SBB CFF FFS

Java Persistence API 2.0 Upgrade

Simon Martinelli, simas GmbH

sm@simas.ch - www.simas.ch

simas

SBB CFF FFS

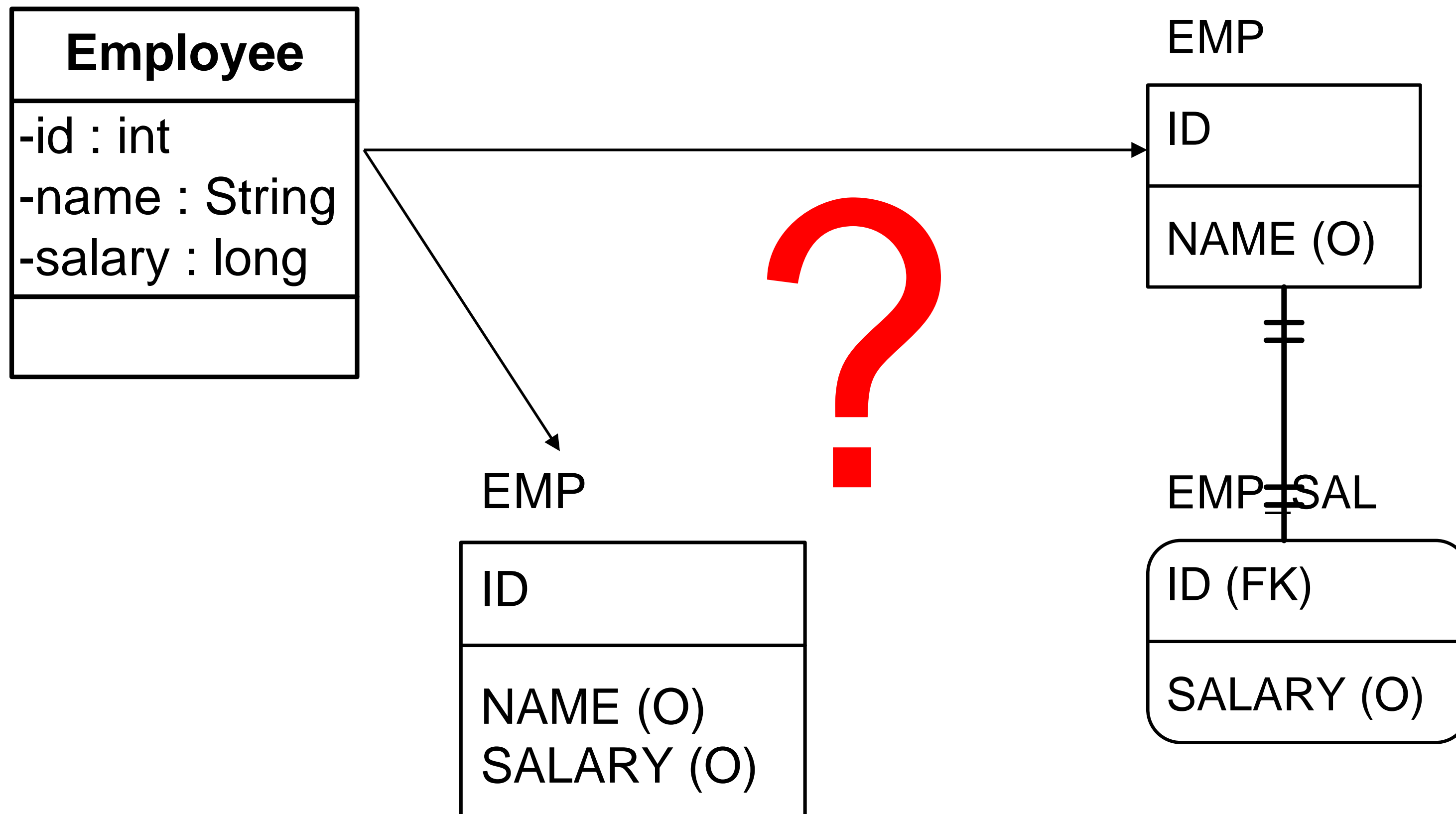
Speaker

- Simon Martinelli
  Mail: sm@simas.ch
  Web: www.persistence.ch, www.simas.ch
  Blog: simonmartinelli.blogspot.com
- Software Architekt, Entwickler, Berater und Trainer im Java EE Umfeld
- Nebenamtdozent an der Berner Fachhochschule (SWS)
  - Java Persistence API
  - Architektur und Design verteilter Systeme
  - Datenbanken und Data Warehouse Systeme
- IT seit 1995, J2EE/Java EE seit 2000

