JAVA BACKEND DEVELOPMENT PROGRAM

Hibernate

OUTLINE

- Introduction
    - JDBC & Terminologies
- Hibernate
    - Introduction & features
    - Configuration & Mapping
    - SessionFactory & Session
    - Transaction

JDBC

- Before we go to Hibernate, let’s review JDBC
  - What is JDBC?
  - How to use JDBC?

TERMINOLOGIES

- Persistence
- Connection Pool

PERSISTENCE

- Persistence — The process of storing data to permanent place and retrieving data from permanent place.

- Persistent logic — the required logic to add, remove, read and modify.

- Persistent store — the place where data will be stored permanently.

CONNECTION POOL

- As we know, we have to open connection to a database whenever we are using JDBC.
- But database connections are fairly expensive operations, and as such, should be reduced to a minimum in every possible use case.
- Here is where connection pool comes into the play

CONNECTION POOL

- Connection pooling is a well-known data access pattern, whose main purpose is to reduce the overhead involved in performing database connections and read/write database operations.

- By just simply implementing a database connection container, which allows us to reuse a number of existing connections, we can effectively save the cost of performing a huge number of expensive database trips, hence boosting the overall performance of our database-driven applications.

- It is very hard to set up connection pool with JDBC.

CONNECTION POOL

- Configuration:
  - max pool size
  - max idle size
  - min idle size
  - idle wait time

DRAWBACKS OF JDBC

- JDBC used SQL quires to implement persistence logic. JDBC based persistence logic is becomes database dependent.
- Change of database software becomes complex and disturbs persistence logic.
- Programmer is responsible to take about exception handing and transaction management.
- ResultSet Object is not serializable object, we cannot send this object over the network.
- We need to write additional code to have connection pooling.

OUTLINE

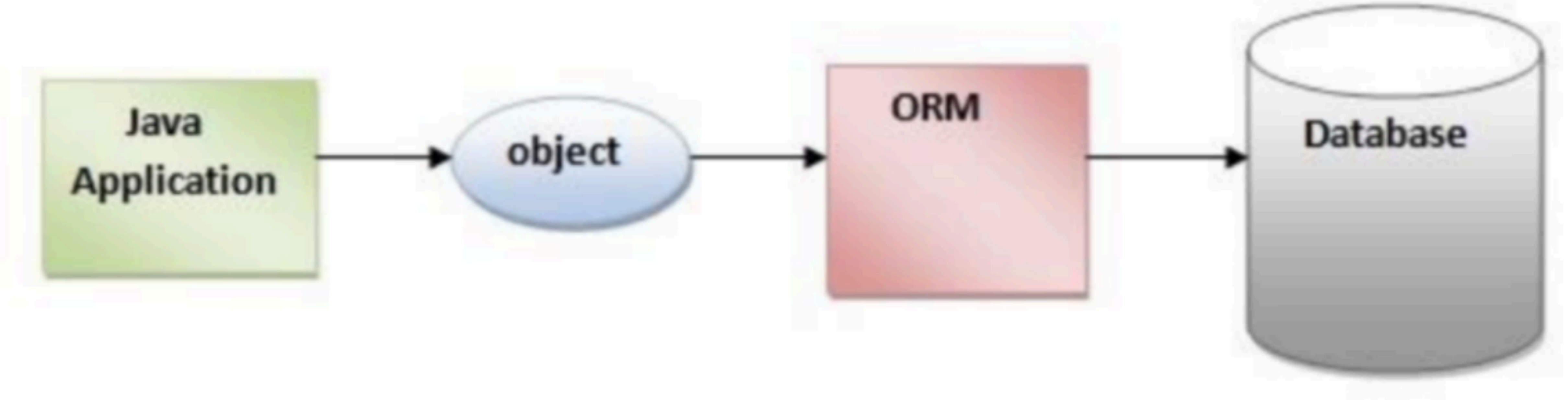
- Introduction
    - JDBC & Terminologies
- Hibernate
    - Introduction & features
    - Configuration & Mapping
    - SessionFactory & Session
    - Transaction

HIBERNATE

- Hibernate is an open source, light weight ORM tool to develop DB independent persistence logic in java based enterprise application.
  - ORM — Object - Relational mapping
    - ORM framework eases to store the data from object instances into persistence data store and load that data back into the same object structure
  - This gives developers a way to map the object structures in Java classes to relational database tables.

