Hochschule der Medien
Stuttgart

Web Development 3

Datenbankzugriff

JEE: Java Persistence Architecture (JPA)

Prof. Dr.
Dirk Heuzeroth

JPA
MOTIVATION UND GRUNDLAGEN

Hintergrund und Historie (1)

- Ziel:
  - Speichern von Daten / Objekten aus Programmen in Datenbanksystemen.
- Proprietäre Datenbankzugriffslösungen gab es in Java Development Kit 1.0 (1996)
- Einbindung von JDBC in JDK 1.1 (1997)
  - Direkte Datenbankanbindung per SQL
  - Bei großen Anwendungen sehr umständlich
- Wunsch der einfachen Übertragung von Objekten in Relationen
- Mit der J2EE 1.0 (1999) werden Enterprise Java Beans (EJB) eingeführt
  - eine Bean bestand aus Klasse, Interfaces und XML-Beschreibungen
  - Keine Vererbung möglich
  - Daher lange Zeit sehr unpopulärer Standard

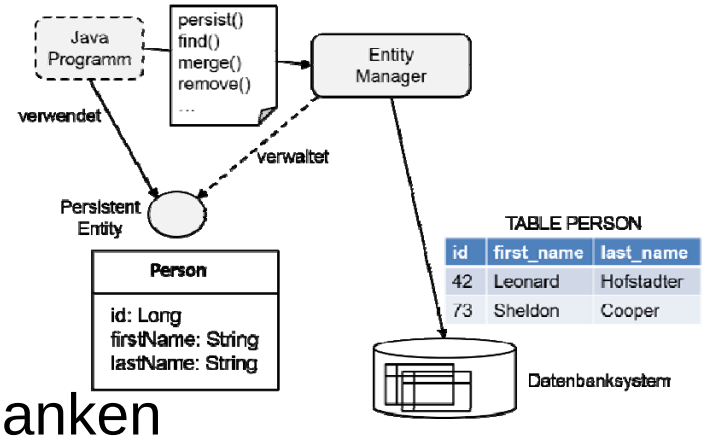
Hintergrund und Historie (2)

- Sun verabschiedet 2001 die Spezifikation der Java Data Objects (JDO, inzwischen von der Apache Foundation geführt)
  - Erreichte trotz sinnvoller Ansätze keine große Akzeptanz in der Community
- Mit Java EE 5 (2006) und EJB 3.0 wird die Java Persistence API spezifiziert
  - Einfache Persistenzschnittstelle
  - Unter Einfluss von JDO und Hibernate entwickelt
  - Eine Komponente = eine einfache Java Klasse
- Seit 2009 existiert JPA in Version 2.0

Begriffe

- POJO
  - Plain Old Java Object
  - Jedes „normale“ Objekt in Java...
- Entity („Informationsobjekt“)
  - Eindeutig zu bestimmendes Objekt, über das Informationen in die Datenbank gespeichert werden sollen
  - persistenzfähig
    - In diesem Zusammenhang: für JPA vorbereitetes POJO
- Entity Bean
  - Begrifflichkeit aus dem EJB Standard
- Persistence Unit
  - Menge von Entitäten, die gemeinsam in einer Datenbank verwaltet werden

JPA-Ablauf (1)



- Persistente Datenspeicherung in Datenbanken
  - Daten, die dauerhaft gespeichert werden müssen, werden i.d.R. in Datenbanken abgelegt.
  - Am häufigsten werden relationale Datenbanken eingesetzt.
    - Diese speichern Datensätze in Tabellen.
    - Die Tabellenspalten definieren die einzelnen Bestandteile (Attribute) eines Datensatzes.
    - Die Tabellenzeilen repräsentieren jeweils einen Datensatz.
- Persistente Objekte in Programmen
  - Java-Programme arbeiten mit Objekten.
  - Objekte, die persistent in der Datenbank gespeichert werden sollen, werden in JEE mittels des JPA (Java Persistence API) Entity Managers verwaltet.
  - Der Entity Manager
    - vermittelt zwischen der Java-Welt (Objekte) und der Datenbankwelt (Tabellen).
    - übersetzt Aufrufe der Methoden `persist()`, `find()`, `merge()`, `remove()` etc. in SQL-Anweisungen an die Datenbank.

JPA-Ablauf (2)

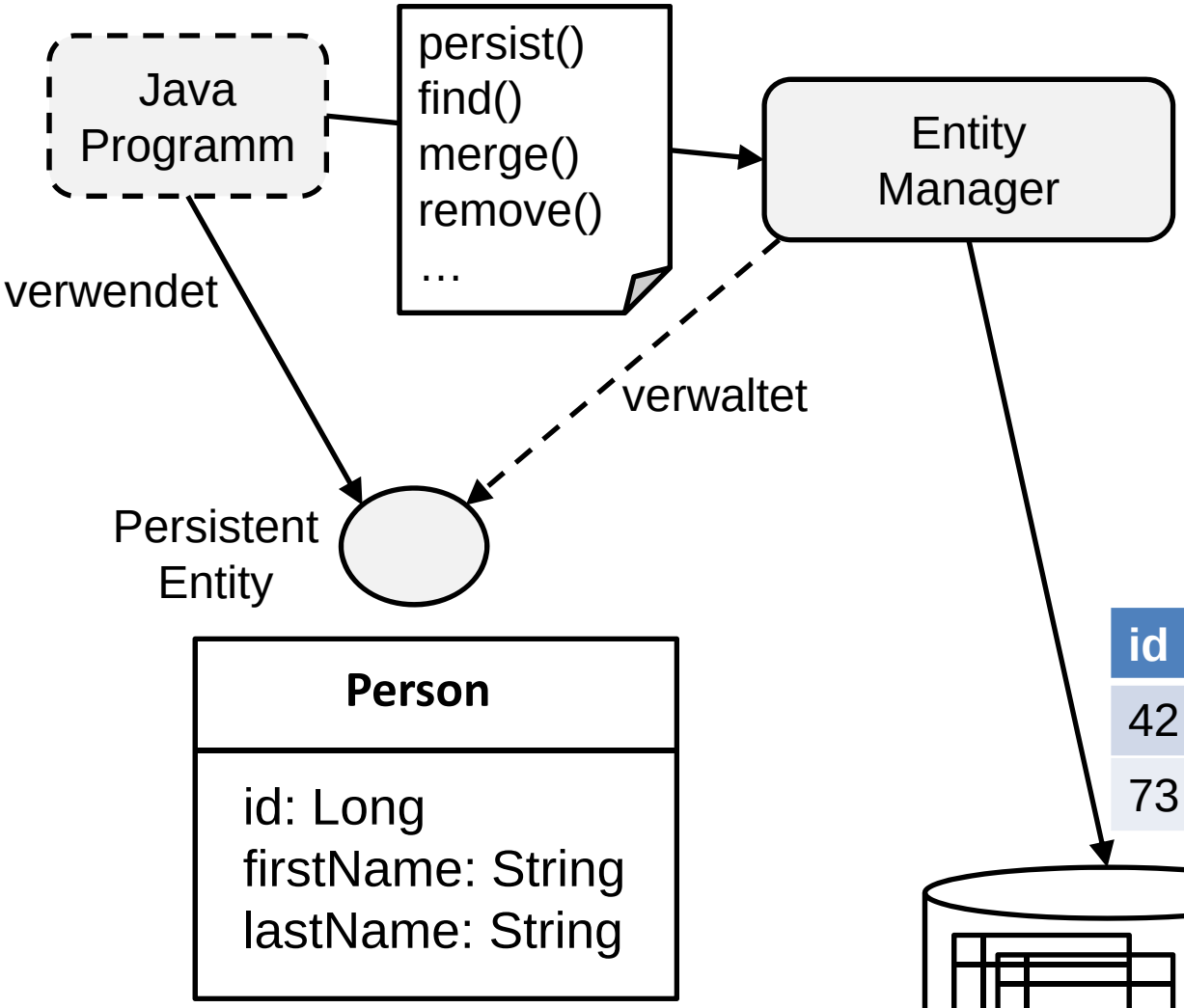
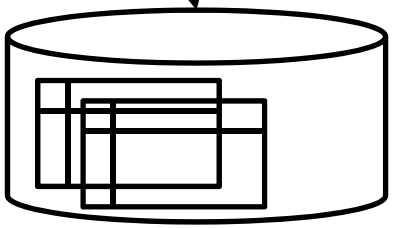


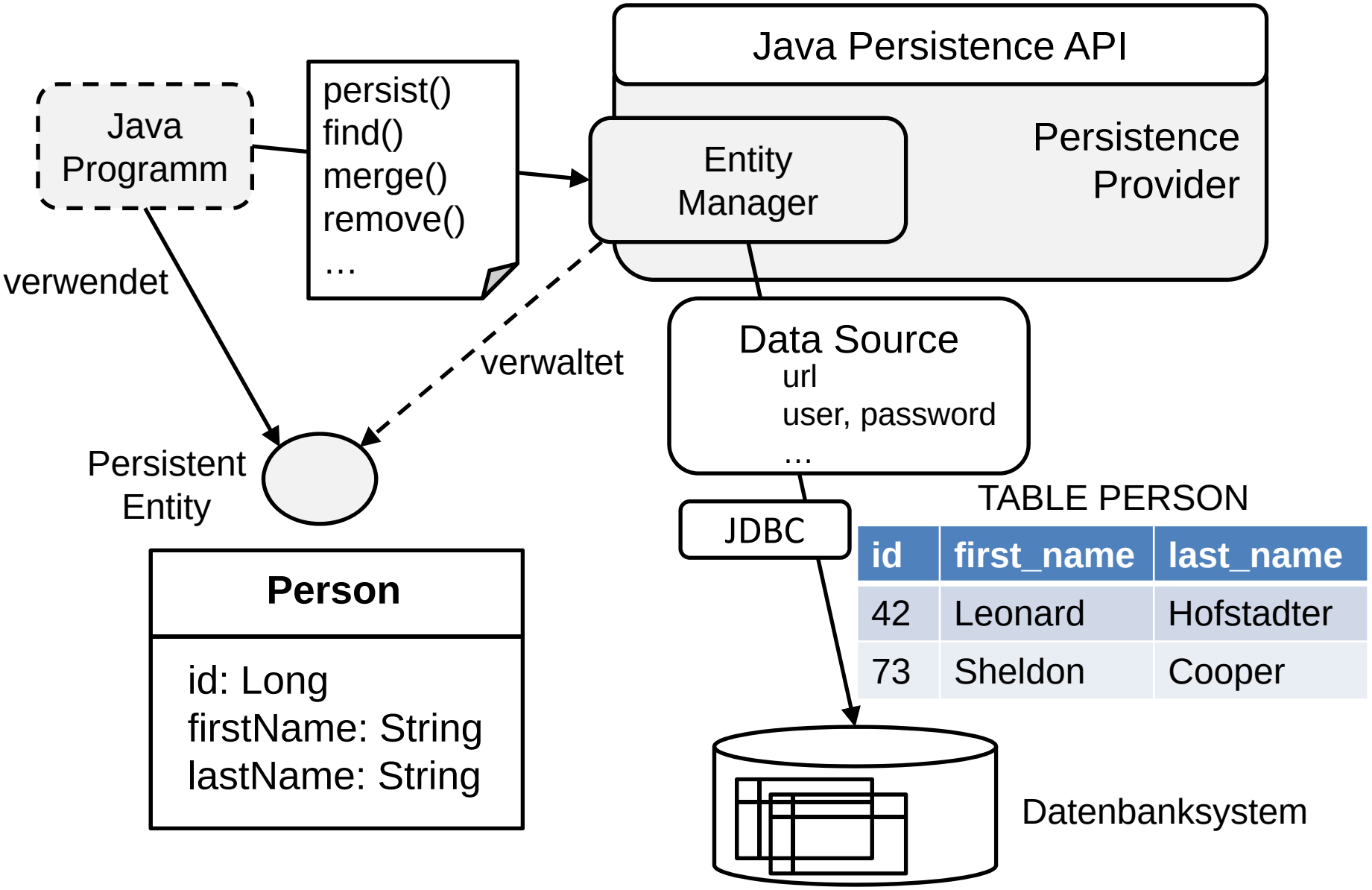
TABLE PERSON

id	first_name	last_name
42	Leonard	Hofstadter
73	Sheldon	Cooper



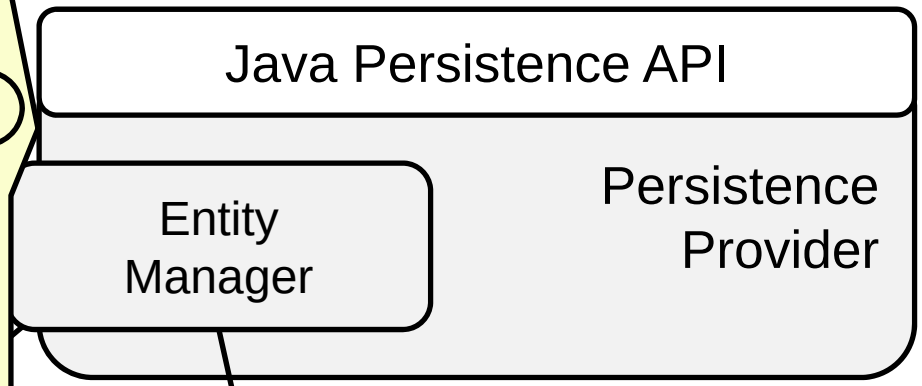
Datenbanksystem

JPA Architektur



id	first_name	last_name
42	Leonard	Hofstadter
73	Sheldon	Cooper

- Entity Manager ist als Teil des Persistence Providers zentrale Instanz für das Persistieren von Objekten.
- Persistence Provider kümmert sich um das "objekt-relationale Mapping", d.h. das Abbilden von Objekten auf Relationen (Tabellen).
- Java Persistence API ist einheitliche Schnittstelle zum Ansprechen / Einbinden existierender Persistence Provider wie z.B. Hibernate



3

et

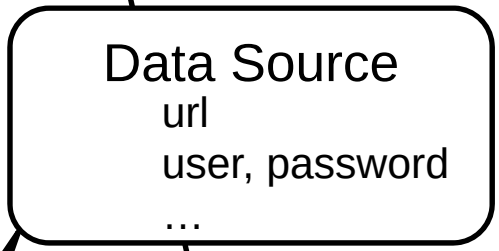


TABLE PERSON

id	first_name	last_name
42	Leonard	Hofstadter

- Enthält alle Informationen, um auf eine konkrete Datenbank zugreifen zu können.
- Wird mit symbolischen Namen im JNDI angemeldet.
- Ermöglicht dem Application Server Datenbankverbindungen wiederzuverwenden – das nennt man Connection Pooling.

