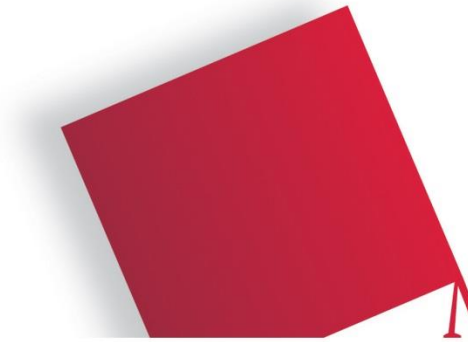


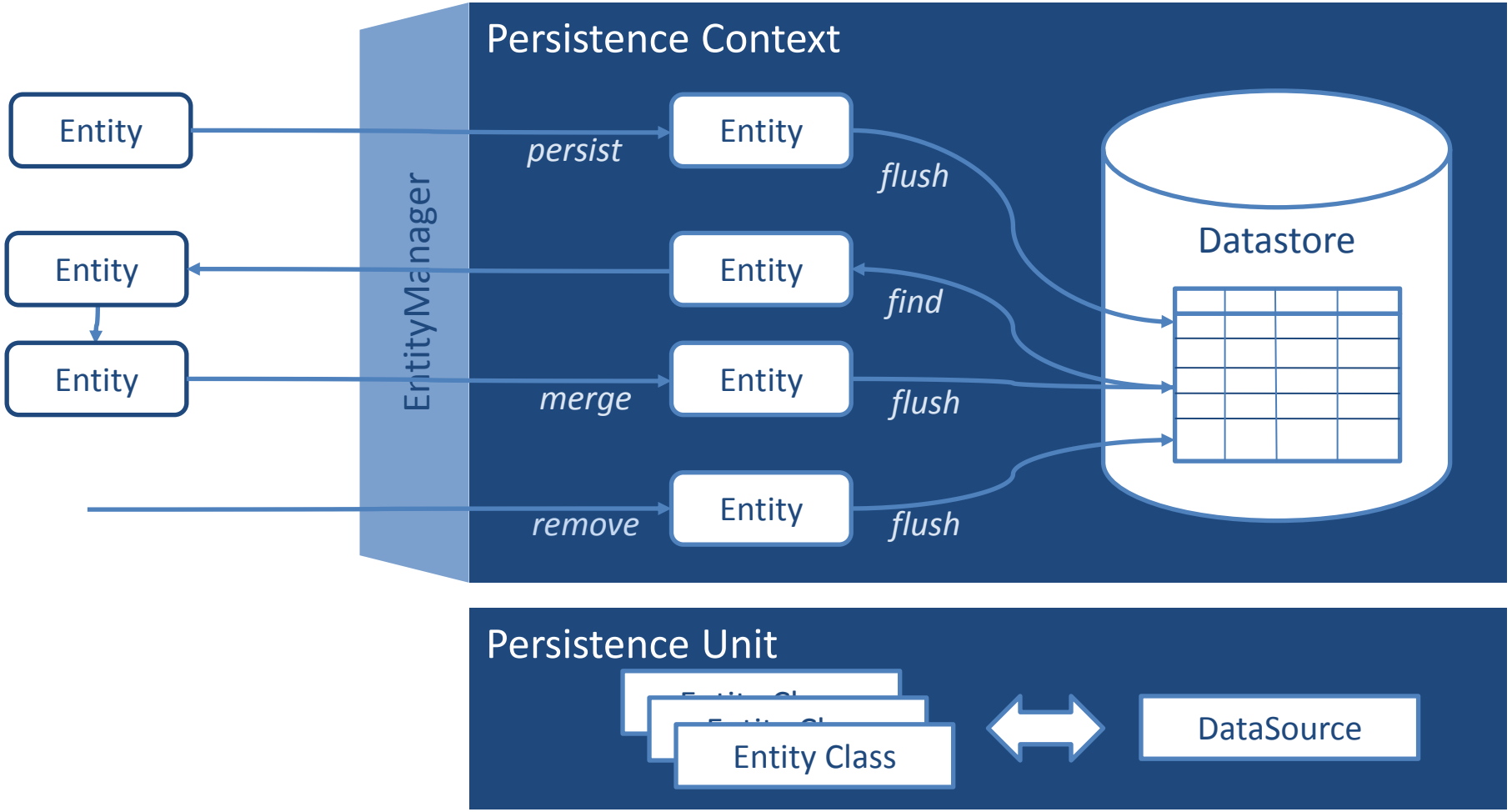
HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

