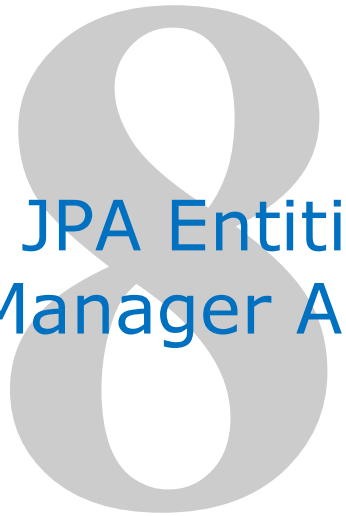


Manipulating JPA Entities With Entity Manager API

Objectives

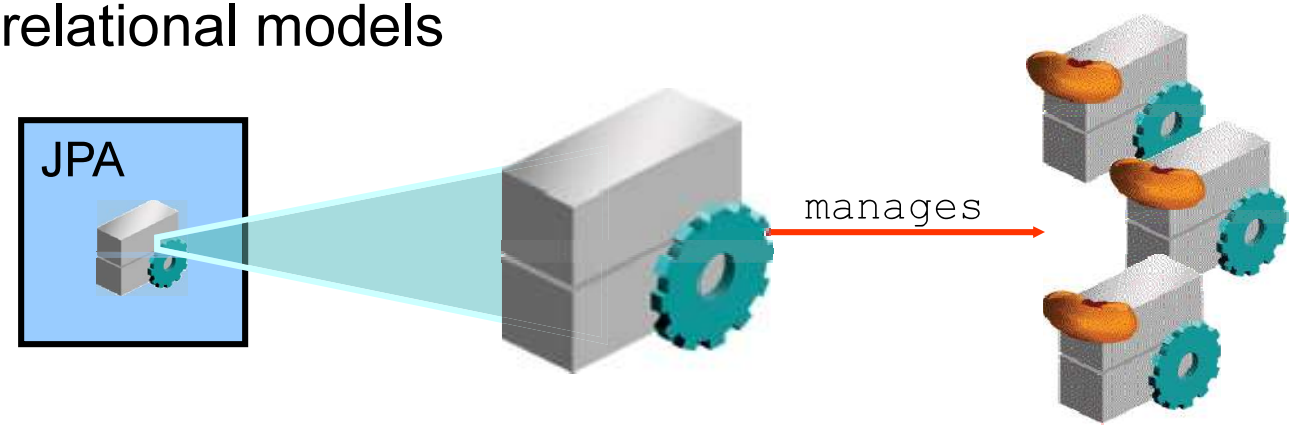
After completing this lesson, you should be able to do the following:

- Declare an `EntityManager` reference with the `@PersistenceContext` annotation
- Look up an `EntityManager` reference by using dependency injection
- Use the `EntityManager` API to:
  - Find an entity by its primary key
  - Insert a new entity
  - Modify an existing entity
  - Delete an entity
- Execute dynamic queries with the `Query` API
- Write simple JPQL queries

What is `EntityManager`?

`EntityManager`:

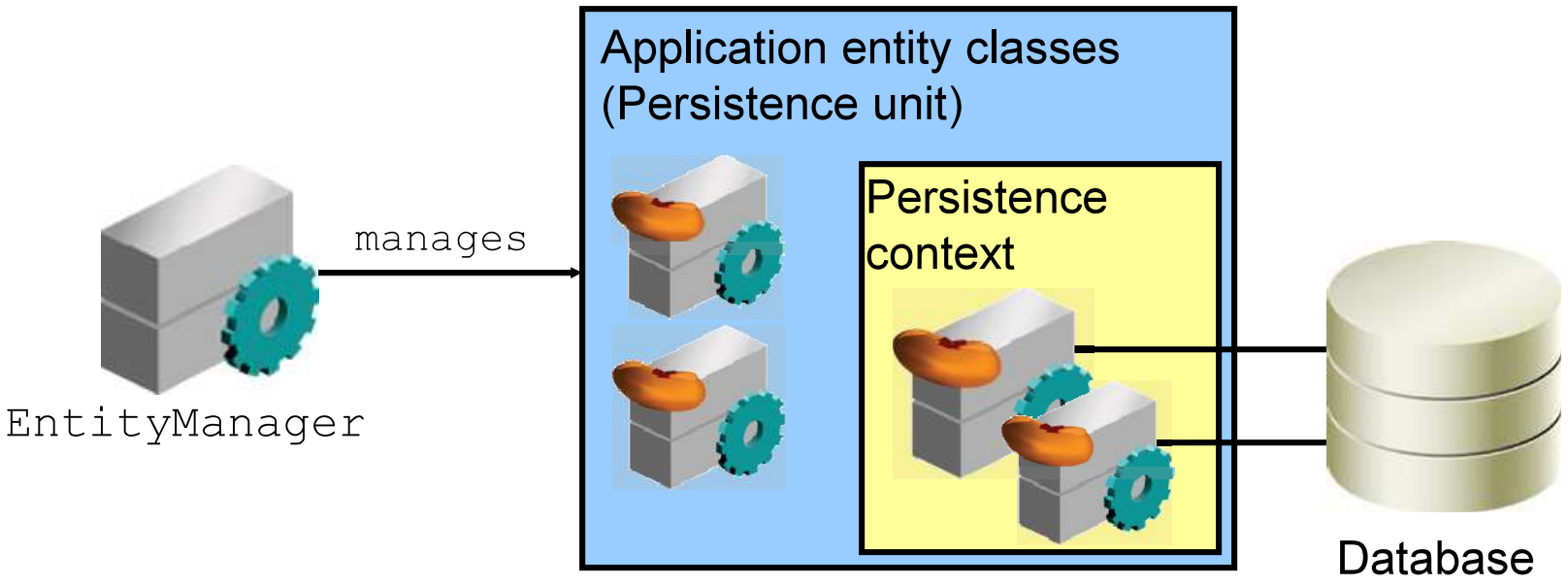
- Is an interface defined in JPA
- Is a standard API for performing CRUD operations for entities
- Acts as a bridge between the object-oriented and the relational models



