UNIT – II

ORM & Hibernate: What is Object Relational Mapping, How ORM Works, Features of ORM, Advantages, Java ORM- Hibernate, JAVA Persistence API(JPA), ORM implementation.

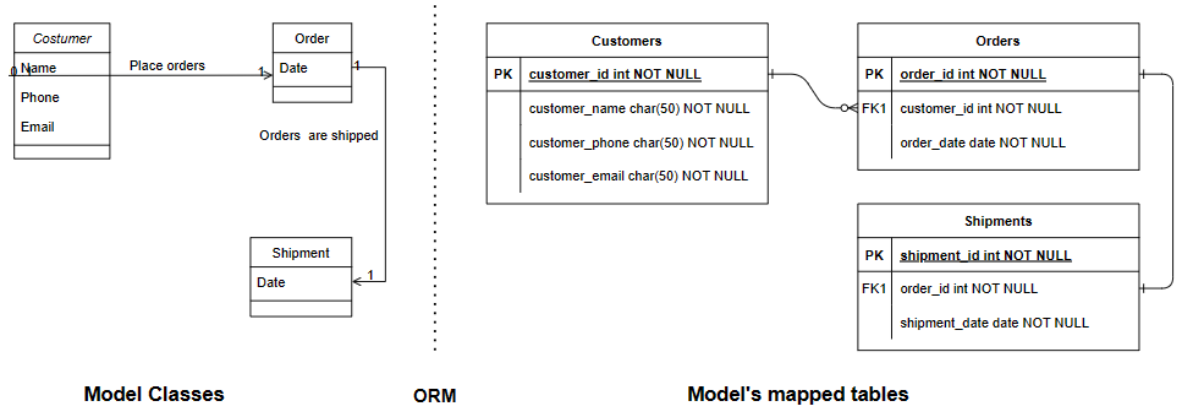
Hibernate: Overview of Hibernate, Hibernate Architecture, Hibernate Mapping Types, Hibernate O/R Mapping, Hibernate Annotation, Hibernate Query Language

Object-Relational Mapping (ORM):

Definition: Object-Relational Mapping (ORM) is a programming paradigm that enables developers to work with relational databases using object-oriented programming languages, bridging the gap between the two paradigms.

Designing and deploying information systems nowadays can be truly a hard task. In most cases, we must know multiple programming languages and frameworks. Every application layer has its own tech stack. And, while some evolve very fast, others use decades-old technology. For instance, in most corporations' old-school relational databases are the data persistence standard still, and rightly so.

An Object-Relational Mapping tool, ORM, is a framework that can help and simplify the translation between the two paradigms: objects and relational database tables. It can use class definitions (models) to create, maintain and provide full access to objects' data and their database persistence. The figure shows an extract of how an ORM would map a set of classes into relational tables:



Impedance Mismatch between Object Model and Relational Model

Impedance mismatch refers to the inherent differences between the relational model

used by databases and the object-oriented model used in programming languages. In the context of Object-Relational Mapping (ORM), impedance mismatch becomes evident when trying to bridge the gap between these two different paradigms. ORM frameworks like Hibernate and technologies like the Java Persistence API (JPA) aim to address this mismatch.

Key Aspects of Impedance Mismatch:

1. **Data Models:**
    - Relational Model: Relational databases organize data in tables with rows and columns, representing entities and their attributes.
    - Object-Oriented Model: Programming languages like Java use classes and objects to model entities with properties and behaviors.
2. **Data Types:**
    - Relational Model: Databases have specific data types for columns, such as integers, strings, dates, and more.
    - Object-Oriented Model: Programming languages have a wider range of data types, and developers can define custom classes and data structures.
3. **Data Relationships:**
    - Relational Model: Relationships between tables are established using foreign keys and primary keys.
    - Object-Oriented Model: Relationships are established using references or pointers between objects.
4. **Inheritance:**
    - Relational Model: Inheritance is not directly supported in most relational databases.
    - Object-Oriented Model: Inheritance is a fundamental concept in object-oriented programming, allowing classes to inherit properties and behaviors from parent classes.
5. **Identity and Identity Management:**
    - Relational Model: Databases often use auto-incrementing primary keys to uniquely identify rows.

- Object-Oriented Model: Objects can have identity based on their memory address or other attributes.