# Datenzugriffskomponenten mit JPA

FWP Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

# GRUNDBEGRIFFE DER JAVA PERSISTENCE ARCHITECTURE

# Java Persistence Architecture



# Entitäten

- Entity = POJO + @Entity
- Verwaltete Entitäten (managed/attached) werden von einem Entity Manager überwacht und automatisch mit einem Datenspeicher synchronisiert
- Nicht verwaltete Entitäten (unmanaged/detached) werden nicht überwacht und synchronisiert, können aber wieder an einen Entity Manager gebunden werden

# Persistence Context (PC)

- Umfasst alle verwalteten Entitäten zu einem bestimmten Zeitpunkt
- Wird über einen EntityManager verwaltet
- Art bestimmt die Lebensdauer
  - *transaktionsgebunden*: Kontext  lebt für die Dauer einer Transaktion; automatische Synchronisation zum Ende der Transaktion
  - *erweitert*: Kontext überdauert Transaktionen; explizite Synchronisation durch den Erzeuger des Kontextes

# Entity Manager (EM)

- Verwaltet Entitäten in einem Persistence Context
- Bietet Methoden zur Manipulation von Persistence Contexten
- Art des Persistence Context bestimmt Art des Entity Managers
  - Transaktionsgebundener PC ⇔ Vom Container verwalteter (container-managed) Entity Manager
  - Erweiterter PC ⇔ Von der Applikation verwalteter (application-managed) Entity Manager

# Container-Managed EM

- Container injiziert einen Entity Manager in ein mit @PersistenceContext markiertes Feld
- Entity Manager zeigt auf einen transaktions-gebundenen Persistence Context
- Transaktion bestimmt Lebensdauer des PC und Zeitpunkt der Synchronisation mit der Datenbank

```
@Stateless
public class UserRepository {
   @PersistenceContext
   private EntityManager entityManager;
...
```

*Einsatz in Stateless Session Beans*

# Application-Managed EM

- Container injiziert eine Entity Manager Factory in ein mit @PersistenceUnit markiertes Feld
- Entity Manager zeigt auf einen erweiterten PC
- Komponente muss EM über Factory erzeugen
- Komponente bestimmt Lebensdauer des PC und Zeitpunkt der Synchronisation mit der Datenbank

```
@Stateful
public class UserGateway {
   @PersistenceUnit
   private EntityManagerFactory entityManagerFactory;
...
```

*Einsatz in Stateful Session Beans*

# Persistence Unit (PU)

- Bestimmt, welcher JPA-Provider verwendet werden soll und wie sich dieser verhalten soll
- Bindet Entitäten-Klassen über eine Datenquelle (DataSource) an einen bestimmten Datenspeicher
- Werden in Form einer XML-Datei /META-INF/persistence.xml konfiguriert
- Mehrere PU werden über Attribut unitName in @PersistenceContext/@PersistenceUnit unterschieden

