

# Java EE Architecture, Part Three

# Content

- Requirements on the Integration layer
- The Database Access Object, DAO Pattern
- Frameworks for the Integration layer

# Requirements on the Integration Layer

- Performance
  - The SQL must make use of stored procedures, prepared statements, indexes and so on.
  - Good knowledge about the database and the DBMS is required.
  - The O/R mapping technology (e.g. JPA) must be configured to use caches, lazy loading etc in an efficient way.
  - DBMS tuning is not included in this course.
  - JPA tuning is mentioned *very* briefly below.

# Requirements on the Integration Layer

- Concurrency
  - Must behave correct even if the same entity is updated concurrently by different threads.

# Requirements on the Integration Layer

- Low coupling between business and resource
  - The very purpose of the integration layer is to separate business and resource.
  - The definition is adapted to the needs of the business layer and the implementation to the data store.
  - Shall it be possible to change DBMS without updating the business logic?

# Requirements on the Integration Layer

- O/R-mapping
  - Conversion between rows in the database and objects in Java.
  - How are relations, inheritance and other object oriented paradigms mapped to the database?

# Requirements on the Integration Layer

- Primary keys
  - When and by who are they generated?
  - How do we ensure that they are unique?

# Requirements on the Integration Layer (cont)

- What data shall be used for communication with business?
  - Not primitive data since that would lead to very long and complicated method signatures.
  - May the business logic objects themselves be used?



# The Database Access Object, DAO Pattern

- The responsibility of a DAO is to handle database calls.
- It should have no dependencies on the business layer and should contain no business logic.
- Its public interface is designed to meet the needs of the business layer.

# DAO, an example

```
public class ProductDao {

    @PersistenceContext(unitName = "productPU")
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
        Query query = em.createQuery(
            "from Product as p where p.category = :category");
        query.setParameter("category", category);
        return query.getResultList();
    }

    //Other database access methods.

}
```

# Frameworks for the Integration Layer

- Java Persistence API (JPA) (Sun, part of Java EE)
- Hibernate (JBoss)

# A Comparison

- JPA and Hibernate have very similar architecture and functionality.
- Since JPA is part of Java EE and quite easy to use it should be the default choice.
- Choose Hibernate if there are specific reasons, like existing applications, developer knowledge or some particular feature.

## A Comparison (cont)

- JPA contains only definitions, it does not contain the implementation (provider) that makes the actual database calls.
  - Hibernate contains both definition and implementation.
- When using JPA a provider is needed. Java EE ships with EclipseLink (Oracle).
  - Another idea is to use Hibernate as provider.

# JPA: Java Persistence API

JPA Home Page:

<http://www.oracle.com/technetwork/java/javasee/tech/persistence-jsp-140049.html>

Specification:

<http://www.jcp.org/en/jsr/detail?id=220>

# JPA Architecture

- Configured with annotations
- Easy to call from both Spring and EJB.
- Transactions are propagated from calling framework.
- Entities are plain java objects.
- Reads entity's byte code and uses post compilation when needed.

# Entity

- Persistent object
  - Typically in the model.
- Typically (but not necessarily) one entity per table and one instance per row in that table.
- Either fields or properties (JavaBeans style) are persisted.
  - If fields or properties are persisted is decided either explicitly by the **Access** annotation or implicitly by the location of other annotations (close to fields or properties).
- Object/Relational (O/R) mapping with annotations to map objects to underlying relational data store.



# Requirements for Entity Classes

- Annotated with the `javax.persistence.Entity` annotation.
- **public** or **protected**, no-argument constructor.
  - May have other constructors as well.
- Must not be declared **final**. Nor methods or persistent instance variables may be declared **final**.

## Requirements for Entity Classes (cont)

- Persistent instance variables must be declared **private**, **protected**, or package-private, and can only be accessed directly by the entity class's methods.
- Instance variables must not be accessed by clients of the entity.

# Requirements for Entity Classes (cont)

- Persistent fields or properties may be of the following types:
  - Any primitive type
  - Time specification classes, that is `java.util.Date` or `java.util.Calendar`
  - Any `Serializable` type
  - Enums
  - Any entity type
  - Collections (`java.util.Collection`, `java.util.List`, `java.util.Set` or `java.util.Map`) of entities.
  - Embeddable classes (explained below).

# Persistent Fields

- Persistence runtime accesses entity class instance variables directly.
- All fields not declared **transient** and not annotated **Transient** will be persisted.

