REVATURE

Introduction To Hibernate 5.5.Final

ORM

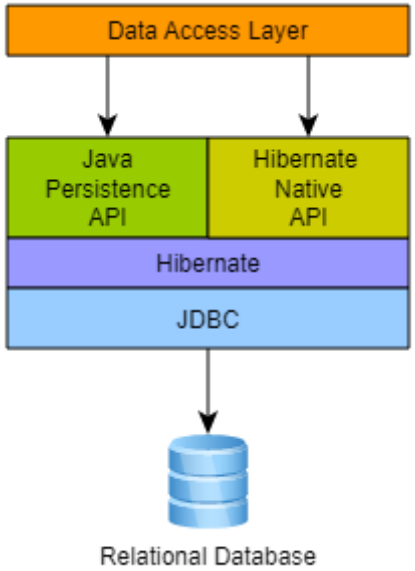
Object-Relational Mapping (ORM) is the process of converting Java objects to database tables.
In other words, this allows us to interact with a relational database without any SQL.
The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications.
The primary focus of JPA is the ORM layer.

Hibernate is one of the most popular Java ORM frameworks.
Its first release was almost twenty years ago, and still has excellent community support and regular releases.
Additionally, Hibernate is a standard implementation of the JPA specification, with a few additional features that are specific to Hibernate.

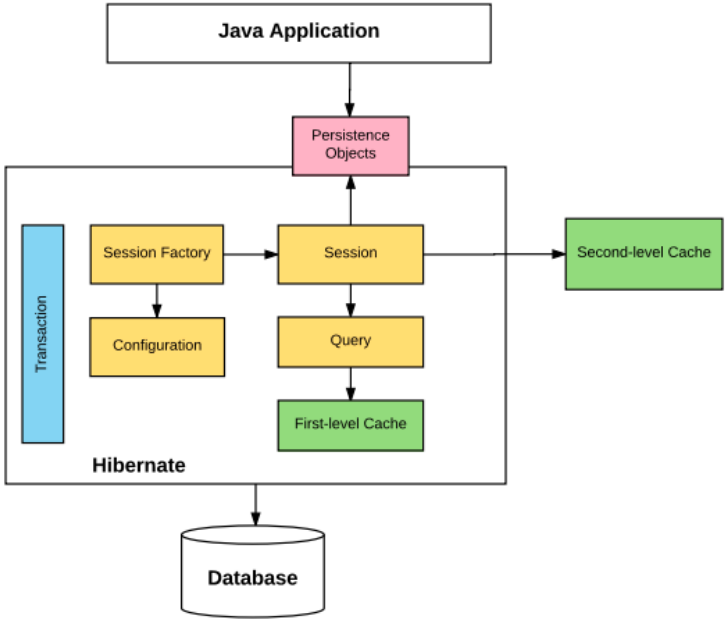
https://hibernate.org/orm/releases/5.6/

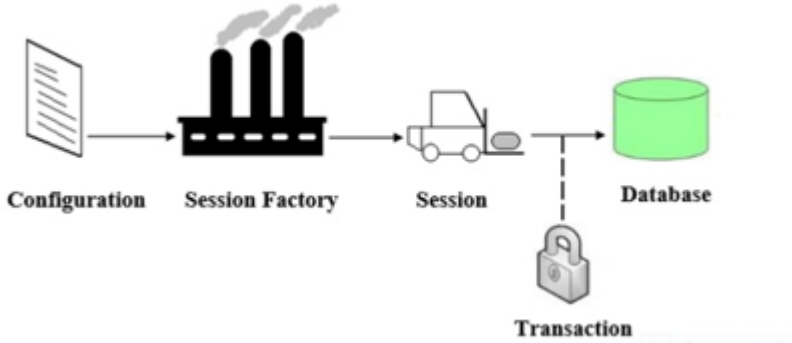
Architecture

Hibernate, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database. The Java application makes use of the Hibernate APIs to load, store, query, etc. its domain data.



