UNIVERSITAS SCIENTIARUM SZEGEDIENSIS
UNIVERSITY OF SZEGED
Department of Software Engineering

10. Data Handling JPA (JSR 338)-Hibernate

Vilmos Bilicki PhD
University of Szeged
Department of Software Engineering

UNIVERSITAS SCIENTIARUM SZEGEDIENSIS
UNIVERSITY OF SZEGED
Department of Software Engineering

Oveview

- JDBC
- JPA
- Hibernate

Issues

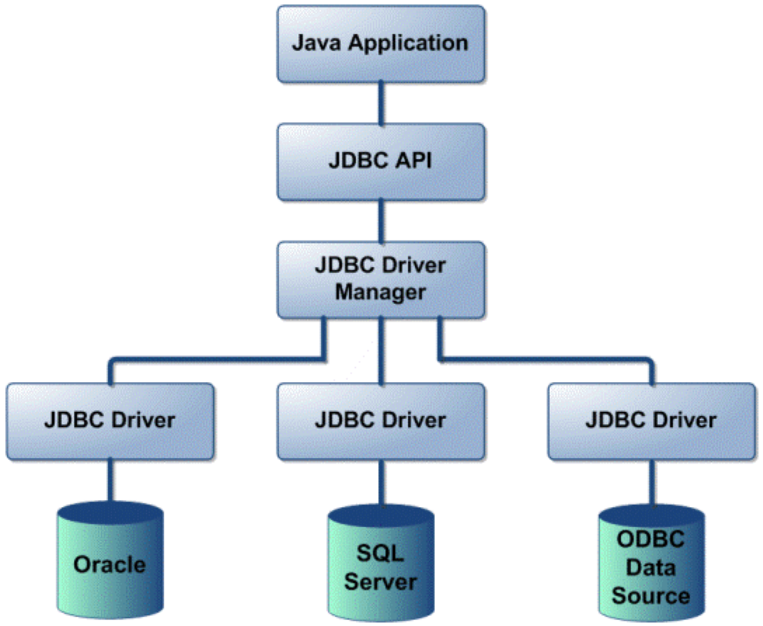
- Accessing the database from Java - JDBC
- ACID vs. Long Processes (Unit of Work?)
- Relation vs. Object Oriented
- Memory vs. Database

Peristence

- Persitent object: it is saved on durable storage
- Saving/Loading a part of object hierarchy
  - Into file (Object Serialization API)
    - Types (byte series)
    - References? Searching? Update? Security?
  - Object Oriented database
    - No need for conversion(object→relation)
    - The data handling is not efficient
    - New unstable technology
  - Relational Database
    - The object realtional mapping is painfull

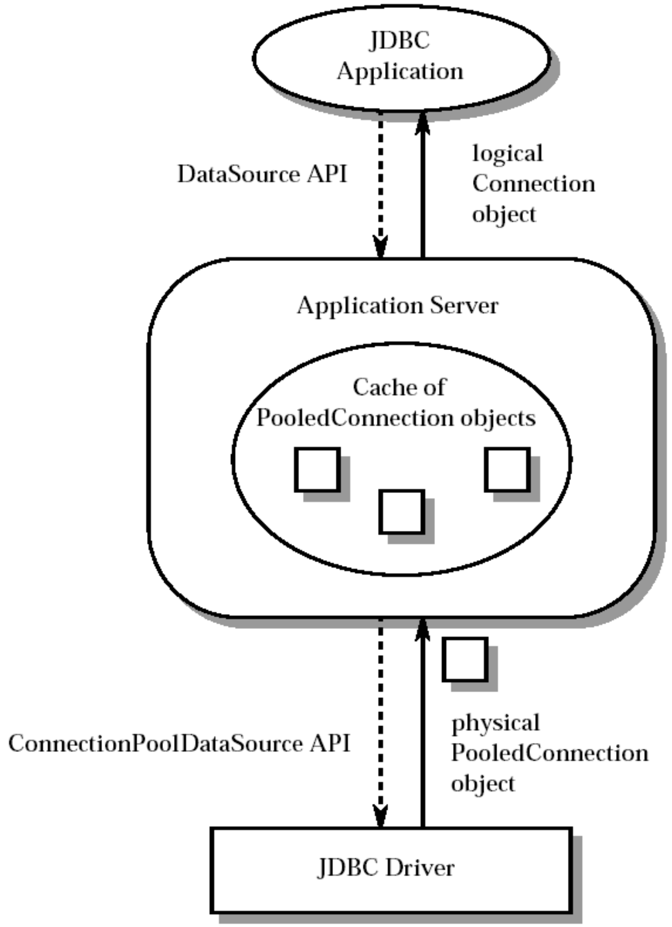
JDBC

- Platform and solution provider independent
- A simple Java API for developers
- JDBC driver manager
- Provider specific drivers
- Similar solution to that of ODBC (C)
- The encryption of the data is the duty of the proivdier

