FH Bielefeld
University of
Applied Sciences

Campus Minden

Webbasierte Anwendungen

SS 2018

Datenbankanbindungen

Dozent: B. Sc. Florian Fehring

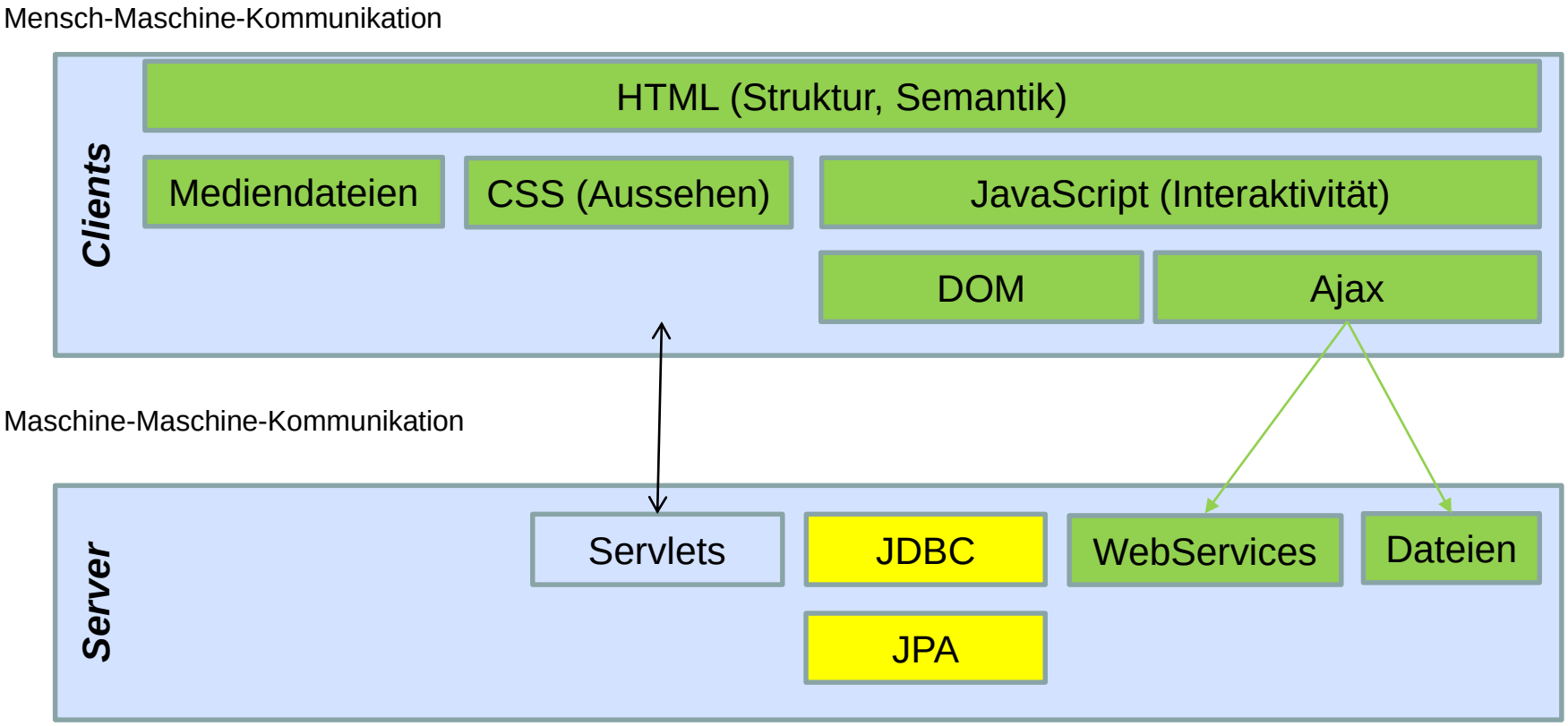
mailto: florian.fehring@fh-bielefeld.de

Studiengang Informatik

Datenbankanbindungen

1. **Kontext und Motivation**
2. Java Database Connectivity
3. Java Persistence API
4. Darüber hinaus
5. Projekt

Problemfelder



Anforderungen

Welche Anforderungen werden als nächstes bearbeitet?

TODO

- Daten aus Datenbank lesen
- Artikel speichern
- Kommunikation untereinander

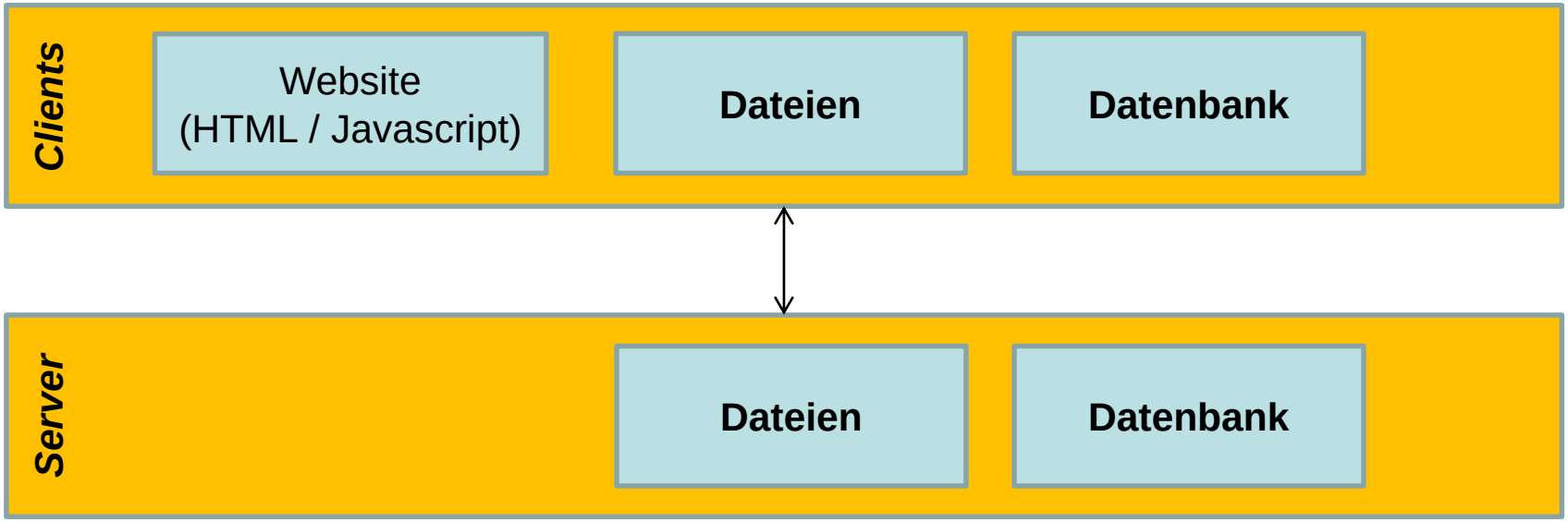
DONE

- …
- Formular für Kommentare
- Schickes Design für die Seite
- Mediendatein einbinden
- Animationen
- Mehrsprachen-Fähigkeit
- (lokales) Speichern von Artikeln
- Client-Position anzeigen
- Offline-Verwendung ermöglichen
- Inhaltsverzeichnisse
- Formlareingaben in Seite einfügen
- Navigation über Tastaturkürzel
- Externe Inhalte einbinden
- Artikel vom Server einbinden
- Kommentare vom Server
- Medien hochladen
- Kommentare hochladen

Problemstellung

Anforderung
Benutzer sollen nicht bei jeder Bestellung ihre Daten erneut eingeben müssen. Darüber hinaus sollen wesentlich mehr Produkte auf der Website präsentiert werden.

Problem
Sowohl Benutzerdaten als auch Produktdaten müssen irgendwo gespeichert werden. Aber wo? Welche Anforderungen müssen erfüllt sein?



Anforderungsanalyse

Funktionale Anforderungen:

- Daten **müssen** abgespeichert werden können
- Daten müssen geladen werden können
- Daten müssen sicher vor unberechtigtem Zugriff sein
- Daten müssen vor Verlust geschützt werden
- Daten **sollen** nicht unnötig hin und her gesendet werden

Nicht funktionale Anforderungen:

- Datenschutzbestimmungen müssen eingehalten werden
  - Allgemeine Informationen über Datenschutz

Anforderungsanalyse

Anforderung: Benutzer sollen nicht bei jeder Bestellung ihre Daten erneut eingeben müssen.

