Java Persistence API and Hibernate

Jaroslav Dytrych

Faculty of Information Technology Brno University of Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole
dytrych@fit.vutbr.cz

31 October 2023

Java Persistence API

- Database management
  - Java Persistence API
    - uses JDBC (Java Database Connectivity)
  - Object-relational mapping
  - Query language
- Entities
  - SQL tables – Java classes
- Fields
  - Table columns – class properties
    - Transient – temporary, not stored in the database
    - Persistent – stored in the database
    - Inverse – stored in the database in another entity table

- **Must be indicated as an Entity**
  - `@Entity` annotation on the class

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

  - Entity entry in XML mapping file

    ```
    <entity class="com.acme.Employee"/>
    ```

- **Must have a persistent identifier (primary key)**

```
@Entity
public class Employee {
    @Id int id;
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}
```

Persistent Identity

- Identifier (id) in entity is a primary key in the database
- Uniquely identifies entity in memory and in DB
- Simple id – single field/property

```
@Id int id;
```

- Compound id – multiple fields/properties

```
@Id int id;
@Id String name;
```

- Embedded id – single field of primary key (PK) class type

```
@EmbeddedId EmployeePK id;
```

- Identifiers can be generated in the database by specifying `@GeneratedValue` on the identifier

```
@Id @GeneratedValue
int id;
```

- `AUTO` – the persistence provider should pick an appropriate strategy for the particular database.
- `IDENTITY` – supports identity columns in DB2, MySQL, MS SQL Server, ...

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

- `SEQUENCE` – uses a sequence in DB2, PostgreSQL, Oracle, ...
- `TABLE` – simulates a sequence using a table to support this strategy
- HiLo – uses a hi/lo algorithm to efficiently generate identifiers that are unique only for a particular database

- `native` – the persistence provider should use default generator for the particular database.

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO, generator="native")
@GenericGenerator(name = "native")
@Column(name = "id")
private int id;
```

- `HiLo` – uses a hi/lo algorithm.

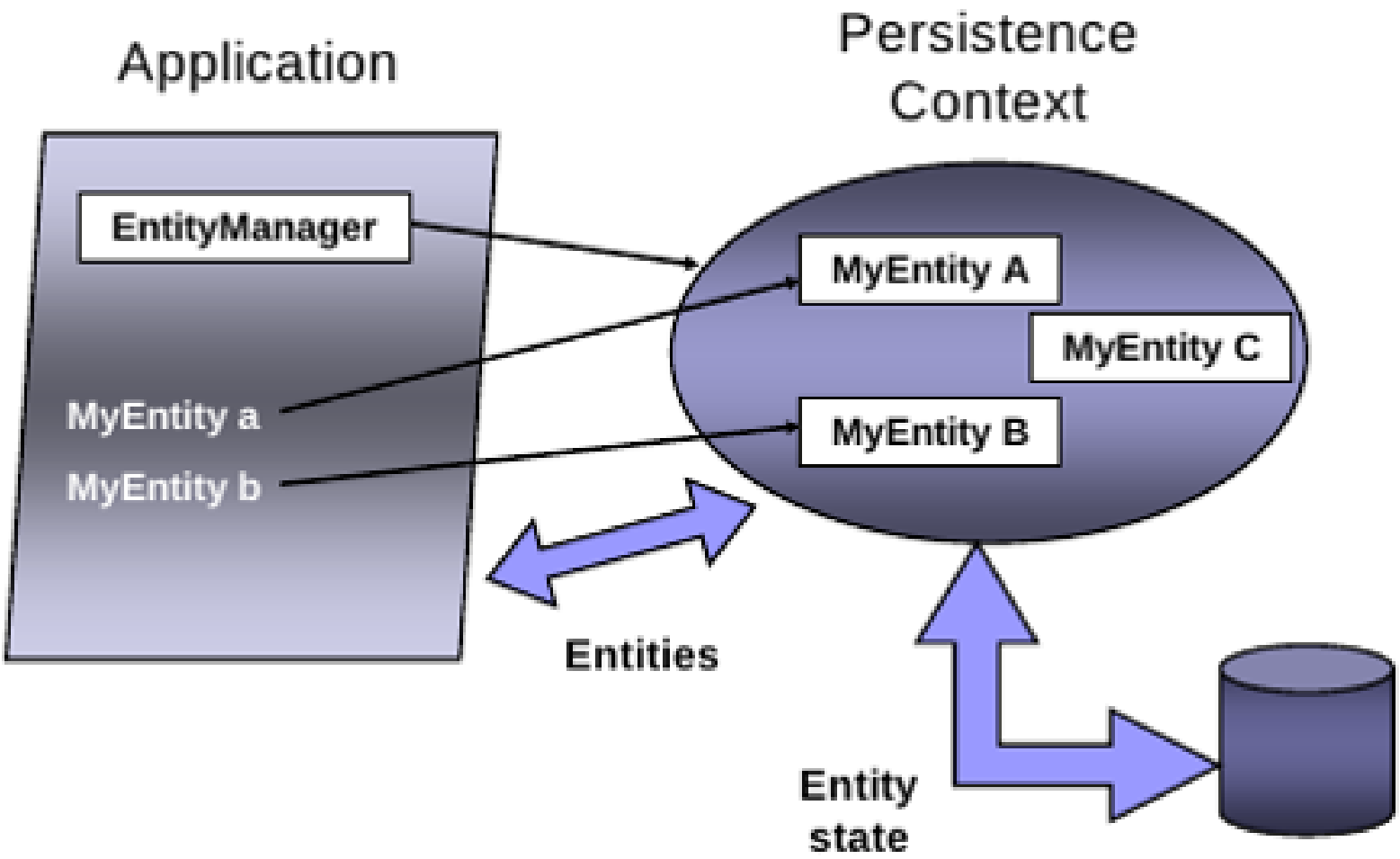
```
@GenericGenerator(name="table-hilo-generator",
  strategy="org.hibernate.id.TableHiLoGenerator",
  parameters={
    @Parameter(value="hibernate_id_generation",
      name="table")
  })

