Managing Persistent Object in JPA

JPA Slides #3

Transactions

Developing a persistence capable applications is very much about

1. Mapping, done in previous slides
2. Transaction handling, upcoming...

- Very complicated subject, can't cover
- Will use simplest possible approach ...
-... and hopefully get some help from environment
- On transaction exceptions we normally get a **rollback** i.e. transaction restore to state before transaction begin

Managing Entities

The Entities do not persist/remove themselves when created/garbage-collected... (this design has been tested and rejected, why..?)

... instead we use an **EntityManager (EM)**...

(Practical use of EntityManager not as complicated as the following explanation... sorry)

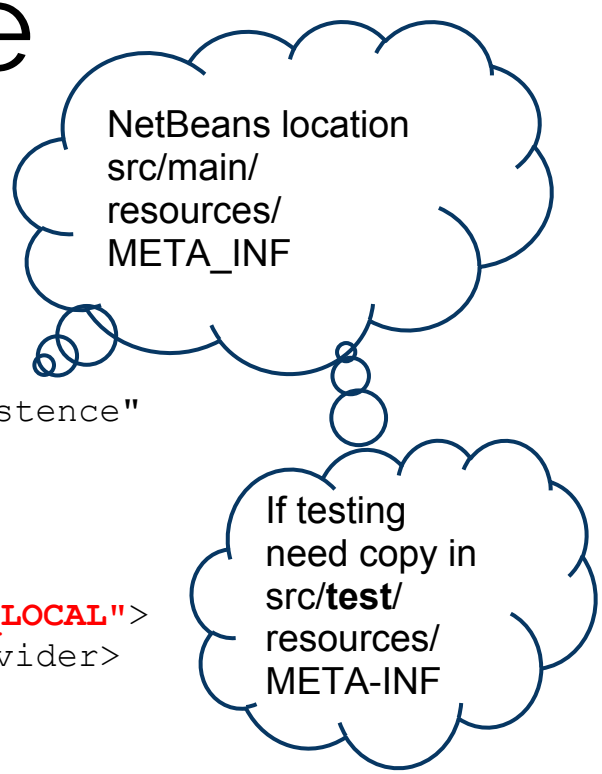
Persistence Unit

Database configuration defined in persistence.xml
- PU's have a **names** (string to use in code)
- Contains database info
- Lists which entities to "manage", if not listed exception, **"not a known entity type" watch out!!**
- Transactional type for EM, upcoming...
- Table generating strategies (as in previous slide series)
- ...

All types (classes) in a PU must be collocated in same database

Possible many PU's (many databases).

Persistence Unit Sample




```
// Generated by NetBeans, puhiii...
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

  <persistence-unit name="managing_pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>jpa.core.Customer</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
     <propertyname="javax.persistence.jdbc. url"value="jdbc:derby://localhost:1527/WebShop"/>
      <property name="javax.persistence.jdbc. password" value="app"/>
      <property name="javax.persistence.jdbc. driver"
                                          value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc. user" value="app"/>
      <property name="eclipselink.ddl-generation" value=" drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Persistence Context

"A set of managed entity instances in which for any **persistent entity identity** [primary key] there is a **unique entity instance**. Within the persistence context, the entity instances and their life cycle are managed by the **EntityManager**"

I.e a table row (unique by primary key) will be represented as a unique in memory object in PC (identity shall use ==)

- Every PC is associated with a PU
- Every EntityManager is associated with a PC

Entity Managers

JPA has three different types, ... huuhhh..

Container Managed Entity Managers (container manages life cycle of EM)
- Transaction-Scoped: PC managed by EM follows the transaction, transaction committed EM gone (Use: **stateless** environment)
- Extended: Single EM bound to life cycle of some **stateful** managed object
- Both use **JTA** transaction (Java EE server transactions)
- ...more to come...

Application Managed Entity Managers
- Application manages EM life cycle
- PC managed by EM follows life cycle of EM
- Simplified: Uses **RESOURCE_LOCAL** transactions (native transactions of JDBC driver)
- Application still possible runs in container
- If running in JSE must use this kind of EM (i.e. Junit test)

Application Managed Entity Manager

When running Java SE (JUnit) we use this to get an EM.

