Object/Relational Mismatch

Object/Relational Mismatch refers to the challenges and discrepancies that can arise when dealing with the differences between **object oriented programming (OOP) and relational database management systems** (**RDBMS**). **OOP and RDBMS** are two distinct paradigms for representing and managing data, and when trying to **bridge the gap between them, certain mismatches can occur.** Here are some common aspects of **Object/Relational** Mismatch:

1. Data Model Differences:

- In **OOP**, data is organized into objects with properties (attributes) and behaviors (methods), while in **RDBMS**, data is organized into tables with rows and columns.
- **OOP** allows for complex relationships, inheritance, and polymorphism, which are not directly supported in traditional relational databases.

2. Data Type Mismatch:

- **OOP languages** often have a richer set of data types compared to relational databases. For example, **OOP** supports complex types, while **relational databases** typically have simpler types.

3. Identity and Identity Mapping:

- **Objects in OOP** have identity, and the identity of an object is usually determined by its memory address. In **relational databases,** identity is typically represented by **primary keys.**
- **Object relational mapping (ORM)** frameworks attempt to map objects to **database tables**, and this process may involve challenges in managing identity and identity mapping.

4. Navigational vs. Declarative Approach:

- **OOP** relies on a navigational approach, where relationships between objects are navigated directly through references. In contrast, **relational databases** use a declarative approach with queries expressed in SQL to fetch related data.

5. Performance Considerations:

- **Object oriented systems** may need to perform multiple queries to retrieve and assemble related data, which can lead to performance issues when compared to the efficiency of **relational databases** handling joins.

6. Concurrency Control:

- Managing concurrent access to data can be handled differently in **OOP and RDBMS**. **OOP** often relies on mechanisms like locks at the application level, while **RDBMS** uses database level locks and transactions.

To mitigate these mismatches, developers often use **Object Relational Mapping (ORM) frameworks, such as Hibernate, JPA (Java Persistence API), Entity Framework (for .NET)**, or **others**. These frameworks aim to provide a seamless integration between the **object oriented code and relational databases**, handling the translation between the two paradigms and addressing the challenges posed by the **Object/Relational Mismatch**.

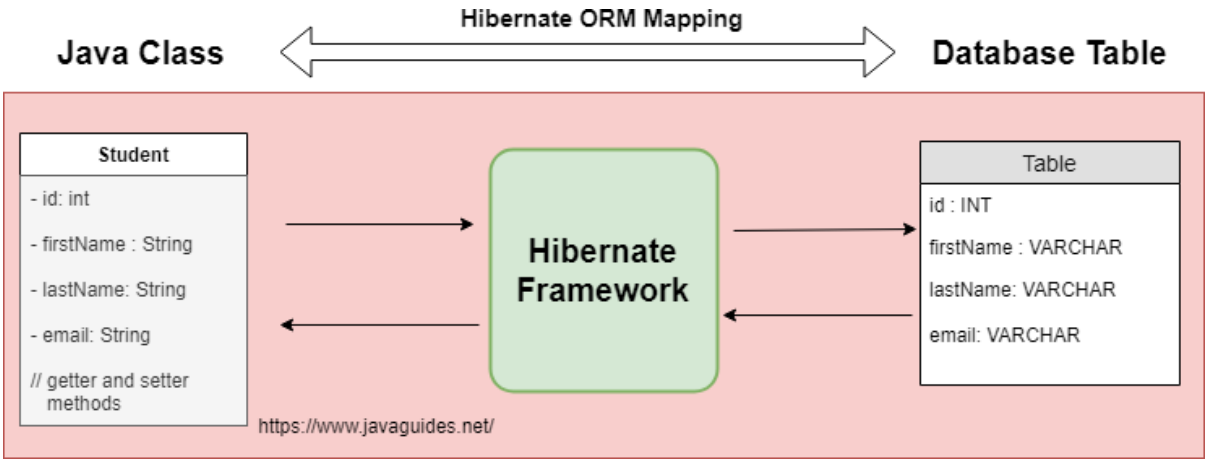
Hibernate Architecture and API

Developed by **Gavin King and first released in early 2002**, Hibernate is an ORM library for Java. King developed Hibernate as an alternative to entity beans for persistence.