simas

SBB CFF FFS

Das Problem



simas

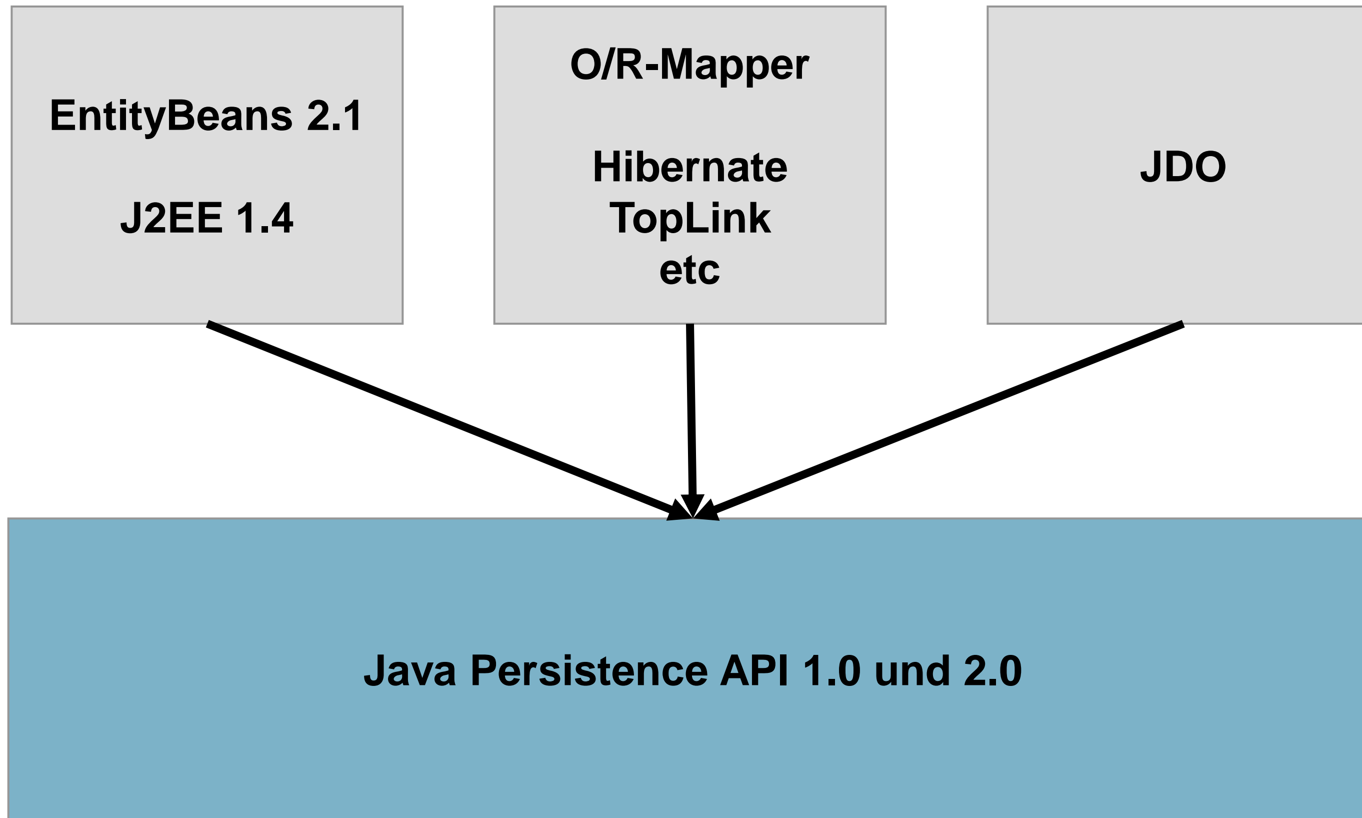
Grundsätze Objekt-Relationales Mapping

- **Objekte nicht Tabellen.** Applikationen arbeiten grundsätzlich nur mit dem Klassenmodell
- **Richtig nutzen, nicht ignorieren.** Um "gutes" O/R-Mapping zu betreiben, muss man sich der Probleme bewusst sein und die relationale Technologie kennen
- **Unauffällig, nicht transparent.** Persistenz ist nicht transparent. Die Applikation muss die Kontrolle über den Objekt-Lifecycle haben
- **Alte Daten, neue Objekte.** Es ist sehr häufig, dass eine Applikation bestehende Daten verwenden muss. Deshalb ist der Support für Legacy Daten zentral.
- **Ausreichend, aber nicht zuviel.** Applikationen dürfen nicht durch ein schwergewichtiges Persistenzmodell erdrückt werden
- **Lokal und mobil.** Entitäten müssen transportiert werden können

Eigenschaften moderner Persistenz Frameworks

- Arbeiten mit gewöhnlichen Java-Klassen für Daten (POJOs)
- Objekte können transient oder persistent sein
- Innerhalb und ausserhalb von Applikationsservern verwendbar
- Vererbung, Aggregation, Komposition abbildbar
- Transitive Persistenz oder Persistence by Reachability
- Lazy Fetching
- Automatic Dirty Checking
- Datenbank-Roundtrips minimieren, Outer Join Fetching
- SQL-Generierung zur Laufzeit

Entstehung



SBB CFF FFS

Technologie Stack

Schicht	Technologie
Session Beans / Java Applikation	Java 5+
Java Persistence API	EJB 3.x
Java Persistence API Implementation	EclipseLink, Hibernate, OpenJPA etc.
JDBC API	JDBC 4.0
JDBC - Driver	Herstellerspezifisch
SQL	SQL >= 2008 oder Dialekte

simas

Entity Metadata

- Kennzeichnung mit **Annotation** **`@Entity`** oder Mapping mit **XML**
- Klasse kann Basisklasse oder abgeleitet sein
- Klasse kann abstrakt oder konkret sein
- Serialisierbarkeit ist bezüglich Persistenz nicht erforderlich
- Anforderungen:
  1. Standardkonstrukutor muss vorhanden sein.
  2. Klasse darf nicht **`final`**, kein Interface und keine Enumeration sein und keine **`final`**-Methoden enthalten
  3. Felder müssen **`private`** oder **`protected`** sein.
  4. Zugriff von *Clients* auf Felder nur über get/set- oder Business-Methoden erlaubt.
  5. Jede Entity muss einen Primärschlüssel (**`@Id`**) haben

- **Configuration by Exception / Conventions over Configuration**

Entity, Beispiel

```
@Entity
public class Employee {

  @Id
  protected Integer id;
  protected String name;
  protected BigDecimal salary;

  public Employee() {
  }
  ...
}
```

simas

SBB CFF FFS

Persitence Unit

- Eine Persistence Unit ist eine *logische* Einheit von Entities. Sie wird beschrieben durch:
  - Einen Namen
  - Die zu dieser Unit gehörenden Entity-Klassen
  - Angaben zum Persistence Provider
  - Angabe zum Transaktionstyp
  - Angaben zur Datenquelle
  - Weitere Properties
  - Namen von XML O/R-Mapping Files
- Technisch wird die Beschreibung einer Persistence Unit in der Datei `META-INF/persistence.xml` abgelegt.
- Persitence Archive = JAR

simas

Persistence Unit, Beispiel

META-INF/persistence.xml

```
<persistence>
  <persistence-unit name="emp"
     transaction-type="RESOURCE_LOCAL">
  <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>examples.model.Employee</class>
    <properties>
      <property name="toplink.jdbc.driver"
        value="oracle.jdbc.driver.OracleDriver" />
      <property name="toplink.jdbc.url"
        value="jdbc:oracle:thin:@localhost:1521:xe" />
      <property name="toplink.jdbc.user" value="emp3" />
      <property name="toplink.jdbc.password" value="emp3" />
    </properties>
  </persistence-unit>
</persistence>
```

simas

SBB CFF FFS

JSR 317: Java Persistence API, Version 2.0

- Specification Lead:
  Linda DeMichiel, Sun Microsystems, Inc.

