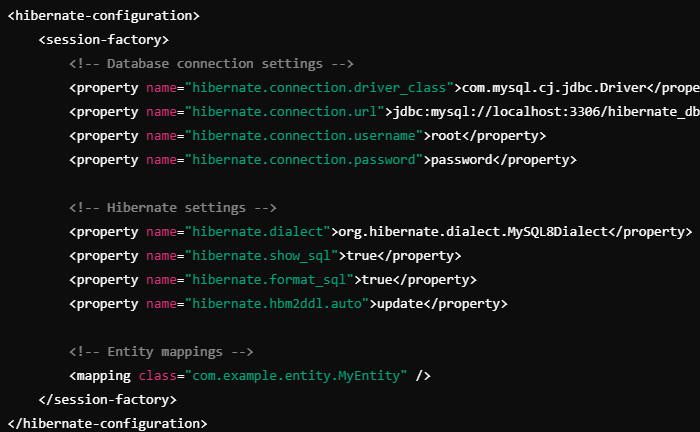
**Basic Hibernate Concepts:**

1. What is Hibernate, and why is it used in Java applications?
   * Hibernate is a ORM tool
   * It is open source, lightweight
   * It simplifies the development of java applications to interact with databases by mapping java objects to database tables.
   * Features:
     1. ORM tool: provides abstraction layer that eliminates the JDBC boilerplate code.
     2. HQL: A powerful query language similar to SQL but operates on entity objects instead of tables.
     3. Automatic table mapping
     4. Transparent Persistence: Automatically saves, Updates and retrieves objects without SQL.
   * Why to use:
     1. Reduces boilerplate code: by writing SQL internally
     2. Portable: can switch database easily
     3. Efficient: Handles complex association and relations between tables
     4. Scalable: Built in catching and lazy loading.
2. What are the advantages of using Hibernate over JDBC?
   * Eliminates boilerplate code: Reduces boilerplate code like creating connections, managing statements, handling exceptions.
   * ORM: maps objects to database tables.
   * Database independent: You can switch databases without significant changes.
   * Supports HQL:
   * Automatic Schema generation: can generate database schema based on entity mapping.
   * Caching: Built in first level and second level caching mechanism.
   * Supports Lazy and Egar loading:
3. How to create session Factory using XML file.
   * Create configuration file as below (Can us properties file of programmatic approach to set properties)



* + To create SessionFactory



1. What is Session and SessionFactory?
   * SessionFactory: It reads hibernate configuration and sets up application for database interaction, such as mapping java classes to database tables
     1. Heavy weight thread safe object to create session object
     2. Created once per application
   * Session: session object is used to directly connect to database to perform CRUD operations.
     1. Light weight non-thread safe object used to interact with database
     2. Created per user request or per unit of work and has shorter lifecycle.
2. Explain the difference between Session and SessionFactory in Hibernate.

A screenshot of a black screen

Description automatically generated

1. What is an Entity in Hibernate, and how is it defined?
   * These are the POJOs (Plain old java objects) that are managed by hibernate persistence.
   * It is an object that represents table in database, each instance of an entity represents row in a table
   * Entities are defined by POJOs annotated with Hibernate specific annotations or mapped through XML configuration.
2. What is the significance of the @Entity annotation in Hibernate?
   * This annotation marks java class as an Entity.
   * It signals hibernate that this class should be mapped with table in database.
   * If you don’t use @Entity annotation hibernate will treat this as normal class and any attempt to store and retrieve the class using hibernate will result in error.
   * Mapping of Entity without using @Entity annotation (XML mapping)

A screen shot of a computer program

Description automatically generated

1. Steps to store object in database using Hibernate.
   * Create SessionFactory:

SessionFactory factory = new Configuration().configure().buildSessionFactory()

* + Open Session using SessionFactory

Session session = factory.openSession();

* + Begin transaction

Transaction tx = session.beginTransaction();

* + Save object using sassion.