@GeneratedValue(generator="table-hilo-generator") @Id
private Long id;
```

- ...

Persistence Context

- Persistence Context (PC) is an abstraction representing a set of "managed" entity instances.
  - Entities are keyed by their persistent identity.
  - Only one entity with a given persistent identity may exist in the system.
- PC is controlled and managed by `EntityManager`
  - Contents of PC change as a result of operations on EntityManager API.

Persistence Context



- Client-visible artifact for operating on entities.
- API for all the basic persistence operations.
- Can think of it as a proxy to a persistence context.
- **May access multiple different persistence contexts throughout its lifetime.**

Entity manager API

- `persist()` – insert the state of an entity into the DB (no return value).
- `merge()` – synchronize the state of a detached entity with the PC. **Returns the managed instance** that the state was merged to.
- `remove()` – delete the entity state from the DB.
- `refresh()` – reload the entity state from the DB.
- `find()` – execute a simple PK query.
- `createQuery()` – create query instance using dynamic JPQL (Java Persistence Query Language).
- `createNamedQuery()` – create an instance for a predefined query.
- `createNativeQuery()` – create an instance for an SQL query.
- `contains()` – determine if entity is managed by the PC.
- `flush()` – force synchronization of the PC to the database.

- Save the persistent state of the entity and any owned relationship references.
- Entity instance becomes managed.

```
public Customer createCust(int id, String name) {
    Customer cust = new Customer(id, name);
    entityManager.persist(cust);
    return cust;
}
```

find() and remove()

- `find()`
  - obtains a managed entity instance with a given persistent identity – returns `null` if not found.

- `remove()`
  - deletes a managed entity with the given persistent identity from the database.

```
public void removeCustomer(Long custId) {
    Customer cust =
        entityManager.find(Customer.class, custId);
    entityManager.remove(cust);
}
```

Overview:

- Dynamic or statically defined (named queries)
- Criteria using JPQL (extension of EJB QL)
- Native SQL support (when required)
- Named parameters bound at execution time
- Pagination and ability to restrict size of result
- Single/multiple-entity results, data projections
- Bulk update and delete operation on an entity
- Standard hooks for vendor-specific hints

- Query instances are obtained from factory methods on `EntityManager`.
- Interface `Query`
  - `getResultList()` – execute query returning multiple results.
  - `getSingleResult()` – execute query returning single result.
  - `executeUpdate()` – execute bulk update or delete.
  - `setFirstResult()` – set the first result to retrieve.
  - `setMaxResults()` – set the maximum number of results to retrieve.
  - `setParameter()` – bind a value to a named or positional parameter.
  - `setHint()` – apply a vendor-specific hint to the query (`timeout`, `cache.retrieveMode`, `cache.storeMode`,...).
  - `setFlushMode()` – apply a flush mode to the query when it gets run.
  - ...

- Use `createQuery()` factory method at runtime and pass in the JPQL query string.
- Use correct execution method
  - `getResultList()`
  - `getSingleResult()`
  - `executeUpdate()`
- Query may be compiled/checked at creation time or when executed.
- Maximal flexibility for query definition and execution.

```
public List findAll(String entityName) {
    return entityManager.createQuery(
        "select e from " + entityName + " e")
        .setMaxResults(100)
        .getResultList();
}
```

- Returns all instances of the given entity type.
- JPQL string contains the entity type. For example, if "Account" was passed in, then JPQL string would be: "**select e from Account e**".

Named Queries

```
@NamedQueries({
    @NamedQuery(name="Sale.findByCustId",
        query="select s from Sale s
            where s.customer.id = :custId
            order by s.salesDate")})
public class Sale implements Serializable {
...

