HIBERNATE
WITH JPA

DRAWBACKS OF JDBC

The drawbacks of JDBC are:

- We need to write lengthy code
- Remembering SQL queries
- Creating tables in the database explicitly
- Follow 5 steps every time to perform CRUD operations.

WHAT IS HIBERNATE?

1. Hibernate is a Java framework that simplifies the development of Java application to interact with the database
2. Hibernate is a open-source ORM tool .
3. Hibernate implements JPA specification for data persistence.
4. Hibernate framework can be used to create tables in the database implicitly.
5. It has inbuilt methods which implicitly calls the SQL queries

WHY HIBERNATE?

1. Hibernate Reduces lines of code by maintaining object-table mapping itself and returns result to application in form of Java objects.
2. Hibernate does the implementation internally like writing queries & Establishing connection etc.
3. Hibernate removes a lot of boiler-plate code that comes with JDBC API, the code looks more cleaner and readable.
4. Hibernate supports inheritance, associations and collections. These features are not present with JDBC API.
5. JDBC is database dependent i.e. one needs to write different codes for different database. Whereas **Hibernate** is database independent and same code can work for many databases with minor changes.
6. Automatic SQL Query Generation. There is no problem of remembering sql queries.

ORM

1. ORM is an acronym of **OBJECT-RELATIONAL MAPPING.**
2. ORM acts as a converter between java object and a database.
3. ORM Handles the logic required to interact with databases.
4. You write less code when using ORM tools than with SQL.
5. There are some popular ORM tools:

- Hibernate
- TopLink
- Eclipse Link
- Open JPA
- MyBatis (ibatis)

6. The most used ORM tool is Hibernate.
7. The ORM tool internally uses the **JDBC API** to interact with the database

JPA

1. JPA is a acronym of **JAVA PERSISTENCE API**.
2. JPA is a specification, not an implementation.
3. It is the standard application programming interface that makes database operations simple for developers to carry out.
4. Hibernate is an implementation of the Java Persistence API (JPA).
5. A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database.
6. As JPA is a specification it does not perform any operations by itself ,thus it requires implementation so ORM tools like Hibernate, ibatis, Toplink, EclipseLink implements JPA specification for data persistence.
7. JPA specification is same for all the ORM tools and it follows same standards ,so in the future if we want to switch our application from one ORM tool to another then we can do it easily.

DIFFERENCE BETWEEN JPA AND HIBERNATE

JPA(JAVA PERSISTENCE API)	HIBERNATE
JPA manages the relational databases in java applications	Hibernate is an ORM tool which is used to save the state of the object into the database
JPA is a specification various ORM tools implement it for data persistence	Hibernate is an implementation and it's a most frequently used ORM tool
It is defined in javax.persistence package	It is defined in org.hibernate package
It uses JPQL (Java persistence query language)as a object oriented programming language to perform database operations	It uses Hibernate query language(HQL) to perform database operations
To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager	To create Session instances, it uses **SessionFactory** interface
It uses EntityManager to perform crud operations like create, update, delete, read	It uses Session interface to perform crud operations

CLASSES AND INTERFACES OF JPA

ENTITYMANAGERFACTORY:

1. It is an interface present in javax.persistence package
2. It provides EntityManager
3. It provides an efficient way to construct multiple entitymanager instances.
4. With the help of an helper class called Persistence which has a static method called createEntityManagerFactory(String persistenceUnitName) which accepts persistence unit name present in persistence.xml file , it returns an instance of EntityManagerFactory