Architecture





SessionFactory, Session, Transaction

REVATURE

SessionFactory *is an interface.*

- *SessionFactory can be created by providing Configuration object, which will contain all DB related property details pulled from either hibernate.cfg.xml file or hibernate.properties file.*
- *SessionFactory is a factory for Session objects.*
- *We can create one SessionFactory implementation per database in any application.*
- *If your application is referring to multiple databases, then you need to create one SessionFactory per database.*
- *The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use.*
- *The SessionFactory is a thread safe object and used by all the threads of an application.*
- *Session Factory can be considered as the factory which will yield us the sessions for a particular datasource or database. In other words, if our application has more than one database, then we should create as many session factories as are our number of databases. So, Session factory is long-lived.*

Session *is an interface*

- *It is used to get a physical connection with a database.*
- *The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database.*
- *Persistent objects are saved and retrieved through a Session object.*
- *The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.*
- *The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.*
- *Session is the short lived instance used to perform discrete transactions to the database. Usually, at any time, if a transaction is required to be done with the database, a short lived session object is obtained from the appropriate Session Factory instance and after the transaction is completed, the instance is no more available.*
- *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.*

SessionFactory, Session, Transaction

REVATURE

Interface SessionFactory (org.hibernate.SessionFactory):

1. *It is one intance per datasource/database.*
2. *It is thread-safe.*
3. *It is a heavyweight object, because it maintains data sources, mappings, hibernate configuration information etc.*
4. *Sessionfactory will create and manage the sessions.*
5. *If you have say, 4 datasources/databases, then you must create 4 session factory instances.*
6. *sessionfactory is an immutable object and it will be created as a singleton while the server initializes itself.*

Interface Session (org.hibernate.Session):

1. *It is one instance per client/thread/one transaction.*
2. *It is not thread-safe.*
3. *It is lightweight.*
4. *sessions will be opened using sessionfactory.openSession() and some database operations will be done finally session will be closed using session.close().*

Interface Transaction (org.hibernate.Transaction)

1. *A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries.*
2. *EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).*
3. *Transaction will be opened using **session.beginTransaction()** and committed using **session.getTransaction().commit()** & rollbacked using **session.getTransaction().rollback()***

Hibernate Type

REVATURE

Hibernate understands both the Java and JDBC representations of application data.
The ability to read/write this data from/to the database is the function of a Hibernate type.
A type, in this usage, is an implementation of the org.hibernate.type.Type interface.
This Hibernate type also describes various behavioral aspects of the Java type such as how to check for equality, how to clone values, etc.
The Hibernate type is neither a Java type nor a SQL data type. It provides information about mapping a Java type to an SQL type as well as how to persist and fetch a given Java type to and from a relational database.
When you encounter the term type in discussions of Hibernate, it may refer to the Java type, the JDBC type, or the Hibernate type, depending on the context.
In the broadest sense, Hibernate categorizes types into two groups:

- ***Value types***
- ***Entity types***

Value & Entity Type

A value type *is a piece of data that does not define its own lifecycle. It is, in effect, owned by an entity, which defines its lifecycle.*
Looked at another way, all the state of an entity is made up entirely of value types.
These state fields or JavaBean properties are termed persistent attributes. The persistent attributes of the Contact class are value types.
Value types are further classified into three sub-categories:
Basic types
-in mapping the Contact table, all attributes except for name would be basic types. Basic types are discussed in detail in Basic types
Embeddable types
-the name attribute is an example of an embeddable type, which is discussed in details in Embeddable types.
Collection types
although not featured in the aforementioned example, collection types are also a distinct category among value types. Collection types are further discussed in Collections

Entity types
Entities, by nature of their unique identifier, exist independently of other objects whereas values do not. Entities are domain model classes which correlate to rows in a database table, using a unique identifier. Because of the requirement for a unique identifier, entities exist independently and define their own lifecycle. The Contact class itself would be an example of an entity.
Mapping entities is discussed in detail in Entity types.

```
create table Contact (
    id integer not null,
    first varchar(255),
    last varchar(255),
    middle varchar(255),
    notes varchar(255),
    starred boolean not null,
    website varchar(255),
    primary key (id)
)
```

```
@Entity(name = "Contact")
public static class Contact {

        @Id
        private Integer id;

        private Name name;

        private String notes;

        private URL website;

        private boolean starred;

        //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

        private String firstName;

        private String middleName;

        private String lastName;

        // getters and setters omitted
}
```

@org.hibernate.annotations.Type(type="uuid-char")

REVATURE

- *There are three levels of data types:*
  - *Java types*
  - *Hibernate's types*
  - *Database Specific types.*
- *Hibernate data type presentation is a bridge between Java data type and Database types to be independent from database.*
- *You can check this mappings. As you can find there java.util.UUID can be mapped to diferent types (binary or char/varchar). uuid-binary is key to hibernate's UUIDBinaryType, you get this type by default and it will be mapped to BINARY of your database.*
- *If you want to get CHAR type under your UUID, you should explain to hibernate that you want his UUIDCharType. To do that you use uuid-char key and as you can check in JavaDoc of @Type annotation: Defines a Hibernate type mapping.. So, you use annotation to explain hibernate which bridge it should use.*