public List findSalesByCustomer(Customer cust) {
    return
        entityManager.createNamedQuery("Sale.findByCustId")
            .setParameter("custId", cust.getId())
            .getResultList();
}
```

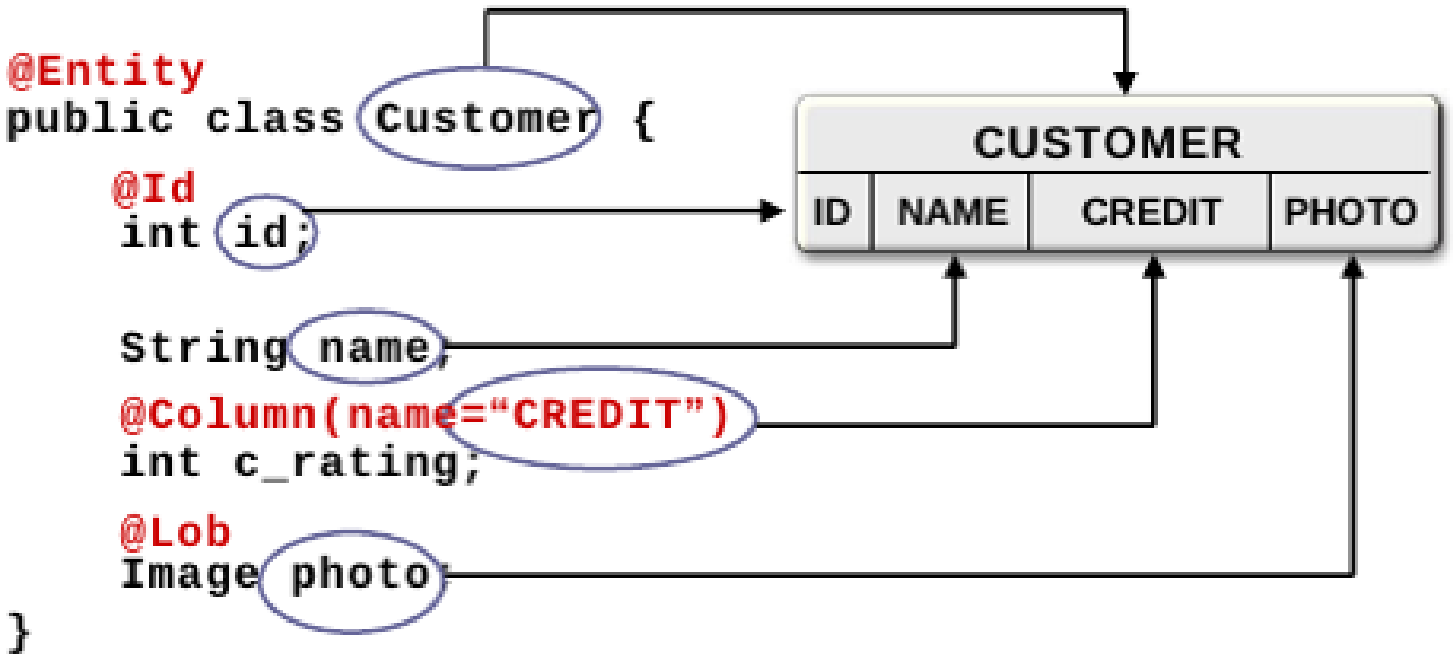
- Returns all sales for a given customer.

- ORM maps persistent object state to relational database.
- ORM maps relationships to other entities.
- Metadata may be annotations or XML (or both).
- Annotations
  - Physical – DB tables and columns (e.g. `@Table`).
  - Logical – object model (e.g. `@OneToMany`).
- XML
  - can additionally specify scoped settings or defaults.
- Standard rules for default DB table/column names.

Simple Mappings

- Direct mappings of fields/properties to columns
  - `@Basic` – optional annotation to indicate simple mapped attribute.
- Maps any of the common simple Java types
  - primitives, wrappers, enumerated, serializable, etc.
- Used in conjunction with `@Column`
  - allows to have different name for column and corresponding property.
- Defaults to the type which is most appropriate if no mapping annotation is present.
- It is possible to override any of the defaults.

Simple Mappings

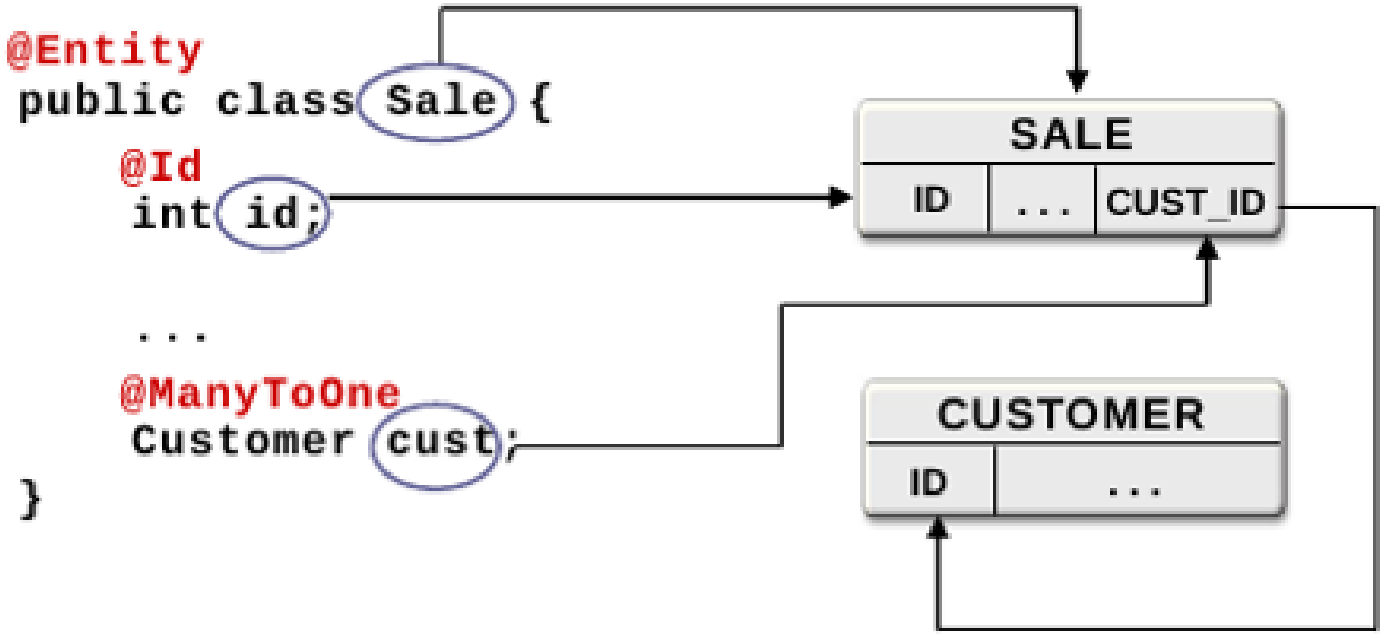


Simple Mappings

