Reference links

<https://hibernate.org/orm/releases/5.6/> latest

|  |  |
| --- | --- |
| till spring boot 2.9 & hib5 all anno are from  import javax.persistence.\*;  Starting spring boot 3 & Hib 6 all @Entity anno will be from  import jakarta.persistence.\*; | Hibernate 5= spring 5 =spring boot 2.7 (always rem spring boot version is half of spr)  Till spring boot 2.7 hibernate compatible version = hib5.9.x = spring 5.3  starting spring boot 3 = hibernate is 6  latest hibernate is 6.1 for spring boot 3, Hibernate 6.1 by default which requires Jakarta EE  all those packages from javax have been migrated to jakarta |
| main hibernate site | <https://hibernate.org/orm/documentation> |
| Full documentation | <https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html> |
| spring data jpa | <https://docs.spring.io/spring-data/jpa/reference/index.html> |
| all spring properties found here | <https://docs.spring.io/spring-boot/appendix/application-properties/index.html> |
| Trainer git hub link – all his hib proj available here | <https://github.com/bharaththippireddy> |
| hibernate props | All hibernate properties are available here  <https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/session-configuration.html>  <https://docs.jboss.org/hibernate/orm/current/quickstart/html_single/> |
| jpa finder methods keywords | <https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html> |
|  | <https://backendhance.com/en/blog/2023/open-session-in-view/#but-what-about-performance-panic> |
| oustanding EH cache | <https://www.ehcache.org/> , <https://www.ehcache.org/documentation/3.10/getting-started.html>  <https://www.ehcache.org/documentation/3.10/107.html>  <https://github.com/ehcache/ehcache3/tree/mydyn_3.9> |
|  |  |

Udemy course name :- Spring Date JPA using hibernate

|  |  |
| --- | --- |
| sir programs from book | <https://github.com/ctudose/java-persistence-spring-data-hibernate> |
|  |  |

Book & course reading status

|  |  |
| --- | --- |
| Front matter done |  |
| chapter -2 | completed |
| chapter -3 | more theory, skipped for now |
| chater -4 |  |

Spring data jpa is developed on top of spr data commons

Udemy video -98

Theory

Whether the SQL was written by hand and embedded in the Java code or generated on the fly by Java code, you use the JDBC API to bind arguments when preparing query parameters, executing a query, scrolling through query results, retrieving values from a result set, and so on. These are low-level data access tasks; as application engineers, we’re more interested in the business problem that requires this data access. What we’d really like to write is code that saves and retrieves instances of our classes, relieving us of this low-level labor.

Hibernate is a ORM tool- which can map the objects and relational database

Object/relational mapping (ORM) is a programming technique for making the connection between the incompatible worlds of object-oriented systems and relational databases.

|  |  |
| --- | --- |
| Hibernate ORM | Hibernate ORM consists of a core, a base service for persistence with SQL databases, and a native proprietary API. |
| Hibernate EntityManager | Hibernate’s native features are a superset of the JPA persistence features in every respect. |
| Hibernate Validator |  |
| Hibernate Envers |  |
| Hibernate Search | —Hibernate Search keeps an index of the domain model data up to date in an Apache Lucene database. |
| Hibernate OGM | —This Hibernate project is an object/grid mapper. It provides JPA support for NoSQL solutions, reusing the Hibernate core engine but persisting mapped entities into key/value-, document-, or graph-oriented data stores. |

A persistence unit is a pairing of our domain model class mappings with a database connection, plus some other configuration settings. Every application has at least one persistence unit; some applications have several if they’re talking to several (logical or physical) databases.

Important points

For batch applications / mission critical applications we should use only JDBC not jpa, because in jpa there is some cost associated with mapping each row with table

For all the common operations, hibernate would have already implemented the solution ex:- to check whether that emp exists or not –we already have exists() method in CRUDRepository

|  |  |
| --- | --- |
|  |  |

1. Session object= connection ++
2. any object fetched by jpa all those objects will be stored in session memory
3. hibernate may be slow because of session / session level caching – In a transaction, when session is opened, all finder methods/ database query outputs will be cached

ex:- select query output is cached/ update query output is cached …even delete

Spring boot autocfg

As part of spring boot auto cfg- These beans are getting created automatically 🡪 data source bean, entity manager factory, transaction manager,

Real time standards

1. In real time code, we will not fetch all columns, - we fetch only few columns – so always we should use projections with finder methods

List<StudenSomeFields> getStudents(); // if u give student, all the fields declared class will the fetch

If u give return type as a class which contains few fields – then only those fields will be fetched

1. Note:-
2. <property name="hibernate.connection.pool\_size">50</property> in real time configure the correct connection pool size

|  |  |  |  |
| --- | --- | --- | --- |
| Operation:- select  to fetch the data we have 3 options | finder methods /derived methods   * Developer friendly for small queries | jpql   * Use this when queries are becoming complex | native queries |
| to update | here we don’t have any updateBy() methods  we always have to fetch and modify and again save- its 2 step , un- necessarily we are fetching   * If u fetch and update   Person p= repo.findByName(“mani”);  p.setSalary(400)  personRepo.save(p) – if u pass full object it will take more time as it needs to compare what has changed from previous state , it should maintain the old state too | 1. this is best because here we can directly write update query (single operation)   instead of fetch and modify and save (2 operations)   1. bit speed- as no comparision with old object to see which fields has been changed |  |

**For performance optimization Always prefer lazy loading within a transaction.** Don’t unneccesarily load all the child data This is the most important rule.

If u do eager loading all parent and childs will be fetched at a time – this causes loading un necessary data, so avoid it

JDBc

|  |  |
| --- | --- |
| the JDBC RowSet makes CRUD operations even easier. |  |
|  |  |

Db design

Always normalize tables

1. Don’t duplicate the data / avoid redundant data- so that in future it is difficult to update
2. All the columns of a table must depend only on primary key

Main advantages/features

|  |  |
| --- | --- |
| Dirty checking | when an object is associated with session / when an obj is in transient state,  after making any changes and while commiting/ saving that entity , hib will compare the latest object with snapshot (obj when it is loaded) and it will identify which fields are changed  so update query will be framed only for the changed fields |
| Generators | only in hibernate we have generators support, in jdbc there is no generator support, |
| Session level cache | for every transaction / for every session 1 cache will be available  so all data will be kept in that L1 cache, 2nd time if same query fired it will fetch from that L1 cache, instead of hitting database again |
| session factory level cache | if u want same data for all sessions , then keep in ses fact cache, as this is long lived, but ses cache will be deleted when ses is closed |

Problems

1. Heap consumption & additional overhead to store data in cache- Be careful while enabling session factory level cache, if u enable it all those annotated entities will be cached , ensure to delete those cached items
2. Session cache- during a txn all the get/ load method output data will be cached and stored in session level cache (by default all data will be cached )and data in cache will be deleted after txn completes- this may cause latency

**bankAccountRepo**.save(konda);

1. Every time before performing save operation, it performs fetch , to see if record is there or not, if present it will fire – update query, else insert query
2. As per book, Spring data jpa takes more time to execute insert and update queries than hibernate, jpa – so if project is mission critical prefer plain hib or jdbc

Why spring data is slow?

