Datenzugriffskomponenten mit JPA

FWP Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Theis Michael - Senior Developer UniCredit Global Information Services S.C.p.A

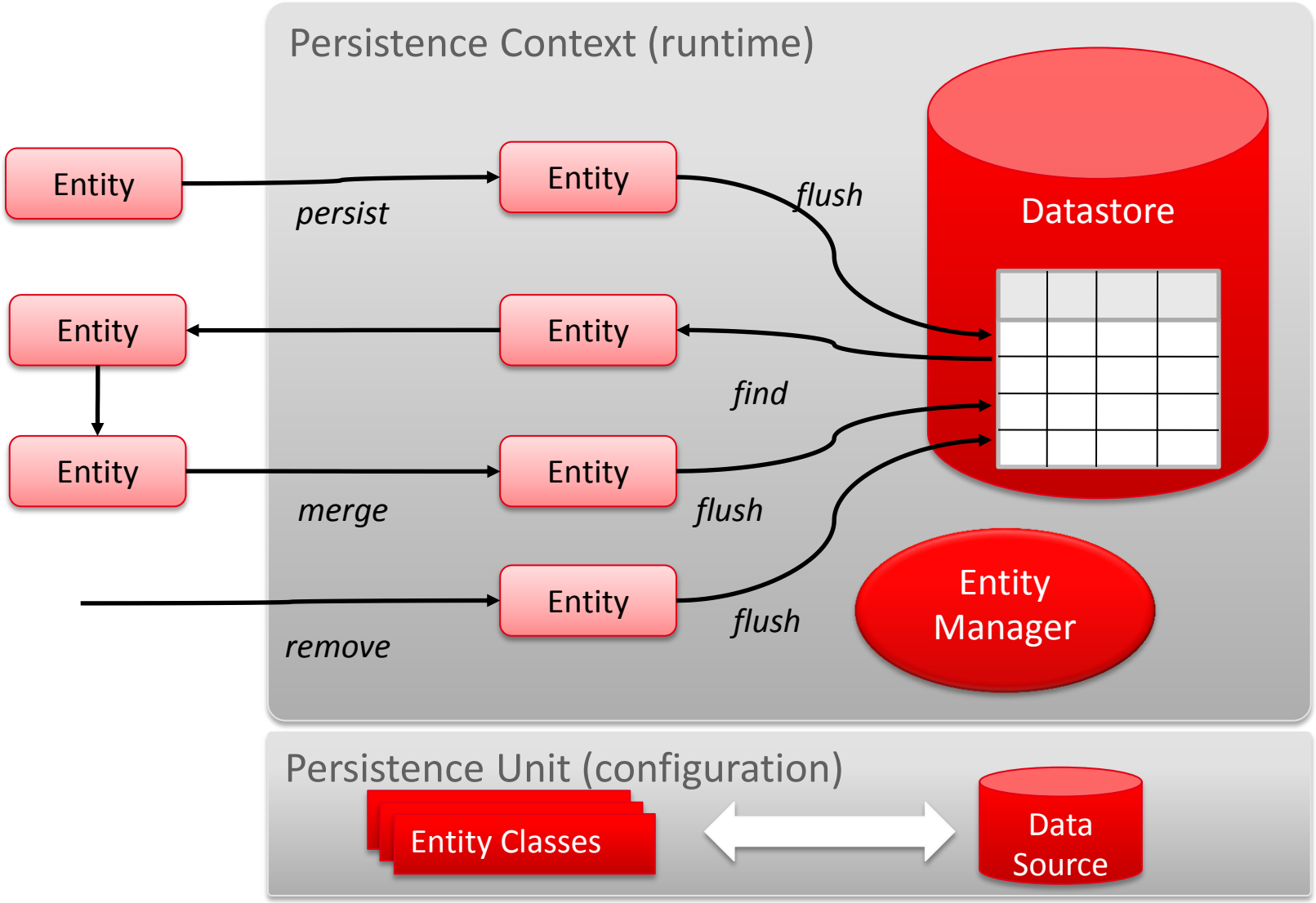
Sommersemester 2012

Datenzugriffskomponenten mit JPA

- Einführung in die Java Persistence Architecture
- Datenzugriffskomponenten mit Java Persistence Architecture