```
public class SomeClass {
     EntityManagerFactory emf =
         Persistence.createEntityManagerFactory("...puName...");

    public void someMethod( ... ){
       EntityManager em = emf.
                    createEntityManager();  // Get EM, PC Created

        // Application must handle transactions
         em.getTransaction().begin();
        // Any write operation here...  (reads, no transaction)
        em.getTransaction().commit(); //Make writes durable
        em.isOpen()  // PC alive?? Yes
        em.close();  // PC gone, and em when method finished
    }
}
In persistence.xml:   transaction-type="RESOURCE_LOCAL"
```

Application Managed EM Exceptions

Need to signal for rollback (em = EntityManager)

```
EntityTransaction tx = em.getTransaction();
try {
    tx.begin()
     ...
    tx.commit()
} catch( ... ){
    if( tx.isActive(){
        tx.rollback()
}finally{
    em.close();
}
```

More later ...

Container Managed Entity Managers

More to come … need Enterprise Java Beans (EJB's)

EntityManager API

javax.persistence.EntityManager

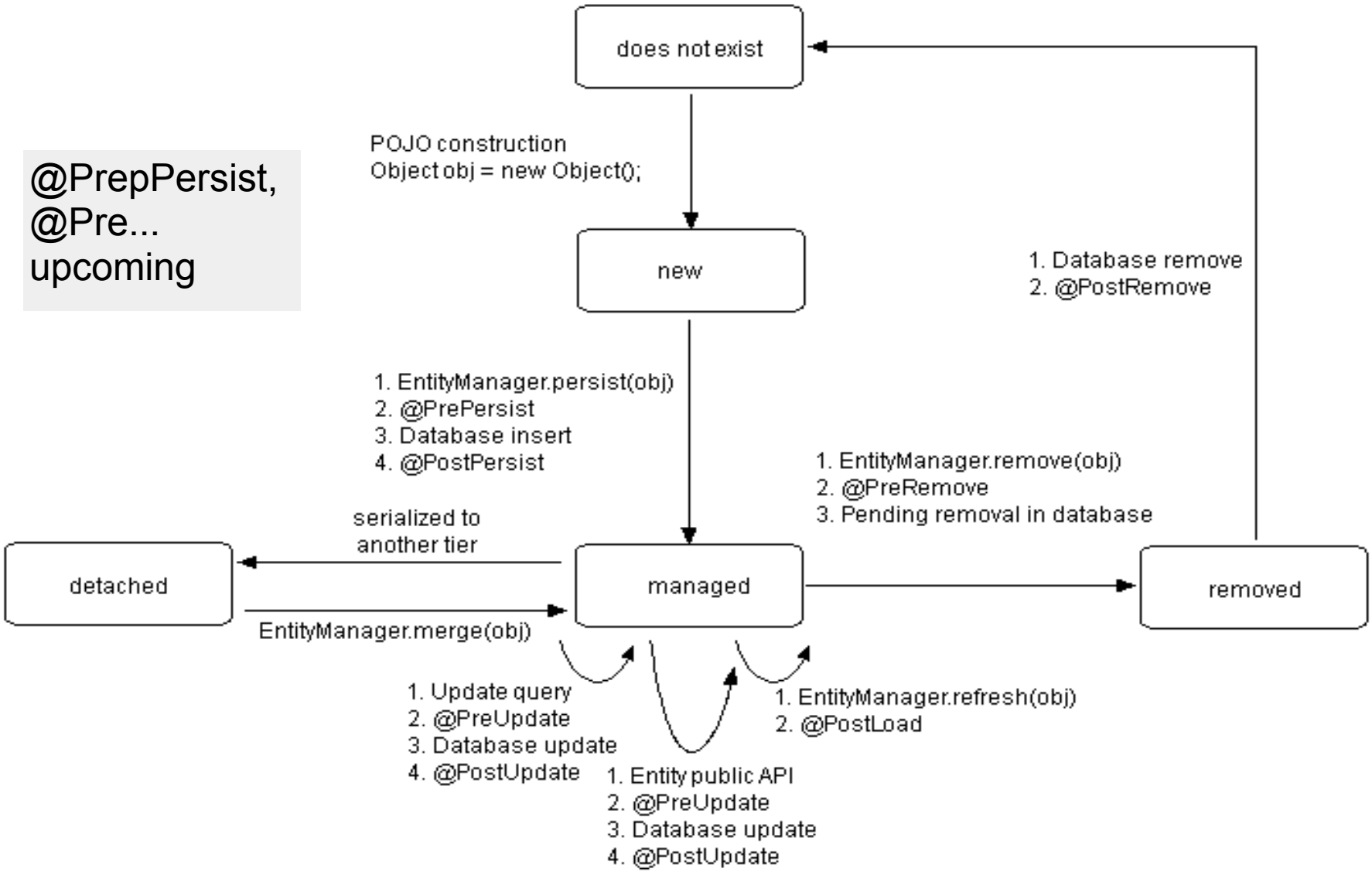
The CRUD operations
- em.persist(object) // Note: After this call object has id
- em.find(... primaryKey)
- em.merge(object) // update
- em.remove(object)
- ...

Entity Instance Life Cycle

Entity Instance (@Entity) may be
- **New**, no persistent identity
- **Managed**, has identity in PC, will be synchronized with real database
- **Detached** (not managed), has identity but not in PC will **not** be synchronized with database
- **Removed**, has identity in PC but will be removed from real database

Entity Instance Life Cycle cont.

@PrepPersist,
@Pre...
upcoming



Create, Find and Delete Application Managed (handle transaction in code)