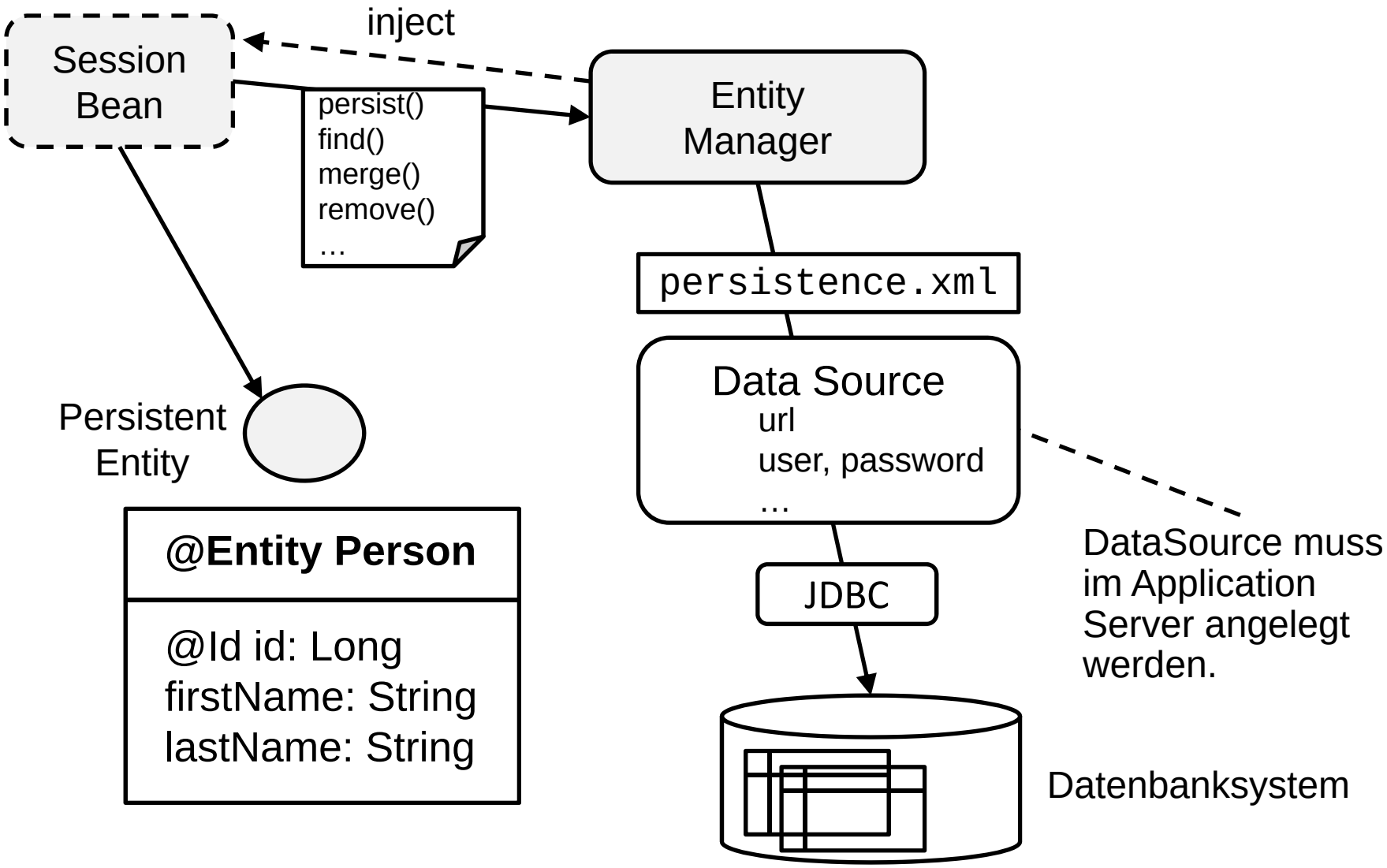
2

1

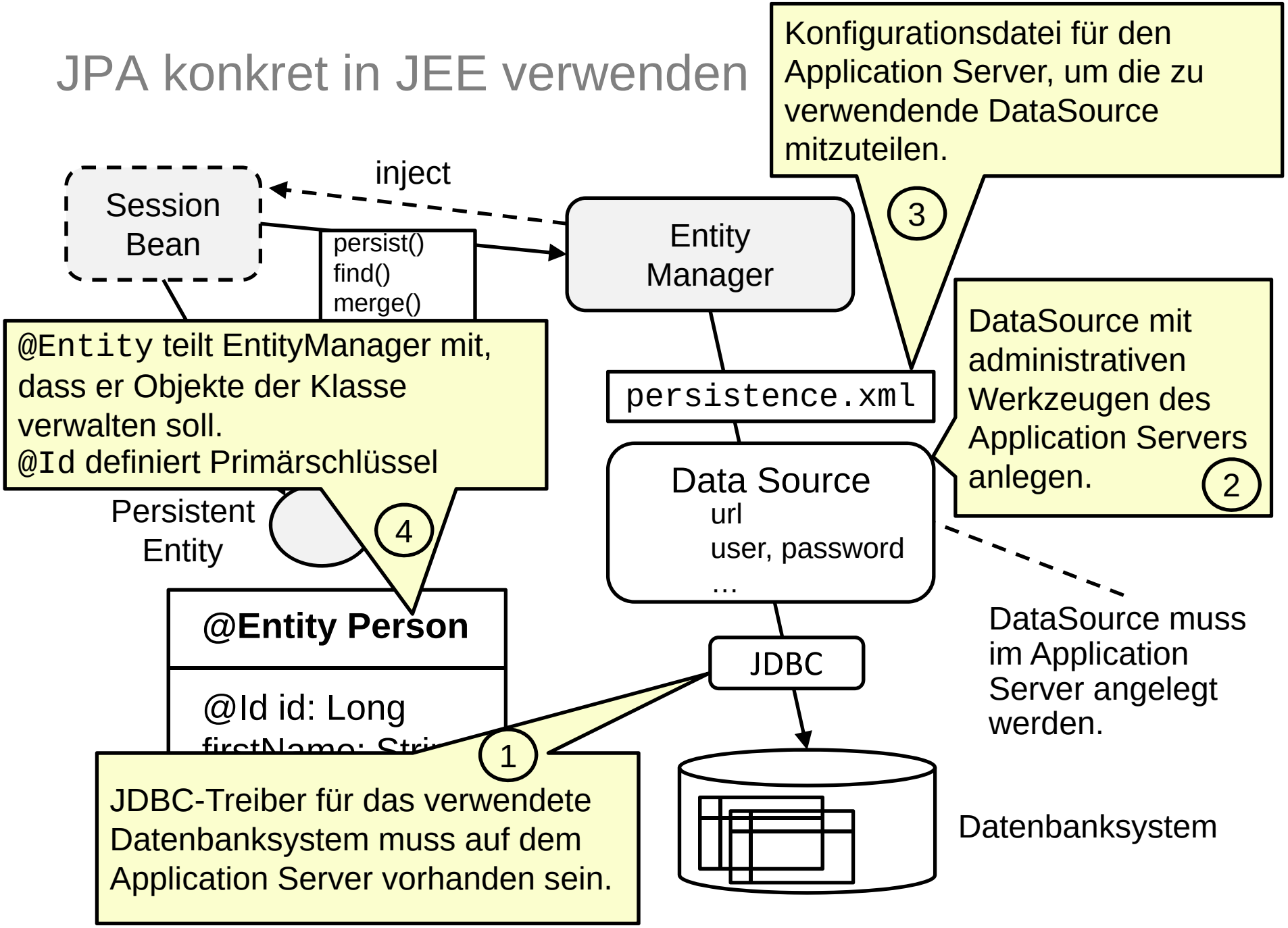
JDBC = Java Database Connectivity:
Menge von Java Klassen, die von dem konkreten Datenbanksystem abstrahieren und dem Aufrufer eine standardisierte Schnittstelle zur Verfügung stellen.

JPA
PRAKTISCH VERWENDEN

JPA konkret in JEE verwenden



JPA konkret in JEE verwenden



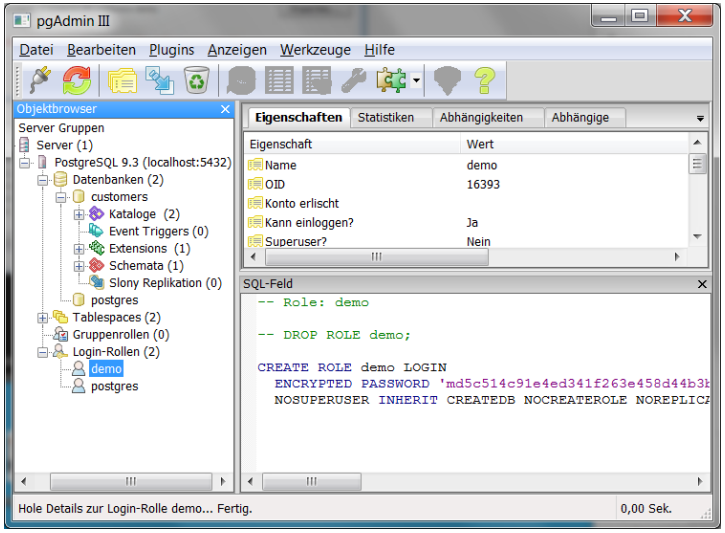
VORBEREITENDE INSTALLATIONEN UND KONFIGURATIONEN

Vorbereitungen:
Datenbank installieren und konfigurieren (1)

- Die Beispiele verwenden PostgreSQL
- Download
  - http://www.enterprisedb.com/products-services-training/pgdownload
- Installationsprogramm ausführen
- Stack Builder installieren und ausführen
  - Zusätzlich das JDBC Treiberpaket installieren.
- JDBC-Treiber `postgresql-9.3-1100.jdbc4.jar` aus dem PostgreSQL-Installationsverzeichnis "`C:\Program Files (x86)\PostgreSQL\pgJDBC`" in das `lib`-Verzeichnis von TomEE plus kopieren.

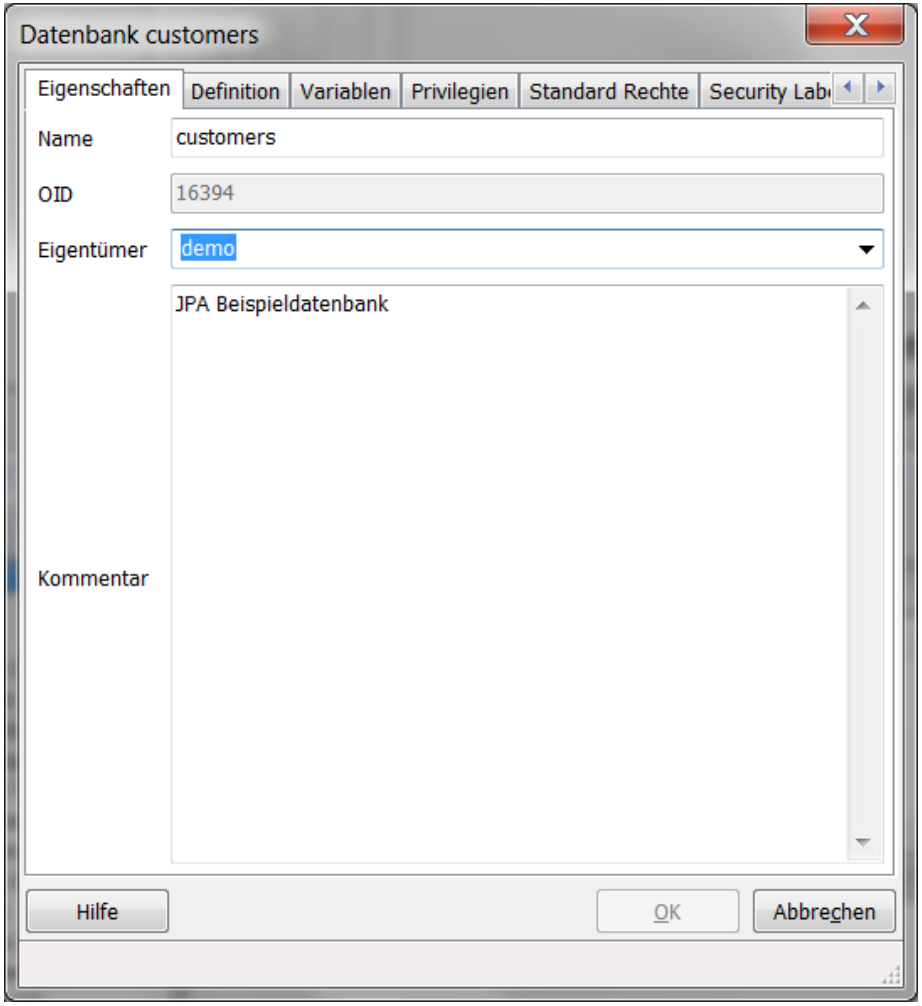
Vorbereitungen:
Datenbank installieren und konfigurieren (2)

- `pgAdmin III` starten
- Neue Login-Rolle "demo" durch erstellen
  - Rechtklick auf "Login-Rollen"
  - Dann "Neue Login-Rolle…" anklicken
  - Name: demo
  - Reiter "Definition" auswählen und als Passwort "demo" eingeben
  - Im Reiter "Rollenprivilegien" die Checkbox "Can create databases" anklicken.



Vorbereitungen:
Datenbank installieren und konfigurieren (2)

- Optional
  neue Datenbank erstellen:
  - Rechtsklick auf Datenbanken
  - Dann "Neue Datenbank…" anklicken
  - Name: customers
  - Eigentümer: demo
- Statt die Datenbank so manuell zu erstellen, kann man sie auch vom O/R-Mapper erstellen lassen (siehe spätere Folien in diesem Foliensatz).