- Expert Group:
  Adobe Systems Inc., akquinet tech@spree , BEA Systems, Adam Bien, DataDirect Technologies, Ericsson AB, Antonio Goncalves, IBM, Chris Maki, Oracle, OW2, Pramati Technologies, Red Hat, SAP AG, SpringSource, Sun Microsystems Inc., Sybase, Tmax Soft Inc.

- Referenzimplemention
  EclipseLink (ehemals Oracle TopLink), www.eclipse.org/eclipselink

simas

SBB CFF FFS

Fokus JPA 2.0

- Properties standardisieren
- Lücken im O/R-Mapping füllen
- Flexibilität erhöhen
- 2nd Level Cache standardisieren
- Erweitere Locking Möglichkeiten
- Neue Hooks für Herstellererweiterungen
- Bessere API für Toolsupport
- JPQL Erweiterungen
- Java API basierte Abfragesprache
- Integration JSR-303 Bean Validation

simas

Standardisierte Properties

- Bis anhin waren Properties herstellerspezifisch
- Standardisierung von oft gebrauchen Properties im persistence.xml:
  - **`javax.persistence.jdbc.driver`**
  - **`javax.persistence.jdbc.url`**
  - **`javax.persistence.jdbc.user`**
  - **`javax.persistence.jdbc.password`**

Unidirektionals @OneToMany Mapping

- JPA 1.0 konnte unidirektionale 1-n Beziehungen nur mittels Beziehungstabelle abbilden (= ManyToMany)

- In JPA 2.0 kann der Target Foreign Key mit @JoinColumn angegeben werden

```
@Entity
public class Employee {
    ...
    @OneToMany @JoinColumn(name="employee_id")
    private List<Phone> phones;
    ...
}
```

simas

Collections von Nicht-Entities und Embeddables

```
@Entity
public class Employee {
   ...
   @ElementCollection
   @Column(name="SERVICE_DATE")
   private List<Date> serviceDates;
   ...
}
```

Persistente Sortierung

➔ Die Reihenfolge einer List kann durch JPA verwaltet werden

```
@Entity
public class Employee {
    ...
    @OneToMany
    @OrderColumn(name="PHONE_POS")
    List<Phone> phones;
    ...
}
```

SBB CFF FFS

Erweiterte Unterstützung von Maps

➔ Objekte, Embedabbles und Entities als Map Key und Value

```
@Entity
public class Vehicle {
    ...
    @OneToMany
    @MapKeyJoinColumn(name="PART_ID")
    Map<Part,Supplier> suppliers;
    ...
}
```

simas

Erweiterte Embeddables

➔ Embeddables können verschachtelt werden und können Beziehungen haben

```
@Embeddable
public class Assembly {
    ...
    @Embedded
    ShippingDetail shipDetails;

    @ManyToOne
    Supplier supplier;
    ...
}
```

Access Typ Options

- ➔ Access Typ kann in der Hierarchie gemischt werden
- ➔ Verschiedene Access Typen innerhalb einer Klasse sind möglich

```
@Entity @Access(FIELD)
public class Vehicle {
   ...
   @Transient double fuelEfficiency; // Stored in metric

   @Access(PROPERTY)
   protected double getDbFuelEfficiency() {
       return convertToImperial(fuelEfficiency);
   }
   ...
}
```

Zusammengesetzte Primärschlüssel in Beziehungen

```
@Entity @IdClass(PartPK.class)
public class Part {
   @Id int partNo;
   @Id @ManyToOne
   Supplier supplier;
}

public class PartPK {
   int partNo;
   int supplier;
}
```