```
// Create (tx is transaction object)
Product p = new Product();
tx.begin(); // using RESOURCE_LOCAL have to begin/commit
em.persist(p);// p managed (will have id after call)
tx.commit();  // After this p detached

// Read
// p managed if inside transaction else detached
p = em.find(Product.class, 123L);

// Delete
// No data, just the reference
p = em.getReference(Product.class, id);
tx.begin();
em.remove(p);
tx.commit(); // p state as when remove was called
```

Refresh and Detach

Refresh

```
// State refreshed from database
em.refresh(p); // p managed (before and after)
```

Entities detached at

- tx.commit(), if transaction scoped container managed EM
- transaction rollback (exception)
- explicitly detaching the entity

```
em.flush()  // Must do before...
em.detach(p) // ...this
```

- Clearing the PC: em.clear()
- Closing the EM: em.close()
- Passing by value (serializing)

Merging and Checking

Merge will create new managed copy or state copied

```
// p detached
p1 = em.merge(p)     // p1 != p !!! New object

p = em.merge(p)      // Could look like this, tricky...
```

Possible to check if managed

```
    /* True if
     *  p retrieved with em.find(), em.getRefrence()
     *  em.persist(p) called or
     *  persist cascaded
     */
    em.contains(p)
```

Surviving...?!?

What's happening when...
- Persist an already managed instance?
- Merge a removed instance?
- Detach a new instance?
- Refresh a detached instance?
- ...

JPA typical behavior
- If possible harm: **Exception**
- If harmless: **Nothing happens**
- Others: Have to find out ...

Gotcha's

Must keep track if detached or not

Bad

```
company.setPerson(detachedPerson);
em.persist(company);  //Will probably not work
```

Correct

```
// Get managed copy of detached person
managedPerson = em.merge(detachedPerson);
company.setPerson(managedPerson)
em.persist(company)  // Ok
```

More Gotcha's

LazyInitializationException

- Combination: Lazy fetching/detached object
- Object is detached, trying to access non fetched parts, Exception!

Efficiency and Concurrency

EntityManagerFactory is a costly object

- Might maintain a meta data cache, object state cache, EntityManager pool, connection pool, ...
- Don't open and close repeatedly (keep open)
- Thread safe

EntityManager is lightweight

- Get one (open), use, ..., close (as in previous slide)
- Not thread safe

Entity Life Cycle Callbacks

Can annotate methods to be called during life object cycle

- @PrePersist, @PostPersist,...
- See previous slide
- Bean validation triggered just before @PrePersist, @PreUpdate
- Method (logic) only related to the annotated entity (single class)
- Signature for callback method; void anyName()
- Possible customization: Separate listener class

Cascade

Should storing, deleting, etc. apply to associated objects?

- If car deleted probably engine should be deleted (also in database)!
- Possible to specify