Entscheidung für einen Persistence Provider

- Es gibt mehrere Persistence Provider Produkte – auch O/R-Mapper (Object-Relation Mapper / ORM) genannt – d.h. Produkte die die Abbildung von Objekten auf Tabellen einer relationalen Datenbank realisieren.
- TopLink
  - http://www.oracle.com/technology/products/ias/toplink/index.html
  - War mal ein StartUp, das dann von Oracle gekauft wurde.
  - Kostenlos verfügbar, Relativ ausgereift
  - Referenzimplementierung für JPA 1.0
- EclipseLink
  - OpenSource ORM von der Eclipse Foundation
  - Standardmäßig in Eclipse integriert.
  - Basiert auf TopLink
  - Referenzimplementierung für JPA 2.0
  - Siehe auch den Abschnitt "JPA und Eclipse" in diesem Foliensatz.
- Hibernate
  - http://www.hibernate.org/
  - Quasi-Standard-O/R-Mapper
  - OpenSource ORM von JBoss / Red Hat
  - Großer Funktionsumfang: Implementiert JPA und JDO
  - Sehr ausgereift
- OpenJPA
  - http://stackoverflow.com/questions/8352742/tomee-plus-and-jpa
- Wir verwenden hier: Hibernate

Hibernate als OR-Mapper (1)

- Herunterladen von Hibernate von
  - http://www.hibernate.org
  - Konkret:
    - http://sourceforge.net/projects/hibernate/files/hibernate4/4.2.12.Final/hibernate-release-4.2.12.Final.zip/download
    - **Achtung:**
      - Die aktuelle Version 4.3.5 funktioniert nicht mit TomEE 1.6.
- Herunterladen von Hibernate Validator
  - http://sourceforge.net/projects/hibernate/files/hibernate-validator/4.3.1.Final/hibernate-validator-4.3.1.Final-dist.zip/download
  - **Achtung:**
    - Die aktuelle Version 5.1.0 funktioniert nicht mit TomEE 1.6.

Hibernate als OR-Mapper (2)

- Installation
  - Pakete entpacken
  - Die JAR-Dateien aus den folgenden Verzeichnissen in das `lib`-Verzeichnis von TomEE kopieren:
    - `hibernate-release-4.2.12.Final\lib\required`
    - `hibernate-release-4.2.12.Final\lib\jpa`
    - `hibernate-validator-4.3.1.Final\dist`
  - Ergebnis:
    - <tomee-home>/lib/antlr-2.7.7.jar
    - <tomee-home>/lib/dom4j-1.6.1.jar
    - <tomee-home>/lib/hibernate-commons-annotations-4.0.2.Final.jar
    - <tomee-home>/lib/hibernate-core-4.2.12.Final.jar
    - <tomee-home>/lib/hibernate-entitymanager-4.2.12.Final.jar
    - <tomee-home>/lib/hibernate-validator-4.3.1.Final.jar
    - <tomee-home>/lib/jboss-logging-3.1.0.GA.jar
  - Weitere Informationen finden sich auf:
    - http://tomee.apache.org/tomee-and-hibernate.html
    - http://stackoverflow.com/questions/10852035/tomee-and-hibernate-dependencies

Hibernate als OR-Mapper (2)

- Hibernate braucht eine Verbindung zur Datenbank
- Diese kann erfolgen über eine
  - DataSource (Variante 1)
  - direkte JDBC-Verbindung (Variante 2)
- Wir verwenden hier nur die "saubere" Variante 1.

DataSource anlegen

- TomEE-Konfigurationsdateien (tomee.xml, logging.properties, system.properties) in Eclipse importieren:
  - http://tomee.apache.org/tomee-and-eclipse.html
- DateSource wird für TomEE konfiguriert in `conf\tomee.xml`:

```
<Resource id="CustomerDS" type="DataSource">
    JdbcDriver          org.postgresql.Driver
    JdbcUrl             jdbc:postgresql://localhost:5432/customers
    UserName            demo
    Password            demo
    JtaManaged          true
    DefaultAutoCommit   false
</Resource>
```

- In diesem Fall
  - wird PostgreSQL als Datenbanksystem verwendet
  - und eine Verbindung zur Datenbank `customers` eingerichtet.
- Weitere Informationen zu Ressourcen in TomEE siehe:
  - http://tomee.apache.org/containers-and-resources.html
  - http://tomee.apache.org/configuring-datasources.html

Hibernate mit DataSource in JPA verwenden

- Verwendung von Hibernate mit dieser DataSource wird definiert in: `persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="JPAWebApp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>CustomerDS</jta-data-source>
    <non-jta-data-source>CustomerDS</non-jta-data-source>

    <class>de.webapps.Customer</class>

    <properties>
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />

      <!-- Drop and re-create the database schema on startup -->
      <property name="hibernate.hbm2ddl.auto" value="create" />
      <!-- update: the database schema will not be created if exists -->
      <!-- property name="hibernate.hbm2ddl.auto" value="update"/ -->
    </properties>
  </persistence-unit>
</persistence>
```

Variante 2: Hibernate mit JDBC in JPA verwenden

- `persistence.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
       xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="JPAWebApp" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>de.webapps.Customer</class>

    <properties>
      <!-- SQL dialect -->
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />

      <property name="hibernate.connection.driver_class"
                value="org.postgresql.Driver" />
      <property name="hibernate.connection.url"
                value="jdbc:postgresql://localhost:5432/customers" />
      <property name="hibernate.connection.username" value="demo" />
      <property name="hibernate.connection.password" value="demo" />

      <!-- Drop and re-create the database schema on startup -->
      <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
  </persistence-unit>
</persistence>
```

Strategie zur Tabellenerzeugung (Table Generation Strategy)

- Definiert, wie der Persistence Provider vorgehen soll, wenn er Tabellen benötigt, die noch nicht in der Datenbank vorhanden sind.
- Optionen
  - Create (create)
    - Fehlende Tabellen werden erzeugt.
    - Für den Produktivbetrieb NICHT geeignet.
  - Drop and Create (create-drop)
    - Bei jeder neuen Verbindung mit Datenbank, d.h. bei jedem neuen Deployment der Anwendung, werden die vorhandenen Tabellen zunächst gelöscht und dann alle Tabellen neu erzeugt.
    - Für den Produktivbetrieb NICHT geeignet.
  - None (update)
    - Nur auf bereits vorhandenen Tabellen arbeiten.
    - Für den Produktivbetrieb geeignet.
- Wird definiert in persistence.xml für Hibernate über die Property:
  - `hibernate.hbm2ddl.auto`