Session.save(object);

* + Commit transaction.

tx.commit()

* + Close session

Session.close()

1. What is the purpose of the @Table annotation in Hibernate?
   * Provides flexibility to map entities to database table with different name or structure than entity
   * Allows defining database schema or table to which table belongs
   * Annotation can also be used to define unique constraint

@Table(

name = "table\_name",

schema = "schema\_name", // Optional

catalog = "catalog\_name", // Optional

uniqueConstraints = { @UniqueConstraint(columnNames = { "column1", "column2" }) } // Optional

)

1. How does Hibernate map a Java class to a database table?
   * Hibernate reads class annotated with @Entity annotation and maps it to specified table using annotations or xml configuration.
   * SessionFactory Initialization: Hibernate creates metadata based on mapping and generates SQL statements to interact with database.
   * SQL generation: At runtime uses mapping to translate object-oriented queries into SQL queries of database.
   * Then save, retrieves and deletes objects using mapping details.
2. What are the different ways to define a Hibernate mapping?
   * Annotation based:
     1. Can use @Entity, @Table, @Column annotations.
     2. Clear, Concise and part of class definition which improves readability
   * XML based:

<hibernate-mapping>

<class name="com.example.Employee" table="employees">

<id name="id" column="id">

<generator class="identity"/>

</id>

<property name="name" column="full\_name" not-null="true" length="50"/>

<property name="salary" column="salary"/>

</class>

</hibernate-mapping>

* + 1. Useful in scenarios where mapping configurations might need to be changed without modifying the code.
  + Programmatic approach:

1. Explain the concept of Hibernate configuration. What is a hibernate.cfg.xml file?
   * Hibernate configuration refers to the setup that hibernate requires to interact with database manage persistence.
   * It includes connection properties, hibernate settings and mapping files.
   * hibernate.cfg.xml is a primary configuration file. It is XML configuration files used to define
     1. Database connection details
     2. Hibernate specific settings
     3. Entity Mapping
   * It centralizes database properties, hibernate settings and mapping making it easy to maintain and modify.
2. What is the purpose of the @Id annotation in Hibernate?
   * Used to specify primary key of entity.
   * It allows hibernate to uniquely identify, persist and manage java object in relation to their corresponding database records.
   * Every entity must have an @Id
3. What is difference between get and load to get data in hibernate?
   * get() returns null if object not found in cache or in database, and returns fully initialized object which may involve lot of database hits i.e. Egar loading
   * load() throws ObjectNotFoundException if object not found in cache or in database, and returns proxy instead (Object containing id only no any data) and hits database only of method other than getId() is called on persist object i.e. lazy loading for all fields
4. What is @Embeddable annotation
   * This annotation is used to define a class that can be embedded as part of another entity.
   * It represents a composite type that itself is not an entity but can be embedded as part of another entity.
   * Suppose we have two classes Student and Address

@Entity

Class Student{

@Id

Private int id;

Private String name;

@Embedded

Private Address addr;

//Constructor and Getter Setter

}

@Embeddable

Class Address{

Private String lane;

Private String city;

//Constructor and Getter Setter

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Name** | **Lane** | **City** |
|  |  |  |  |
|  |  |  |  |

**Object-Relational Mapping:**

1. What is the difference between @OneToMany and @ManyToOne relationships in Hibernate?

A screenshot of a computer

Description automatically generated

* + mappedBy, cascade and orphanRemovel these attributes are used with @OneToMany annotation
  + mappedBy specifies field in child entity that specifies map this relationship
  + cascade propagates operation from parent to child like delete, save
  + orphanRemoval deletes child entity if they are removed from relationship
  + @JoinColumn(name = “”) is used to specify join column in child entity along with @ManyToOne annotation.

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

// Getters and setters

}

@Entity

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval = true)

private List<Employee> employees = new ArrayList<>();

// Getters and setters

}

1. What is use of mappedBy property in Mapping annotations?
   * The mappedBy property is critical for defining bidirectional relationships in Hibernate.
   * It tells Hibernate to map the relationship from the other side, ensuring there is a single source of truth for the relationship management.
   * Proper use of mappedBy simplifies mappings and prevents database inconsistencies.
2. What are the different fetch strategies in Hibernate? Explain LAZY and EAGER fetching.
3. How can you handle bidirectional relationships in Hibernate?
4. What is @ManyToMany annotation, and how is it implemented in Hibernate?
   * It is used to represent many to many relationships
   * Many to many means multiple records form one entity is related to multiple records of another entity.
   * Extra Join table is required to handle this relationship
   * One side is Owning side where relationships is managed
   * Other side is Inverse side, marked with mappedBy
   * @JoinTable annotation is used to customize join table.
   * By default @ManyToMany is lazily loaded.