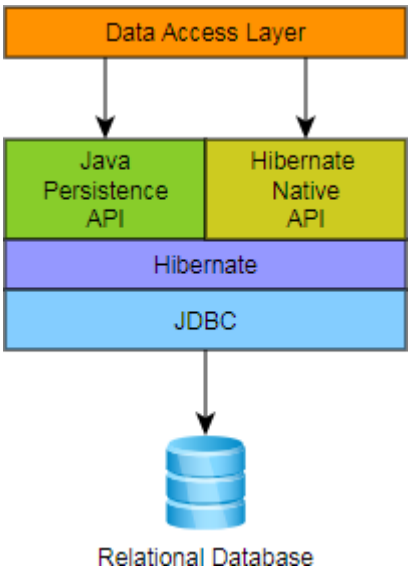
Hibernate is a java based ORM tool that provides a framework for mapping application domain objects to the relational database tables and vice versa.

Hibernate is probably the most popular JPA implementation and one of the most popular Java frameworks in general. Hibernate acts as an additional layer on top of JDBC and enables you to implement a database-independent persistence layer. It provides an object-relational mapping implementation that maps your database records to Java objects and generates the required SQL statements to replicate all operations to the database.

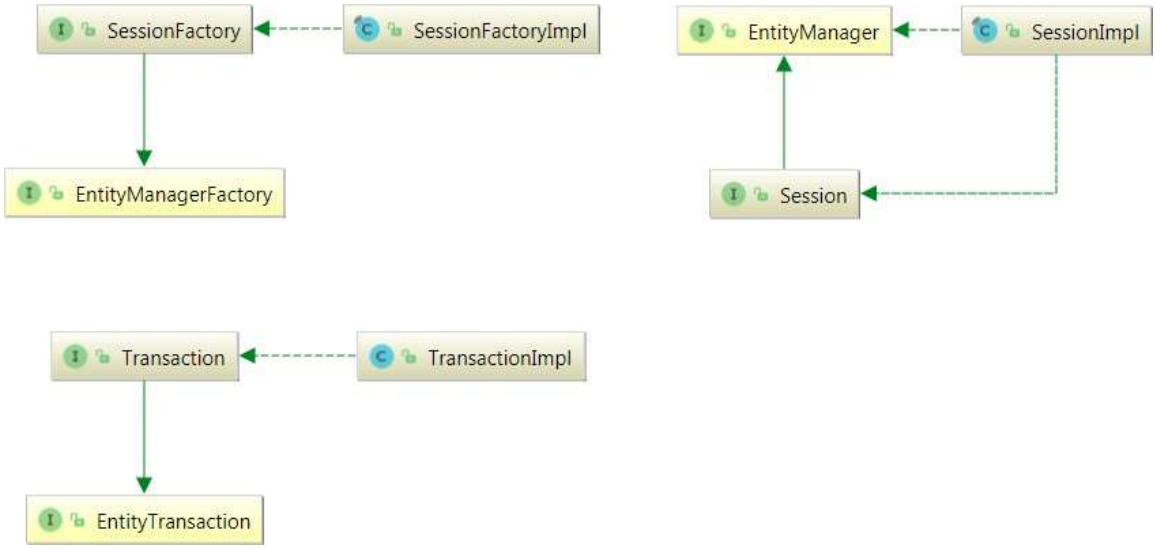
Example: Below diagram shows an *Object Relational Mapping* between **Student** Java class and **student** table in the database



Architecture



Hibernate, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database, as can be seen in the diagram above. The Java application makes use of the Hibernate APIs to load, store, query, etc its domain data. Here we will introduce the essential Hibernate APIs.



JPA Introduction

The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like **Hibernate, TopLink and iBatis** implements JPA specifications for data persistence.

Hibernate and JPA relationship

As a JPA provider, **Hibernate** implements the *Java Persistence API* specifications and the association between JPA interfaces and Hibernate specific implementations can be visualized in the following diagram:

SessionFactory (org.hibernate.SessionFactory)
A thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for *org.hibernate.Session* instances. The *EntityManagerFactory* is the JPA equivalent of a *SessionFactory* and basically, those two converge into the same *SessionFactory* implementation.

A *SessionFactory* is very expensive to create, so, for any given database, the application should have only one associated SessionFactory. The *SessionFactory* maintains services that Hibernate uses across all *Session*(s) such as second-level caches, connection pools, transaction system integrations, etc.

Session (org.hibernate.Session)

A single-threaded, short-lived object conceptually modeling a "Unit of Work". In JPA nomenclature, the *Session* is represented by an *EntityManager*.

Behind the scenes, the Hibernate Session wraps a JDBC *java.sql.Connection* and acts as a factory for *org.hibernate.Transaction* instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (org.hibernate.Transaction)

A single-threaded, short-lived object is used by the application to demarcate individual physical transaction boundaries. *EntityTransaction* is the JPA equivalent and both acts as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).