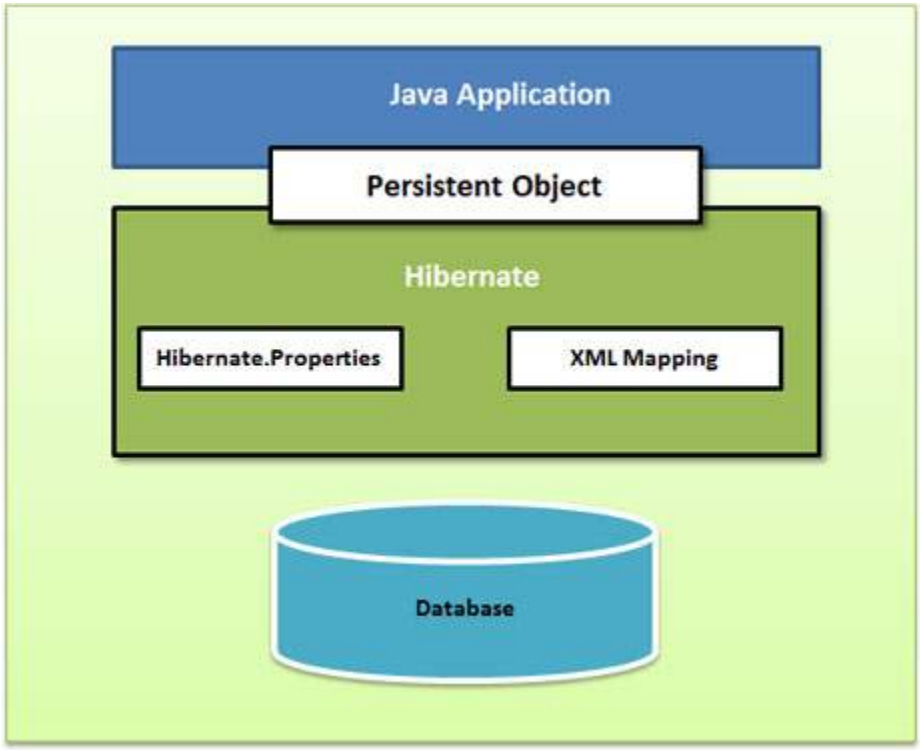
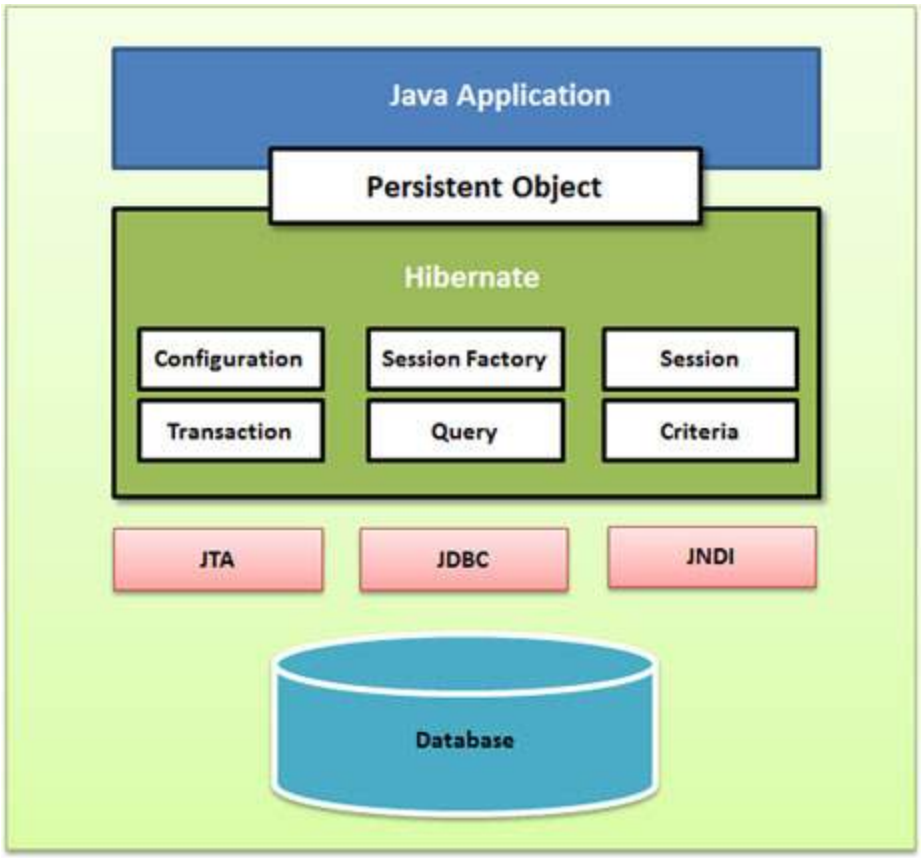
Hibernate ORM Framework - JPA Implementation

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.