Definieren der zu persistierenden Klassen

- In der Datei `persistence.xml`
  über das Element `<class>`
  innerhalb des Elements `<persistence-unit>`
- Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
        xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="JPAWebApp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>de.webapps.Customer</class>
    ...
  </persistence-unit>
</persistence>
```

JPA-Konfiguration: `persistence.xml` (1)

- Kann in einem der folgender Verzeichnisse liegen:
  - `src/META-INF`
  - `WebContent/WEB-INF` (bei Web-Projekten)
  - `WebContent/META-INF` (bei sonstigen JEE-Projekten)
- Inhalt:
  - Persistenzeinheiten definiert durch `<persitence-unit>`-Elemente.
  - Angabe des Persistence Providers
  - Angabe der DataSource oder JDBC-Verbindung
  - Diverse Eigenschaften für Persistence Provider und Datenbankverbindungen.
  - Auf die Datenbank abzubildende Klassen.

JPA-Konfiguration: `persistence.xml` (2): Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
       http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="JPAWebApp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>CustomerDS</jta-data-source>
    <non-jta-data-source>CustomerDSNonJTA</non-jta-data-source>

    <!-- Classes that should be mapped to the database. -->
    <class>de.webapps.Customer</class>

    <properties>
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="cache.provider_class"
                value="org.hibernate.cache.NoCacheProvider" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

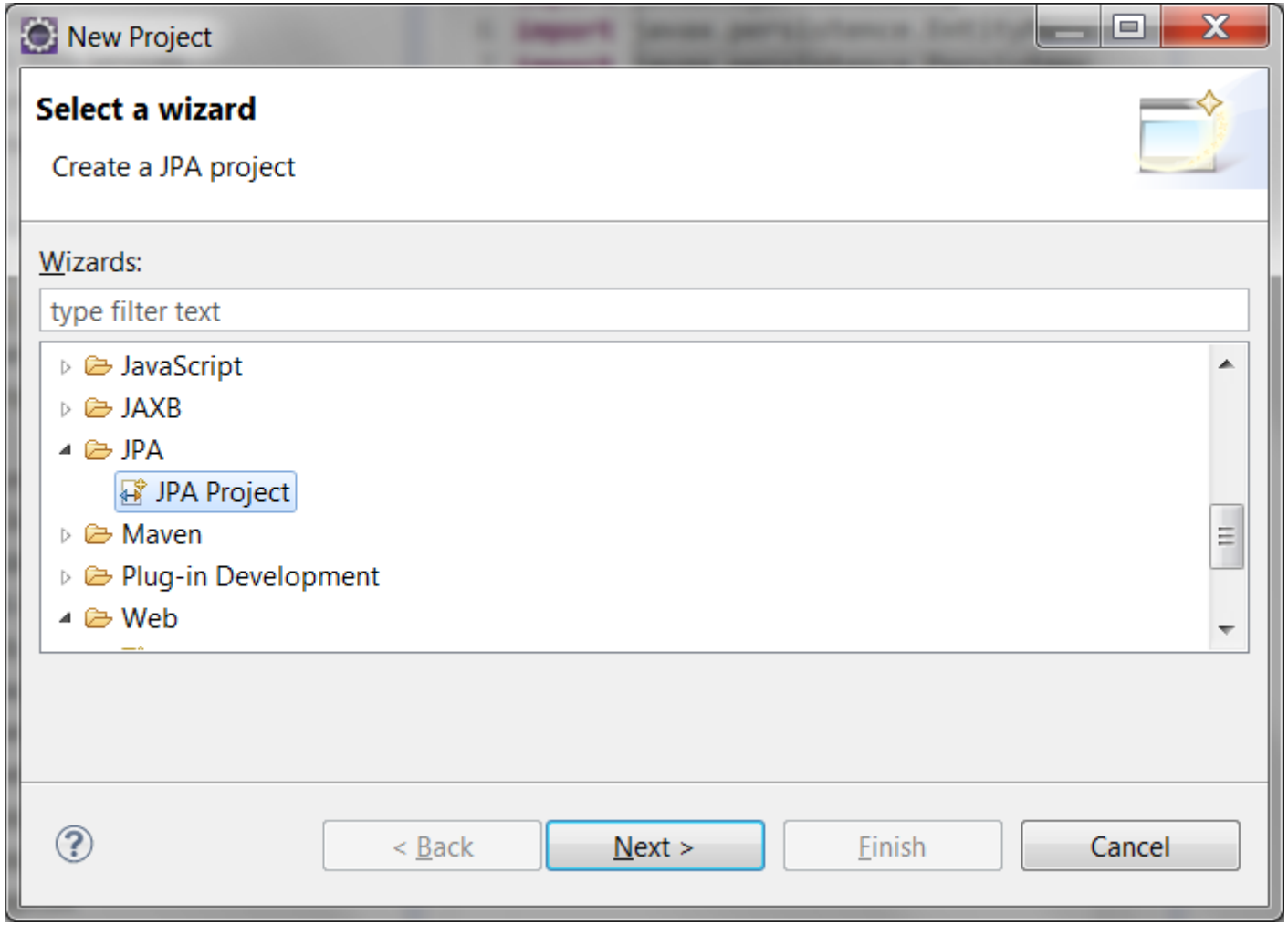
JPA UND ECLIPSE

JPA und Eclipse: Grundlagen

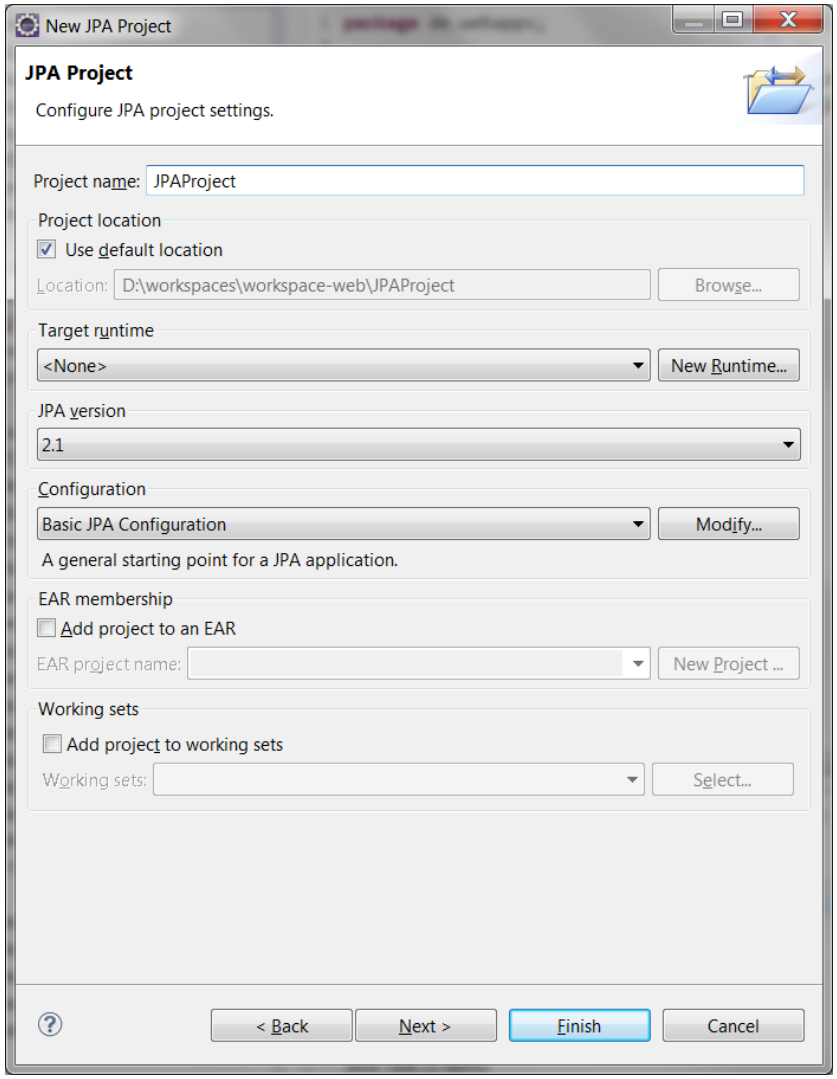
- Es gibt diverse Plugins für Eclipse, um mit JPA-Projekten zu arbeiten.
- In der "Eclipse for JEE Developers"-Version enthält bereits die wichtigsten Plugins.
- Die folgenden Folien zeigen beispielhaft
  - das Erstellen eines JPA-Projektes bzw. das Aktivieren der JPA Project Facet für ein Web-Projekt.
  - das Anlegen einer JPA Entity.

Neues JPA-Beispielprojekt erstellen (1)

- File → New → Project → JPA → JPA Project



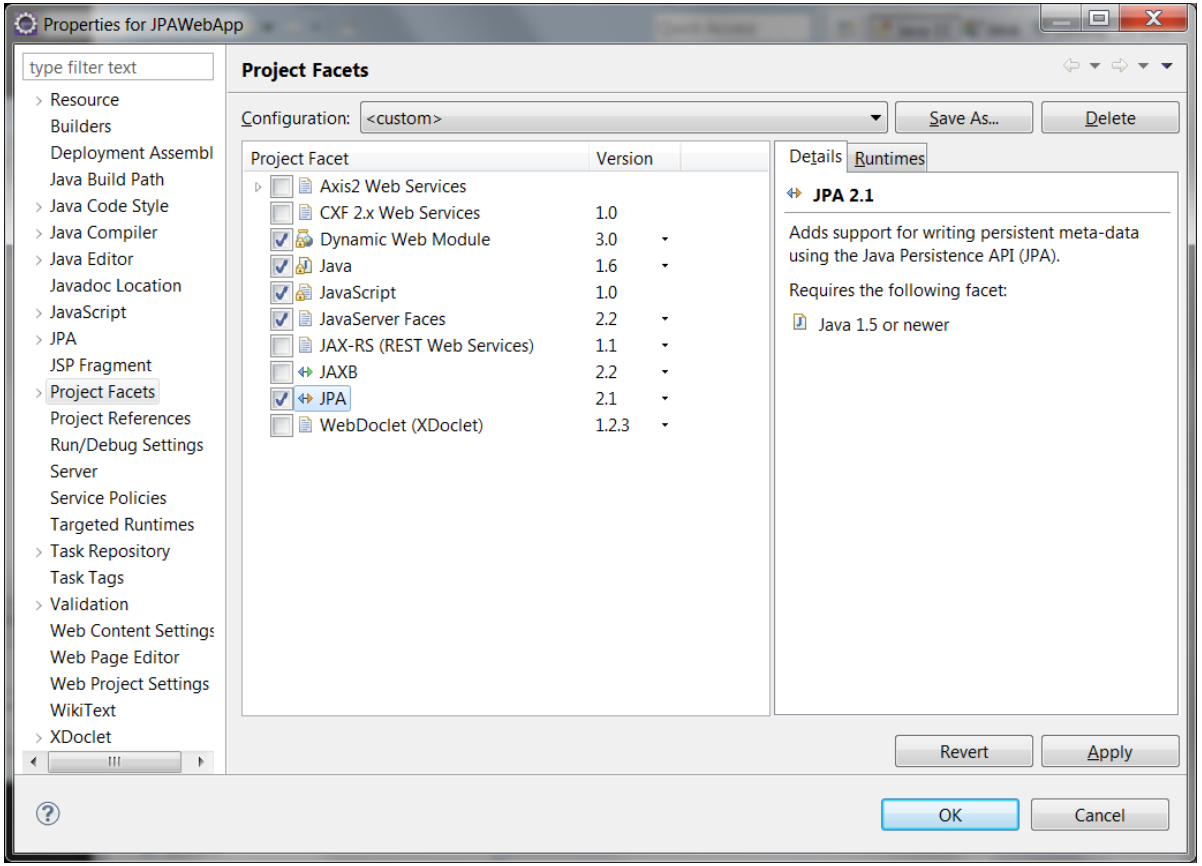
Neues JPA-Beispielprojekt erstellen (2)



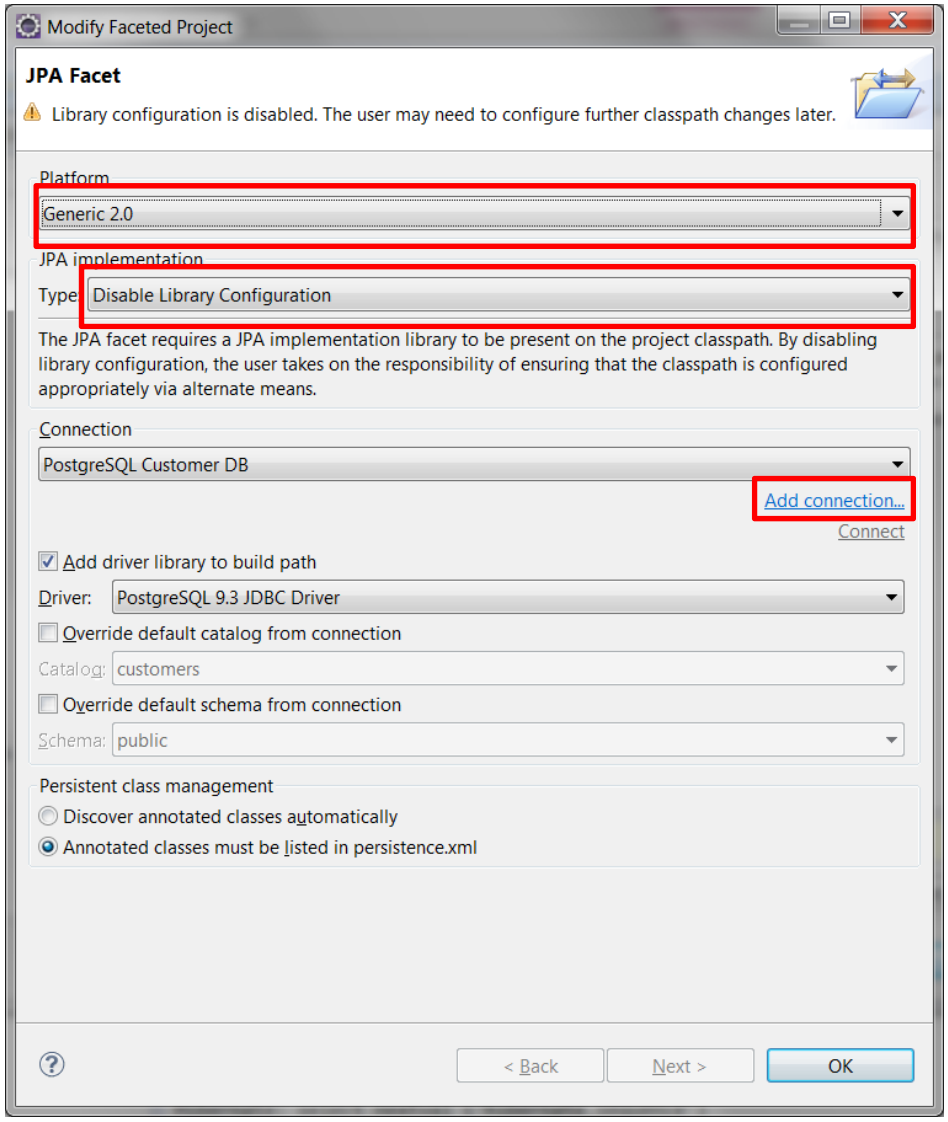
- Dann weiter bei Folie: "JPA Facet"

JPA in einem Web-Projekt aktivieren

- File → New → Project → Dynamic Web Project
- Rechtsklick auf das Projekt