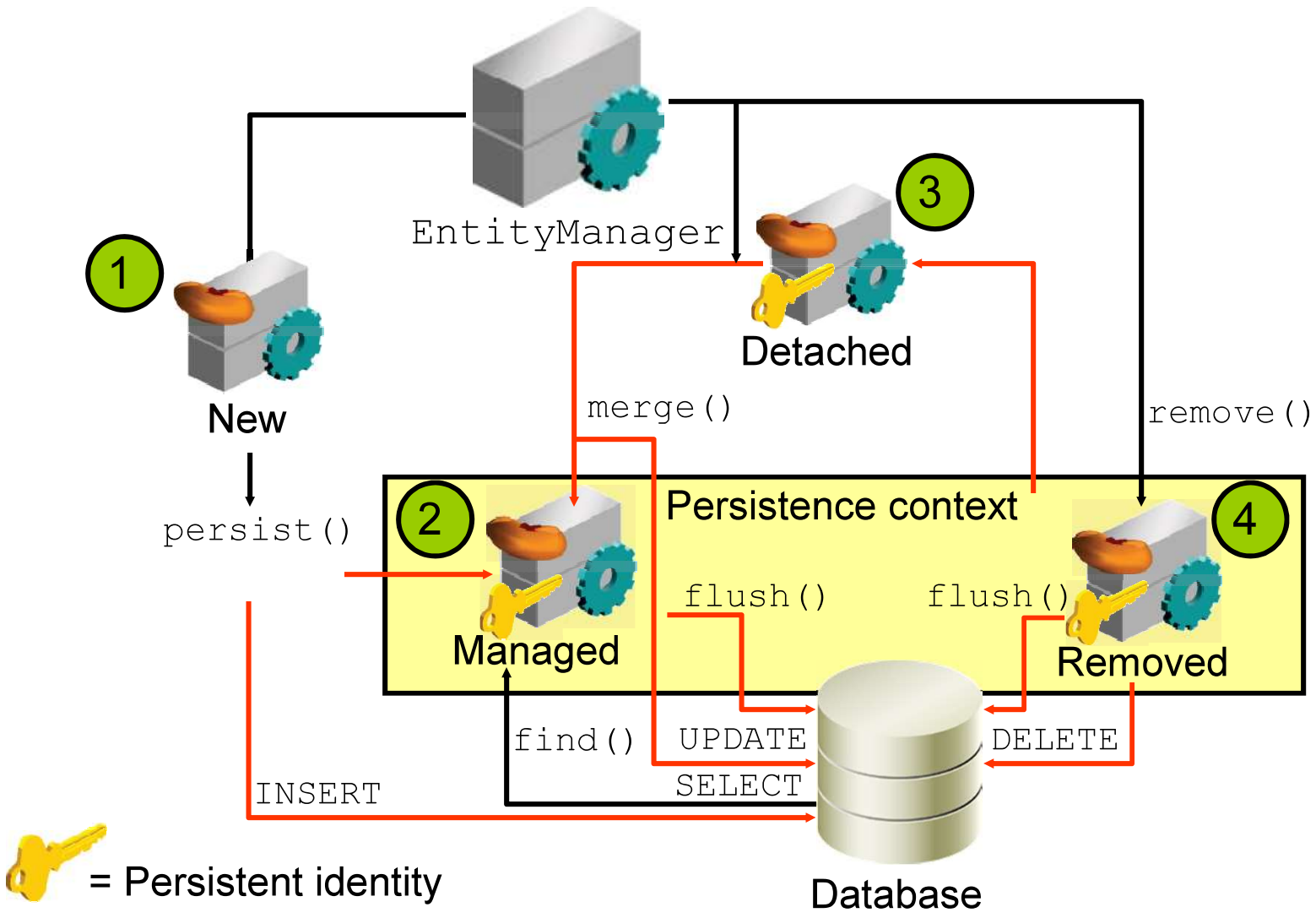
What Is `EntityManager`?