SessionFactory Configuration

REVATURE

Key points of above configuration for getting object of sessinoFactory in Hibernate are as:

- *The **dialect** property informs hibernate to generate database specific (MySQL here) instructions.*
- ***driver_class** defines the database specific driver hibernate will use to make the connection.*
- ***username,password** & **url** are general connection properties, nothing special.*
- ***show_sql** will instruct hibernate to log all the statements on console and **format_sql** instructs it to display properly formatted SQL.*
- ***mapping** tag instructs hibernate to perform mapping for mentioned **resource**(in case of XML Mapping) or **class**(in case of Annotation mapping).*

Three Types Of Configurations-

1. *Using **hibernate.cfg.xml** file*
2. *Using **hibernate.properties** file*
3. *Programmatically*

Automatic DB Schema Generation

hibernate.hbm2ddl.auto *possible options are,*

- ***validate:*** *validate the schema, makes no changes to the database.*
- ***update:*** *update the schema.*
- ***create:*** *creates the schema, destroying previous data.*
- ***create-drop:*** *drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped. (Preferably use this for development)*
- ***none:*** *does nothing with the schema, makes no changes to the database*

Note: These options seem intended to be developer's tools and not to facilitate any production level database.
Hibernate creators discourage doing so in a production environment in their book "Java Persistence with Hibernate":

> WARNING: We've seen Hibernate users trying to use SchemaUpdate to update the schema of a production database automatically. This can quickly end in disaster and won't be allowed by your DBA.

Hibernate Configuration With & Without Using XML File

REVATURE

*Demo using **hibernate.cfg.xml** file*

*Demo using **hibernate.properties** file*

Programatically

Understanding Entity in Hibernate

REVATURE

Demo using ***<ModelClass>.hbm.xml*** *file*

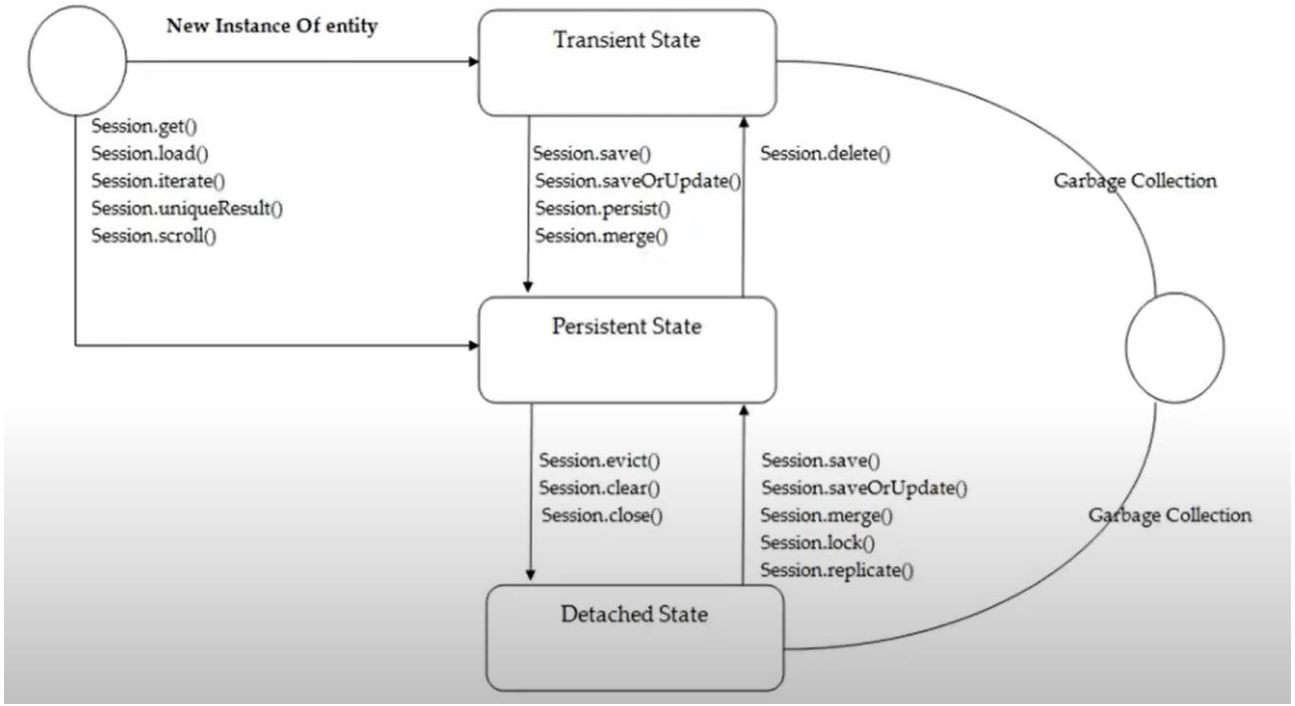
Hibernate Object States

Transient *- An object is in transient status if it has been instantiated and is still not associated with a Hibernate session.*