Einführung in die Java Persistence Architecture

Java Persistence Architecture



Entitäten

- Entitäten (ehemals Entity Beans) sind POJOs:
  - Entitäten müssen keine EJB-Interfaces mehr implementieren
  - Es müssen keine Deployment-Deskriptoren mehr erstellt werden (mit Ausnahme des Persistence Unit Descriptors persistence.xml)
  - Mit der Annotation @Entity wird ein beliebiges POJO als Entity Bean gekennzeichnet.
- Verwaltete und nicht-verwaltete Entitäten
  - Verwaltete Entitäten (managed/attached) werden von einem EntityManager überwacht, der automatisch Änderungen erkennt und diese mit dem zugrundeliegenden Datenspeicher synchronisiert.
  - Nicht verwaltete Entitäten (unmanaged/detached) werden von keinem Entity Manager überwacht und Änderungen daher auch nicht abgespeichert.

Persistence Context

- Umfasst alle zu einem bestimmten Zeitpunkt verwalteten Entitäten, die über in einem bestimmten Datenspeicher abgelegt sind
  => dynamische Zuordnung und Synchronisierung von Java-Objekten und einem Datenspeicher zur Laufzeit
- Persistenz-Kontexte werden von Entity-Managern verwaltet.
- Der Entity Manager erkennt alle Änderungen an Entities sowie Zugänge und Abgänge von Entities und synchronisiert diese mit dem zugrundeliegenden Datenspeicher.
- 2 unterschiedliche Arten von Persistenz-Kontexten:
  - Transaktionsgebundene Persistenz-Kontexte (transaction-scoped) leben nur für die Dauer einer Transaktion.
  - Erweiterte Persistenz-Kontexte (extended) überdauern eine Transaktion und die darin enthaltenen Entitäten werden weiterhin verwaltet.

Entity Manager

- Entitäten werden von Entity Managern in einem Persistenz-Kontext verwaltet.
- Entity Manager bieten die Methoden zur Manipulation von Persistenz-Kontexten.
- 2 unterschiedliche Arten von Entity Managern:
  - Container-managed Entity Manager werden automatisch vom EJB-Container an alle Applikationskomponenten propagiert (über Dependency Injection), die innerhalb einer Transaktion einen Entity Manager benötigen.
    - `@PersistenceContext EntityManager em;`
  - Application-managed Entity Manager werden über die Applikations-komponenten selber erzeugt; mit jedem Entity Manager wird ein neuer lokaler Persitenz-Kontext erzeugt.
    - ```
      @PersistenceUnit EntityManagerFactory emf;
      EntityManager em = emf.createEntityManager();
      ```
      ```

Persistence Units

- Persistence Units definieren alle von Entity Managern verwalteten Entitäten-Klassen einer Applikation, die in dem gleichen Datenspeicher abgelegt werden sollen
  => statische Zuordnung von Java-Typen zu einem Datenspeicher
- Alle Entitäten-Klassen in einer Persistence Unit werden im gleichen Datenspeicher (über die gleiche DataSource) abgespeichert.
- Persistence Units werden im Form einer XML-Datei namens persistence.xml definiert (DD der Persistence Unit).
- Mehrere Persistence Units können über ihren Namen unterschieden werden.

Datenquellen und Verbindungen

- Interface `javax.sql.DataSource` aus der JDBC API repräsentiert eine abstrakte Factory für Verbindungen zu einem relationalen Datenbanksystem
- Interface `javax.sql.Connection` repräsentiert eine Verbindung zu einer bestimmten Datenbank
  - SQL Anweisungen werden im Kontext einer Connection ausgeführt
- DataSources werden üblicherweise als Ressourcen in einem Applikationsserver konfiguriert, die über JNDI-Lookup ermittelt werden können
- Konkrete Implementierungen von DataSource werden von den Herstellern datenbank-spezifischer JDBC-Treiber zur Verfügung gestellt

Object Relational Mapping

- Entitäten sind POJOs
  - POJO + Annotation @Entity + Persistence Unit = Entität
  - Alternativ: POJO + orm.xml (XML Mapping) + Persistence Unit = Entität
- Abbildung von Klassen und Felder auf Tabellen und Spalten
  - Mit der Annotation @Table werden Klassen auf Tabellen gemappt.
  - Mit der Annotation @Column werden Felder auf Spalten gemappt.

```
@Entity
@Table(name = "T_ORDER")
public class Order implements Serializable {
   private long id;
   @Id
   @Column(name="ORDER_ID”)
   public long getId() { return this.id; }
   public void setId(long id) { this.id = id; }
}
```