Sring Data JPA's repositories and abstractions might involve additional processing, such as finder method name parsing, proxy creation for repo classes, and dynamic query generation for finder methods.

with practically no difference between Hibernate and JPA so u can use any other than spring for more performance

Main Interfaces

|  |  |
| --- | --- |
| Main interfaces | CrudRepository |
| all paging and sorting related interface | PagingAndSortingRepository  Pageable (i)  |  PageRequest (c) |
| JpaRepository | interface *JpaRepository*<*T*, *ID*> extends *ListCrudRepository*<*T*, *ID*>, *ListPagingAndSortingRepository*<*T*, *ID*>, *QueryByExampleExecutor*<*T*>  JpaRepository offers JPA-related methods, such as flushing the persistence context and deleting records in a batch. Additionally, JpaRepository overwrites a few methods from CrudRepository, such as findAll, findAllById, and saveAll to return List instead of Iterable |
|  |  |

Annotations

|  |  |
| --- | --- |
| @entity  @Table, @Column | @Entity means mapping a table with class  @Table, is required only when table name is diff from class name  @Column is required only when Column name is diff from java fields name |
| @Query | @Modifying |
| @PrimaryKeyJoinColumn | if we want to map child table foreign key to parent table primary key from java entities, (actually we can do this through script also)  we should use this annotation in 1-many cases, inheritance parent to child tables |
| @NoRepositoryBean | |  |  | | --- | --- | | @NoRepositoryBean  public interface BaseRepository<T, ID> extends JpaRepository<T, ID> {  // Common methods for all your repositories  List<T> findActive();  } | | | public interface CustomerRepository extends BaseRepository<Customer, Long> {  // Customer-specific methods  } | public interface ProductRepository extends BaseRepository<Product, Long> {  // Product-specific methods  } |   By adding @NoRepositoryBean to the BaseRepository interface, you tell Spring Data: "This interface is just a marker interface for sharing common repository methods. Its like advicing jpa to not to create a repository bean for it and don’t even create a proxy class for it." |
| @AttributeOverride | @AttributeOverride(  name = "owner",  column = @Column(name = "CC\_OWNER", nullable = false))  we can override fields In 2 scenarios – in inheritance mappings, and when we are embedding 1 class into another class   |  |  | | --- | --- | |  | 07601 | |  | @Embeddable  public class Address {  public class User {  @Embedded  @AttributeOverride(name = "street",  column = @Column(name = "BILLING\_STREET"))  private Address billingAddress; | |
| @Embeddable | with this anno we are saying don’t create a separate table for this  this is not a direct table, but the fields declared in this entity needs to be mapped into another table  @Embedded  Address add; this  @Embedded says import those columns into current entity  so finally with this, from java side we are defining 2 classes, but in db side we are creating 1 table |
| @JoinColumn | Generally foreign key col is the join column which joins current table with with parent table pk col  so @JoinColumn is to define foreign key col- use @JoinColumn(name = "author\_id")  here this means author\_id will be the foreign key column |
| @Temporal | this annotation is required only on top of java.util.Date &  @Temporal(TemporalType.DATE)  private Date eventDate;  @Temporal(TemporalType.DATE): Indicates that eventDate should be stored as a DATE in the database (e.g., "2023-10-27")  **No Need of @Temporal Annotation for LocalDate API:** Importantly, you'll notice that the @Temporal annotation is not used. Modern JPA implementations automatically recognize and handle these java.time types, mapping them to the appropriate database columns. |
| @Transactional | @Transactional creates a transactional boundary. Within this boundary, Spring manages the EntityManager and its associated Session |

Writing Tests

@RunWith(SpringRunner.class)

@SpringBootTest()

deleteByID()

Simple Crud operations

Exists() , deleteById(), count(), findOne()

Object states

|  |  |
| --- | --- |
| transient |  An object is in the transient state when it has been created using the new operator but is not yet associated with any Hibernate Session.   It has no corresponding row in the database.   Changes made to a transient object are not tracked by Hibernate. And changes made to that object it will not be synced to database row   Essentially, it's a plain Java object. |
| persistent / Managed |  An object enters the persistent state when it is associated with a Hibernate Session. This typically occurs through operations like save(), persist(), or get().   In this state, the object is actively tracked by Hibernate and all changes made to that object will be synced to database row on txn.commit()(There’s no need to call any save or update method explicitly just commit is enough).   Changes made to a persistent object are tracked this means this obj is managed by session  Employee emp = new Employee(); //Here this is transient object  ses.save(emp) //when we passed this to ses.save that obj will be tracked by hibernate as this is associated with a session object  emp.setSal(100) //  ses.getTrans().commit() ;; now eventhough u didn’t save the obj using ses.save(), since this object is managed by JPA,  when we commited all the new sal will be comitted bec it will internally compare the current object data with the snapshot saved (this is called dirty checking - Hibernate checks the persistence context for dirty state, and it executes the SQL UPDATE automatically to synchronize in-memory objects with the database state)  ses.emp(emp)  ses.detach(emp)// when u detach a particular obj, then that obj is not managed by hib any more  Entities in the “Managed” state: If an entity is already in a managed state and is associated with the current persistence context, any changes to it will be automatically synchronized with the database upon transaction completion. In such cases, explicitly calling save() is redundant.  Unintended updates: Calling save() on an already managed entity can result in unintended updates of all fields of the entity, even if they were not modified.(before making any update internally it will perform dirty checking) |
| detached | *Session* session = **entityManager**.unwrap(*Session*.class); session.evict(passport);   An object becomes detached when it was previously persistent/managed by jpa but is no longer associated with a Hibernate Session. This can happen when the Session is closed or the object is evicted from the Session.   Ex:-   A detached object still has a corresponding row in the database, but changes made to it are not automatically persisted when we issued txn.commit().   You can re-attach a detached object to a new Session to make it persistent again – like re onboarding to same client  **An object will become detached In JPA**  **End of Persistence Context (Transaction):**   * **Closing the EntityManager:**   + The most common reason is when the EntityManager that was managing the entity is closed. Once the EntityManager is closed, all entities it was managing become detached.’ * **Transaction Completion:**   + When a transaction completes (either by commit or rollback), the persistence context associated with that transaction typically ends. This results in the detachment of all managed entities.   **2. Explicit Detachment:**   * **EntityManager.detach(entity):**   + You can explicitly detach an entity using the detach() method of the EntityManager. This removes the entity from the persistence context, making it detached. * **EntityManager.clear():**   + Calling clear() on the EntityManager removes all entities from the persistence context, effectively detaching them.   (i.e., it’s in the “Detached” state), an explicit call to save() or merge() is required to save any modifications. |
| Removed |  An object enters the removed state when it is deleted from the database using the delete() method of the Session.   At this point, the object is no longer associated with the database.   Attempts to modify or re-associate a removed object with a Session may result in exceptions. |