Persistent/Managed *- A persistent instance has a representation in the database and an identifier value. It might just have been saved or loaded, however, it is by definition in the scope of a Session.*

Detached *- A detached instance is an object that has been persistent, but its Session has been closed.*

Hibernate Lifecycle



If you try to save
Object with ***persist()***
Method. It will not be
Inserted immediately
In DB rather when you
flush or close session
Just before that the
Object is inserted into DB.
This method is from JPA.
save() *method saves the object*
Immediately into DB which can be
seen from hibernate show_sql
Configuration on console.
save() is hibernate api method not JPA.

@GeneratedValue

REVATURE

Generated Identifiers

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.

If we don't explicitly specify a value, the generation type defaults to AUTO.

AUTO Generation

If we're using the default generation type, the persistence provider will determine values based on the type of the primary key attribute. This type can be numerical or UUID.

For numeric values, the generation is based on a sequence or table generator, while UUID values will use the UUIDGenerator.

IDENTITY Generation

This type of generation relies on the IdentityGenerator, which expects values generated by an identity column in the database. This means they are auto-incremented. To use this generation type, we only need to set the strategy parameter. One thing to note is that IDENTITY generation disables batch updates.

SEQUENCE Generation

To use a sequence-based id, Hibernate provides the SequenceStyleGenerator class.

This generator uses sequences if our database supports them. It switches to table generation if they aren't supported.

In order to customize the sequence name, we can use the @GenericGenerator annotation with SequenceStyleGenerator strategy:

***SEQUENCE* is the generation type recommended by the Hibernate documentation.**

The generated values are unique per sequence. If we don't specify a sequence name, Hibernate will reuse the same hibernate_sequence for different types.

@GeneratedValue

REVATURE

TABLE Generation

The TableGenerator uses an underlying database table that holds segments of identifier generation values.
You need to customize the table name using the @TableGenerator annotation
The disadvantage of this method is that it doesn't scale well and can negatively affect performance.
We do not use Table Generation in Production

Custom Generator

Let's say we don't want to use any of the out-of-the-box strategies. In order to do that, we can define our custom generator by implementing the IdentifierGenerator interface.
We'll create a generator that builds identifiers containing a String prefix and a number

https://www.baeldung.com/hibernate-identifiers
Note: https://vladmihalcea.com/why-you-should-never-use-the-table-identifier-generator-with-jpa-and-hibernate/

Storing UUID as string in mysql using JPA

REVATURE

- *The purpose of the hibernate annotation @Type is to define the type of data stored in database and it's compatible with the previous line @Column(name = "uuid", columnDefinition = "char(36)")*
- *You can see here more examples in different use cases for org.hibernate.annotations.Type*

```
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Type;

import javax.persistence.*;
import java.util.UUID;
/** imports */

@Entity()
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(name = "UUID", strategy = "org.hibernate.id.UUIDGenerator")
    @Column(name = "uuid", columnDefinition = "char(36)")
    @Type(type = "org.hibernate.type.UUIDCharType")
    private UUID uuid;

   /** ... */
}
```

Types of Queries

REVATURE

There are three basic types of JPA Queries:

- **Query**, written in HQL Syntax & Java Persistence Query Language (JPQL) syntax
- **Native Query**, written in plain SQL syntax:

```
String sql = "SELECT * FROM EMPLOYEE_NATIVE_QUERY WHERE EMPLOYEE_NAME = :employeeName";
Query<Employee> query = session.createNativeQuery(sql, Employee.class);
query.setParameter("employeeName", employyeeNameToSearch);
List<Employee> results = query.list();
if (results.isEmpty())
      System.out.println("No Employee Found");
else
      System.out.println(results.get(0));
```

- **Named Query**
- **Criteria Query**, constructed programmatically via different methods

HQL & JPQL are same in syntax:

- *In hibernate we have **Hibernate Query Language (HQL)** and*
- *In JPA we have **Java Persistent Query Language(JPQL)***

