

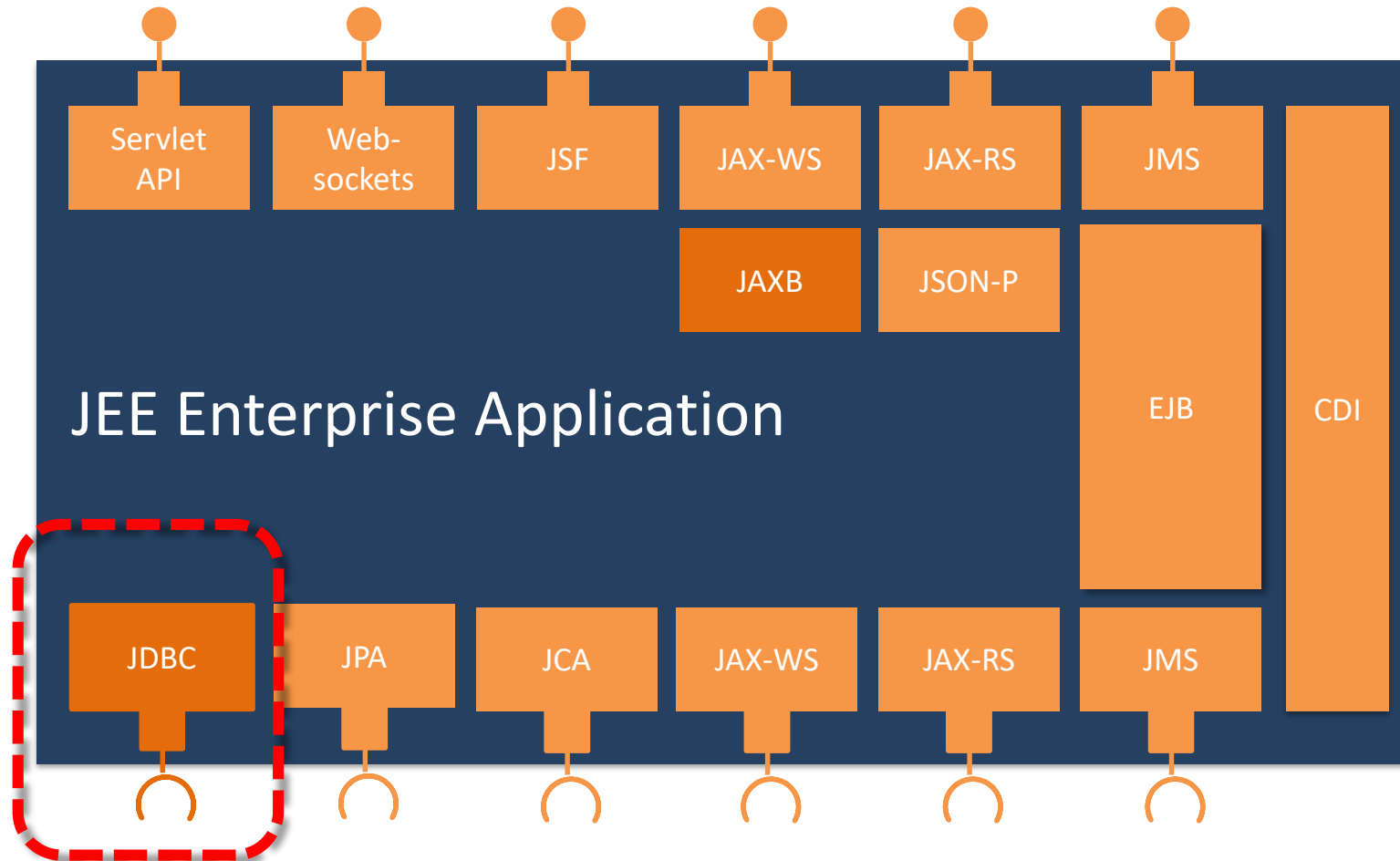
Zugriff auf relationale Datenbanken mit JDBC und JPA