|  |  |
| --- | --- |
|  | when u exe ses.save() / crudRepo.save(obj)  object will just be placed into hibernate persistence context(ses level cache) to track the state of this entity.  it wont immediately insert the records in db, it will insert only when we commit the txn |
|  |  |

Terminologies

|  |  |
| --- | --- |
| hibernate persistence context == session cache | this is a cache of objects that are currently being worked with.  JPA/Hibernate maintains a "persistence context," which is essentially a cache of entities that have been loaded or saved within the current transaction.  ex;- ses.save(obj)- now obj will be linked with persistence context/cache (initially obj from transient state will be moved to persistent state)  ses.load(obj) – now obj will be fetched from db and kept into cache |
|  |  |

JPA

Java Persistence API) is the specification defining an API that manages the persistence of objects and object/relational mappings

* Unwrapping means fetching the native objects using JPA EntityManager, like hib specific session object

Using entityManagerFactory.**unwrap**(SessionFactory.class); method u can get main native objects like session, sessionFactory

|  |  |
| --- | --- |
| sessionFactory extends EntityManagerFactory  EntityManager (JPA interface) ,  |  |  Session (hibernate interface)    The EntityManager is a standard JPA interface that provides methods for interacting with the persistence context (managing entities). It's designed to be vendor-neutral.   Session is a Hibernate-specific interface that provides more advanced and granular control over the persistence context. It offers functionalities beyond what the standard JPA EntityManager provides  @Autowired *EntityManager* **entityManager**;  @Autowired *SessionFactory* **sessionFactory**;  these 2 objects are provided can be autowired directly | *Session* session = **entityManager**.unwrap(*Session*.class); session.evict(passport);  you can "unwrap" the EntityManager to get access to the Hibernate Session.  this line of code retrieves the Hibernate Session object from the JPA EntityManager.  This is done when you need to use Hibernate-specific features that are not available through the standard JPA API.  ex:- session.evict(passport); evicting single obj is available inhib only, not in jpa  with jpa we can only evict all objects using entityManager.clear() not single object  In simpler terms, it's like having a universal remote (EntityManager) that can control many devices, and then using a special button (unwrap()) to get the original remote (Session) for a specific device (Hibernate) to access its advanced features. |
| plain jpa code – never used in real time  persistence.xml (not recommended because below are jpa properties with this we cant go for other propeties)  Path: Ch02/helloworld/src/main/resources/META-INF/persistence.xml    <persistence xmlns="http://java.sun.com/xml/ns/persistence"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  ➥ http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"  version="2.0">    <persistence-unit name="*ch02*"> Ⓐ  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> Ⓐ  <properties>  <property name="javax.persistence.jdbc.driver" Ⓑ  value="com.mysql.cj.jdbc.Driver"/> Ⓒ  <property name="javax.persistence.jdbc.url" Ⓓ  value="jdbc:mysql://localhost:3306/CH02?serverTimezone=UTC "/> Ⓓ  <property name="javax.persistence.jdbc.user" value="root"/> Ⓔ  <property name="javax.persistence.jdbc.password" value=""/> Ⓕ    <property name="hibernate.dialect" Ⓖ  value="org.hibernate.dialect.MySQL8Dialect"/> Ⓖ    <property name="hibernate.show\_sql" value="true"/> Ⓗ  <property name="hibernate.format\_sql" value="true"/> Ⓘ    <property name="hibernate.hbm2ddl.auto" value="create"/> Ⓙ  </properties>  </persistence-unit>    </persistence>  we can’t access all the configuration features of Hibernate with properties in persistence.xml  To configure the native Hibernate, we can use a hibernate.properties Java properties file or a hibernate.cfg.xml XML | *EntityManagerFactory emf = Ⓐ*  *Persistence.createEntityManagerFactory("ch02"); Ⓐ*    *try {*  *EntityManager em = emf.createEntityManager(); Ⓑ*  *em.getTransaction().begin(); Ⓒ*  *Message message = new Message(); Ⓓ*  *message.setText("Hello World!"); Ⓓ*  *em.persist(message); Ⓔ*  *em.getTransaction().commit();*  *//INSERT into MESSAGE (ID, TEXT) values (1, 'Hello World!')*    *em.getTransaction().begin(); Ⓖ*      *List<Message> messages = Ⓗ*  *em.createQuery("select m from Message m", Message.class) Ⓗ*  *.getResultList(); Ⓗ*  *//SELECT \* from MESSAGE Ⓗ*    *messages.get(messages.size() - 1). Ⓘ*  *setText("Hello World from JPA!"); Ⓘ*    *em.getTransaction().commit(); Ⓙ*  *//UPDATE MESSAGE set TEXT = 'Hello World from JPA!'*  *➥ where ID = 1*    *assertAll( Ⓚ*  *() -> assertEquals(1, messages.size()), Ⓚ*  *() -> assertEquals("Hello World from JPA!", Ⓛ*  *messages.get(0).getText()) Ⓛ*  *);*    *em.close();* |