Funktionale Anforderungen:

- Daten **müssen** abgespeichert werden können
  - Von wem? -> Endnutzer
- Daten müssen geladen werden können
  - Von wem? -> Endnutzer und Anbieter
- Daten müssen sicher vor unberechtigtem Zugriff sein
  - Vor wem? -> Andere Endnutzer, Hacker,…
- Daten müssen vor Verlust geschützt werden
  - Priorität? -> Gering (Endnutzer) bis Hoch (Anbieter)
- Daten **sollen** nicht unnötig hin und her gesendet werden

Nicht funktionale Anforderungen:

- Datenschutzbestimmungen müssen eingehalten werden
  - Bei Benutzerdaten immer besonders wichtig!

Anforderungsanalyse

Anforderung: Es sollen wesentlich mehr Produkte auf der Website angeboten werden.

Funktionale Anforderungen:

- Daten **müssen** abgespeichert werden können
  - Von wem? -> Anbieter
- Daten müssen geladen werden können
  - Von wem? -> Endnutzer, Anbieter
- Daten müssen sicher vor unberechtigtem Zugriff sein
  - Umfang? -> Änderungen nur für Anbieter
- Daten müssen vor Verlust geschützt werden
  - Priorität? -> Hoch
- Daten **sollen** nicht unnötig hin und her gesendet werden

Nicht funktionale Anforderungen:

- Datenschutzbestimmungen müssen eingehalten werden
  - Bei öffentlichen Daten nicht relevant

Tool-Analyse

Bekannte Tools:

- Cookies beim Endnutzer
  - Nachteile: eingeschränkte Größe, bei jeder Anfrage gesendet
- Dateien auf dem Server des Anbieters
  - Nachteile: Bei großen Datenmengen unhandlich, ineffizient
- Datenbank auf dem Server des Anbieters
  - Vorteile: Daten jederzeit von überall zugreifbar.

Weitere Möglichkeiten:

- Datenbank lokal beim Endnutzer
  - Vorteile: Je nach Anwendungsfall Einsparung von Datentransfer, Kontrolle beim Benutzer
  - Nachteile: Daten nicht auf jedem Gerät vorhanden

Tool-Analyse

Anforderung: Benutzer sollen nicht bei jeder Bestellung ihre Daten erneut eingeben müssen.

Bekannte Tools:

- ~~Cookies beim Endnutzer~~
  - Nachteile: ~~eingeschränkte Größe~~, bei jeder Anfrage gesendet
- ~~Dateien auf dem Server des Anbieters~~
  - Nachteile: Bei großen Datenmengen unhandlich, ineffizient
- Datenbank auf dem Server des Anbieters
  - Vorteile: Daten jederzeit von überall zugreifbar.

Weitere Möglichkeiten:

- Datenbank lokal beim Endnutzer
  - Vorteile: Je nach Anwendungsfall Einsparung von Datentransfer, Kontrolle beim Benutzer
  - Nachteile: Daten nicht auf jedem Gerät vorhanden

Tool-Analyse

Anforderung: Es sollen wesentlich mehr Produkte auf der Website angeboten werden.

Bekannte Tools:

- ~~Cookies beim Endnutzer~~
  - Nachteile: eingeschränkte Größe, bei jeder Anfrage gesendet
- ~~Dateien auf dem Server des Anbieters~~
  - Nachteile: Bei großen Datenmengen unhandlich, ineffizient
- Datenbank auf dem Server des Anbieters
  - Vorteile: Daten jederzeit von überall zugreifbar.

Weitere Möglichkeiten:

- Datenbank lokal beim Endnutzer
  - Vorteile: Je nach Anwendungsfall Einsparung von Datentransfer, ~~Kontrolle beim Benutzer~~
  - Nachteile: ~~Daten nicht auf jedem Gerät vorhanden~~

Tool-Analyse II - Fazit

Anforderung: Benutzer sollen nicht bei jeder Bestellung ihre Daten erneut eingeben müssen.

- Datenbank auf dem Server des Anbieters

Anforderung: Es sollen wesentlich mehr Produkte auf der Website angeboten werden.

– Datenbank auf dem Server des Anbieters

Folgefragen:
Welche Techniken stehen zur Anbindung einer Datenbank zur Verfügung?
Wie kann eine lokale Datenbank beim Endnutzer genutzt werden?

Datenbankanbindungen

1. Problemstellung und Motivation
2. **Java Database Connectivity**
3. Java Persistence API
4. Weitere Konzepte

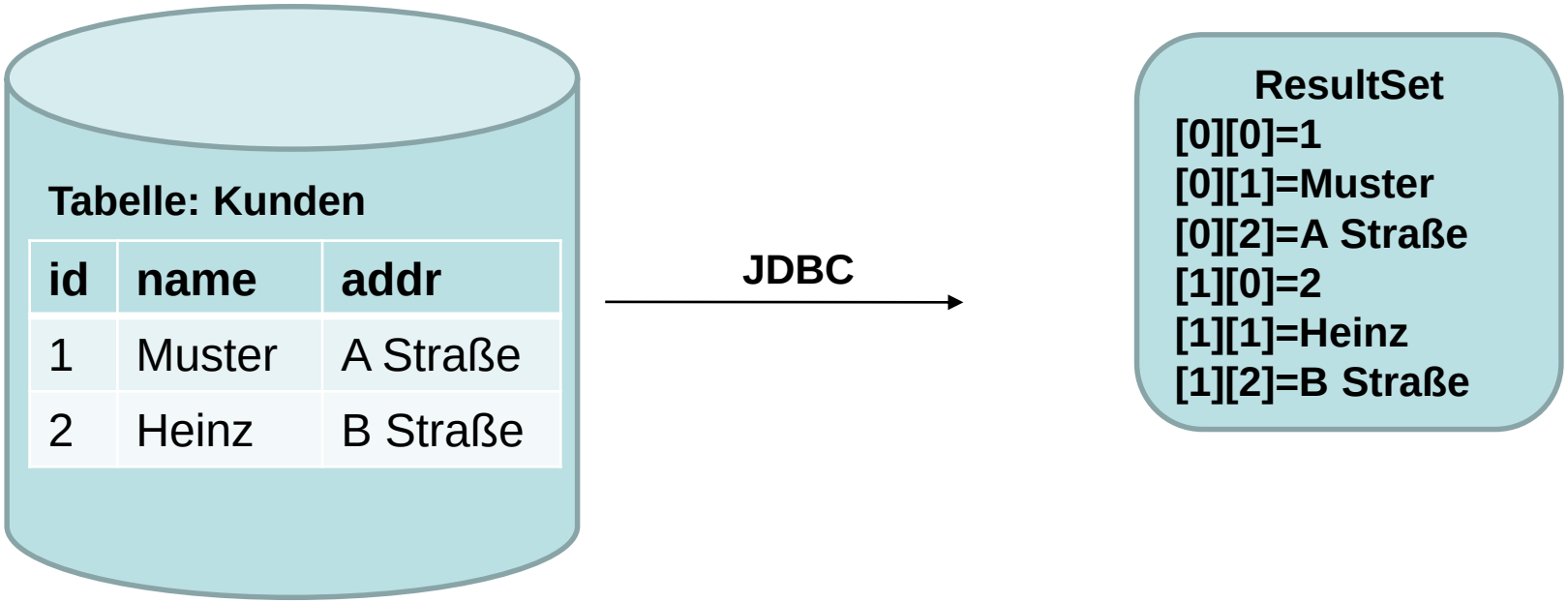
JDBC

Möglichkeiten

- Datenbankzugriff aus Java-Applikationen
- Objektrelationales Mapping
- stellt relationale Datenbankobjekte bereit

Eigenschaften

- *JDBC-API* setzt auf Java-Basisklassen auf (Paket `java.sql`)
- im *JDK (Java Development Kit)* enthalten



JDBC

Vorgehen bei der Persistierung von Objekten in 6 Schritten

1. **Laden des JDBC-Treibers**
2. **Connect zur Datenbank**
3. **Erzeugen eines Statements**
4. **Ausführen eines Statements**
5. **Auswerten des Ergebnisses**
6. **Abmelden von der DB**

Voraussetzungen

1. **JDBC Interfaces einbinden**

Syntax: `import java.sql.*`

2. **JDBC-Datenbanktreiber im Projekt verfügbar machen**

Je nach IDE unterschiedlich

JDBC

1) Laden des JDBC-Treibers:

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

- Laden eines Datenbanktreibers, welcher die Anweisungen für jeweiliges DB-System umsetzt
- Classloader mit Methode `Class.forName();`

2) Connect zur Datenbank:

Syntax:
```
Connection my_con = DriverManager.getConnection(db_url,
                                    username, passwort);
```

- Methode `getConnection` der Klasse `Driver.Manager` aus `java.sql.*`
- Parameter:

`db_url`	url der DB, zu der Verbindung erstellt werden soll
`username`	Anmeldename auf der DB
`passwort`	Passwort für die DB-Anmeldung

JDBC

3) Erzeugen eines Statements:

Syntax: `Statement my_stmt = my_con.createStatement();`

- Methode `createStatement` erfolgt für Objekt der Klasse `Connection` (bestehende Verbindung)

4) Ausführen eines Statements:

Syntax:
`ResultSet my_result = my_stmt.executeQuery(„SELECT * FROM TAB“);`
od. `int my_result = my_stmt.executeUpdate(„UPDATE TAB SET …“);`

- `executeQuery()` für Abfragen
- `executeUpdate()` für Update, Insert oder Delete
- bezieht sich immer auf Objekt der Klasse `statement` (bestehendes Statement `my_stmt)`
- `ResultSet` – Ergebnis in Form einer Tabelle

JDBC

5) Auswerten des Abfrageergebnisses:

- Typ `ResultSet` beinhaltet spezifische Ergebnistabelle für jeweilige konkrete Anfrage
- `ResultSet` stellt *get-Methoden* zur Datenauswertung bereit

```
String sql = "SELECT * FROM autor WHERE id=" + id;
ResultSet r =createConnection().createStatement().executeQuery(sql);
if (r.next())
        {
        name    = r.getString("Name");
        vorname = r.getString("Vorname");
        }
```

6) Abmelden von der DB:

- Aufrufen der Methode close für das entsprechende Verbindungsobjekt :
  `my_con.close();`

JDBC - Fazit

Nachteile von JDBC:

- Hoher Implementierungsaufwand um aus Daten konkrete Java-Objekte zu erstellen
- SQL Statements sind schnell spezifisch für eine Datenbank

Weitere Informationen

- Postgres-Datenbank:
  - https://www.postgresql.org/
- JDBC und Postgres-Datenbanken:
  - https://jdbc.postgresql.org/
- JDBC + Postgres + Netbeans-Tutorial:
  - http://www.postgresqltutorial.com/postgresql-jdbc/connecting-to-postgresql-database/

Datenbankanbindungen

1. Problemstellung und Motivation
2. Java Database Connectivity
3. **Java Persistence API**
4. Weitere Konzepte

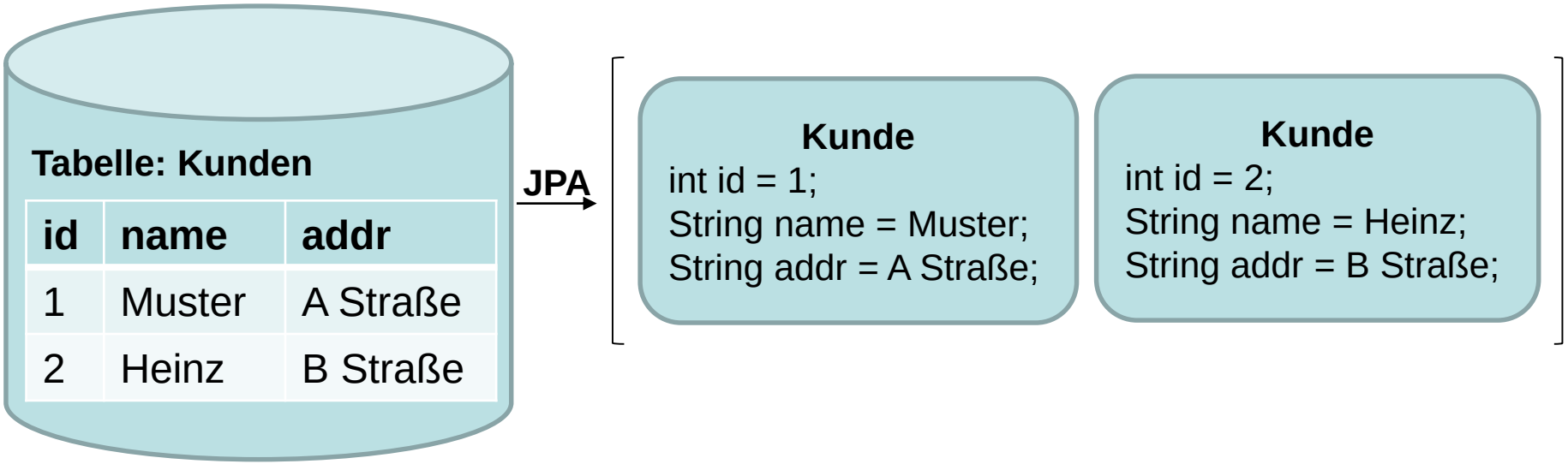
JPA

Möglichkeiten

- Datenbankzugriff aus Java-Applikationen
- Objektrelationales Mapping
- stellt *konkrete* Javaobjekte bereit

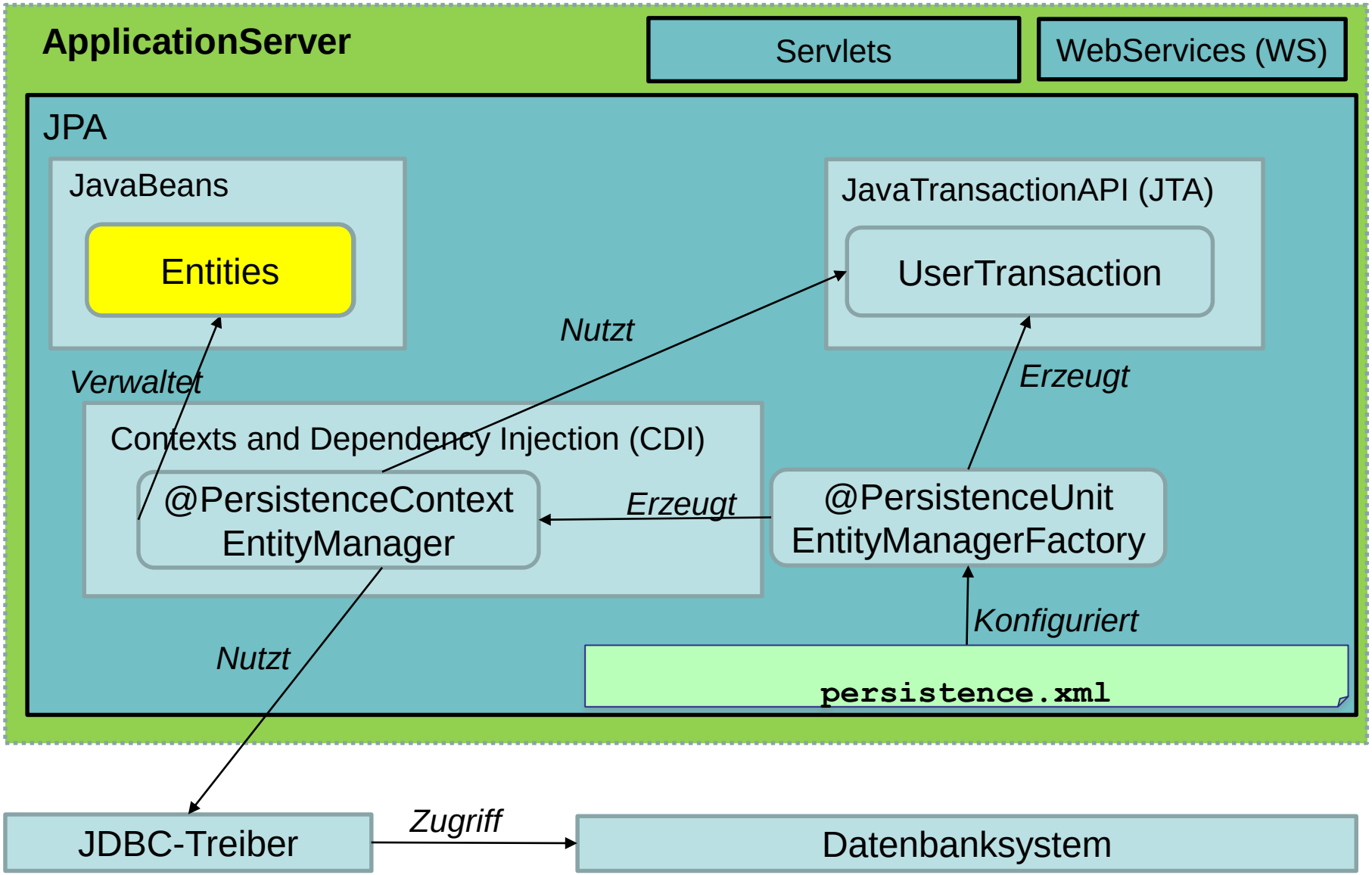
Eigenschaften

- **Etablierter Standard** für OR-Mapping
- Bestandteil der ***Java-Enterprise- Edition*** (ab JavaEE6 mit JPA2.0)
- Spezifikation mit div. Implementierungen (eclipse-link, hibernate, openJPA)
- **Weniger Implementierungsaufwand** als bei JDBC (auch weniger SQL)
- **Deklarativ** durch Annotationen