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("pun");
```

5. It provides a methods like
   - createEntityManager()
   - createEntityManager(Map map)

ENTITYMANAGER

1.It is interface present in javax.persistence package.

2.It manages the lifecycle of entity instance.

3.It performs some of the operations and it has a methods like to insert,update,delete and fetch the data.

4.It provides different methods which Implicitly call the sql queries

- **persist(Object entity):** It is used to insert the data into database (acts as insert query)
- **merge(Object entity):** it is used to update the data into the database based on primary key

(NOTE: if the PK is available in the database it acts as update query ,else It acts as insert query)

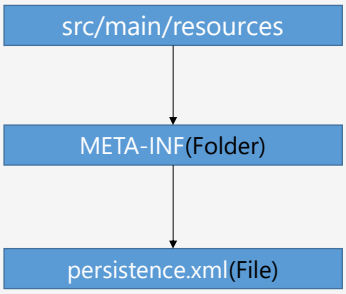
- **remove(Object entity):** It is used to delete the data from the database.(acts as delete query)
- **find(Class<T> entityClass, Object primaryKey):** It is used to fetch the single data based on primary key from the database(it acts as select query with where condition as PK)

ENTITYTRANSACTION

- It is an interface which provides important method to handle transaction in JPA application.
- Transaction is set of operations that either fail or succeed.
- A database transaction consist of a set of sql operations that are committed or rollback
- This interface is used to control transactions on resource-local entity managers.
- EntityTransaction Interface provides important methods to handle Transactions in JPA based applications. The below are important methods of EntityTransaction Interface.
  - Void `begin()` - Start a resource transaction.
  - `void commit()` - Commit the current resource transaction, writing any unflushed changes to the database.
- The EntityManager.getTransaction() method returns the EntityTransaction interface.

persistence.xml File

- The persistence. xml file is a standard configuration file in JPA
- It has to be included in the META-INF directory
- The persistence.xml file must define a persistence-unit with a unique name
- The persistence.xml file is used to configure Entity classes, provide database driver, database connection information and other relational mapping details
- The persistence.xml file is created inside:



Persistence

- Persistence is a helper class
- Persistence class provides a method called createEntityManagerFactory() which accepts the Persistence unit name present in persistence.xml file
- With the help of EntityManagerFactory we can access the persistence unit name.
- EntityManagerFactory emf=Persistence.createEntityManagerFactory("pun");

HAS-A-RELATIONSHIP

- To map the java objects to Relational table we make use of some Annotations.
- In JDBC we use to create a table but in hibernate creation of table is done by using a annotation called **@Entity.**
- To make any attribute as Primary key annotate it with **@Id**
- **Entity:** Creation of table
- **Id:** To make the given attribute has primary key

Example:

Entity Class

```
//create class Employee
@Entity
class Employee{
@Id
private int id;
private String name;
private double salary;
//getters and setters
}
```


Table

id	name	salary
1	Ram	10000
2	Arun	3000

MAPPING

- Mapping can be done as below:

1. **Uni-direction:**
   - Onetoone uni
   - Onetomany uni
   - Manytoone uni
   - Manytomany uni
2. **Bi-direction:**
   - Onetoone bi
   - Onetomany bi
   - Manytoone bi
   - Manytomany bi

OneToOne-Unidirection

```
//create a class Person
@Entity
class  Person{
@Id
private int id;
private String name;
private String address;
@onetoone
private Passport passport;
//getters and setters
}
```

1 — 1

```
//create a class Passport
@Entity
class PassPort{
@Id
private int pid;
private String name;
private long phone;
//getters and setters
```

id	name	address	pid
1	ram	bangalore	10
2	vijay	mumbai	20

pid	name	phone
10	gagan	987654234
20	ajay	768945321

@onetoone- it is a annotation which will create a relationship b/w two classes in sql and generates foreign key.

OneToMany-unidirection

```
//create class Company
class  Company{
@Id
private int id;
private String name;
private String address;
@onetomany
private List<Employee>
employee;
//getters and setters
```

1 — n

```
//create class Employee
class Employee{
@Id
private int id;
private String name;
private String email;
private String password;
//getters and setters
```

id	name	address
1	ram	Blr
2	raj	chn

company_employee

c_id	e_id
1	10
2	20

id	name	Email	password
10	arun	a@gmail.com	123
20	rajath	r@gmail.com	234

ManytoOne-unidirection

```
//create class Employee
class Employee{
@Id
private int id;
private String name;
private String email;
private String password;
@manytoone
private Company company;
//getters and setters
```

n — 1

```
//create class Company
class  Company{
@Id
private int id;
private String name;
private String address;
//getters and setters
```

id	name	Email	password	c_id
10	arun	a@gmail.com	123	1
20	rajath	r@gmail.com	234	2

id	name	address
1	wipro	Blr
2	infosys	chn

ManytoMany-unidirection

```
//create class Person
class Person{
@Id
private int id;
private String name;
private String address;
@manytomany
private Language  language;
//getters and setters
```

n — 1

```
//create class Language
class  Language{
@Id
private int id;
private String name;
private String origin;
//getters and setters
```

id	name	Address
10	arun	Blr
20	rajath	chn

person_language

p_id	l_id
10	1
10	2
20	1
20	2
20	3

id	name	origin
1	Kannada	Karnataka
2	Telugu	Andhra
3	Tamil	Tamilnadu

OneToOne-bidirection

