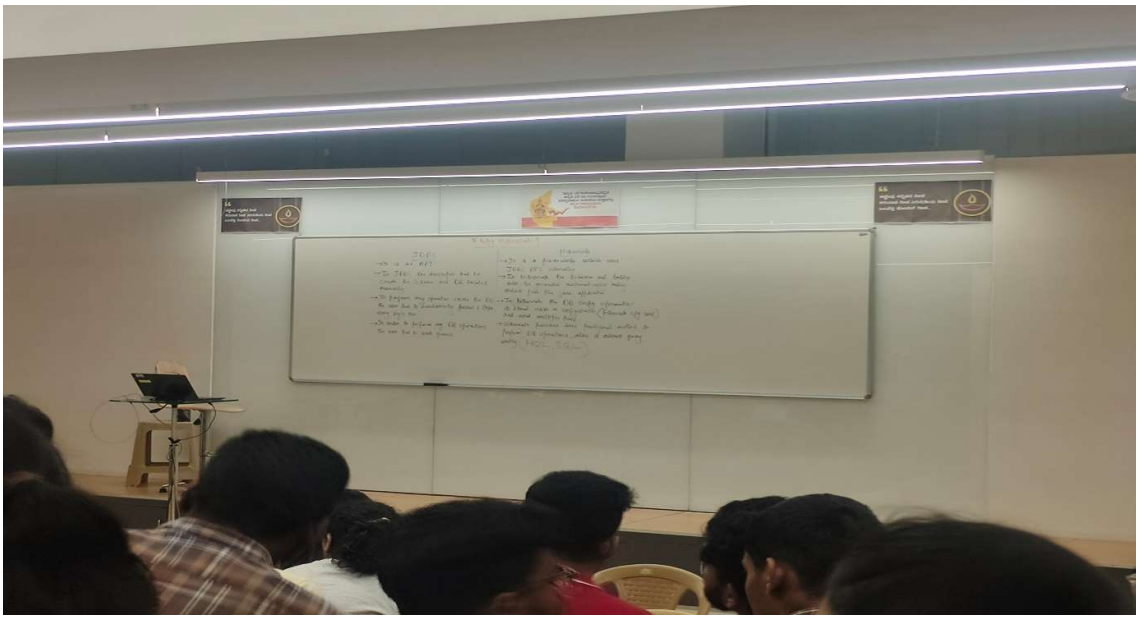


Framework:-

- A Framework is a foundational platform or structure that provides a standardized way to build and develop software applications.
- It typically includes predefined classes, functions, and tools designed to facilitate the development process and provide common functionality needed for various applications.
- It is a set of generic code or a set of generic program using which a developer can develop a software application or he can enhance the existing program or an application.

Hibernate:-

- Hibernate is an open-source Object-Relational Mapping (ORM) framework for Java, which eases the work of developers in establishing the communication wrt RDBMS.
- It simplifies the development of Java applications that interact with relational databases by providing a high-level abstraction over the low-level database interactions typically handled by JDBC.
- Hibernate maps Java classes to database tables and Java data types to SQL data types. This mapping is typically defined using annotations or XML configuration.
- Hibernate makes it easier for the developer to build relational mapping between tables, write queries, and achieve caching.