Plain hib code

|  |  |
| --- | --- |
| we need one SessionFactory to interact with one database.  session factory (I) == entity manager factory |  |
| Path: Ch02/helloworld/src/test/resources/hibernate.cfg.xml  <?xml version='1.0' encoding='utf-8'?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD//EN"  ➥ "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration> Ⓐ  <session-factory> Ⓑ  <property name="hibernate.connection.driver\_class"> Ⓒ  com.mysql.cj.jdbc.Driver Ⓒ  </property> Ⓒ  <property name="hibernate.connection.url"> Ⓓ  jdbc:mysql://localhost:3306/CH02?serverTimezone=UTC Ⓓ  </property> Ⓓ  <property name="hibernate.connection.username">root</property> Ⓔ  <property name="hibernate.connection.password"></property> Ⓕ  <property name="hibernate.connection.pool\_size">50</property> Ⓖ  <property name="show\_sql">true</property> Ⓗ  <property name="hibernate.hbm2ddl.auto">create</property> Ⓘ  </session-factory>  </hibernate-configuration> | public class HelloWorldHibernateTest {    private static SessionFactory createSessionFactory() {  Configuration configuration = new Configuration(); Ⓐ  configuration.configure().addAnnotatedClass(Message.class); Ⓑ  ServiceRegistry serviceRegistry = new Ⓒ  StandardServiceRegistryBuilder(). Ⓒ  applySettings(configuration.getProperties()).build(); Ⓒ  return configuration.buildSessionFactory(serviceRegistry); Ⓓ  }    @Test  public void storeLoadMessage() {    try (SessionFactory sessionFactory = createSessionFactory(); Ⓔ  Session session = sessionFactory.openSession()) { Ⓕ    session.beginTransaction(); Ⓖ    Message message = new Message(); Ⓗ  message.setText("Hello World from Hibernate!"); Ⓗ    session.persist(message); Ⓘ    session.getTransaction().commit(); Ⓙ  // INSERT into MESSAGE (ID, TEXT)  // values (1, 'Hello World from Hibernate!')  session.beginTransaction(); Ⓚ    CriteriaQuery<Message> criteriaQuery = Ⓛ  session.getCriteriaBuilder().createQuery(Message.class); Ⓛ  criteriaQuery.from(Message.class); Ⓜ    List<Message> messages = Ⓝ  session.createQuery(criteriaQuery).getResultList(); Ⓝ  // SELECT \* from MESSAGE    session.getTransaction().commit(); Ⓞ    assertAll( Ⓟ  () -> assertEquals(1, messages.size()), Ⓟ  () -> assertEquals("Hello World from Hibernate!", Ⓠ  messages.get(0).getText()) Ⓠ  );    }  }  } |
| Session session = sessionFactory.openSession();  Transaction transaction = null;  try {  transaction = session.beginTransaction();  // Example: Fetch all records from a "User" table. Replace "User" with your entity class.  List<User> users = session.createQuery("from User", User.class).list();  for (User user : users) {  System.out.println("User ID: " + user.getId() + ", Name: " + user.getName());  // Access other user properties as needed.  }  transaction.commit(); |
| * + - 1. **Why session.save()** wont insert immediately places the object into Hibernate's persistent context/cache and schedules it for insertion.(it may not insert immediately) – it will insert when we perform tx.commit()- why it delays until txn commit – it will keep that obj into cache and tracks that object for any changes it will finally frames a single insert/update query and executes it, else   why its like it- if it runs the insert query immediately after ses.save() – after saving also developer can modify that object, since hib has to track that obj untill he commits the transaction – it will monitor that obj, if changes happens again it needs to run update query – so that case 2 queries will be fired, to avoid it- it will run single query at last  @Test @Transactional @Rollback(value = false) public void saveTest(){  BankAccount konda= BankAccount.*builder*()  .id(3)  .balance(1)  .firstName("a")  .lastName("b")  .build();   System.***out***.println("Before save");  *Session* ses= **entityManager**.unwrap(*Session*.class);  ses.save(konda);  *// bankAccountRepo.save(konda);* System.***out***.println("sop after save method");  System.***out***.println("method last line"); }  output  Before save  sop after save method  method last line  Hibernate: insert into bankaccount (bal, firstname, lastname, accno) values (?, ?, ?, ?)  notice here after last sop insert query fired  error in program…. | * + 1. The execution of the configure method will load the content of the default hibernate.cfg.xml file.     2. So here Build a SessionFactory using the configuration and the service registry we have previously created.   J-Synchronize the session with the database, and close the current session on commit of the transaction automatically. |

Spring boot

|  |  |
| --- | --- |
| As part of spring boot autoconfiguration, many beans will be auto configured by spring boot –datasource, LocalContainerEntityManagerFactoryBean(=entityManagerFactory/like session factory) , JpaTransactionManager, JpaVendorAdapter, | 1. spring boot will create all the main 5-10 beans automatically 2. it will take care of txn management using @Transactional anno, we don’t need to begin 3. it will take of closing session and session factory objects |
| @EnableJpaRepositories("com.manning.javapersistence.ch02.repositories") Ⓐ  public class SpringDataConfiguration {  @Bean  public DataSource dataSource() { Ⓑ  DriverManagerDataSource dataSource = new DriverManagerDataSource(); Ⓑ  dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver"); Ⓒ  dataSource.setUrl( Ⓓ  "jdbc:mysql://localhost:3306/CH02?serverTimezone=UTC"); Ⓓ  dataSource.setUsername("root"); Ⓔ  dataSource.setPassword(""); Ⓕ  return dataSource; Ⓑ  }    @Bean  public JpaTransactionManager Ⓖ  transactionManager(EntityManagerFactory emf) { Ⓖ  return new JpaTransactionManager(emf); Ⓖ  }    @Bean  public JpaVendorAdapter jpaVendorAdapter() { Ⓗ  HibernateJpaVendorAdapter jpaVendorAdapter = new Ⓗ  HibernateJpaVendorAdapter(); Ⓗ  jpaVendorAdapter.setDatabase(Database.MYSQL); Ⓘ  jpaVendorAdapter.setShowSql(true); Ⓙ  return jpaVendorAdapter; Ⓗ  }    @Bean  public LocalContainerEntityManagerFactoryBean entityManagerFactory(){ Ⓚ  LocalContainerEntityManagerFactoryBean Ⓚ  localContainerEntityManagerFactoryBean =  new LocalContainerEntityManagerFactoryBean(); Ⓛ  localContainerEntityManagerFactoryBean.setDataSource(dataSource());  Properties properties = new Properties(); Ⓜ  properties.put("hibernate.hbm2ddl.auto", "create"); Ⓜ  localContainerEntityManagerFactoryBean. Ⓜ  setJpaProperties(properties); Ⓜ  localContainerEntityManagerFactoryBean. Ⓝ  setJpaVendorAdapter(jpaVendorAdapter()); Ⓝ  localContainerEntityManagerFactoryBean. Ⓞ  setPackagesToScan("com.manning.javapersistence.ch02"); Ⓞ  return localContainerEntityManagerFactoryBean; Ⓚ  }  } | will crudRepository.save() trigger insert or update query immediately?  No, It will perform query execution when we commit the transaction,  these crudRepo.save(obj) will just push that object into hib pers context to make that obj tracked by hibernate  ex:-  @Test  @Transactional  @Rollback(value = false)  public void saveTest(){  BankAccount accnt= BankAccount.*builder*()  .id(3)  .balance(1)  .firstName("a")  .lastName("b")  .build();   System.***out***.println("Before save");  **bankAccountRepo**.save(accnt);  System.***out***.println("sop after save method");  accnt.setFirstName("kiran abbavaram"); *// entityManager.flush();* System.***out***.println("method last line");  }  output:-  Before save  Hibernate: select …where bankaccoun0\_.accno=?  sop after save method  method last line  Hibernate: insert into bankaccount (bal, firstname, lastname, accno) values (?, ?, ?, ?)  Notice here, the query executed at method end. |
|  |  |