Note: There is also on variation called **NamedNativeQuery**

https://www.baeldung.com/jpa-queries

Types of Queries

REVATURE

HQL:

```
String hql = "FROM Employee E WHERE E.employeeName = :employeeName";
Query<Employee> query = session.createQuery(hql, Employee.class);
query.setParameter("employeeName", employyeeNameToSearch);
List<Employee> results = query.list();
if (results.isEmpty())
      System.out.println("No Employee Found");
else
      System.out.println(results.get(0));
```

Types of Queries

REVATURE

Named Query:

Step 1: Declaration

```
@Entity
@Table(name = "employee_named_query")
@DynamicUpdate
@NamedQuery(name = "Employee.findByFirstName", query = "FROM Employee E WHERE E.employeeName =
:employeeName")
public class Employee {
```

Step 2: Invocation

```
Query<Employee> query = session.createNamedQuery("Employee.findByFirstName", Employee.class);
      query.setParameter("employeeName", employyeeNameToSearch);
      List<Employee> results = query.list();
      if (results.isEmpty())
            System.out.println("No Employee Found");
      else
            System.out.println(results.get(0));
```

Types of Queries

REVATURE

Criteria Query:

```
CriteriaBuilder criteriaBuilder = session.getCriteriaBuilder();
CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);
Root<Employee> emeployeeRoot = criteriaQuery.from(Employee.class);
List<Employee> results = session.createQuery(criteriaQuery.select(emeployeeRoot)
                         .where(criteriaBuilder.equal(emeployeeRoot.get("employeeName"),
                         employyeeNameToSearch))).list();
if (results.isEmpty())
      System.out.println("No Employee Found");
else
      System.out.println(results.get(0));
```

Types of Queries

REVATURE

Named Native Query:

```
@NamedNativeQuery(name = "findAllEmployeePancardMapping", resultClass = EmployeePancardDto.class,
resultSetMapping = "findAllEmployeePancardMapping", query = "SELECT `e`.`employee_id`, `e`.`employee_name`,
`e`.`email`, `e`.`date_of_joing`, `e`.`salary`, `p`.`pancard_id`, `p`.`pancard_number`, `p`.`date_of_birth` FROM
gd_hibernate.employee_table as e, gd_hibernate.pancard_table as p WHERE e.pancard_id_fk=p.pancard_id;")
@SqlResultSetMapping(name = "findAllEmployeePancardMapping", classes = @ConstructorResult(targetClass =
EmployeePancardDto.class, columns = {
            @ColumnResult(name = "employee_id", type = Integer.class),
            @ColumnResult(name = "employee_name", type = String.class),
            @ColumnResult(name = "email", type = String.class),
            @ColumnResult(name = "date_of_joing", type = Date.class),
            @ColumnResult(name = "salary", type = Double.class),
            @ColumnResult(name = "pancard_id", type = Integer.class),
            @ColumnResult(name = "pancard_number", type = String.class),
            @ColumnResult(name = "date_of_birth", type = Date.class) }))
@Entity
@Table(name = "employee_table")
@DynamicUpdate
public class Employee {
```

Caching in Hibernate

REVATURE

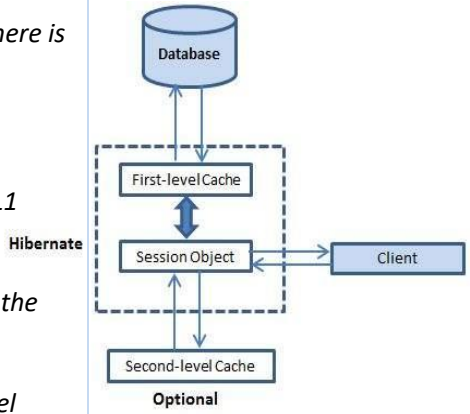
Hibernate performs caching to optimize the performance of an application. It's used to reduce the number of database hits by storing data locally in a cache.

First-Level Cache / L1 cache

- *The first-level cache is the first place that Hibernate checks for cached data. L1 caching in hibernate is enabled by default, and we can't disable it. It is a mandatory cache through which all requests must pass.*
- *This type of cache is associated with the Session object. Each Session object caches data independently, so there is no sharing of cached data across sessions, and the cached data is deleted when the session closes.*
- *An L1 cache is internal to a Session object and not accessible to any other session object in an application.*
- *This type of cache is only useful for repeated queries in the same session.*
- *When we query an entity first time, it retrieves the data from the database and stores it in the L1 cache associated with the session. If we query again with the same session object, then it loads the data from the L1 cache.*