# Exkurs: JDBC API

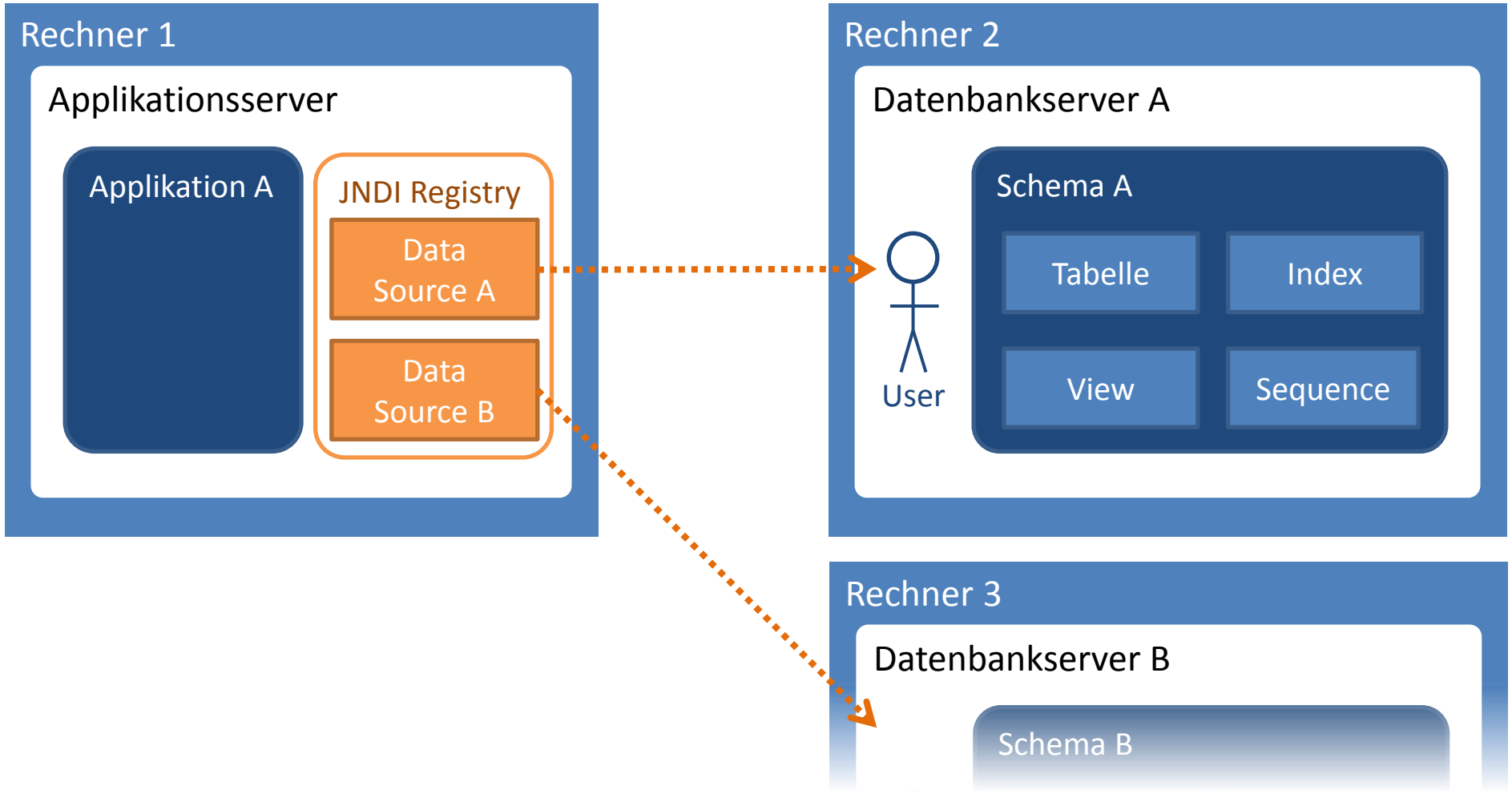
- JPA basiert auf bestehenden Java-Standards
- Java Database Connectivity (JDBC) API definiert Standard für Zugriff auf relationale Datenbanken
- javax.sql.DataSource repräsentiert eine abstrakte Factory für Verbindungen zu Datenbanken
- javax.sql.Connection repräsentiert eine Verbindung zu einer bestimmten Datenbank
- SQL-Anweisungen werden über eine Connection ausgeführt

# Data Sources

- DataSources werden üblicherweise als Ressourcen in einem Applikationsserver konfiguriert
- Konkrete Implementierungen von DataSources werden von den Herstellern datenbank-spezifischer JDBC-Treiber zur Verfügung gestellt
- Es gibt drei Implementierungstypen

| Implementierungstyp | Unterstützte Eigenschaften |
|---|---|
| Basic | Connection |
| Connection Pooling | Connection + Connection Pool |
| Verteilte Transaktionen (XA) | Connection + Connection Pool + Verteilte Transaktionen |

# DataSource als Tor zur Datenbank



# Object Relational Mapping

- Entitäten sind Java-Klassen mit Feldern
- In der Datenbank gibt es Tabellen und Spalten
- Implizite Abbildung erfolgt über Namen
  - Java-Namen müssen DB-Namen entsprechen
- Explizite Abbildung erfolgt über Annotationen (oder XML)
  - @Table mappt Klasse auf Tabelle
  - @Column mappt Feld auf Spalte