FWP Aktuelle Technologien zur Entwicklung
verteilter Java-Anwendungen

Einführung in die Grundbegriffe von JDBC

GRUNDBEGRIFFE VON JDBC

Kontext: JEE Technologien



JDBC API

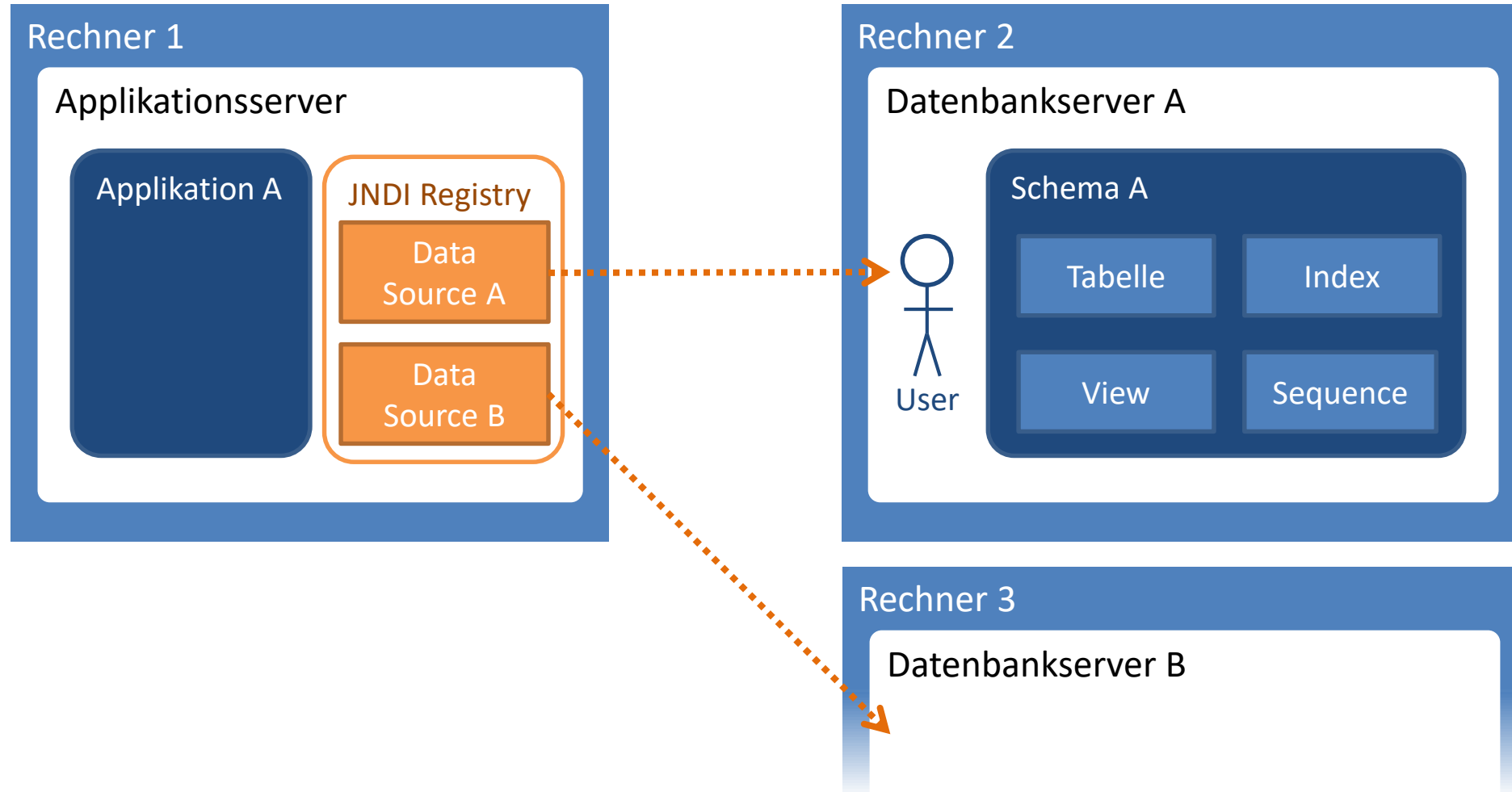
- **Java Database Connectivity (JDBC) API** definiert Standard für Zugriff auf relationale Datenbanken
- **javax.sql.DataSource** repräsentiert eine abstrakte Factory für Verbindungen zu Datenbanken
- **javax.sql.Connection** repräsentiert eine Verbindung zu einer bestimmten Datenbank
- SQL-Anweisungen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