Second-Level Cache/ L2 Cache

- *L2 cache is responsible for caching objects and sharing data across sessions. The L2 cache is associated with the SessionFactory object and is shared among all sessions created using the same session factory.*
- *If the requested query results are not in the first-level cache, then the second-level cache is checked. Any technology that supports out-of-the-box integration with Hibernate can be plugged in to act as a second-level cache.*
- *The L2 cache is disabled by default but we can enable it through configuration. If L2 caching is enabled for an entity, the following workflow is used:*
    1. *If an instance is already present in the L1 cache, then it is returned from there.*
    2. *If an instance is not found in the L1 cache, the L2 cache is searched, and if found the data is fetched and returned from there.*
    3. *If the data is not cached in the L2 cache, then the necessary data is loaded from the database and an instance is assembled and returned.*



Hibernate Second-Level Cache

REVATURE

- *One of the advantages of database abstraction layers such as ORM (object-relational mapping) frameworks is their ability to transparently cache data retrieved from the underlying store. This helps eliminate database-access costs for frequently accessed data.*
- *Performance gains can be significant if read/write ratios of cached content are high, especially for entities which consist of large object graph.*

What Is a Second-Level Cache?

- *As most other fully-equipped ORM frameworks, Hibernate has the concept of first-level cache. It is a session scoped cache which ensures that each entity instance is loaded only once in the persistent context.*
- *Once the session is closed, first-level cache is terminated as well. This is actually desirable, as it allows for concurrent sessions to work with entity instances in isolation from each other.*
- *On the other hand, second-level cache is SessionFactory-scoped, meaning it is shared by all sessions created with the same session factory. When an entity instance is looked up by its id (either by application logic or by Hibernate internally, e.g. when it loads associations to that entity from other entities), and if second-level caching is enabled for that entity, the following happens:*
- *If an instance is already present in the first-level cache, it is returned from there*
- *If an instance is not found in the first-level cache, and the corresponding instance state is cached in the second-level cache, then the data is fetched from there and an instance is assembled and returned*
- *Otherwise, the necessary data are loaded from the database and an instance is assembled and returned*
- *Once the instance is stored in the persistence context (first-level cache), it is returned from there in all subsequent calls within the same session until the session is closed or the instance is manually evicted from the persistence context. Also, the loaded instance state is stored in L2 cache if it was not there already.*

Hibernate Second-Level Cache

Inteface org.hibernate.cache.spi.RegionFactory

Hibernate second-level caching is designed to be unaware of the actual cache provider used. Hibernate only needs to be provided with an implementation of the org.hibernate.cache.spi.RegionFactory interface which encapsulates all details specific to actual cache providers. Basically, it acts as a bridge between Hibernate and cache providers.

- *We add the Ehcache region factory implementation to the classpath with the following Maven dependency:*

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.2.2.Final</version>
</dependency>
```

- *Take a look here for latest version of hibernate-ehcache. However, make sure that hibernate-ehcache version is equal to Hibernate version which you use in your project, e.g. if you use hibernate-ehcache 5.2.2.Final like in this example, then the version of Hibernate should also be 5.2.2.Final.*
- *The hibernate-ehcache artifact has a dependency on the Ehcache implementation itself, which is thus transitively included in the classpath as well.*

Hibernate Second-Level Cache

REVATURE

Enabling Second-Level Caching

- *With the following two properties we tell Hibernate that L2 caching is enabled and we give it the name of the region factory class:*

```
hibernate.cache.use_second_level_cache=true
hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

Or

<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
      <session-factory>
            <property name="hibernate.cache.use_second_level_cache">true</property>
            <property name="hibernate.cache.region.factory_class">
            org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
      </session-factory>
</hibernate-configuration>
```

- *To disable second-level caching (for debugging purposes for example), just set hibernate.cache.use_second_level_cache property to false.*

Hibernate Second-Level Cache

REVATURE

Making an Entity Cacheable

- *In order to make an entity eligible for second-level caching, we annotate it with Hibernate specific @org.hibernate.annotations.Cache annotation and specify a cache concurrency strategy.*
- *Some developers consider that it is a good convention to add the standard @javax.persistence.Cacheable annotation as well (although not required by Hibernate), so an entity class implementation might look like this:*