Legende: API Konzept

JPA - Konzepte



JPA - Entities

Eigenschaften

- **Datentragende** Objekte
  - Besitzen Attribute und dazugehörige get- und set-Methoden
  - Implementieren keine Geschäftslogik
- Sind **JavaBeans**
  - Besitzen mindestens eine default-Konstruktor
  - Sind serialisierbar
  - Besitzen öffentliche Zugriffsmethoden
- Erweitert um **deklarative** Anweisungen (Annotationen)
  - kennzeichnen das Objekt als persistierbar mit JPA @Entity
  - steuern das Verhalten bei Persistierung @Table,@Column
  - legen Strategien für die Persistierung fest @OneToOne

JPA – Entity-Beispiel

```
@Entity
@Table(name = "kunden")
public class Kunde implements Serializable {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = „name")
    private String name;

    private Person() {}

    public long getID(){
       return this.id;
    }
    ...
}
```

Annotation **@Entity** Markierung der Klasse zur Verarbeitung mit JPA.

Entitätsklasse **Person** wird auf Tabelle **PersonenJPA** abgebildet. **@Table**

Annotation **@Id** **kennzeichnet Primärattribut und** bestimmt Generierung das Primärschlüssels auf Tabelle **PersonenJPA**

Durch Annotation **@GeneratedValue** wird Primärschlüssel automatisch durch die DB vergeben.

Annotation **@Column** vergibt für Attribute Spaltennamen in DB

JPA - Entities

Überblick der wichtigsten Annotationen

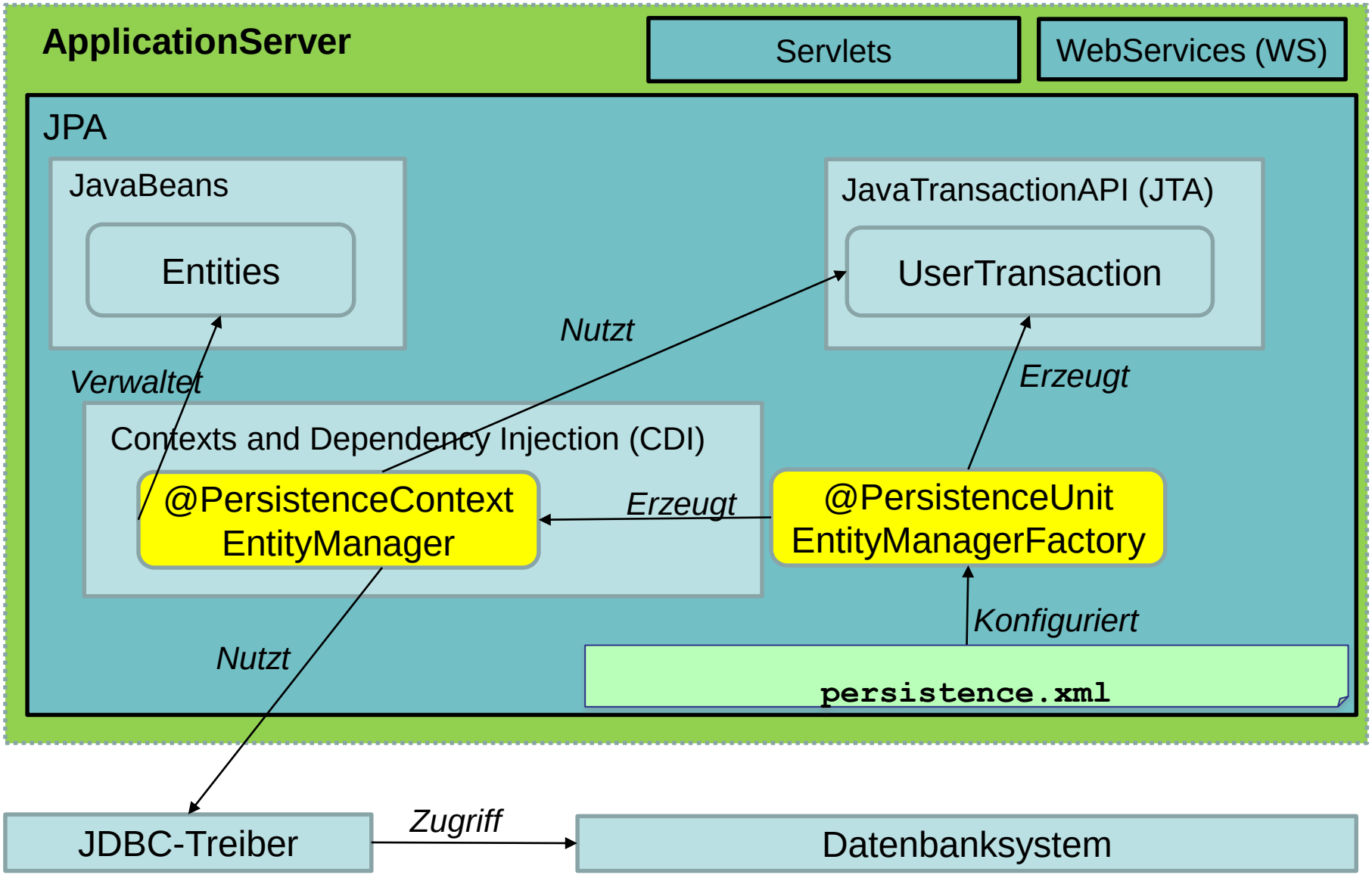
Annotation	Bedeutung
@Entity	Kennzeichnet ein Objekt als mit JPA persistierbar
@Table	Legt Einstellungen zur Tabelle fest
@Id	Kennzeichnet ein Attribut als Primärschlüssel
@GeneratedValue	Kennzeichnet ein Attribut als autom. generierten Wert
@Column	Legt Einstellungen zur Spalte (des Attributs) fest
@OneToOne	Legt eine Referenz als 1:1 Beziehung fest
@OneToMany	Legt eine Referenz als 1:n Beziehung fest
@ManyToOne	Legt eine Referenz als n:1 Beziehung fest
@ManyToMany	Legt eine Referenz als n:n Beziehung fest

Hinweise:

- Optionen der Annotationen in der jeweiligen Implementieurng nachsehen
- Alle JPA Annotationen befinden sich im Package javax.persistence. Sie sind mit Annotationen der ValidationAPI kombinierbar

Legende: API Konzept

JPA - Konzepte



JPA – EntityManager

Eigenschaften

- EntityManager werden per **Dependency Injection** hinzugefügt
  - PersistenceUnit wird als EntityManagerFactory beim Programmstart (Deployment) erzeugt (konfiguriert durch die persistence.xml)
  - ApplicationServer fordert einen EntityManager von der PersistenceUnit an
  - ApplicationServer ***injiziert*** den EntityManager in alle Attribute, wo er gefordert wird
  - Wo ein EntityManager gefordert wird, wird durch die Annotation **@PersistenceContext(unitName=[Name])** festgelegt
- Ein EntityManager ist ein PersistenceContext
  - **Mehrere** PersistenceContexte pro Applikation möglich
  - Hat **Referenzen** auf alle von ihm verwalteten Entities
  - **Bietet Methoden** zum persistieren, lesen, aktualisieren und löschen
  - Sorgt dafür, dass Objekte und Datenbank **synchron** bleiben
  - **Kennt den Zustand** der Objekte außerhalb und innerhalb der Datenbank