ORM

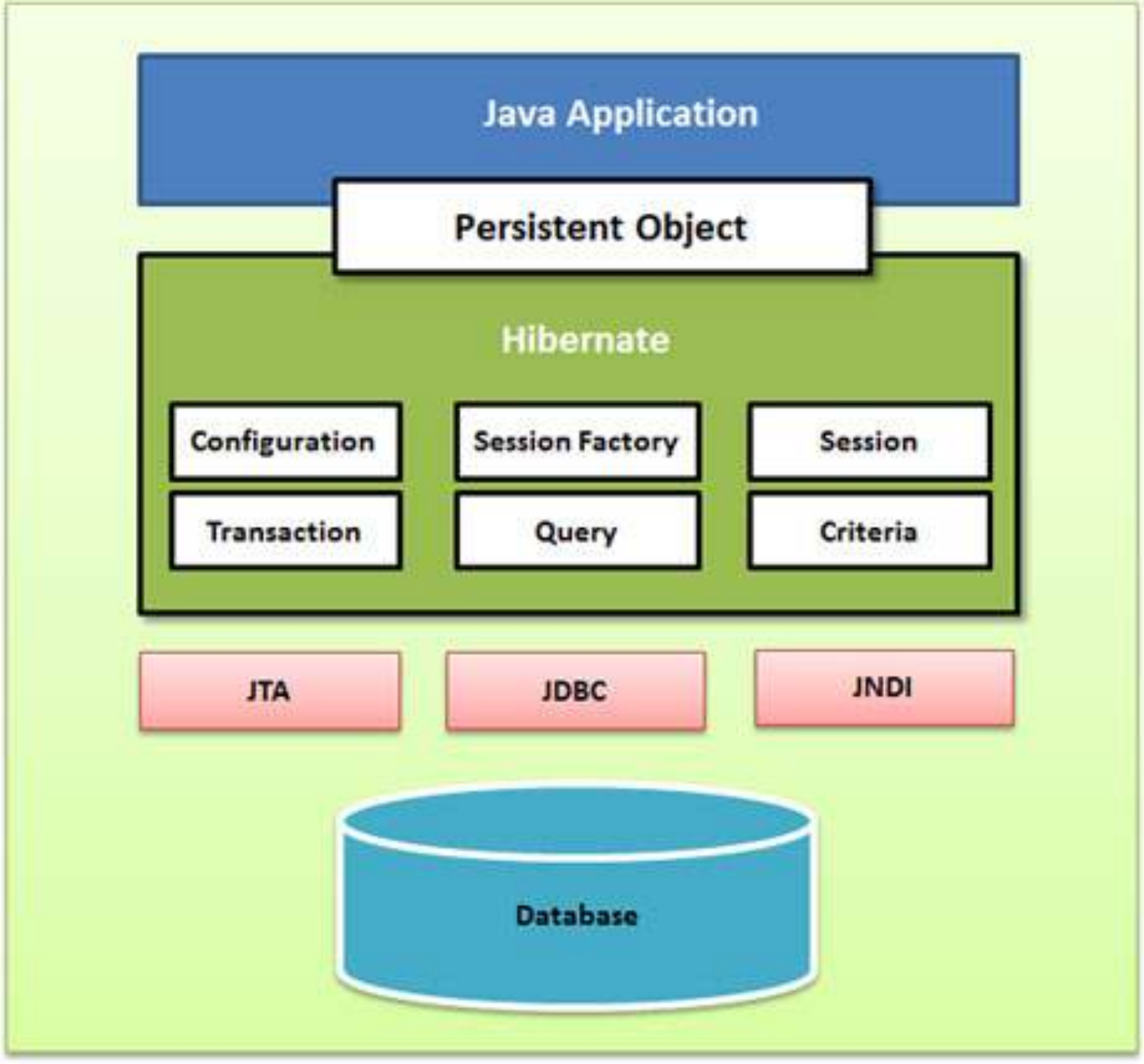
- The process of mapping java class with database table, java class members with database table columns