```
<entity class="com.acme.Customer">
    <attributes>
        <id name="id"/>
        <basic name="c_rating">
            <column name="CREDIT"/>
        </basic>
        <basic name="photo"><lob/></basic>
    </attributes>
</entity>
```

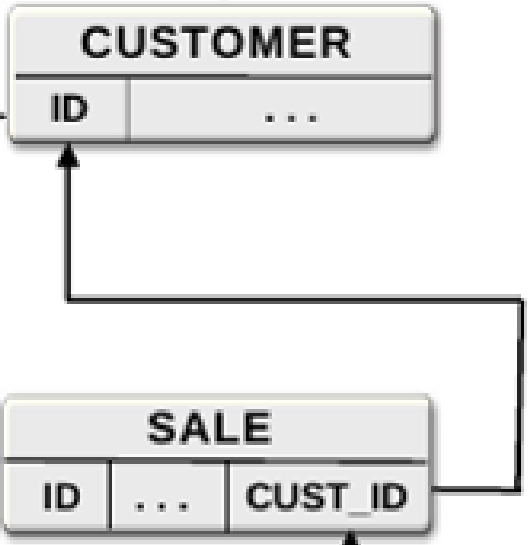
Relationship Mappings

- Common relationship mappings supported:
  - `@ManyToOne`, `@OneToOne` – single entity,
  - `@OneToMany`, `@ManyToMany` – collection of entities.
- Unidirectional or bidirectional.
- Every bidirectional relationship have owning and inverse side.
- Owning side specifies the physical mapping
  - `@JoinColumn` to specify foreign key column.


@Entity
public class Sale {
@Id
int id;
...
@ManyToOne
Customer cust;
}
SALE
ID
...
CUST_ID
CUSTOMER
ID
...

OneToMany Mapping

```
@Entity
public class Customer {
   @Id
   int id;
   ...
   @OneToMany(mappedBy="cust")
   Set<Sale> sales;
}
@Entity
public class Sale {
   @Id
   int id;
   ...
   @ManyToOne
   Customer cust;
}
```



- No deployment phase
  - We have no `EntityManagerFactory` from the container.
  - Application must use a "Bootstrap API" to obtain an `EntityManagerFactory` (usage of global `Persistence` object)

```
private EntityManagerFactory emFactory =
  Persistence.createEntityManagerFactory(
    "persistence-unit-name");
```

- Resource-local EntityManager
  - Application uses a local `EntityTransaction` obtained from the `EntityManager`.
- New application-managed persistence context for each and every `EntityManager`.
  - No propagation of persistence contexts.

Bootstrap Classes

- `javax.persistence.Persistence`
  - root class for bootstrapping an `EntityManager`,
  - locates provider service for a named persistence unit,
  - invokes on the provider to obtain an `EntityManagerFactory`.
- `javax.persistence.EntityManagerFactory`
  - Creates `EntityManager` for a named persistence unit or configuration.

Entity Transactions

- Only used by Resource-local EntityManagers.
- Isolated from transactions in other EntityManagers.
- Transaction demarcation under explicit application control using interface. `EntityTransaction`:
  - `begin()`
  - `commit()`
  - `rollback()`
  - `isActive()`
- Underlying (JDBC) resources allocated by `EntityManager` as required.

Example

persistence.xml

```
<persistence-unit name="jpa-example"
    transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <properties>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://localhost/jpa_example" />
    <property name="javax.persistence.jdbc.user"
              value="example" />
    ...

public class PersistenceProgram {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence
            .createEntityManagerFactory("jpa-example");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        // Perform finds, execute queries,
        // update entities, etc.
        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

Examples JPA-SE, Guestbook

- Java Persistence WikiBook
  - `http://en.wikibooks.org/wiki/Java_Persistence`
- Documentation
  - `http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html`
- Others
  - `https://dzone.com/articles/jpa-tutorial-setting-jpa-java`

Hibernate

- What is Hibernate
- Object-relational mapping
- Configuration
- Annotations
- Criteria
- Interceptors

- Hibernate is an Object-Relational Mapping (ORM) solution for Java.
- Open source persistent framework.
- Hibernate maps Java classes to the database tables.
- Relieves the developer of 95 % of common data persistence related programming tasks (according to the documentation).