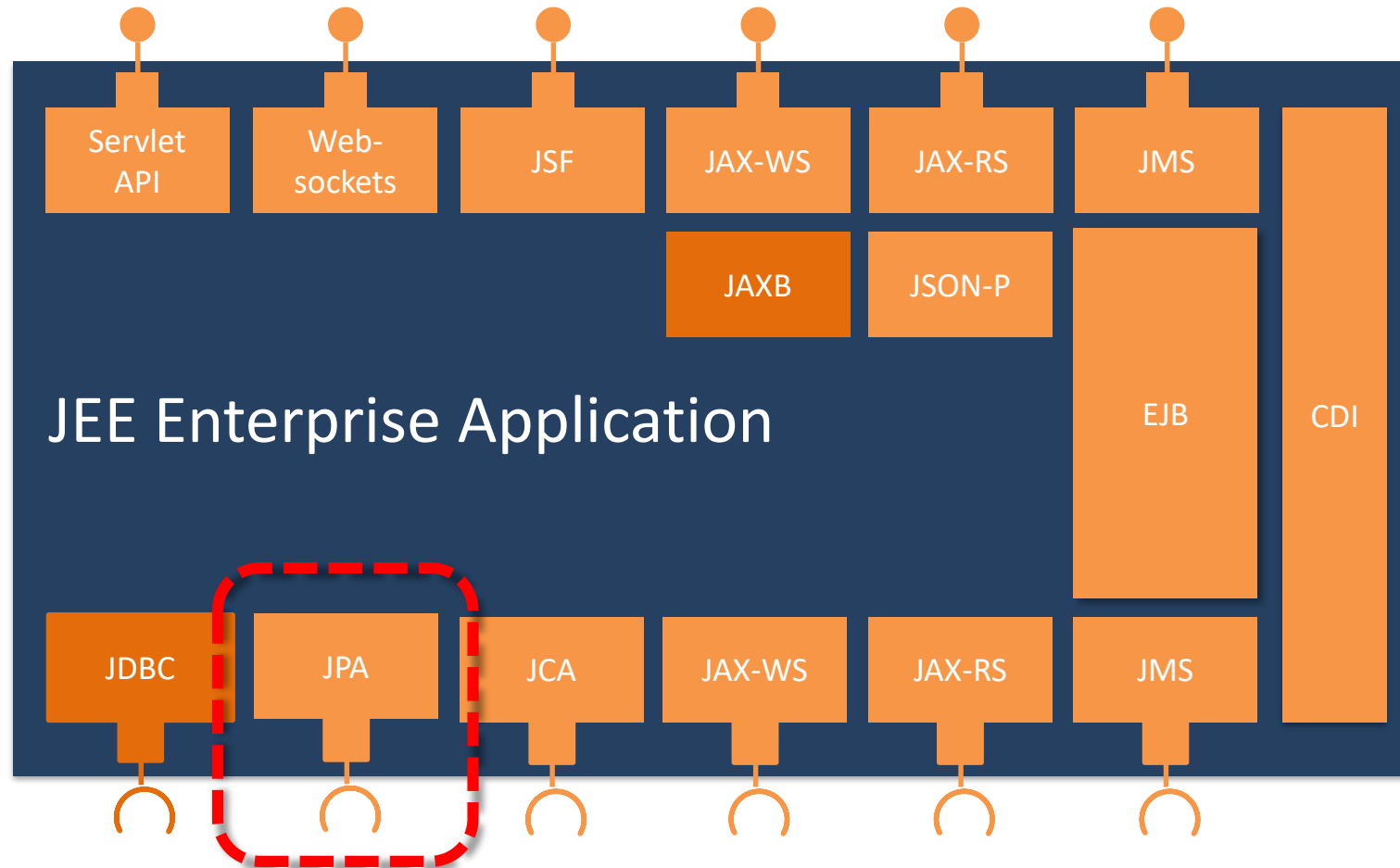
DataSource als Tor zur Datenbank



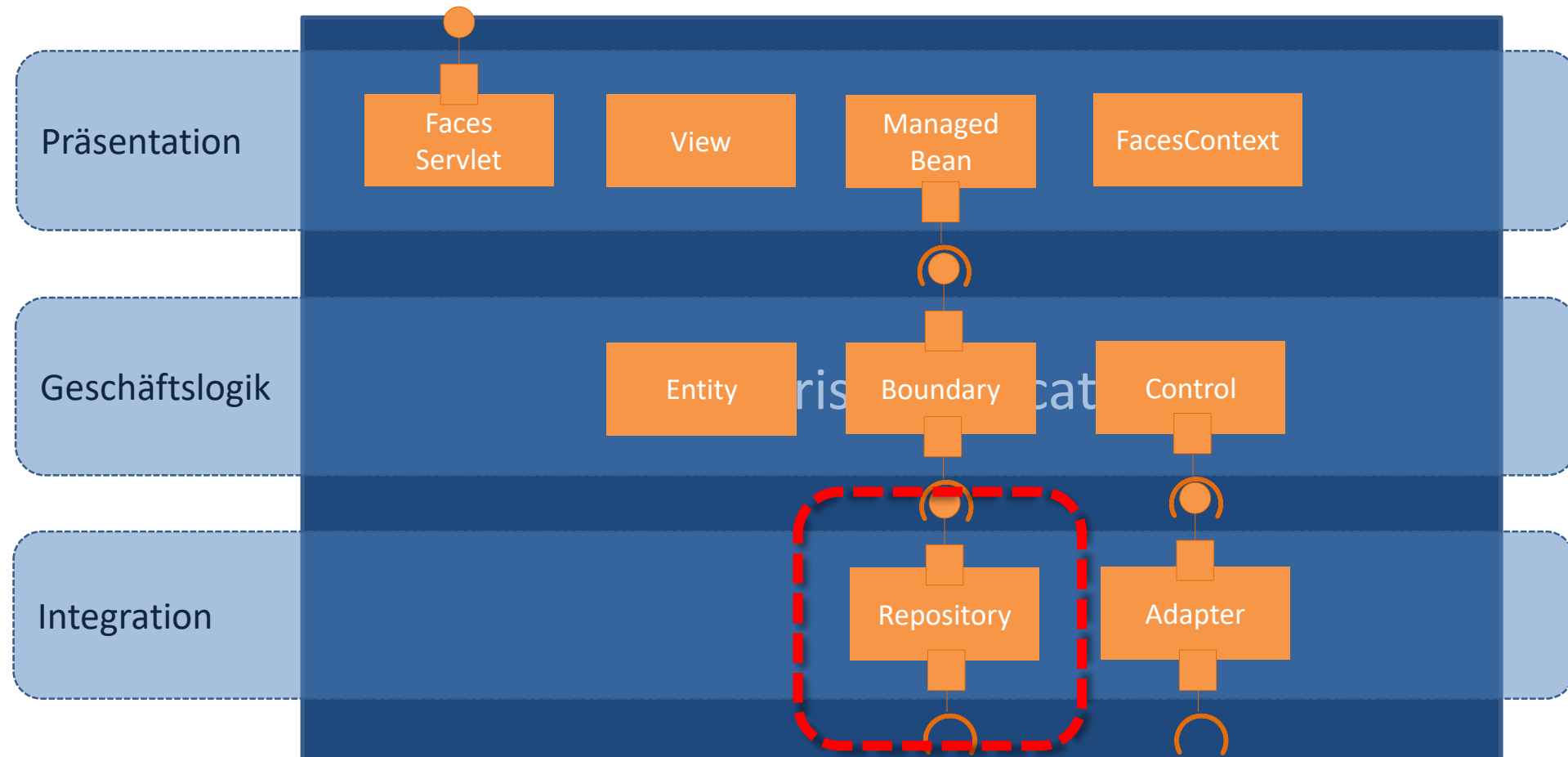
Entity, Persistence Context und Persistence Unit als Grundbausteine