`EntityManager` is:

- Associated with a persistence context
- An object that manages a set of entities defined by a persistence unit



Managing an Entity Life Cycle



Accessing an `EntityManager` Instance

- Container-managed `EntityManager` instances:
  - Are implemented inside a Java EE container
  - Use the `@PersistenceContext` annotation
  - Are obtained in an application through dependency injection or JNDI lookup
- Application-managed `EntityManager` instances:
  - Are implemented outside a Java EE container
  - Are obtainable by using the `EntityManagerFactory` interface

Creating a Container-Managed `EntityManager` Instance

A session bean using container injection:

```
@Stateless

public class CustomerBean {
    @PersistenceContext(unitName="Model")
    private EntityManager em;
    ...
    public void createCustomer() {
        final Customer cust = new Customer();
        cust.setName("Valli Pataballa");
        ...
        em.persist(cust);
    }
    ...
}
```

Creating an Application-Managed `EntityManager` Instance

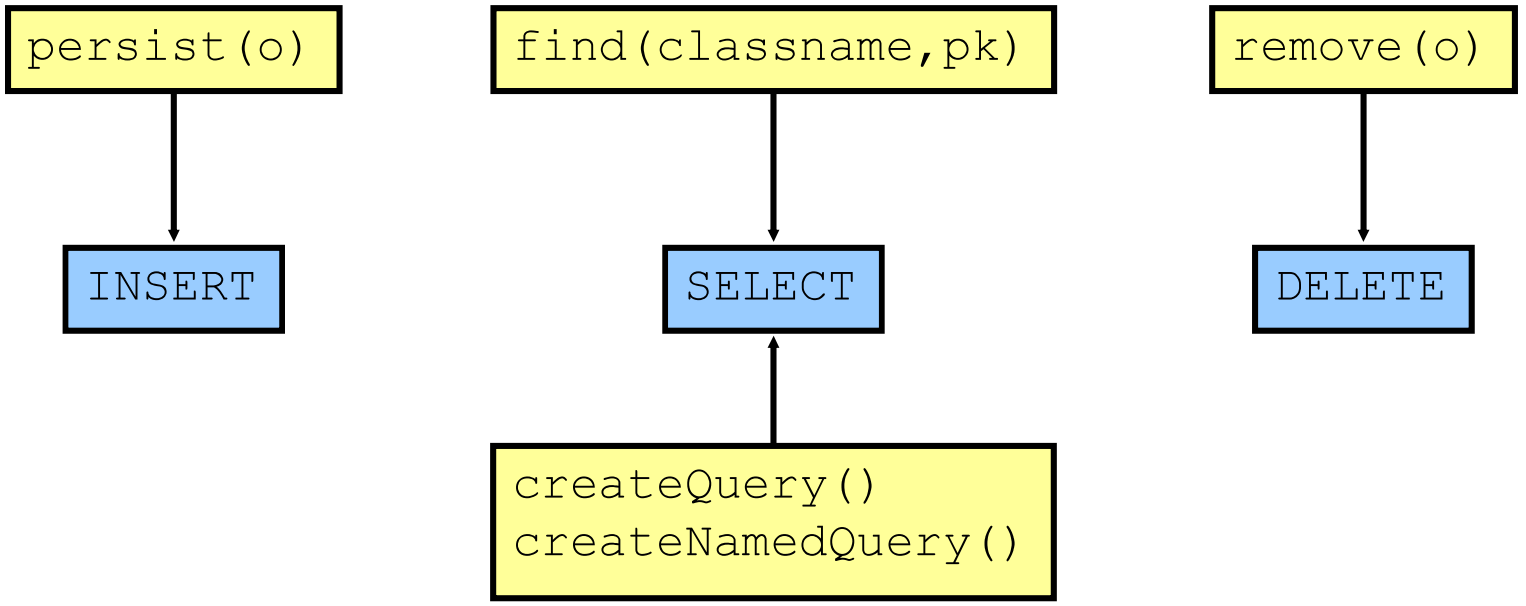
A Java SE application using an `EntityManagerFactory` API:

```
public class CustomerApp {
    public static void main(String args[]) {
      final EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("Model");
      final EntityManager em =
        emf.createEntityManager();
      final Customer cust = new Customer()
      cust.setName("Ron Howard");
      ...
      em.persist(cust);

    }
}
```

Specifying Database Operations with

The `EntityManager` API provides the following methods that map to CRUD database operations:



Commonly Used Methods in the

Method signature of the most commonly used methods of the `EntityManager` interface:

```
public void persist(Object entity);
public <T> T merge(T entity);
public void remove(Object entity);
public <T> T find(class<T> entityClass, Object
                                          primaryKey);
public void flush();
public void refresh(Object entity);
public void clear();
public Query createQuery (String jpqlString);
public Query createNamedQuery (String name);
```

Quiz

The application-managed `EntityManager` is implemented outside a Java EE container.