Identität und Primarschlüssel

- Jede Entität muss analog dem Primärschlüssel einer Tabelle eine Identität besitzen
- Abbildung eines Feldes auf einen einspaltigen Primarschlüssel
  - Annotation @Id markiert ein Feld einer Entität als Primarschlüssel.
  - Über die Annotation @GeneratedValue ist die automatische Generierung der Primärschlüssel nach verschiedenen Strategien möglich
    - **Beispiel für einen über eine SEQUENCE generierten Primärschlüssel**

```
@Entity @Table(name="T_ORDER)
@SequenceGenerator(name="ISHOP_SEQUENCE",
sequenceName="SEQ_ISHOP")
public class Order {
   @Id @GeneratedValue(strategy=GenerationType.SEQUENCE,
                       generator="ISHOP_SEQUENCE")
   public int getId() {...}
```

Komplexe Primarschlüssel

- Abbildung mehrerer Felder auf einen mehrspaltigen Primarschlüssel
  - Entweder über @IdClass/@Id oder @EmbeddedId/@Embeddable oder @EmbeddedId/@AttributeOverride(s) lassen sich mehrere Felder mehreren Spalten eines zusammengesetzten Primärschlüssels zuordnen.

Beziehungen

- 7 unterschiedliche Beziehungen zwischen Entitäten sind möglich:
  - 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)

Beziehungen

- Beispiel für eine bidirektionale 1-zu-n Beziehungen
  - Eine Bestellung besteht aus mehreren Einzelposten

```
@Entity public class Order {
   @OneToMany(cascade=CascadeType.ALL,
                  fetch=FetchType.EAGER, mappedBy="order")
   public Collection<OrderItem> getOrderItems() { ... }
}

@Entity public class OrderItem {
         @ManyToOne @JoinColumn(name="ORDER_ID")
 public Order getOrder() { ... }
}
```

Lebenszyklus von Entitäten

- Zustände von Entitäten
  - new: Neue Entitäten haben keine persistente Identität und sind noch nicht mit einem Persistence Context verknüpft.
  - managed: Verwaltete Entitäten haben eine persistente Identität und sind mit einem Persistence Context verknüpft.
  - detached: Losgelöste Entitäten haben eine persistente Identität, sind aber nicht mit einem Persistence Context verknüpft.
  - removed: Entfernte Entitäten haben eine persistente Identität, sind mit einem Persistence Context verknüpft und sind für die Löschung aus dem Datenspeicher vorgemerkt.

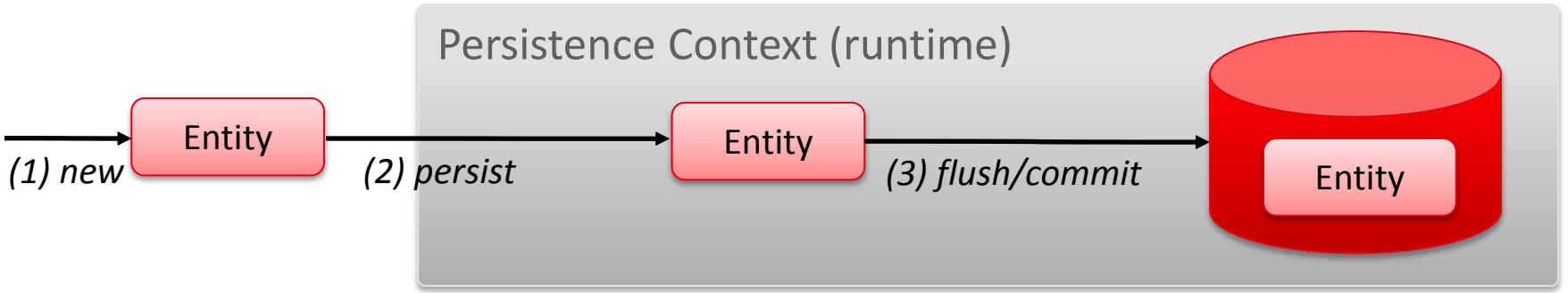
Lebenszyklus von Entitäten

- Zustände von Entitäten
  - ***new***: Neue Entitäten haben keine persistente Identität und sind noch nicht mit einem Persistence Context verknüpft.
  - ***managed***: Verwaltete Entitäten haben eine persistente Identität und sind mit einem Persistence Context verknüpft.
  - ***detached***: Losgelöste Entitäten haben eine persistente Identität, sind aber nicht mit einem Persistence Context verknüpft.
  - ***removed***: Entfernte Entitäten haben eine persistente Identität, sind mit einem Persistence Context verknüpft und sind für die Löschung aus dem Datenspeicher vorgemerkt.