Following is a detailed view of the Hibernate Application Architecture with its important core classes.



Configuration Object

The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization.

The Configuration object provides two keys components −

1. Database Connection configuration
2. Class Mapping Setup

SessionFactory Object

Configuration object is used to create a SessionFactory object.
The SessionFactory is a thread safe object and used by all the threads of an application.
We would need one SessionFactory object per database using a separate configuration.

(**Thread safety** is a computer programming concept applicable to multi-**threaded** code. **Thread**-**safe** code only manipulates shared data structures in a manner that ensures that all **threads** behave properly and fulfill their design specifications without unintended interaction.)

Session Object

A Session 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.

Transaction Object

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA-Java Transaction API).

Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object(Filtered Select query with many Where condition)

Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

HIBERNATE_OBJECT_STATES

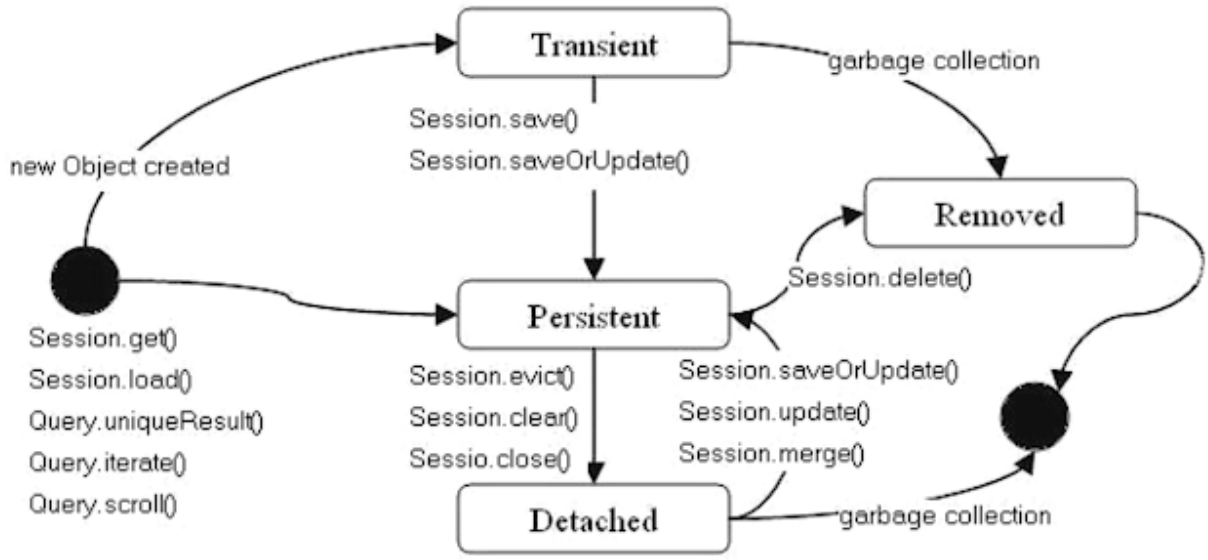
A new instance of a persistent class which is not associated with a Session, has no representation in the database and no identifier value is considered ***transient*** by Hibernate:

```
Person person = new Person(); person.setName("Foobar");
// person is in a transient state
```

A ***persistent*** instance has a representation in the database, an identifier value and is associated with a Session. You can make a transient instance ***persistent*** by associating it with a Session:

Long id = (Long) session.save(person); **// person is now in a persistent state**

Now, if we close the Hibernate Session, the persistent instance will become a ***detached*** instance: it isn't attached to a Session anymore (but can still be modified and reattached to a new Session later though).



HIBERNATE 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.

Caching is important to Hibernate as well. It utilizes a multilevel caching scheme as explained below −