HIBERNATE FEATURES

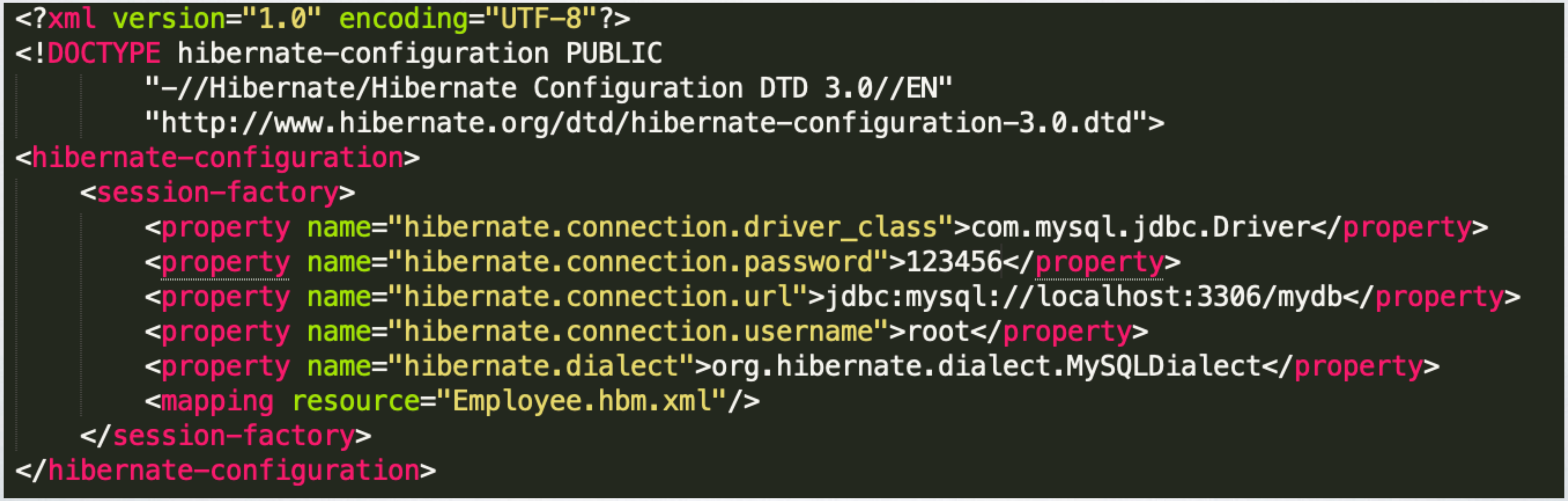
- O-R mapping using ordinary Java class
- Database independent persistence logic and mapping style.
- Pluggable with any Java/J2EE based frameworks.
- object-oriented query language
- It provides APIs for storing and retrieving objects directly to and from the database.
- Transaction management with rollback

HIBERNATE STRUCTURE



HIBERNATE CONFIGURATION

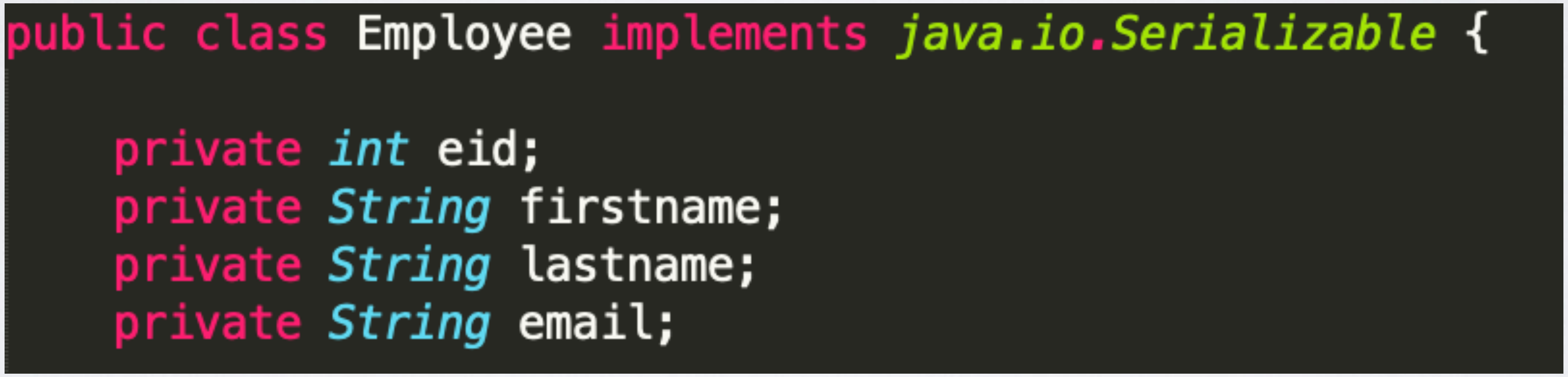
- Hibernate needs to know where it can look for mapping between Java classes and relational database tables.
- Along with this mapping, Hibernate needs some database configuration settings and parameters.This information is provided through *hibernate.cfg.xml*.



