**How to add second level cache in hibernate?**

Ehcache uses Last Recently Used (LRU) as the default eviction strategy for the memory stores.

1.add dependency in pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

<dependency>

<groupId>javax.cache</groupId>

<artifactId>cache-api</artifactId>

</dependency>

2.write a configuration class

@Configuration

@EnableCaching

public class EhcacheConfig {

}

3.we need to specify the class which needs to be cacheable

@Cacheable(value = "areaOfCircleCache", key = "#radius", condition = "#radius > 5")

4.need to specify ehcache.xml with cache specification like event ordering like below

<event-firing-mode>ASYNCHRONOUS</event-firing-mode>

<event-ordering-mode>UNORDERED</event-ordering-mode>

<events-to-fire-on>CREATED</events-to-fire-on>

<events-to-fire-on>EXPIRED</events-to-fire-on>

<events-to-fire-on>EVICTED</events-to-fire-on>

5.need to specify the location of ehcache.xml location inapplication.properties file

spring.cache.jcache.config=classpath:ehcache.xml

annotations in springboot

@Cacheable -A cache entry is placed in the cache

@CacheEvict -A cache can become very large very quickly. The problem with large caches is that they occupy a lot of important main memory and mostly consist of stale data that is no longer needed.

@CacheEvict(allEntries=true)

@CachePut -for the values which are present in cache for updation purpose

@CacheConfig -this is used in class level if the cache configuration in same for all in the class

**Connection pooling**

We can integrate Apache’s connection pooling solutions (DBCP) or can use the C3P0 pooling framework with Hibernate

**Transaction propagation levels in hibernate**

|  |  |
| --- | --- |
| **REQUIRED** | **Always executes in a transaction.** If there is any existing transaction it uses it. If none exists then only a new one is created  Eg:registering new user |
| SUPPORTS | It may or may not run in a transaction. If current transaction exists then it is supported. If none exists then gets executed with out transaction.  **if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having SUPPORT propagation then registerUser and addNote will make use of existing transaction created from addNoteToSpecficUser** |
| **NOT\_SUPPORTED** | **Always executes without a transaction.** If there is any existing transaction it gets suspended  if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having NOT\_SUPPORTED propagation then registerUser and addNote will not make use of existing transaction created from addNoteToSpecficUser and nor it will creates it's own, rather it runs without any transaction. |
| **REQUIRES\_NEW** | **Always executes in a new transaction.** If there is any existing transaction it gets suspended  In this case, if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having REQUIRES\_NEW propagation then registerUser and addNote will always creates it's own transaction and doesn't utilizes the existing/calling service transaction. |
| **NEVER** | **Always executes with out any transaction. It throws an exception if there is an existing transaction**  In this case, if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having NEVER propagation then registerUser and addNote will not make use of existing transaction rather it will throw EXECEPTION. And if calling service addNoteToSpecficUser doesn't have any transaction, then registerUser and addNote will not create it's own transaction and it'll run without transaction. |
| **MANDATORY** | **Always executes in a transaction. If there is any existing transaction it is used. If there is no existing transaction it will throw an exception.**  In this case, if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having MANDATORY propagation then registerUser and addNote will make use of existing transaction. And if calling service (addNoteToSpecficUser) doesn't have trsation then registerUser and addNote having MANDATORY propagation, will throw EXCEPTION.  So i short, calling service (addNoteToSpecficUser) should transaction else service which calls calling service will throw exception. |

**Criteria query**

 criteria = session.createCriteria(Employee.class);

        criteria.add(Restrictions.gt("salary",3000));

        employee = criteria.list();