```
//create a class Person
@Entity
class  Person{
@Id
private int id;
private String name;
private String address;
@onetoone
private Aadhar aadhar;
//getters and setters
}
```

1 — 1

```
//create a class Aadhar
@Entity
class Aadhar{
@Id
private int id;
private String name;
private long phone;
@onetoone
private Person person;
//getters and setters
```

id	name	address	a_id
1	ram	bangalore	10
2	vijay	mumbai	20

pid	name	phone	p_id
10	gagan	987654234	1
20	ajay	768945321	2

- Inside Person table , there is a_id which acts as foreign key(FK) in the same way Inside aadhar table there is p_id which acts as foreign key (because both are owning side). To make any one as the owning side (to avoid duplicate data in db) will make use of @joincolumn on one side(owning side) mappedBy="ref"(reference of mapped entity present in owning side) on the other side(non-owning side)

OneToOne-bidirection

```
//create a class Person
@Entity
class  Person{
@Id
private int id;
private String name;
private String address;
@onetoone(cascade=Cascade
Type.ALL)
@JoinColumn
private Aadhar aadhar;
//getters and setters
}
```

1 — 1

```
//create a class Aadhar
@Entity
class Aadhar{
@Id
private int id;
private String name;
private long phone;
@onetoone(mappedBy="aadhar")
private Person person;
//getters and setters
```

id	name	address	a_id
1	ram	bangalore	10
2	vijay	mumbai	20

id	name	phone
10	gagan	987654234
20	ajay	768945321

OneToMany-bidirection/ManyToOne-bidirection

```
//create class Company
class  Company{
@Id
private int id;
private String name;
private String address;
@onetomany
private List<Employee>
employee;
//getters and setters
```

1 — n

```
//create class Employee
class Employee{
@Id
private int id;
private String name;
private long phone
@manytoone
private Company company;
//getters and setters
```

id	name	address
1	wipro	Blr

id	name	phone
10	arun	7868
20	rajath	876876

company_employee

c_id	e_id
1	10
1	20

id	name	phone	c_id
10	arun	7868	1
20	rajath	876876	1

id	name	address
1	wipro	Blr

In the previous slide ,

- when hibernate looks at the annotation @OneToMany ,it will create 3 tables one for owning side ,one for non-owning side and another for both FK's .
- When hibernate looks at the annotation @ManyToOne ,it will create 2 tables one for owning side, one for non-owning side
- Since both are owning side 3 tables will be created , one for company ,one for employee(which has c_id as FK), another one for both the FK's
- Since @ManyToOne will create 2 tables (to decrease from 3), we will make employee as owning side. To make employee as owning side we should annotate it with @JoinColumn and company as non-owning side we should make use of mappedBy="ref"(reference of mapped entity present inside owning side).

OneToMany-bidirection/ManyToOne-bidirection

```
//create class Employee
class Employee{
@Id
private int id;
private String name;
private String email;
private String password;
@ManyToOne(cascade=CascadeType.ALL)
@JoinColumn
private Company company;
//getters and setters
```

n — 1

```
//create class Company
class  Company{
@Id
private int id;
private String name;
private String address;
@onetomany(mappedBy="company")
private List<Employee>employee;
//getters and setters
```

id	name	Email	password	c_id
10	arun	a@gmail.com	123	1
20	rajath	r@gmail.com	234	2

id	name	address
1	wipro	Blr
2	infosys	chn

ManytoMany-bidirection

```
//create class Person
class Person{
@Id
private int id;
private String name;
private String address;
@ManyToMany
private Language  language;
//getters and setters
```

n — 1

```
//create class Language
class  Language{
@Id
private int id;
private String name;
private String origin;
@ManyToMany
private List<Person>person;
//getters and setters
```

id	name	address
10	arun	Blr
20	rajath	chn

id	name	origin
1	Kannada	KAR
2	Telugu	AP

p_id	l_id
10	1
10	2
20	1
20	2

person_language

id	name	origin
1	Kannada	KAR
2	Telugu	AP

id	name	address
10	arun	Blr
20	rajath	chn

l_id	p_id
1	10
1	20
2	10
2	20

language_person

In the previous slide,

- When the hibernate looks at the annotation @ManyToMany in Person class ,it will create 3 tables one for Person ,one for Language , one for person_language (FK's of both the tables)
- When the hibernate looks at the annotation @ManyToMany in Language class ,it will create 3 tables one for language ,one for person , one for language_person (FK's of both the tables)
- So in total 4 tables will be created one for Person , one for language, one for person_language, another one for language_person.
- Since person_language and language_person is same (duplicate tables) to eliminate any one of the table, we should make one as owning side that should be annotated with