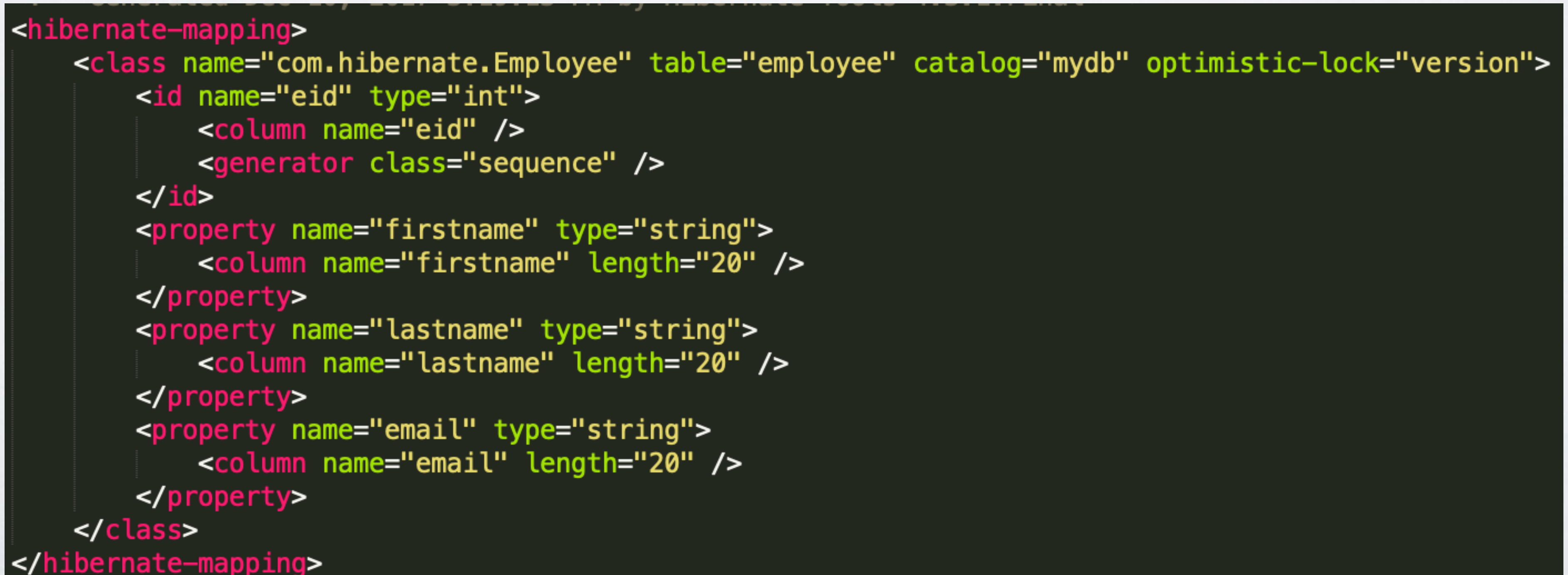
```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password">123456</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydb</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <mapping resource="Employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

HIBERNATE MAPPING

- Hibernate provides a way to map Java objects to relational database tables through an XML file. This mapping file tells hibernate how to map the defined class or classes to the database table.



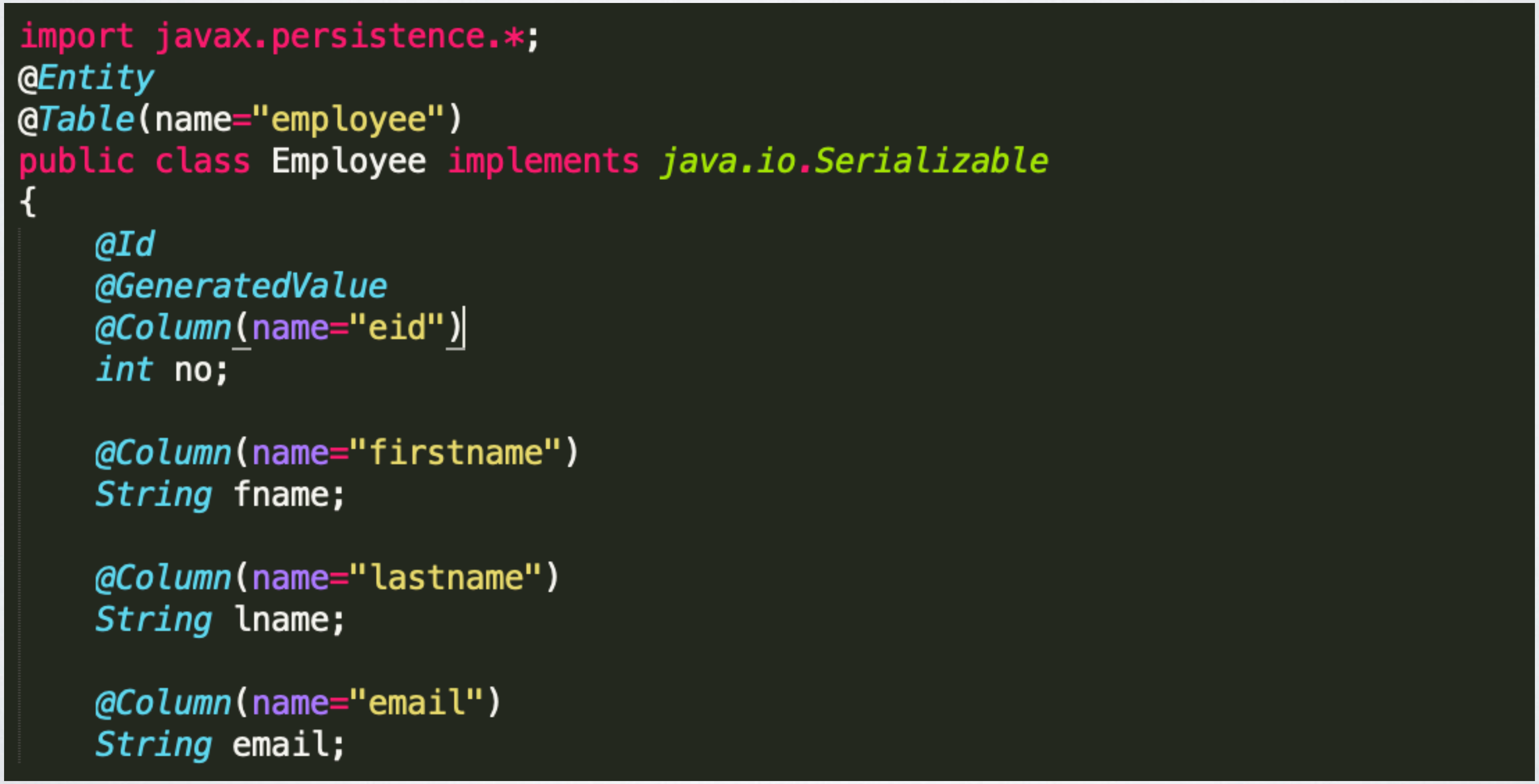
```java
public class Employee implements java.io.Serializable {

