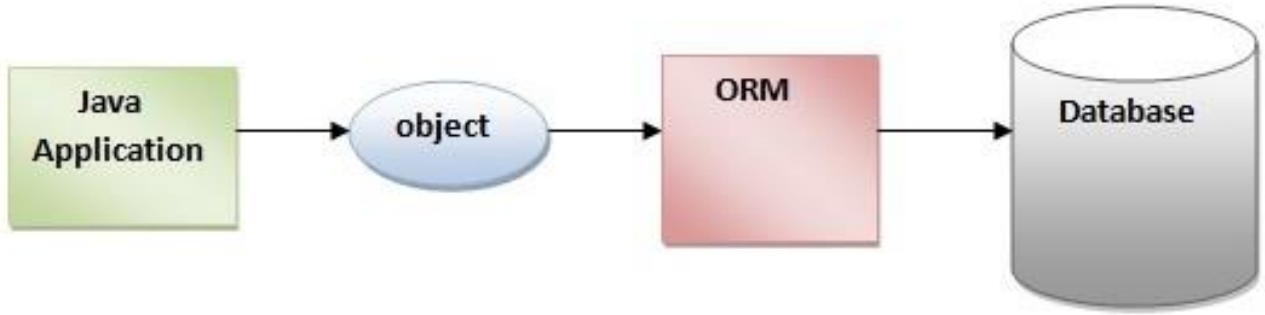
HIBERNATE

**SUBJECT:**
Advanced Java Programming

**TOPIC:**
Hibernate

# Introduction to Hibernate

- Hibernate framework simplifies the development of java application to interact with the database.
- Hibernate is an open source, lightweight, ORM tool - **Object-Relational Mapping (ORM)**
  - ORM tool simplifies the data creation, data manipulation and data access.
  - ORM tool internally uses the JDBC API to interact with the database.
- A programming technique that maps the object to the data in database.



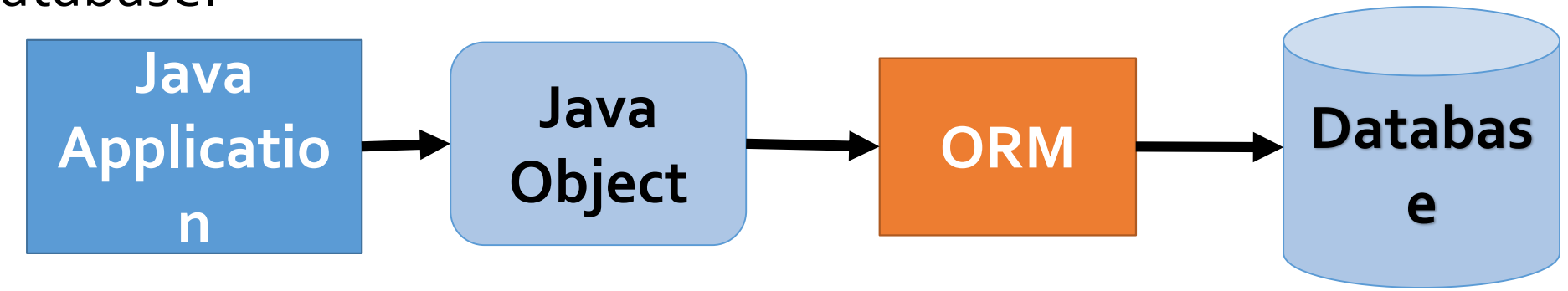
# Introduction to Hibernate (cont.)

- An open source persistent framework created by Gavin King in 2001.
- Powerful, high performance Object-Relational Persistence and Query service for any Java Application.
- Maps Java classes to database tables and from Java data types to SQL data types
- Relieves developer from 95% of common data persistence related programming tasks.

# ORM Tool

ORM tool

- Simplifies **data creation**, **manipulation** & its **access**.
- A programming technique that **maps the object to the data** stored in the database.




- **What is JPA?**
- Java Persistence API (JPA) is a Java specification that provides certain functionality and standard to ORM tools.
- javax.persistence package contains the JPA classes and interfaces.

# Introduction (cont.)

- Sits **between traditional Java objects** & **database server**
- Handles all the work for **persisting Java objects** based on the appropriate **O/R mechanisms** and patterns.

# Hibernate Advantages

- **Mapping** Java classes to database tables
- **Using XML files** & without writing any line of Java code.
- Provides **simple APIs** for
  - Storing and Retrieving Java objects directly to and from the database.
  - If there is change in the database or in any table, then you need to change the XML file properties only.

# Hibernate Advantages (cont.)

- **Abstracts away the unfamiliar SQL types** and provides a way to work around familiar Java Objects.
- Hibernate **does not require an application server** to operate.
- **Manipulates Complex associations** of objects of your database.
- Minimizes database access with **smart fetching strategies**.
- Provides **simple querying of data**.