@Entity

public class Course {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

@ManyToMany(mappedBy = "courses") // Refers to the field name in Student

private List<Student> students;

// Getters and Setters

}

@Entity

public class Student {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToMany

@JoinTable(

name = "student\_course", // Name of the join table

joinColumns = @JoinColumn(name = "student\_id"), // FK in join table referencing Student

inverseJoinColumns = @JoinColumn(name = "course\_id") // FK in join table referencing Course

)

private List<Course> courses;

// Getters and Setters

}

1. How does Hibernate handle @OneToOne relationships?
   * It is used when one entity instance is associated with exactly one instance of another entity
   * It can be unidirectional or bidirectional
   * Foreign key column of one table reference the primary key column of another table
   * Can use cascading operations to propagate actions like persist, remove, merge
   * By default uses fetch type Egar for @OneToOne
2. What is default fetch types of @OneToOne, @ManyToOne, @OneToMany and @ManyToMany and explain if there is specific reason?

A screenshot of a computer

Description automatically generated

**CRUD Operations:**

1. How do you perform CRUD operations in Hibernate?
2. What is the difference between save() and persist() in Hibernate?
   * Save(): is a Hibernate specific method that eagerly INSERT the entity and returns serializable identifier.
   * Persist(): It is a JPA compliant delay the INSERT query until the transaction is committed and session is flushed.
3. What is the difference between get() and load() methods in Hibernate?
   * Get return null if object doesn’t exists in database, and executes SQL query immediately.
   * Load return proxy object, and delays the query execution until the entity’s data is accessed
4. How do you update an entity in Hibernate?
   * Update(): Use when you are working with detached entity and want to reattach it to the session.
   * Merge(): Use when you are not sure if the entity is detached or Persistent, or wan to work with detached entity.
   * SaveOrUpdate(): when you want to save or update entity, depending on whether the entity exist in the database.
   * HLQ/JPQL: Use for bulk update where you don’t want to load data into memory.
5. How do you delete an entity in Hibernate?
   * Delete(): deletes persistence entity that is currently associated with session.
   * Remove(): similar to delete() in JPA, used to delete persistent entity.
   * HQL/JPQL: to perform bulk delete directly without loading objects to memory
   * Cascade delete: if there is relation between entities and Cascade property is set to Cascade.ALL or Cascade.ROMOVE it will delete related enitites
   * Detached Entities : use merge() or update() to reattach and then use delete() method to delete entity.

**Lifecycle States:**

* + Transient State: when object is created using new keyword and it is not yet associated with any hibernate session or database.
  + Persistent State: Object is in persistent state when it is associated with hibernate session and represents row in a database.
    1. Any changes in an object are tracked and updated in database when session is flushed, and transaction is committed.
  + Detached State: The object is in detached state when it is once associated with hibernate session, but the session has been closed or the object was explicitly detached.
  + Removed State: The object is in removed state when the object is marked for deletion using delete() method but the transaction is not committed yet.

A black box with white text

Description automatically generated

**Querying:**

1. What is HQL (Hibernate Query Language), and how is it different from SQL?
   * HQL is a hibernate query language is an object-oriented query language provided by hibernate. It is used to perform database operations like retrieving, updating, deleting data into database in object-oriented way.
   * HQL operates on persistent objects and their properties rather than Table and their columns.
   * It is Similar to SQL but works with hibernate entities and their properties.
   * Differences:
     1. Syntax: HQL uses entity class names and their properties while SQL uses tables and columns.
     2. HQL is object oriented and works with persistent objects, SQL is relational and works with tables and columns
     3. HQL is database independent, while SQL is database dependent
     4. HQL returns entity objects while SQL returns rows
     5. Joins: HQL joins are based on relationships between properties i.e. @OneToOne, @OneToMany , SQL requires explicit joins