ORM :-

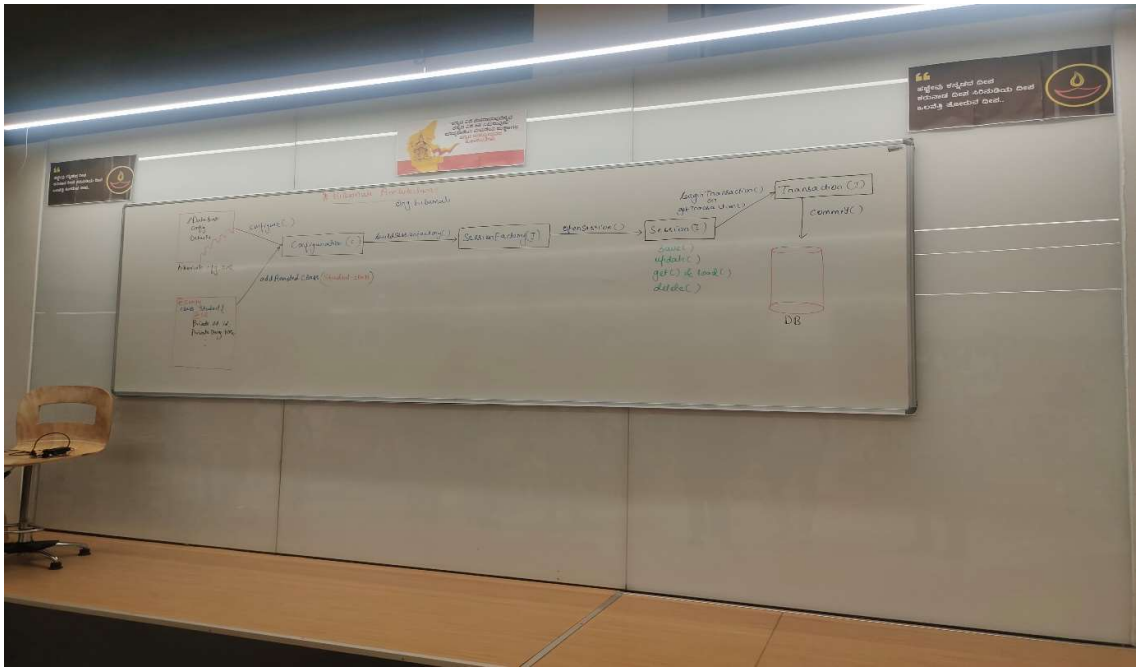
- ORM is a concept which represents the process of mapping Java objects to respective relational database tables.
- Here table name is represented by the class name and column names are represented by the attribute names.
- There are various Java frameworks that are built using the concept of ORM

  1. Hibernate

  2. Ibatis

  3.Toplink

  4.EclipseLink



Hibernate Architecture

Configuration:-

- It is a class present in org.hibernate package.
- Configuration in hibernate represents the component which is responsible to read and process the configuration information such as database, mapping and entity related information.
- We store database information in an xml file known as hibernate.cfg.xml file.
- Configuration object is responsible to read and process the above xml files in order to perform database related operations.

- Configuration object contains a method known as configure() method in order to read and process the configuration file
- configure() method is an overloaded method, The no argument configure() method reads the configuration file by default.
- In hibernate there is a provision to provide custom names to the configuration files, and in that case we need to pass the configuration file name as parameter inside the configure() method.

SessionFactory:-

- A SessionFactory in hibernate is an interface which is present in the package org.hibernate
- SessionFactory object contains compiled, immutable database and mapping information.
- We can get SessionFactory instance by invoking the buildSessionFatory() method using Configuration object.
- It is the one which is responsible to establish the connection with the specific database with the respective database details provided by Configuration.
- It is also responsible to create Sessions.

Session:-

- Session in hibernate is an interface which is present in org.hibernate package.
- Session objects are generated by invoking the openSession() method using SessionFactory Instance.
- One SessionFactory can generate n number of session objects.
- It is the one responsible to perform any database operation for the user.
- It has some inbuilt methods using which we can perform CRUD operation in the database, it is also responsible for generating Query and Criteria.