# Hibernate Advantages (cont.)

## Open Source and Lightweight

- Hibernate framework is open source under the GPL license and lightweight.

## Fast Performance

- The performance of hibernate framework is fast because of the cache that is internally used in hibernate framework.
- Two types of cache in hibernate framework
- First level cache & Second level cache.
- First level cache is enabled by default.

# Hibernate Advantages (cont.)

**Database Independent Query**

- HQL (Hibernate Query Language) is the object-oriented version of SQL.
- Generates the database independent queries.
- NO need to write database specific queries.
- Before Hibernate, If database is changed for the project, then we need to change the SQL query as well that leads to the maintenance problem.

# Hibernate Advantages (cont.)

**Automatic Table Creation**

- Facility to create tables of the database automatically.
- No need to create tables in the database manually.

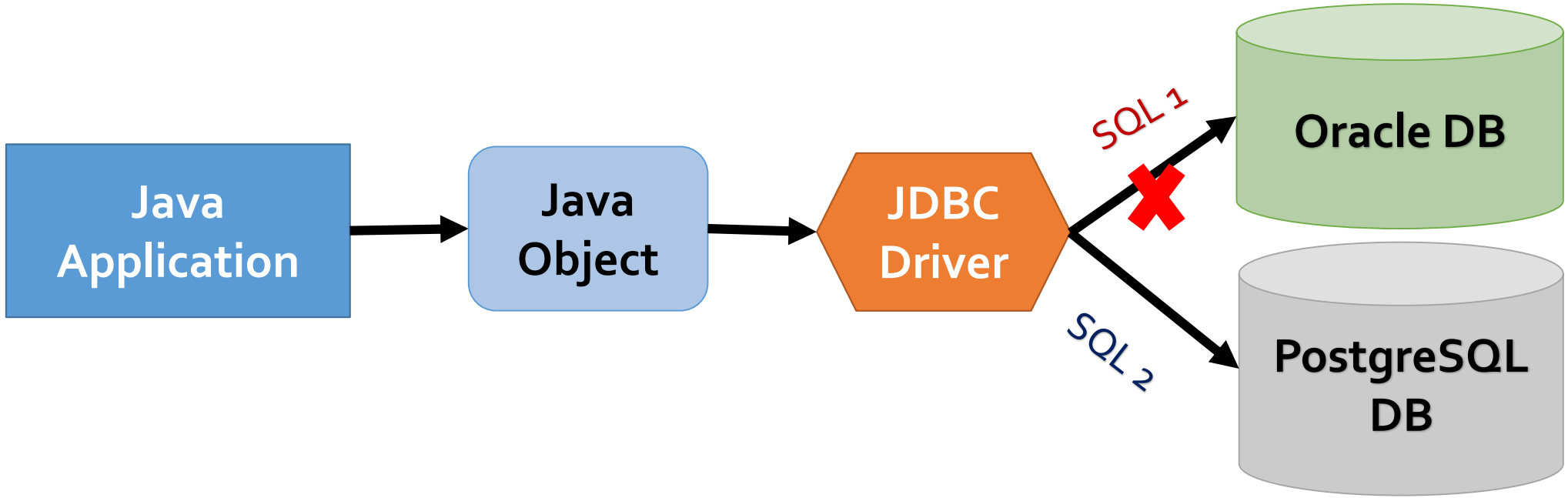
**Simplifies Complex Joins**

- Fetching data from multiple tables is easy.

**Provides Query Statistics and Database Status**

- Supports Query cache and provide statistics about query and database status.

# Hibernate Advantages (cont.)



# Supported Databases

Hibernate supports almost all the major RDBMS.