# Identität und Primärschlüssel

- Jede Entität muss eindeutige Identität besitzen
- @Id markiert ein Feld einer Entität als Primärschlüssel
- Primärschlüssel können
  - von der Applikation erzeugt werden
  - über JPA oder die Datenbank automatisch generiert werden (@GeneratedValue)

# Komplexe Primärschlüssel

- Setzen sich aus mehreren Feldern/Spalten zusammen
- Zwei Strategien möglich
  - @EmbeddedId markiert komplexes Feld der Entityklasse; @Embeddable markiert Primärschlüsselklasse
  - @IdClass definiert Primärschlüsselklasse; mehrere Felder der Entityklasse werden mit @Id markiert

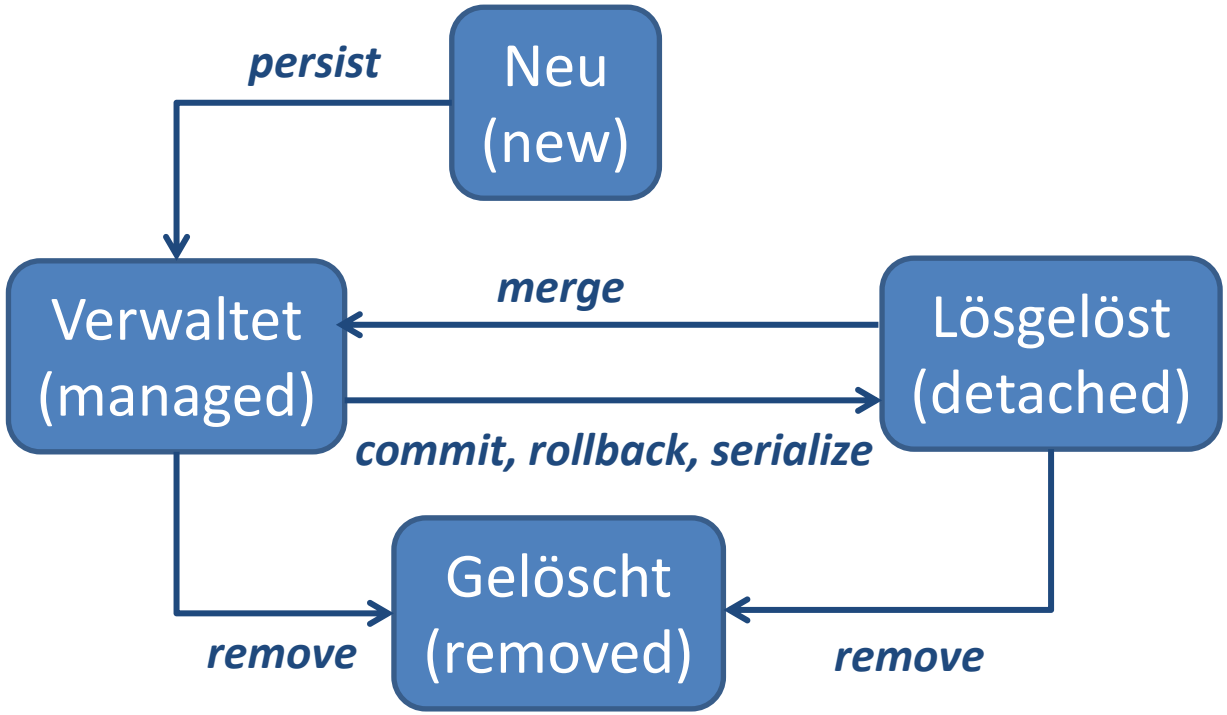
# 7 Formen der Beziehung

- 1-zu-1 unidirektional (one-to-one unidirectional)
- 1-zu-1 bidirektional (one-to-one bidirectional)
- 1-zu-n unidirektional (one-to-many unidirectional)
- 1-zu-n bidirektional (one-to-many bidirectional)
- n-zu-1 unidirektional (many-to-one unidirectional)
- m-zu-n unidirektional (many-to-many unidirectional)
- m-zu-n bidirektional (many-to-many bidirectional)