**isolation levels**

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation Level** | **Phantom Read**  Transaction (T1) executes same query twice and the result sets is different each time, probably because another transaction (T2) has either added or deleted the records. | **Unrepeatable Read**  transaction (T1) reads the same row twice and state of row is different probably because another transaction (T2) updates the row. Special case would be if T1 updates and committed the same data again, then updates done by T2 will be lost. | **Dirty Read**   One transaction (T1) reads the uncommitted data (updates are done, but not committed) by another transaction (T2) |
| Read Uncommitted | Allowed | Allowed | Allowed |
| Read Committed | Allowed | Allowed | Not Allowed |
| Repeatable Read | Allowed | Not Allowed | Not Allowed |
| Serializable | Not allowed | Not allowed | Not allowed |

**Hibernate mappings**

**Many to one**

@Entity

**public** **class** OrderItem {

    @ManyToOne

 @JoinColumn(name = “fk\_order”)

**private** Order order;

**one to Many**

@Entity

**public** **class** Order {

    @OneToMany

**private** List<OrderItem> items = **new** ArrayList<OrderItem>();

**Many to many**

@Entity

**public** **class** Store {

    @ManyToMany

    @JoinTable(name = “store\_product”,

           joinColumns = { @JoinColumn(name = “fk\_store”) },

           inverseJoinColumns = { @JoinColumn(name = “fk\_product”) })

**private** Set<Product> products = **new** HashSet<Product>();

@Entity

**public** **class** Product{

    @ManyToMany(mappedBy=”products”)

**private** Set<Store> stores = **new** HashSet<Store>();

**ONE TO ONE**

@Entity

**public** **class** Customer{

    @OneToOne

    @JoinColumn(name = “fk\_shippingaddress”)

**private** ShippingAddress shippingAddress;

**Entity States**

transient , persistent , detached

**Save /persist**

| **Basis of Difference** | **Save in Hibernate** | **Persist in Hibernate** |
| --- | --- | --- |
| **Return type** | The return type pertaining to the save () is serializable object. | The return type of persist () method is void. |
| **Assigning of identifier value** | Th save() method allows for the assigning of identifier value instantly. | The persist() method fails to guarantee that an identifier value is assigned to its persistent state instantly. |
| **Execution of insert query** | The save() method provides an identifier with the intent of an insert query being executed immediately for getting the identifier. It does not matter whether it is outside or inside a transaction. | The persist() method fails to execute a given insert query in case it is placed outside transaction boundaries. |
| **Utility** | The save method proves to be of less use in a long-running conversation that has extended a given Session context. | As the persist method is called outside the transaction boundaries, it is utilized in long-running conversations that offer an extended Session context. |
| **Support** | Save() method gets support only through Hibernate. | Persist() method is aptly supported by JPA |

**Composite Primary Key**

* The ID class must be public
* It must implement Serializable interface
* It must have a no-argument constructor
* It must override equals() and hashCode() methods.

Way 1

public class AirportID implements Serializable { -values are present column

@IdClass(AirportID.**class**) annotation needs to be present in entity

Way 2

@Embeddable

**public** **class** PhoneID **implements** Serializable {

@Entity

@Table(name = "phone\_contacts")

**public** **class** PhoneContact {

  @EmbeddedId

**private** PhoneID id;

using the @Embeddable and @EmbeddedId annotations, we don’t have to repeat the composite ID’s fields in the entity class.

Index

**CREATE** INDEX index\_name /\* Create Index \*/

**ON** table\_name (column\_1, column\_2);

**DROP** INDEX index\_name; /\* Drop Index \*/

Generation Type

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

**Hibernate N+1 issue:**

If there is lazy association in between two entities for example employee and addree entities.first hibernate will query to get all employess and then when I click to get address of each employee it will query n times to get address of n employees.Below are the ways to solve

Ways to solve the problem

1.@BatchSize(size = 10)

2.left join

**Cascading Types**

1. **CascadeType.ALL**
2. **CascadeType.PERSIST**
3. **CascadeType.MERGE**
4. **CascadeType.REMOVE**
5. **CascadeType.REFRESH**
6. **CascadeType.DETACH**
7. **CascadeType.REPLICATE**
8. **CascadeType.SAVE\_UPDATE**