# Persistent Properties

- Persistence runtime accesses entity state via the property accessor methods.
- All properties not annotated **Transient** will be persisted.
- Property accessor methods must be **public** or **protected**.

# Persistent Properties (cont)

- The following accessor methods must exist for each property:
  - **Type** `getProperty()`
  - **void** `setProperty(Type type)`

# Primary Keys

- Each entity has a unique object identifier, a primary key.
- A simple (non-composite) primary key must correspond to a single persistent field or property of the entity class.
  - The **Id** annotation is used to denote a simple primary key.

# Composite Primary Keys

- Typically used when mapping from databases where the primary key is comprised of several columns.
- Composite primary keys must be defined in a primary key class.



# Composite Primary Keys (cont)

- Composite primary keys must correspond to either a single persistent property or field, or to a set of single persistent properties or fields in the primary key class.
- Composite primary keys are defined using the `javax.persistence.EmbeddedId` and `javax.persistence.IdClass` annotations.

# An example

```
package account;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Account {

    @Id
    private int acctNo;
    private String firstName;
    private String lastName;
    private int balance;
```

# An example (cont)

```
public Account() {  
    }  
    public Account(int acctNo, String firstName, String lastName,  
                   int balance) {  
        this.acctNo = acctNo;  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.balance = balance;  
    }  
  
    public int getAcctNo() {  
        return acctNo;  
    }  
  
    // More business methods.
```

# Context

- A persistence context is a set of managed entity *instances* that exist in a particular data store.
- A context is the scope under which entity *instances* exist.

# EntityManager

- The **EntityManager** interface defines the methods that are used to interact with the context, for example create, remove and find.
- Each **EntityManager** instance is associated with a single context.

# EntityManager (cont)

- Applications that are container-managed (for example EJB applications) can obtain entity managers with injection:

**@PersistenceContext**

**EntityManager em;**

- The container will create an entity manager instance and store it in the em field.
- The container will assure that all entity managers used in the same transaction will handle the same context.

# Persistence Unit

- Applications that are not container-managed (for example servlet application and Java SE applications) must call EntityManagerFactory to have an entity manager created:

```
@PersistenceUnit
```

```
EntityManagerFactory emf;
```

```
EntityManager em = emf.createEntityManager();
```

# Persistence Unit (cont)

- Persistence Unit
  - Defines the entities that are managed by an entity manager.
  - Defines where to store the entities persistently.



# Entity Instance's Life cycle

- The life cycle of an entity instance is managed by the **EntityManager**.
- Entity instances are in one of four states: *new*, *managed*, *detached*, or *removed*.

# Entity Instance's Life cycle (cont)

- *New* entity instances have no persistent identity and are not yet associated with a persistence context.
- *Managed* entity instances have a persistent identity and are associated with a persistence context.

# Entity Instance's Life cycle (cont)

- *Detached* entity instances have a persistent identity and are not currently associated with a persistence context.
- *Removed* entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

# Entity Instance's Life cycle (cont)

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product
product, int quantity) {
    LineItem li = new LineItem(order, product, quantity);
// new
    order.getLineItems().add(li);
    em.persist(li); // managed
}
```

- The entity (**li**) is *new* after this statement.
- The entity is *managed* after this statement.

# Entity Instance's Life cycle (cont)

```
public void removeOrder(Integer orderId) {  
    try {  
        Order order = em.find(Order.class,  
            orderId);  
        em.remove(order);  
    }  
}
```

- Entities are looked up with the **EntityManager** method **find** (more on queries below).
- Entities are removed with the **EntityManager** method **remove**.

## Entity Instance's Life cycle (cont)

- The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits.
- To force synchronization of the managed entity to the database, invoke the `flush` method of the `EntityManager`.

# Transactions

- Container-managed
  - The preferred way.
  - Can only be used when JPA entities stays in a transaction aware container (e.g EJB or Spring)
  - Transactions propagate from the calling container and are not handled by JPA code.
  - Use declarative transaction demarcation in the container.

# Transactions (cont)

- Application-managed
  - The only choice if JPA is not used inside a transaction aware container.
  - Typically used when JPA is called from a standalone Java program or from a Servlet container.
  - Transaction must be started and stopped programmatically through the `EntityManagerTransaction` interface.
  - Easy to make mistakes!



# Embeddable classes

- Ordinary java class that is a persistent property or field of an entity.
- Have no identity and can not be shared between entities.
- Follow the same rules as entities (no-arg constructor, not **final** etc) except that they are annotated **@Embeddable** instead of **@Entity**.
- Its persistent fields/properties may be primitive types, other embeddable classes, entities or collections of these three kinds.

# Relationships

- Relationships are persistent.
- Can be between two entities or between an entity and an embeddable class.
- Can be unidirectional or bidirectional.
- Can be one-to-one, one-to-many, many-to-one or many-to-many
- Changes cascade (if so is specified) when saved to the database.

# Relationships, example