Transaction:-

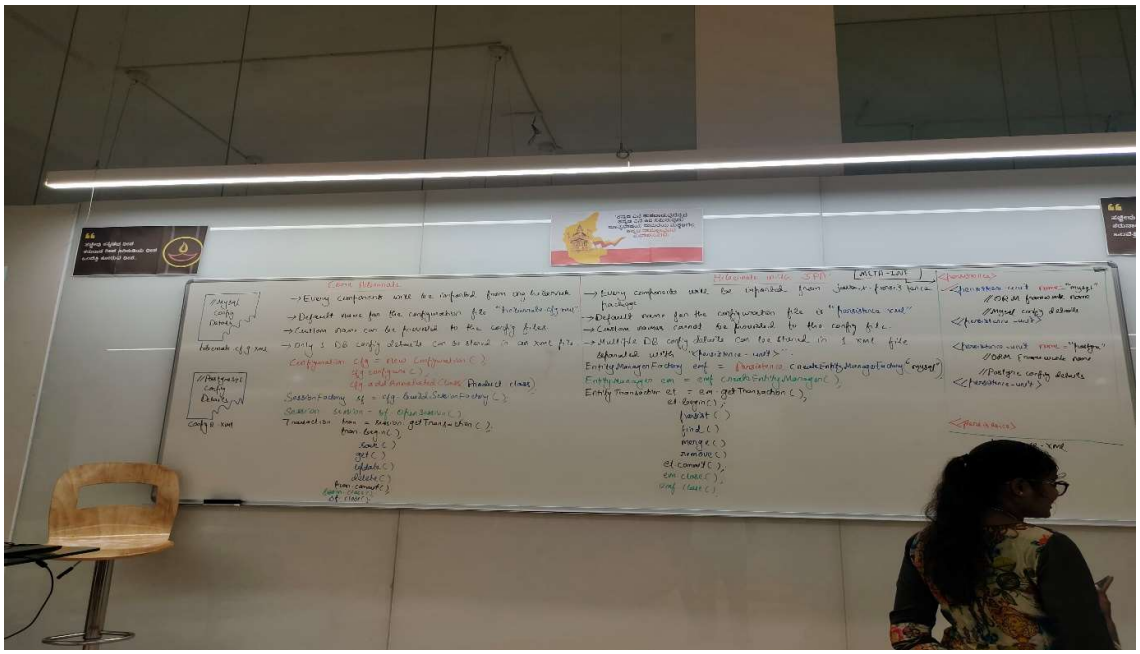
- It is also an interface present in org.hibernate package.
- Transaction Objects are generated by invoking the getTransaction() method or beginTransaction() method using the Session instance.
- This acts as a gateway which lets the user to manipulate the database table.
- When we need to perform any kind of DML operations we must have to write the logic for it within the begin() and commit() of the Transaction.

hbm2ddl.Auto:-

- The hbm2dd1.auto configuration property in hibernate configuration file is used to control the behavior of the schema generation and database manipulation.
- The "create" value represents a fresh creation of database tables with no updates of data, in other words create parameter always drops the tables if it exists and creates new table with fresh data.
- The "update" value always updates the data in the current database table without dropping it, if the table is not present it will create a fresh table and then perform the operation in the database.

Annotations:-

- @Entity - The @Entity annotation in Hibernate is used to specify that a class is an entity and is mapped to a database table.
- @Id - The @Id annotation is used to specify the primary key of an entity.
- @Table - We can specify the custom table name explicitly using the @Table annotation.
- @Column - We can customize the column names using the @Column annotation.



JPA:-

- JPA represents Java Persistence API.
- JPA only provides specifications, it doesn't provide any implementation.
- ORM frameworks like Hibenate are known for implementation of JPA.
- All the specifications of Hibernate JPA is present in javax.persistence package.

- Hibernate JPA internally uses JPQL(Java Persistence Query Language) in order to talk to the database.
- JPA uses EntityManagerFactory(I) interface in order to initiate the database connection process.
- All the configurations related to the database connectivity are stored in persistence.xml file.
- EntityManagerFactory interface helps in creating the instances of EntityManager interface, which inturn creates EntityTransaction (I) instance and performs the database related operations.

Persistence:-

- It is a class present in javax.persistence package.
- It is used as an helper class which is used to create an object for EntityManagerFactory by invoking a static method createEntityManagerFactory().

EntityManagerFactory (I):-

- It is an interface present in javax.persistence package.
- It is the one responsible for reading the configuration details from the persistence.xml file, building the connection with the respective database based on the persistence-unit name provided and creating table for the classes annotated with @Entity.
- We can create an object for EntityManagerFactory by invoking the createEntityManagerFactory() method using the Persistence class.
- It is the one which is responsible to create EntityManager instance.