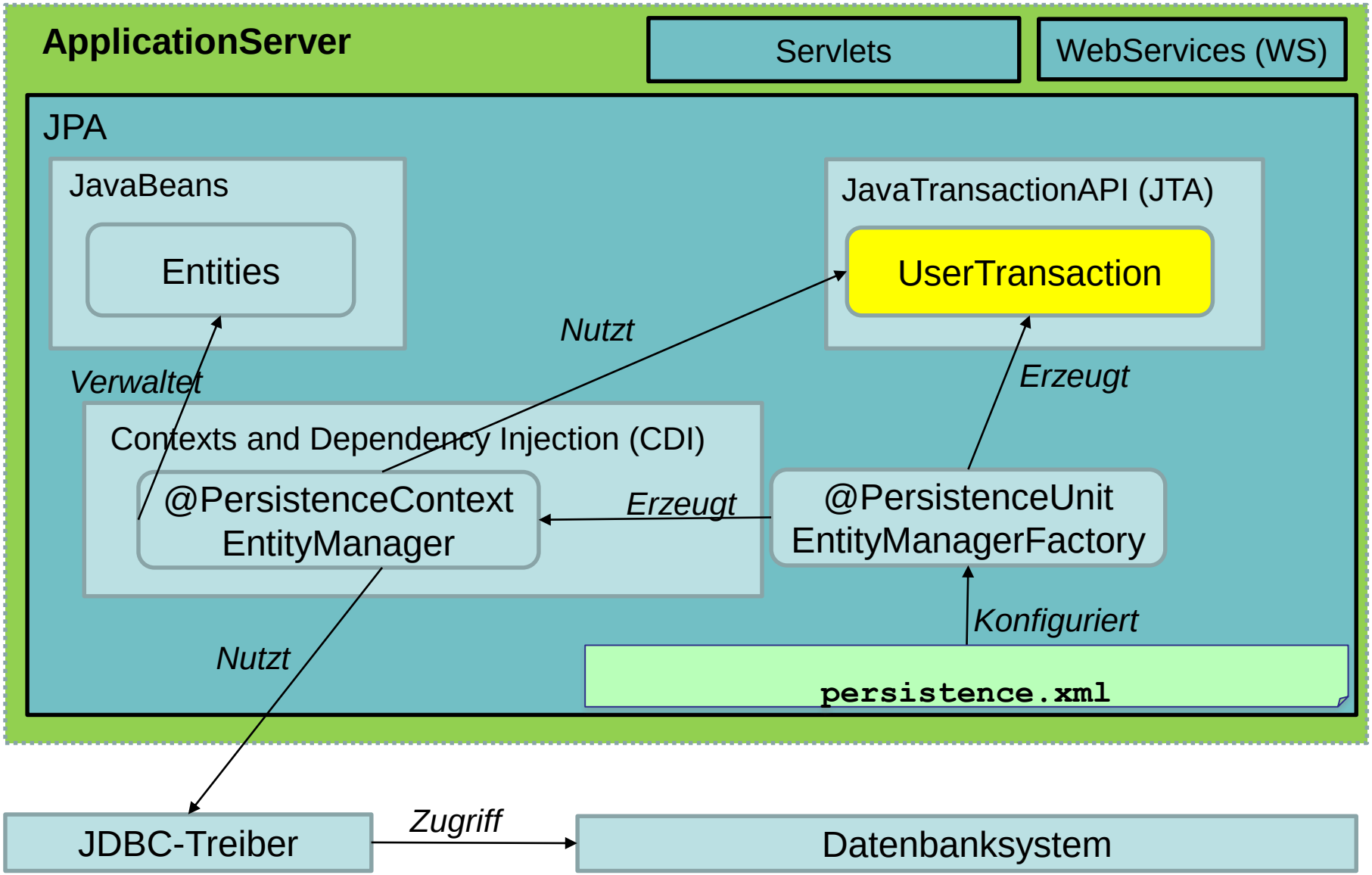
JPA – EntityManager Beispiel

```
public class WebShopAPI {
    /**
     * Entity Manager em, basierend auf der in presistence.xml ge-
     * speicherten Datenbankverbindung
     */
    @PersistenceContext(unitName = „WebShopPU")
    private EntityManager em;

    public Person getPerson(int id) {
        // Tue irgend etwas um die Person aus der Datenbank zu laden
    }
}
```

Legende: API Konzept

JPA - Konzepte



JPA – JavaTransactionAPI

Eigenschaften

- JTA kümmert sich um das Transaktionsmanagement
  - In JPA größtenteils automatisch und im Hintergrund
  - Entwickler kann Einfluss auf die Transaktionen nehmen, durch eine UserTransaction.
- UserTransaction werden per **Dependency Injection** hinzugefügt
  - PersistenceUnit wird als EntityManagerFactory beim Programmstart (Deployment) erzeugt
  - ApplicationServer fordert ein UserTransaction-Objekt von der PersistenceUnit an
  - ApplicationServer ***injiziert*** das UserTransaction-Objekt in alle Attribute, wo es gefordert wird
  - Wo ein UserTransaction-Objekt gefordert wird, wird durch Anlegen eines Atttibutes vom Typ UserTransaction mit zugehöriger Annotation @Resource festgelegt
- Die UserTransaction
  - Bietet Methoden um Transaktionen zu starten, zu beenden, Rollback,…
  - Muss bei schreibenden Zugriffen (mit Referenzen) genutzt werden

Hinweis: @Resource wird auch für die Injizierung anderer Objekte genutzt

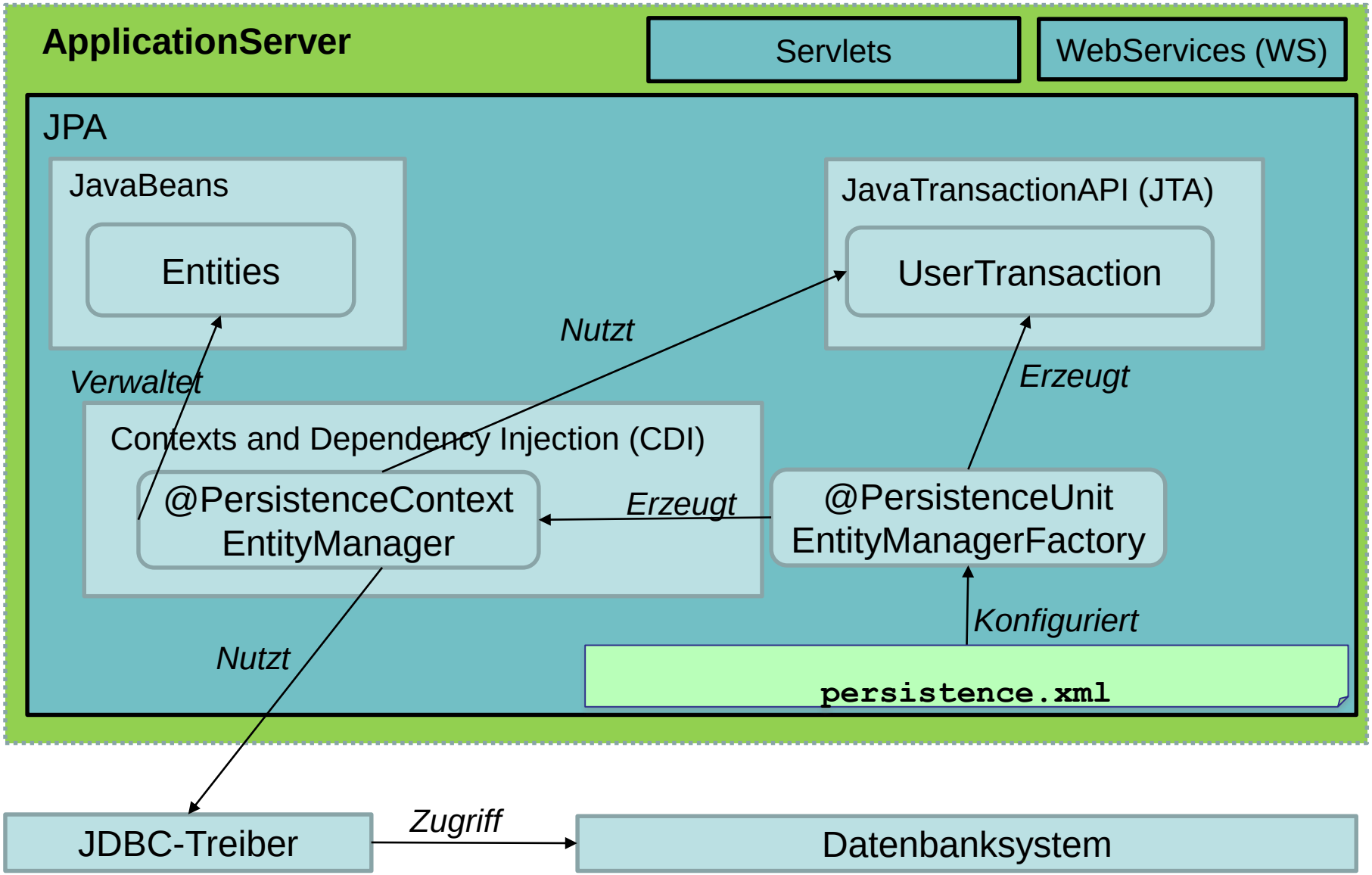
JPA – JTA Beispiel

```
public class WebShopAPI {
    /**
     * Entity Manager em, basierend auf der in presistence.xml ge-
     * speicherten Datenbankverbindung
     */
    @PersistenceContext(unitName = „WebShopPU")
    private EntityManager em;

    /**
     * User Transaction utx zur Kommunikation mit Datenbank
     */
    @Resource
    private UserTransaction utx;
    …
}
```