    private int eid;
    private String firstname;
    private String lastname;
    private String email;
```



```xml
<hibernate-mapping>
    <class name="com.hibernate.Employee" table="employee" catalog="mydb" optimistic-lock="version">
        <id name="eid" type="int">
            <column name="eid" />
            <generator class="sequence" />
        </id>
        <property name="firstname" type="string">
            <column name="firstname" length="20" />
        </property>
        <property name="lastname" type="string">
            <column name="lastname" length="20" />
        </property>
        <property name="email" type="string">
            <column name="email" length="20" />
        </property>
    </class>
</hibernate-mapping>
```

HIBERNATE ANNOTATION

- Instead of XML configuration, there is an alternative way to configure Hibernate Mapping by Java Annotations.



```
import javax.persistence.*;
@Entity
@Table(name="employee")
public class Employee implements java.io.Serializable
{
    @Id
    @GeneratedValue
    @Column(name="eid")
    int no;

    @Column(name="firstname")
    String fname;

    @Column(name="lastname")
    String lname;

    @Column(name="email")
    String email;
```

HIBERNATE ANNOTATION

- JPA — Java Persistence API
  - JPA entities are plain POJOs. (Plain Old Java Object — not bound by any special restriction)
  - Their mappings are defined through JDK 5.0 annotations instead of hbm.xml files
  - JPA annotations are in the *javax.persistence.** package
  - https://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/

HIBERNATE ANNOTATION

- @Entity declares the class as an entity (i.e. a persistent POJO class)

- @Table is set at the class level; it allows you to define the table, catalog, and schema names for your entity mapping. If no @Table is defined the default values are used: the unqualified class name of the entity.

- @Id declares the identifier property of this entity.

- @GeneratedValue annotation is used to specify the primary key generation strategy to use. If the strategy is not specified by default AUTO will be used.

- @Column annotation is used to specify the details of the column to which a field or property will be mapped. If the @Column annotation is not specified by default the property name will be used as the column name.

ENTITY MANAGEMENT

- Apart from object-relational mapping itself, one of the problems that Hibernate was intended to solve is the problem of managing entities during runtime.