1. True
2. False

Inserting New Data

To insert new data, perform the following steps:

1. Create a new entity object.
2. Call the `EntityManager.persist()` method.

```
@PersistenceContext(unitName="Model")
private EntityManager em; // inject the EntityManager
...                       // object
public void persistUser() {
  Users user = new Users();
  user.setFirstName("Steve");
  user.setLastName("King");

  em.persist(user);
  // On return the user object contains persisted state
  // including fields populated with generated id values
}
```

Deleting Data

To delete data, perform the following steps:

1. Find, set, or refresh the state of the entity to be deleted.
2. Call the `EntityManager.remove()` method.

```
@PersistenceContext(unitName="Model")
private EntityManager em;
...
// Remove a Product by primary key Id value
 public void removeProducts(Products products) {
   products = em.find(Products.class,
                                products.getProdId());
   em.remove(products);
 }
...
```

Updating and Synchronizing the Entity with the Database

1. Retrieve the rows in a database table into the entity by using the `EntityManager` API.
2. Use the setter methods of the entity to update the data attributes.
3. Use the `flush()` method to synchronize the state of the entities in the `EntityManager`'s persistence context with the database.
4. Use the `merge()` method to reattach an entity to the persistence context to synchronize it with the database.

Updating Data

To update data, perform the following steps:

1. Find, set, or refresh the state of the entity to be updated.
2. Call the `EntityManager.merge()` method.

```
@PersistenceContext(unitName="Model")
private EntityManager em;
...
// Update a Product by primary key Id value
 public void updateProducts(Products products) {
   products = em.find(Products.class,
                                products.getProdId());
   products.setListPrice("1000");
...
   em.merge(products);
 }
...
```

Finding an Entity by Primary Key

To locate an entity by primary key, perform the following steps:

1. Create and set the primary key object and value.
2. Call the `EntityManager find()` method with the following parameters:
   - Entity class
   - Primary-key object

```
import org.srdemo.persistence.Users;
@PersistenceContext(unitName="Model")
private EntityManager em;
...
public Users findUserByPrimaryKey(Long id) {
  Users user = null;
  user = em.find(Users.class, id);
  return user;
}
```

Quiz

The `EntityManager.flush()` method:

1. Retrieves the rows in a database table into the entity
2. Reattaches an entity to the persistence context
3. Synchronizes the state of the entity in the `EntityManager`'s persistence context with the database

What Is JPA `Query` API?

The JPA `Query` API:

- Includes:
  - `EntityManager` methods to create queries
  - `Query` interface methods for executing queries
  - Java Persistence Query Language (JPQL)
- Supports:
  - Named queries
  - Dynamic queries

Retrieving Entities by Using the `Query` API

The `EntityManager` interface provides the `Query` API methods to execute JPQL statements:

`EntityManager`

- `createQuery(String jpql)`
- `createNamedQuery(String name)`

`Query` instance methods:

- `setParameter(String, Object)`
- `Object getSingleResult()`
- `List getResultList()`

- `Query setMaxResults(int)`
- `Query setFirstResult(int)`

- `int executeUpdate()`

Query Language

- The Java Persistence Query Language (JP QL) is the standard query language of JPA.
- It is a portable query language designed to combine the syntax and simple query semantics of SQL with the expressiveness of an object-oriented expression language.
- Queries written using this language can be portably compiled to SQL on all major database servers.
- JP QL is a language for querying entities. Instead of tables and rows, the currency of the language is entities and objects.

Writing a Basic JPQL Statement

- Syntax for a simple JPQL statement:

```
SELECT object(o)
FROM abstract-schema-name o
[WHERE <conditional_expression>]
[ORDER BY <order_by_clause>]
```

  - Find all Users entities:

```
SELECT object(o) FROM Users o
```

```
SELECT object(o) FROM Users o
WHERE o.email = 'steve.king@srdemo.org'
```

```
SELECT object(o) FROM Users o
WHERE o.firstName = :givenName
```

Writing a Basic JPQL Statement

- Selecting few Attributes

```
SELECT id, name, salary, manager_id, dept_id, address_id
FROM emp
```

- To remove the duplicates, the DISTINCT operator must be used

```
SELECT DISTINCT e.department FROM Employee e
```

Joins

- A join is a query that combines results from multiple entities. Joins in JP QL queries are logically equivalent to SQL joins.
- Ultimately, once the query is translated to SQL, it is quite likely that the joins between entities will produce similar joins among the tables to which the entities are mapped.