  - Project Facets auswählen
  - JPA aktivieren

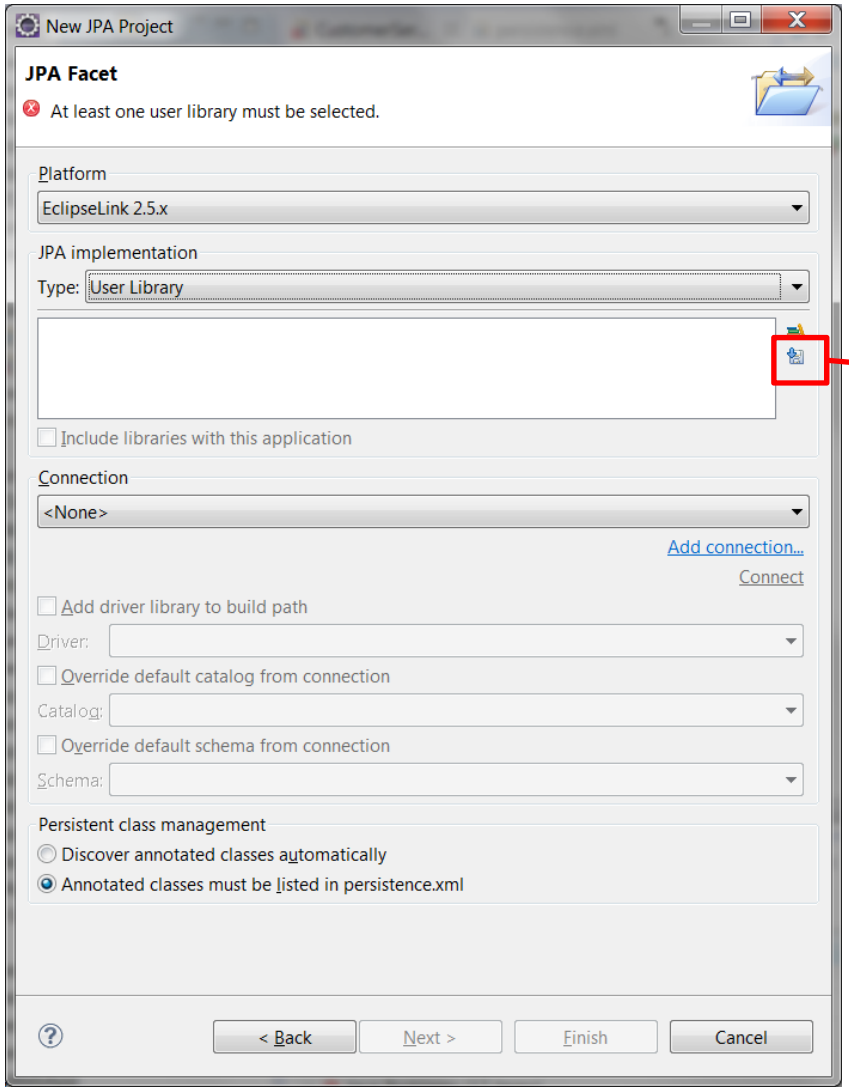


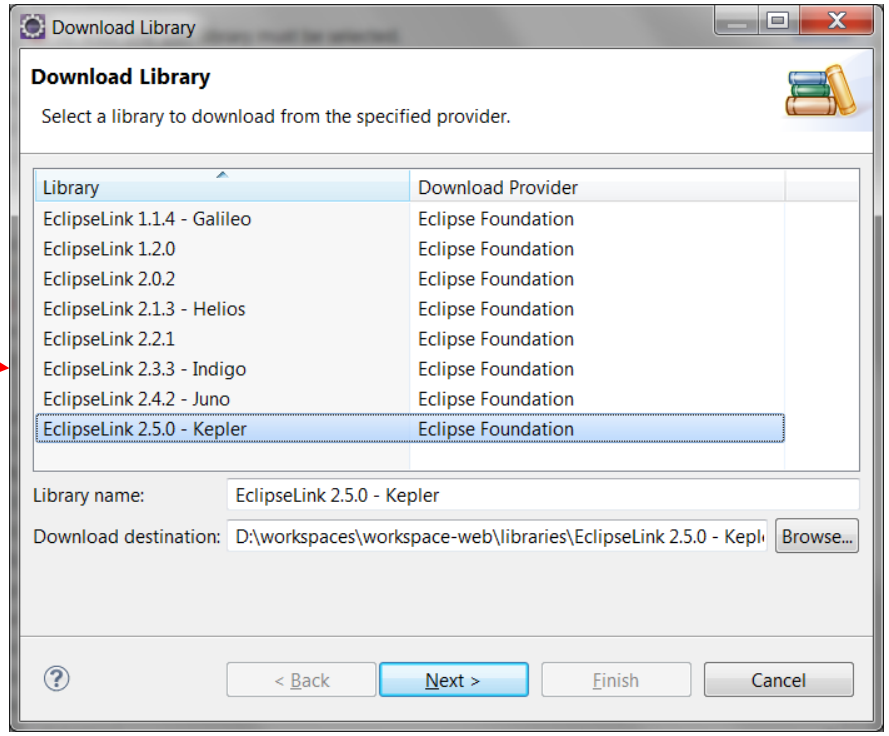
JPA Facet: Generisch



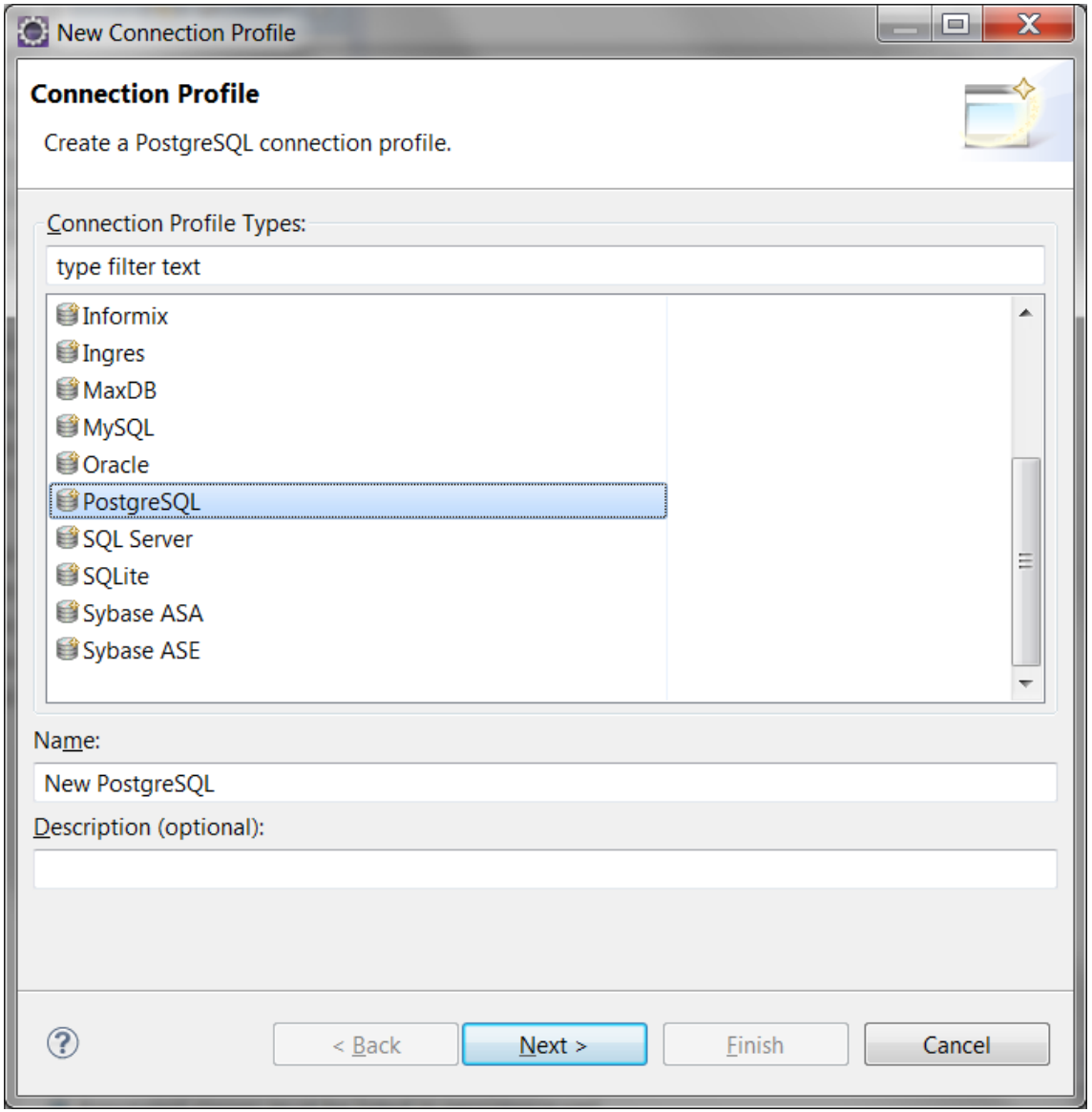
- Bei Plattform sollte man, wie hier gezeigt, Generic 2.0 auswählen.
  - Generic 2.1 wird von TomEE 1.6 noch nicht unterstützt.
  - Falls die Auswahl nicht möglich ist, in den Project Facets die JPA Version von 2.1 auf 2.0 setzen.
- Mit "Disable Library Configuration" angeben, dass die zugehörige JPA Implementierung auf dem Server vorhanden ist.
  - Das bedeutet, dass man die Klassen des Persistence Providers selbst zur Verfügung stellen muss, üblicherweise auf dem Application Server, bei uns also im lib-Verzeichnis des TomEE-Servers.
- Altenativ kann man als Plattform EclipseLink auswählen und über den Download der Bibliotheken als "User Library" einbinden (siehe nächste Folie).
- In beiden Fällen muss man noch die Datenbank Verbindung angeben: "Add Connection" (siehe übernächste Folie)

JPA Facet: EclipseLink

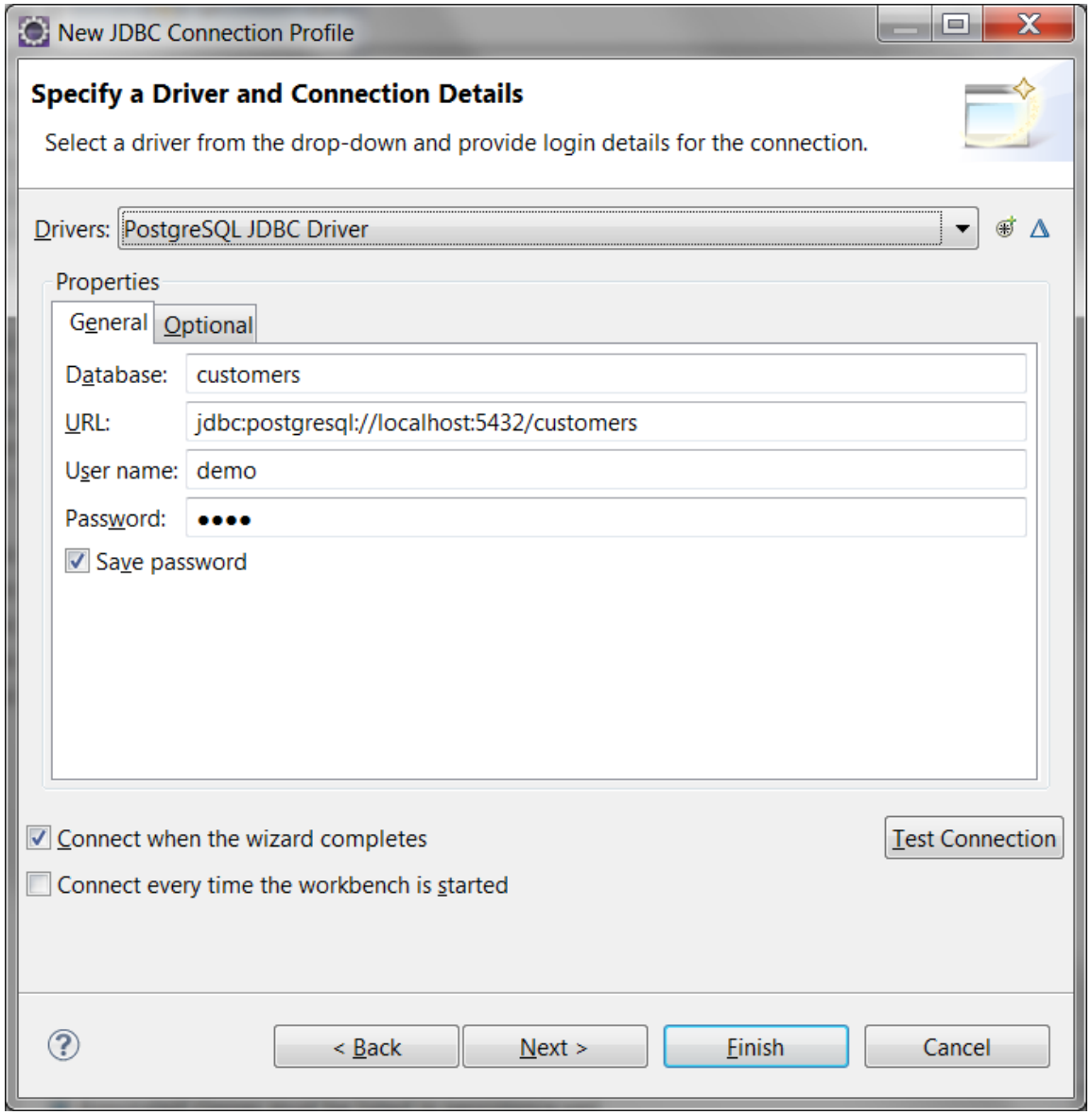




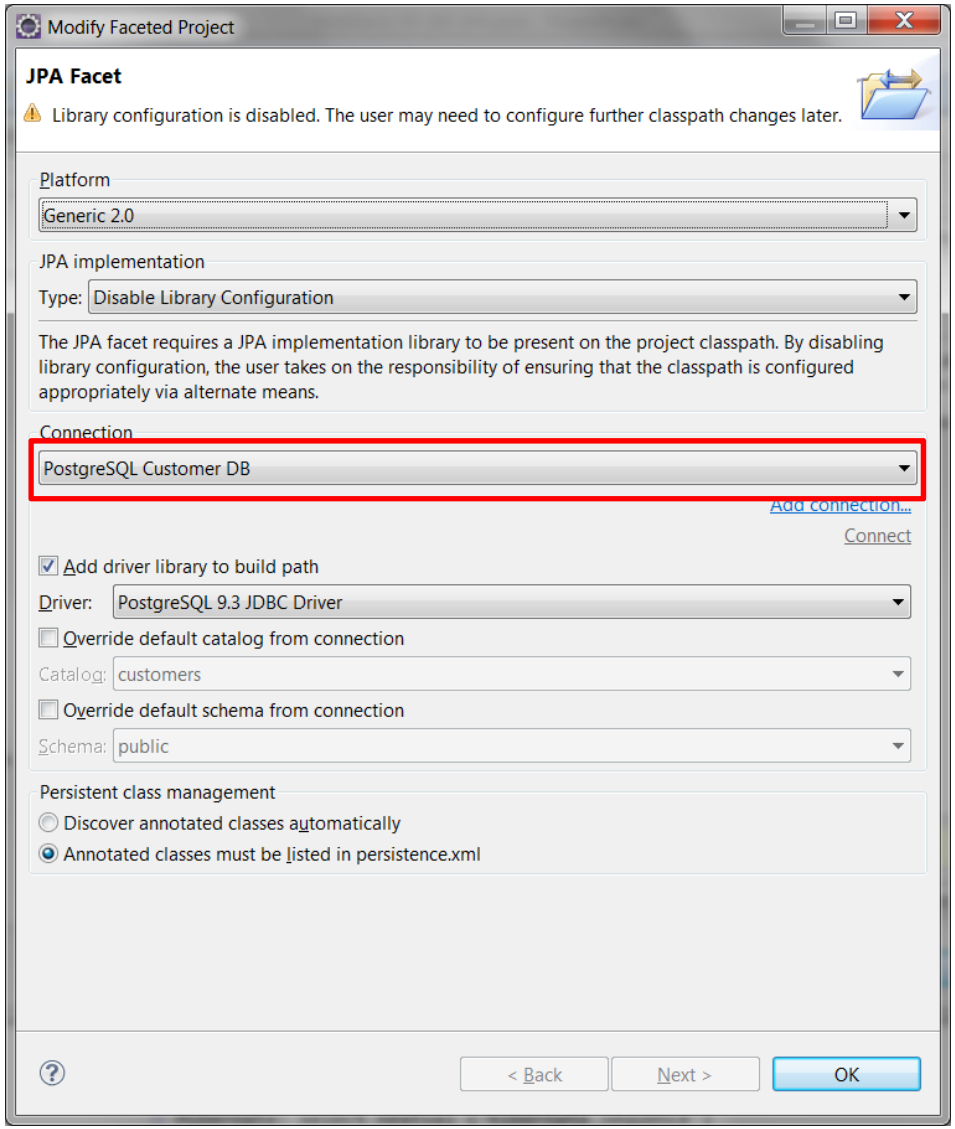
Datenbankverbindung in Eclipse konfigurieren (1)



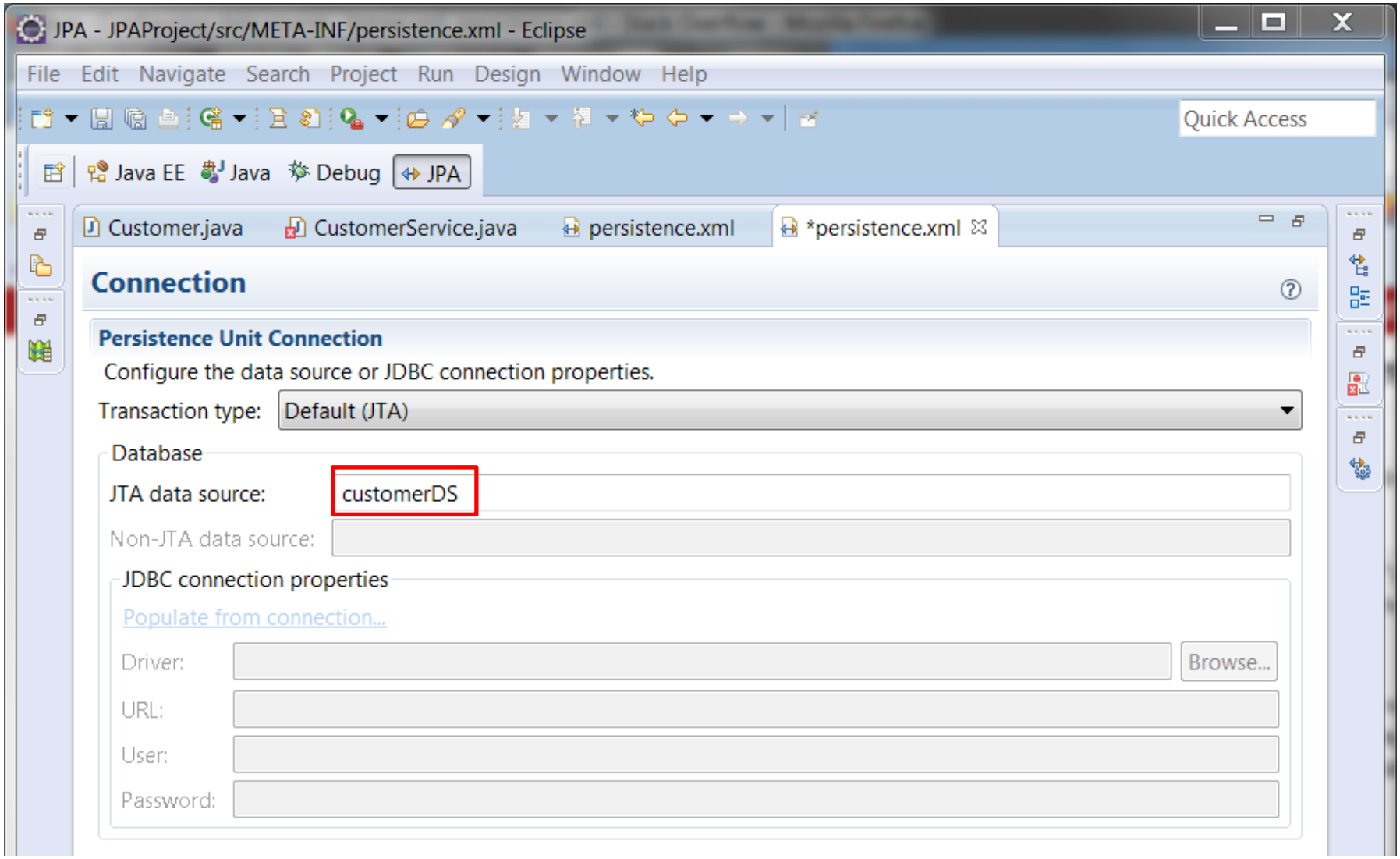
Datenbankverbindung in Eclipse konfigurieren (2)



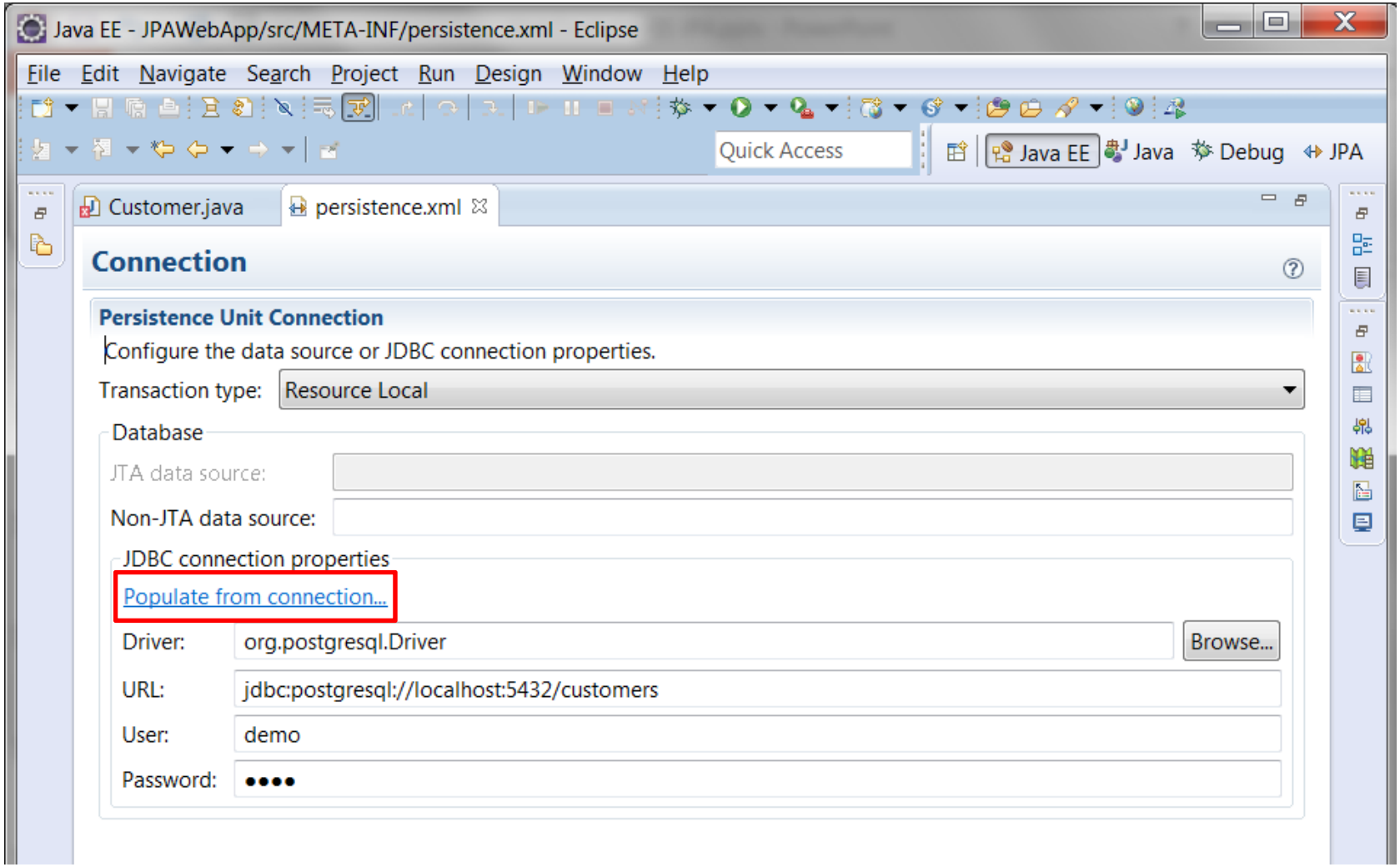
Konfigurierte JPA Facet



DataSource konfigurieren in `persistence.xml`

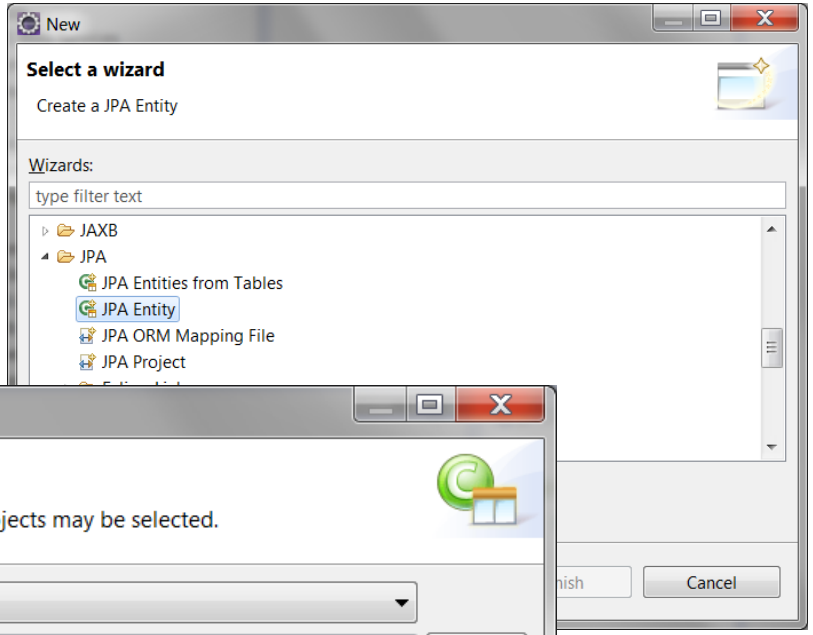


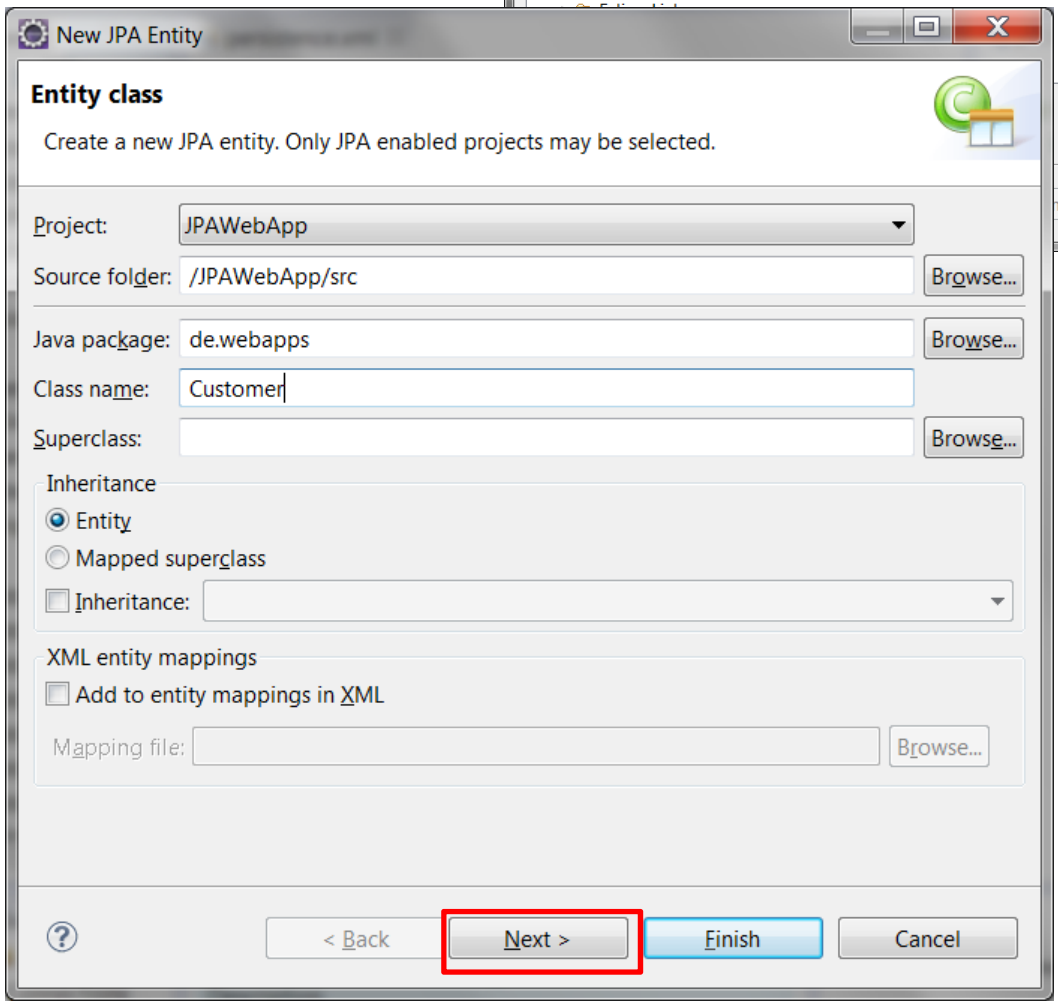
Variante 2: JDBC Datenbankverbindung konfigurieren in `persistence.xml`



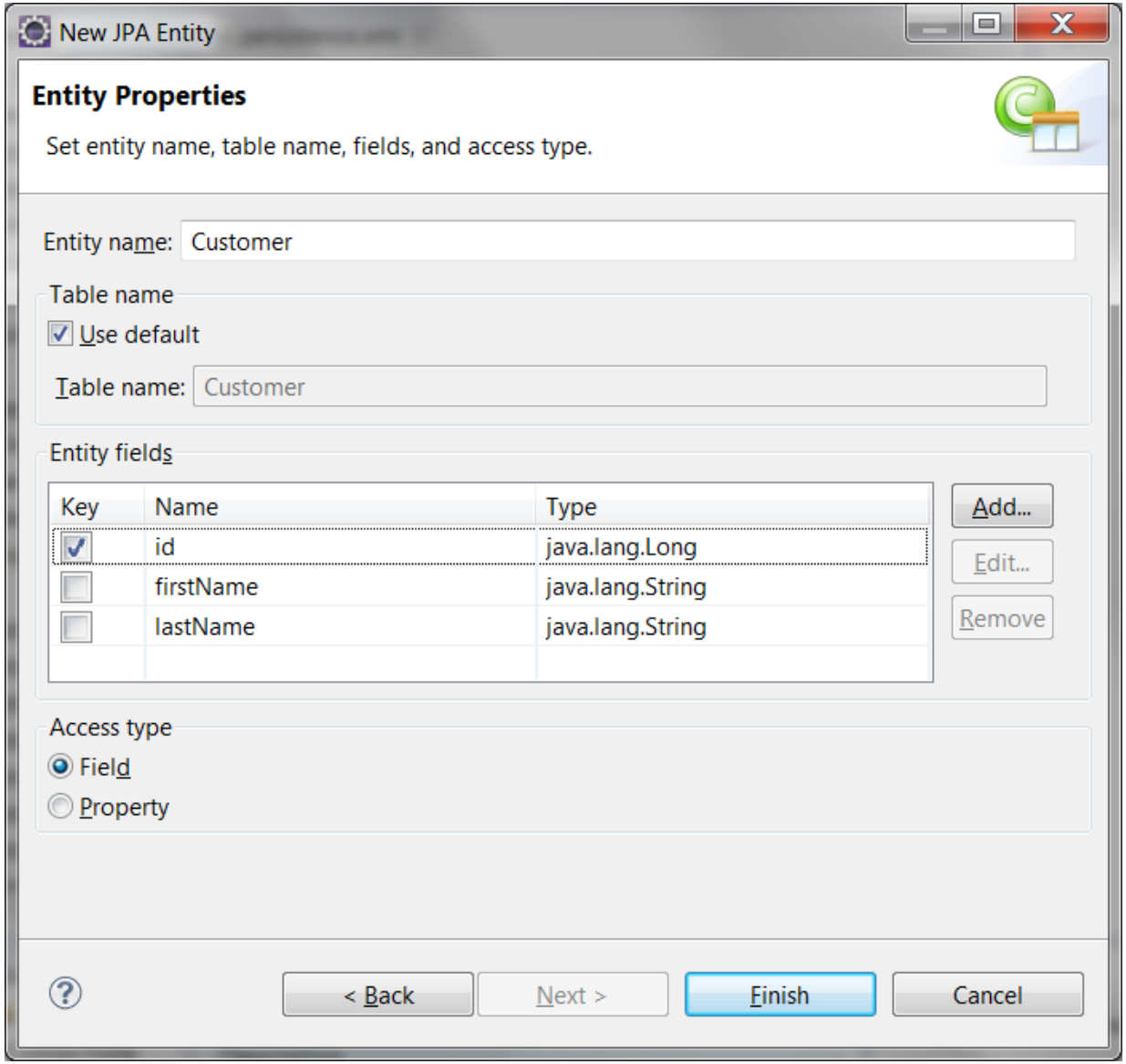
JPA Entity Klasse erstellen (1)

- File → New → JPA Entity





JPA Entity Klasse erstellen (2)



PROGRAMMIERUNG DER JAVA-KLASSEN

Entity-Klasse (1): Grundlagen

- Eine Entity-Klasse repräsentiert eine Tabelle in der Datenbank.
- Instanzen der Klasse entsprechen Einträgen in der Tabelle.
- Eine Java-Klasse wird zur Entity-Klasse durch die Annotation:
  - `@Entity`
- Entity-Klassen müssen immer einen Default-Konstruktor haben, damit der Persistence Provider Instanzen erstellen kann.
- Das Datenfeld, das den Primärschlüssel repräsentiert wird annotiert mit:
  - `@Id`
- Die folgende Annotation an einem Schlüsseldatenfeld für IDs sorgt dafür, dass die Datenbank die Werte für die IDs automatisch vergibt:
  - `@GeneratedValue(strategy = GenerationType.AUTO)`

Entity-Klasse (2): Beispiel