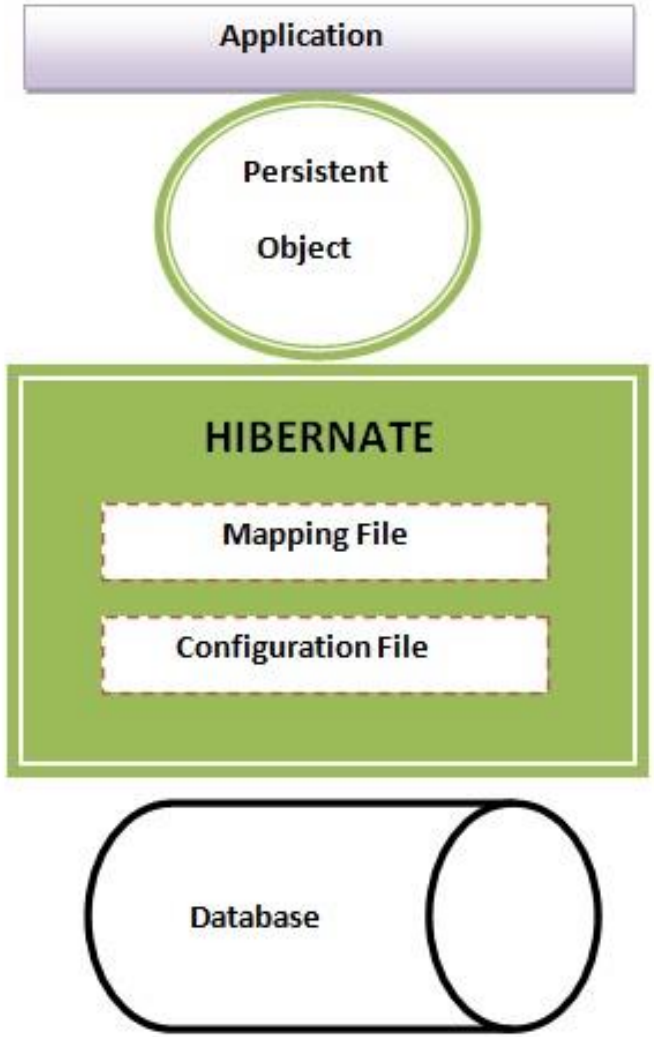
Following is a list of few of the database engines supported by Hibernate –

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

# Hibernate Architecture

4 layers in hibernate architecture:

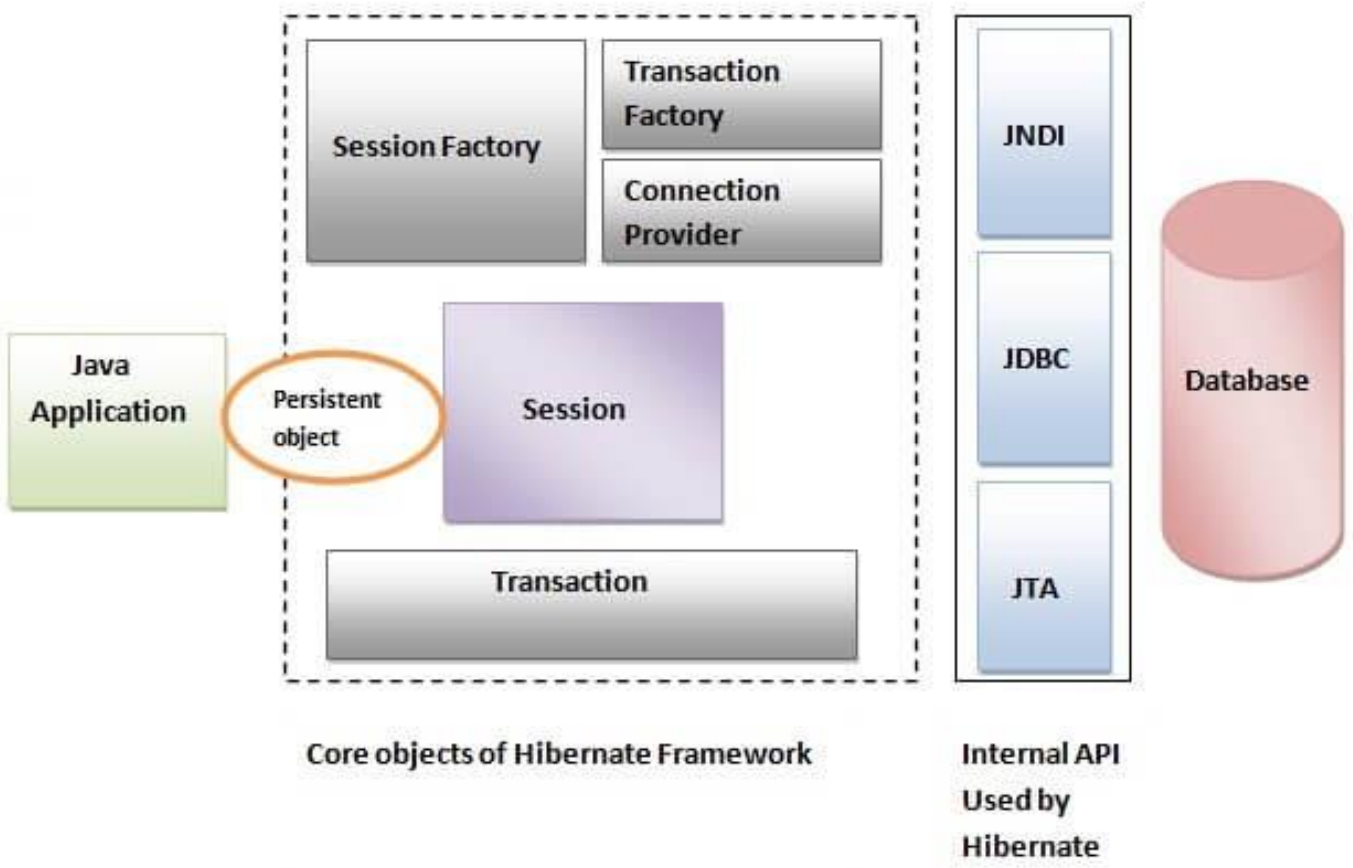
1. Java Application layer,
2. Hibernate framework layer,
3. Backhand API layer and
4. Database layer.