EntityManager(I):-

- It is an interface present in javax.persistence package.
- We can create an object for EntityManager by invoking the createEntityManager() method using EntityManagerFactory object.
- It is similar to as that of Session in core hibernate.
- It has several in-built methods using which we can perform respective database operations, also it is the one which is used to build different types of query and Criteria as well.
- Methods to perform CRUD operation:-

  1.persist() - to insert the data

  2.find() - to fetch the data

  3.merge() - to update the data

  4.remove() - to remove the data

EntityTransaction:-

- It is also an interface present in javax.persistence package.
- It behaves as a gateway for managing transactions when interacting with a relational database.
- We can create an object for EntityTransaction by invoking the getTransaction() method using EntityManager object.

Relational Mapping:-

- Relational mapping in Hibernate involves defining the relationships between Java objects and the corresponding tables in a relational database.
- There are 4 different types of mapping in hibernate:-

  1. @OneToOne

  2. @OneToMany

  3. @ManyToOne

  4. @ManyToMany

1. @OneToOne:-

- One-to-One mapping in Hibernate establishes a relationship between two entities where one instance of an entity is associated with one and only one instance of associated entity.
- In case of uni-directional mapping the Entity class where the mapping information is provided is called the owner entity and the other entity is termed as the non-owning or the associated entity. In database there will be one extra column added to the owner entity table to represent the foreign key relationship.
- In case of bi-directional relationship both the entities will be having the mapping information so both of them are termed as the Owner entity. In database one extra column will be added to respective table of both the entities.
- While building the bi-directional relationship we can use the "mappedBy" attribute inside either of the class to reduce the lines of code in the java application and also can reduce the extra column creation in either of the table inside the database.

  Example - Person and Passport

  Student and IdCard

2. @OneToMany:-

- One-to-Many mapping in Hibernate defines a relationship where one instance of an entity is related to multiple instances of another entity.
- In case of uni-directional mapping the Entity class where the mapping information is provided is called the owner entity and the other entity is termed as the non-owning or the associated entity. In database one extra table will be created to represent the foreign key relationship between both entities.
- In case of bi-directional relationship both the entities will be having the mapping information so both of them are termed as the Owner entity. In database one extra table will be created to represent mapping information of the entity where we have annotated with @OneToMany and one extra column will be add the table of the entity where we have annotated with @Many ToOne.
- While building the bi-directional relationship we can use the "mappedBy" attribute in the @OneToMany side to reduce the lines of code in the java application and also can reduce the extra table creation inside the database.

Example - Company and Employee

Person and Vehicle

3.@ManyToOne:-

- @ManyToOne mapping in Hibernate defines a relationship where multiple instances of an entity are associated with one instance of another entity.
- This is often the inverse side of a one-to-many relationship.
- In case of uni-directional mapping the Entity class where the mapping information is provided is called the owner entity and the other entity is termed as the non-owning or the associated entity. In database one extra column will be added to the owner entity table to represent the foreign key relationship between both entities.
- In case of bi-directional it behaves exactly same as that of @OneToMany bi-directional.
- We cannot use the "mappedBy" attribute with @ManyToOne mapping.

Example - Employee and Company

Teachers and Department

4. @ManyToMany:-

- @ManyToMany mapping in Hibernate defines a relationship where multiple instances of one entity are associated with multiple instances of another entity.
- In case of uni-directional mapping the Entity class where the mapping information is provided is called the owner entity and the other entity is termed as the non-owning or the associated entity. In database one extra table will be created to represent the foreign key relationship between both entities.
- In case of bi-directional relationship both the entities will be having the mapping information so both of them are termed as the Owner entity. In database two extra table will be created to represent mapping information of the both the entities.

- While building the bi-directional relationship we can use the "mappedBy" attribute in either of the entity to reduce the lines of code in the java application and also to reduce the extra table creation inside the database.

  Example - Student and Course

  Customer and Product

mappedBy :-

- The mappedBy attribute in Hibernate is used to define the inverse side of a bidirectional relationship.
- It is used to specify the field that owns the relationship in the owning entity.
- The mappedBy attribute ensures that Hibernate understands which side of the relationship owns the foreign key which helps us to avoid duplication and maintain consistency in the database and entity relationships.

There are 3 different ways other than in-built methods to perform database operations in hibernate.