```
@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  private String firstName;
  private String lastName;

  public Customer() { super(); }

  public Customer(String firstName, String lastName) {
    super();
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }
...
}
```

Entity-Klasse (3): EntityManager

- Zur Verwaltung von Entity-Klassen und somit Zugriff auf die Datenbank ist der `EntityManager` erforderlich.
- Der EntityManager wird per Depedency-Injection instantiiert:
  - ```
    @PersistenceContext
    EntityManger em;
    ```
- Beim Erzeugen der Entity-Klasse besorgt der Container ein Objekt der Klasse EntityManager und ermittelt die zugehörige Konfiguration aus der Datei:
  - `persistence.xml`
  - Siehe nächste Folien.
- Falls in der Datei `persistence.xml` mehrere Persistenzeinheiten (`<persistence-unit>`-Elemente) definiert sind, kann man durch den Parameter `name` der Annotation `@PersistenceContext` die gewünschte auswählen:
  - `@PersistenceContext(name="JPAWebApp")`
    ```

Entity-Klasse (4): Verwendung in stateless Session-Bean (1)

- Objekt in Datenbank persistieren:
  - `em.persist(`*`Objekt`*`);`
- Anfrage an Datenbank stellen
  - `TypedQuery<`*`ErwarteterRückgabetyp`*`> query = em.createQuery("`*`JPQL-Query`*`", `*`ErwarteterRückgabetyp`*`);`
  - Anfragesprache ist JPQL (Java Persistence Query Language)
    - Sieht aus wie SQL, allerdings arbeitet JPQL auf Objekten und Attributen statt auf Tabellen- und Spaltennamen.
  - Anfrage abschicken und Ergebnisliste abholen:
    - `query.getResultList();`
  - Anfragen lassen sich per Annotation vordefinieren und ähnlich wie ein Prepared Statement verwenden:

```
@Entity
@NamedQueries({
  @NamedQuery(name = Project.FIND_ALL,
      query = "select p from Project p order by
p.position"),
  @NamedQuery(name = Project.FIND_BY_NAME,
      query = "select p from Project p where
p.name=:name")
})
public class Project implements Serializable { …
```