```
@Entity
public class Employee {
    private Cubicle assignedCubicle;

    @OneToOne
    public Cubicle
getAssignedCubicle() {
    return assignedCubicle;
}

    public void setAssignedCubicle(
        Cubicle cubicle) {
        assignedCubicle = cubicle;
    }
    ...
}
```

```
@Entity
public class Cubicle {
    private Employee residentEmployee;

    @OneToOne(mappedBy="assignedCubicle")
    public Employee getResidentEmployee()
    {
        return residentEmployee;
    }

    public void setResidentEmployee(
        Employee employee) {
        residentEmployee = employee;
    }
    ...
}
```

# Relationships, direction

- Unidirectional relationships can only be navigated in one way.
  - Cascading updates
  - Searches
- Have relationship annotation only on one side.

## Relationships, direction (cont)

- Bidirectional relationships can be navigated in both ways.
- Have relationship annotations on both sides.
- Inverse (not owning) side specifies that it is mapped by the property or field on the owning side.

```
@OneToOne(mappedBy="assignedCubicle")
```

## Relationships, direction (cont)

- The relationship itself is persisted based on the owning side.
- The owning side has the foreign key.

# Relationships, multiplicity

- The following annotations exist:
  - **OneToOne**
  - **OneToMany**
  - **ManyToOne**
  - **ManyToMany**
- For **OneToOne** and **ManyToMany** relationships any side may be the owning side.

# Relationships, multiplicity (cont)

@Entity

```
public class Employee {  
    private Department department;
```

@ManyToOne

```
public Department getDepartment() {  
    return department;  
}
```

```
public void setDepartment(Department  
department) {  
    this.department = department;  
}
```

...

```
}
```



# Relationships, multiplicity (cont)

@Entity

```
public class Department {
    private Collection<Employee> employees = new
HashSet();

    @OneToMany(mappedBy="department")
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee>
employees) {
        this.employees = employees;
    }
    ...
}
```

# Relationships, cascading updates

- Updates to the database may cascade along relationships.
  - Specified by the **cascade** element of the relationships annotations.
  - **ALL**, Cascade all operations
  - **MERGE**, Cascade merge operation
  - **PERSIST**, Cascade persist operation
  - **REFRESH**, Cascade refresh operation
  - **REMOVE**, Cascade remove operation

# Relationships, cascading updates (cont)

Example:

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() {
    return orders;
}
```

# Queries

- Query methods are in **EntityManager**.
- The **find** method can be used to find instances by primary key.

```
em.find(Order.class, orderId);
```

# Queries (cont)

- The `createQuery` method is used to create dynamic queries, queries that are defined directly within an application's business logic.

```
@PersistenceContext
public EntityManager em;
...
public List findWithName(String name) {
    Query query = em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE
:custName");
    query.setParameter("custName", name);
    return query.getResultList();
}
```

# Queries (cont)

- The **createNamedQuery** method is used to create static queries, queries that are defined in meta data using the **NamedQuery** annotation.

```
@NamedQuery(  
    name="findAllCustomersWithName",  
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"  
)
```

```
@PersistenceContext  
public EntityManager em;  
...  
Query query = em.createNamedQuery("findAllCustomersWithName");  
query.setParameter("custName", "Smith");  
customers = query.getResultList();
```

# Java Persistence Query Language, JPQL

- The two preceding slides use Java Persistence query language.
  - SQL-like language.
  - See for example the Java EE tutorial or the specification.