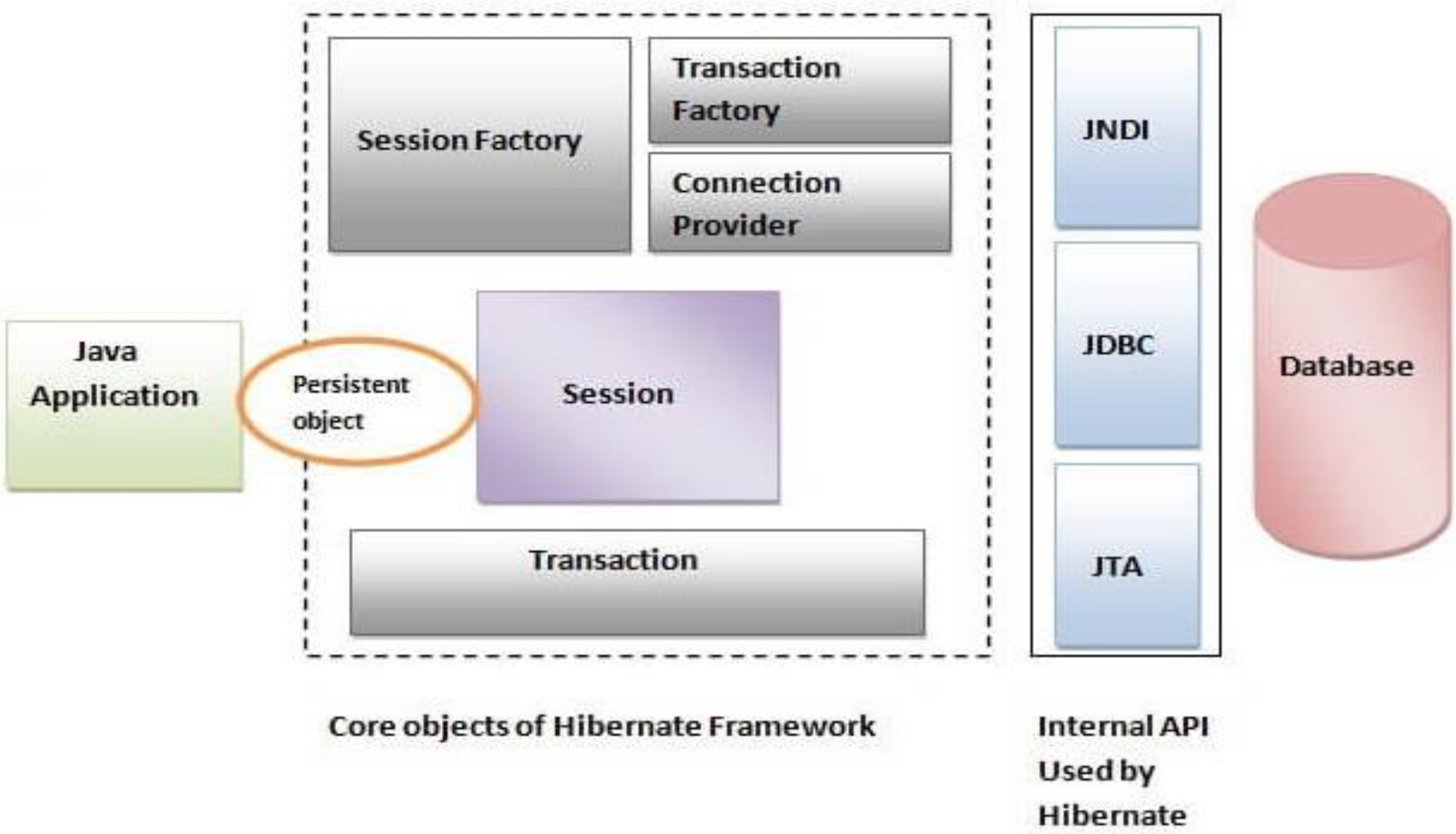
# Hibernate Architecture

Includes many objects such as

- persistent object
- session factory
- transaction factory
- connection factory
- session, transaction etc.