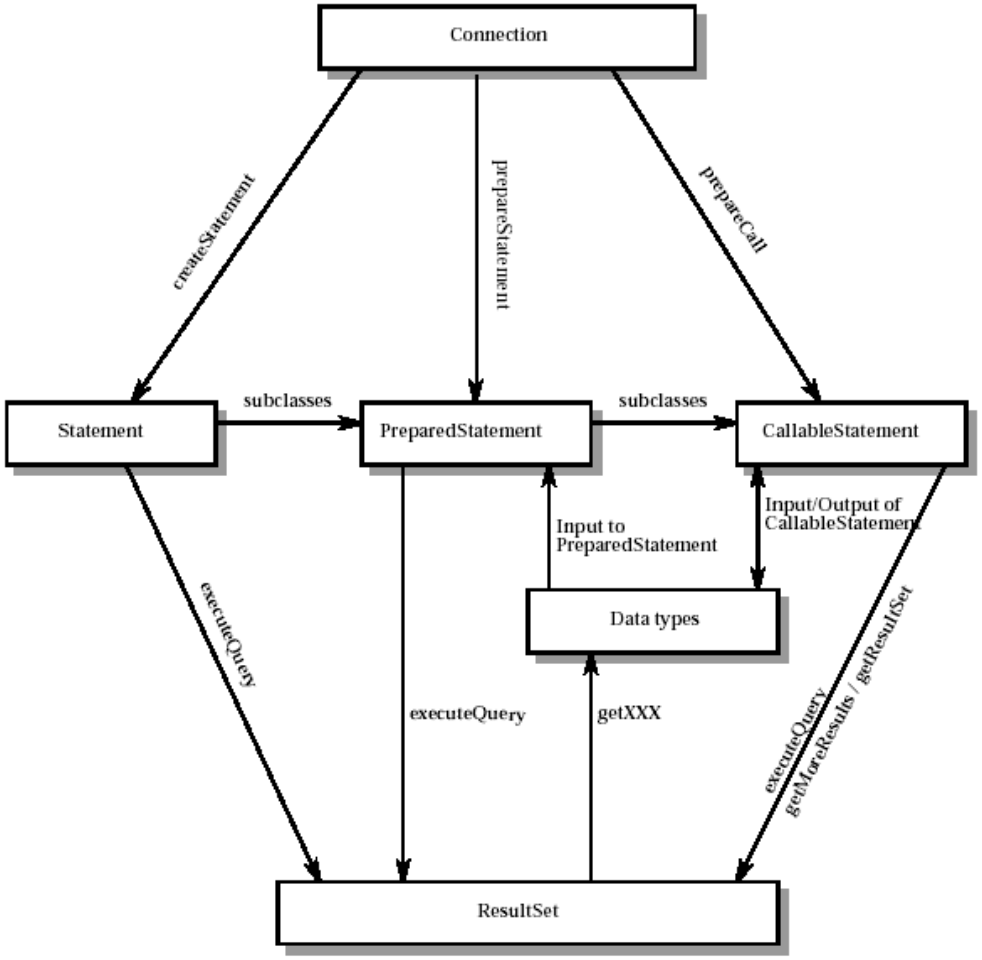


Estabilishing a JDBC connection

- Connection Pooling
  - ConnectionPoolDataSource interface
    - 1:X connection
    - Using logical connection inseatad of physical one
    - It is transpartent for the client
    - In most cases this is provided by the application server


JDBC Application
DataSource API
logical Connection object
Application Server
Cache of PooledConnection objects
ConnectionPoolDataSource API
physical PooledConnection object
JDBC Driver

JDBC objects



Statement

- Simple expressions without parameters
- String based SQL expression
- executeQuery (
  - Simple query Select * from t
- executeUpdate
  - INSERT
  - UPDATE
  - DELETE
  - CREATE TABLE
  - DROP TABLE
  - The return value is an intereger giving the number of affected rows
- execute
  - It is applied when more than one answer is expected

```
Connection con = DriverManager.getConnection(url, "sunny", "");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table2");
```

Prepared Statement

- Is a sublclass of the Statement class
- Precompiled SQL expressions
- One or more parameters (IN)
- One can use multiple methods for setting the IN parameteres
- It is more efficient than the simple statement (it is precomplied)
- It is shuold be used in the case of freqeuently applied queries
- It could run multiple times, the parameteres are retained

Callable Statement

- We can use it for executing stored procedures on server side.
- supportsStoredProcedures()
- getProcedures()
- {? = call procedure_name[(?, ?, ...)]}
- IN parameters
- OUT parameters
  - It should be registered
  - There not other way for handling the data
- INOUT parameteres

Result Set

- The result of the prevous three statements
- In default scenarion it is not writeable and it can be iterated only once
- The JDBC 2.0 API enables this
- The capabiliyt depends on the driver (eg.: postgresql)
- getXXX(name or serial) method (select a, select *)
- getMetaData
- updateRow(), insertRow(), deleteRow(), refreshRow()
- JDBC 2.0
  - previous
  - first
  - last
  - absolute
  - relative
  - afterLast
  - beforeFirst

Result set (JDBC 3.0)