Inner Joins

The syntax of an inner join using the JOIN operator is

[INNER] JOIN <path_expression> [AS] <identifier>

```
SELECT  p
FROM Employee e JOIN e.phones p
```

```
SELECT p.id, p.phone_num, p.type, p.emp_id
FROM emp e, phone p
WHERE e.id = p.emp_id
```

Outer Joins

An outer join is specified using the following syntax:

LEFT [OUTER] JOIN <path_expression> [AS] <identifier>

```
SELECT e, d
FROM Employee e LEFT JOIN e.department d
```

```
SELECT e.id, e.name, e.salary, e.manager_id, e.dept_id,
e.address_id,
d.id, d.name
FROM employee e LEFT OUTER JOIN department d
ON (d.id = e.department_id)
```

BETWEEN And Like Expressions

The BETWEEN operator can be used in conditional expressions to determine whether the result of an expression falls within an inclusive range of values. Numeric, string, and date expressions can be evaluated

```
SELECT e
FROM Employee e
WHERE e.salary BETWEEN 40000 AND 45000
```

JP QL supports the SQL LIKE condition to provide for a limited form of string pattern matching.

```
SELECT d
FROM Department d
WHERE d.name LIKE '__Eng%'
```

Subqueries

Subqueries can be used in the WHERE and HAVING clauses of a query. A subquery is a complete select query inside a pair of parentheses that is embedded within a conditional expression.

```
SELECT e
FROM Employee e
WHERE e.salary = (SELECT MAX(emp.salary)
FROM Employee emp)
```

IN Expressions

The IN expression can be used to check whether a single-valued path expression is a member of a collection.

```
SELECT e
FROM Employee e
WHERE e.address.state IN ('NY', 'CA')
```

ORDER BY Clause

Queries can optionally be sorted using ORDER BY

```
SELECT e
FROM Employee e
ORDER BY e.name DESC
```

Function Expressions

Function	Description
ABS(number)	The ABS function returns the unsigned version of the number argument. The result type is the same as the argument type (integer, float, or double).
CONCAT(string1, string2)	The CONCAT function returns a new string that is the concatenation of its arguments, string1 and string2.
CURRENT_DATE	The CURRENT_DATE function returns the current date as defined by the database server.
CURRENT_TIME	The CURRENT_TIME function returns the current time as defined by the database server.
CURRENT_TIMESTAMP	The CURRENT_TIMESTAMP function returns the current timestamp as defined by the database server.
INDEX(identification variable)	The INDEX function returns the position of an entity within an ordered list.
LENGTH(string)	The LENGTH function returns the number of characters in the string argument.
LOCATE(string1, string2 [, start])	The LOCATE function returns the position of string1 in string2, optionally starting at the position indicated by start. The result is zero if the string cannot be found.
LOWER(string)	The LOWER function returns the lowercase form of the string argument.
MOD(number1, number2)	The MOD function returns the modulus of numeric arguments number1 and number2 as an integer.

Cont...

Function	Description
SIZE(collection)	The SIZE function returns the number of elements in the collection, or zero if the collection is empty.
SQRT(number)	The SQRT function returns the square root of the number argument as a double.
SUBSTRING(string, start, end)	The SUBSTRING function returns a portion of the input string, starting at the index indicated by start up to length characters. String indexes are measured starting from one.
UPPER(string)	The UPPER function returns the uppercase form of the string argument.
TRIM([[LEADING|TRAILING|BOTH] [char] FROM] string)	The TRIM function removes leading and/or trailing characters from a string. If the optional LEADING, TRAILING, or BOTH keyword is not used, both leading and trailing characters are removed. The default trim character is the space character.

Aggregate Queries

An aggregate query is a variation of a normal select query. An aggregate query groups results and applies aggregate functions to obtain summary information about query results. A query is considered an aggregate query if it uses an aggregate function or possesses a GROUP BY clause and/or a HAVING clause.

```
SELECT <select_expression>
FROM <from_clause>
[WHERE <conditional_expression>]
[GROUP BY <group_by_clause>]
[HAVING <conditional_expression>]
[ORDER BY <order_by_clause>]
```

Avg Function

The AVG function takes a state field path expression as an argument and calculates the average value of that state field over the group. The state field type must be numeric, and the result is returned as a Double.

```
SELECT AVG(e.salary)
FROM Employee e
```

```
SELECT d.name, AVG(e.salary)
FROM Department d JOIN d.employees e
WHERE e.directs IS EMPTY
GROUP BY d.name
HAVING AVG(e.salary) > 50000
```

Count Function

The COUNT function takes either an identification variable or a path expression as its argument. This path expression can resolve to a state field or a single-valued association field.

```
SELECT e, COUNT(p), COUNT(DISTINCT p.type)
FROM Employee e JOIN e.phones p
GROUP BY e
```

Max Min and Sum Function

MAX

- The MAX function takes a state field expression as an argument and returns the maximum value in the group for that state field.