```
@OneToOne( cascade = { CascadeType.REMOVE })
private Engine e;
```

- Cascade types: ALL, PERSIST, REMOVE, MERGE, REFRESH, CLEAR, ALL
- If no cascade: Have to persist each single objects in correct order

Releasing Resources

Always important to release resources
- Always close (as soon as possible) EM/PC
- Else possible exhaustion of database resources

Pitfalls, Exceptions!!
- Not enough to just close()
- If exception possible not released
-http://javanotepad.blogspot.com/2007/06/how-to-close-jpa-entitymanger-in-web.html

Persistence Layer Testing

If application managed

- A plain old JUnit test (extra dependencies) with database server running
    - Possible to inspect tables
- Use embedded Derby i.e. database runs embedded in application (in JUnit test)
    - No server needed
    - Not possible to inspect tables, have to rely on test outcomes (Assert...)
    - Need special test PU, see code samples. Must be in "Other Test Sources" in NetBeans
    - **Will create invisible tmp-directory** (don't put in Git, possible remove)

Data Access Object (DAO)

Entity classes never accessed directly , always via a data access layer (object)

We have the ProductCatalogue class etc. acting as DAO's
- Implementation …
- Stateless (or immutable), session scoped beans like CDI (or EJB … upcoming)

So far....

... we have used Application Managed EM's ...

..... now for container managed....
- EM API is the same, PU the same (simpler) but

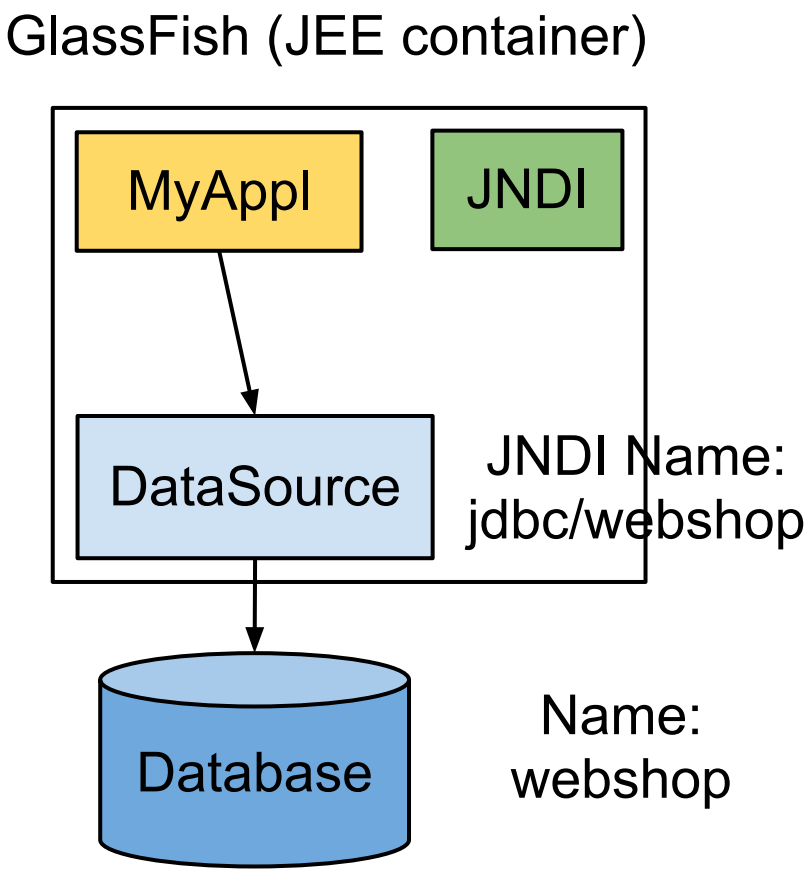
...we need Java Enterprise Beans for this so, ... switch to EJB slides, come back and then continue on next slide...

Container Managed EM Environment

Application in container

DataSource object (refactored database connection, cross application, in glassfish-resources.xml). Created outside application (server admin). Has a JNDI name

JNDI, Java Naming and Directory Interface, key/value-store (imagine: global Tree<String, Object>). String is the name of the object.



Container Managed Entity Manager/Transaction scoped

When running Java EE we use this to get an EM

```
@Stateless   // This is an EJB
public class SomeClass {
    @PersistenceContext(unitName = "nameOfMyPU")
    private EntityManager em;  // EM injected

    // Method starts transaction (if needed) ...
    public void someMethod( ... ){
       //  PC follows transaction (transaction scoped)
       em.persist( object );
    }     // .. and commits. PC gone.
}
```

In persistence.xml: **transaction-type="JTA"**

Container Managed Entity Manager/Extended

You probably don't need this, avoid..

```
@Stateful   // This is an EJB
// PC life cycle tied to bean life
cycle                                    public class SomeClass {
    @PersistenceContext(unitName = "nameOfMyPU",
                                type=PersistenceContextType.EXTENDED)

     private EntityManager em;  // EM injected
     public void someMethod( ... ){
         em.persist( object );      // Same PC as ...
     }
    public void someOtherMethod( ... ){
         em.persist( object );     // .. here
     }
}
```

A transaction can span multiple method calls

Not so Obvious Benefits

...using container managed EM, automatic PC propagation, all involved beans use same PC (in Application Managed have to pass EM around)

... many, many, details...

Summary EM, PC

Container Application see book JPA 2

Persistence Layer Testing

If container managed (i.e. session beans)

- Use embedded EJB container

… and embedded database … (hard)

The MOST IMPORTANT JPA SLIDE

Finding errors

- Is database running (or do we use an embedded?)?
- Is it the correct PU, check location! Is PU correct? All classes listed? JTA/RESOURCE_LOCAL? Validate XML!
- Are the dependencies correct? JUnit need many...
- Is it a database constraint violation? Not null?
- Are the mappings, correct?
- Should the objects be managed? Are they..?
- Should we use the same PC? Do we...?
- Should it be a cascade or eager fetch? Do we have a cascade?
- Delete the Files > tmp dir. (when using embedded Derby), no version handling for tmp dir.