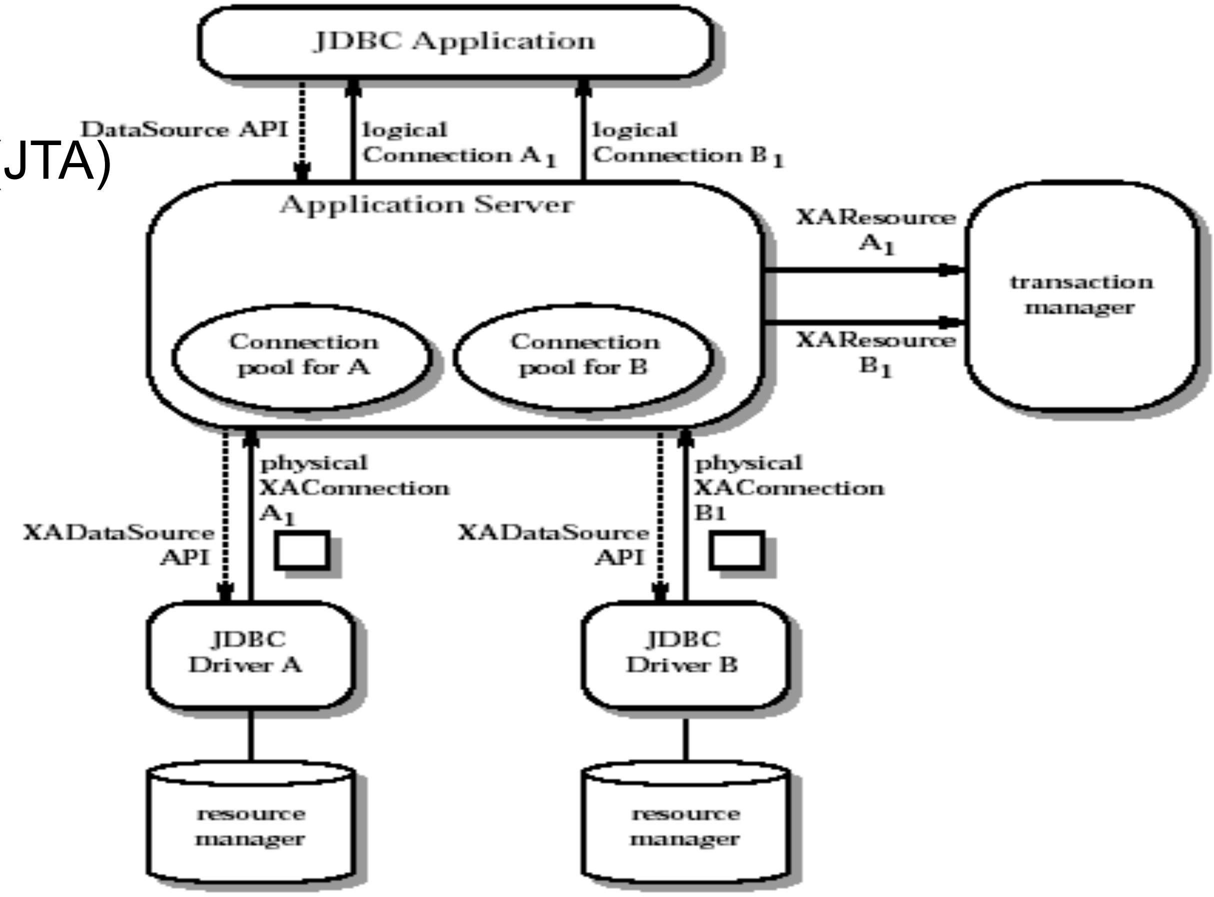
- Cursor:
  - TYPE_FORWARD_ONLY
  - TYPE_SCROLL_INSENSITIVE
  - TYPE_SCROLL_SENSITIVE
- Paralell issues
  - CONCUR_READ_ONLY
  - CONCUR_UPDATABLE
- Holding:
  - HOLD_CURSORS_OVER_COMMIT
  - CLOSE_CURSORS_OVER_COMMIT
- Example:

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt = conn.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY,
ResultSet.CLOSE_CURSORS_AT_COMMIT);
ResultSet rs = stmt.executeQuery(“select author, title, isbn from
    booklist”);
```

```
con.setAutoCommit( false );
  bError = false;
    try
    {
       for( ... )
      {
         if( bError )
        {
          break;
        }
        stmt.executeUpdate( ... );
      }
      if( bError )  { con.rollback(); }
      else
      { con.commit(); }
    } /
    catch ( SQLException SQLe)
    { con.rollback();
      ... } // end catch
    catch ( Exception e)
    { con.rollback();
      ... } // end catch
```

Distribueted transactions

- Transaction manager(JTA)
- JDBC driver:
  - XADataSource
  - XAConnection
  - XAResource
- Application server



Isseues with JDBC

- ORM
- Caching
- Business logic
- Transactions

ORM questions?

- How should we store a persisted object?
- How can we describe the mapping metadata?
- How can we persist the inheritance hierarchy?
- ORM vs. Business logic?
- What is the life cycle of the object?
- Beyond the ORM what level of aggregation, query is supported?
- How to handle the assiociations?
- Transactions?
- Caching?

ORM vs RDBM

- Collections
- Identity
- Inheritance
- Navigation

ORM benefits

- Development cycle (CRUD - @Entity + @Id)
- Maintainability
- Speed
- Vendor independence

Java Persistence API

- Java Community Program: JSR 317
- Handling the entities (in the past EJB 2.x < Entity Bean)
  - Lightweight persistent domain objects
  - Inheritance, polymorfic behaviour

Entity

- @Entity + @Id
- @Embeddable
- Field vs. Property access
  - Side effects (Field+)
  - Access(FIELD/PROPERTY)
- Table @Table
  - Multiple tables @SecondaryTable
  - View
  - Replication
  - History

Identity

- Refernce vs. Databased primary key
- It is not recommended to use database based key
- It is recommended to use database based key
- Many diiferenct ID geneartion solution
- @Id, @EmbeddedId, @IdClass
- Inherited identity

```
@Entity
public class Employee {
  @Id long empId;
  String empName;
  ...
}
```

```
public class DependentId {
   String name;  // matches name of @Id attribute
   long emp; // matches name of @Id attribute and type of Employee PK
}

@Entity
@IdClass(DependentId.class)
public class Dependent {
  @Id String name;

  // id attribute mapped by join column default
  @Id @ManyToOne Employee emp;
  ...
}
```

```
@Embeddable
public class DependentId {
  String name;
  long empPK;   // corresponds to PK type of Employee
}