Entity-Klasse (5): Beispiel
Verwendung in stateless Session-Bean (2)

```
package de.webapps;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;

@Stateless
public class CustomerService {

  @PersistenceContext()
  private EntityManager em;

  public void create(Customer customer) {
    em.persist(customer);
  }

  public List<Customer> getAll() {
    TypedQuery<Customer> query =
        em.createQuery("select c from Customer c", Customer.class);
    return query.getResultList();
  }
}
```

Entity-Klasse (6): Nutzung der Session-Bean aus einem Servlet

```
@WebServlet("/CustomerServlet")
public class CustomerServlet extends HttpServlet {

  @EJB
  private CustomerService service;

  protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
                                throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();

    service.create(new Customer("Sheldon", "Cooper"));
    service.create(new Customer("Leonard", "Hofstadter"));
    service.create(new Customer("Howard", "Wollowitz"));
    service.create(new Customer("Rajesh", "Koothrappali"));

    try {
      out.println("<html><head><title>JPA Web App</title></head>");
      out.println("<body>");
      out.println("<h1>Kundenliste</h1>");
      for (Customer customer : service.getAll()) {
        out.println(customer.getFirstName() + " " + customer.getLastName() + "<br/>");
      }
      out.println("</body>");
      out.println("</html>");
    } finally {
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Entity-Klasse (7): Abbildung auf die Datenbank(tabelle)

Prinzip:
Convention over Configuration

- Wenn man keine weiteren Angaben macht, gelten Standardeinstellungen:
  - Datenbanktabelle muss dann den gleichen Namen haben wie die Klasse.
  - Spalten der Datenbanktabelle müssen jeweils den gleichen Namen haben wie die Datenfelder der Klasse.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String firstName;
    private String lastName;

    public Customer() { super(); }

    public Customer(String firstName, String lastName) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public Long getId() {
        return id;
    }
```

Entity-Klasse (8): Abbildung auf die Datenbank(tabelle)

- Konfiguration von der Konvention abweichender Einstellungen:
  - Anderer Tabellenname kann festgelegt werden mit der Annotation:
    - `@Table(name="AndererName")`
  - Anderer Spaltenname kann festgelegt werden mit der Annotation:
    - `@Column(name="AndererName")`
  - Spalten, die den Wert null enthalten dürfen, also leere Werte:
    - `@Column(nullable=true)`
  - Spaltenwerte muss eindeutig sein:
    - `@Column(unique=true)`
  - etc.
  - Datenfelder, die nicht in der Datenbank gespeichert werden sollen, z.B. Caches, werden annotiert mit:
    - `@Transient`

Hinweis:
`@Column` kann an die Deklaration des Datenfelds oder die `get()`-Methode geschrieben.

Entity-Klasse (9): Abbildung auf die Datenbank(tabelle)

- Datenfelder, die Datums- und Uhrzeitangaben enthalten, müssen besonders annotiert werden, damit der Peristence Provider weiss, auf welchen Datenbanktyp diese Felder abgebildet werden sollen:
  - `@Temporal(`*`Datenbanktyp`*`)`
  - Beispiel:
    - ```
      @Temporal(javax.persistence.TemporalType.DATE)
      private java.util.Date dateOfBirth;
      ```
  - Mögliche Datentypen sind:
    - `DATE: java.sql.Date`
    - `TIME: java.sql.Time`
    - `TIMESTAMP: java.sql.Timestamp`
      ```

Entity-Klasse (10): Abbildung auf die Datenbank(tabelle)

- Statt durch Annotationen kann die Abbildung von Klassen auf Tabellen und deren Attribute auch durch eine XML-Datei spezifiziert werden:

```
<entity class="de.webapps.Customer">
  <attributes>
    <id name="id"/>
    <basic name="firstName">
      <column name="firstname" />
    </basic>
    <basic name="lastName">
      <column name="lastname" />
    </basic>
  </attributes>
</entity>
```

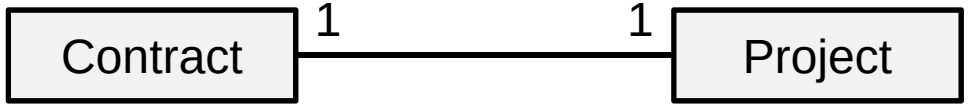
Entity-Klasse (11): Strategien zur Erzeugung von IDs für Primärschlüssel

- Ohne weitere Annotation an dem Schlüsseldatenfeld geht der Container davon aus, dass sich die Anwendung um die Erzeugung und eindeutige Vergabe der IDs kümmert.
- Verwenden einer anderen speziellen Strategie:
  - `@GeneratedValue(strategy="`*`Strategie`*`")`
- Für *Strategie* gibt es die folgenden Möglichkeiten, wobei die Auswahl vom verwendeten Datenbanksystem abhängt:
  - `GenerationType.AUTO`
    - Persistence Provider wählt selbständig unter einer der drei folgenden Strategien aus.
  - `GenerationType.IDENTITY`
    - Datenbank verwendet IDENTITY-Spalten, die dafür sorgen, dass einem neuen Tabelleneintrag eine neue IDs zugewiesen wird.
  - `GenerationType.SEQUENCE`
    - Die IDs werden durch ein Sequenz-Objekt (Sequence) der Datenbank erzeugt.
  - `GenerationType.TABLE`
    - Die IDs werden von der Datenbank in einer separaten Tabelle verwaltet.
    - Diese Lösung ist portabel.

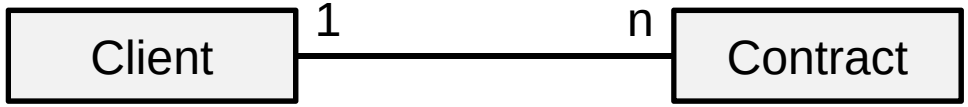
BEZIEHUNGEN ZWISCHEN ENTITIES

Beziehungen zwischen Entities

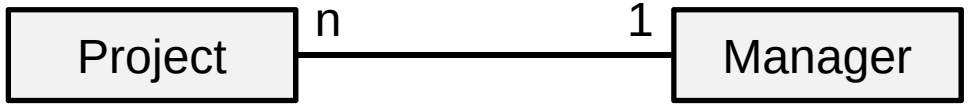
- Mit Entities realisieren wir Objekte einer Domäne.
- Objekte stehen in einer Domäne typischerweise in einer Beziehung.
- Beziehungstypen:
- 1:1 - `@OneToOne`



- 1:n - `@OneToMany`



- n:1 - `@ManyToOne`



- n:m - `@ManyToMany`

- bidirektionale Beziehungen:
  - Annotationen werden jeweils an die zugehörigen Attribute der beteiligten Enitity-Klassen geschrieben.
- unidirektionale Beziehungen:
  - Annotationen werden nur an das Attribut einer Entity-Klasse geschrieben.

Beispiel (1): Klasse `Developer`

```
@Entity
public class Developer implements Serializable {
    @Id
    private Long id;

    @ManyToMany
    private Set<Project> projects;

    public Long getId() {
        return id;
    }
}
```

Beispiel (2): Klasse `Project`

```
@Entity
public class Project implements Serializable {
    @OneToOne
    private Contract contract;

    @ManyToOne
    private Manager manager;

    @ManyToMany(mappedBy = "projects")
    private List<Developer> developers;

    @Id
    private Long id;

    public Long getId() {
        return id;
    }
}
```

- Das Attribut `mappedBy` gibt an, welche Seite der Beziehung die Beziehung besitzt, d.h. der Ausgangspunkt ist.

Beispiel (3): Klasse `Contract`

```
@Entity
public class Contract implements Serializable {
    @Id
    private Long id;

    @OneToOne(mappedBy = "contract", cascade=CascadeType.ALL)
    private Project project;

    @ManyToOne
    private Client client;

    public Long getId() {
        return id;
    }
}
```

- Das Attribut `mappedBy` gibt an, welche Seite der Beziehung die Beziehung besitzt, in diesem Fall ist der Ausgangspunkt das Datenfeld `contract` der Klasse `Project`.
- Das Attribut `cascade` legt fest, welche Aktion der EntityManager mit den Entities auf der anderen Seite der Beziehung durchführen soll.

Kaskadierung von Aktionen (1)

- `cascade=`*`Kaskadierungstyp`*
- *`Kaskadierungstyp`* kann sein:
  - `CascadeType.ALL`
    - Alle der folgenden Aktionen ausführen.
  - `CascadeType.DETACH`
    - Loslösen des / der anderen Entities
  - `CascadeType.MERGE`
    - Wieder mit der Datenbank synchronisieren, d.h. Änderungen am Enitity in die Datenbank schreiben.
  - `CascadeType.PERSIST`
    - Neues Entitiy in die Datenbank schreiben.
  - `CascadeType.REFRESH`
    - Ist Gegenteil von MERGE, d.h. Änderungen am Datensatz in der Datenbank werden neu in das Entity eingelesen.
  - `CascadeType.REMOVE`
    - Datensatz aus der Datenbank entfernen.

Kaskadierung von Aktionen (2)

- Im Beispiel bedeutet der Kaskadierungstyp `ALL` am Datenfeld `project` in der Klasse `Contract`, dass wenn das `Contract`-Entity-Objekt entfernt wird, dann wird auch das zugehörige Project entfernt.
- Das mag sinnvoll sein, muss aber im Einzelfall genau betrachtet werden.
- Wenn andererseits ein Projekt entfernt wird, dann will man in seiner Firma sicher den Manager und die Entwickler auch weiterhin für andere Projekte beschäftigen wollen.
- Wenn der Projektleiter die Firma verlässt, will man sicher auch nicht gleich das ganze Projekt löschen, sondern eher einen neuen Projektleiter einsetzen.

Verhindern von verwaisten Datenbankeinträgen

- Bei `@OneToOne`- und `@OneToMany`-Beziehungen gibt es noch das Attribut
  - `orphanRemoval`
- Setzt man dieses auf `true` und entfernt aus der Datenstruktur (Datenfeld, Menge, Liste etc.) der Zielseite ein Objekt, dann wird dieses auch in der Datenbank gelöscht.
- Beispiel:

```
@Entity
public class Client implements Serializable {
  @Id
  private Long id;

  @OneToMany(mappedBy = "client", orphanRemoval=true)
  private Set<Contract> contracts;
```

- Beim Löschen eines `Contract`-Objekts aus dem Set des Datenfelds `contracts` wird auch der Eintrag für das `Contract`-Objekt in der Datenbank gelöscht.
  - Andernfalls gäbe es in der Datenbank ein `Contract`-Objekt, das nirgends mehr zugeordnet ist.

Lazy Loading (1)

- Problem: Lazy Loading
  - Hibernate lädt Objekte von x:n-Beziehungen (x=1 oder x=m) nur bei Bedarf
  - Wenn wir aus dem Programm heraus auf ein Objekt zugreifen wollen, das bisher nicht geladen wurde, wird eine `LazyInitializationException` ausgelöst, weil keine Datenbankverbindung mehr vorhanden ist.
- Lösungsalternativen: (http://www.javacodegeeks.com/2012/07/four-solutions-to-lazyinitializationexc_05.html)
  - Immer alle Sammlungen vollständig laden
    - Attribut `fetch=FetchType.EAGER` setzen
  - Umschließende Benutzertransaktion für alle Anfragen definieren
    - D.h. Filter-Klasse für Benutzertransaktion in `web.xml` definieren
  - Stateful Session Bean definieren mit `PersistenceContextType.EXTENDED`
  - Sammlungen durch Tabellen-Join laden
  - Ab JEE 7: Aufruf aus ManagedBean in einer Transaktion laufen lassen
    - `@Transactional`

Lazy Loading (2)

- Immer alle Sammlungen vollständig laden
  - Attribut `fetch=FetchType.EAGER` setzen
  - Beispiel:
    - `@OneToMany(fetch=FetchType.EAGER)`
  - Nachteil:
    - Auch bei großen Sammlungen werden immer alle Elemente geladen, so dass die Performance der Anwendung sich drastisch verschlechtert.

Lazy Loading (3)

- Umschließende Benutzertransaktion für alle Anfragen definieren (1)
  - D.h. Filter-Klasse für Benutzertransaktion in `web.xml` definieren.

```
<filter>
  <filter-name>ConnectionFilter</filter-name>
  <filter-class>com.filter.ConnectionFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>ConnectionFilter</filter-name>
  <url-pattern>/faces/*</url-pattern>
</filter-mapping>
```

- Nachteil:
  - Für geschachtelte x:n-Beziehungen werden ständig neue Datenbankanfragen erzeugt.

Lazy Loading (4)

- Umschließende Benutzertransaktion für alle Anfragen definieren (2)

```
public class ConnectionFilter implements Filter {

 @Override
 public void destroy() { }

 @Resource
 private UserTransaction utx;

 @Override
 public void doFilter(ServletRequest request, ServletResponse response,
                      FilterChain chain) throws IOException,
ServletException {
  try {
   utx.begin();
   chain.doFilter(request, response);
   utx.commit();
  } catch (Exception e) {
   e.printStackTrace();
  }

 }

 @Override
 public void init(FilterConfig arg0) throws ServletException { }
}
```

Lazy Loading (5)

- Stateful Session Bean definieren mit `PersistenceContextType.EXTENDED`

```
@Stateful
public class SystemDAOStateful {
 @PersistenceContext(unitName = 'LazyPU',
                     type=PersistenceContextType.EXTENDED)
 private EntityManager entityManager;

 public Client findByName(String name) {
  Query query =
    entityManager.createQuery('select c from Client c where name = :name');
  query.setParameter('name', name);

  Client result = null;
  try {
   result = (Client) query.getSingleResult();
  } catch (NoResultException e) {
   // no result found
  }
  return result;
 }
}
```

- Nachteile:
  - Für geschachtelte x:n-Beziehungen werden ständig neue Datenbankanfragen erzeugt.
  - Funktioniert mit vollem JEE (Full Profile)

Lazy Loading (6)

- Sammlungen durch Tabellen-Join laden:

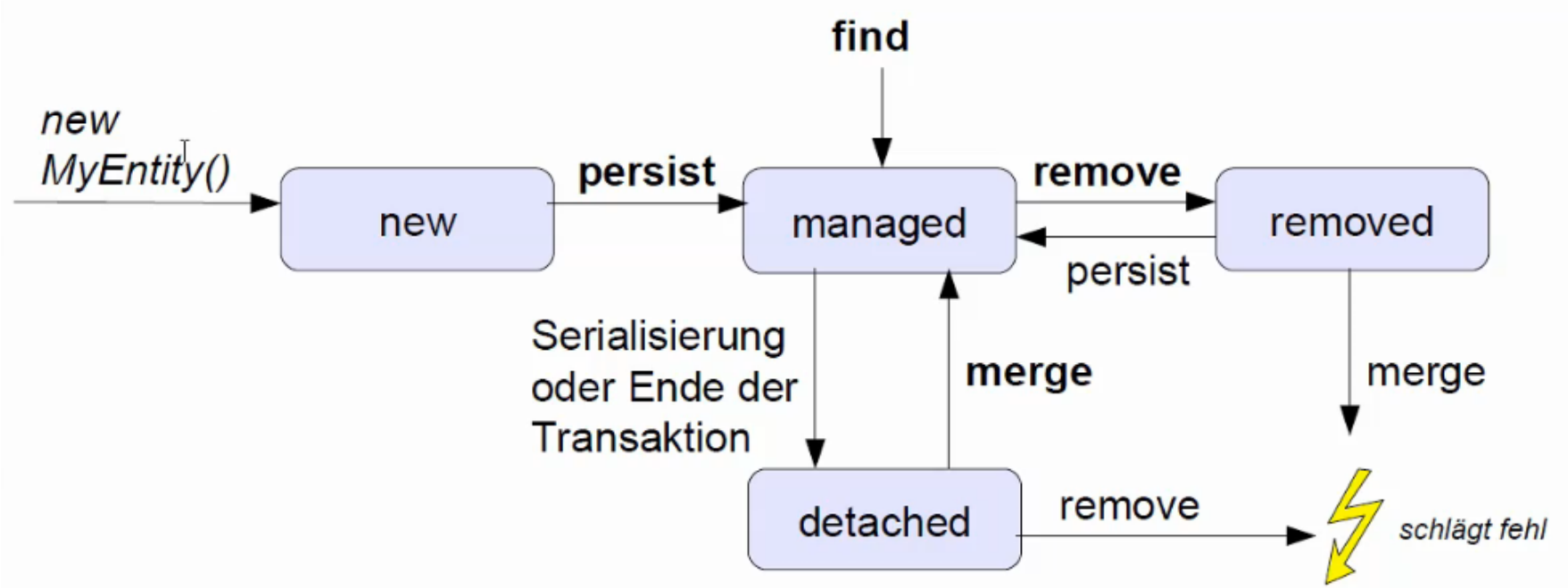
```
 public Client findByName(String name) {
  Query query =
    entityManager.createQuery('select c from Client c join
fetch c.contracts where c.name = :name');
  query.setParameter('name', name);

  Client result = null;
  try {
   result = (Client) query.getSingleResult();
  } catch (NoResultException e) {
   // no result found
  }
  return result;
 }
}
```

- Nachteil:
  - Für jede Sammlung einer x:n-Beziehung muss eine eigene Anfrage (Query) geschrieben werden.

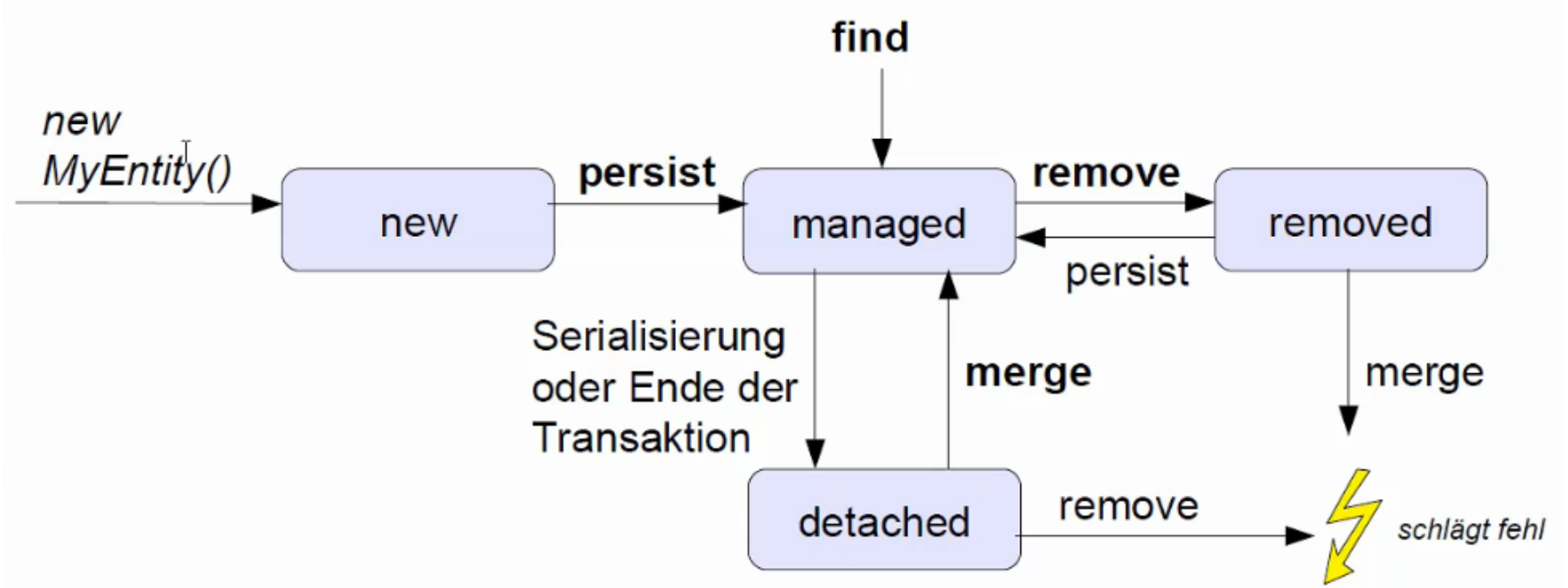
LEBENSZYKLUS VON JPA-ENTITIES

JPA-Entity-Lebenszyklus (1)



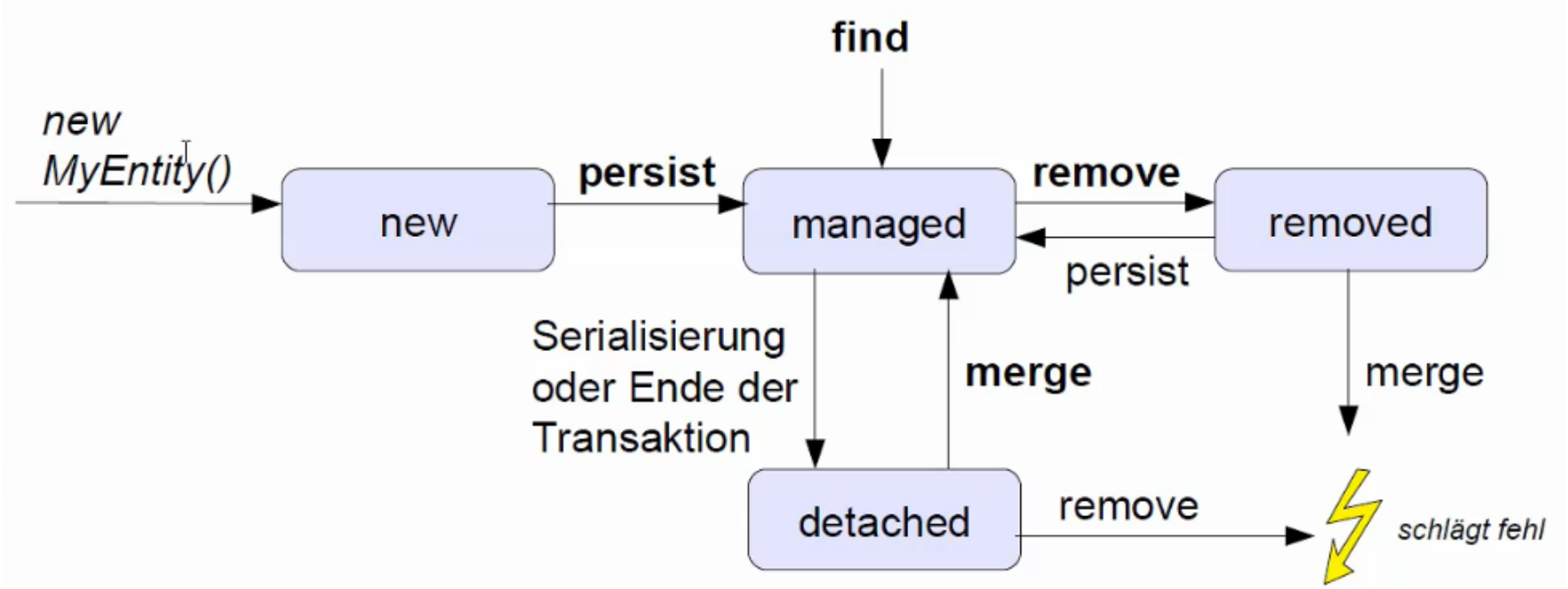
- Neue Objekte sind dem EntityManager zunächst nicht bekannt (Zustand **new**).
- Erst durch Aufruf von `persist()` wird ein Entity unter die Verwaltung des EntityManagers gestellt (Zustand **managed**), der dann einen Datensatz in der Datenbank anlegt und Änderungen am Objekt automatisch im Datensatz nachzieht.
- Durch das Ausführen einer Suche auf der Datenbank mit der Methode `find()` wird eine Menge von Ergebnisobjekten (Entity-Objekten) zurückgeliefert, die unter der Verwaltung des EntityManagers stehen und somit auch den Zustand **managed** haben.

JPA-Entity-Lebenszyklus (2)



- Entity-Objekte dürfen als POJOs zum Client übertragen werden.
  - Dazu werden die Objekte serialisiert und gehen in den Zustand **detached** über, d.h. es gibt dann keine Verbindung mehr zum entsprechenden Datensatz in der Datenbank.
  - Die Verbindung zum Datensatz wird durch Aufruf von `merge()` wiederhergestellt und zwischenzeitlich am Entity-Objekt durchgeführte Änderungen in den Datensatz in der Datenbank übernommen.
- Der Aufruf von `remove()` ist auf Entity-Objekten im Zustand **detached** nicht möglich, weil die Verbindung zum Datensatz in diesem Zustand nicht vorhanden ist.

JPA-Entity-Lebenszyklus (3)



- Der Aufruf von `remove()` auf einem Entity-Objekt im Zustand **managed** löscht den zugehörigen Datensatz in der Datenbank, das Objekt selbst existiert aber weiterhin und bekommt den Zustand **removed**.
  - Ein Objekt im Zustand **removed** kann durch Aufruf von `persist()` wieder in die Datenbank geschrieben werden und wird so wieder in den Zustand **managed** überführt.
  - Im Zustand **removed** ist der Aufruf von `merge()` nicht möglich, weil kein Datenbankeintrag exisitert.

JPA-Entity-Lebenszyklus (4)

- Anmerkung:
  - Entities dürfen auch an höhere Schichten z.B. Präsentationsschicht durchgereicht werden
- Begründung:
  - Entity-Objekte sind lediglich POJOs.
- ABER:
  - Es ist im Einzelfall zu entscheiden ob dies sinnvoll ist und ob dadurch nicht das Prinzip der losen Kopplung verletzt wird.

JPA OHNE EJB: BENUTZERTRANSAKTIONEN

JPA ohne EJBs (1)

- JPA kann auch ohne EJBs verwendet werden.
- Problem 1:
  - `EntityManager` ist nicht Thread-sicher.
- Bei Verwendung eines EntityManagers z.B. in einem Servlet, wird nur ein EntityManager-Objekt instantiiert, weil es auch nur eine Instanz des Servlets gibt, die alle Client-Anfragen beantwortet.
  - Alle Threads des Servlets verwenden dann dasselbe EntityManager-Objekt.
- Abhilfe:
  - Verwendung von `EntityManagerFactory`:
    - Wird vom Container zur Verfügung gestellt und liefert jeweils ein neues EntityManager-Objekt.

JPA ohne EJBs (2): EntityManagerFactory

```
@PersistenceUnit
private EntityManagerFactory emf;
…
EntityManager em = emf.createEntityManager();
…
em.close();
```

JPA ohne EJBs (3): `UserTransaction`

- Problem 2:
  - Transaktion erforderlich, um mit der Datenbank zu arbeiten, aber ohne EJBs erzeugt der Container Transaktionen nicht automatisch.
- Abhilfe:
  - `UserTransaction`: Transaktion im Programm selbst erzeugen und verwalten
- Code:
  - ```
    @Resource
    private UserTransaction utx;
    …
    try {
      utx.begin();
      …
      utx.commit();
    } catch (…) {
      utx.rollback();
    }
    ```
    ```

JPA ohne EJBs (4): Beispiel

- Projekt: JPAUserTransactionWebApp

```
public class CustomerService {

  @PersistenceUnit
  private EntityManagerFactory emf;

  @Resource
  UserTransaction utx;

  public void createCustomers() {
    try {
      EntityManager em = emf.createEntityManager();
      utx.begin();
      em.persist(new Customer("Sheldon", "Cooper"));
      em.persist(new Customer("Leonard", "Hofstadter"));
      em.persist(new Customer("Howard", "Wollowitz"));
      em.persist(new Customer("Rajesh", "Koothrappali"));
      utx.commit();
      em.close();
    } catch (Exception e) {
      try {
        utx.rollback();
      } catch (Exception e1) {
        e1.printStackTrace();
      }
  }
}
```

ZUSAMMENFASSUNG

Zusammenfassung (1)

- Historie und Entstehung von JPA
- Begriffe
- Architektur und Abläufe
- Datenbank installieren
- Hibernate installieren
- DataSource in TomEE konfigurieren
- Persistenzkonfiguration der Anwendung durch persistence.xml
- Strategien zur Tabellenerzeugung: create, drop and create, none
- JPA Plugins in Eclipse

Zusammenfassung (2)

- Entity-Klasse: `@Entity`
- Schlüsselfelder: @Id
- Schlüsselgenerierungstypen:
  - `AUTO`, `TABLE`, `SEQUENCE`, `IDENTITY`
- Abbildung von Entity auf Datenbanktabellen, insbesondere spezielle Datentypen
- `EntityManager` und `@PersistenceContext`
- Methoden des `EntityManager`:
  - `persist()`, `merge()`, `remove()`, `find()`, `createQuery()`
- Beziehungen zwischen Entities:
  - `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`
  - bidirektional, unidirektional
- Kaskadierung von Aktionen:
  - `ALL`, `DETACH`, `MERGE`, `PERSIST`, `REFRESH`, `REMOVE`

Zusammenfassung (3)

- Verhindern verwaister Einträge: `orphanremoval=true`
- Lebenszyklus von JPA-Entities
- JPA ohne EJB:
  - `EntityManagerFactory`, `UserTransaction`

LITERATUR

Literatur

- JPA-Spezifikation
  - http://jcp.org/en/jsr/detail?id=317
- Keith, Schincariol: "Pro JPA 2"
- Panda, Rahman, Lane: "EJB 3 in Action"