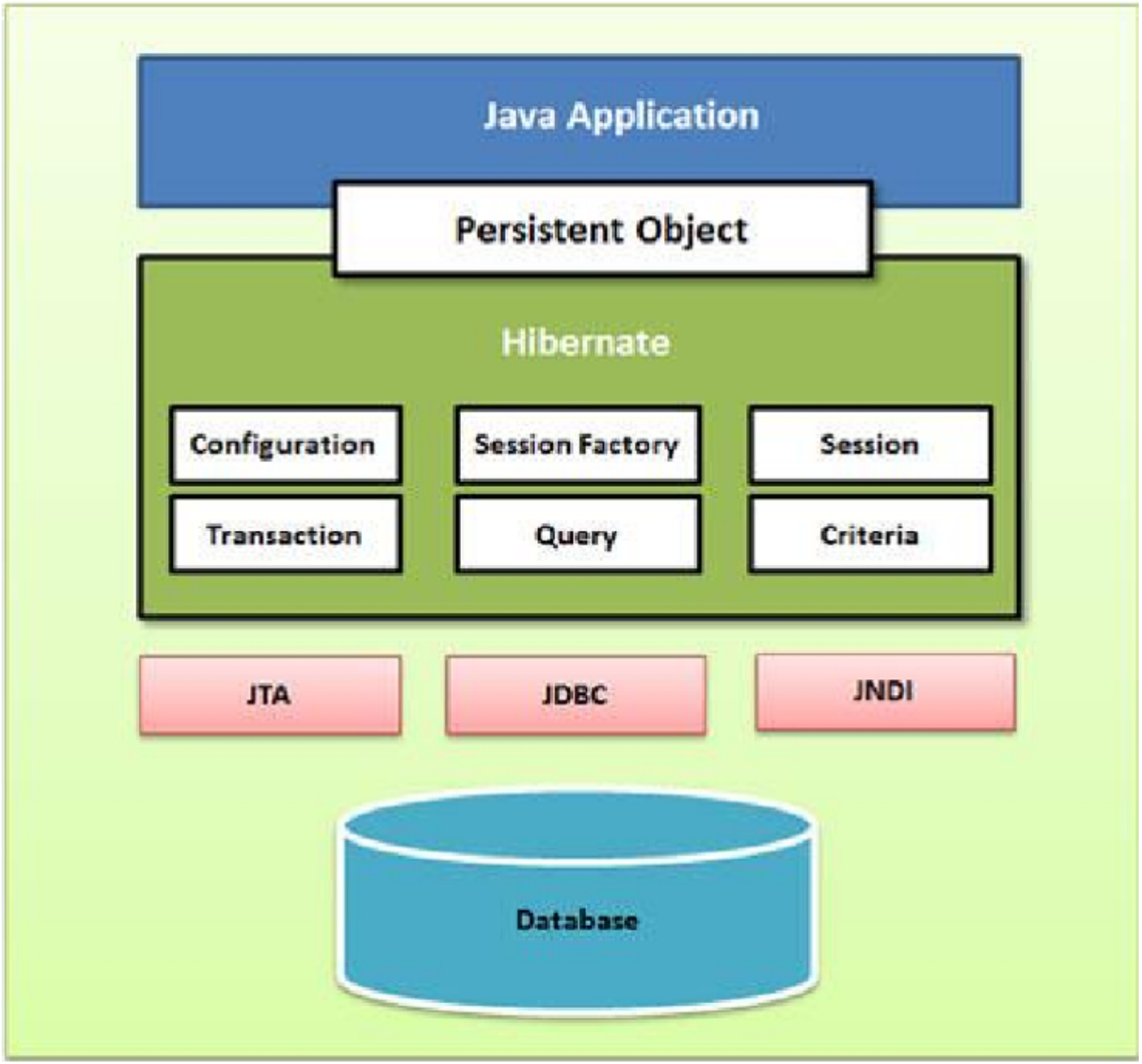
Hibernate advantages

- ORM using XML files, without writing a line of code.
- Simple API for storing/retrieving Java objects.
- Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.
- Doesn't require running application server.
- Minimizes database access with smart fetching strategies.