GRUNDBEGRIFFE DER JAVA PERSISTENCE API

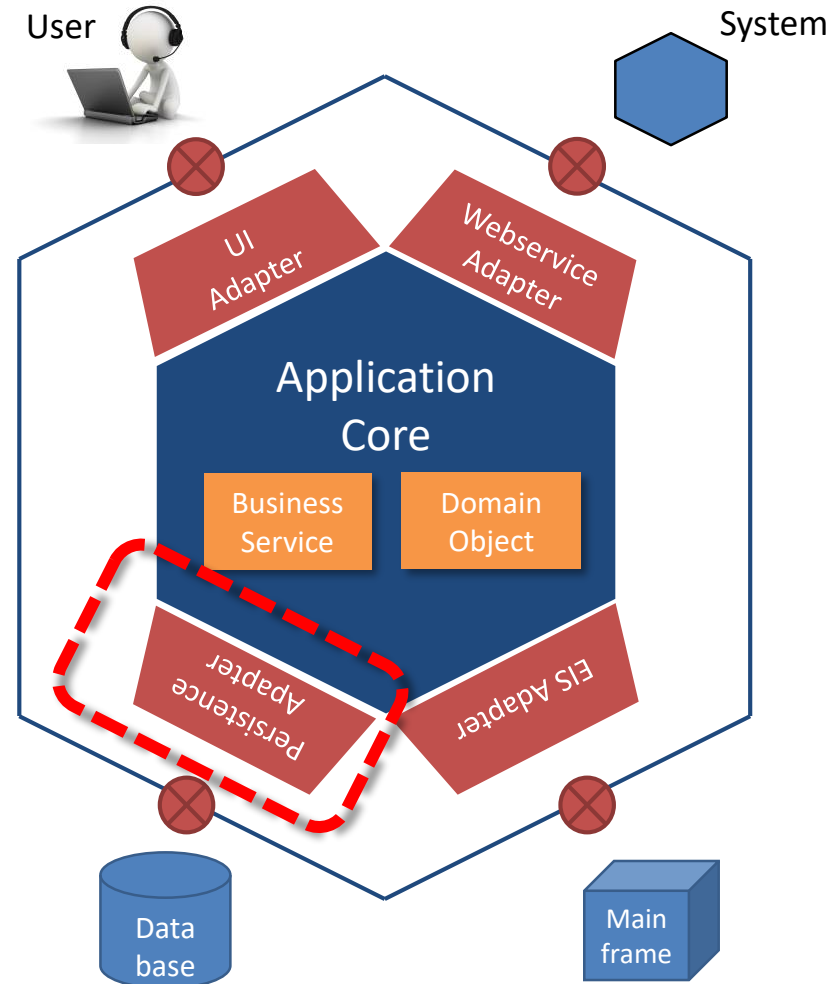
Kontext: JEE Technologien



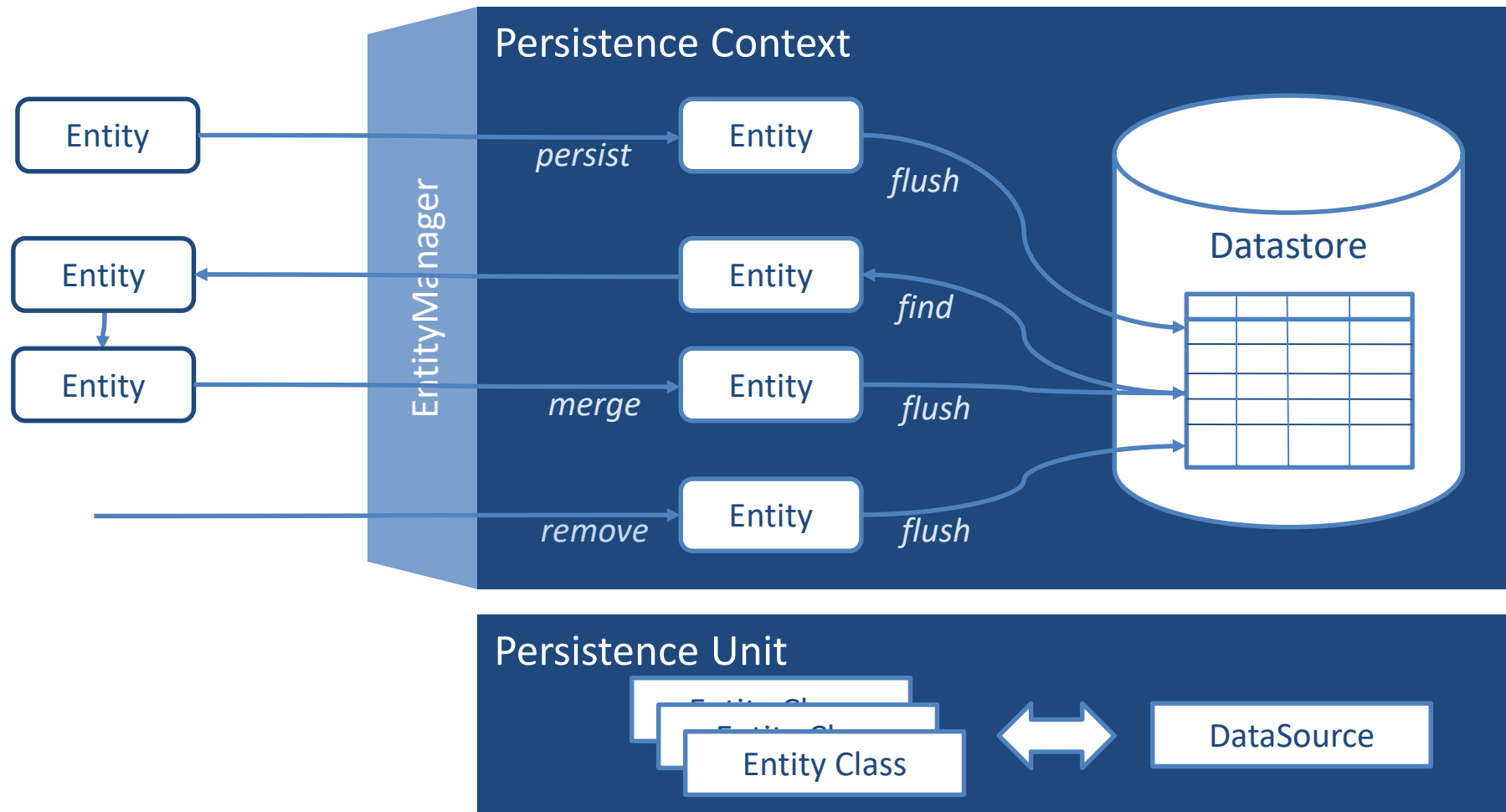
Kontext: Einheitliches Schichtenmodell



Kontext: Hexagonale Architektur



Java Persistence API aus 10000m Höhe



Entitäten (Entity)

- Entity = POJO + @Entity
- Verwaltungte 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