Generators

If u don’t give any @GeneratedValue() annotation then hib assumes value to the primary key column will be given by programmer not framework

|  |  |
| --- | --- |
| @Id @GeneratedValue(strategy = *GenerationType*.***SEQUENCE***,generator = "my\_sequence") @SequenceGenerator(name = "my\_sequence",allocationSize = 1,sequenceName = "prod\_seq\_gen") Integer **productId**;   **SEQUENCE:** Good for most databases, offers flexibility and good performance.  we need to create a separate sequence in db , every time for insertion we will use this sequence and get the value  @GeneratedValue –means the value for this PK column will be generated by hib or spring and developer don’t need to give ,if u don  if u don’t give **@GeneratedValue** then we should manually assign a value to the primary key col | **SEQUENCE:** Uses a database sequence object (like Oracle sequences) to generate unique IDs.  Note:- mysql doesn’t support seq, but it will get the same effect by storing sequences in separate table |
| GenerationType.TABLE**:** More portable, but can be less performant.  along with existing employee table, we need to create a special table where the primary keys will be stored  like in sequence and new value will be incremented and stored in that special table  and that value will be used in employee table as primary key  create table id\_gen( gen\_name varchar(60) PRIMARY KEY, gen\_val int(20) )  @Id @GeneratedValue(strategy = *GenerationType*.***TABLE***,generator = "my\_sequence") @TableGenerator(name = "my\_sequence",table = "id\_gen",pkColumnName = "gen\_name",allocationSize = 1,valueColumnName = "gen\_value") Integer **productId**; | **TABLE:** Uses a dedicated table to simulate a sequence. This is a more portable approach but can be less performan  problem here is HB needs to insert the new primary in special table and the same id must be used in this employee table  instead of inserting the pk value in 2 tables better to use ID generator,  so that pk value will be auto incremented and stored in same table |
| How these table,sequence generators works internally? | 1. **It will get the latest value from seq / table , use it in emp table insertion** 2. **And increment the sequence value and store the updated value in same sequence object/ table**     **note every time that previous value will be updated**  scope:- this special table can store pk values for multiple tables  as this special table has 2 cols 🡪 pk latest value, pk name (same pk name for 1 table), |
| GenerationType.AUTO**:** Convenient, but you might not have full control over the strategy. |  **AUTO:** (Default) JPA will talk to dialect and choose the most appropriate strategy based on your database and how you've defined your primary key field. It often defaults to SEQUENCE or IDENTITY. |
| GenerationType.IDENTITY**:** Simple for databases that support it, but less flexible.  here db will give a new PK value by incrementing previous value – dev doesn’t even need to pass the pk value   * Here we must create a table with primary key with auto increment constraint   create table employee (  id int primary key AUTO\_INCREMENT  )  @GeneratedValue(strategy = GenerationType.IDENTITY)   * In my view identity generator is simple to use   CREATE TABLE Person (  id INT(10) NOT NULL AUTO\_INCREMENT,  Name VARCHAR(45) NOT NULL,  Product VARCHAR(45) DEFAULT NULL,  Country VARCHAR(25) DEFAULT NULL,  Year INT NOT NULL,  Salary INT NOT NULL,  PRIMARY KEY (id) -- Add the primary key constraint here  ); | **IDENTITY:** Relies on the database to auto-increment the EXISTING table primary key.  ex:- if we created a starting primary key with value “1” , then that pk value will be auto incremented- database will take care of auto incrementing the value – no need of separate sequence and no need of other tables to store pk  Note:- if u use auto increment , then in database also while inserting we don’t need to pass the pk value  INSERT INTO Person(Name, Product, Country, Year,Salary)  VALUES ('Stephen', 'Computer', 'USA', 2015,200),  ('Joseph', 'Laptop', 'India', 2016,500),  ('John', 'TV', 'USA', 2016,700),  ('Donald', 'Laptop', 'England', 2015,600),  ('Joseph', 'Mobile', 'India', 2015,159),  ('Peter', 'Mouse', 'England', 2016,999); |

Repository query keywords

## Supported query method subject keywords

The following table lists the subject keywords generally supported by the Spring Data repository query derivation mechanism to express the predicate. Consult the store-specific documentation for the exact list of supported keywords, because some keywords listed here might not be supported in a particular store.

|  |  | Table 1. Query subject keywords | |
| --- | --- | --- | --- |
|  |  | **Keyword** | **Description** |
|  |  | find…By, read…By, get…By, query…By, search…By, stream…By | General query method returning typically the repository type, a Collection or Streamable subtype or a result wrapper such as Page, GeoResults or any other store-specific result wrapper. Can be used as findBy…, findMyDomainTypeBy… or in combination with additional keywords. |
|  |  | exists…By | Exists projection, returning typically a boolean result. |
|  |  | count…By | Count projection returning a numeric result. |
|  |  | delete…By, remove…By | Delete query method returning either no result (void) or the delete count. |
|  |  | …First<number>…, …Top<number>… | Limit the query results to the first <number> of results. This keyword can occur in any place of the subject between find (and the other keywords) and by. |
|  |  | …Distinct… | Use a distinct query to return only unique results. Consult the store-specific documentation whether that feature is supported. This keyword can occur in any place of the subject between find (and the other keywords) and by. |

## Reserved methods

The following table lists reserved methods that use predefined functionality (as defined in CrudRepository). These methods are directly invoked on the backing (store-specific) implementation of the repository proxy. See also “[Defining Query Methods](https://docs.spring.io/spring-data/jpa/reference/repositories/query-methods-details.html#repositories.query-methods.reserved-methods)”.