MIN

- The MIN function takes a state field expression as an argument and returns the minimum value in the group for that state field.

SUM

- The SUM function takes a state field expression as an argument and calculates the sum of the values in that state field over the group.

Group By Clause

The GROUP BY clause defines the grouping expressions over which the results will be aggregated.

```
SELECT d.name, COUNT(e)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

```
SELECT d.name, COUNT(e), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d.name
```

Having Clause

The HAVING clause defines a filter to be applied after the query results have been grouped.

```
SELECT e, COUNT(p)
FROM Employee e JOIN e.projects p
GROUP BY e
HAVING COUNT(p) >= 2
```

Creating Named Queries

To create a named query, perform the following steps:

1. Define the query with the `NamedQuery` annotation.

```
@NamedQuery(name="findUsersByCity",
   query="SELECT object(o) FROM Users o " +
         "where o.city = :city");
```

2. Use the `createNamedQuery()` method, setting parameters and returning results.

```
public List<Users> findUsersinCity(String cityName) {
  Query query = em.createNamedQuery("findUsersByCity");
  query.setParameter("city", cityName);
  return query.getResultList();
}
```

Writing Dynamic Queries

Example: Find service requests by primary key and a specified status.

```
public List findServiceRequests(Long id, String status){
  if (id != null && status != null ) {
    Query query = em.createQuery(
     "select object(sr) from ServiceRequests sr " +
     "where sr.svrId = :srvId and sr.status = :status");
    query.setParameter("srvId", id);
    query.setParameter("status", status);
    return query.getResultList();
  }
  return null;
}
```

Update Queries

Update queries provide an equivalent to the SQL UPDATE statement but with JP QL conditional expressions. The form of an update query is the following:

```
UPDATE <entity name> [[AS] <identification variable>]
SET <update_statement> {, <update_statement>}*
[WHERE <conditional_expression>]
```

```
UPDATE Employee e
SET e.salary = 60000
WHERE e.salary = 55000
```

Delete Queries

The delete query provides the same capability as the SQL DELETE statement, but with JP QL conditional expressions. The form of a delete query is the following:

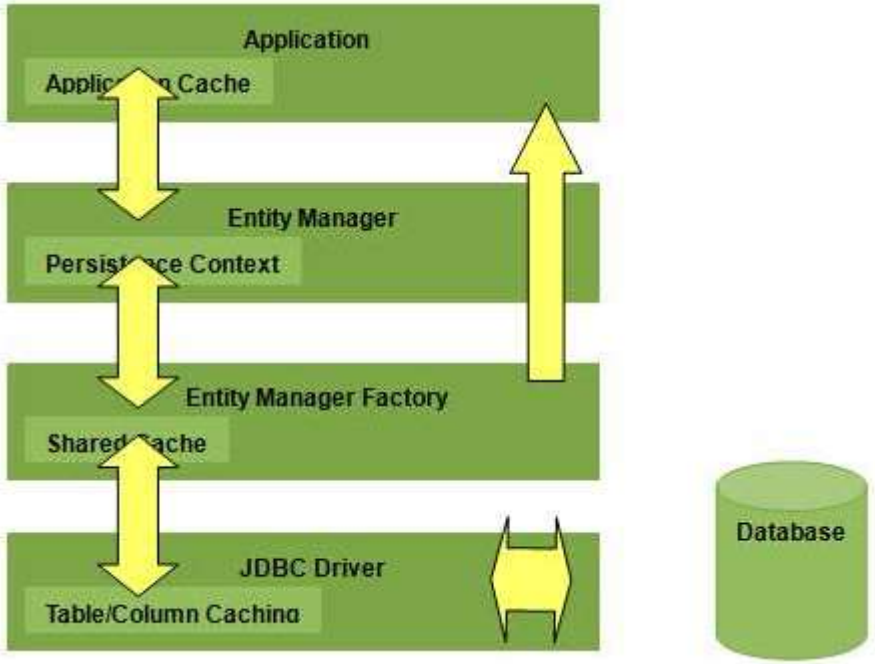
```
DELETE FROM <entity name> [[AS] <identification
variable>]
[WHERE <condition>]
```