# Hibernate Architecture (cont.)



# Elements of Hibernate Architecture

- **SessionFactory :** Provides factory method to get the object of Session.

- **Session :** Provides an interface between the application and data stored in the database.
- A short-lived object
- It wraps the JDBC connection
- The org.hibernate.Session interface provides methods to insert, update and delete the object.
- It also provides factory methods for Transaction and Query.

# Elements of Hibernate Architecture (cont.)

- **Transaction :** org.hibernate.Transaction interface provides methods for transaction management.

- **ConnectionProvider :** It is a factory of JDBC connections.

- **TransactionFactory :** It is a factory of Transaction. It is optional.

# Configuration Object

- First Hibernate object that you create in any Hibernate application.
- Usually created only once during application initialization.

- It provides two keys components:
- **Database Connection:**
  - Handled through one or more configuration files supported by Hibernate.
  - **hibernate.properties** and **hibernate.cfg.xml**.
- **Class Mapping Setup:**
  - It creates the connection between the Java classes & database tables.

# SessionFactory Object

- Configuration object is used **to create** a SessionFactory object which in turn **configures Hibernate** for the application using the supplied configuration file **and allows for** a Session object to be instantiated.

- The **SessionFactory** is ***a thread safe object*** and used by all the threads of an application.

- The SessionFactory is a **heavyweight object**;
  - Usually created during application **start up and kept for** later use.

- One SessionFactory object is needed **per database** using **a separate configuration file**.

- Multiple databases → then create multiple SessionFactory objects

# Session Object

- Used **to get a physical** connection with a database.
- **Lightweight**
  - Instantiated each time an interaction is needed with the database.

- **Persistent objects** are **saved** and **retrieved** through Session.

- **Should not be kept open** for a long time
  - **Not usually thread safe**
  - Should be **created** and **destroyed *as needed***.

# Transaction Object

- **Represents a unit of work** with the database
- Most of the RDBMS support transaction functionality
- Handled by an underlying **Transaction manager** and transaction objects (from JDBC or JTA).

- An **optional object**
  - Hibernate applications may choose not to use this interface
  - Instead choose managing transactions **in their own application** code

## Query Object

- Query objects use
  - SQL / Hibernate Query Language (HQL) **strings**
    - **To retrieve data** from the database and create objects.

- A Query instance is used…
  - **To bind query** parameters
  - **To limit** the number of results returned
  - **To execute** the query

## Criteria Object

- To create and execute
  - **Object oriented criteria queries** to retrieve objects.

# Hibernate Configuration

- Hibernate requires to know in advance
  - Where to find the mapping information
    - Defines how your Java classes relate to DB tables.

- Requires a set of configuration settings related to database and other related parameters.

- All such information is usually supplied
  - as a standard Java properties file **hibernate.properties**,
  - or as an XML file **hibernate.cfg.xml**.

# Hibernate Properties

**hibernate.dialect**

- Makes Hibernate generate the appropriate SQL for the chosen database.

**hibernate.connection.driver_class**

- The JDBC driver class.

**hibernate.connection.url**

- The JDBC URL to the database instance.

**hibernate.connection.username**

- The database username.

# Hibernate Properties

**hibernate.connection.password**

- The database password.

**hibernate.connection.pool_size**

- Limits the number of connections waiting in the Hibernate database connection pool.

**hibernate.connection.autocommit**

- Allows autocommit mode to be used for the JDBC connection.

# Hibernate O/R MAPPING

- The mapping document is an XML document having **<hibernate-mapping> as the root element which contains <class> elements corresponding class.**
- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.
- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class.
- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** to create primary key depending upon the capabilities of the underlying database.
- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table.
  The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

# Hibernate O/R MAPPING

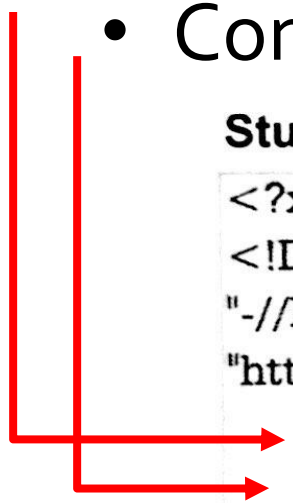
- O/R Mapping is usually defined in **XML document**.
- The mapping language is java-centric, meaning that mapping is constructed around **persistence class declaration**.