|  |
| --- |
| Table 2. Reserved methods |
| deleteAllById(Iterable<ID> identifiers) |
| deleteById(ID identifier) |
| existsById(ID identifier) |
| findAllById(Iterable<ID> identifiers) |
| findById(ID identifier) |

## Supported query method predicate keywords and modifiers

The following table lists the predicate keywords generally supported by the Spring Data repository query derivation mechanism. However, consult the store-specific documentation for the exact list of supported keywords, because some keywords listed here might not be supported in a particular store.

| Table 3. Query predicate keywords | |
| --- | --- |
| **Logical keyword** | **Keyword expressions** |
| AND | And |
| OR | Or |
| AFTER | After, IsAfter |
| BEFORE | Before, IsBefore |
| CONTAINING | Containing, IsContaining, Contains |
| BETWEEN | Between, IsBetween |
| ENDING\_WITH | EndingWith, IsEndingWith, EndsWith |
| EXISTS | Exists |
| FALSE | False, IsFalse |
| GREATER\_THAN | GreaterThan, IsGreaterThan |
| GREATER\_THAN\_EQUALS | GreaterThanEqual, IsGreaterThanEqual |
| IN | In, IsIn |
| IS | Is, Equals, (or no keyword) |
| IS\_EMPTY | IsEmpty, Empty |
| IS\_NOT\_EMPTY | IsNotEmpty, NotEmpty |
| IS\_NOT\_NULL | NotNull, IsNotNull |
| IS\_NULL | Null, IsNull |
| LESS\_THAN | LessThan, IsLessThan |
| LESS\_THAN\_EQUAL | LessThanEqual, IsLessThanEqual |
| LIKE | Like, IsLike |
| NEAR | Near, IsNear |
| NOT | Not, IsNot |
| NOT\_IN | NotIn, IsNotIn |
| NOT\_LIKE | NotLike, IsNotLike |
| REGEX | Regex, MatchesRegex, Matches |
| STARTING\_WITH | StartingWith, IsStartingWith, StartsWith |
| TRUE | True, IsTrue |
| WITHIN | Within, IsWithin |

In addition to filter predicates, the following list of modifiers is supported:

| Table 4. Query predicate modifier keywords | |
| --- | --- |
| **Keyword** | **Description** |
| IgnoreCase, IgnoringCase | Used with a predicate keyword for case-insensitive comparison. |
| AllIgnoreCase, AllIgnoringCase | Ignore case for all suitable properties. Used somewhere in the query method predicate. |
| OrderBy… | Specify a static sorting order followed by the property path and direction (e. g. OrderByFirstnameAscLastnameDesc). |

Ways to fetch from database

|  |  |
| --- | --- |
| finderMethods – with projections to fetch only few columns by declaring a class which contains only few fields |  |
| with JPQL - @Query()  JPQL With projections - @Query() | I think this is best as if others read my code they will understand |
| with Native queries | if our jpql queries are becoming complex and if we have multiple joins then use native query only |
| Criteria Builder api |  |

Finder methods

With finder methods we just need to declare the method names in the interface with arguments, spring will provide the actual implementation , we should write the method names as per <https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html>

Problem here is:- Finder methods are designed to retrieve all columns, if we want only few columns then we have to write finder methods with projections either class or

Interface based projections

Finder methods supports delete keyword also , in all below operations we can replace findWith delete keyword

public *List*<Product> findByProductName(String *name*);

Note:-

* + 1. every finder method will internally execute in a txn, if there is an existing txn it will use that, else that finder method alone will be executed in a separate transaction
    2. Problem of Repetable Reads? - within a transaction, if you perform the same fetch operation multiple times, you'll get the same results bec data was cached

For Every transaction we have a separate cache called session cahe, once db is hit, the data will be cached in 1st, 2nd level cache

So inside a txn if same query is fired, then data will be fetched from ses cache, instead of db hit,

so in this case if u want to fetch the data by hitting db,

* either you should run that method in sep txn , bec sep txn will have separate ses cache, in that cache prev data wont be there, hence db hit will happen
* or since root issue is prev data was present in cache , simply evict that data from cache or clear the full ses cache using entityManager.clear()- in this case also since cache is empty it will freshly hit the database

What happens if we run every finder method in separate txn?

|  |  |
| --- | --- |
| Note:- if the finder method name is wrong  app wont even start | findBy<Column name here>  public *List*<Product> findByProductName(String *name*);  public *List*<Product> deleteByProductName(String *name*); |
| And condition | public *List*<Product> findByProductNameAndPrice(String *name*, Double *price*); |
| Case in sensitive search | List<Person> findByFirstnameIgnoreCaseAndLastnameIgnoreCase(String firstname, String lastname);  public Optional<Vendor> findFirst1ByNameIgnoreCase(String *name*);   * Ignore case for both parameters   List<Person> findByAllIgnoreCaseFirstnameAndLastname(String firstname, String lastname); |
| GreaterThan  GreaterThanEqual  LessThan  LessThanEqual | public *List*<Product> findByPriceGreaterThanEqual(Double *start*);  findByRegistrationDateLessThan  findByRegistrationDateLessThanEqual |
| Contains | public *List*<Product> findByProductNameContains(String *s*); |
| between | public *List*<Product> findByPriceBetween(Double *start*, Double *end*); |
| like – for multiple characters use %, for single char use \_ underscore  NotLike | public *List*<Product> findByProductNameLike(String *likeChar*);  findByUsernameNotLike -- where e.username not like ?1 |
| in | findByIdIn(List<Integer>) |
| For finder methods not only arguments, u can pass spring interfaces like paging and sorting  spring is capable to implement those methods |  |
|  |  |