Legende: API Konzept

JPA - Konzepte



JPA – Laden von Entities

```
public  <T> T find(Class<T> entityClass, Object primaryKey)
public  <T> T getReference(Class<T> entityClass, Object
primaryKey)
```

- Die Methode **`find()`** liefert das gefundene Entity bzw. **`null`** **(bei nicht Vorhandensein)** zurück, wenn es einen entsprechenden Datenbankeintrag gibt. Als Parameter sind die Entity-Klasse und der Primärschlüssel zu übergeben.
- Zuerst wird im Persistenzkontext gesucht, erst dann per SELECT in der Datenbank

- Die Methode **`getReference()`** ist ähnlich, nur wirft Sie bei nicht Vorhandensein die Exception **`EntityNotFoundException`** statt **`null.`**

```
// Kunden anhand des Primärschlüssels finden
Kunde kunde = this.em.find(Kunde.class,id);

// Kunden anhand des Primärschlüssels finden
try {
  Kunde kunde = this.em.getReference(Kunde.class, id);
} catch(EntityNotFoundException ex) {…}
```

JPA – Speichern eines Entity

- Mit der Methode **`persist()`** vom Interface **`EntityManager`** wird ein transientes Entity in der Datenbank gespeichert und in den Zustand ***managed*** überführt.

```
// Eine neue Position anlegen
TblLocation location = new TblLocation();
this.utx.begin();
location.setStreet(street);
…
location.setLatitude(lat);
location.setLongitude(lon);
location.setName(nameLoc);
this.em.persist(location);
this.utx.commit();
```

JPA – Aktualisieren von Entities

- ***Managed*** Entities werden im Persistenzkontext überwacht.
- Der **`Entity-Manager`** synchronisiert Änderungen mit der Datenbank
- -> auch genannt: ***Automatic Dirty Checking***
- -> Objekt, welches Änderungen enthält, die noch nicht in der Datenbank synchronisiert sind: ***Dirty Object***
- Am Ende einer Transaktion wird automatisch ein UPDATE an die DB gesendet.

```
this.utx.begin();
// Werte einer Konfiguration setzten
loction.setName("FH Bielefeld");
// Änderungen mit DB synchronisieren
// this.em.update(location) nicht notwendig
this.utx.commit();
```

JPA – Löschen von Entities

- ***Persistente JPA-Entities müssen explizit gelöscht werden***, da der Garbage-Collector die Daten in der DB von nicht mehr referenzierten Objekten nicht automatisch löschen würde.
- Die Methode **`remove()`** löscht das als Parameter übergebene Entity (z.B. config).

```
// Location eines beobachtbaren Objekts löschen
TblLocation location : observerobject.getLocation();
this.utx.begin();
this.em.remove(location);
this.utx.commit();
```

JPA – Entity Zustände

- Entities nehmen nach JPA-Spezifikation **vier verschiedene Zustände** an:
  1. **Transient**

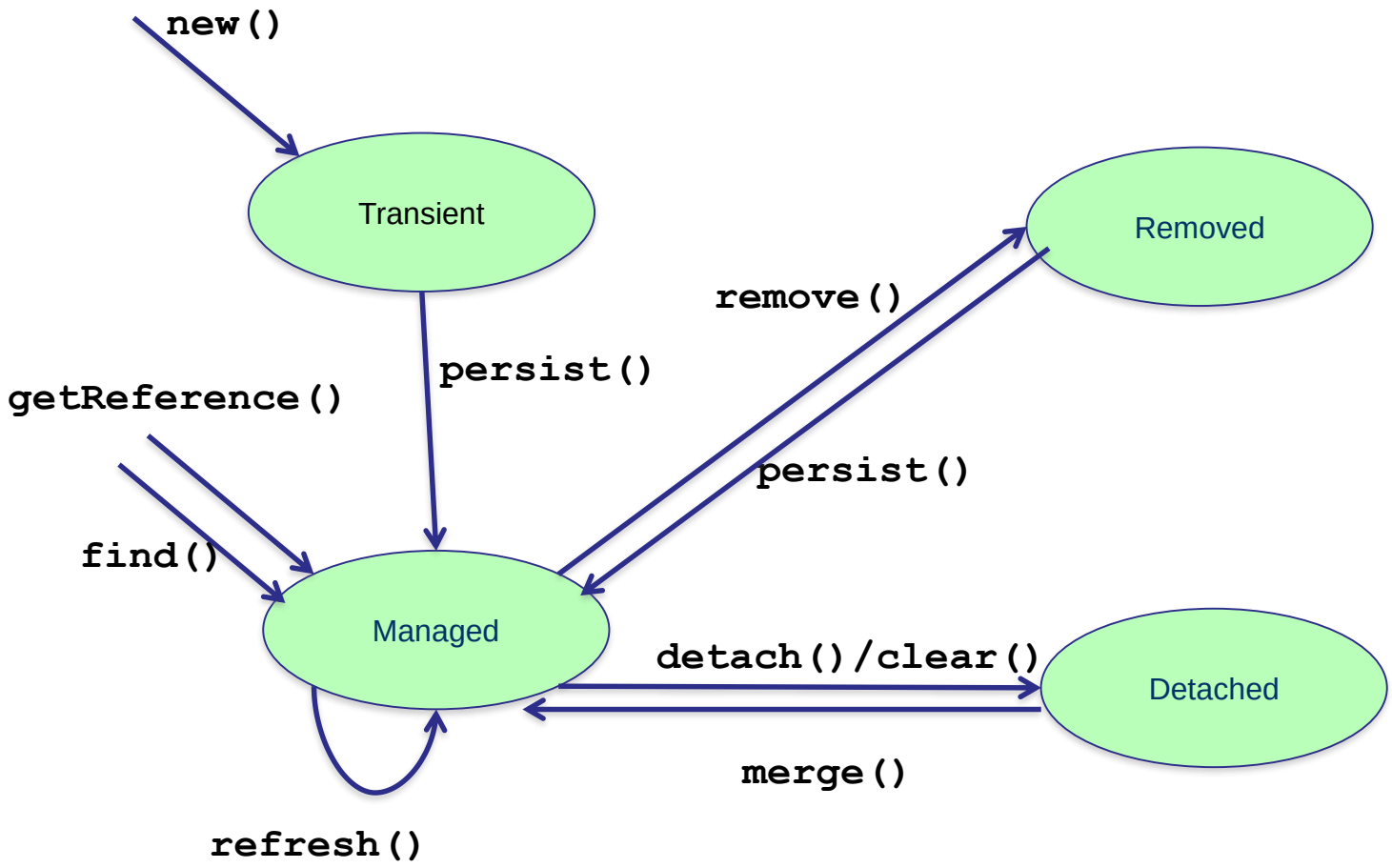
     Objekt noch nicht an Entity-Manager übergeben, noch kein Äquivalent in der DB
  2. **Managed**