**Student.hbm.xml**

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping>
 <class name="com.mypackage.Student" table="STUDENT">
  <id name="roll">
   <generator class="assigned"></generator>
  </id>
   <property name="studname"></property>
  <property name="branch"></property>
 </class>
 </hibernate-mapping>
```

# Hibernate O/R MAPPING (cont.)

- An XML document having
  - **<hibernate-mapping>** root element
    - Contains **<class>** elements



**Student.hbm.xml**

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
  <hibernate-mapping>
  <class name="com.mypackage.Student" table="STUDENT">
   <id name="roll">
    <generator class="assigned"></generator>
   </id>
    <property name="studname"></property>
   <property name="branch"></property>
  </class>
  </hibernate-mapping>
```

# Hibernate O/R MAPPING (cont.)

- **<class>** elements
  - To define specific mappings from a Java classes to the database tables.
    - ***Class name*** → **name** attribute of the class element
    - ***Database table*** name → the **table** attribute

```
<class name="com.mypackage.Student" table="STUDENT">
 <id name="roll">
  <generator class="assigned"></generator>
 </id>
  <property name="studname"></property>
  <property name="branch"></property>
</class>
</hibernate-mapping>
```

# Hibernate O/R MAPPING (cont.)

- **<id>** element
  - Maps the unique ID attribute in class to the primary key of the database table.
  - **name** attribute of the id element ➔property in the **class**

```
<class name="com.mypackage.Student" table="STUDENT">
 <id name="roll">
  <generator class="assigned"></generator>
 </id>
  <property name="studname"></property>
  <property name="branch"></property>
</class>
</hibernate-mapping>
```

# Hibernate O/R MAPPING (cont.)

- The **<generator>** element within the id element
  - Used to generate the **primary key** values automatically
  - **class** attribute of the generator element is set to
    - assigned, increment, hilo, sequence, native, etc.

```
<class name="com.mypackage.Student" table="STUDENT">
 <id name="roll">
  <generator class="assigned"></generator>
 </id>
  <property name="studname"></property>
  <property name="branch"></property>
</class>
</hibernate-mapping>
```

# Hibernate O/R MAPPING (cont.)

- **<property>** element
  - To map a Java class property to a column in the database table.
  - **name** attribute ➔ property in the class
  - **column** attribute ➔ column in the database table
  - **type** attribute holds the hibernate mapping type
    - This mapping type will be converted from Java to SQL data type.

```
<class name="com.mypackage.Student" table="STUDENT">
 <id name="roll">
  <generator class="assigned"></generator>
 </id>
  <property name="studname"></property>
  <property name="branch"></property>
</class>
</hibernate-mapping>
```

# Hibernate Annotation

- A powerful way ***to provide the metadata*** for the Object and Relational Table mapping.

- All the metadata is clubbed into the **POJO** Class file along with the code
  - Helpful to understand the table structure & POJO during the development.

- If you going to make your application portable to other EJB 3 compliant ORM applications, you must use annotations to represent the mapping information, but still if you want greater flexibility, then you should go with XML-based mappings.

# Hibernate Annotation (cont.)

**@Entity Annotation**

- On Employee class
  - Marks this class as an entity bean
  - It must have a **no-argument constructor**
    - with at least protected scope.

**@Table Annotation**

  - To specify the details of the table ➔ used to persist the entity in the database.
- Provides four attributes:
  - Allowing you to override the **name** of the table
  - Its **catalogue**, and its **schema**, and enforce **unique constraints** on columns in the table.
  - For now just table name, which is EMPLOYEE.

# Example Continue… mapping of Employee class with annotations to map objects with the defined EMPLOYEE table

```
import javax.persistence.*;
@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;

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

# Example Continue… mapping of Employee class with annotations to map objects with the defined EMPLOYEE table

```
import javax.persistence.*;
@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;
public Employee() {}
public int getId()
{ return id; }
public void setId( int id )
 { this.id = id; }
:
```

# Hibernate Annotation (cont.)

- **@Id** and **@GeneratedValue** Annotations

- Each entity bean will have a primary key
  - Can be annotated with **@Id** annotation.
    - Primary key can be a single field or a combination of multiple fields depending on your table structure.
  - By default, it will automatically determine the most appropriate primary key generation strategy to be used.
  - Can be overridden
    - By applying the **@GeneratedValue** annotation
      - Takes two parameters **strategy** and **generator**

# Example Continue… mapping of Employee class with annotations to map objects with the defined EMPLOYEE table

```
import javax.persistence.*;
@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;
public Employee() {}
public int getId()
{ return id; }
public void setId( int id )
 { this.id = id; }
:
```

# Hibernate Annotation (cont.)

**@Column Annotation**

- To specify the details of the column to which a field or property will be mapped. Commonly used attributes:
  - **name**
    - permits the name of the column to be explicitly specified.
  - **length**
    - permits the size of the column used to map a value particularly for a String value.
  - **nullable**
    - permits the column to be marked NOT NULL when the schema is generated.
  - **unique**
    - permits the column to be marked as containing only unique values.

# Example Continue… mapping of Employee class with annotations to map objects with the defined EMPLOYEE table

