Multitenancy und JPA
— Eine Bestandsaufnahme —

Bernd Müller

Fakultät Informatik
Ostfalia
Hochschule Braunschweig/Wolfenbüttel

DOAG 2011

Vorstellung Referent

- Studium Informatik, Uni Stuttgart
- Promotion in Informatik, Uni Oldenburg
- IBM, HIS
- Professor für Wirtschaftsinformatik, HS Harz
- Professor für Software-Technik, Fakultät Informatik, Ostfalia
- Buchautor (JSF, JPA, Seam, ...)
- Mitglied JUG Ostfalen
- GF PMST GmbH
- ...

JAVA PERSISTENCE API 2

bernd MÜLLER
harald WEHR

HIBERNATE, ECLIPSE LINK,
OPENJPA UND ERWEITERUNGEN

EXTRA: Mit kostenlosem E-Book

HANSER

erscheint
Frühjar 2012

stay tuned

Motivation und Standortbestimmung

JSR 342, Java EE 7

The main theme for this release is the Cloud. The Java EE platform is already well suited for cloud environments thanks to its container-based model and the abstraction of resource access it entails. In this release we aim to further enhance the suitability of the Java EE platform for cloud environments. As a result, Java EE 7 products will be able to more easily operate on private or public clouds and deliver their functionality as a service with support for features such as multitenancy and elasticity (horizontal scaling). Applications written for Java EE 7 will be better able to take advantage of the benefits of a cloud environment.

JSR 338, JPA 2.1

- Support for the use of custom types and transformation methods in object/relational mapping.
- Support for the specification of immutable attributes and readonly entities.
- Support for user-configurable naming strategies for use in O/R mapping and metamodel generation.
- More flexibility in the use of generated values; support for UUID generator type.
- Additional mapping metadata to provide better standardization for schema generation.
- Support for multitenancy.
- Additional event listeners and callback methods; availability of entity manager to callbacks.
- Methods for dirty detection.
- Improved ability to control persistence context synchronization.
- ...

Was ist Multitenancy ?

- Tenant = Mandant, also mehrere Mandanten, Mandantenfähigkeit
- Kann immer programmatisch realisiert werden
- Ziel ist jedoch: möglichst wenig Zusatzaufwand, möglichst deklarativ
- In Java-EE potentiell betroffen:
  - Oberfläche (JSF)
  - Anwendungslogik (Session-Beans)
  - Persistenz (JPA)

Mandantenfähigkeit und JPA — Alternativen

- Getrennte/separate Datenbanken
- Getrennte/separate Schemata
- Gemeinsames Schema

Java Community Process

- Mittlerweile sehr öffentlich (Mailing-Listen, ...)
- Mitarbeit auch für Individuen möglich
- Lässt sich durch existierende Systeme motivieren
- Autor nicht Mitglied in JSR 338 EG, Infos durch eigene Recherchen
- Mittlerweile 3. Draft JPA 2.1 verfügbar

Die Provider

- EclipseLink (RI), verwendet in GlassFish
- OpenJPA, verwendet in Geronimo, WebSphere
- Hibernate, verwendet in JBoss-AS

Multitenancy in EclipseLink

Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: REQUIRED | OPTIONAL DETAIL: ELEMENT

EclipseLink 2.3.0, build 'v20110604-r9504' API Reference

org.eclipse.persistence.annotations

Annotation Type Multitenant

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
public @interface Multitenant
```

Multitenant specifies that a given entity is shared amongst multiple tenants of a given application. The multitenant type specifies how the data for these entities are to be stored on the database for each tenant. Multitenant can be specified at the Entity or MappedSuperclass level.

See Also:
MultitenantType, TenantDiscriminatorColumn, TenantDiscriminatorColumns

Author:
Guy Pelletier

Since:
EclipseLink 2.3

Optional Element Summary	
MultitenantType	**value** (Optional) Specify the multi-tenant strategy to use.

Overview Package **Class** Tree Deprecated Index Help

EclipseLink 2.3.0, build 'v20110604-r9504' API Reference

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: REQUIRED | OPTIONAL DETAIL: ELEMENT

org.eclipse.persistence.annotations

Annotation Type TenantDiscriminatorColumn

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
public @interface TenantDiscriminatorColumn
```

Tenant discriminator column(s) are used with a SINGLE_TABLE multitenant strategy. Tenant discriminator column(s) are completely user specified and there is no limit on how many tenant discriminator columns an application can define (using the TenantDiscriminatorColumns annotation) Tenant discriminator column(s) can be specified at the Entity or MappedSuperclass level and must always be accompanied with a Multitenant(SINGLE_TABLE) specification. It is not sufficient to specify only tenant discriminator column(s).