|  |  |  |
| --- | --- | --- |
| Keyword | Is keyword is optional- whether u include “is “ keyword or not it doesn’t matter |  |
| Is, Equals | findByUsername  findByUsernameIs  findByUsernameEquals | wheree.username = ?1 |
| Or | findByUsernameOrRegistrationDate | where e.username = ?1 or e.registrationdate = ?2 |
| OrderBy | findByRegistrationDateOrderByUsernameDesc | where e.registrationdate = ?1 order by e.username desc |
| Before  After | findByRegistrationDateBefore  findByRegistrationDateAfter | where e.registrationdate < ?1  where e.registrationdate > ?1 |
| Null, IsNull | findByRegistrationDateIsNull  findByRegistrationDateNull | where e.registrationdate is null |
| NotNull, IsNotNull | findByRegistrationDateNotNull  findByRegistrationDateIsNotNull | where e.registrationdate is not null |
| Not | findByUsernameNot | where e.username <> ?1 |
| order by - Fetch top 1 records | findFirstByOrderByUsernameAsc();  findTopByOrderByRegistrationDateDesc(); | if u don’t mention any number after First/Top keywords then by default it will return only 1 record  User user2 = userRepository.findTopByOrderByRegistrationDateDesc();  here after top as we didn’t mention number of rec, it will fetch only 1 record |
| limit  Fetch top 2 records | * Here first /top both keywords are allowed   *List*<Person> findFirst3ByOrderBySalaryDesc();  *List*<Person> findTop3ByOrderBySalaryDesc(); | in db we use limit keyword & in java8 we use limit(anyNumber)  SELECT \* FROM aims.person order by salary desc limit 3; |
|  | findByCountry(String *country*,*Pageable pageable*);  findByCountry(String *country*,*sort sort*); | Even for finder methods we can pass pageable, sort as parameters |
|  | Streamable<User> findByEmailContaining(String text);  Streamable<User> findByLevel(int level); | these finder methods can directly return a stream instead of collection  try(Stream<User> result = Ⓐ  userRepository.findByEmailContaining("someother") Ⓐ  .and(userRepository.findByLevel(2)) Ⓑ  .stream().distinct()) { |

For deleting rows, you’d have to replace find with delete in the names of the methods

If you want to find a User and return it in an Optional container, the method return type will be Optional<User>

|  |  |
| --- | --- |
| A full list of possible return types, together with detailed explanations, can be found in appendix D of the Spring Data JPA reference documentation (<http://mng.bz/o51y>). | |
| void | Denotes no return value. |
| Primitives | Java primitives. |
| Wrapper types | Java wrapper types. |
| T | A unique entity. Expects the query method to return one result at most. If no result is found, null is returned. More than one result triggers an IncorrectResultSizeDataAccessException. |
| Iterator<T> | An Iterator. |
| Collection<T> | A Collection. |
| List<T> | A List. |
| Optional<T> | A Java 8 or Guava Optional. Expects the query method to return one result at most. If no result is found, Optional.empty() or Optional.absent() is returned. More than one result triggers an IncorrectResultSizeDataAccessException. |
| Option<T> | Either a Scala or Vavr Option type. Semantically the same behavior as Java 8’s Optional, described earlier. |
| Stream<T> | A Java 8 Stream. |
| Streamable<T> | A convenience extension of Iterable that directy exposes methods to stream, map and filter results, concatenate them etc. |
| Types that implement Streamable and take a Streamable constructor or factory method argument | Types that expose a constructor or ….of(…)/….valueOf(…) factory method taking a Streamable as argument. See [Returning Custom Streamable Wrapper Types](https://docs.spring.io/spring-data/jpa/docs/2.5.2/reference/html/#repositories.collections-and-iterables.streamable-wrapper) for details. |
| Vavr Seq, List, Map, Set | Vavr collection types. See [Support for Vavr Collections](https://docs.spring.io/spring-data/jpa/docs/2.5.2/reference/html/#repositories.collections-and-iterables.vavr) for details. |
| Future<T> | A Future. Expects a method to be annotated with @Async and requires Spring’s asynchronous method execution capability to be enabled. |
| CompletableFuture<T> | A Java 8 CompletableFuture. Expects a method to be annotated with @Async and requires Spring’s asynchronous method execution capability to be enabled. |
| ListenableFuture | A org.springframework.util.concurrent.ListenableFuture. Expects a method to be annotated with @Async and requires Spring’s asynchronous method execution capability to be enabled. |
| Slice<T> | A sized chunk of data with an indication of whether there is more data available. Requires a Pageable method parameter. |
| Page<T> | A Slice with additional information, such as the total number of results. Requires a Pageable method parameter. |
| GeoResult<T> | A result entry with additional information, such as the distance to a reference location. |
| GeoResults<T> | A list of GeoResult<T> with additional information, such as the average distance to a reference location. |
| GeoPage<T> | A Page with GeoResult<T>, such as the average distance to a reference location. |
| Mono<T> | A Project Reactor Mono emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, Mono.empty() is returned. More than one result triggers an IncorrectResultSizeDataAccessException. |
| Flux<T> | A Project Reactor Flux emitting zero, one, or many elements using reactive repositories. Queries returning Flux can emit also an infinite number of elements. |
| Single<T> | A RxJava Single emitting a single element using reactive repositories. Expects the query method to return one result at most. If no result is found, Mono.empty() is returned. More than one result triggers an IncorrectResultSizeDataAccessException. |
| Maybe<T> | A RxJava Maybe emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, Mono.empty() is returned. More than one result triggers an IncorrectResultSizeDataAccessException. |
| Flowable<T> | A Rx |

Finder() with projections

List<StudenSomeFields> getStudents(); // if u give student, all the fields declared class will the fetch

If u give return type as a class which contains few fields – then only those fields will be fetched

you need to create an interface or class that defines the subset of columns you want to retrieve.

|  |  |
| --- | --- |
| For finder methods both interface based projection and class based projections will work  It will internally identifies the return type of   * If it is interface again it will create a proxy for that interface | |
| Class based projection  public static class NameOnlyClass{  String **name**;  public NameOnlyClass(String *name*) {  System.***out***.println("calling cons");  this.**name** = *name*;  }  public String getName() {  return **name**;  }  }  it will geneuinely fetch only those columns which are mentioned in class | *//Class based projection  List*<Person.NameOnlyClass> findByCountry(String *country*);  **personRepository**.findByCountry("USA").forEach(*e*-> System.***out***.println(*e*.getName()));  – if it is a class based projection it will use constructor to create object,  so in future If u are using @Query(JPQL) – you must call that class constructor to initialise object like how jpa is doing here  Class based projection is recommended because here proxy class creation for that interface will be avoided   * Here defining constructor is mandatory that to only 1 – if we declare both parameterized and normal , it will get confused which cons to call * Jpa will internally see the return type if it is a class it will check all those params and it will fetch only those params * Internally query - Hibernate: select p1\_0.name from person p1\_0 where p1\_0.country=?   How it works   1. it will check return type class or interface and fetches only those fields and frames select <fields> 2. it will check finder methods and builds the where clause and it will executes query gets those columns 3. it will call the return class parameterised constructor to inject data in that class object |
| interface based projection  public static interface *IProductOnly*{  String getName() ;  Integer getSalary(); } | *// interface based projection – a proxy implementation class will be created for the declared interface*  *List*<Person.*IProductOnly*> findByProduct(String *productName*);  in this interface based projection  **personRepository**.findByProduct("Laptop").forEach(*e*-> System.***out***.println(*e*.getName() +"--"+*e*.getSalary()));   1. in interfaces after exec the query, to populate the data into interfaces, JPA will create a proxy child IMPL class in runtime   and it will populate the data into that proxy class  query - Hibernate: select p1\_0.name,p1\_0.salary from person p1\_0 where p1\_0.product=?  in interface based projection we can use spel  public interface UserSummary {    String getUsername(); Ⓐ    @Value("#{target.username} #{target.email}") Ⓑ  String getInfo(); Ⓑ    } |