```
import javax.persistence.*;

@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;

public Employee() {}

public int getId()
{ return id; }

public void setId( int id )
 { this.id = id; }
:
```

# Annotated Class Example

- In Hibernate Annotation, all the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

- Consider the following EMPLOYEE table SQL:
- ```
  create or replace table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name VARCHAR(20) default NULL,
   salary INT default NULL,
   PRIMARY KEY (id)
   );
  ```
  ```

# Example Continue… mapping of Employee class with annotations to map objects with the defined EMPLOYEE table

```
import javax.persistence.*;

@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;

public Employee() {}

public int getId()
{ return id; }

public void setId( int id )
  { this.id = id; }
```

# Example Continue… mapping of Employee class with annotations to map objects with the defined EMPLOYEE table

```
public String getFirstName()
{ return firstName; }

public void setFirstName(
String first_name )
{this.firstName =
first_name;}

public String getLastName()
{ return lastName; }

public void setLastName
   (String last_name)
{this.lastName = last_name; }

public int getSalary()
 { return salary; }

public void setSalary( int
salary )
  { this.salary = salary; }

} // End of Employee class
```

# Example Continue…

**Compilation and Execution**

- Create an application class with main( ) method.

**Database Configuration**

- Create **hibernate.cfg.xml** configuration file to define database related parameters.

**Compilation and Execution**

- Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.
- Delete Employee.hbm.xml mapping file from the path.
- Create Employee.java source file as shown above and compile it.
- Create ManageEmployee.java source file as shown above and compile it.
- Execute ManageEmployee binary to run the program.

# HIBERNATE QUERY LANGUAGE(HQL)

- Hibernate is supported by a very powerful query language which is just similar to SQL.
- The Hibernate query language is an object oriented query language.
- It works with persistent object and its property.
- HQL does not depends upon table of database. Instead of table name we use class name in HQL.
- The Hibernate query are converted into conventional query that in turn perform operation.
- These query are case sensitive.

# Advantages of HQL:

1. The HQL can perform bulk of operation at a time on Hibernate.
2. HQL supports object oriented feature such as inheritance, polymorphism, association and so on.
3. Instead of returning plain data HQL return object. These objects can be easily accessed or programmed.
4. HQL is database independent. The same HQL can be executed on different databases.

# SQL Vs HQL

1. HQL stands for Hibernate Query Language and SQL stands for Structured Query Language

2. The structure of HQL is similar to SQL, but the main deference is that HQL makes use of class name instead of table and property name.

3. HQL is object oriented query language and sql is not.

4. HQL is database independent where as SQL is database dependent.

# HQL Interface

- **public int executeUpdate()** is used to execute the update or delete query.
- **public List list()** returns the result of the ralation as a list.
- **public Query setFirstResult(int rowno)** specifies the row number from where record will be retrieved.
- **public Query setMaxResult(int rowno)** specifies the no. of records to be retrieved from the relation (table).

# Some common Queries in HQL

**HQL to get all the records**

```
Query query = session.createQuery("from Emp");
//here persistent class name is Emp
List list = query.list();
```

**HQL to get records with pagination**

```
Query query = session.createQuery("from Emp");
query.setFirstResult(5);
query.setMaxResult(10);
List list = query.list();
//will return the records from 5 to 10th no.
```

# Some common Queries in HQL

## HQL delete query

```
Query query =
session.createQuery("delete from Emp where id=100");


//specifying class name (Emp) not tablename
query.executeUpdate();
```

# Hibernate Generator Classes

**Hibernate Generator Classes**

- While saving an object into the database
  - Generator informs to the hibernate
    - how the primary key value for new record is to be generated

- Hibernate uses different primary key generator algorithms
  - For each algorithm internally
    - A class is created by hibernate for its implementation

# Hibernate Generator Classes (cont.)

**Hibernate Generator Classes**

```
<id ...>
    <generator class="..."></generator>
</id>
```

- a sub-element of id
- to generate the unique identifier for the objects of persistent class
- implements the **org.hibernate.id.IdentifierGenerator** interface
- application programmer may create his/her own generator classes
  - by implementing the IdentifierGenerator interface

# Hibernate Generator Classes (cont.)

**Framework provides many built-in generator classes:**

- assigned
- increment
- sequence
- native
- identity
- hilo
- seqhilo
- uuid
- guid
- select
- foreign
- sequence-identity

# Hibernate Generator Classes (cont.)

**1. assigned**

- It is the default generator strategy if there is no <generator> element . In this case, application assigns the id.
- Supports all the databases
- Default generator class
  - If No <generator> element specified under id element
  - Then hibernate by default assumes it as “assigned”
- Programmer is responsible for assigning the primary key value to object
  - Object to be saved into the database