@Entity
public class Dependent {
  @EmbeddedId DependentId id;
  ...
  // id attribute mapped by join column default
  @MapsId("empPK")  //  maps empPK attribute of embedded id
  @ManyToOne Employee emp;
}
```

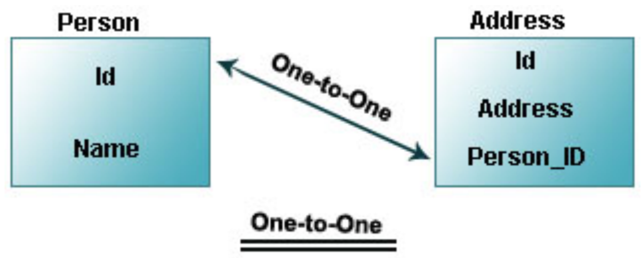
UNIVERSITAS SCIENTIARUM SZEGEDIENSIS
UNIVERSITY OF SZEGED
Department of Software Engineering

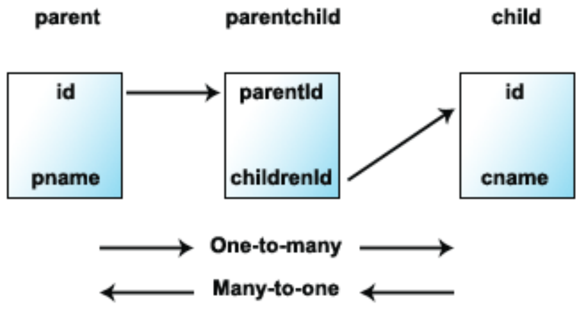
Collections

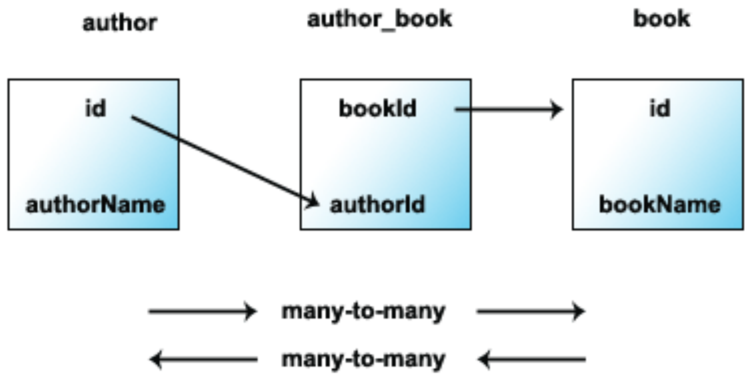
- Java.util.Map
- Basic type, embedded (hashCode, equelas)
  - @ElementCollection
  - @MapKeyColumn
- @OneToMany, @ManyToMany
  - @MapKeyJoinColumn

Associations

- Association(one-way/two-way)
  - @OneToOne (JoinTable, mappedBy)
  - @OneToMany (JoinColumn, JoinTable, mappedBy)
  - @ManyToOne (JoinTable)
  - @ManyToMany (JoinTable, mappedBy)
- Cascade (remove)
- orphanRemoval (null)







One - to - one two way

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

   @OneToOne
   public Cubicle getAssignedCubicle() {
     return assignedCubicle;
   }
   public void setAssignedCubicle(Cubicle cub
     this.assignedCubicle = cubicle;
   }
  ...
}
```

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

    @OneToOne(mappedBy="assignedCubicle")
    public Employee getResidentEmployee() {
      return residentEmployee;
    }
    public void setResidentEmployee(Employee employee) {
      this.residentEmployee = employee;
    }
  ...
}
```

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `Cubicle` is mapped to a table named `CUBICLE`.

Table `EMPLOYEE` contains a foreign key to table `CUBICLE`. The foreign key column is named `ASSIGNEDCUBICLE_`<PK of CUBICLE>, where <PK of CUBICLE> denotes the name of the primary key column of table `CUBICLE`. The foreign key column has the same type as the primary key of `CUBICLE`, and there is a unique key constraint on it.

UNIVERSITAS SCIENTIARUM SZEGEDIENSIS
UNIVERSITY OF SZEGED
Department of Software Engineering

One – to – one one way

```
@Entity
public class Employee {
   private TravelProfile profile;

   @OneToOne
   public TravelProfile getProfile() {
     return profile;
   }
   public void setProfile(TravelProfile profile) {
     this.profile = profile;
   }
  ...
}
```

```
@Entity
public class TravelProfile {
    ...
}
```

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `TravelProfile` is mapped to a table named `TRAVELPROFILE`.

Table `EMPLOYEE` contains a foreign key to table `TRAVELPROFILE`. The foreign key column is named `PROFILE_`<PK of TRAVELPROFILE>, where <PK of TRAVELPROFILE> denotes the name of the primary key column of table `TRAVELPROFILE`. The foreign key column has the same type as the primary key of `TRAVELPROFILE`, and there is a unique key constraint on it.

One – to - many two way

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

   @ManyToOne
   public Department getDepartment() {
     return department;
   }
   public void setDepartment(Department department) {
     this.department = department;
   }
  ...
}


@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;
   }
  ...
}
```

Entity `Employee` is mapped to a table named `EMPLOYEE`.

Entity `Department` is mapped to a table named `DEPARTMENT`.

Table `EMPLOYEE` contains a foreign key to table `DEPARTMENT`. The foreign key column is named `DEPARTMENT_`<PK of DEPARTMENT>, where <PK of DEPARTMENT> denotes the name of the primary key column of table `DEPARTMENT`. The foreign key column has the same type as the primary key of `DEPARTMENT`.

Many – to - many two way

```
@Entity
public class Project {
   private Collection<Employee> employees;

   @ManyToMany
   public Collection<Employee> getEmployees() {
     return employees;
   }

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

@Entity
public class Employee {
   private Collection<Project> projects;

   @ManyToMany(mappedBy="employees")
   public Collection<Project> getProjects() {
     return projects;
   }

   public void setProjects(Collection<Project> projects) {
     this.projects = projects;
   }
  ...
}
```

Entity `Project` is mapped to a table named `PROJECT`.

Entity `Employee` is mapped to a table named `EMPLOYEE`.