```
@JoinTable(joinColumns={@JoinColumn(name="ref of owning side")},
inverseJoinColumns=@JoinColumn(name="ref of non-owning side"))
```

and mappedBy="ref of owning side" at the other side

ManytoMany-bidirection

```
//create class Person
class Person{
@Id
private int id;
private String name;
private String address;
@ManyToMany
@JoinTable(joinColumns={@JoinColumn(name="id")},inverseJoinColumns=@JoinColumn(name="cid"))
private Language  language;
//getters and setters
```

n — 1

```
//create class Language
class  Language{
@Id
private int id;
private String name;
private String origin;
@ManyToMany(mappedBy="courses")
private List<Person>person;
//getters and setters
```

id	name	address
10	arun	Blr
20	rajath	chn

person_language

p_id	l_id
10	1
10	2
20	1
20	2

id	name	origin
1	Kannada	Karnataka
2	Telugu	Andhra

FETCH TYPES

- FetchType. In general, FetchMode defines how Hibernate will fetch the data.
- FetchType defines whether Hibernate will load data eagerly or lazily.
- Fetch type supports two types of loading: **Lazy and Eager**.
- FetchType.**EAGER**: When Hibernate fetch the data from owning side, both owning side and non-owning side data will get fetched this type of Fetch Type is called as **FetchType.EAGER**.
- FetchType.**LAZY:** When Hibernate fetch the data from owning side, only owning side data will be fetched non-owning side will be not fetched so this type of FetchType is called as **FetchType.LAZY.**

MAPPING	DEFAULT FETCH TYPE
OneToOne	EAGER
OneToMany	LAZY
ManyToOne	EAGER
ManyToMany	LAZY

CASCADE

- **CASCADE:** It is the feature provided by hibernate to automatically manage the state of mapped entity(non-owning side) ,whenever the state of owner entity(owning side) is affected.
- In other words, whatever the modifications like persist, merge, delete is done at owning side will get affected to non-owning side.
- Cascading is done only at the owning side.
- To achieve cascading there are some cascade types:

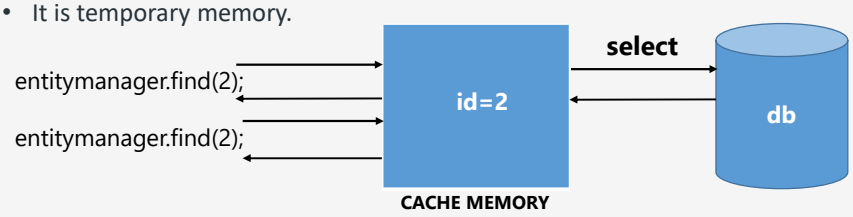
1. **CascadeType.PERSIST:** Whenever the programmer saves the owning side, automatically non-owning side will also be saved.
2. **CascadeType.MERGE: :** Whenever the programmer updates the owning side, automatically non-owning side will also be updated.
3. **CascadeType.REMOVE:** Whenever the programmer deletes the owning side, automatically non-owning side will also be deleted.
4. **CascadeType.ALL**: cascade type all is shorthand for all of the above cascade operations.

GENERATED VALUE:

- Hibernate supports some generation strategies to generate a primary key in the database table.
- **@GeneratedValue: This annotation is used to specify the primary key generation strategy to use**. i.e Instructs database to generate a value for this field automatically. If the strategy is not specified by default AUTO will be used