Neue Entitäten speichern

- Erzeugen von neuen Entitäten (new->managed)
  - Neue Instanzen von Entity Beans werden einfach über den new-Operator erzeugt und über EntityManager.persist() abgespeichert.

```
Order order = new Order();
this.repository.addOrder(order);
…
@PersistenceContext EntityManager entityManager;
public Order addOrder(Order order) {
   this.entityManager.persist(order); // add to persistence context
   this.entityManager.flush(); // synchronize persistence context -> DB
   this.entityManager.refresh(); // synchronize DB -> persistence context
   return order; }
```

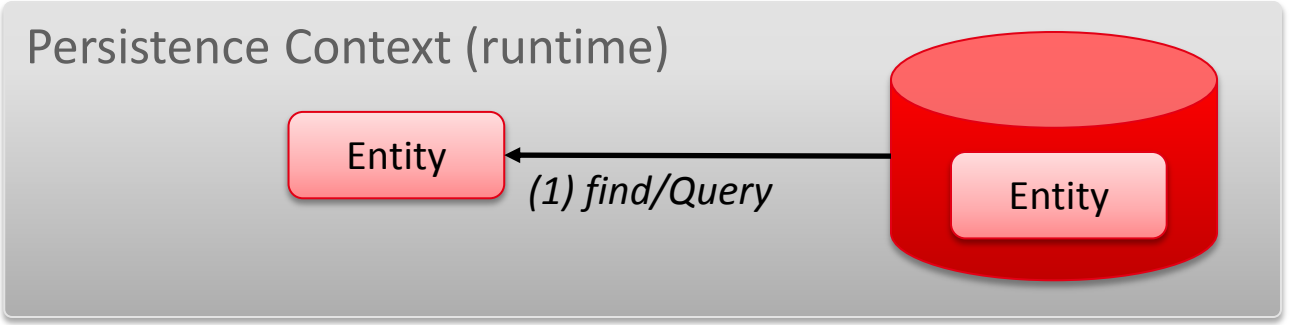


Gespeicherte Entitäten lesen

- Lesen von Entitäten
  - Bestehende Instanzen können über ihrem Primary Key mit EntityManager.find oder EntityManager.getReference gelesen werden

```
@PersistenceContext EntityManager entityManager;
public Order getOrderById(long orderId) {
   return this.entityManager.find(orderId, Order.class);
}
```

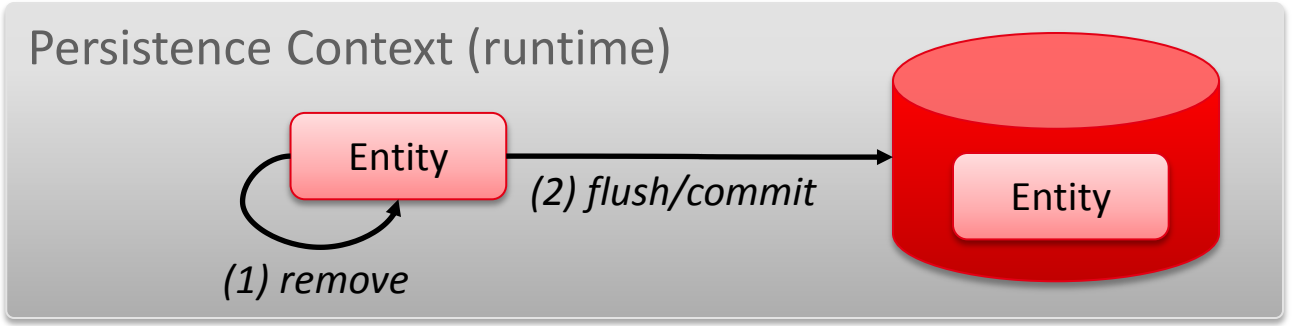
- Komplexere Abfragen sind mit über Queries möglich (siehe Queries)