**NOTE: when using joins in HQL you do not always require to provide join condition, hibernate will understand with mapped relationships between entities, OR you can also explicitly specify join condition.**

1. What is Lazy and Egar loading
   * Lazy: It is a feature of hibernate that delays the initialization and fetching of data from database until it is actually needed.
     1. Hibernate uses proxies to implement lazy loading of associated objects.
     2. It is used to achieve
        1. Performance Optimization avoiding unnecessary fetching
        2. Memory Efficiency
        3. Reduce initial load time
   * Egar: Mechanism in hibernate where associated data is fetched immediately along with the main entity.
     1. Used when
        1. Immediate Access Required
        2. Avoiding LazyInitializationException
        3. Performance Tuning: For small, fixed datasets or closely associated data.
   * One more concept is fetch modes:
     1. JOIN: fetch main entity with associated entities in one query
     2. SELECT: Fetches the main entity first, then retrieves associated data with separate queries
     3. SUBSELECT: Fetches all associated data for a collection in **a single query** when accessing the parent entity for the first time.
     4. BATCH: Fetches data for a collection or entity in **batches**
2. Explain the use of Criteria API in Hibernate.
   * Criteria API is a powerful tool to query the database programmatically.
   * Unlike HQL where we use string-based queries, Criteria API allows developers to create queries using java objects.
   * Features:
     1. Type Safe: prevents error by checking query syntax compile time.
     2. Dynamic query construction:
     3. Programmatic Approach:
     4. Integrates with projections and Aggregates: Supports groupingBy, having and Projections.
   * Components of Criteria API:
     1. **CriteriaBuilder:** A factory for creating Criteria API queries.
     2. **CriteriaQuery:** Represents a query definition.
     3. **Root:** Represents the entity you are querying.
     4. **Predicate:** Represents conditions or filters applied in the query.
     5. **Path:** Used to navigate to specific fields or attributes of the entity.
     6. **TypedQuery:** Executes the query and retrieves the results.

A screenshot of a computer code

Description automatically generated

1. How to execute HQL query using hibernate.
   * Create Session.
   * Begin transaction
   * Create query string based on entity class and its properties.
   * Create HQL query using createQuery() method of session. It will return Query Object.
   * Set parameters if required

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

String hql = "FROM Employee WHERE department = :deptName";

Query<Employee> query = session.createQuery(hql, Employee.class);

query.setParameter("deptName", "HR");

List<Employee> employees = query.list(); // Fetch results

for (Employee emp : employees) {

System.out.println(emp.getName());

}

transaction.commit();

session.close();

1. How can you fetch data using HQL? Provide an example.
2. What is the advantages of using named parameters over using string concatenation to create HQL query.
   * They prevent SQL Injection.
     1. As parameters are bound using placeholders and setParameter() method, user input is treated as data and not executable SQL code.
     2. Automatic Escaping: Hibernate escapes special characters in parameter values preventing malicious code execution
     3. Query parsing: parameterized queries are parsed by database engines with placeholders ensuring query structure is not altered.
3. What are the advantages of using Criteria API over HQL?
   * Dynamic query construction: HQL requires query string to be written explicitly
   * Type Safety: String based HQL is prone to syntax errors.
   * More Readable and Maintainable
   * Strong integration with JPA: while HQL is hibernate specific.
   * Prevention of SQL injection
4. What is the difference between HQL and JPQL?
   * JPQL is the standardized, portable query language under the JPA specification, ensuring compatibility across ORM providers.
   * HQL, while similar in syntax, extends JPQL with Hibernate-specific features, making it more powerful but less portable. Use HQL when working strictly with Hibernate and JPQL for broader JPA-compliant applications.
5. What are native queries and what are they used for in Hibernate?
   * Native queries are the SQL queries written in native SQL languages (database specific)
   * They are executed against underlying database bypassing hibernate ORM mechanism.
   * Uses:
     1. To use database specific features: querying non-mapped table
     2. Performance Optimized: optimized which involves complex joins
     3. Handling legacy databases: which are not compatible with hibernate
   * How to use native Queries:

Session session = sessionFactory.openSession();

// Native SQL query to fetch data directly from the database

String sql = "SELECT \* FROM employee WHERE employee\_id = :empId";