```
@Entity
@Cacheable  // JPA 2.0 annotation recommended by hibernate
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Foo {
```

Cache Concurrency Strategy

Based on use cases, we are free to pick one of the following cache concurrency strategies:

- ***READ_ONLY:*** *Used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It is very simple and performant. Very suitable for some static reference data that don't change*
- ***NONSTRICT_READ_WRITE:*** *Cache is updated after a transaction that changed the affected data has been committed. Thus, strong consistency is not guaranteed and there is a small time window in which stale data may be obtained from cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency*
- ***READ_WRITE:*** *This strategy guarantees strong consistency which it achieves by using 'soft' locks: When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database*
- ***TRANSACTIONAL:*** *Cache changes are done in distributed XA transactions. A change in a cached entity is either committed or rolled back in both database and cache in the same XA transaction*

Hibernate Second-Level Cache - (Optional Reading)

REVATURE

- ***Eviction Policies*** *- If expiration and eviction policies are not defined, the cache could grow indefinitely and eventually consume all of available memory. In most cases, Hibernate leaves cache management duties like these to cache providers, as they are indeed specific to each cache implementation. e.g.We could define the Ehcache configuration to limit the maximum number of cached Foo instances to 1000*
- ***Collection Cache-*** *Collections are not cached by default, and we need to explicitly mark them as cacheable.*
- ***Query Cache-*** *Results of HQL queries can also be cached. This is useful if you frequently execute a query on entities that rarely change.*

  To enable query cache, set the value of hibernate.cache.use_query_cache property to true:

  ```
  hibernate.cache.use_query_cache=true
  ```

  Then, for each query you have to explicitly indicate that the query is cacheable (via an org.hibernate.cacheable query hint):

  ```
  entityManager.createQuery("select f from Foo f")
    .setHint("org.hibernate.cacheable", true)
    .getResultList();
  ```

Cache

JCACHE Specification- https://www.jcp.org/en/jsr/detail?id=107
eCache implementation of JCACHE Specification.

Caching strategy can be of following types:
•**none** : No caching will happen.
•**read-only** : If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
•**read-write** : If the application needs to update data, a read-write cache might be appropriate.
•**nonstrict-read-write** : If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.
•**transactional** : The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager_lookup_class.

Owning Side and Inverse Side

REVATURE

Owning side is where you have defined foreign key referencing primay key of other side.
Inverse side is opposite side to Owing side.

Unidirectional and Bidirectional Relation

REVATURE

In case of RDBMS we have Unidirectional relations.
We need to apply join to fetch column values from both side of tables.

In Hibernate we can configure Relations to be Uni or Bidirectional.
Bidirectional relations are required only on specific case bases and mostly Unidirectional relations works fine.

Association Mapping | Annotaions | javax.persistence.xxxxx

REVATURE

@OneToOne
@OneToMany | @ManyToOne
@ManyToMany
@JoinColumn *- marks a column as a join column for an entity association or an element collection.*
@JointTable *- indicates that there is a link table which joins two tables via containing there keys. This annotation is mainly used on the owning side of the relationship. JoinTable is must in case of many-may relations, but it can also be used if we want attributes for one-many, one-one relation as per use case.*

cascade = CascadeType.ALL - *perform cascading while persisting (update/delete) owning entity tuple, inverse entoty tuples will also be persisted (updated/deleted).*

Important remark *: In case of *Many* association, always override hashcode and equals method which are looked by hibernate when holding entities into collections.*

@OneToOne | Unidirectional

REVATURE

```java
@Entity
@Table(name = "employee_table")
@DynamicUpdate
public class Employee {
    @Id
    @Column(name = "employee_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer employeeId;
```

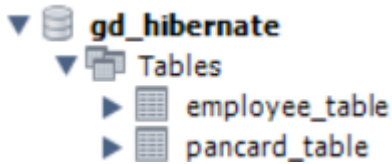
```java
@Entity
@Table(name = "pancard_table")
@DynamicUpdate
public class Pancard {
    @Id
    @Column(name = "pancard_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer pancardId;
```

Inside Employee Entity (Owning Side) define below unidirectional relation:

```java
// Employee entity is owning side as it will have foreign key column from Pancard entity
// @JoinColumn marks a column as a join column for an entity association or an element collection.
// cascade = CascadeType.ALL, perform cascading while persisting (update/delete)
// Employee tuples, Pancard tuples will also be persisted (updated/deleted).

@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "pancard_id_fk")
private Pancard pancard;
```

The outcome in DB



@OneToMany/@ManyToOne | Unidirectional

REVATURE

```java
@Entity
@Table(name = "employee_table")
@DynamicUpdate
public class Employee {
    @Id
    @Column(name = "employee_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer employeeId;
```

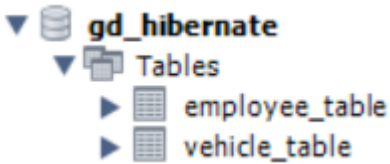
```java
@Entity
@Table(name = "vehicle_table")
@DynamicUpdate
public class Vehicle {
    @Id
    @Column(name = "vehicle_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer vehicleId;
```

Inside Vehicle Entity (Owning Side) define below unidirectional relation:

```java
// Vehicle entity is owning side as it will have foreign key column from
// Employee entity
// @JoinColumn marks a column as a join column for an entity association or an
// element collection.
// cascade = CascadeType.ALL, perform cascading while persisting (update/delete)
// Vehicle tuples, Employee tuples will also be persisted (updated/deleted).