There is a join table that is named `PROJECT_EMPLOYEE` (owner name first). This join table has two foreign key columns. One foreign key column refers to table `PROJECT` and has the same type as the primary key of `PROJECT`. The name of this foreign key column is `PROJECTS_`<PK of PROJECT>, where <PK of PROJECT> denotes the name of the primary key column of table `PROJECT`. The other foreign key column refers to table `EMPLOYEE` and has the same type as the primary key of `EMPLOYEE`. The name of this foreign key column is `EMPLOYEES_`<PK of EMPLOYEE>, where <PK of EMPLOYEE> denotes the name of the primary key column of table `EMPLOYEE`.

Inheritance

- The root class shoud be decorated in order to specify the persistence strategy
  - One table per class hierarchy
    - Special column for identification
  - One table for each subclass
    - Weak polymorfic behaviour (Union)
  - Join strategy only the own data properties are stored in a table
    - Slow (many JOIN)

Cascade operations

- CascadeType.PERSIST : means that save() or persist() operations cascade to related entities.
- CascadeType.MERGE : means that related entities are merged into managed state when the owning entity is merged.
- CascadeType.REFRESH : does the same thing for the refresh() operation.
- CascadeType.REMOVE : removes all related entities association with this setting when the owning entity is deleted.
- CascadeType.DETACH : detaches all related entities if a “manual detach” occurs.
- CascadeType.ALL : is shorthand for all of the above cascade operations.

Entity manager

- The life cycle of the entities is managed by the EM
  - Handled by the container
    - Automatic propagation
      - Transaction lifetime
      - Extenede
  - Handled by the application
    - The transaction should be handled dierctly
- Not threadsafe
- Few important methods
  - persist()
  - find()
  - update()
  - remove()
  - merge()
  - detach()

```
@PersistenceContext
EntityManager em;
```

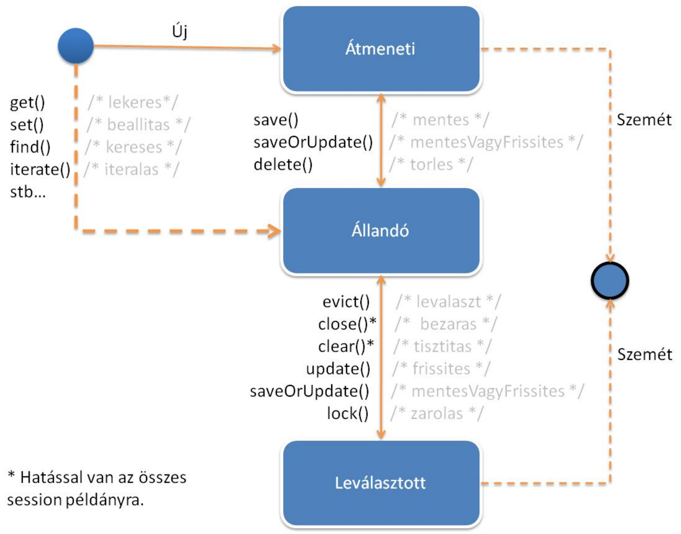
```
@PersistenceUnit
EntityManagerFactory emf;
```

```
@Stateless public class OrderEntryBean implements OrderEntry {

  @PersistenceContext EntityManager em;

  public void enterOrder(int custID, Order newOrder) {
     Customer cust = em.find(Customer.class, custID);
     cust.getOrders().add(newOrder);
     newOrder.setCustomer(cust);
     em.persist(newOrder);
   }
}
```

UNIVERSITAS SCIENTIARUM SZEGEDIENSIS
UNIVERSITY OF SZEGED
Department of Software Engineering

States of the Entity

- New (transient): an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- Managed (persistent): a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- Detached: the entity instance is an instance with a persistent identity that is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- Removed: a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.



Lifecycle

- Database vs. Cache vs. Memory
  - load()
- Creating an entity
  - save()/saveOrUpdate()
- Removing an entity
- Synchronization
  - merge(), refresh()
- Refreshing
- Detached entities
  - evict()
- Managed entities
- Loading a state

Synchronization

- At the end of the transaction the persistence contetxt is going to be synchronized
- Other persistence context could also be accessed: EntityManager joinTransaction
- The enitity itself could be synchronized with refresh
- merge()
- detach()

Navigation

- Navigation?
  - Java exact
    - Object graf admittance x.d.g.getZ();
  - SQL arbitrary:
    - N+1 select issue
    - We should minimalize the number of queries – join
    - We should know beforehead what we would like to query
      - User
      - User join Billing details

```
select * from USER u where u.USER_ID = 123
```

```
select *
from USER u
left outer join BILLING_DETAILS bd on bd.USER_ID = u.USER_ID
where u.USER_ID = 123
```

Loading

- Lazy
  - Only proxy
- Batch
  - More than one proxies are resolved
- Eager
  - Loads all

Concurency

- In the case of concurrent access the data is accessed from multiple threads simultaneously
- Optimistic
  - Optimistic locking assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back[1] (Hibernate provides two different mechanisms for storing versioning information, a dedicated version number or a timestamp.
- Pessimistic
  - Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

Locking levels

- LockMode.WRITE acquired automatically when Hibernate updates or inserts a row.
- LockMode.UPGRADE acquired upon explicit user request using SELECT ... FOR UPDATE on databases which support that syntax.
- LockMode.UPGRADE_NOWAIT acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAIT in Oracle.
- LockMode.READ acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. It can be re-acquired by explicit user request.
- LockMode.NONE The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start out in this lock mode.