First-level Cache

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

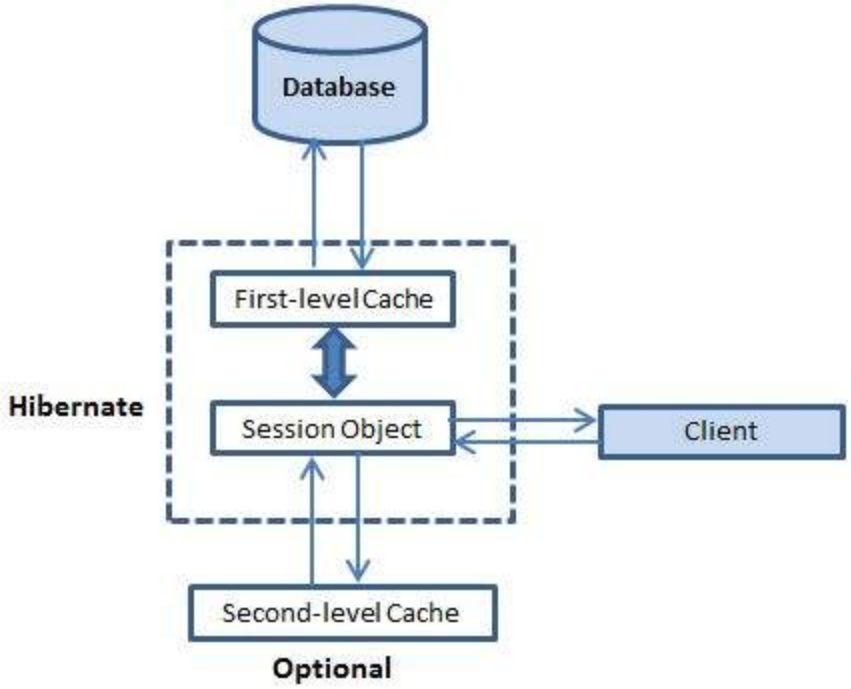
Second-level Cache

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Query-level Cache

Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.



JPA or Hibernate Properties:

#database configuration

spring.h2.console.enabled=true // allows to see the h2 database in browser

spring.h2.console.path=/h2-console // url of h2 database in browser

spring.datasource.url=jdbc:h2:file:~/h2/mydb // file based database

spring.datasource.username=sa // database username

spring.datasource.password= //database password

#jpa configuration

spring.datasource.driverClassName=org.h2.Driver // JDBC driver for h2 database

spring.jpa.hibernate.ddl-auto=update //make sure to only create table once and from next server startup only update

spring.jpa.show-sql=true // prints sql queries in logs/console

spring.jpa.properties.hibernate.format_sql=true // prints sql in formatted way

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

This property makes Hibernate generate the appropriate SQL for the chosen database.

JPA offers 4 different ways to generate primary key values: @Id

1. *AUTO*: Hibernate selects the generation strategy based on the used dialect,
2. *IDENTITY*: Hibernate relies on an auto-incremented database column to generate the primary key,
3. *SEQUENCE*: Hibernate requests the primary key value from a database sequence,
4. *TABLE*: Hibernate uses a database table to simulate a sequence.

SPRING DATA JPA - Abstraction layer built on top of Hibernate

Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

Different types of Repository in Spring Data JPA:

- **CrudRepository**: provides CRUD functions
- **PagingAndSortingRepository**: provides methods to do pagination and sort records
- **JpaRepository**: provides JPA related methods such as flushing the persistence context and delete records in a batch

Different ways of writing queries in Spring Data JPA

findByFieldName

```
@Repository
public interface SurveyRepository extends JpaRepository<Survey, Long> {
    Optional<Survey> findById(Long surveyId);

    Page<Survey> findByCreatedBy(Long userId, Pageable pageable);

    long countByCreatedBy(Long userId);

    List<Survey> findByIdIn(List<Long> surveyIds);

    List<Survey> findByIdIn(List<Long> surveyIds, Sort sort);
}
```

@Query

```
@Repository
public interface VoteRepository extends JpaRepository<Vote, Long> {

    @Query("SELECT NEW com.survey.model.ChoiceVoteCount(v.choice.id, count(v.id)) FROM Vote v WHERE v.survey.id in :surveyIds GROUP BY v.choice.id")
    List<ChoiceVoteCount> countBySurveyIdInGroupByChoiceId(@Param("surveyIds") List<Long> surveyIds);

    @Query("SELECT NEW com.survey.model.ChoiceVoteCount(v.choice.id, count(v.id)) FROM Vote v WHERE v.survey.id = :surveyId GROUP BY v.choice.id")
    List<ChoiceVoteCount> countBySurveyIdGroupByChoiceId(@Param("surveyId") Long surveyId);

    @Query("SELECT v FROM Vote v where v.user.id = :userId and v.survey.id in :surveyIds")
    List<Vote> findByUserIdAndSurveyIdIn(@Param("userId") Long userId, @Param("surveyIds") List<Long> surveyIds);

    @Query("SELECT v FROM Vote v where v.user.id = :userId and v.survey.id = :surveyId")
    Vote findByUserIdAndSurveyId(@Param("userId") Long userId, @Param("surveyId") Long surveyId);

    @Query("SELECT COUNT(v.id) from Vote v where v.user.id = :userId")
    long countByUserId(@Param("userId") Long userId);

    @Query("SELECT v.survey.id FROM Vote v WHERE v.user.id = :userId")
    Page<Long> findVotedSurveyIdsByUserId(@Param("userId") Long userId, Pageable pageable);

    long deleteByChoice_IdIn(List<Long> ids);
}
```

Adding restrictions on Entity fields

```
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;
```

```
@Entity
@Table(name = "tbl_user")
@Data
public class User extends DateAudit {
        @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    @Size(max = 40)
    private String name;

    @NotBlank
    @Size(max = 15)
    private String username;
```

OR you can also do

@Column(name =”Name”, nullable=false, length=40)

private String name;