@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "employee_id_fk")
private Employee employee;
```

The outcome in DB



@ManyToMany | Unidirectional

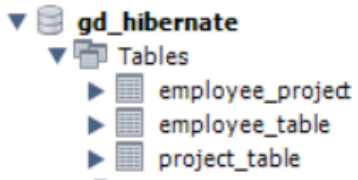
```java
@Entity
@Table(name = "employee_table")
@DynamicUpdate
public class Employee {
    @Id
    @Column(name = "employee_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer employeeId;
```

```java
@Entity
@Table(name = "project_table")
@DynamicUpdate
public class Project {
    @Id
    @Column(name = "project_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer projectId;
```

Inside Employee Entity define below unidirectional relation:

```java
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(name = "EMPLOYEE_PROJECT", joinColumns = { @JoinColumn(name = "EMPLOYEE_ID") }, inverseJoinColumns = {
        @JoinColumn(name = "PROJECT_ID") })
private List<Project> projects = new ArrayList<Project>();
```

The outcome in DB



Join Query - Declaration & Invocation

REVATURE

```
@NamedNativeQuery(name = "findAllEmployeeProjectMapping", resultClass = EmployeeProjectDto.class, resultSetMapping = "findAllEmployeeProjectMapping", query =
"SELECT `e`.`employee_id`, `e`.`employee_name`, `e`.`email`, `e`.`date_of_joing`, `e`.`salary`, `p`.`project_id`, `p`.`project_name`, `p`.`project_client_name`,
`p`.`total_budget`, `p`.`billing_currency` FROM gd_hibernate.employee_table as e, gd_hibernate.employee_project as ep, gd_hibernate.project_table as p WHERE
e.employee_id=ep.EMPLOYEE_ID AND ep.PROJECT_ID=p.project_id Order by 1; ")
@SqlResultSetMapping(name = "findAllEmployeeProjectMapping", classes = @ConstructorResult(targetClass = EmployeeProjectDto.class, columns = {
        @ColumnResult(name = "employee_id", type = Integer.class),
        @ColumnResult(name = "employee_name", type = String.class),
        @ColumnResult(name = "email", type = String.class),
        @ColumnResult(name = "date_of_joing", type = Date.class),
        @ColumnResult(name = "salary", type = Double.class),
        @ColumnResult(name = "project_id", type = Integer.class),
        @ColumnResult(name = "project_name", type = String.class),
        @ColumnResult(name = "project_client_name", type = String.class),
        @ColumnResult(name = "billing_currency", type = String.class),
        @ColumnResult(name = "total_budget", type = String.class) }))
@Entity
@Table(name = "employee_table")
@DynamicUpdate
public class Employee {
    @Id
    @Column(name = "employee_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer employeeId;
```

```
private static List getAllEmployeeProjectDetails(Session session) {
    Query query = session.createNamedQuery("findAllEmployeeProjectMapping", EmployeeProjectDto.class);
    List results = query.getResultList();
    if (results.isEmpty())
        System.out.println("No Employee Found");
    else {
        for (Object currentRow : results) {
            System.out.println(currentRow);
        }
    }
    return results;
}
```

Bi-directional (Optional) | @ManyToOne

REVATURE

Inverse Side

```
@Entity
public class Employee {
    @Id
    private Long id;
    @OneToMany(mappedBy = "employee")
    private List<Email> emails;
}
```

This code of opposite type annotation & mappedBy makes relation bi-directional

Owning Side

```
@Entity
public class Email {
    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "employee_id")
    private Employee employee;
}
```

Association Mappings | XML (Optional)

REVATURE

Using XML File:
Hibernate - One-to-One Mappings (tutorialspoint.com)
Hibernate - Many-to-One Mappings (tutorialspoint.com)
Hibernate - One-to-Many Mappings (tutorialspoint.com)
Hibernate - Many-to-Many Mappings (tutorialspoint.com)

Appendix

JAP Join Types: https://www.baeldung.com/jpa-join-types
How to Store Time: https://www.baeldung.com/hibernate-date-time