See Also:
TenantDiscriminatorColumns, Multitenant, MultitenantType

Author:
Guy Pelletier

Since:
EclipseLink 2.3

Optional Element Summary

java.lang.String	**columnDefinition** (Optional) The SQL fragment that is used when generating the DDL for the discriminator column.
java.lang.String	**contextProperty** (Optional) The name of the context property to apply to the tenant discriminator column.
DiscriminatorType	**discriminatorType** (Optional) The type of object/column to use as a class discriminator.
int	**length** (Optional) The column length for String-based discriminator types.
java.lang.String	**name** (Optional) The name of column to be used for the tenant discriminator.
boolean	**primaryKey** Specifies that the tenant discriminator column is part of the primary key of the tables.
java.lang.String	**table** (Optional) The name of the table that contains the column.

Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: REQUIRED | OPTIONAL DETAIL: ELEMENT

EclipseLink 2.3.0, build 'v20110604-r9504' API Reference

org.eclipse.persistence.annotations

Annotation Type TenantDiscriminatorColumns

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
public @interface TenantDiscriminatorColumns
```

A TenantDiscriminatorColumns annotation allows the definition of multiple TenantDiscriminatorColumn.

See Also:
TenantDiscriminatorColumn

Author:
Guy Pelletier

Since:
EclipseLink 2.3

Beispiel: Ein Kunde kann mehrere Konten haben ...

```
@Entity
@Multitenant
public class Kunde {

  @Id @GeneratedValue
  private Integer id;
  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  @OneToMany(mappedBy = "kunde",
             cascade = CascadeType.ALL)
  private List<Konto> konten;
  ...
```

Das Konto

```
@Entity
@Multitenant
public class Konto {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
                  generator = "KontoSeq")
  @SequenceGenerator(name = "KontoSeq",
      sequenceName="KontoSeq", allocationSize = 5,
      initialValue = 1000000)
  private Integer kontonummer;
  private BigDecimal kontostand;
  @ManyToOne
  @JoinColumn(name = "kunde")
  private Kunde kunde;
  ...
```

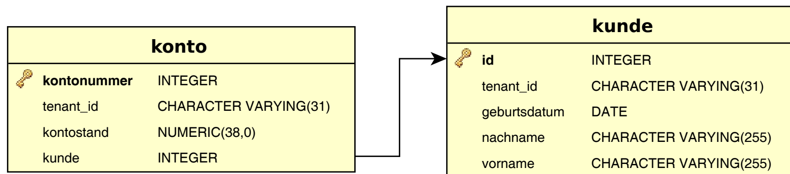
Verwendung: persistence.xml oder programmatisch

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence ...>
    ...
    <properties>
      <property name="eclipselink.tenant-id"
                value="007"/>

Map<String, String> props =
    new HashMap<String, String>();
props.put("eclipselink.tenant-id", "Mandant-1");
EntityManagerFactory emf = Persistence
    .createEntityManagerFactory("bank", props);
EntityManager em = emf.createEntityManager();
```

Schema

- Kann erzeugt werden:
  `<property name="eclipselink.ddl-generation" ...`



Daten

```
bank=> select * from kunde;
 id | tenant_id | geburtsdatum | nachname |   vorname
----+-----------+--------------+----------+-------------
  2 | Mandant-1 | 1965-09-27   | Atze     | Schröder
  3 | Mandant-1 | 1966-04-03   | Michael  | Mittermeier
  1 | Mandant-1 | 1972-11-01   | Barth    | Mario
 52 | Mandant-2 | 1965-12-21   | Anke     | Engelke
 51 | Mandant-2 | 1976-01-04   | Bülent   | Ceylan
 53 | Mandant-2 | 1960-10-29   | Dieter   | Nuhr
(6 Zeilen)
```

Verwendung: Lesen

- Transparente Verwendung

```
...
props.put("eclipselink.tenant-id", "Mandant-1");
...

List<Kunde> list = em
    .createQuery("Select k from Kunde k",
                 Kunde.class)
    .getResultList();
...
Kunde kunde = em.find(Kunde.class, <pk>));
...
```

Alternativen

```
@Entity
@Multitenant
@Multitenant(MultitenantType.SINGLE_TABLE)
// noch nicht unterstuetzt:
@Multitenant(MultitenantType.TABLE_PER_TENANT)
// nur in Verbindung mit @Multitenant
@TenantDiscriminatorColumn(name = "mandant")
@TenantDiscriminatorColumn(name = "mandant",
                           primaryKey = true)
// Tenant-Id muss konvertierbarer String sein:
@TenantDiscriminatorColumn(name = "mandant",
  discriminatorType = DiscriminatorType.INTEGER)
public class Kunde {
  ...
```