SQLQuery query = session.createSQLQuery(sql);

query.setParameter("empId", 101);

query.addEntity(Employee.class); // Optional: Map result set to a specific entity

List<Employee> employees = query.list();

session.close();

**Performance and Optimization:**

1. What is the N+1 problem in Hibernate, and how do you avoid it?
   * The **N+1 problem** in Hibernate refers to an inefficiency that arises when fetching data with associations. It occurs when
     1. **One query** (1) is executed to fetch the main entities (e.g., N parent entities).
     2. **N additional queries** are executed to fetch the associated data for each entity.
   * Impact:
     1. Performance degradation
     2. Increased database load
   * To Avoid
     1. Analyze and optimize queries using Hibernate's SQL logging.
     2. Use lazy loading by default but override it with fetch joins or batch fetching for specific use cases.
     3. Avoid eager fetching for large or unnecessary relationships.
2. How does caching work in Hibernate?
   * Caching is a mechanism to enhance the performance of applications.
   * Caching is used to reduce the number of database queries in applications.
3. What is the difference between first-level cache and second-level cache in Hibernate?
   * Fist is level cache is associated with the current active session while the second level cache is associated with Session Factory.
   * The first level cache is enabled by default, for the second level cache we need to manually enable it.
   * First level cache stored in memory of current session, second level cache stored in external cache providers like Ehcache, Redis, infispan.
   * The first level cache is a Transaction/session level cache, while second level cache is a Application level cache.
   * The first level cache is cleared when session is closed, second level cache persist across multiple session.
   * **Query caching:**
   * Query caching in Hibernate refers to the process of caching the results of database queries to improve performance by reducing the need to re-execute the same queries repeatedly.
   * When a query is executed, the result can be stored in the cache, and for subsequent executions of the same query (with the same parameters), the result is fetched from the cache instead of querying the database again.
   * This can significantly reduce the number of database hits and improve application performance.
   * For query caching to be effective, the query itself needs to be marked as cacheable.
   * Cached query results have a lifecycle, and they need to be invalidated or evicted when data changes in the database to avoid serving stale results. This eviction can be done automatically based on the cache provider’s settings.
   * Marking Queries as cacheable:

A screen shot of a computer program

Description automatically generated

1. What is the purpose of flush() and clear() methods in Hibernate?
   * **flush():** this method is used to synchronize the current in memory state of hibernate session with the underlying database. It forces hibernate to persist any changes made in the session to the database.
   * It does not commit the changes; it just makes sure that all pending changes in the session are sent to the database.
   * It can be called manually or hibernate will invoke it automatically before transaction is committed or query is executed.
   * **Clear():** used to remove all the objects from session cache and free up memory, it detaches all the entities from the current session.
   * After calling clear() any changes made to the entities in the session are forgotten, it does not affects the database, it only clears the session cache.

**Transactions and Concurrency:**

1. How does Hibernate handle transactions?
   * Hibernate handles transactions to ensure ACID properties are maintained during database operation.
   * Transactions in hibernate are managed either programmatically using Hibernate Transaction API or declaratively using framework like Spring.
   * Hibernate uses session object to manage database operations, Transaction object obtained from session object used to begin and commit/rollback the transction.
   * In a distributed environment hibernate uses JTA (Java Transaction API) to manage transaction across multiple resources like multiple databases.
   * When integrated with transaction management becomes simple. Using @Transactional annotation we can define transaction boundaries and spring handles Commit and Rollback automatically.
2. Why @Transactional is preferred.
   * Declarative Management: reduces boilerplate code.
   * Automatic commit and rollback
   * Flexibility in propagation
   * Compatibility with multiple resources: Works seamlessly with JDBC, Hibernate, JPA and other persistence APIs.
3. What is the difference between session.beginTransaction() and session.getTransaction().begin()?
   * Transaction management is handled using Session interface, which provides methods to mange transactions both session.beginTransaction() and session.getTransaction().begin() method are used to initiate transaction, but they differ in how they are invoked and managed.
   * Functionally, there is no major difference between these two. And choice between them often comes down to coding style and preferences.

