[12] http://blog.squins.com/2014/12/adding-a-mysql-jndi-datasource-to-jboss-wildfly-via-cli/

Als Codebeispiel wurde ein Servlet (Dynamisches Web Projekt in Eclipse) verwendet.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.*;
import javax.annotation.Resource;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class SQLAbfrage
 */
@WebServlet("/SQLAbfrage")
public class SQLAbfrage extends HttpServlet {
	private static final long serialVersionUID = 1L;

	// DataSource für den SQL Zugriff über eine Annotation injecten
	@Resource(lookup = "java:/jdbc/fwpFach")
	private DataSource dataSource;

	protected void doGet(HttpServletRequest request,
			HttpServletResponse response) throws ServletException,
IOException {
		PrintWriter out = response.getWriter();
		out.println("<h1>Actor Ausgabe:</h1>");

		// Verwenden der SQL DataSource
		// Abhandlung wie Beispiel 1
		try (Connection connection = dataSource.getConnection();
				PreparedStatement preparedStatement = connection
						.prepareStatement("SELECT * FROM actor");
				ResultSet resultSet = preparedStatement.executeQuery();)
{
			while (resultSet.next()) {
				String lastName = resultSet.getString("last_name");
				String firstName = resultSet.getString("first_name");
				out.println("Actor: " + lastName + " " + firstName +
"<br>");
			}
		} catch (SQLException e) {
			throw new IllegalStateException("Failed to fetch Actors", e);
		}

	}

}
```

Code 7 Java EE DataSource Servlet

Zu sehen ist, dass lediglich über die Annotation **@Resource** ein **DataSource Bean** injected wird.

```
@Resource(lookup = "java:/jdbc/name=fwpFach")
private DataSource dataSource;
```

Die Infos bezüglich Datenbank sowie Benutzername und Passwort werden dynamisch zur Laufzeit vom Server benutzt. Dieser verwaltet sämtliche Datenzugriffe für z.B. weitere Datenbanken oder andere Benutzerinformationen.

Das **DataSource-Objekt** kann ganz normal wie aus Beispiel 1 bereits bekannt verwendet werden. Es wird eine Connection erfragt und mit dieser Verbindung ein SQL-Statement an den SQL Server übergeben. Die Antwort kann wieder in einem ResultSet verarbeitet werden.