Gemappter Tenant

- Muss read-only sein

```
@Entity
@Multitenant
@TenantDiscriminatorColumn(name = "mandant")
public class KundeMapped {

  @Id @GeneratedValue
  private Integer id;
  ...
  // Die gemappte Diskriminatorspalte:
  @Column(insertable = false, updatable = false)
  private String mandant;
  ...
```

Verwendung in Java-EE

```
<persistence ...>

  <persistence-unit name="tenant-1" transaction-type="JTA">
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <property name="eclipselink.tenant-id"
                value="Mandant-1"/>
      ...
    </properties>
  </persistence-unit>

  <persistence-unit name="tenant-2" transaction-type="JTA">
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <property name="eclipselink.tenant-id"
                value="Mandant-2"/>
      ...
    </properties>
  </persistence-unit>

</persistence>
```

Verwendung in Java EE (cont'd)

```
@Singleton
@Startup
public class KundeService {

  @PersistenceContext(unitName = "tenant-1")
  EntityManager em1;

  @PersistenceContext(unitName = "tenant-2")
  EntityManager em2;

  @PostConstruct
  public void init(){
    em1.persist(new Kunde("Barth", "Mario", "1.11.1972"));
    em1.persist(new Kunde("Michael", "Mittermeier", "3.4.19
    em2.persist(new Kunde("Anke", "Engelke", "21.12.1965"))
    em2.persist(new Kunde("Dieter", "Nuhr", "29.10.1960"));
  }
}
```

Alternative: Qualifier mit CDI

```
public class DatabaseProducer {

  @Produces
  @PersistenceContext(unitName = "tenant-1")
  @Tenant1PC
  private EntityManager em1;

  @Produces
  @PersistenceContext(unitName = "tenant-2")
  @Tenant2PC
  private EntityManager em2;

}
```

Alternative: Qualifier mit CDI (cont'd)

```
@Singleton
@Startup
public class KundeService2 {

  @Inject
  @Tenant1PC
  EntityManager em1;

  @Inject
  @Tenant2PC
  EntityManager em2;

  @PostConstruct
  public void init(){
    em1.persist(new Kunde("Barth", "Mario", "1.11.1972"));
    em1.persist(new Kunde("Michael", "Mittermeier", "3.4.19
    em2.persist(new Kunde("Anke", "Engelke", "21.12.1965"))
    em2.persist(new Kunde("Dieter", "Nuhr", "29.10.1960"));
  }
}
```

Multitenancy in OpenJPA

OpenJPA verfolgt völlig anderen Ansatz

- Slice: „*Slice is a module for distributed persistence in OpenJPA. Slice enables an application developed for a single database to adapt to a distributed, horizontally partitioned, possibly heterogeneous, database environment. This all occurs without any change in the original application code or the database schema. See how to leverage this flexibility for your own applications, especially those destined for the cloud or Software as a Service.*“
- Features
  - Transparency
  - Scaling
  - Distributed Query
  - Data Distribution
  - Data Replication
  - Heterogeneous Database
  - Distributed Transaction
  - Collocation Constraint

Ansatz

- Anwendung unverändert
- Separate Datenbanken
- Flexible Verteilung bzw. Redundanz durch Callback-Methoden

Konfiguration in persistence.xml

```
<property name="openjpa.BrokerFactory" value="slice" />
<property name="openjpa.slice.Names"
          value="mandant1,mandant2,mandant3" />
<property name="openjpa.slice.Master" value="mandant1" />
<property name="openjpa.slice.Lenient" value="true" />

<property name="openjpa.ConnectionDriverName"
          value="org.postgresql.Driver"/>
<property name="openjpa.slice.mandant1.ConnectionURL"
          value="jdbc:postgresql://localhost/bank1"/>
<property name="openjpa.slice.mandant2.ConnectionURL"
          value="jdbc:postgresql://localhost/bank2"/>

<property name="openjpa.slice.mandant3.ConnectionDriverName
          value="com.mysql.jdbc.Driver" />
<property name="openjpa.slice.mandant3.ConnectionURL"
          value="jdbc:mysql://localhost/bank3" />
```

Weitere Konfiguration

```
<property name="openjpa.slice.DistributionPolicy"
          value="de.jpainfo.DistributionPolicyMandant
<property name="openjpa.slice.QueryTargetPolicy"
          value="de.jpainfo.QueryPolicyMandant"/>
<property name="openjpa.slice.FinderTargetPolicy"
          value="de.jpainfo.FinderPolicyMandant"/>
```

Interface DistributionPolicy

Method Detail

distribute

```
String distribute(Object pc,
                  List<String> slices,
                  Object context)
```

Gets the name of the target slice where the given newly persistent or the detached, to-be-merged instance will be stored.
If the current state of the given instance is sufficient to determine the target slice, return null. In that case, the runtime will callback this method again before the instance being flushed to the datastore. By then, the policy *must* be able to determine the target slice.