     Objekt unter Kontrolle des Entity-Managers
  3. **Detached**

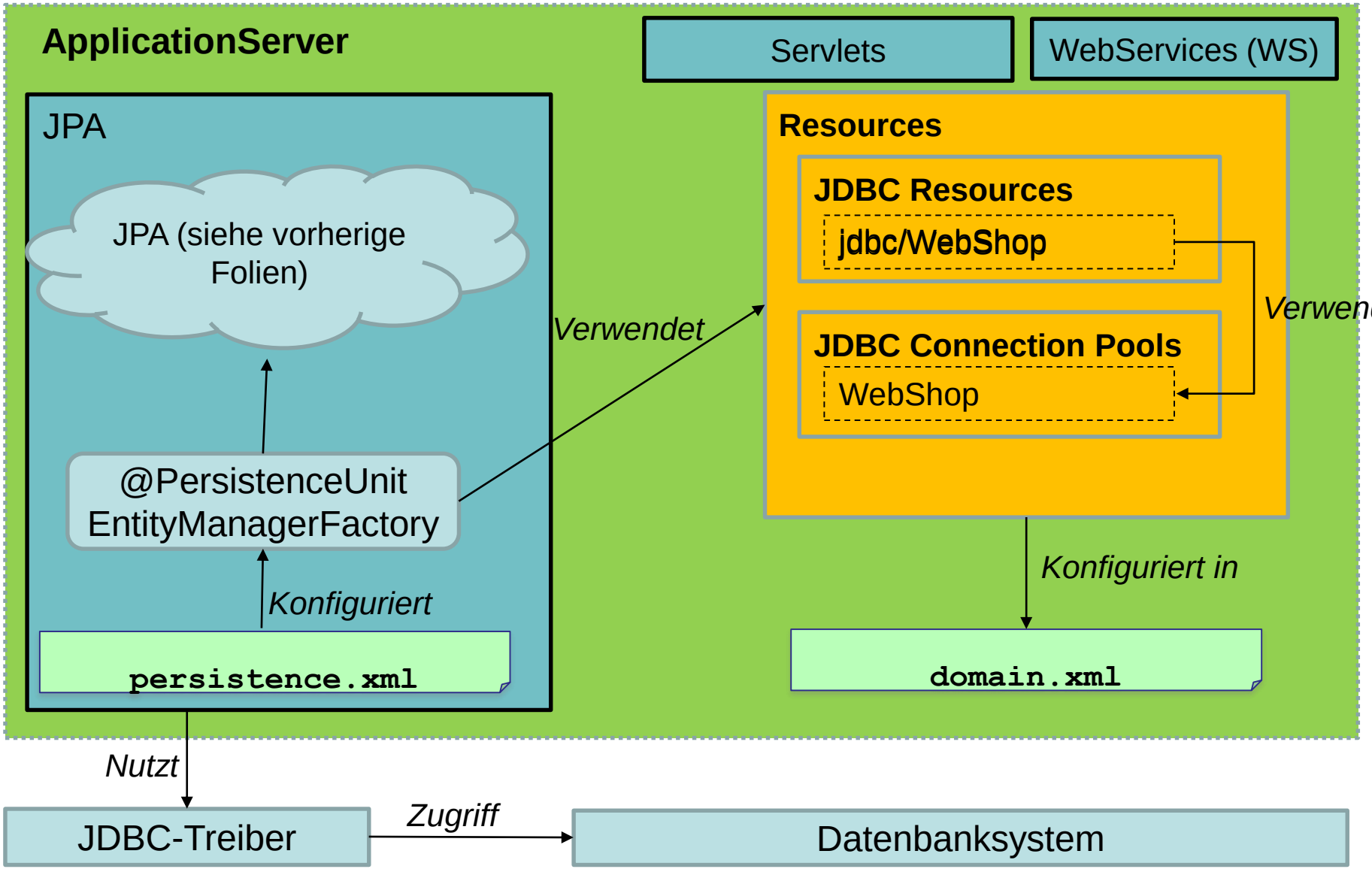
     Objekt besitzt Äquivalent in der DB ist aber aktuell nicht unter Kontrolle des Entity-Managers
  4. **Removed**

     Objekt unter Kontrolle des Entity-Mangers, in der DB gespeichert, aber ***zum Löschen vorgemerkt***
- Die Entity-Manager-API besitzt Methoden die die Zustandübergänge einleiten. z.B. **`persist(), remove(), detach(), merge(),find(), getReference()refresh()`**

JPA – Objektlebenszyklus eines Entity



JPA im ApplicationServer



JPA im ApplicationServer

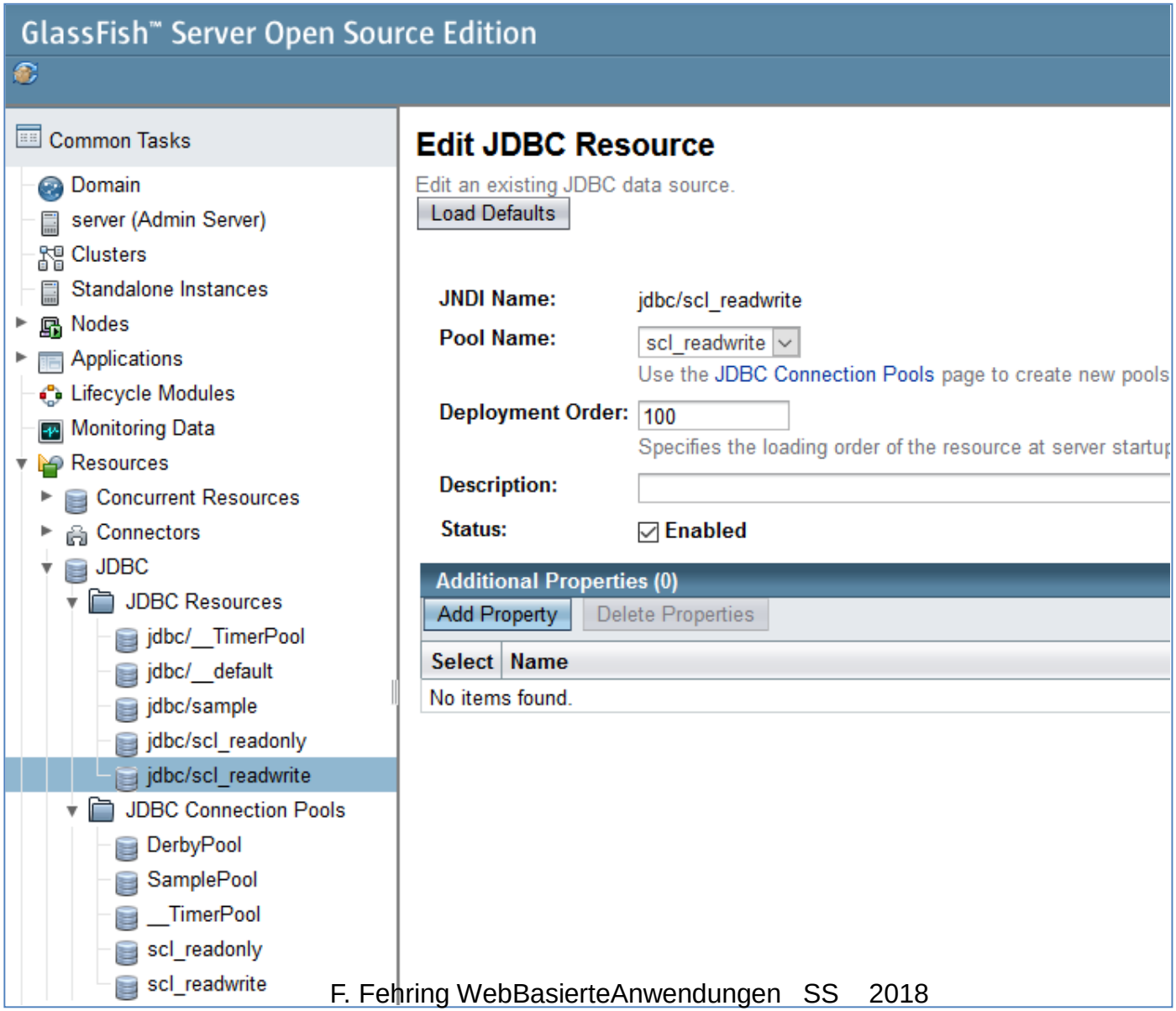
Eigenschaften

- Datenbank-Verbindungen werden in ConnectionPools vom ApplicationServer verwaltet
- ConnectionPools werden von (beliebig vielen) JDBC Resourcen verwendet
- JDBC Resourcen werden von (beliebig vielen) Anwendungen verwendet
- Erleichtert die Wartung: Passwort muss nur an einer Stelle für alle Anwendungen geändert werden
- Erhöht die Sicherheit: Zugangsdaten müssen nicht mehr im Quelltext oder in einer Konfigurationsdatei gelagert werden

Einrichtungsschritte

1. Connection-Pool anlegen
   Eintragen der Datenbank-Zugangsdaten. (URL, Benutzer, Passwort)
2. JDBC-Ressource anlegen
   Auswählen des Connection-Pools, der die Ressource mit Verbindungen versorgt
3. Ressourcen-Verwendung in der Applikation festlegen
   Eine persistence-unit in der persistence.xml mit Angabe des Namens der JDBC Ressource anlegen

JPA – domain.xml



Einstellungen

- Über GUI des AppServers
- Oder domain.xml

- Hinweis:
- PayaraFish-Server Version 4.1.1.171 verwenden!
- https://www.payara.fish/previous_releases

JPA – persistence.xml

Eigenschaften

- Definiert mindestens eine PersistenceUnit und legt ihre Einstellungen fest
  - Name der PersistenceUnit durch Attribute „name“
  - Typ des Transaktionsmanagements durch Attribute „transaction-type“ (oft JTA)
  - Datenquelle als Name einer vom ApplicationServer gestellten JDBC-Resource
  - Weitere Optionen: z.B. autom. Verwendung aller @Entity annotierten Klassen