```
DELETE FROM Employee e
WHERE e.department IS NULL
```

Using Second Level Caching in a JPA Application

- Caching structurally implies a temporary store to keep data for quicker access later on.
- Second level shared cache is an auxiliary technique, mainly used in JPA, to enhance performance; it is used especially during large inflow, outflow of data between the database and the application.
- Caching also reduces search time when the searched entity is already in the cache, otherwise it is fetched from the database to serve the purpose.
- However, when a subsequent search query is fired it takes little time as the searched entity is already in the cache. Entity Manager stores entities for almost every CRUD operation in this shared cache before flushing the content to the database.

•Second level cache is complementary to its first level counterpart. In fact, to get an overview of caching in particular, there are several layers of caching in JPA

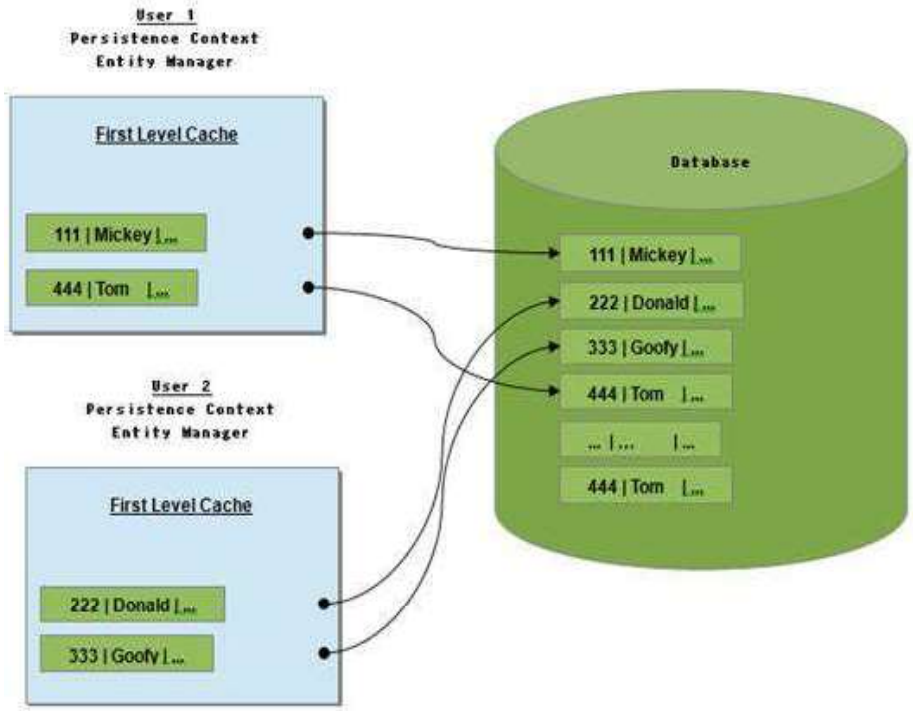


First Level Cache

- Entity Manager always maintains a cache called the first level cache, so it makes sense to call another shared caching technique in addition to first – a second level cache.
- In first level cache CRUD operations are performed per transaction basis to reduce the number of queries fired to the database.
- That is, an entity modified several times within the same transaction is done in the cache only, modification at the database level is slated until final UPDATE statement is fired at the end of the transaction.

Cont…

- JPA entities are cached at the persistence context level and guarantees that there will be one object instance per persistence context for a specific row of a database table.
- Concurrent transactions affecting the same row are managed by applying an appropriate locking mechanism in JPA.



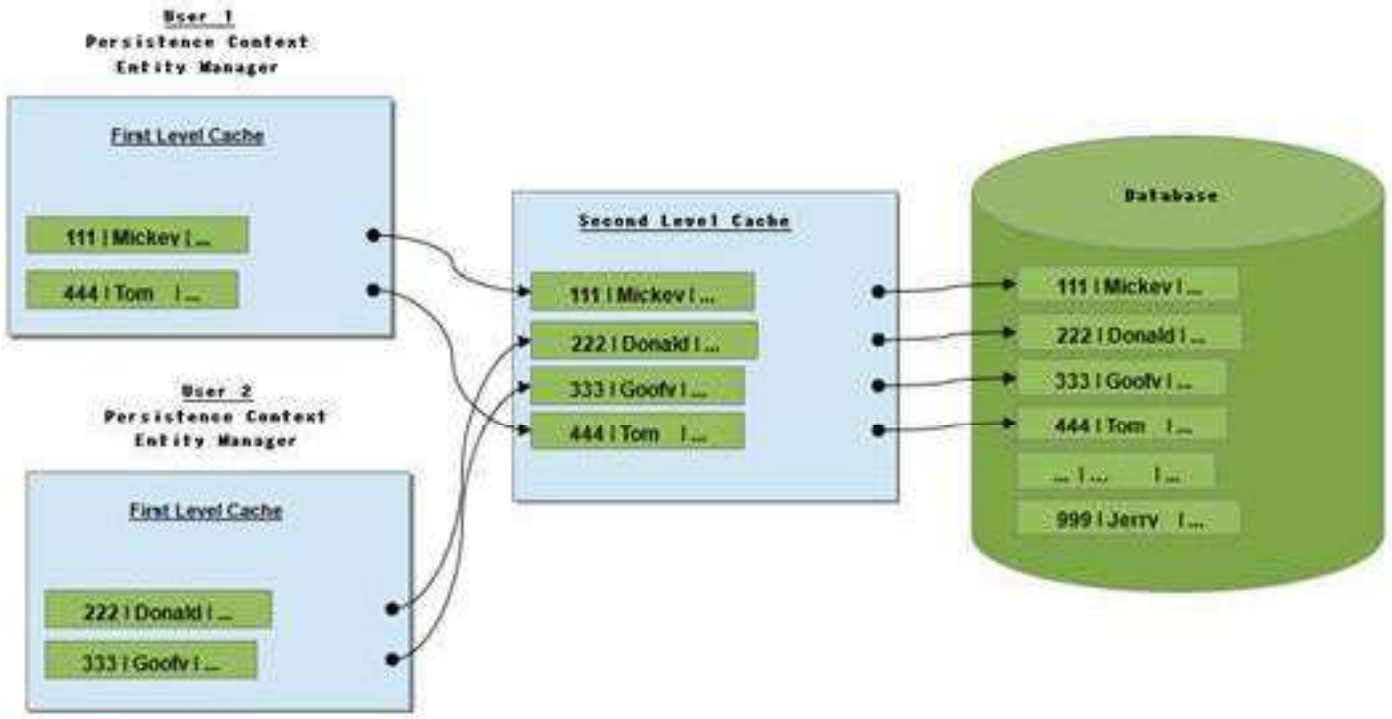
Second Level Cache

- This level of cache emerged more due to performance reasons than absolute necessity; in fact adding more shared cache increases the possibility of the problem of a stale read.
- Second level cache sits between Entity Manager and the database.
- Persistence context shares the cache, making the second level cache available throughout the application.
- Database traffic is reduced considerably because entities are loaded in to the shared cache and made available from there.

Second Level Caching Benefits

- Persistence provider manages local store of entity data
- Leverages performance by avoiding expensive database calls
- Data are kept transparent to the application
- CRUD operation can be performed through normal entity manager functions
- Application can remain oblivious of the underlying cache and do its job inadvertently

Diagrammatical Representation



Sample