- The notion of "persistence context" is Hibernate's solution to this problem.

- Persistence context can be thought of as a container or a first-level cache for all the objects that you loaded or saved to a database - **SESSION**.

SESSION

- The session object provides an interface between the application and data stored in the database.
  - A Session is a light weight and a non-threadsafe object that represents a single unit-of-work with the database.
  - It is a short-lived object and wraps the JDBC connection. It is factory ofTransaction, Query and Criteria.

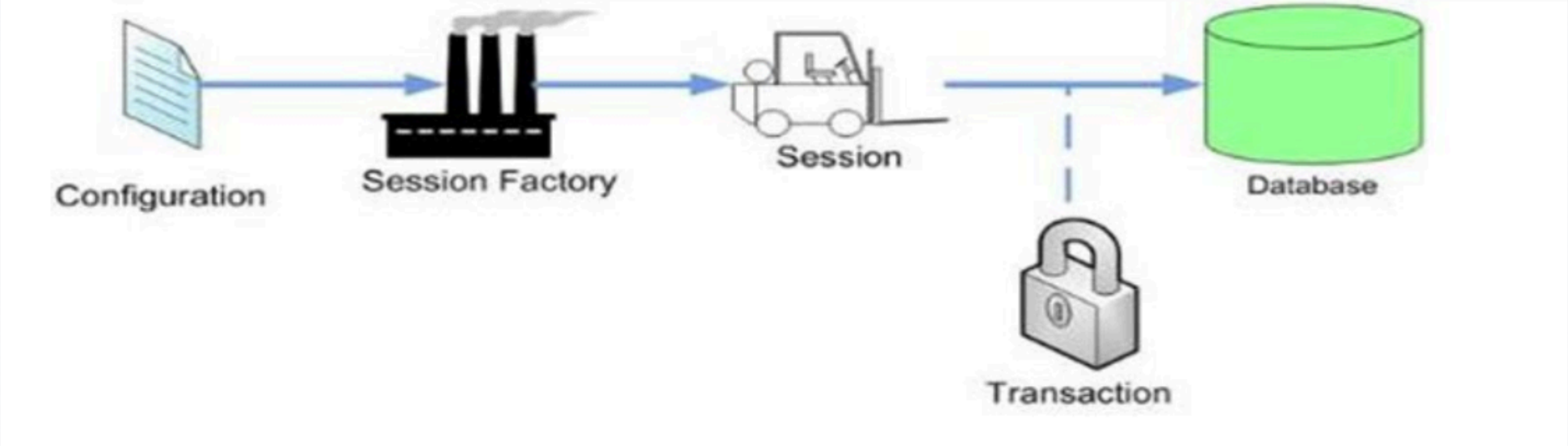
SESSION

- Hibernate provide us a collection of APIs to manage the transaction with in each session
  - void begin() — starts a new transaction.
  - void commit() — ends the unit of work unless we are in FlushMode.NEVER.
  - void rollback() — forces this transaction to rollback.
  - void setTimeout(int seconds) — it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
  - boolean isAlive() — checks if the transaction is still alive.
  - boolean wasCommited() — checks if the transaction is committed successfully.
  - boolean wasRolledBack() — checks if the transaction is rolled back successfully.

SESSION FACTORY

- SessionFactory is Hibernate’s concept of a single datastore and is thread-safe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database.

HIBERNATE SESSION & SESSION FACTORY



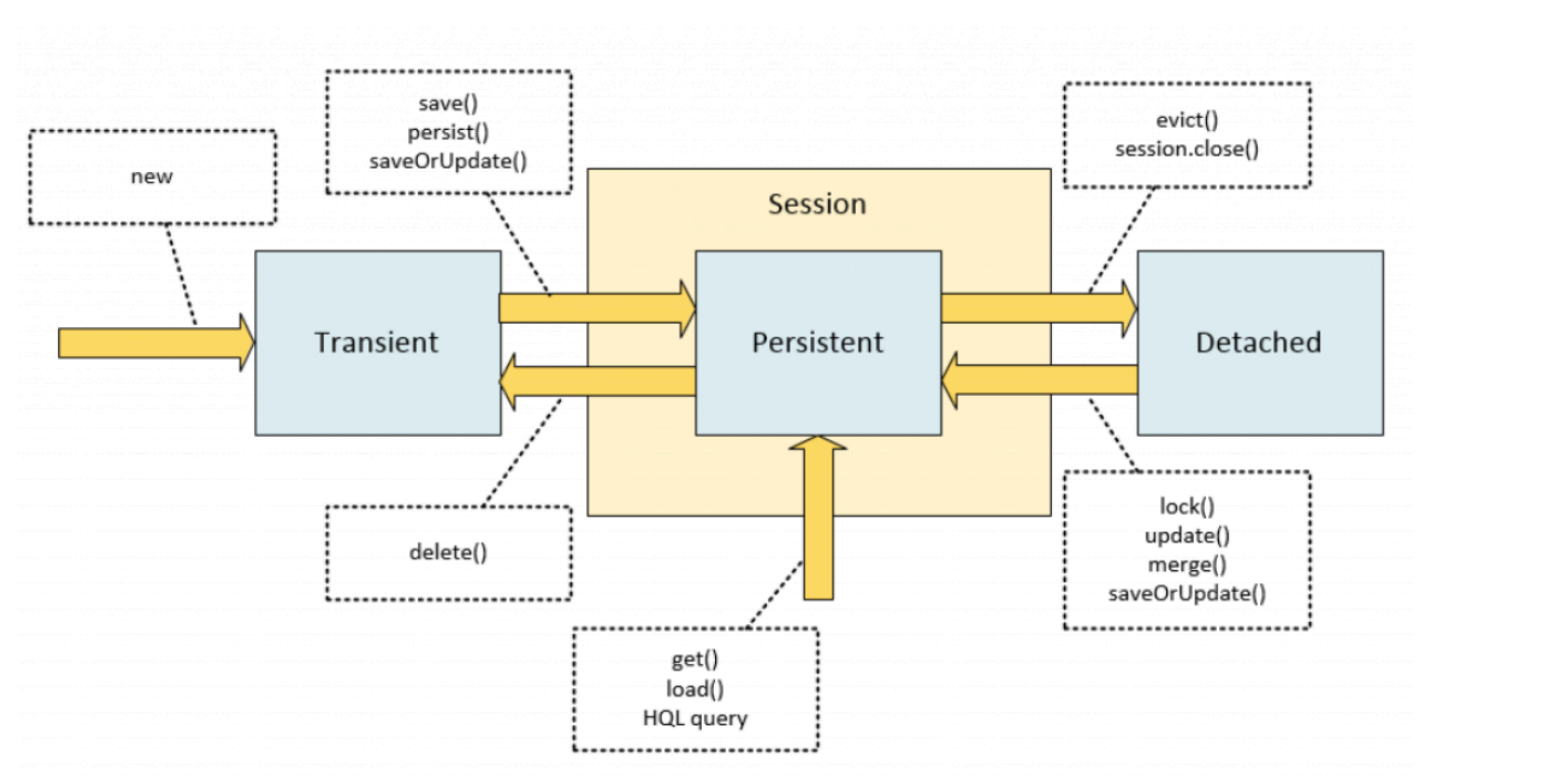
HIBERNATE SESSION

- A Session object can be obtained from SessionFactory in two ways:
  - getCurrentSession()
    - Creates a new Session if not exists, else use the same session which is in current hibernate context. It automatically flush and close the Session.
  - openSession()
    - Create a new Session and give it to you, need to explicitly flush and close the Session.

ENTITY MANAGEMENT

- Any entity instance in your application appears in one of the three main states in relation to the *Session* persistence context:
  - *transient* — this instance is not, and never was, attached to a *Session*; this instance has no corresponding rows in the database; it's usually just a new object that you have created to save to the database;
  - *persistent* — this instance is associated with a unique *Session* object; upon flushing the *Session* to the database, this entity is guaranteed to have a corresponding consistent record in the database;