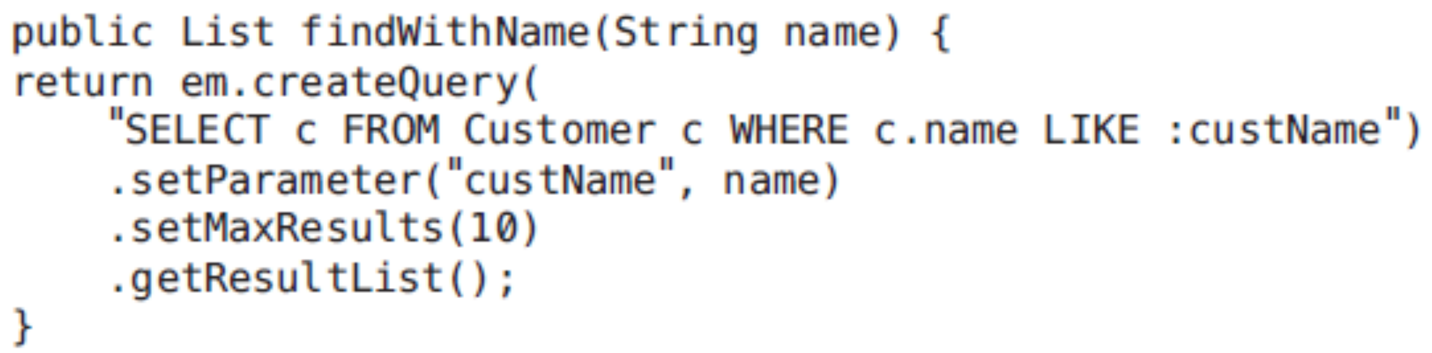
Bean validation

- Not only JPA capability
- It is part of the WEB and EJB containers
- Rules could be atached to fields and properties
- Javax.validation.constrains – extendable
- Time+ sample+ value

```
@Entity
public class Contact implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NotNull
    protected String firstName;
    @NotNull
    protected String lastName;
    @Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
        +"[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
        +"(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
            message="{invalid.email}")
    protected String email;
    @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message="{invalid.phonenumber}")
    protected String mobilePhone;
    @Pattern(regexp="^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message="{invalid.phonenumber}")
    protected String homePhone;
    @Temporal(javax.persistence.TemporalType.DATE)
    @Past
    protected Date birthday;
    ...
}
```

