Datenbanksysteme 2

Persistence

- Java Platform
- JEE
- Persistence
- Jakarte Persistence

Inhalt

- Java & Java Plattform
- Java Enterprise Edition (JEE 6.0)
  - Historie
  - Infrastruktur
  - Application Server
  - API‘s
  - Unterschiedlichen JEE Container
  - Packaging
- Servlets
- JSP / JSF
- EJB (JPA)
  - Exkurs: O/R-Mapping
  - Enterprise JAVA Beans
  - JPA (Jakarta Persistence)
- Tomcat (Konfiguration)
- Glassfish (Konfiguration)

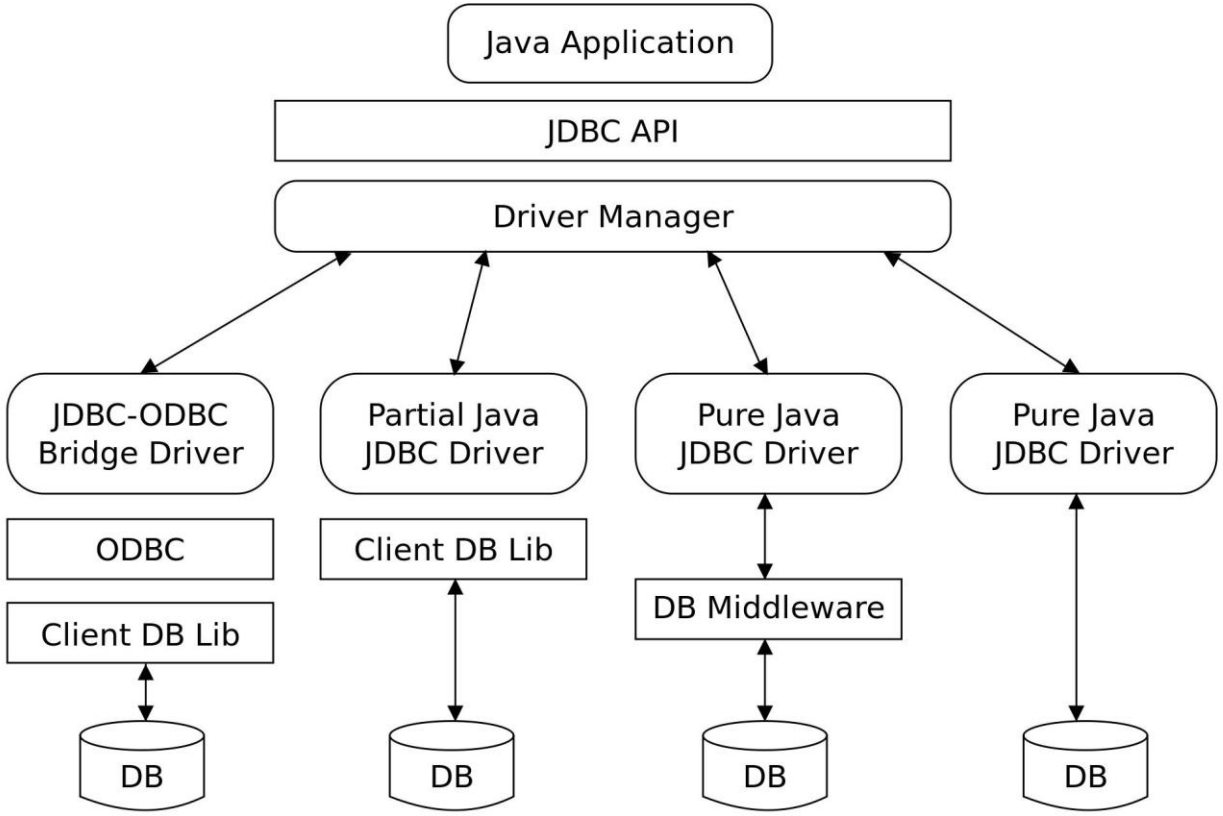
Persistence

- Grundlagen
- O/R-Mapping

JDBC – Java Database Connectivity

- **Java Database Connectivity** ist eine Datenbankschnittstelle der Java-Plattform, die eine einheitliche Schnittstelle zu Datenbanken verschiedener Hersteller bietet und speziell auf relationale Datenbanken ausgerichtet ist.
- JDBC ist in seiner Funktion als **universelle Datenbankschnittstelle** vergleichbar mit z. B. **ODBC** unter Windows oder DBI unter Perl.
- Zu den Aufgaben von JDBC gehört es, Datenbankverbindungen aufzubauen und zu verwalten, SQL-Anfragen an die Datenbank weiterzuleiten und die Ergebnisse in eine für Java nutzbare Form umzuwandeln und dem Programm zur Verfügung zu stellen.
- Für jede spezifische Datenbank sind eigene Treiber erforderlich, die die JDBC-Spezifikation implementieren. Diese Treiber werden meist vom Her-steller des Datenbank-Systems geliefert.