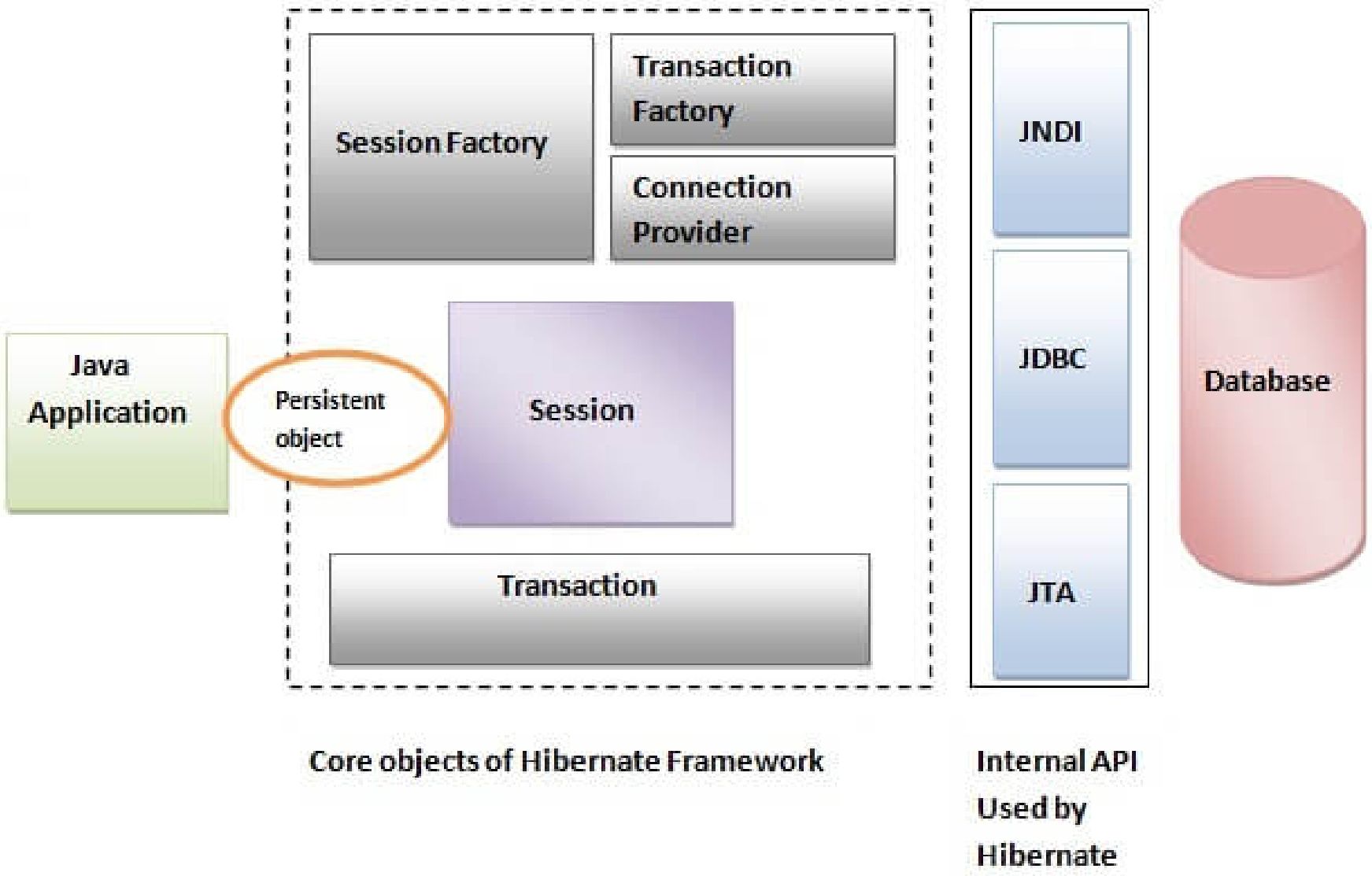
Supported databases

- HSQL Database Engine
- DB2/NT
- MySQL
- PostgreSQL
- FrontBase
- Oracle
- Microsoft SQL Server Database
- Sybase ASE
- Informix Dynamic Server
- ...

Hibernate architecture



Hibernate architecture



Technologies used by Hibernate

- Hibernate uses various existing Java APIs, like JDBC, Java Transaction API (JTA) and Java Naming and Directory Interface (JNDI).
- JDBC provides a basic level of abstraction of functionality common to the relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate.
- JNDI and JTA allows Hibernate to be integrated with JavaEE application servers.

Hibernate objects

- `Configuration`
  - typically first hibernate object created,
  - database connection,
  - class mapping setup.
- `SessionFactory`
  - created by configuration object,
  - needed one per database.
- `Session`
  - for physical connection with the database.
  - Persistent objects stored by this object.
- `Transaction`
  - optional, transaction functions from JTA or JDBC.
- `Query`
- `Criteria`

- Dialect
  - appropriate SQL for the target database
- Connection driver (e.g. JDBC driver for MySQL)
- Connection URL
- Connection username
- Connection password
- Pool size
  - number of waiting connections
- Autocommit (not recommended) – specifies when Hibernate should release JDBC connections (default behavior is that connection is held until the session is explicitly closed or disconnected).

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.driver_class">
            com.mysql.cj.jdbc.Driver
            </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/test
        </property>
        <property name="hibernate.connection.username">
            test
        </property>
        <property name="hibernate.connection.password">
            test1234
        </property>
        <!-- List of XML mapping files -->
        <mapping resource="Employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Mappping configuration file

- Mappping configuration file is needed for each table.
- `<class>` elements are used to define specific mappings from a Java classes to the database tables.
- `<meta>` element is optional element and can be used to create the class description.
- `<id>` element maps the unique ID attribute in class to the primary key of the database table.
- `<generator>` element within the id element is used to automatically generate the primary key values.
- `<property>` element is used to map a Java class property to the column in the database table.

Mapping configuration file

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
    <hibernate-mapping>
        <class name="Employee" table="EMPLOYEE">
            <meta attribute="class-description">
                This class contains the employee detail.
            </meta>
            <id name="id" type="int" column="id">
                <generator class="native"/>
            </id>
            <property name="firstName" column="first_name"
                      type="string"/>
            <property name="lastName" column="last_name"
                      type="string"/>
            <property name="salary" column="salary" type="int"/>
      </class>
  </hibernate-mapping>
```

- `@Entity`
  - must have non-argument constructor,
  - denotes entity bean.
- `@Table`
  - allows specifying of details of an entity, that will be persisted in the database.
  - attributes – `name`, `schema` (namespace), `catalogue` (named collection of schemas), constraints

```
@Entity
@Table(name = "contact",
    uniqueConstraints = @UniqueConstraint(
        columnNames = {"name", "company_id"}))
public class Contact {
  ...
```

- `@Id`
  - Each entity bean has a primary key designated by this annotation.
- `@GeneratedValue`
  - Parameter `strategy`
  - Use default generator if possible.
- `@Column`
  - `name`, `nullable`, `unique`

Hibernate annotations example

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {

    @Id @GeneratedValue
    @Column(name = "id")
    private int id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "salary")
    private int salary;

    ...
}
```

Example HibernateAnnotation

- The main function of the Session is to offer create, read, update and delete operations for instances of mapped entity classes.