A screenshot of a computer

Description automatically generated

1. What is the purpose of commit() and rollback() in a Hibernate transaction?
   * The methods commit() and rollback() are used to finalize or undo the transaction.
   * Commit():
     1. Saves all the changes to the database permanently.
     2. Called after the successful execution of all operation in transaction
     3. The database is updated permanently with the changes made during transaction,
   * Rollback():
     1. Reverts all the changes made during transaction.
     2. Used when error or exception occurs during the transaction.
     3. Ensures database remains consistent by discarding the incomplete and erroneous changes.
2. What is the purpose of the @GeneratedValue annotation in Hibernate?
   * Used to specify generation strategy for primary key values of entity when new records are inserted.
   * Used in combination with @Id annotation.
   * The strategy attribute of @GeneratedValue annotation defines how the primary key values are generated.
     1. AUTO: Hibernate chooses generation strategy based on database dialect.
     2. IDENTITY: values generated using auto-increment column
     3. SEQUENCE: uses database sequence to generate value
     4. TABLE: Uses database table to generate unique values
   * Purpose of using annotation:
     1. Automatic Id generation:
     2. Database independence:
     3. Simplifies uses:
3. Explain the difference between @Transient and @Column annotations.
   * @Transient is used to mark the field in java class as non-persistent. Hibernate will ignore this field and will not be mapped to any column in database table.
   * @Column is used to map the persistent filed of an entity to a specific column in the database. It provides control over how the fields are represented in the database.
4. What is the @Version annotation used for in Hibernate?
   * @Version annotation in hibernate is used for optimistic locking, which helps to prevent lost objects In concurrent transactions.
   * How it works:
     1. Version field is added to the entity.
     2. Each time the entity is updated hibernate increments the version value.
     3. Before performing an update hibernate checks if the version in database matches version in entity:
        1. If they match, update is processed and version is incremented.
        2. If They don’t match OptimisticLockingExecption is thrown.
   * It provides consistency without requiring database level lcoking.
5. What are the diverse types of cascade operations in Hibernate? Provide examples.
   * Cascade: it defines how the operation performed on parent entities are propagated to the associated child entities.
   * Type of cascading

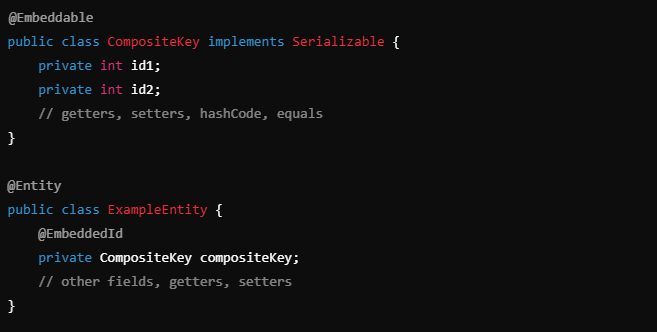
A screenshot of a computer

Description automatically generated

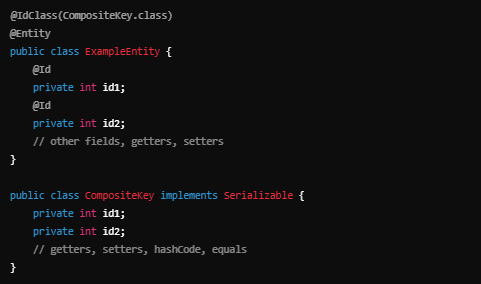
1. What is optimistic locking, and how is it implemented in Hibernate?
   * Optimistic locking is a concurrency control mechanism which is used to handle simultaneous access to the same data by multiple transactions.
   * It assumes that multiple transactions can complete without affecting each other, and therefore locks are not required until the transaction is ready to commit.
   * So instead of preventing access, it checks for conflicts at the time of transaction commit.
   * If any conflict occurs then the transaction is rollbacked or conflict prevention strategy is applied.
   * In Hibernate Optimistic locking is implemented using versioning mechanism.
   * Hibernate tracks the version of each entity and if transaction tries to modify the entity that has been modified by another transaction, a conflict is detected.