Entitäten löschen

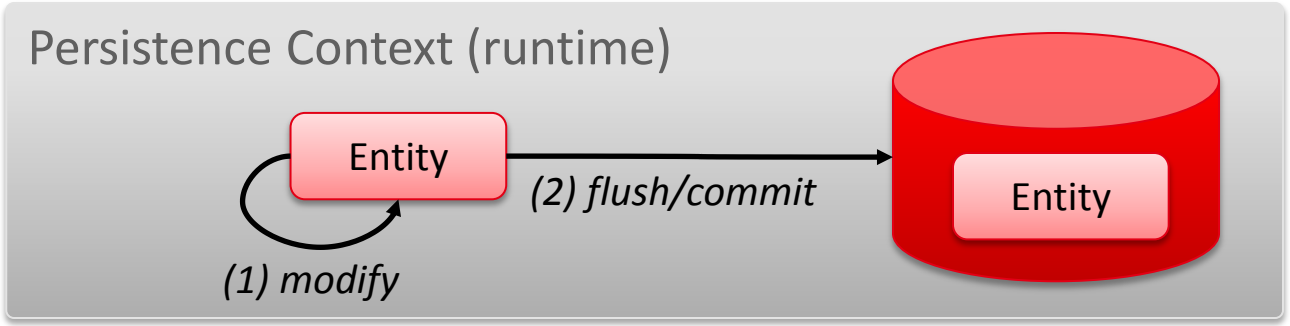
- Löschen von Entitäten (managed->removed)
  - Bestehende Entity Beans werden mit EntityManager.remove zur Löschung vorgemerkt und mit dem Abschluss der aktuellen Transaktion tatsächlich aus

```
@PersistenceContext EntityManager entityManager;
public void removeOrder(Order order) {
   Order merged = this.entityManager.merge(order);
   this.entityManager.remove(merged);
}
public void removeOrderById(long orderId) {
   Order order = this.entityManager.getReference(orderId, Order.class);
   this.entityManager.remove(order);
}
```



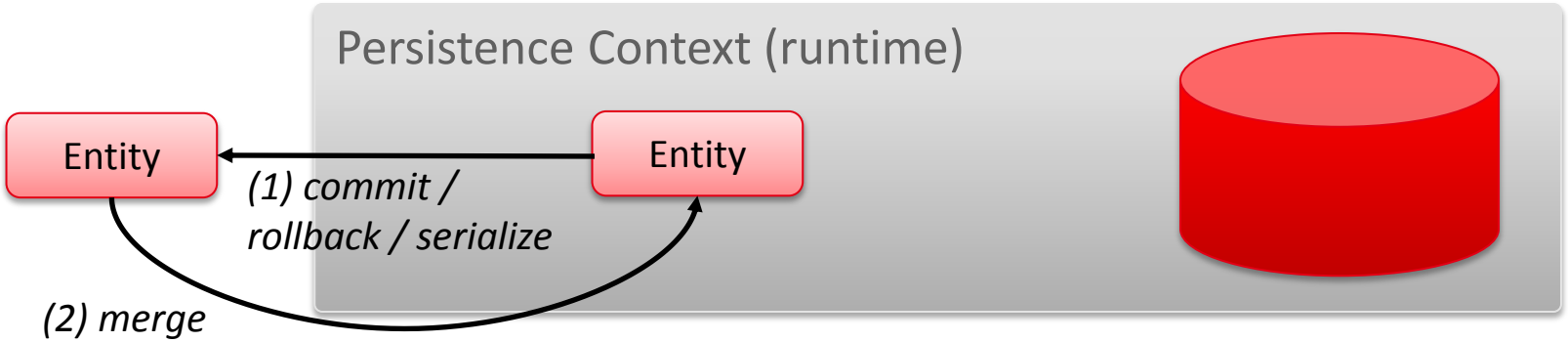
Entitäten ändern

- Ändern von Entitäten
  - Änderungen an verwalteten Entity Beans werden automatisch vom EntityManager erkannt und bei Abschluss der aktuellen Transaktion in den Datenspeicher geschrieben.
  - Die am Ende einer Transaktion oder nach EntityManager.flush geschriebenen Änderungen im Datenspeicher werden nicht automatisch in den verwalteten Entitäten nachgezogen. Hierfür muss EntityManager.refresh aufgerufen werden.