Impedance Mismatch Challenges:

1. **Data Conversion:** ORM frameworks need to convert data between object-oriented representations and relational structures. This process can lead to performance overhead.
2. **Complex Relationships:** Mapping complex relationships (many-to-many, one-to-one, etc.) between objects and tables can be challenging.
3. **Inheritance Mapping:** Mapping inheritance hierarchies from object-oriented models to relational databases can result in complex and less efficient structures.
4. **Performance Overhead:** ORM frameworks often introduce additional layers of abstraction, which might impact performance compared to writing raw SQL queries.
5. **Mapping Complexity:** Complex database structures might not map neatly to object-oriented models, requiring developers to create workarounds or custom mappings.

ORM as a Solution:

ORM frameworks like Hibernate and JPA attempt to alleviate the impedance mismatch by providing mechanisms to map object-oriented concepts to relational database structures:

1. **Mapping Annotations:** ORM frameworks use annotations to define how classes, attributes, and relationships map to database tables, columns, and keys.
2. **Query Languages:** ORM frameworks often introduce their query languages (e.g., HQL in Hibernate) that let developers work with objects instead of tables.
3. **Caching:** ORM frameworks implement caching mechanisms to optimize data retrieval and reduce database round-trips.
4. **Lazy Loading:** ORM frameworks offer lazy loading to fetch related data only when needed, addressing the performance impact of complex joins.
5. **Inheritance Strategies:** ORM frameworks provide various strategies for mapping inheritance hierarchies to tables.

While ORM provides solutions to impedance mismatch, it's essential for developers to understand both the object-oriented and relational paradigms to make informed decisions when

designing and implementing applications with ORM frameworks.

How ORM Works:

1. **Mapping Configuration:** Developers define how classes and attributes in their application map to tables and columns in the database. This is often done using annotations or XML configuration files.
2. **Data Access Layer:** ORM frameworks provide APIs to interact with the database. Developers use these APIs to perform CRUD operations and query data without writing raw SQL queries.
3. **Object-Relational Transformation:** When an object is saved to the database, the ORM framework generates the necessary SQL statements to insert or update rows in the corresponding table. Similarly, when objects are retrieved, the ORM framework converts database records into objects.
4. **Lazy Loading:** ORM frameworks allow lazy loading of related objects. This means that related objects are only loaded from the database when accessed, reducing unnecessary data retrieval.

Features of ORM:

1. **Mapping Annotations:** ORM frameworks offer annotations (such as **@Entity**, **@Table**, **@Column**) to define how classes and attributes are mapped to the database schema.
2. **Automatic Query Generation:** ORM tools generate SQL queries based on method calls, allowing developers to focus on business logic instead of query syntax.
3. **Caching Mechanisms:** ORM frameworks often include caching mechanisms to store frequently accessed data in memory, reducing database round-trips and improving performance.
4. **Lazy Loading:** As mentioned earlier, lazy loading helps load related data only when needed, enhancing efficiency.
5. **Transaction Management:** ORM frameworks handle transactions, ensuring that a series of database operations are either fully completed or fully rolled back in case of errors.
6. **Database Agnostic:** Developers can switch between different databases without

altering the application code, thanks to the abstraction provided by ORM.

7. **Object-Oriented Inheritance:** ORM supports object-oriented concepts like inheritance, allowing developers to model class hierarchies that map to the database.
8. **Schema Evolution:** ORM frameworks offer mechanisms to manage database schema changes as the application's object model evolves.

Advantages of ORM:

1. **Increased Productivity:** Developers focus on business logic rather than writing low-level SQL queries.
2. **Database Portability:** Applications can be developed using one database and later switched to another without changing code, easing migrations.
3. **Maintainability:** Changes in the application's object model are automatically translated to database schema changes, reducing maintenance effort.
4. **Security:** ORM frameworks often provide safeguards against SQL injection attacks by sanitizing inputs and parameterizing queries.
5. **Object-Oriented Paradigm:** Developers work with familiar object-oriented concepts, improving code readability and maintainability.