- JDBC ist Teil der **Java Standard Edition** seit JDK 1.1. Die JDBC Klassen lie-gen in den Java **packages java.sql** und **javax.sql**. Seit JDBC 3.0 wird JDBC im Rahmen des Java Community Processes weiterentwickelt. JSR 221 ist die Spezifikation der Version JDBC 4.0; aktuell 4.1 (Teil von Java SE 7).

JDBC – Treibertypen #1



JDBC – Treibertypen #2

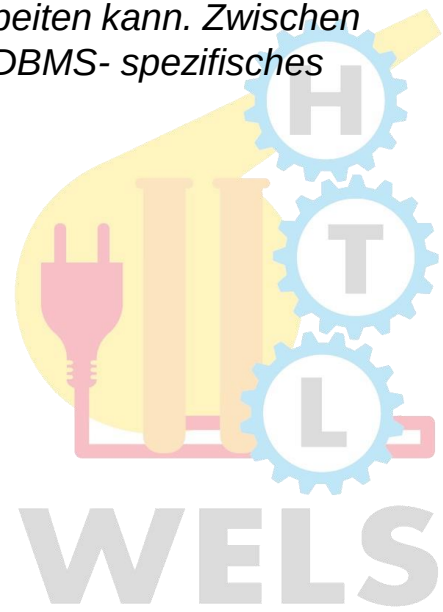
- **Typ 1: JDBC-ODBC-Bridge mit ODBC-Treiber**
  Die Brücke greift über den **ODBC-Treiber** auf die Datenbank zu. Das hat den Vorteil, dass über die ODBC- Treiber auf alle ODBC-Quellen zugegriffen werden kann. Der Nachteil dieser Lösung ist ihre **Plattformabhängigkeit**, da der JDBC-Treiber aus Java und ODBC-Binärcode besteht. In vielen Fällen muss auf dem Client-Rechner die ODBC Quelle eingerichtet werden, und dadurch erhöht sich deutlich der **Installationsaufwand** bei jedem Anwender.
- **Typ 2: Natives API und teilweise in Java geschriebener Treiber**
  Ein solcher Treiber ist sehr gut für Datenbankhersteller geeignet, weil er sich leicht und schnell implementieren lässt. Der Hersteller setzt seinem **bereits vorhandenen Treiber**, der beispielsweise in C geschrieben wurde, eine **JDBC-Schicht** auf. Hier wandelt der Treiber die JDBC-Aufrufe in herstellerspezifische Datenbankaufrufe um. Auch bei diesem Typ handelt es sich um eine plattformabhängige Lösung, da der vorhandene DBMS-Treiber in binärer Form vorliegt.