# Zustände einer Entität

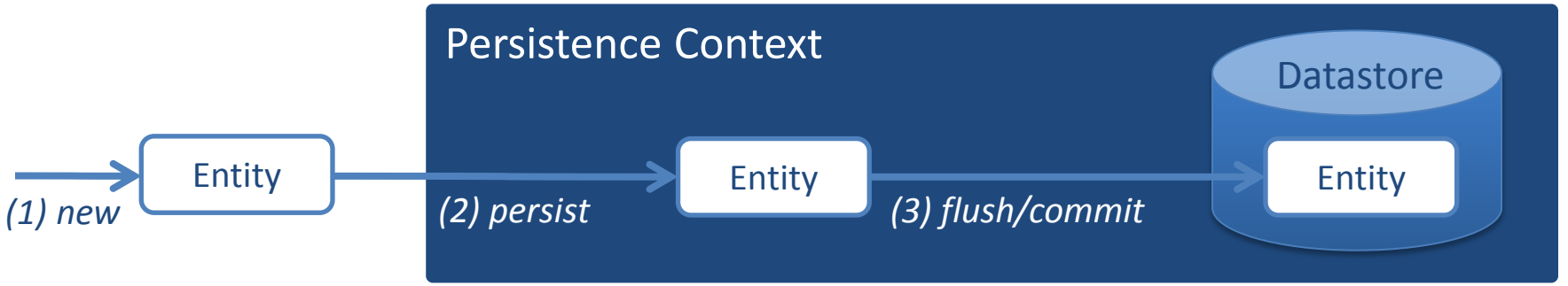
- Neu (new): Entität besitzt keine persistente Identität und ist mit keinem Persistence Context verknüpft
- Verwaltet (managed): Entität besitzt eine persistente Identität und ist mit einem Persistence Context verknüpft
- Losgelöst (detached): Entität besitzt eine persistente Identität und ist mit keinem Persistence Context verknüpft
- Gelöscht (removed): Entität besitzt eine persistente Identität, ist mit einem Persistence Context verknüpft und für das Löschen aus dem Datenspeicher vorgemerkt

# Zustände einer Entität



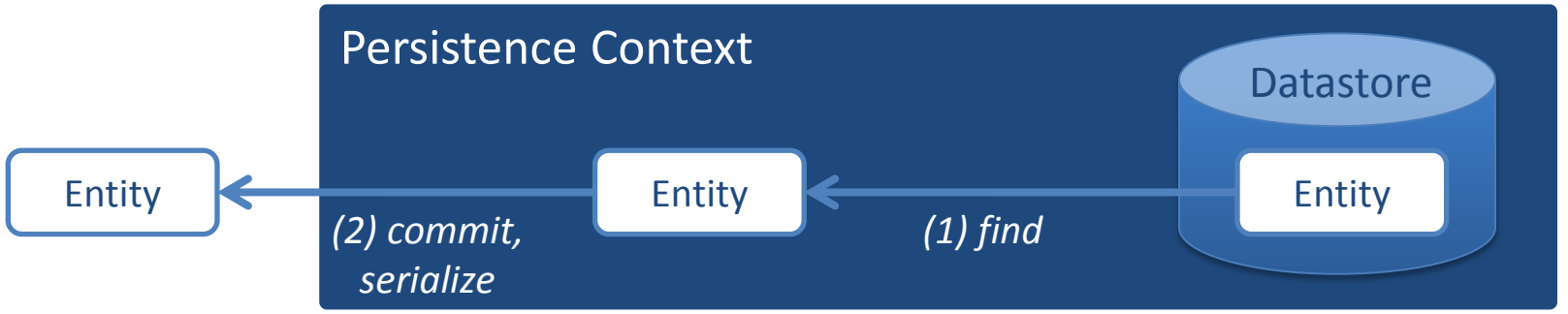
# Neue Entität hinzufügen

- Neue Entität wird mit new-Operator erzeugt
- Neue Entität wird mit EntityManager.persist() persistiert
- Endgültiges Hinzufügen erfolgt mit Commit der aktuellen Transaktion