Java ORM - Hibernate:

Hibernate is a powerful and widely-used Java ORM framework. Some of its features include:

1. **Session Factory:** Hibernate's central component responsible for creating and managing sessions, which represent a connection to the database.
2. **Entity Mapping:** Developers define entity classes and annotate them to specify how they map to database tables and columns.
3. **Hibernate Query Language (HQL):** Similar to SQL, HQL is used to query objects instead of tables, allowing developers to perform complex queries using an object-oriented syntax.
4. **Caching:** Hibernate supports first-level caching (session-level) and second-level caching (application-level) to improve performance.
5. **Association Mapping:** Hibernate supports various types of associations between entities, including one-to-one, one-to-many, and many-to-many relationships.

6. **Inheritance Mapping:** Hibernate supports various strategies for mapping object-oriented inheritance hierarchies to database tables.

Java Persistence API (JPA):

JPA is a specification for ORM in Java EE applications. It defines standard interfaces and annotations for ORM in Java. Hibernate is one of the implementations of JPA. Key concepts include:

1. **EntityManager:** The core interface for JPA operations, allowing developers to manage entities, perform CRUD operations, and execute queries.
2. **Annotations:** JPA provides annotations (**@Entity**, **@Table**, **@Column**, etc.) to define mappings and relationships in entity classes.
3. **Persistence Unit:** Configuration information is defined in the **persistence.xml** file, including database connection details and the entity manager factory.

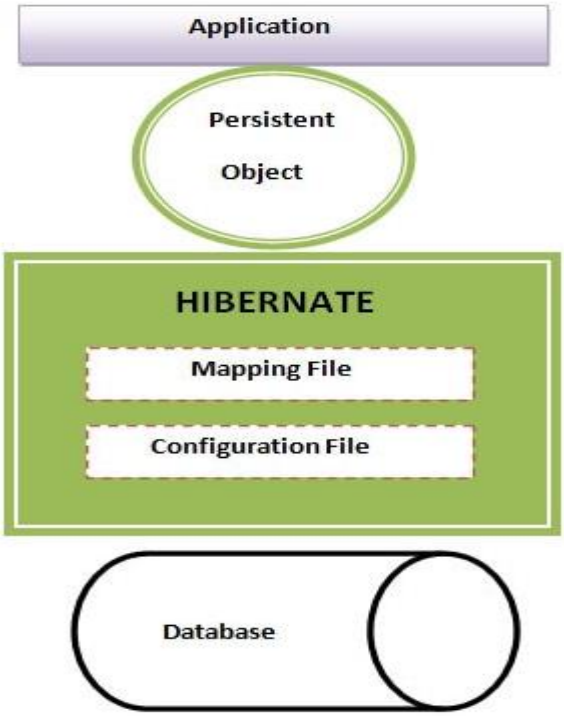
ORM Implementation:

1. **Entity Classes:** Define Java classes and annotate them with **@Entity**. Define attributes and annotate them with **@Column** to specify column mappings.
2. **Persistence Unit:** Configure the persistence unit in the **persistence.xml** file. Define the database connection, ORM provider, and entity classes.
3. **EntityManager:** Obtain an instance of **EntityManager** from the **EntityManagerFactory** to perform database operations.
4. **Transactions:** Use **@Transactional** annotations or programmatically manage transactions using the **EntityManager**'s transaction API.
5. **Querying:** Utilize JPQL or native SQL queries for retrieving data from the database.
6. **Caching Configuration:** Configure caching settings at the session or entity level to optimize data retrieval.