1. HQL (using Query interface)

2. SQL (using Query interface)

3. Criteria and CriteriaBuilder interface.

HQL (Hibernate Query Language)

- It is an object oriented query language which is similar to that of SQL but instead of table name and column name we use class name and variable names.
- HQL is case sensitive with respect to class names and variable names.
- HQL supports various Clauses and aggregate functions similar to as that of SQL.
- Generally, developers are recommended to use HQL because it is very close to object resemblance.

Query interface: -

- It is an interface present in javax.persistence package.
- It is the component using which we can write HQL and SQL queries in hibernate.
- If we are dealing with HQL queries we can provide implementation to Query (I) by invoking the createQuery() method using the EntityManager instance and pass the query in the parameter.
- If we are dealing with SQL queries we can provide implementation to Query(I) by invoking the createNativeQuery() method using the EntityManager instance and pass the query in the parameter.

- There are certain methods in Query interface such as:-

  1. public int executeUpdate()

  2. public List list() etc.

- We can assign placeholders for the values inside the query in two ways:-

  1. ?1 - ? with the placeholder index number

  2. :key - : with a String type key.

- The values can be set to the placeholders using the setParameter() method from Query interface.

CriteriaBuilder:-

- It is an interface present in javax.persistence.criteria package.
- It was introduced to overcome the drawbacks of Criteria API.
- Using CriteriaBuilder we can create different types of criteria in order to perform specific operations.
- CriteriaBuilder acts as a builder which is responsible to generate objects for different types of Criteria.
- We can create an object of CriteriaBuilder by invoking the getCriteriaBuilder() method using EntityManager object.
- Using CriteriaBuilder we don't need to hardcode the query as string, instead we will be using some inbuilt methods which will help us to construct the query.
- It is only used to construct the query and not responsible to execute it.
- In order to execute the query we again need to create an instance of Query interface.
- We can build 3 types of Criteria:-

  1. CriteriaQuery(I) - for retrieving the data

  2. CriteriaUpdate (I) - for updating the data

  3. CriteriaDelete(I) - for deleting the data

1. CriteriaQuery (I):-

- It is an interface present in javax.persistence.criteria package.
- It is a type of Criteria which is specifically used for data retrieval from the database.
- We can create an object of CriteriaQuery by invoking the createQuery() method using CriteriaBuilder object.
- The createQuery() method accepts the entity class name as the parameter for which it need to construct the query.
- It has a method called from() which again accepts the entity class name as the parameter in order to fetch all the records from the respective table and store it inside a root element.
- Root - It is an interface present in the javax.persistence.criteria package which is meant to hold the root object fetched form the database.
- Here the user is provided with several methods using which we can represent the where clause and various other expressions needed for our query.
- In order to execute the query after it is constructed we need to use the Query Interface in order to execute the query using respective methods (list() or getsingleResult()).

2. CriteriaUpdate (I):-

- It is an interface present in javax.persistence.criteria package.
- It is a type of Criteria which is specifically used for updating data from the database.
- We can create an object of CriteriaUpdate by invoking the createCriteriaUpdate() method using CriteriaBuilder object.
- The createCriteriaUpdate() method accepts the entity class name as the parameter for which it need to construct the query.
- It has a method called from() which again accepts the entity class name as the parameter in order to fetch all the records from the respective table and store it inside a root element.
- Then we use the set method from the CriteriaUpdate in order to set the values for column we need to update.
- Similar to that of CriteriaQuery here also the user is provided with several methods using which we can represent the where clause and various other expressions needed for our query.
- In order to execute the query after it is constructed we need to use the Query interface in order to execute the query using executeUpdate() method.

3. CriteriaDelete(I):-

- It is an interface present in javax.persistence.criteria package.
- It is a type of Criteria which is specifically used for deleting data from the database.
- We can create an object of CriteriaDelete by invoking the createCriteriaDelete() method using CriteriaBuilder object.
- The createCriteriaDelete() method accepts the entity class name as the parameter for which it need to construct the query.
- It has a method called from() which again accepts the entity class name as the parameter in order to fetch all the records from the respective table and store it inside a root element.
- Similar to that of CriteriaQuery and CriteriaUpdate here also the user is provided with several methods using which we can represent the where clause and various other expressions needed for our query.
- Here as well in order to execute the query after it is constructed we need to use the Query interface in order to execute the query using executeUpdate() method.