- States of instances
  - **transient** – A new instance of a persistent class which is not associated with the Session and has no representation in the database and has no identifier value is considered transient by Hibernate.
  - **persistent** – You can make a transient instance persistent by associating it with the Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
  - **detached** – Once we close the Hibernate Session, the persistent instance will become a detached instance.

Session methods

- `save()`
  - persists the given transient instance, first assigning a generated identifier,
  - returns the **generated identifier**.
  - deprecated – use `persist()` instead.
- `persist()`
  - persists the given transient instance,
  - doesn't return an identifier (physical save will be done later),
- `merge()`
  - copy the state of the given object onto the persistent object with the same identifier,
  - returns an updated persistent instance.
- `update()`
  - updates the persistent instance with the identifier of the given detached instance.
  - deprecated – use `merge()` instead.

Session methods

- `delete()`
  - removes a persistent instance from the datastore,
  - argument may be an instance associated with the session or a transient instance with an identifier associated with existing persistent state.
  - Deprecated.
- `remove()`
  - removes a persistent instance from the datastore,
  - argument may be an instance associated with the session or a transient instance with an identifier associated with existing persistent state.
- `contains()`
  - checks if this instance is associated with this session.
- `get()`
  - returns the persistent instance of the given entity class with the given identifier.
  - returns the persistent instance of the given named entity with the given identifier.

Session methods

- `flush()`
  - forces this session to flush.
- `clear()`
  - completely clears the session. Evict all loaded instances and cancels all pending saves, updates and deletions,
  - frees the memory.

Examples HibernateExample, HibernateBatch

- HQL is an object-oriented language, similar to SQL.
- Instead of tables and columns HQL works with persisted objects and their properties.
- Use HQL whenever possible.
- Clauses (similar to SQL)
  - `FROM` – loads persistent object to the memory

```
String hql = "FROM Employee";
Query<Employee> query = session.createQuery(hql,
                                            Employee.class);
List<Employee> results = query.list();
```

  - `AS` – optional, aliases in HQL

```
String hql = "FROM Employee AS E";
Query<Employee> query = session.createQuery(hql,
                                            Employee.class);
List<Employee> results = query.list();
```

Hibernate clauses

- SELECT

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

- WHERE

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

- ORDER BY

```
String hql = "FROM Employee E WHERE E.id > 10 " +
             "ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

- GROUP BY

```
String hql = "SELECT SUM(E.salary), E.firstName " +
             "FROM Employee E " +
             "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

- Using named paramaters
  - no need to defend SQL injection

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

Hibernate clauses

- INSERT

```
String hql =
    "INSERT INTO Employee(firstName, lastName, salary) " +
    "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate(); // returns the number
                                    // of entities updated
                                    // or deleted
```

- UPDATE

```
String hql = "UPDATE Employee set salary = :salary " +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
```

- DELETE

```
String hql = "DELETE FROM Employee " +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
```

Aggregate methods and pagination

- Aggregate functions
  - avg
  - count
  - max
  - min
  - sum

```
String hql = "SELECT count(distinct E.firstName) FROM
    Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

- Pagination
  - setFirstResult(int startPosition)
  - setMaxResults(int maxResults)

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

- Criteria are very useful method for restricting of results

```
Criteria cr = session.createCriteria(Employee.class);
List results = cr.list();
```

- Put some constraints on result set

```
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));
// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));
// To get records having fistName starting with mary
cr.add(Restrictions.like("firstName", "mary%"));
// Case insensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "mary%"));
// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));
// To check if the given property is null
cr.add(Restrictions.isNull("salary"));
// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));
// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));
// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

- Sorting
  - Ascending `cr.addOrder(Order.asc("salary"))`
  - Descending `cr.addOrder(Order.desc("salary"))`
- Projections & aggregations

```
Criteria cr = session.createCriteria(Employee.class);
// To get total row count.
cr.setProjection(Projections.rowCount());

// To get average of a property.
cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.
cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.
cr.setProjection(Projections.max("salary"));

// To get minimum of a property.
cr.setProjection(Projections.min("salary"));

// To get sum of a property.
cr.setProjection(Projections.sum("salary"));
```

Example HibernateCriteria

- Criteria are deprecated, and removed in the newest version.
- We can replace Criteria by

`jakarta.persistence.criteria.CriteriaBuilder`

`jakarta.persistence.criteria.CriteriaQuery`

```
CriteriaBuilder cb = session.getCriteriaBuilder();
// To get total row count
CriteriaQuery<Long> cq = cb.createQuery(Long.class);
Root<Employee> root = cq.from(Employee.class);
cq.select(cb.count(root));
Long count = session.createQuery(cq).getSingleResult();
```

- Some functionality was removed

```
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
```

Example HibernateBatch

- Scalar query
  - the most basic SQL query

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List results = query.list();
```

- Entity query
  - selects raw values from the resultset

```
String sql = "SELECT * FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List results = query.list();
```

- Parameters