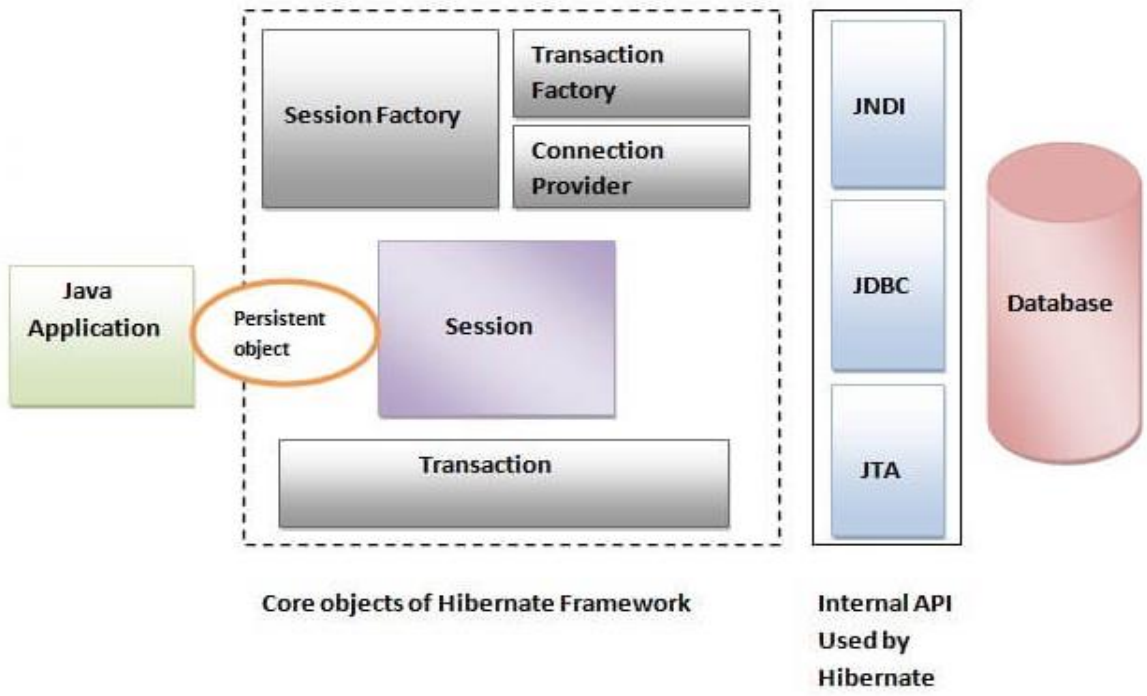
Hibernate:

Overview: Hibernate is a popular Java-based Object-Relational Mapping (ORM) framework that simplifies database access by providing an abstraction layer over JDBC (Java Database Connectivity). It enables developers to work with databases using object-oriented concepts, eliminating the need to write complex SQL queries.

Hibernate Architecture:



1. **Application Layer:** This is where your application code resides. It interacts with Hibernate through the Hibernate API.
2. **Hibernate Core:** The core components of Hibernate include the SessionFactory, Session, and Transaction. The SessionFactory is responsible for creating sessions, which are used to interact with the database. Transactions manage the unit of work.
3. **Mapping Metadata:** Developers define mapping metadata using XML configuration or annotations. This metadata specifies how Java classes and objects map to database tables and columns.
4. **Connection Provider:** Hibernate uses a connection provider to manage database connections. It abstracts away the low-level details of connection management.
5. **Dialect:** The dialect translates Hibernate's HQL (Hibernate Query Language) queries into database-specific SQL queries.
6. **Caching:** Hibernate provides caching mechanisms, including first-level (session-level) and second-level (application-level) caching, to improve performance.
7. **Query Execution:** HQL queries are converted to SQL queries by Hibernate and executed against the database. Results are then transformed back into Java objects.



Hibernate Mapping Types:

Hibernate provides various mapping types to map Java data types to corresponding database types. Examples include:

- **StringType**: Maps Java strings to database VARCHAR columns.
- **IntegerType**: Maps Java integers to database INT columns.
- **DateType**: Maps Java dates to database DATE or TIMESTAMP columns.

Hibernate Object/Relational Mapping (O/R Mapping):

O/R Mapping is the process of mapping object-oriented concepts to relational database structures. Hibernate achieves this through entity classes and their corresponding database tables. Key aspects include:

- **Entity Classes:** Java classes annotated with **@Entity** that represent entities to be stored in the database.
- **Attributes:** Class properties annotated with **@Column** to define their mapping to database columns.
- **Associations:** Relationships between entities, such as **@OneToOne**, **@OneToMany**, **@ManyToOne**, and **@ManyToMany**.

Hibernate Annotations:

Hibernate supports annotations to define mapping metadata within the Java code itself. Common annotations include:

- **@Entity**: Specifies a class as an entity.
- **@Table**: Specifies the database table name for the entity.
- **@Column**: Maps a class attribute to a database column.
- **@Id**: Marks a property as the primary key.

Hibernate Query Language (HQL):

HQL is a powerful query language provided by Hibernate that allows developers to write queries using object-oriented concepts, treating tables as entities and rows as objects. It abstracts the underlying database and allows for querying directly against entity classes. Some key features of HQL include:

- Supports SELECT, UPDATE, DELETE queries.
- Includes support for filtering, sorting, and joining.
- Provides aggregate functions, subqueries, and parameter binding.