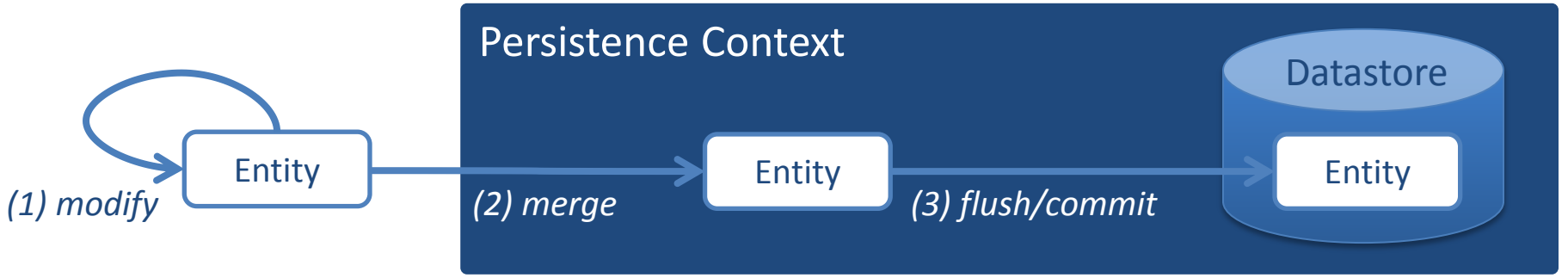
# Bestehende Entität lesen

- Bestehende Entitäten können mit Entity-Manager.find() über ihren Primärschlüssel gelesen werden
- Komplexere Abfragen erfolgen über Queries



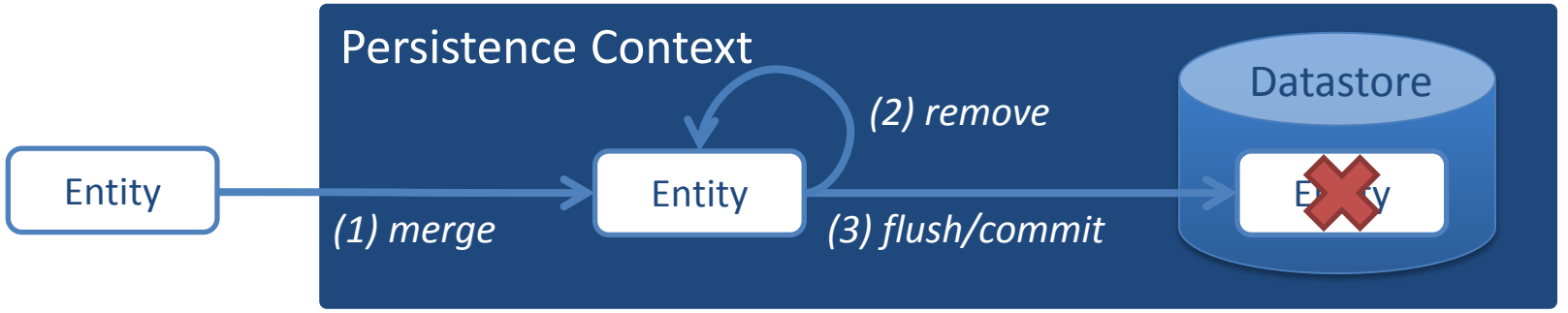
# Bestehende Entität aktualisieren

- Änderungen an verwalteten Entitäten werden automatisch erkannt
- Losgelöste Entitäten müssen erst mit Entity-Manager.merge() in den PC gebracht werden
- Endgültige Aktualisierung erfolgt mit Commit der aktuellen Transaktion



# Bestehende Entität löschen

- Verwaltete Entitäten können mit Entity-Manager.remove() zur Löschung markiert werden
- Losgelöste Entitäten müssen erst mit Entity-Manager.merge() in den PC gebracht werden
- Endgültiges Löschen erfolgt mit Commit der aktuellen Transaktion