- 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

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

Container Managed Entity Manager

- Container injiziert Entity Manager in mit `@PersistenceContext` markiertem Feld eines Stateless Session Beans
- 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;
    ...
}
```

Application-Managed Entity Manager

- Container injiziert Entity Manager Factory in ein mit `@PersistenceUnit` markiertes Feld eines Stateful Session Beans
- Entity Manager zeigt auf einen erweiterten Persistence Context
- Komponente muss Entity Manager über Factory erzeugen
- Komponente bestimmt Lebensdauer des Persistence Contexts und Zeitpunkt der Synchronisation mit der Datenbank

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

Persistence Unit

- 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

Abbildung von Klassen und Feldern auf Tabellen und Spalten

OBJECT RELATIONAL MAPPING

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 Primärschlüsselfeld der Entityklasse; `@Embeddable` markiert Primärschlüsselklasse
 - ◉ `@IdClass` definiert Primärschlüsselklasse; mehrere Felder der Entityklasse werden mit `@Id` markiert

Vererbung bei Entitäten

(Abstract) Entites

- Basistyp für Entitäten ist abstrakte Entität
- Basistyp muss mit `@Entity` annotiert sein
- Vererbungsstrategie bestimmt Ablage des Basistyps
- Nach Entität-Basistypen kann abgefragt werden

Mapped Supertypes

- Basistyp für Entitäten ist keine Entität, besitzt aber persistierbare Felder
- Basistyp muss mit `@MappedSuperclass` annotiert werden