  - *detached* — this instance was once attached to a *Session* (in a *persistent* state), but now it's not; an instance enters this state if you evict it from the context, clear or close the Session, or put the instance through serialization/deserialization process.

STATES OF ENTITY INSTANCES



STATES OF ENTITY INSTANCES

- When the entity instance is in the *persistent* state, all changes that you make to the ***mapped*** fields of this instance will be applied to the corresponding database records.

- This means that when you change fields of a *persistent* object, you *don*'t have to call *save*, *update* or any of those methods to get these changes to the database: all you need is to ***commit*** the transaction or ***close*** the session, when you're done with it.

- The *persistent* instance can be thought of as "**online**", whereas the *detached* instance has gone "**offline**" and is not monitored for changes.

SESSION INTERFACE

- To Persistent State:
  - Get
  - Load
  - Save
  - Persist
  - Update
  - Merge
  - saveOrUpdate
- Note:These methods do not immediately result in the corresponding SQL *UPDATE* or *INSERT* statements. The actual saving of data to the database occurs on committing the transaction or flushing the *Session*.

GET

- Used to fetch data from the database for a given identifier
- Return null if no object can be found using given identifier
- Eager loading - Return a fully initialized object
- Slower performance

LOAD

- Also used to fetch data from the database for a given identifier
- Throw exception if not object can be found using the given identifier
- Lazy Loading - Return proxy object
- Slightly faster performance

SAVE

- The method strictly states that it persists the instance, "first assigning a generated identifier".
  - The method is guaranteed to return the *Serializable* value of this identifier.
- The method has a return type of Serializable
- The reference of the passed in object pointing to the persisted object.
- Note: it does not conform to the JPA specification.

PERSIST

- The *persist* method is intended for adding a new entity instance to the persistence context,
  - i.e. transitioning an instance from transient to *persistent* state.
  - We usually call it when we want to add a record to the database (persist an entity instance)
- The *persist* method has *void* return type. It operates on the passed object "in place", changing its state.
  - The object passed in now actually pointing to the persisted object