2nd Level Cache

- Cache für alle EntityManager innerhalb einer Persistence Unit
- Zugriff über EntityManagerFactory
- Momentan nur Basic Funktionen
- Kann durch Hersteller erweitert werden

```
public interface Cache {
   boolean contains(Class cls, Object primaryKey);
   void evict(Class cls, Object primaryKey);
   void evict(Class cls);
   void evictAll();
}
```

Erweitertes Locking

- In JPA 1.0 gab es nur Unterstützung für Optimistic Locking über ein Versionsfeld → @Version
- JPA 2.0 führt neue Locking-Strategien ein:
  - OPTIMISTIC (= READ)
  - OPTIMISTIC_FORCE_INCREMENT (= WRITE)
  - PESSIMISTIC_READ → Repeatable Read
  - PESSIMISTIC_WRITE → Serialized
  - PESSIMISTIC_FORCE_INCREMENT -> Kombination mit Versionsfeld
- Optimistic Locking auch im Pessimistic Locking unterstützt

Erweitertes Locking

- Locking kann über verschiedene Wege verwendet werden.
- Hier ein Beispiel mit EntityManager.refersh()

```
public void applyCharges() {
   Account acct = em.find(Account.class, acctId);
   // calculate charges, etc.
   int charge = … ;
   if (charge > 0) {
       em.refresh(acct, PESSIMISTIC_WRITE);
       double balance = acct.getBalance();
       acct.setBalance(balance - charge);
   }
}
```

simas

API Erweiterungen

- LockMode Parameter bei find, refresh
- Properties Parameter bei find, refresh, lock
- Weitere nützliche Erweiterungen im EnttityManager
  - void detach(Object entity)
  - EntityMangerFactory getEntityManagerFactory()

API Erweiterungen

- Tools brauchen Zugriff auf die Konfiguration und die Metadaten
- Erweiterter EntityManager:
  - `Set<String> getSupportedProperties()`
  - `Map getProperties()`
  - `LockModeType getLockMode(Object entity)`
- Erweiteres Query Interfaces:
  - `int getFirstResult()`
  - `int getMaxResults`
  - `Map getHints()`
  - `Set<String> getSupportedHints()`
  - `FlushModeType getFlushMode()`
  - `Map getNamedParameters()`

SBB CFF FFS

JPQL

- Timestamp

```
SELECT t from BankTransaction t
WHERE t.txTime > {ts '2008-06-01 10:00:01.0'}
```

- Non-polymorphic Queries

```
SELECT e FROM Employee e
WHERE CLASS(e) = FullTimeEmployee OR e.wage = "SALARY"
```

- Collection Paramters in IN Expression

```
SELECT emp FROM Employee emp
WHERE emp.project.id IN [:projectIds]
```

simas

JPQL

- Ordered List Index

```
SELECT t FROM CreditCard c JOIN c.transactionHistory t
WHERE INDEX(t) BETWEEN 0 AND 9
```

- CASE Statement

```
UPDATE Employee e SET e.salary =
CASE
    e.rating WHEN 1 THEN e.salary * 1.1
WHEN
    2 THEN e.salary * 1.05
ELSE
    e.salary * 1.01
END
```

Criteria API

- **QueryBuilder**
  - Factory zum Erstellen von CriteriaQuery Objekten
  - Definiert viele Utility Methoden: Vergleich, Erstellung von Literals, Collection Operationen, Subqueries, Boolean-, String-, Numeric-Funktionen etc.
- **CriteriaQuery**
  - Enthält das Query
  - Zentraler Bestandteil der Query API
  - Enthält ein oder mehrere „Query Roots“ welche den Domain Typ representieren
  - Hat Funktionen um das die Selektionskriterien, Sortierung usw. zu definieren

Beispiel Modell

```
@Entity public class Customer {
    @Id int custId;
    String name;
    @OneToMany(mappedBy="customer")
    Set<Order> orders;
}

@Entity public class Order {
    @Id int orderId;
    @ManyToOne
    Customer customer;
    @OneToMany(mappedBy="order")
    Set<LineItem> items;
}

@Entity public class LineItem {
    @Id int id;
    @ManyToOne
    Order order;
    @ManyToOne
    Product product;
}

@Entity public class Product {
    @Id int productId;
    String name;
    String productType;
}
```