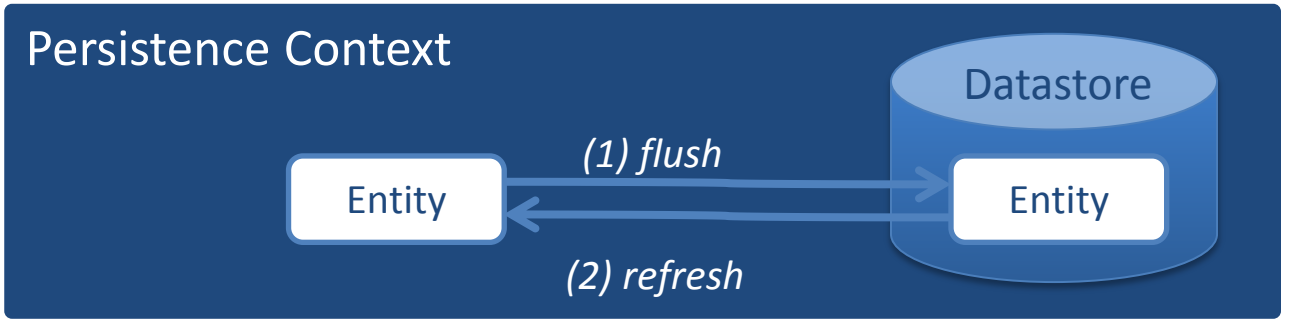


# Lösgelöste Entitäten

- Entitäten werden häufig vom PC losgelöst und nicht mehr überwacht
  - durch Commit/Rollback der aktuellen Transaktion
  - durch Serialisierung über Remote Interfaces
- Losgelöste Entitäten müssen mit EntityManager.merge() wieder an einen PC gebunden werden

# Synchronisation gezielt steuern

- Beim Commit einer Transaktion werden Entitäten automatisch mit Datenspeicher synchronisiert
- Mit EntityManager.flush() können Entitäten früher in den Datenspeicher geschrieben werden
- Mit EntityManager.refresh() können Entitäten mit Zustand des Datenspeichers aktualisiert werden



# Beispiel für gezielte Synchronisation

- Persistieren einer Entität mit automatisch durch den Datenspeicher generiertem Primärschlüssel

```
@Stateless
public class OrderRepository {
   @PersistenceContext
   private EntityManager entityManager;
   …
   public Order addOrder(Order newOrder) {
      // Order für Einfügen vormerken
      this.entityManager.persist(newOrder);
      // Einfügen der Order in den Datenspeicher erzwingen
      this.entityManager.flush();
      // Order mit generiertem Primärschlüssel aktualisieren
      this.entityManager.refresh(newOrder);
      return newOrder;
   }
```

# Komplexe Abfrage von Entitäten

- EntityManager.find() kann nur einzelne Entitäten über Primärschlüssel lesen
- Für komplexere Queries bietet JPA an:
  - Queries auf Basis von Abfragesprachen (JPQL/SQL)
  - Objekt-orientierte Queries auf Basis der Criteria API

Im Rahmen dieses Faches werden nur Abfragesprache-basierte Queries betrachtet!

# Queries

- Repräsentiert durch javax.persistence.TypedQuery
- Erzeugt über Factory-Methoden im EntityManager
- Können zu verschiedenen Zeiten definiert werden:
  - Dynamische Queries zur Laufzeit
  - Statische (named) Queries zur Compilezeit
- Abfragen in zwei Sprachen möglich:
  - Java Persistence Query Language (JPQL)
  - Standard Query Language (SQL)
- Auch Bulk-Updates und Bulk-Deletes möglich

# Dynamische Queries

- Werden über EntityManager.createQuery() durch Angabe eines Statements erzeugt:

```
public List<Order> findAllOrdersOfCustomer(String customerId) {
   TypedQuery<Order> query = this.entityManager.createQuery(
      "SELECT o FROM Order o WHERE o.customerId = :customerId",
      Order.class);
   query.setParameter( "customerId", customerId );
   return query.getResultList();
}
```

# Statische Queries (Named Queries)

- Werden über @NamedQuery an Entity definiert

```
@Entity
@NamedQueries({@NamedQuery(name="QUERY_ALL_OF_CUSTOMER",
   query="SELECT o FROM Order o WHERE o.customerId = :customerId")})
public class Order { …
```

- Werden über EntityManager.createNamedQuery() durch Angabe ihres Namens erzeugt:

```
public List<Order> findAllOrdersOfCustomer(String customerId) {
   TypedQuery<Order> query = this.entityManager.createNamedQuery(
      „QUERY_ALL_OF_CUSTOMER", Order.class);
   query.setParameter( "customerId", customerId );
   return query.getResultList();
}
```