Losgelöste Entitäten

- Loslösen von Entitäten (managed -> detached)
  - Commit oder Rollback der aktuellen Transaktion, das Zurücksetzen des Persistence Contexts, das Schließen eines EntityManagers, Serialisierung über eine Remote-Schnittstelle führt zu einer Loslösung einer Entität vom Persistence Context.
  - Das Abspeichern von Änderungen ist nicht mehr gewährleistet
- Synchronisierung von losgelösten Entitäten (detached -> managed)
  - Losgelöste Entity Beans müssen erst wieder mit EntityManager.merge mit einem Persistence Context verknüpft werden.



Queries

- Queries werden durch das Interface `javax.persistence.Query` repräsentiert
- JPA unterstützt zwei Abfragesprachen
  - ***Java Persistence Query Language (JPA QL):*** portable objekt-orientierte Abfragesprache eng angelehnt an SQL
  - ***Standard Query Language (SQL):*** standardisierte Abfragesprache der darunterliegenden relationalen Datenbank, häufig mit spezifischen Erweiterungen
- Queries werden über den `EntityManager` erzeugt und liefern bei Ausführung die Ergebnisse als typisierte Liste zurück
- Neben gewöhnlichen Abfragen können über Queries auch sog. Bulk-Updates und –Deletes ausgeführt werden

Dynamische Queries

- Queries können über EntityManager.createQuery mit einem JPAQL SELECT Statement erzeugt werden
- Da so erzeugte Queries JPA erst zur Laufzeit bekannt gemacht werden, werden solche Queries als ***dynamische*** Queries bezeichnet

```
@PersistenceContext EntityManager entityManager;
...
 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();
}
```

Named Queries

- Queries können über `@NamedQueries` zur Kompilezeit an Entityklassen gebunden werden
- `@NamedQuery` verknüpft ein Querystatement mit einem eindeutigen Namen
- Diese vordefinierten Queries werden ***named queries*** genannt da sie nur über ihren Namen erzeugt werden können

```
@NamedQuery( "findAllOrdersOfCustomer",
      "SELECT o FROM Order o WHERE o.customerId = :customerId" )
@Entity public class Order { ... }
```

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

Abfragesprache JPA QL

- Deklarative Abfragesprache mit großer Ähnlichkeit zu SQL
- Eher auf die Arbeit Java-Objekten als die Arbeit mit relationalen Datenschemata zugeschnitten
- Leicht zu lernen und präzise genug, um in nativen Datenbankcode übersetzt werden zu können
- JPA QL ist hersteller-unabhängig und portabel
  - die Übersetzung von JPA QL in einen hersteller-spezifischen SQL-Code wird von JPA beim Ausführen einer Query übernommen
- Manchmal ist JPA QL nicht ausreichend
  - Stored Procedures können nur über natives SQL ausgeführt werden
  - JPA bietet die volle Unterstützung von nativem SQL über native Queries

JPA QL 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

Einfache Queries (Beispiele)

- Parameterlose Query, die eine Liste von Entitäten zurückliefert

```
public List<Course> getCourses() {
    TypedQuery<Course> coursesQuery =
        this.entityManager.createQuery(
            "SELECT c FROM Course c ORDER BY c.name", Course.class);
    return coursesQuery.getResultList();
}
```

- Query mit Parameter, die eine einzelne Entität zurückliefert

```
public User getUserByName(String userName) {
    TypedQuery<User> userByNameQuery =
        this.entityManager.createQuery(
            "SELECT u FROM User u WHERE u.userName = :userName", User.class);
    userByNameQuery.setParameter("userName", userName);
    User result = null;
    try {
        result = userByNameQuery.getSingleResult();
    } catch (NoResultException ex) {
        // it's OK when there is no user with the specified name,
        // so we simply swallow this exception and return null
    }
    return result;
}
```