JDBC – Treibertypen #3

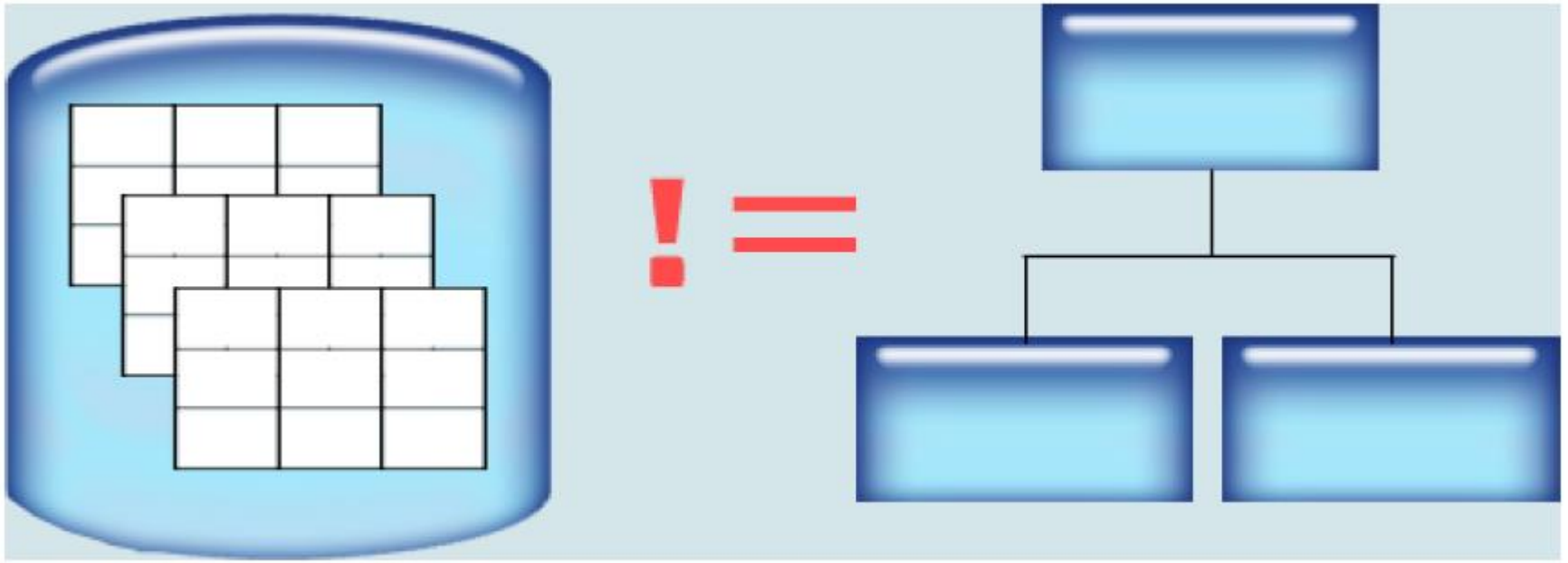
- **Typ 3: Middleware mit in Java geschriebenem Treiber**
  - Hierbei handelt es sich um einen JDBC-Treiber, der nur in Java geschrieben worden ist. Er ist der einzige, der als **Dreischichtenmodell** implementiert wurde.
  - Der Treiber übersetzt die JDBC- Aufrufe **in ein vom DBMS unabhängiges Netzwerkprotokoll**, das durch einen Konnektor (Middleware) in ein DBMS-spezifisches Protokoll umwandelt wird. Die Middleware selbst muss nicht aus Java-Code bestehen, sondern kann in beliebiger Programmiersprache entwickelt worden sein. Die Voraussetzung ist nur ein gemeinsames **Kommunikationsprotokoll**.
  - Die Middleware ist in der Lage den Java-Client mit unterschiedlichen Datenbanken zu verbinden, wenigstens mit diesen, für welche sie die notwendigen DBMS-Treiber hat.
  - Diese Lösung ist die **flexibelste** von allen **vier Typen**, denn man braucht nur die Middleware anzupassen um ein neues DBMS unterstützen zu können. Das Client-Programm bleibt unverändert.

JDBC – Treibertypen #4

- **Typ 4: Natives API mit in purem Java geschriebenem Treiber**
  - Es handelt sich ebenso um einen reinen Java-Treiber, der aber nur nach dem **Zweischichtenmodell** arbeitet.
  - Dieser Typ ist ähnlich dem Typ 2 aufgebaut und wird vor allem durch diejenigen Datenbankhersteller entwickelt, die bereits einen Typ 2 Treiber haben und ihn durch einen **Java-Treiber ersetzten möchten**.
  - Der Treiber wandelt **die JDBC-Aufrufe** direkt in das von dem **DBMS verwendete Netzwerkprotokoll** um. Das ermöglicht direkte Kommunikation des DBMS-Servers mit dem Java-Client.
  - Jedoch ist der Treiber nur aus der Sicht des Anwendungsentwicklers monolithisch aufgebaut. *In der Realität ist seine Struktur komplexer, denn es gibt noch kein in Java geschriebenes DBMS, das die JDBC-Aufrufe direkt verarbeiten kann. Zwischen dem Treiber und dem DBMS liegt ebenfalls eine Zwischenschicht (Middleware), die die JDBC-Aufrufe in ein DBMS- spezifisches Protokoll umwandelt.*
  - Der Unterschied zum Typ 3 ist die Existenz einer herstellerspezifischen Schicht, die für den Anwendungsprogrammierer unsichtbar ist.