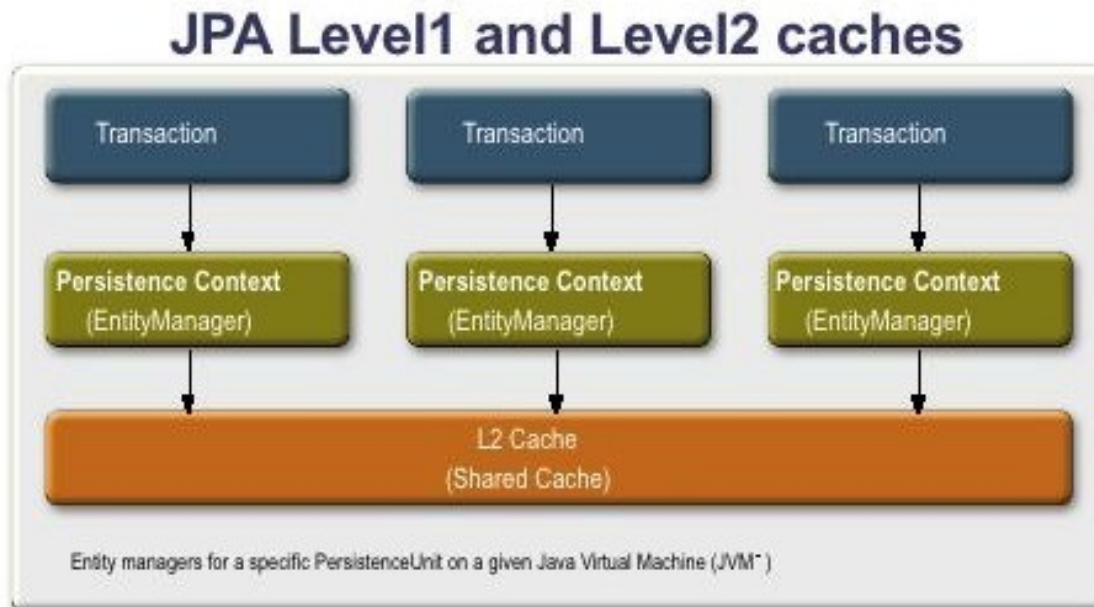
# Criteria API

- The criteria API provides a way to generate queries in an object-oriented way with ordinary method calls, as opposed to the string manipulation used by JPQL.
- The advantage over JPQL is that it is type safe and that it is not required to know field names at compile time.
- The disadvantage is that notably more code is required to generate queries and that it is harder to read the queries.



# Cache

- The JPA specification includes optional support for caching in the JPA provider (e.g. EclipseLink).
  - EclipseLink provides such a cache.
- This is called *second-level cache*, as opposed to the *first-level cache* which is maintained by the persistence context .