# Vergleich der Queryarten

## Dynamische Queries

- ☺ Statements müssen erst zur Laufzeit konkretisiert werden
- ☹ Ablage der Statements nicht geregelt
- ☹ Fehler in den Statements werden erst bei deren erster Ausführung erkannt

## Named Queries

- ☺ Statements werden an einer einheitlichen Stelle definiert (an den Entities)
- ☺ Fehler in den Statements werden bereits beim Deployment der Applikation erkannt

**Empfehlung**: Bevorzuge immer Named Queries! Wende dynamische Queries erst dann an, wenn es die fachlichen Anforderungen explizit erfordern!

# JPQL ist eine mächtige Sprache

ABS
ALL
AND
ANY
AS
ASC
AVG
BETWEEN
BIT_LENGTH
BOTH
BY
CASE
CHAR_LENGTH
CHARACTER_
LENGTH
CLASS
COALESCE
CONCAT
COUNT
CURRENT_DATE
CURRENT_
TIMESTAMP
**DELETE**
DESC
DISTINCT
ELSE
EMPTY
END
ENTRY
ESCAPE
EXISTS
FALSE
FETCH
FROM
**GROUP**
HAVING
IN
INDEX
INNER
IS
**JOIN**
KEY
LEADING
LEFT
LENGTH
LIKE
LOCATE
LOWER
MAX
MEMBER
MIN
MOD
NEW
NOT
NULL
NULLIF
OBJECT
OF
OR
ORDER
OUTER
POSITION
**SELECT**
SET
SIZE
SOME
SQRT
SUBSTRING
SUM
THEN
TRAILING
TRIM
TRUE
TYPE
UNKNOWN
**UPDATE**
UPPER
VALUE
WHEN
**WHERE**

# Was ist JPQL?

- Deklarative, SQL-ähnliche Abfragesprache
- Eher auf Arbeit an Java-Objekten als auf Arbeit mit relationalen Datenschemata zugeschnitten
- Leicht zu lernen und präzise genug, um in nativen Datenbankcode übersetzt werden zu können
- Hersteller-unabhängig und portabel
- Wird zur Laufzeit in natives SQL übersetzt
- Ausweg natives SQL, falls JPQL nicht ausreicht

# JPQL am Beispiel

- Einfache Query ohne Parameter

```
SELECT c FROM Course c ORDER BY c.name
```


- Einfache Query mit Parametern

```
SELECT u FROM User u WHERE u.userName = :userName
```

# Query-Parameter

- Parameter müssen der Query vor ihrer Ausführung mit Query.setParameter() übergeben werden:

```
public List<Order> findAllOrdersOfCustomer(String customerId) {
   TypedQuery<Order> query = this.entityManager.createQuery(
      "SELECT o FROM Order o WHERE o.customerId = :customerId",
      Order.class);
   query.setParameter( "customerId", customerId );
   return query.getResultList();
}
```

- Angegebener Parametername muss mit Parameternamen aus Statement übereinstimmen
- Parameterangabe über Index möglich (aber nicht empfehlenswert)

# Paginierung

- Ermöglicht „Blättern“ durch große Ergebnis-mengen
  - Query.setMaxResult(): Anzahl an Entitäten pro Seite
  - Query.setFirstResult(): Startposition der Seite innerhalb der gesamten Ergebnismenge

```
public List<User> getAllUsers(int firstPosition, int pageSize) {
    TypedQuery<User> query = this.entityManager.createNamedQuery(
        User.QUERY_ALL,  User.class);
    query.setFirstResult(firstPosition);
    query.setMaxResults(pageSize);
    return query.getResultList();
}
```

# Fragen?

# ANHANG

# Quellen

- Eric Jendrock et. al.: ***The Java EE 7 Tutorial Part VIII Persistence***
  http://docs.oracle.com/javaee/7/tutorial/partpersist.htm
  Oracle September 2014

# Kontakt

**Michael Theis**
Lehrbeauftragter Hochschule München

email michael.theis@hm.edu
mobile + 49 170 5403805
web http://www.tschutschu.de/Lehrauftrag.html