O/R – Mapping Core Challenge



Motivation Persistenz

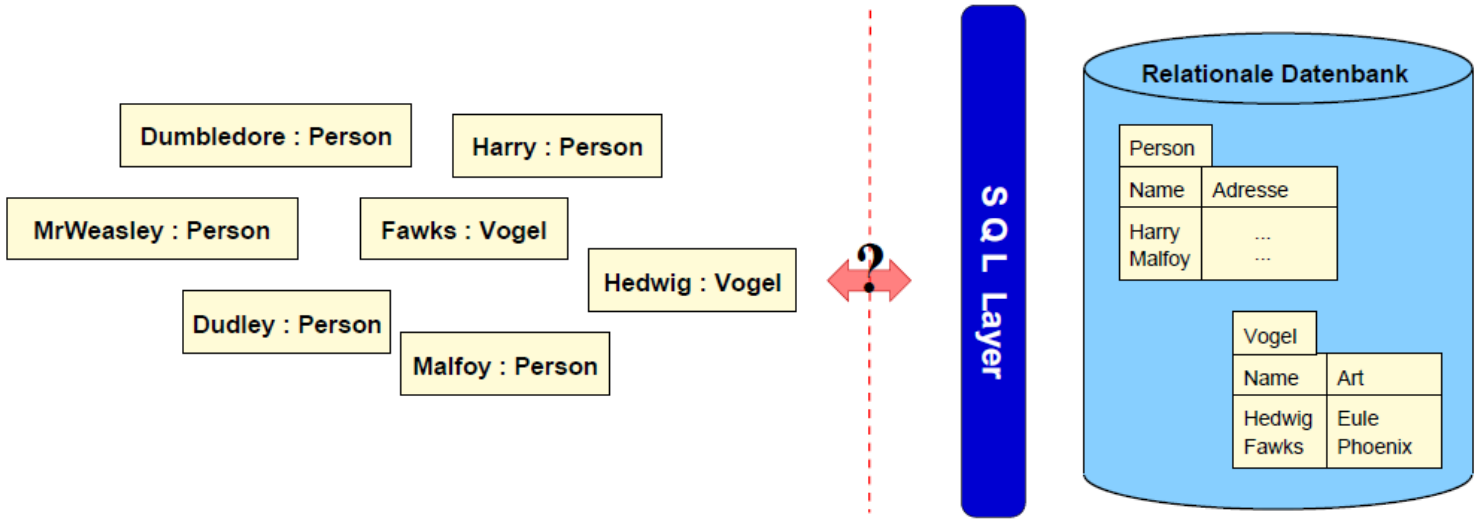
- In Java Programmen müssen Daten so gespeichert werden, das sie bei einem späteren Aufruf wieder verfügbar sind
- Möglichkeiten:
  - Einfache Dateien: (Serialized Objects, XML, …)
    - Für große Datenmengen ungeeignet
    - Keine Sicherheitsmechanismen, …
  - Persistenz durch Speicherung in DB
    - Optimiert für große Datenmengen
    - Transaktionskonzept
    - Backups
    - …

Motivation - Speichermöglichkeiten

- **Objektdatenbanken**
  - Unterstützen das OO-Modell direkt
  - nicht sehr verbreitet
- **Relationale Datenbanken**
  - passen nicht direkt zum OO-Modell
  - massive Anpassungen sind notwendig
  - weit verbreitet
- Sonstigen (hierarchische DB‘s, einfache Dateien, XML)
  - auch hier sind Anpassungen notwendig

Die Wirklichkeit

- Relationalen Datenbanken sind die überwiegende Mehrzahl der verwendeten Systeme



- Java Welt vs. Relationale Welt
  - ➔ Zwei Welten müssen zusammenfinden

Impedance Mismatch

- OO und Relationale Welt sind stark unterschiedlich

	OO	Relational
Modellierung	Zustand & Verhalten	nur Daten
Identität	ja	nein
Navigation	über Assoziationen	Daten duplizieren, joins. . .
Anfragen	über Attributwerte	Selektion und Projektion
Granularität	einzelne Objekte	Ergebnis*mengen*

- Moderne Systeme sind in der Regel OO-Systeme
- ➔ Saubere und performante Anpassung ist notwendig

Manuelles vs. Automatisiertes Persistieren

- Verwaltung persistenter Daten ein **Standardproblem** der Software-Entwicklung
- lange Zeit Diskussionen, ob **CRUD**-Operationen händisch codiert oder automatisiert werden sollten
  ➔ mittlerweile Konsens: automatisiert
- **händische JDBC-Konsistenz ist immer schneller**/effizienter als eine automatisierte Lösung, aber eine automatisierte Lösung hat nachstehende Vorteile:
  - Keine exakten und detaillierten DB-Kenntnisse notwendig
  - flexibler bei Änderungen der Objekt-Struktur
  - Programmcode ist logischer
  - …