```
package training.dto;
@Entity
@Cacheable(true)
public class Account{
        @Id @GeneratedValue(strategy=GenerationType.AUTO)
        private Long accountNumber;
        private String name;
        @Temporal(TemporalType.DATE)
        private Date createDate;
        private Float balance;

        //... constructors, getters, setters
}
```

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="...">
<persistence-unit name="JPALockingDemo" transaction-type="RESOURCE_LOCAL">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<class>org.mano.dto.Account</class>

<shared-cache-mode>ALL</shared-cache-mode>

<properties>
<property name="javax.persistence.jdbc.url"
value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="javax.persistence.jdbc.user" value="FOD"/>
<property name="javax.persistence.jdbc.password" value="FUSION"/>
<property name="javax.persistence.jdbc.driver"
value="oracle.jdbc.driver.OracleDriver"/>
<property name="javax.persistence.schema-generation.database.action"
value="create"/>
</properties>
</persistence-unit>
</persistence>
```

Caching Parameters

Value	Description
ALL	All entities are cached
NONE	Disable caching
ENABLE_SELECTIVE	Enable caching only for those entities which has @Cacheable(true)
DISABLE_SELECTIVE	Enable caching only for those entities which are not specified with @Cacheable(false)
UNSPECIFIED	Applies persistence provider-specific default behavior

Sample

```
package training.app;

public class Main {

        public static void main(String[] args) {
                EntityManagerFactory factory = Persistence
                                .createEntityManagerFactory("JPALockingDemo");
                EntityManager manager = factory.createEntityManager();

                Account account = new Account(1111l,"John",new Date(),23000f);
                manager.getTransaction().begin();
                manager.persist(account);
                manager.getTransaction().commit();

                Cache cache = factory.getCache();
                System.out.println("cache.contain should return true:
"+cache.contains(Account.class, account.getAccountNumber()));
                cache.evict(Account.class);
                System.out.println("cache.contain should return false:
"+cache.contains(Account.class, account.getAccountNumber()));
                manager.close();
                factory.close();
        }
}
```

Summary

In this lesson, you should have learned how to:

- Declare an `EntityManager` reference with the `@PersistenceContext` annotation
- Look up an `EntityManager` reference by using dependency injection
- Use the `EntityManager` API to:
  - Find an entity by its primary key
  - Insert a new entity
  - Modify an existing entity
  - Delete an entity
- Execute dynamic queries with the `Query` API
- Write simple JPQL queries

Practice: Overview

This practice covers the adding named queries to entities.