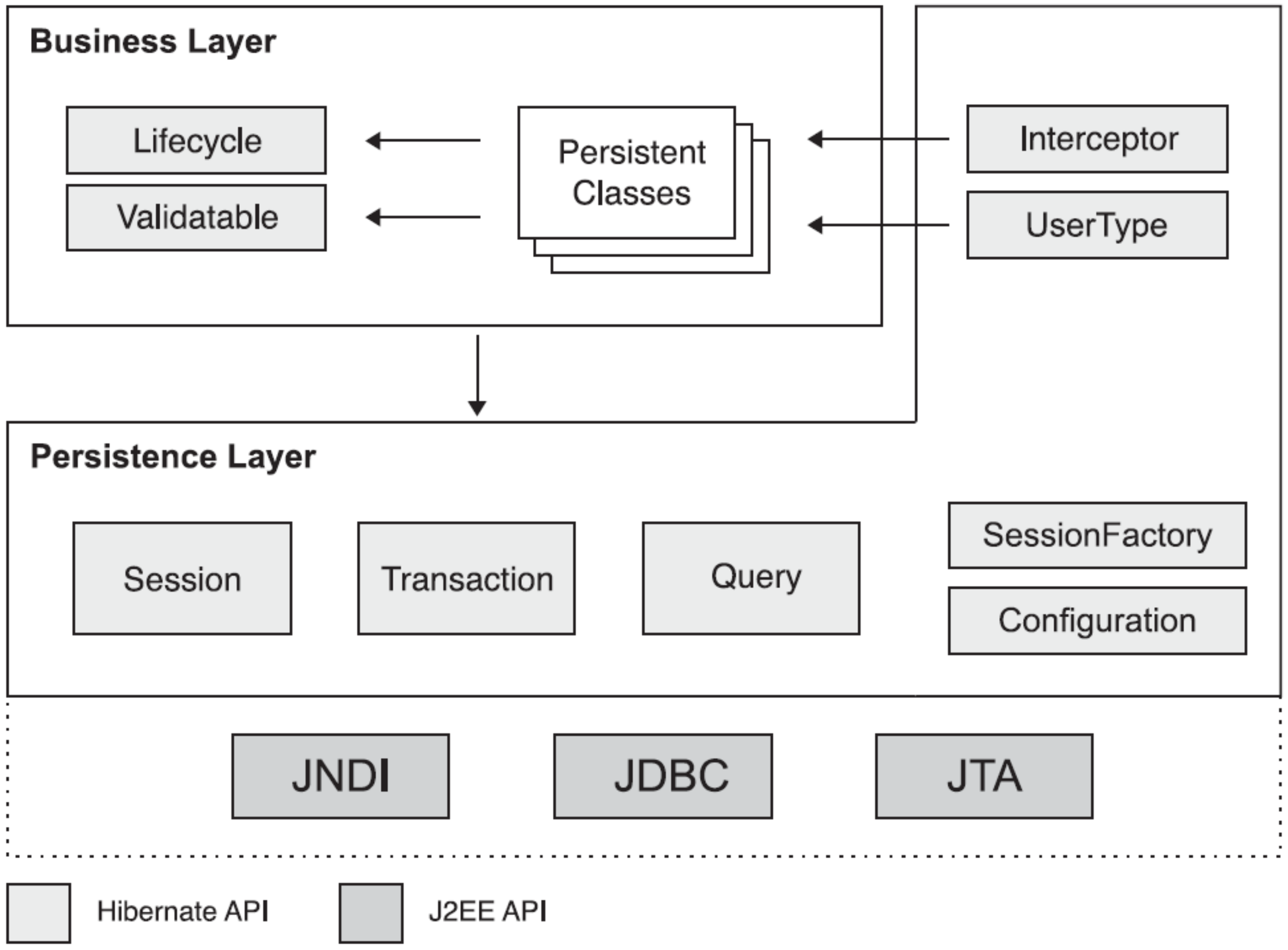
Query language

- Similar to the SQL but Object Oriented
- It enables the adressing of the nodes on oject graph



```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

Hibernate


Business Layer
Lifecycle
Validatable
Persistent Classes
Interceptor
UserType
Persistence Layer
Session
Transaction
Query
SessionFactory
Configuration
JNDI
JDBC
JTA
Hibernate API
J2EE API

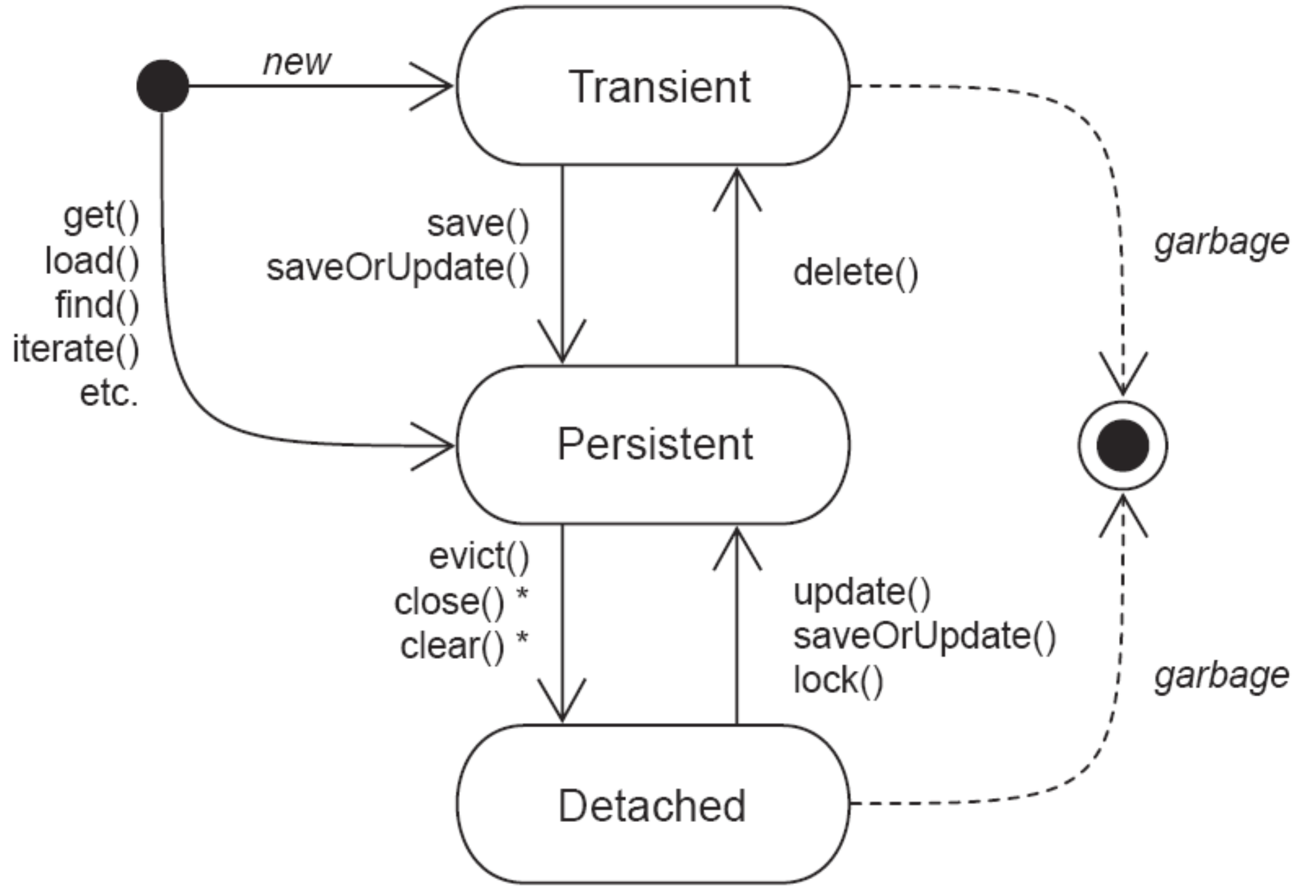
Basics:

- SessionFactory (org.hibernate.SessionFactory)
  - A threadsafe, immutable cache of compiled mappings for a single database. A factory for Session and a client of ConnectionProvider, SessionFactory can hold an optional (second-level) cache of data that is reusable between transactions at a process, or cluster, level.
- Session (org.hibernate.Session)
  - A single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps a JDBC connection and is a factory for Transaction. Session holds a mandatory first-level cache of persistent objects that are used when navigating the object graph or looking up objects by identifier.
- Persistent objects and collections
  - Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one Session. Once the Session is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation).
- Transient and detached objects and collections
  - Instances of persistent classes that are not currently associated with a Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed Session.

Basics:

- Transaction (org.hibernate.Transaction)
  - (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A Session might span several Transactions in some cases. However, transaction demarcation, either using the underlying API or Transaction, is never optional.
- ConnectionProvider (org.hibernate.connection.ConnectionProvider)
  - (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying Datasource or DriverManager. It is not exposed to application, but it can be extended and/or implemented by the developer.
- TransactionFactory (org.hibernate.TransactionFactory)
  - (Optional) A factory for Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.
- Extension Interfaces
  - Hibernate offers a range of optional extension interfaces you can implement to customize the behavior of your persistence layer. See the API documentation for details.

Persitence



* affects all instances in a Session

Persistence manager

- CRUD
- Query
- Transactions
- Cache

```
User user = new User();
user.getName().setFirstname("John");
user.getName().setLastname("Doe");
```

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

session.save(user);

tx.commit();
session.close();
```