- Note: This method does NOT guarantee that the id of the object will be generated after calling the method. It follows JPA specification.

UPDATE

- It acts almost same as Save and Persist method, with small different:
  - It acts upon passed object (its return type is *void*)
  - The *update* method transitions the passed object from *detached* to *persistent* state
  - This method throws an exception if you pass it a *transient* entity
- Note: it does not conform to the JPA specification.

MERGE

- The main intention of the *merge* method is to update a *persistent* entity instance with new field values from a *detached* entity instance
  - Suppose we have a RESTful interface with a method for retrieving an JSON-serialized object by its id to the caller and a method that receives an updated version of this object from the caller.
- An entity that passed through such serialization/deserialization will appear in a *detached* state. So the *merge* method does exactly that:
  - Finds an entity instance by id taken from the passed object
  - Copies fields from the passed object to this instance
  - Returns newly updated instance
- The return type of the method is an Object — It is the object loaded into the persistent state and updated, not the object passed as the argument.
- Note: It follows JPA specification.

SAVEORUPDATE

- Similar to *update*, it also may be used for reattaching instances

- The main difference of *saveOrUpdate* method is that it does not throw exception when applied to a *transient* instance; instead, it makes this *transient* instance *persistent*.

- Note: it does not conform to the JPA specification.

SESSION INTERFACE

- To detached state:
  - session.close()
  - evict
  - clear

EVICT

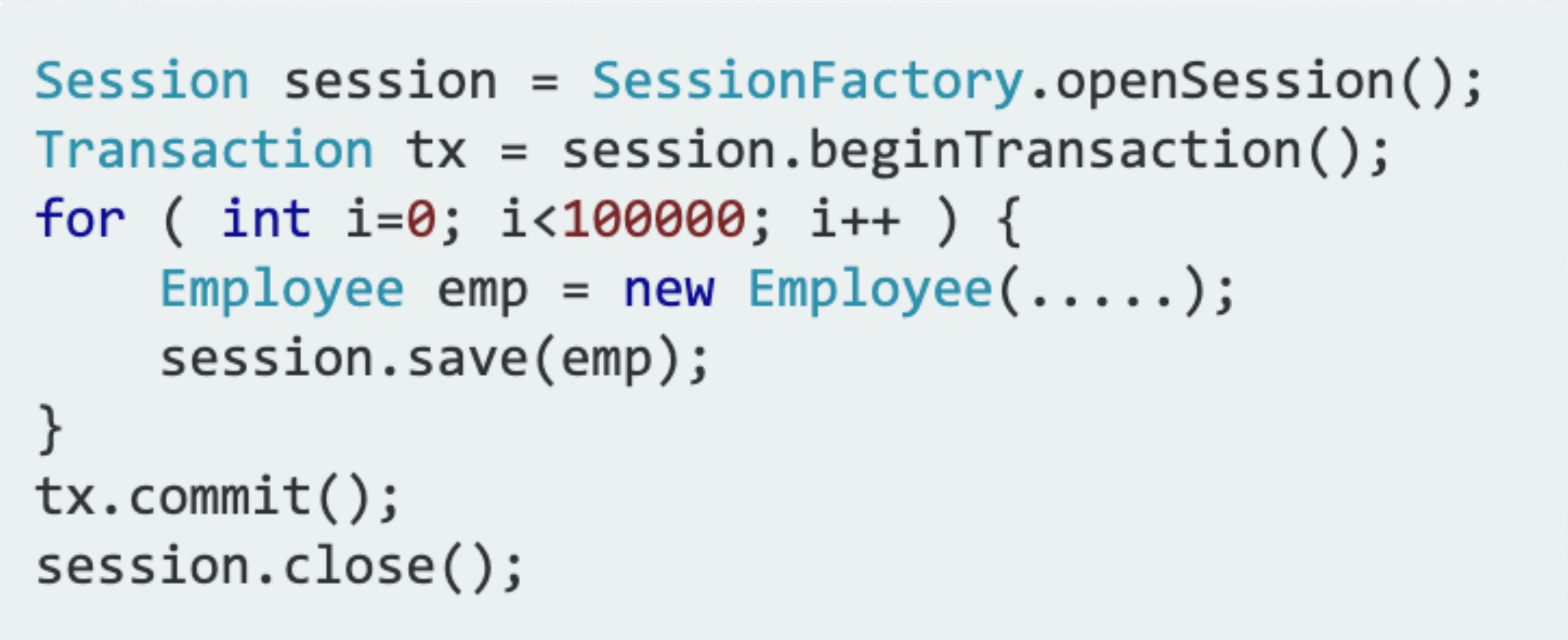
- remove the object from persistent state.
    - After detaching the object from the session, any change to object will not be persisted

CLEAR

- All objects which are currently associate with a session will be disconnected and enter detached state.

FLUSH

- It is used to synchronize session data with database.
  - When we call session.flush(), the statements are executed in database but it will not committed.
  - seesion.flush() just executes the statements in database (but not commits) and statements are NOT IN MEMORY anymore
- Why do we need to call flush()? Consider the following code:



```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee emp = new Employee(.....);
    session.save(emp);
}
tx.commit();
session.close();
```

ANY QUESTIONS