- Mapped Supertypes haben keine eigenen Tabellen
- Mapped Supertypes können nicht abgefragt werden

Non-Entities

- Basistyp ist keine Entität und besitzt keine persistierbaren Felder
- Basistyp bei Persistierung komplett ignoriert

Abbildung von Vererbung über @Inheritance

Single Table

- Eine Tabelle für alle Klassen
- Diskriminator-Spalte (`@DiscriminatorColumn`) bestimmt konkreten Typ
- Gute Unterstützung von polymorphen Beziehungen
- Typfremde Spalten müssen NULL unterstützen

Table per Concrete Class

- Eine Tabelle für jede konkrete Klasse
- Tabelle enthält alle eigenen plus geerbte Felder der Klasse
- Schlechte Performance bei polymorphen Queries (UNION)
- Wird nicht von allen Implementierungen unterstützt

Joined Subclass

- Eigene Tabelle für jede abstrakte und jede konkrete Klasse
- Tabelle enthält nur die eigenen Felder der Klasse
- Gute Performance bei polymorphen Queries (JOIN)
- Manchmal Diskriminator-Spalte (`@DiscriminatorColumn`) erforderlich

7 Formen von Beziehungen

- 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)

Auflösen von Beziehungen

- Fetch-Strategie bestimmt, ob Beziehung während eines Lesezugriffs aufgelöst wird
 - ◉ **LAZY** (default): Beziehung wird beim Lesen nicht aufgelöst, sondern erst beim ersten Zugriff auf das Feld
 - ◉ **EAGER**: Beziehung wird beim Lesen aufgelöst
- Programmatische Kontrolle des Fetch-Verhaltens über Entitäten-Graphen (**Entity Graph**) möglich
- In JPQL kann Fetch-Verhalten über **FETCH JOIN**-Klausel beeinflußt werden

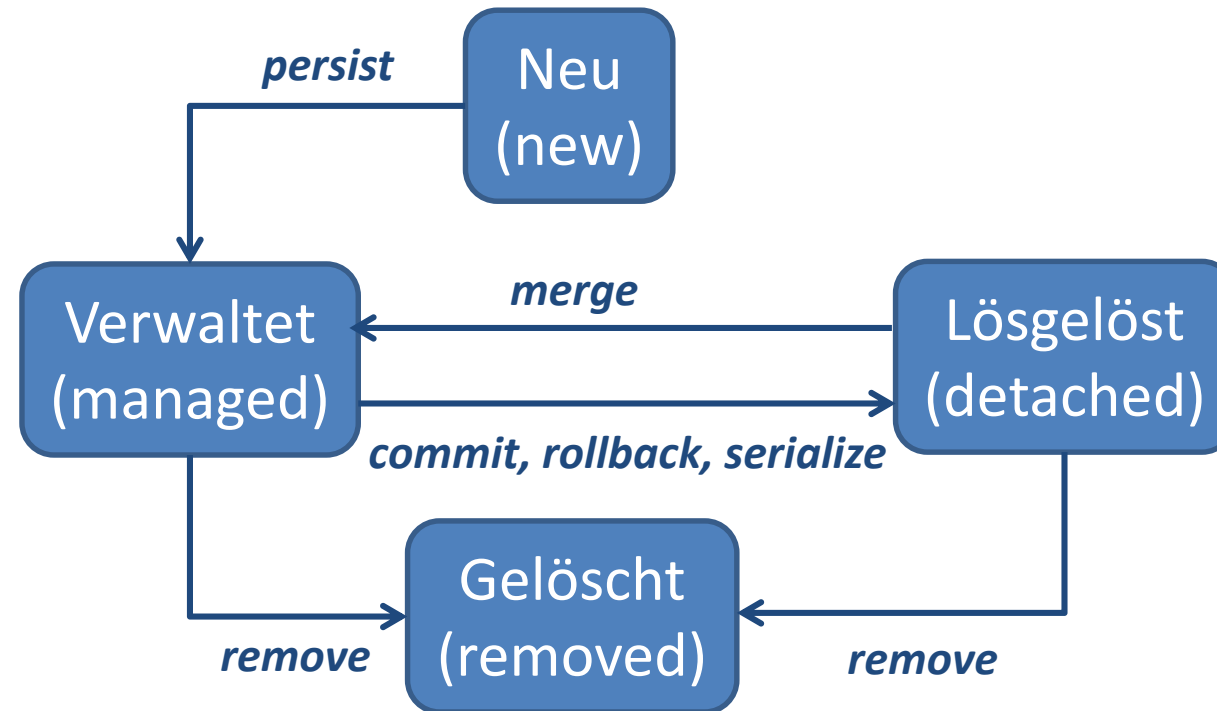