```
Connection connection = dataSource.getConnection();
PreparedStatement preparedStatement = connection.prepareStatement("SELECT * FROM
actor"
```

Die Ausgabe im oben aufgeführten Beispiel erfolgt über den out-PrintWriter der GET-Methode eines Browsers an das Java-Servlet. Die Daten werden mit Hilfe des out-PrintWriters an den Webbrowser übergeben, welcher diese Daten im Anschluss in HTML anzeigt.

Actor Ausgabe:

Actor: GUINESS PENELOPE
Actor: WAHLBERG NICK
Actor: CHASE ED
Actor: DAVIS JENNIFER
Actor: LOLLOBRIGIDA JOHNNY
Actor: NICHOLSON BETTE
Actor: MOSTEL GRACE
Actor: JOHANSSON MATTHEW

6. Fazit

Programmierer, welche in ihrer Java-Anwendung eine Datenbankanbindung benötigen, finden aus meiner Sicht keinen Weg an „JDBC“ vorbei. Durch die Techniken mittels JDBC wird eine Anwendung erzeugt, welche sowohl plattformunabhängig, wie auch DBMS-unabhängig ist. Die JDBC-API alleine würde genügen, weist aber im Detail viele Schwächen auf wie z.B.:
1. Der Quelltext wird durch die permanente Überprüfung mittels try-/catch-Blöcken sehr unübersichtlich. Dies hat zur Folge, dass in größeren Projekten die Fehlersuche sowie die Wartbarkeit des Codes erschwert werden.
2. Des Weiteren ist permanent auf offene Verbindungen zur Datenbank zu achten. Es kann zu einem bestimmten Zeitpunkt nicht genau erkannt werden, wie viele Verbindungen gerade offen sind. Dies hat zur Folge, dass die Anzahl der maximal möglichen Verbindungen bereits überschritten sein kann. Wenn eine Anwendung auf mehreren Clients verteilt läuft, kann dies schnell zu einem großen Problem werden.
3. Das ständige Iterieren über Ergebnisse einzelner SQL Abfragen kann arbeitsintensiv werden. Angenommen eine SQL-Abfrage liefert als Ergebnis mehrere Personen. Diese

Ergebnisliste von Personen soll in ein Art Formular übertragen werden. Bei einer solchen Aufgabenstellung ist permanent auf das ResultSet mit den Ergebniswerten zurückzugreifen. Die einzelnen Einträge müssen dann entweder mit Indizes oder Spaltennamen angesprochen werden.
Spring setzt unter anderem diese drei Punkte einfacher und deutlich komfortabler um.
Exception Handling wird vom Spring-Framework komplett übernommen. Somit müssen im Quelltext an keiner Stelle mehr try-/catch-Blocke verwendet werden. Sämtliche Fehlermeldungen werden in solche vom Typ DataAccessException übersetzt. Diese Art der Fehlermeldungen gibt selbstständig Auskunft über die Art des Fehlers.
Verbindungen zur Datenbank müssen bei Spring nicht überprüft bzw. offen gehalten werden. Das Framework entscheidet selbstständig wann welche Verbindung geschlossen werden sollte. Somit kann es nicht zu Engpässen beim Zugriff auf bestimmte Datenkomponenten kommen.
Zu guter Letzt das Data Access Objekt. Da Spring sämtliche Anfragen in einzelne Objekte übersetzt, kann wie im vorherigen Punkt beschrieben Spring darauf achten, die Verbindung so schnell wie möglich wieder zu schließen. Sämtliche Objekte werden lokal im Speicher gehalten und dort von der JVM auch selbstständig mithilfe der GarbageCollectors aufgeräumt. Es kann natürlich auch umgekehrt sehr einfach ein Objekt wieder zurück in die Datenbank geschrieben werde.

Alles in Allem finde ich, dass Spring keine Alternative zu JDBC sondern eine sehr gute Ergänzung ist. Der Schritt zur Java EE Welt wird durch Applikationsserver wie z.B. den Wildfly deutlich erleichtert. Ein zusätzliches Einpflegen der Benutzerinformationen ist nur an einer Stelle von Nöten und kann hiermit von mehreren Anwendungen falls nötig mitgenutzt werden. Das Endanwendungsprogramm ist somit noch ein wenig schlanker und der Programmierer muss sich nicht um die Pflege der Datenbanktreiber kümmern.

Literaturverzeichnis

[1] Christian Ullenboom (2014). Java SE 8 Standard Bibliothek – Das Handbuch für Java Entwickler. 2. Auflage. Galileo Press

[2] Jussi Prevost: „Datenbankanbindung: JDBC und SQLJ“ http://www.inf.uni-konstanz.de/dbis/teaching/ss01/data-on-the-web/local/jdbcsqlj.pdf (abgerufen am 28.03.2015)

[3] Nitin Nanda: „JDBC drivers in the wild“ http://www.javaworld.com/article/2076117/java-web-development/jdbc-drivers-in-the-wild.html (abgerufen am 01.04.2015)

[4] W. Lehner: „Java Database Connectivity“ https://wwwdb.inf.tu-dresden.de/misc/SS13/DBProg/06-JDBC.pdf (abgerufen am 28.03.2015)

[5] Christian Ullenboom: „Mit Java an eine Datenbank andocken“ http://www.tutego.de/javabuch/Java-7-Mehr-als-eine-Insel/1/1507_16_005.html#dodtp94fd510a-01f2-4fa5-8547-19e353a9446a (abgerufen am 29.03.2015)

[6] Jamie van Bezooijen: „Adding a MySQL JNDI datasource to JBoss Wildfly via CLI“ http://blog.squins.com/2014/12/adding-a-mysql-jndi-datasource-to-jboss-wildfly-via-cli/ (abgerufen am 16.04.2015)

[7] Ramakanta: „How To Configure datasource in WildFly“ http://www.techpaste.com/2014/05/how-to-configure-datasource-in-jboss-wildfly/ (abgerufen am 16.04.2015)