- @GeneratedValue annotation takes two parameters strategy and generator.
- If we want to automatically generate the primary key value, we can add the @GeneratedValue annotation. This can use four generation types: **AUTO, IDENTITY, SEQUENCE and TABLE.**
- **GenerationType.AUTO:** Indicates that the persistence provider should pick an appropriate strategy for the particular database
- **GenerationType. IDENTITY:** Indicates that the persistence provider must assign primary keys for the entity using a database identity column
- **GenerationType. SEQUENCE:** Indicates that the persistence provider must assign primary keys for the entity using a database sequence
- **GenerationType. TABLE:** Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness

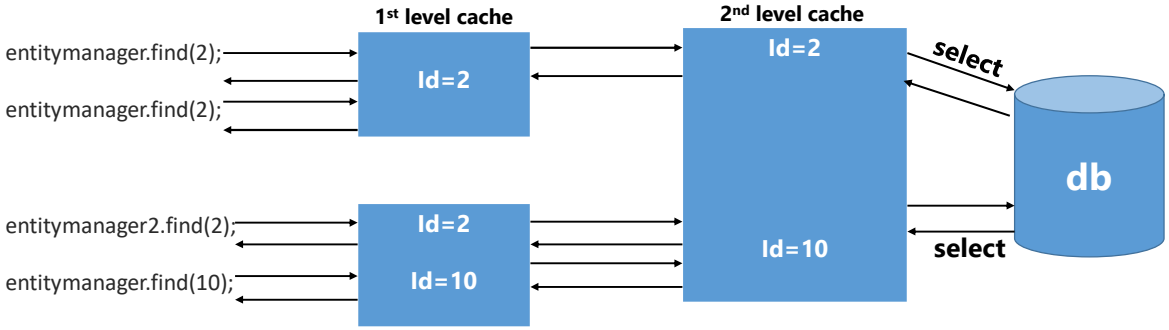
CACHING

- Caching is a mechanism to enhance the performance of a system. It is a buffer memory that lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.
- There are mainly two types of caching: **First Level Cache, and**. **Second Level Cache**
- **First Level Cache:** This is a mandatory cache which is present by default in hibernate. All the request objects passes through this cache. This cache can be utilized by application by sending many session objects. All the cache objects will be stored until one session is open. Database tries to minimize the number of hits to database in case there are many update statements commanded using session cache. Once the session is ended this cache is also cleared and the objects its holding are persisted, committed or disappeared without any updating depending upon the time of session closing.
- It is temporary memory.



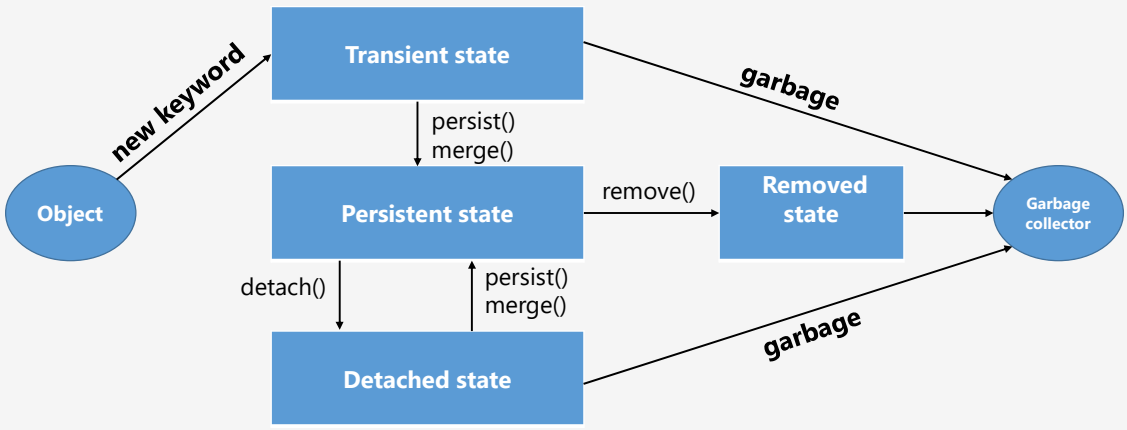
Second Level Cache:First of all, **second level** cache is not enabled by default in Hibernate you have to explicitly enable it. To enable second level cache add Ehcache dependency in pom.xml.

- The second-level cache is accessible by the entire application means data hold by SessionFactory can be accessible to all the sessions. once the session factory is closed all the cache associated with that is also removed from the memory



- When hibernate session try to load an entity, it will first find into the first-level cache, if it does not found then it will look into the second-level cache and return the response (if available), but before returning the response it will store that object/data into first-level also so next time no need to come at the session-level. When data is not found in the second level then it will go to the database to fetch data. Before returning a response to the user it will store that object/data into both levels of cache so next time it will be available at cache stages only.

LIFECYCLE OF HIBERNATE



THANK YOU