CRUD-Operationen auf Entitäten und deren Auswirkungen auf den Zustand einer Entität

DAS LEBEN EINER ENTITÄT

Zustände einer Entität (I)

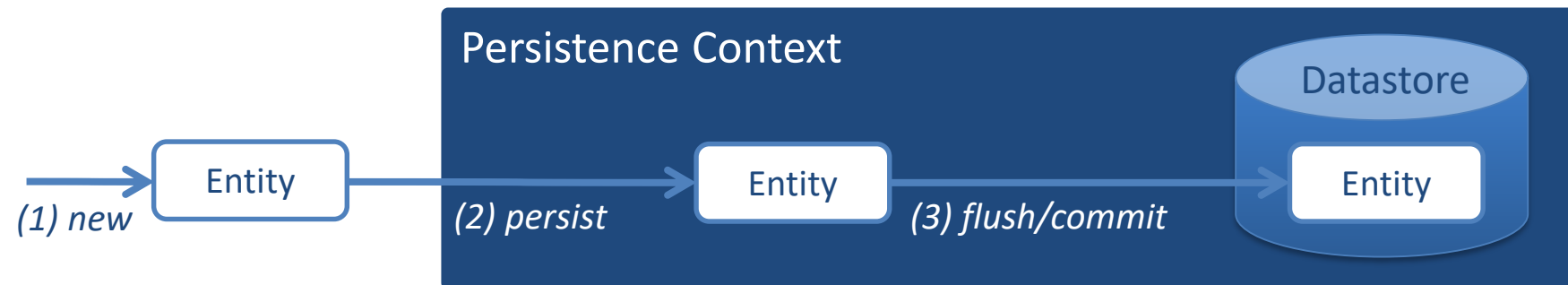
- **Neu (new)**: Entität besitzt keine persistente Identität und ist mit keinem Persistence Context verknüpft
- **Verwaltet (managed)**: Entität besitzt eine persistente Identität und ist mit einem Persistence Context verknüpft
- **Losgelöst (detached)**: Entität besitzt eine persistente Identität und 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 (II)



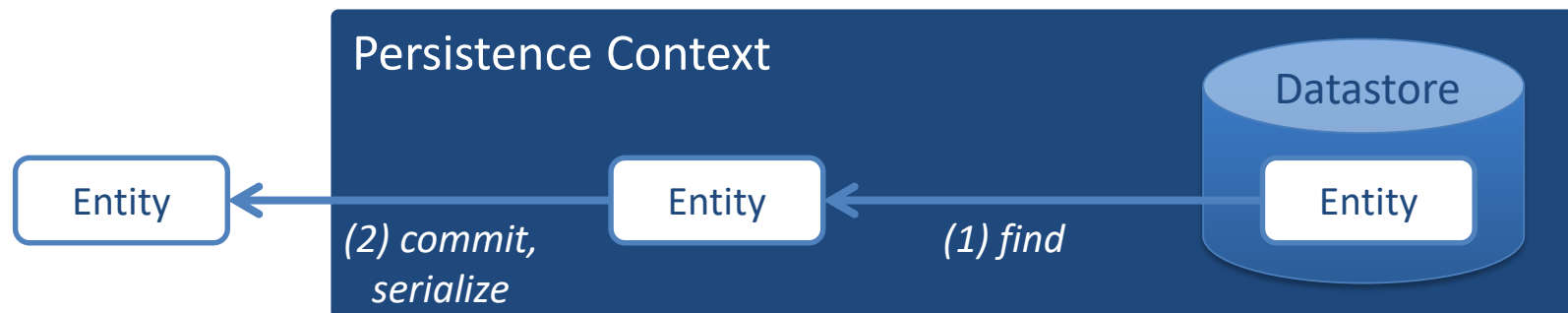
Neue Entität hinzufügen

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



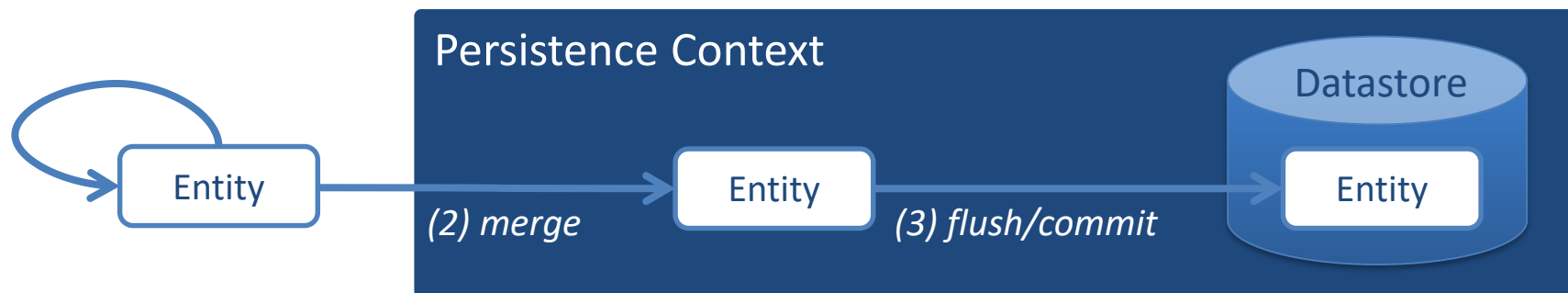
Lesen einer bestehenden Entität

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



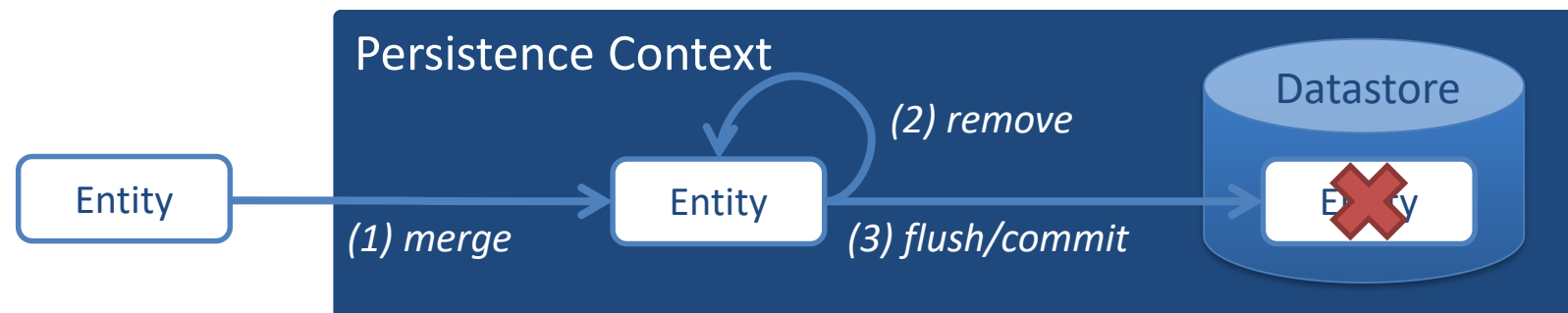
Aktualisieren einer bestehende Entität

- Änderungen an verwalteten Entitäten werden automatisch erkannt
- Losgelöste Entitäten müssen erst mit `EntityManager.merge()` in den Persistence Context gebracht werden
- Endgültige Aktualisierung erfolgt mit `Commit` der aktuellen Transaktion



Löschen einer bestehenden Entität

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



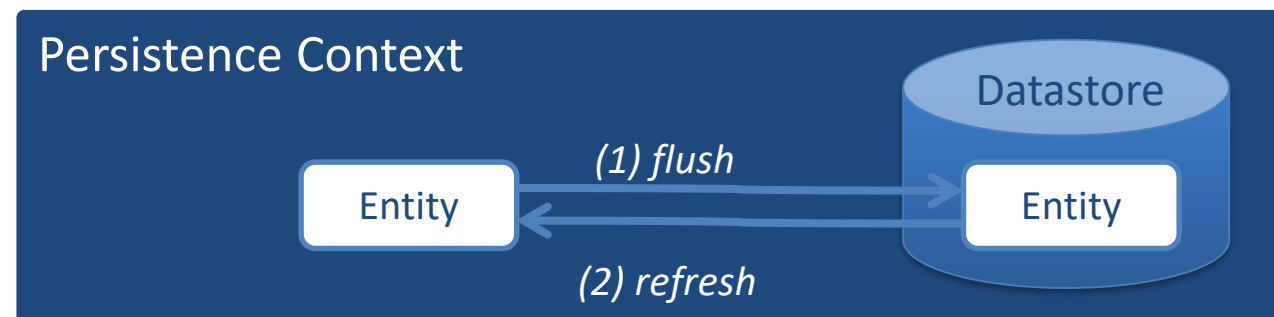
Losgelöste Entitäten

- Entitäten werden häufiger als gedacht vom Persistence Context 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 Abfragen von Entitäten

QUERIES

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 JPQL-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();
}
```