Parameters:

`pc` - The newly persistent or to-be-merged object.
`slices` - list of names of the active slices. The ordering of the list is either explicit `openjpa.slice.Names` property or implicit i.e. alphabetic order of available identifiers if `openjpa.slice.Names` is unspecified.
`context` - the generic persistence context managing the given instance.

Returns:

identifier of the slice. This name must match one of the given slice names.

See Also:

`DistributedConfiguration.getActiveSliceNames()`

Interface QueryTargetPolicy

Method Detail

getTargets

```
String[] getTargets(String query,
                    Map<Object,Object> params,
                    String language,
                    List<String> slices,
                    Object context)
```

Gets the name of the slices where a given query will be executed.

Parameters:

`query` - The query string to be executed.
`params` - the bound parameters of the query
`language` - the query language
`slices` - list of names of the active slices. The ordering of the list is either explicit `openjpa.slice.Names` property or implicit i.e. alphabetic order of available identifiers if `openjpa.slice.Names` is unspecified.
`context` - generic persistence context managing the given instance.

Returns:

identifier of the slices. This names must match one of the given slice names.

See Also:

`DistributedConfiguration.getActiveSliceNames()`

Interface FinderTargetPolicy

Method Detail

getTargets

```
String[] getTargets(Class<?> cls,
                    Object oid,
                    List<String> slices,
                    Object context)
```

Gets the name of the slices where a given finder will be executed.

Parameters:

`cls` - The class of the finder.
`oid` - the primary key for the finder
`slices` - list of names of the active slices. The ordering of the list is either explicit `openjpa.slice.Names` property or implicit i.e. alphabetic order of available identifiers if `openjpa.slice.Names` is unspecified.
`context` - generic persistence context managing the given instance.

Returns:

identifier of the slices. This names must match one of the given slice names.

See Also:

`DistributedConfiguration.getActiveSliceNames()`

Beispiel: Verteilung nach Entity-Typ

```
public class DistributionPolicyMandant
       implements DistributionPolicy {

  public String distribute(Object entity,
                           List<String> slices,
                           Object context) {
    if (entity instanceof Kunde) {
      return "mandant1";
    } else {
      return "mandant2";
    }
  }
```

Beispiel: Verteilung nach Anwendungsdaten

```
public class DistributionPolicyMandant
             implements DistributionPolicy {

  public String distribute(Object entity,
         List<String> slices, Object context) {
    if (entity instanceof Kunde) {
      char anfangsbuchstabe =
           ((Kunde) entity).getNachname().toCharArray()[0];
      if ('A' <= anfangsbuchstabe
          && anfangsbuchstabe < 'M') {
        return "mandant1";
      } else if ('M' <= anfangsbuchstabe
                 && anfangsbuchstabe < 'T') {
        return "mandant2";
      } else {
        return "mandant3";
      }
    } else {
      return "default";
    }
  }
```

Beispiel: Verteilung mit zusätzlichem Mandanten-Property

```
public class DistributionPolicyMandant
             implements DistributionPolicy {

  public String distribute(Object entity,
                           List<String> slices,
                           Object context) {
    if (entity instanceof Kunde) {
      // 'mandant' Property der Entity-Klasse
      return ((Kunde) entity).getMandant();
    } else {
      return "default";
    }
  }
```

Queries

```
public class QueryPolicyMandant implements
                                         QueryTargetPolicy {
  public String[] getTargets(String query,
         Map<Object, Object> params, String language,
         List<String> slices, Object context) {
    return ...
  }
}
```

Queries (cont'd)

```
List<Kunde> list = em.createQuery(
     "Select k from Kunde k where k.nachname = :nachname",
     Kunde.class)
  .setParameter("nachname", "Mittermeier").getResultList();
```

- Parameter query:

```
"Select k from Kunde k where k.nachname = :nachname"
```

- Parameter param Map mit:

```
nachname = Mittermeier
```

Multitenancy in Hibernate

Leider ...

- Dem Referenten bis vor kurzem leider nichts bekannt (:-(
- Hibernate Shards (horizontal partitioning)

Zusammenfassung

- Im 3. Draft JSR 338 noch keine Angaben zur Mandantenfähigkeit
- EclipseLink 2.3 (Indigo) enthält `@Multitenant` und weitere
  - Noch nicht vollständig
- OpenJPA mit Slice Möglichkeit zur Verteilung auf mehrere Datenbanken
  - Kann auch für Mandantenfähigkeit verwendet werden
- In letzter Minute: EclipseLinks Composite Persistence Units
  - Artikel in Java aktuell 1/2012
- Ebenfalls: Hibernate Shards
- Es bleibt spannend . . .

Fragen und Anmerkungen