**Mapping Strategies:**

1. What is the difference between @ManyToOne and @OneToMany in terms of mapping and data retrieval?
   * **@ManyToOne**: Represents a many-to-one relationship, where many instances of the current entity are associated with one instance of another entity. It typically uses **EAGER** fetching by default, meaning the associated entity is fetched immediately when the parent entity is loaded.
   * **@OneToMany**: Represents a one-to-many relationship, where one instance of the current entity is associated with many instances of another entity. It uses **LAZY** fetching by default, meaning the associated collection is only fetched when explicitly accessed.
   * @ManyToOne establishes the "many" sides of the relationship, while @OneToMany establishes the "one" side, and their fetch strategies differ by default.
2. How does Hibernate map composite primary keys?
   * In hibernate composite primary keys are mapped using :
     1. @EmbeddedId and @Embeddable annotation
        1. Create separate class annotated with @Embeddable annotation that contains fields forming the primary key.
        2. In entity class use @EmbeddedId to reference the composite key.



* + 1. @IdClass:
       1. Define separate class to represent the composite key with @IdClass annotation where the class must have the same field name as the entity class.



* + Use @EmbeddedId for better encapsulation and cleaner code, especially when working with complex composite keys.
  + Use @IdClass for simpler scenarios or when migrating existing systems.

1. How can you map a java.util.Date or java.time.LocalDate in Hibernate?
   * You can make use of @Temporal annotation: use @Temporal annotation to specify the type of temporal data.

A screen shot of a computer

Description automatically generated

* + Database column types:
    1. TemporalType.DATE: Maps to a Date type in the database.
    2. TemporalType.TIME: maps to a Time type in the database
    3. TemporalType.TIMESTAMP: maps to a Timestamp type in database

1. What is the purpose of the @ElementCollection annotation in Hibernate?
   * Used to map the collection of basic or embeddable types to as separate table in the database
   * It is commonly used when entity need to maintain set of non-entity elements.
   * Class mapping:

A screenshot of a computer program

Description automatically generated

* + Database structure for above entity:

A screenshot of a computer

Description automatically generated

**Performance Tuning:**

1. How can you optimize Hibernate performance in terms of database access?
   * Use lazy loading for associations: to prevent fetching unnecessary data
   * Leveraging Caching: enable second level cache with cache providers like redis
   * Use projection and DTOs: Instead of fetching entire entities use projection to fetch only required fields of entity.
   * Batch Fetching: configure it for collection and associations for using @BatchSize annotation to minimize number of queries.
   * Avoid n+1 problem: use JOINS to fetch data in single queries instead of multiple queries
   * Use pagination : retrieve data in chunks using pagination methods like setFirstResult() and setMaxResult().
   * Connection Pooling : use connection pooling libraries like HikariCP for efficient database connection and reduced overhead.
   * Index database columns: add proper indexes on frequently searched columns like foreign keys, primary keys.
   * Optimize queries:
   * Audit and monitor queries: use profiling tool to identity slow queries and optimize them.
   * Set Connection timeouts: to avoid long running and hanging queries.
   * Avoid Auto flush mode: control manually using flush() method to avoid excessive writes.
2. What is the use of @Query annotation in Spring Data JPA with Hibernate?
   * @Query is used to define custom database queries directly in the repository interface
   * It is useful when default query derivation from method name is not possible for complex queries.
   * Uses:
     1. Custom JPQL/HQL queries: You can write complex JPQL (Java Persistence Query Language) or HQL (Hibernate Query Language) queries for entity-based operations.



* + 1. Custom native quries: You can write complex JPQL (Java Persistence Query Language) or HQL (Hibernate Query Language) queries for entity-based operations.



* + 1. Projection Support: Custom queries can return DTOs or projections by specifying the required fields in the query.



* + 1. Dynamic query parameters using @Param annotation



1. How do you handle large datasets in Hibernate (e.g., pagination)?
   * Pagination with setFirstResult() and setMaxResults():
     1. **Pagination** allows you to split the dataset into smaller chunks and retrieve a specific page of results at a time. This is achieved using the setFirstResult() and setMaxResults() methods in Hibernate to specify the starting index and the maximum number of records to retrieve.