FetchType:-

- FetchType in hibernate is an enum that defines the fetching strategy for associations between entities.
- It can be added to any kind of mapping. It determines how mapped/associated entities should be loaded from the database.
- By using FetchType developers can control the loading behavior of associated entities and improve the efficiency of their applications.
- There are two types of fetching strategies:

1. FetchType.EAGER
2. FetchType.LAZY

FetchType.EAGER:-

- It is a fetching strategy that indicates that the associated entities should be fetched immediately along with their Owner entity.
- Hibernate will retrieve the related entities In the same query.
- It is used when the associated entities are always needed along with the owner entity.
- @OneToOne and ManyToOne are eager loaders by default.

FetchType.LAZY:-

- It is a fetching strategy that indicates that the associated entities are not loaded from the database immediately when the owner entity is fetched.
- In other words the associated entities data will not be fetched unless it is requested.
- Using this we can improve performance of the application, especially when dealing with large datasets.
- @OneToMany and @ManyToMany are lazy loaders by default.

Hibernate Caching:-

- Hibernate caching is a mechanism which is built to improve the performance of the application, by reducing the number of database queries, reduce number of database hits.
- There are two different types of caching in hibernate: -

1. First level cache or level 1 cache
2. Second level cache or level 2 cache

1. First level cache:-

- Hibernate first level cache is a default caching mechanism in hibernate.
- It is associated with EntityManager objects.
- The first level cache is created when the EntityManager is opened and destroyed when the EntityManager is closed, in other words the objects that are cached in first level cache of EntityManager1 is not visible to first level cache of any other EntityManagers.
- There is a provision to delete or remove the objects from the cache memory, once the EntityManager object is closed all the data present in 1st level cache will be lost.

2.Second level cache

- Second level or level 2 cache memory is another type of caching mechanism in hibernate.
- Second level cache memory is always associated with EntityManagerFactory. It is not enabled by default and needs to be explicitly configured.
- The second level cache memory is always shared among all the EntityManagers created using 1 EntityManagerFactory.
- EntityManagerFactory is solefully responsible for creation and destruction of second level cache memory.
- From EntityManagerFactory point of view second level cache memory is the point of contact

Working of 2nd level cache:-

- When the client triggers a request for an Object from the EntityManager point of view it always checks for 1st level cache memory, if the data is not present in 1st level cache memory then the EntityManager looks for second level cache memory.
- If the data is not present even in second level cache memory then a query gets fired to the database server and correspondingly the response copy will be stored in both 2nd level as well as 1st level cache memory and data will be given to the java application, and next time if the same Object is requested from the same EntityManager it will directly return the data from the 1st level cache of that EntityManager.
- Under the same EntityManagerFactory if a new EntityManager gets created and try to access the same data as previous EntityManager, then the EntityManager object first looks for 1st level cache of its own, if the data is not present then it looks for second level cache memory, and since a copy of the requested data is present in 2nd level cache memory because of the previous request, a copy of the same data will be given to the 1st level cache of the current EntityManager and returns the data to the java application.

Steps to enable 2nd level cache:-

1. Download hibernate-ehcache dependency from maven.

2. Add a property tag in persistence.xml in order to enable 2nd level cache memory and to specify RegionFactory class of hibernate ehcache dependency.

3. The RegionFactory class acts as a bridge between the hibernate framework and the cache providers.

4. Hibernate framework will not have any information related to cache providers. Hence, RegionFactory class acts as a bridge.

5. Add @Cacheble and @Cache annotations with concurrency strategy on top of each entity classes that has to be part of 2nd level cache memory.

=> @Cacheble

- It is an annotation present in javax.persistence package.
- This annotation is used in the context of the second-level cache to indicate that the results of a query or method should be cached.
- It is commonly used with entity classes to mark them as cacheable.

=> @Cache

- This annotation is also used in the context of the second-level cache and works in conjunction with @Cacheable.
- It provides additional configuration for the caching behavior, such as the cache concurrency strategy.
- The usage attribute specifies how the cache should handle concurrent access to cached data. Common strategies include READ_ONLY, READ_WRITE, NONSTRICT_READ_WRITE, and TRANSACTIONAL.