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name=„WebShopPU" transaction-type="JTA">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <jta-data-source>jdbc/localhost</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <shared-cache-mode>NONE</shared-cache-mode>
    <properties>
      <!—Hier müssen Hibernate-Properties stehen -->
    </properties>
  </persistence-unit>
</persistence>
```

Java Persistence Query Language

Eigenschaften

- Definiert Abfragen auf Entitäten
- SQL ähnlicher Syntax, aber vom konkreten DB System unabhängig
- Werden von JPA in DB System abhängige Abfragen übersetzt
- Verwendet (sicherere) prepared Statements
- Vermeiden oft JOIN-Statements durch einfachere Objektnavigation
- Verwendet Objekt-Notationen anstelle von Tabellen/Spalten-Notationen

```
// Location eines beobachtbaren Objekts löschen
String jpql = "SELECT k FROM Kunde k WHERE k.name='Mustermann'";
// Ein TypedQuery erzeugen
TypedQuery query = this.em.createQuery(jpql, Kunde.class);
// Eine Liste von Kunden-Objekten mit dem Namen Mustermann
List<Kunde> kunden = query.getResultList();
```

JPQL- Named Queries

Eigenschaften

- Werden bei den Entity-Klassen definiert
- Besitzen einen eindeutigen Namen
- Können auf Attribute zugreifen
- Können vom Entity-Manager aufgerufen werden
- Erleichtern die Wiederverwendung
- Können Platzhalter für Parameter beinhalten

```
@Entity
@Table(name = „kunde")
@NamedQueries({
 @NamedQuery(    name="Kunde.findAll",
         query="SELECT k FROM Kunde k"),
 @NamedQuery(    name="Kunde.findById",
         query="SELECT k FROM Kunde k WHERE k.id = :id")
})
public class Kunde implements Serializable { …
```

JPQL- Named Queries

Verwendung von NamedQueries

- Zugriff über den EntityManager
- Typisierte Anfragen den untypisierten vorziehen
  - Aus typisierten Anfragen kommen Objekte der entsprechenden Klassen
  - Aus untypisierten Anfragen kommen Objekte vom Typ java.lang.Object
- **`setParameter`**()
  - Setzt den Wert für einen Parameter des NamedQueries
  - Akzeptiert einfache Datentypen und Objekte
- **`getSingleResult()`**
  - Liefert ein einzelnes Objekt
- **`getResultList()`**
  - liefert eine Liste vom Typ **`java.util.List`**

```
Query query = this.em.createNamedQuery("Kunde.findById", Kunde.class);
query.setParameter("id", 1);
Kunde k = query.getSingleResult();
```

JPQL- Native Queries

Eigenschaften

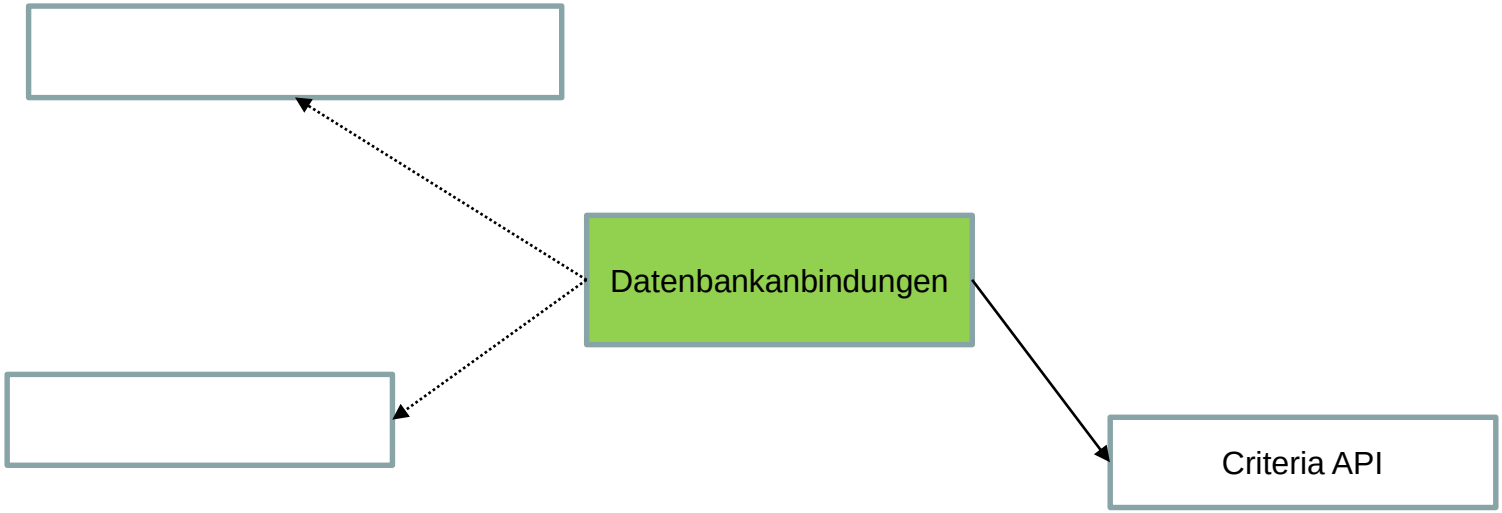
- Zugriff über den EntityManager
- Standard SQL
- Anfragen sind DB-System spezifisch
- Sparsam einsetzen! Machen die Anwendung DB-Systemabhängig!
- Liefern eine Liste von java.lang.Object Objekten

```
String sql = ("SELECT * FROM " + "smartmonitoring.tbl_observedobject“
         + " ORDER BY id DESC;"
List<Object[]> paramList = new ArrayList<>();
paramList = this.em.createNativeQuery(sql).getResultList();
```

Datenbankanbindungen

1. Problemstellung und Motivation
2. Java Database Connectivity
3. Java Persistence API
4. **Darüber hinaus**
5. Projekt

Darüber hinaus



Links:

CriteriaAPI - https://docs.oracle.com/javaee/6/tutorial/doc/gjitv.html

Datenbankanbindungen

1. Problemstellung und Motivation
2. Java Database Connectivity
3. Java Persistence API
4. Darüber hinaus
5. **Projekt**

Anforderungen

Welche Anforderungen werden als nächstes bearbeitet?

TODO

- Daten aus Datenbank lesen
- Artikel speichern
- Kommunikation untereinander

DONE

- …
- Formular für Kommentare
- Schickes Design für die Seite
- Mediendatein einbinden
- Animationen
- Mehrsprachen-Fähigkeit
- (lokales) Speichern von Artikeln
- Client-Position anzeigen
- Offline-Verwendung ermöglichen
- Inhaltsverzeichnisse
- Formlareingaben in Seite einfügen
- Navigation über Tastaturkürzel
- Externe Inhalte einbinden
- Artikel vom Server einbinden
- Kommentare vom Server
- Medien hochladen
- Kommentare hochladen

Literatur:

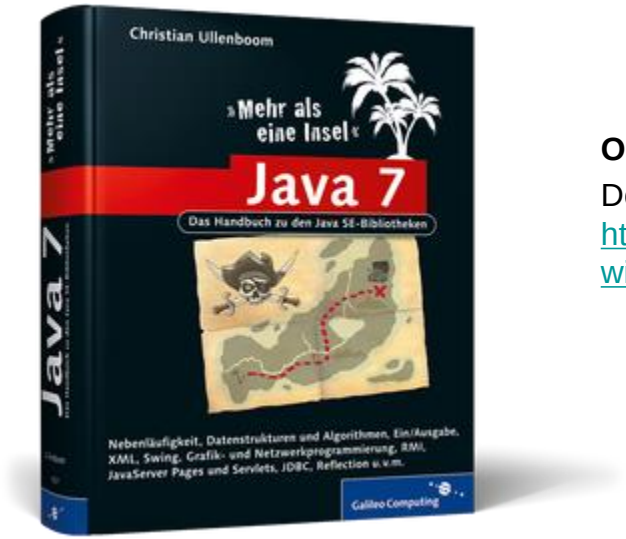
Melzer, Ingo et al. „Service-orientierte Architekturen mit Web Services“ Konzepte – Standards – Praxis 4. Auflage 2010, 381 Seiten, ISBN 978-3-8274-2549-2, Spektrum Akademischer Verlag über Springer Link

Christian Ullenboom: „Java 7 – Mehr als eine Insel

Das Handbuch zu den Java SE-Bibliotheken“

ISBN 978-3-8362-1507-7,

Rheinwerk Verlag 2012



Online-Quellen:

Dokumentation zu Jquery: https://learn.jquery.com/ajax/working-with-jsonp/