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

int userID = 1234;
User user = (User) session.get(User.class, new Long(userID));

tx.commit();
session.close();
```

Loading objects

- By ID

```
User user = (User) session.get(User.class, userID);
```

- HQL

```
Query q = session.createQuery("from User u where u.firstname = :fname");
q.setString("fname", "Max");
List result = q.list();
```

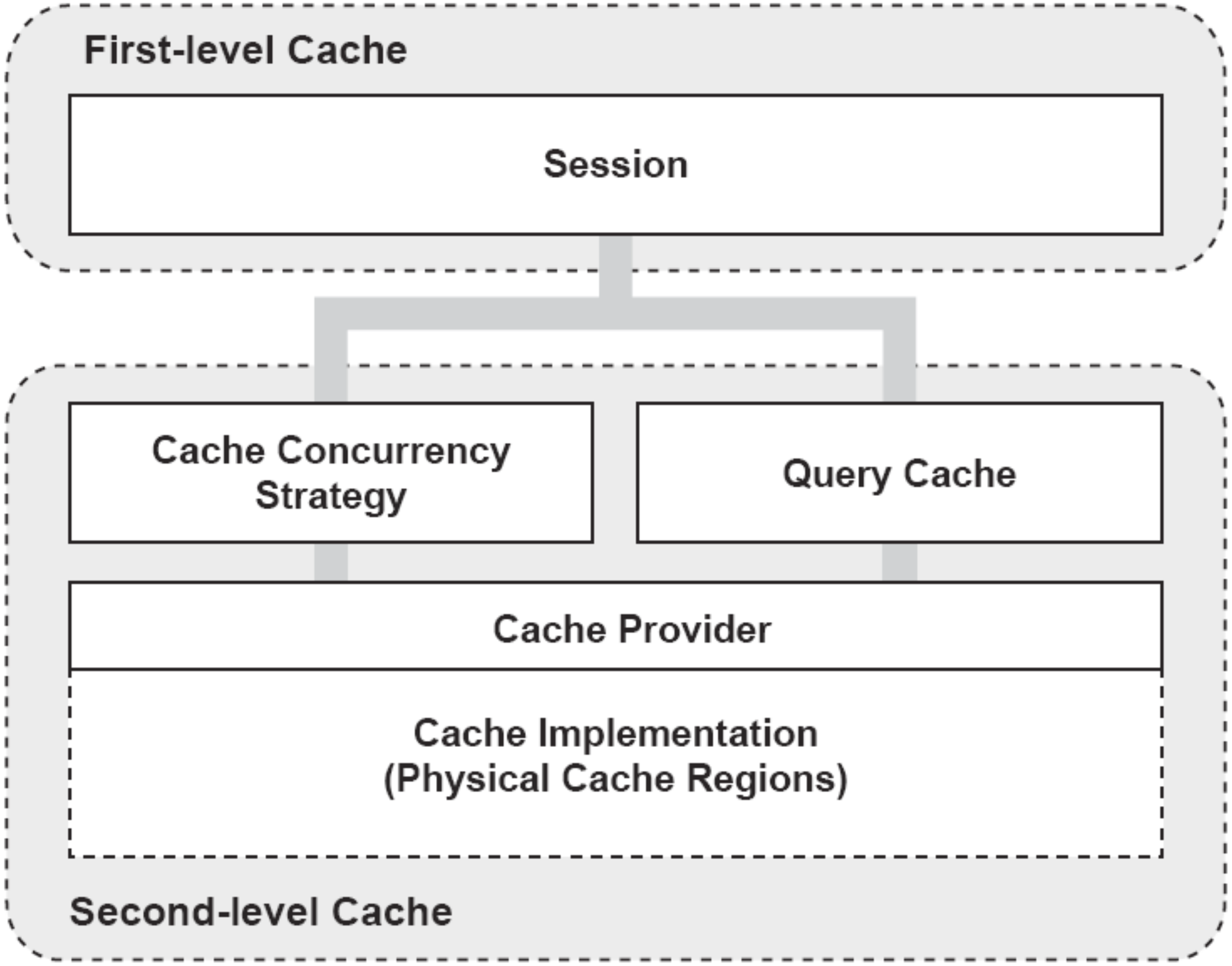
- Criteria query

```
Criteria criteria = session.createCriteria(User.class);
criteria.add( Expression.like("firstname", "Max") );
List result = criteria.list();
```

- By sample

```
User exampleUser = new User();
exampleUser.setFirstname("Max");
Criteria criteria = session.createCriteria(User.class);
criteria.add( Example.create(exampleUser) );
List result = criteria.list();
```

Architecture

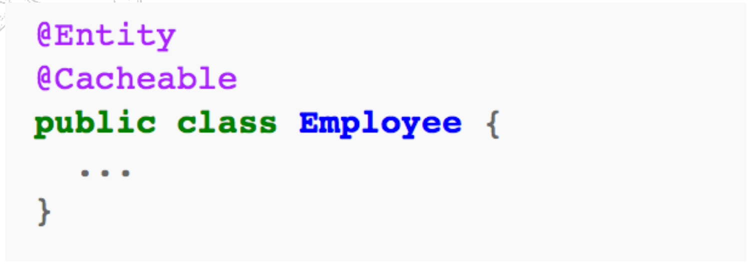

First-level Cache
Session
Cache Concurrency Strategy
Query Cache
Cache Provider
Cache Implementation (Physical Cache Regions)
Second-level Cache

First level cache

- First level cache is associated with "session" object and other session objects in application can not see it.
- The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
- First level cache is enabled by default and you can not disable it.
- When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
- If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
- The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
- The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

Second level cache

- Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
- If cached copy of entity is present in first level cache, it is returned as result of load method.
- If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
- If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
- If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
- Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
- If some user or process make changes directly in database, the there is no way that second level cache update itself until "timeToLiveSeconds" duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.



```
@Entity
@Cacheable
public class Employee {
    ...
}
```

Overview

- JDBC
- JPA
- Hibernate