Gegenüberstellung der Query-Arten

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

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
```

Klassenname statt
Tabellenname

Feldname statt
Spaltenname

- Einfache Query mit Parametern

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

Parametername

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 (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

- Beispiel-Code auf GitHub unter <https://github.com/mikeT92/jeetrain>
MAVEN-Projekt **jeedemo** MAVEN-Modul **jeedemo-jpa**
- Eric Jendrock et. al.: *The Java EE 7 Tutorial Part VIII Persistence*
<http://docs.oracle.com/javaee/7/tutorial/partpersist.htm>
Oracle September 2014
- Stefan Tilkov: *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*
dpunkt.verlag GmbH; 3. Auflage (30. April 2015); ISBN 978-3864901201
- ThoughtWorks Prakash Subramaniam: *REST API Design – Resource Modeling*
<https://www.thoughtworks.com/de/insights/blog/rest-api-design-resource-modeling>
abgerufen am 08.03.2017
- Stormpath Les Hazelwood: *REST+JSON API Design - Best Practices for Developers*
<https://youtu.be/hdSrT4yjS1g>
abgerufen am 08.03.2017



Kontakt



Michael Theis

Lehrbeauftragter Hochschule München

email michael.theis@hm.edu

mobile + 49 170 5403805

web <http://www.tschutschu.de/Lehrauftrag.html>