Criteria API

- Einfaches Query

```
QueryBuilder qb = em.getQueryBuilder();
CriteriaQuery q = qb.create();
Root<Customer> customer = q.from(Customer.class);
q.select(customer);
```

- entspricht

```
SELECT c FROM Customer c;
```

Criteria API

- Parameter

```
QueryBuilder qb = em.getQueryBuilder();
CriteriaQuery q = qb.create();
Root<Customer> c = q.from(Customer.class);
Parameter<Integer> param = qb.parameter(Integer.class);
q.select(c).where(
  qb.equal(c.get("status"), param));
```

- entspricht

```
SELECT c FROM Customer c WHERE c.status = :stat
```

Metamodel API

➔ Generiertes Metamodell dient der Typsicherheit

```
@TypesafeMetamodel
public class Customer_ {
   public static volatile Attribute<Customer, Integer> custId;
   public static volatile Attribute<Customer, String> name;
   public static volatile Set<Customer, Order> orders;
}
@TypesafeMetamodel
public class Order_ {
   public static volatile Attribute<Order, Integer> orderId;
   public static volatile Attribute<Order, Customer> customer;
   public static volatile Set<Order, LineItem> items;
}
```

Metamodel API

- Fortsetzung

```
@TypesafeMetamodel
public class LineItem_ {
   public static volatile Attribute<LineItem, Integer> id;
   public static volatile Attribute<LineItem, Order> order;
   public static volatile Attribute<LineItem, Product> product;
}
@TypesafeMetamodel
public class Product_ {
   public static volatile Attribute<Product, Integer> productId;
   public static volatile Attribute<Product, String> name;
   public static volatile Attribute<Product, String> productType;
}
```

Criteria API + Metamodel API = Typsicherheit

- Neue objektorientierte, typsichere Möglichkeit um Abfragen zu erstellen

```
QueryBuilder qb = em.getQueryBuilder();
CriteriaQuery q = qb.create();
Root customer = q.from(Customer.class);
Join item =
    customer.join(Customer_.orders).join(Order_.items);
q.where(
  qb.equals(item.get(Item_.product)
 .get(Product_.productType), "printer"))
 .select(customer.get(Customer_.name));
```

- entspricht

```
SELECT c.name FROM Customer c JOIN c.orders o JOIN o.items i
WHERE i.product.productType = 'printer'
```

SBB CFF FFS

Integration mit JSR 303 Bean Validation

- Validierung und gewisse Elemente der DDL Generierung werden mit JSR 303 ersetzt
  - `@NotNull` statt `@Column(nullable=false)`
  - `@Size.max` statt `@Column.length`
  - `@Digits` statt `@Column.precision/.scale`
  - `@Min / @Max` bei numerischen Columns
  - `@Future / @Past` bei Datumstypen
  - `@Size` für Collections und Arrays

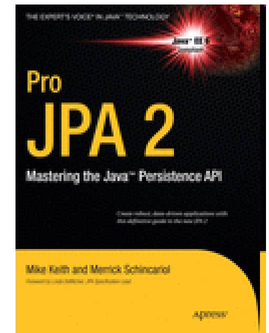
simas

Zusammenfassung

- JPA 2.0 führt viele Erweiterungen ein, welche gefehlt und oft nachgefragt wurden
- JPA ist zu 90 - 95% „fertig“
- JPA wird nie alle Features von Hibernate, EclipseLink etc. abdecken
- Aber es gibt immer weniger Gründe die proprietäre API des O/R-Mappers zu verwenden
- Hinweis:
  Nur weil es ein neues Feature gibt, muss man das noch lange nicht brauchen!

Literatur und Links

- Pro JPA 2: Mastering the Java Persistence API
  Mike Keith, Merrick Schincariol
  ISBN-13: 978-1-4302-1956-9



- Exercises zu Pro JPA 2
  Mike Keith, Simon Martinelli
  COMING SOON ☺

- JSR 317: Java Persistence 2.0
  www.jcp.org/en/jsr/detail?id=317

- EclipseLink (Referenzimplementation)
  www.eclipse.org/eclipselink