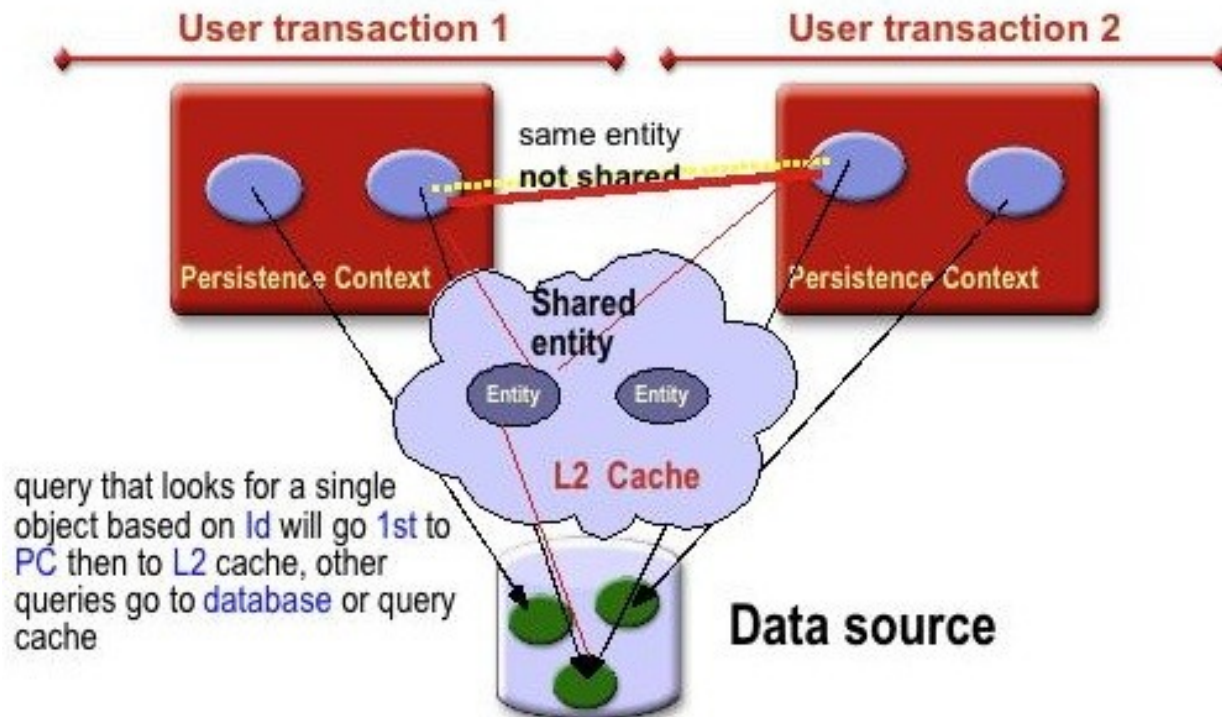


# Cache, Cont'd

- Cache tuning is one of the topics that are important for database performance.
- Information about JPA and EclipseLink caches can be found at the following URLs.
  - `https://blogs.oracle.com/carolmcdonald/entry/jpa\_caching`
  - `http://wiki.eclipse.org/Introduction\_to\_Cache\_%28ELUG%29`

# Locks

- As can be seen below, multiple transactions might be using the same entity instance in the L1 cache, the L2 cache and the database.
- Therefore, locking is required.



# Locks, Cont'd

- The most used locking mechanism is *optimistic locking*.
  - Allows concurrent accesses but detects them.
  - If concurrent access are detected the transaction is rolled back and an exception is thrown.
- Optimistic locking is the best alternative when conflicts are not so frequent, i.e. when updates are not frequent.

# Locks, Cont'd

- Optimistic locking is implemented using a version number for the data (entity).
  - Whenever an entity instance is read, its current version number is also read.
  - When the instance is stored, the version in the database is compared to the Java object's version.
  - If the version numbers differ it means someone else updated the database and there is a conflict.

# Optimistic Lock Example

- In the entities, add a version field.

```
@Version
@Column(name="OPTLOCK")
private int versionNum;
```

- In the queries, specify that the found entities shall be optimistically locked.

```
@NamedQuery (
    name = "findAccountWithName",
    query = "SELECT acct FROM Account acct WHERE
acct.owner.name LIKE :ownerName",
    lockMode =
LockModeType.OPTIMISTIC_FORCE_INCREMENT)
```

# Optimistic Lock Example, Cont'd

- Optimistic locking can also be specified in the entity manager using the **lock** or **find** methods.

# More information on Locking

- `https://blogs.oracle.com/carolmcdonald/entry/jpa\_2\_0\_concurrency\_and`



# Lazy Loading

- Say that we have two entities, A and B, and that A has a reference to B. When A is loaded, B will also be loaded and it will be possible to access B writing something similar to **`a.getB()`**, provided that **`a`** is an instance of A.
- If entity B is never accessed in the program, then it was a waste of resources to read the B instance from the database.

## Lazy Loading, Cont'd

- To avoid this unnecessary read, we can specify that B should be *lazy loaded*.
- This means that the instance of B will not be read from the database when the A instance is read, but when the B instance is accessed, that is when **a.getB()** is called.
- The opposite, that the B instance *is* read from the database when the A instance is read, is called *eager loading*.

# Lazy Loading, Cont'd

- If we always use eager loading we might suffer severe performance penalties by loading (lots of) unused objects.
- Also lazy loading can bring performance penalties if used the wrong way.
  - Suppose that we load entity A, which has a one to many relation to B.
  - Also suppose that we will iterate through all B instances referenced by the A instance.
  - This means that there will be one separate database call for each instance of B instead of loading them all in the same call, which would be the case if eager load was used for B.

## Lazy Loading, Cont'd

- Which loading strategy that is used can be specified in all relationship annotations, i.e **OneToOne**, **OneToMany**, **ManyToOne** and **ManyToMany**.
- More information about loading strategies at [https://blogs.oracle.com/carolmcdonald/entry/jpa\\_performance\\_don\\_t\\_ignore](https://blogs.oracle.com/carolmcdonald/entry/jpa_performance_don_t_ignore)

# Performance Conclusion

- As seen from this very brief overview there are lots of properties related to performance of JPA and of the JPA provider.
- Also the DBMS itself provides caching, locking and lots of other configuration possibilities that are important for performance.
- The bottom line is that good knowledge about the O/R mapping technology and the DBMS is *necessary* to be able to write an efficient application.
  - However, this is not a mandatory part of this course.