# Hibernate Generator Classes (cont.)

**1. assigned example:**

```
<hibernate-mapping>
  <class ...>
    <id ...>
      <generator
      class="assigned"></generator>
    </id>
    .....
  </class>
 </hibernate-mapping>
```

# Hibernate Generator Classes (cont.)

**2. increment**

- Generates id value for the new record by using the formula
  - Max of id value in Database + 1
- Supports in all the databases & database independent
- If there is no record initially in the database, then for the first time this will saves primary key value as 1, as…
  - result = max of id value in database + 1 → 0 + 1 → 1
  - short, int or long type identifier.

# Hibernate Generator Classes (cont.)

**2. increment**

- if we manually assigned the value for primary key for an object, then hibernate doesn't considers that value and uses max value of id in database + 1 concept only
- If a table contains an identifier
  - then the application considers its maximum value
  - else considers the first generated identifier to be 1
- Increments the identifier by 1

# Hibernate Generator Classes (cont.)

**2. increment**

- Syntax:

```
<hibernate-mapping>
 <class ...>
  <id ...>
   <generator class="increment"></generator>
  </id>
    .....
 </class>
</hibernate-mapping>
```

# Hibernate Generator Classes (cont.)

**3. sequence**

- Uses the sequence of the database.
- For inserting a new record in a database
  - Gets next value from the sequence in the DB
    - Assigns that value for the new record
- If no sequence is defined,
  - it creates a sequence automatically
- Ex: in case of Oracle database,
  - it creates a sequence named HIBERNATE_SEQUENCE.
- For Oracle, DB2, SAP DB, Postgre SQL or McKoi,
  - it uses sequence but it uses generator in interbase.

# Hibernate Generator Classes (cont.)

**3. sequence**

- Not supported in MySQL

- Database dependent
  - We cannot use this generator class for all the databases
  - We should know whether the database supports sequence or not before we are working with it

- Programmer has to create a sequence in the database
  - That Sequence name should be passed as the generator

# Hibernate Generator Classes (cont.)

**3. sequence**

- If the programmer has not passed any sequence name,
  - Then hibernate creates its own sequence with name "Hibernate-Sequence"
  - Gets next value from that sequence
  - Than assigns that id value for new record

- To create its own sequence, in hibernate configuration file:
  - hbm2ddl.auto property
    - must be set enabled

# Hibernate Generator Classes (cont.)

**3. sequence**

- Syntax:

```
<hibernate-mapping>
 <class ...>
  <id name="productId" column="pid">
  <generator>
  <param name="sequence">MySequence</param>
  </genetator>
  </id>
 </class>
</hibernate-mapping>
```

# Hibernate Generator Classes (cont.)

**4. native**

- It uses identity, sequence or hilo depending on the database vendor.
- It first checks whether the database supports identity or not,
  - if not, then checks for sequence and
  - if not, then hilo will be used
    - identity
    - sequence
    - hilo

# Hibernate Generator Classes (cont.)

**4. native**

- For example,
  - If we are connecting with oracle
    - We use generator class as native
    - Then it is equal to the generator class sequence
- Syntax:

```
<id ...>
 <generator class="native"></generator>
</id>
```

# Hibernate Generator Classes (cont.)

**5. identity**

- It is used in Sybase, My SQL, MS SQL Server, DB2 and HypersonicSQL to support the id column.
- id is of type short, int or long.
- Responsibility of database to generate unique identifier.
- Database dependent
- Doesn’t work with Oracle DB
- Doesn’t needs any parameters to pass

# Hibernate Generator Classes (cont.)

**5. identity**

- ID value is generated by the database, not by the hibernate
  - For incrementing, hibernate will take over this
- Similar to increment generator,
  - But increment generator is database independent
  - Hibernate uses a select operation for selecting max of id before inserting new record
- But here
  - No select operation will be generated
    - To insert an id value for new record

# Hibernate Generator Classes (cont.)

**6. hilo**

- Database independent
- It uses high and low algorithm to generate the id
  - of type short, int and long

- Syntax:

```
<id ...>
  <generator class="hilo"></generator>
</id>
```

# Hibernate Generator Classes (cont.)

**7. seqhilo**

- It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.

**8. uuid**

- It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

**9. guid**

- It uses GUID generated by database of type string. It works on MS SQL Server and MySQL.

# Hibernate Generator Classes (cont.)

**10. select**

- It uses the primary key returned by the database trigger.

**11. foreign**

- It uses the id of another associated object, mostly used with <one-to-one> association.

**12. sequence-identity**

- It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only.