Blättern durch große Datenmengen

- Große Ergebnismengen können in logische Seiten unterteilt werden
  - Indem man die Anzahl der Entitäten pro Seite festlegt durch `Query.setMaxResults`
  - Und die Position der Seite innerhalb der gesamten Ergebnismenge bestimmt durch `Query.setFirstResult`

```
public List<Service> getAllServices(int firstPosition, int pageSize) {
    TypedQuery<Service> query = this.entityManager.createQuery(
          "SELECT s FROM Service AS s“,
          Service.class);
    query.setFirstResult(firstPosition);
    query.setMaxResults(pageSize);
    return query.getResultList();
}
```

Datenzugriffskomponenten mit Java Persistence Architecture

Repositories und Adapter

- Datenorientiere Zugriffskomponenten werden im allgemeinen als ***Repository*** bezeichnet, service- oder logik-orientierte eher als ***Adapter***
- In JavaEE ab Version 5 erfolgt der Zugriff auf relationale Datenbanken standardkonform über JPA
- Am einfachsten werden Datenzugriffskomponenten als Stateless Session Beans implementiert

```
@Stateless
public class TaskRepositoryBean {
    @PersistenceContext
    private EntityManager entityManager;
    public Task addTask(Task task) { … }
    public Task getTaskById(long taskId) { … }
    public Task setTask(Task task) { … }
    public Task removeTask(Task task) { … }
    public List<Task> getTasks(int firstPosition, int pageSize) { … }
    …
```

Generische Repositories

- Motivation:
  - Die Schnittstellen und Implementierungen der meisten Repositories unterscheiden sich nur im zugrundeliegenden Entitätentyp
    => Widerspruch zum DRY-Prinzip!
- Lösung:
  - Ein generisches Repository bündelt die für alle Typen gleiche CRUD-Funktionalität in einer generischen abstrakten Basisklasse
  - Typ der Entitätenklasse und deren Primärschlüsselklasse bilden die Typparameter der generischen Basisklasse.
  - Die typspezifischen Queries werden als Named Queries an den Entitätentypen gebunden und der Name der zu verwendenden Query als Parameter übergeben.
  - Queryparameter werden als Map<String, Object> oder als List<QueryParameter> übergeben.

Generische Repositories im Vergleich

Herkömmliche Lösung

```
@Stateless
public class TaskRepositoryBean {
  @PersistenceContext
  private EntityManager entityManager;
  public Task addTask(Task task) { … }
  public Task getTaskById(long taskId) { … }
  public Task setTask(Task task) { … }
  public void removeTask(Task task) { … }
  public List<Task> getTasks(
    int firstPosition,
    int pageSize) { … }
    …
```

Eine generische Basisklasse enthält den gesamten Code für alle Repositories!

Alle konkreten Implementierungen bestehen nur aus diesen Zeilen!

Generische Lösung

```
public class GenericRepository<K, T> {
  public T addEntity(T entity) { … }
  public T getEntityById(K entityId) { … }
  public T setEntity(T entity) { … }
  public void removeEntity(T entity) { … }
  public List<T> queryEntities(
    String queryName,
    Map<String, Object> queryParameters,
    int firstPosition,
    int pageSize) { … }
    …
  // EntityManager/Type provided by subclass
  protected abstract
      EntityManager getEntityManager();
  protected abstract T getEntityType ();
```

```
@Stateless
public class TaskRepositoryBean
    extends GenericRepository<Long, Task> {
  @PersistenceContext
  private EntityManager entityManager;
  protected EntityManager getEntityManager()
      { return this.entityManager;}
  protected Class<Task> getEntityType()
      { return Task.class; }
```

Anhänge

Quellen

- Eric Jendrock et. al.
  The Java EE 6 Tutorial
  http://docs.oracle.com/javaee/6/tutorial/doc/
  Oracle February 2012
- Bill Burke & Richard Monson-Heafel
  Enterprise JavaBeans 3.0, 5th Edition
  O'Reilly Media, Inc. May 2006
  ISBN 978-0-596-00978-6
- Adam Bien
  Real World Java EE Patterns: Rethinking Best Practices
  press.adam-bien.com June 2009
  ISBN 978-0-557-07832-5

Fragen?

Vielen Dank!

Michael Theis
Senior Developer
UniCredit Business Integrated Solutions S.C.p.A.

email michael.theis@hm.edu
phone + 49 89 378-46498
mobile + 49 170 7875474
web http://www.tschutschu.de/Lehrauftrag.html