```
String sql = "SELECT * FROM EMPLOYEE WHERE " +
             "id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list();
```

Example HibernateNative

- Named SQL queries

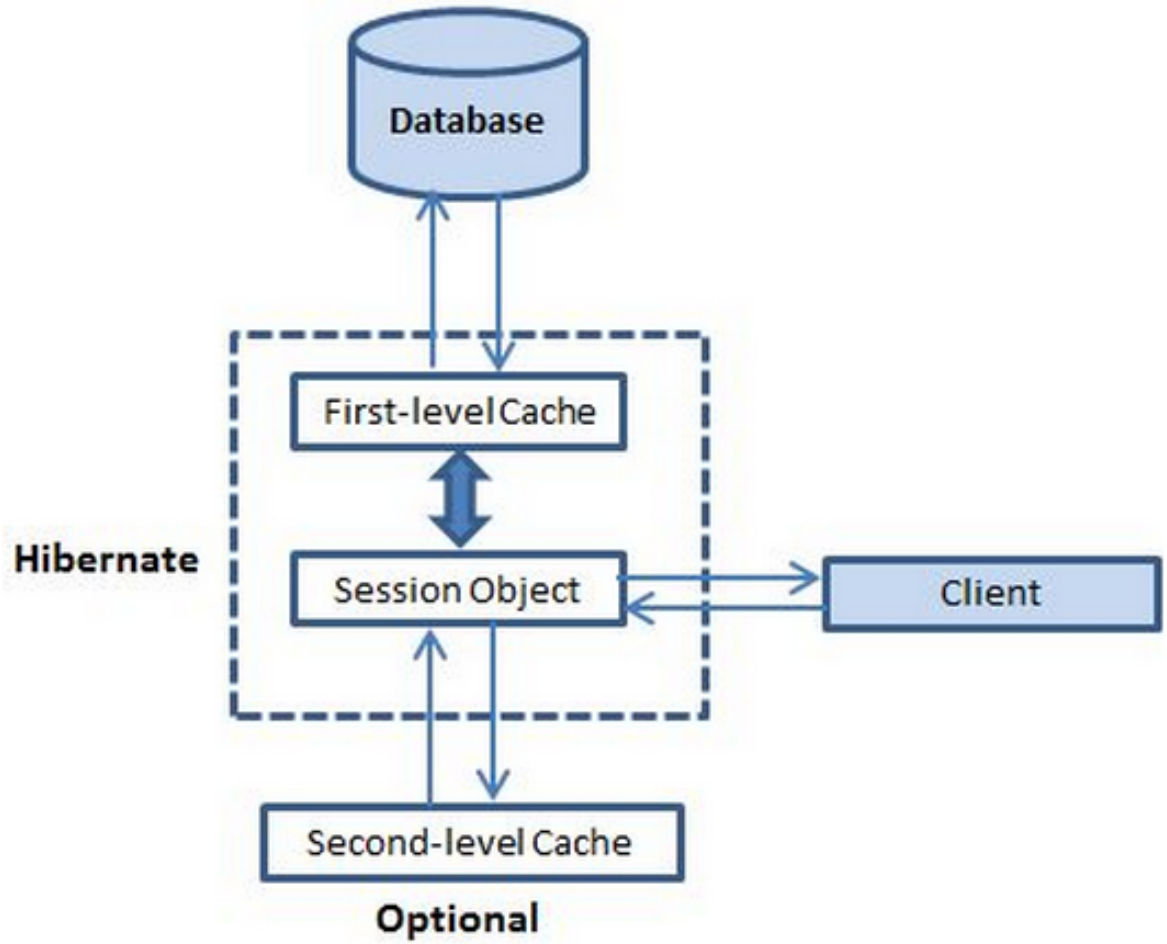
```
List employees = sess.getNamedQuery("employees")
    .setString("namePattern", namePattern)
    .list();
```

- `SQLQuery` is deprecated.
- We can replace it by `NativeQuery`.

Examples HibernateNative2, HibernateNative3

Hibernate caching

- First level cache
  - everything goes through it
  - mandatory
- Second level cache
  - per class or per collection
  - optional
- Query cache
  - for often performed queries



Hibernate interceptors

- Object goes through different stages per lifecycle.
- Interceptors are callbacks when going through these stages.
- Creating interceptor
  - either implement `Interceptor` manually
  - or extend class `EmptyInterceptor` (deprecated)
- Methods
  - `findDirty()` – called from `flush()`, returns whether the entity is updated (an array of dirty property indices).
  - `instantiate(String entityName, EntityMode entityMode, // POJO/DOM4J/MAP Serializable id)`
  - `onDelete()` – called before an object is deleted.
  - `onFlushDirty()` – object is detected to be dirty, during a flush.
  - `onLoad()` – an object is initialized.
  - `onSave()` – before an object is saved.
  - `postFlush()`
  - `preFlush()`
  - `...`

Example HibernateInterceptor

- http://www.tutorialspoint.com/hibernate
- https://docs.jboss.org/hibernate/orm/3.6/javadocs/
- https://docs.jboss.org/hibernate/orm/5.4/javadocs/
- https://docs.jboss.org/hibernate/orm/6.0/
- https://docs.jboss.org/hibernate/orm/6.3/javadocs/
- http://docs.jboss.org/hibernate/orm/3.6/reference/en-US/html/
- https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html
- https://docs.jboss.org/hibernate/orm/5.4/quickstart/html_single/
- http://blog.eyallupu.com/2011/01/hibernatejpa-identity-generators.html

References

- https://developer.jboss.org/docs/DOC-13953
- https://www.baeldung.com/hibernate-criteria-queries
- https://vladmihalcea.com/why-should-not-use-the-auto-jpa-generationtype-with-mysql-and-hibernate/
- https://www.javatpoint.com/hibernate-architecture

Thank you for your attention!