A screen shot of a computer code

Description automatically generated

* + Using Criteria API for Pagination: You can use the **Criteria API** to handle pagination in a more object-oriented way

A black screen with white text

Description automatically generated

* + Use @Query with Pagination in Spring Data JPA: In Spring Data JPA, you can use the Pageable interface to easily implement pagination. It can be used in conjunction with the @Query annotation to define custom queries with pagination.





* + Limit data retrieval : **Select only required columns**: Instead of retrieving entire entities, you can fetch only the columns you need, which will reduce the size of the result set. This can be done using projections or native SQL queries.



* + Batch Processing for Updates/Deletes: For **batch updates** or **deletes**, use Hibernate’s batch processing to execute operations on large datasets without loading them entirely into memory. This helps in improving performance for bulk operations.

A screen shot of a computer

Description automatically generated

* + Caching: **First-level cache** (session cache) and **second-level cache** (shared across sessions) help in reducing database access when data is frequently accessed. This can significantly reduce the load on the database.

1. How does Hibernate handle connection pooling, and how can you configure it?
   * Hibernate does not manage connection pooling directly; instead, it relies on a **connection pool manager** (such as **HikariCP**, **C3P0**, **DBCP**, or **BoneCP**) to manage database connections.
   * The connection pool helps to improve application performance by reusing a set of established connections, reducing the overhead of establishing new connections for every database request.
   * Connection pools maintain a **set of database connections** and keep them open so that when Hibernate needs a connection, it can reuse one from the pool instead of creating a new connection.
   * Once the database operation is done, the connection is returned to the pool rather than being closed.
   * The pool manager controls the **maximum number of connections**, **minimum idle connections**, and **connection timeout** parameters, which are used to efficiently manage the number of active database connections.
   * Configuring connection pool: we can configure connection pool using hibernate.cfg.xml or using application.properties file or programmatically in spring boot using @configuration and @Bean annotation to create data source

A screen shot of a computer program

Description automatically generated

* + Key parameters for connection pooling configuration:
    1. **maximum-pool-size**: Defines the maximum number of connections that can be in the pool at any time.
    2. **minimum-idle**: Defines the minimum number of idle connections that the pool should maintain.
    3. **idle-timeout**: Specifies the amount of time a connection can be idle before being removed from the pool.
    4. **connection-timeout**: Defines how long a thread will wait for a connection before throwing an exception if no connection is available.
    5. **max-lifetime**: Specifies the maximum lifetime of a connection in the pool. After this time, the connection will be closed and a new one will be created.

1. How can you prevent Hibernate from executing unnecessary SQL queries?
   * Use Lazy loading: abling **lazy loading** ensures that associated entities are only loaded when explicitly accessed.
   * Avoid N+1 Query Problem: The N+1 problem occurs when fetching a collection of entities, causing Hibernate to execute one query for the parent and additional queries for each child entity. This can be prevented by using:
     1. Join Fetch: Use JOIN FETCH in HQL or JPQL to fetch the associations in a single query.



* + 1. Entity Graph: Use JPA entity graphs to specify which associations to fetch, avoiding unnecessary SQL execution.



* + Set Proper Fetch Strategy: Choose the appropriate fetch strategy for associations to avoid unnecessary loading.
  + Use projections:
  + Use cache effectively.
  + Optimize queries with batch processing.
  + Disable Auto-Flush and Auto-Commit: Hibernate may flush changes to the database automatically during query execution. Disabling auto-flushing can reduce unnecessary SQL execution.

A black background with white text

Description automatically generated

1. What is @EntityGraph annotation?
   * The @EntityGraph annotation in Spring Data JPA is used to define a JPA Entity Graph, which specifies which related entities should be eagerly loaded when querying for an entity.
   * It helps to control the fetching strategy of associations (like @OneToMany, @ManyToOne, etc.) and avoid the N+1 query problem or unnecessary SQL queries.
   * E.g.

A computer screen with white text

Description automatically generated

In this example, answers is lazily loaded, meaning a separate SQL query will be executed when accessing the answers list for a Question. We can use an entity graph to eagerly load the answers when fetching a Question.

A black screen with white text

Description automatically generated

In this case, the findById method will eagerly fetch the answers association based on the defined entity graph.

================================================================================