Persistenz-Frameworks

Persistenz-Framework - Funktionalität

- eine Menge von Klassen und Schnittstellen, die zusammen ein wiederverwendbares Design für ein bestimmtes Problemfeld anbieten
- „der Rahmen“ für eine konkrete Lösung für ein konkretes Problem
- durch den Nutzer konfigurier- und erweiterbar
  - ➔ durch Ableiten von vorgegebenen Klassen, Implementieren von Schnittstellen
- benutzt das Hollywood-Prinzip: *Don’t call us. We’ll call you.*
  - ➔ Idee: Nutzer-definierte Klassen werden durch die vordefinierten aufgerufen.
- inhärent objekt-orientiert (meist auch so implementiert)
- Spezialform des abstract Factory Pattern

Persistenz-Framework - Grundlagen

- **Beispiele** für Java Frameworks
  - für GUIs: Java Swing, Java AWT
  - für Kryptographie: Java Cryptography Architecture
  - für Application Server: JEE (Jakarta EE)
- **Was wir betrachten wollen**
  - ein Persistence Framework
    - Objekte in die DB abbilden, mit Identität versehen
    - Speichern und Wiederfinden von Objekten
    - Caching, Transaktionen (mit *commit* und *rollback*)

Anforderungen an eine Persistenz-Komponente

- Die Persistenz-Komponente muss **sicherstellen**, dass **komplexe Objektstrukturen auf Tabellen** abgebildet werden können
- **Geschäftslogik** und **Datenbankmodell** voneinander unabhängig sind
- Daten in der Datenbank **konsistent** sind (➔ z.B.: Transaktions-mechanismus)
- Geschäftstransaktionen sinnvoll auf Datenbanktransaktionen abgebildet werden
- Außerdem sollte sie eine **einfache Schnittstelle** für den Anwendungsprogrammierer bieten

Probleme des O/R-Mappings

- Das OO Modell enthält reichere Strukturierungsmechanismen als das relationale Modell.
- Die folgenden **Konzepte** müssen **transformiert** werden:
  - Objektidentität
  - Klassen (Attribute)
  - Vererbungshierarchien
  - Assoziationen
- Objektlebenszyklus(transient vs. persistent)

Alternativen für relationales Persistieren in Java

Tool	Version	Anmerkung	Datum
Serialisierung	8.0	Teil der JSE (Java 8)	2014
JDBC	4.2	Teil der JSE (Java 8)	2014
ODMG 3.0	3.0	Standard	Eingestellt
JDO	3.0	Standard	2010
Hibernate	5.0.4	POJO	2015
EJB	3.1	Teil der JEE 7	2013
JPA	2.1	Teil der JEE 7	2013
Java Persistence	2.2	Teil von Jakarta EE	2018

https://en.wikibooks.org/wiki/Java_Persistence

Eine kurze Geschichte der Java Persistenz

- 1997: JDBC war erstes Framework zum Persistieren in Java (Teil von Java 1.1)
- 1998: EJB 1.0 – Sehr rudimentär
- 1999: EJB 1.0 – Anbindung Entity Beans
- 2001: EJB 2.0 – Wesentliche Umgestaltung
- 2001: Hibernate 1.0 – Wesentliche Vereinfachung (POJOS)
- 2002: JDO 1.0 – „Leichtgewichtige“ Alternative zu EJB
- 2003: EJB 2.1
- 2003: Hibernate 2.0 – Massive funktionale Erweiterung
- 2006: JDO 2.0 - Erweiterung
- 2006: EJB 3.0 – Wesentliche Vereinfachung des gesamten Methodik
- 2008: JDO 2.2 – Weiterentwicklung durch die Apache SF
- 2009: EJB 3.1 – Weitere Vereinfachung
- 2010: JDO 3.0 – Etliche Kommerzielle Anbieter
- 2013: EJB 3.2 – EJB Lite erweitert

Abbildung Java-Klassen auf SQL-Typen

- Es gibt keine strenge Vorschrift, wie Datentypen in Java in Datentypen der Datenbank zu mappen sind. Im Fall der Boolschen Variablen existiert in vielen Datenbanken kein Pendant, da es von SQL 92 nicht vorgegeben ist.
- Wichtige Relationen sind
- java.lang.String <=> VARCHAR, CHAR
- java.sql.Timestamp <=> TIMESTAMP
- java.sql.Date <=> DATE
- double <=> DOUBLE, FLOAT
- float <=> REAL
- boolean <=> char(1), SMALLINT
- int <=> INTEGER

Transaktionen

- 1. Persistente Objekte kann man
  - neu anlegen, modifizieren oder löschen.
- 2. Alle Manipulationen sind Teil einer *Transaktion*.
- 3. Die obigen Operation
  - führen nicht sofort zu einem DB-Update,
  - erst bei ***Commit*** (Transaktionsende) wird die DB geändert,
  - bei ***Rollback*** (Abbruch) wird alter Zustand wiederhergestellt.