|  |  |
| --- | --- |
| interface based projection | class based projection |
| public interface UserView {  String getFirstName();  String getLastName();  } | public class UserSummary {  private String firstName;  private String lastName;  private String fullName;  public UserSummary(String firstName, String lastName) {  this.firstName = firstName;  this.lastName = lastName;  this.fullName = firstName + " " + lastName;  }  // Getters for all fields  } |
| problems with interface based projection   1. closed projection – means interface properties(method names) must match with entity properties 2. cant define any extra methods other than entity col names |  |
|  | advantages   1. you can define constructors, custom methods (which can have some business login), u can happily include other properties that are not defined in entity |

Paging & Sorting

The first and top keywords (used equivalently) can limit the results of query methods. The top and first keywords may be followed by an optional numeric value to indicate the maximum result size to be returned. If this numeric value is missing, the result size will be 1.

Pageable is an interface for pagination information, but in practice we use the PageRequest class that implements it. This one can specify the page number, the page size, and the sorting criterion.

We’ll add the methods shown in listing 4.10 to the UserRepository interface.

Listing 4.10 Limiting query results, sorting, and paging

Path: Ch04/springdatajpa2/src/main/java/com/manning/javapersistence

➥ /springdatajpa/repositories/UserRepository.java

User findFirstByOrderByUsernameAsc();

User findTopByOrderByRegistrationDateDesc();

Page<User> findAll(Pageable pageable);

List<User> findFirst2ByLevel(int level, Sort sort);

List<User> findByLevel(int level, Sort sort);

List<User> findByActive(boolean active, Pageable pageable);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| all paging and sorting related interface | |  |  | | --- | --- | | Pageable (I)  |  PageRequest (C)  pageable like comparable interface | Streamable(I)  | |  Sort (C) Slice (I)  |  Page (I) |   interface *PagingAndSortingRepository*  *Page*<*T*> findAll(*Pageable pageable*);  *Iterable*<*T*> findAll(Sort *sort*);  } | PageRequest(pageNumber,records per page, Sort)  //page number starts with 0  PageRequest(4,10)- it will get 5th page, and that page has 10 records  // since we use order by in sql here sort by in jpa,  Sort soryByDesc = Sort.*by*(Sort.*Direction*.***DESC***, "country");  *Pageable* page= PageRequest.*of*(0,3, soryByDesc);  Here in this case exection will be  1st sorting of all records in that table will happen then  2) segregation into pages will start (like partitions)  3) and then desired page (here 0th page )will be rendered  to create sort and pageable objects we have only static of, by methods |

We can do 1 col sort by ascending, another column sort by descending

Ex- ex:- write an usecase of fetching all txn of a customer in month wise

|  |  |
| --- | --- |
|  |  |
|  |  |
| only paging without criteria  *Page*<*T*> findAll(*Pageable pageable*); | For every finder method not only our custom arguments we can take pageable, sort as arguments  SELECT \* FROM aims.person order by country desc , name asc  Sort sort = Sort.*by*( *List*.*of*(  new Sort.Order(Sort.*Direction*.***DESC***,"country"),  new Sort.Order(Sort.*Direction*.***ASC***,"name")  )); //generallly every order by will have col name and asc/ desc hence here also we should give both  *Pageable* page= PageRequest.*of*(0,100, sort); **personRepository**.findAll(page).stream().forEach(System.***out***::println);  SELECT \* FROM aims.person order by country desc , name asc ; |
| finder method with paging and sorting with Criteria :-  *List*<Person>  findByCountry(String *country*,*Pageable pageable*);   * This is the most used used in real time   ex:- searching for anything in amazon, only with search string we will show pagination results –  Hence we always search with criteria | For any finder methods u can declare pageable argument, as you are passing those arguments to those parameters this will work  Sort sortBySalAsc = Sort.*by*(Sort.*Direction*.***DESC***, "salary"); *Pageable* page2= PageRequest.*of*(1,2, sortBySalAsc); //we got 2nd page where each page has 2 records **personRepository**.findByCountry("USA",page2).stream().forEach(System.***out***::println);  internal query : select \* from aims.person where country="USA" order by salary desc  actual result  Person{id=1, name='Stephen', product='Computer', country='USA', year=2015, salary=200}  Person{id=7, name='Rama', product='Sweeper', country='USA', year=2015, salary=200}  # id, name, product, country, year, salary  '3', 'John', 'TV', 'USA', '2016', '700'  '11', 'Anuradha', 'keyboard', 'USA', '2016', '700'  '1', 'Stephen', 'Computer', 'USA', '2015', '200'  '7', 'Rama', 'Sweeper', 'USA', '2015', '200' |
| *only Sorting without criteria*  *Iterable*<*T*> findAll(Sort *sort*); | Sort sort = Sort.*by*( *List*.*of*(  new Sort.Order(Sort.*Direction*.***DESC***,"country"),  new Sort.Order(Sort.*Direction*.***ASC***,"name")  ));  **personRepository**.findAll(sort).forEach(System.***out***::println);  the problem with this is it wont do paging it will just fetch all records  SELECT \* FROM aims.person order by country desc , name asc ; |
| finder Method with sorting with criteria –  *List*<Person> findTop2ByProduct(String *product*, Sort *sortList*); | Sort orderByCountryAndName= Sort.*by*(*List*.*of*(  new Sort.Order(Sort.*Direction*.***DESC***,"country"),  new Sort.Order(Sort.*Direction*.***DESC***,"name") )); **personRepository**.findTop2ByProduct("Laptop",orderByCountryAndName).forEach(System.***out***::println);  result  Person{id=2, name='Joseph', product='Laptop', country='India', year=2016, salary=500}  Person{id=4, name='Donald', product='Laptop', country='England', year=2015, salary=600}  select \* from person where product = "Laptop" order by country desc , name desc ;   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 2 | Joseph | Laptop | India | 2016 | 500 | | 4 | Donald | Laptop | England | 2015 | 600 | | 9 | Charan | Laptop | England | 2016 | 600 | |
|  |  |

Save() , flush()

Ses.save() /ses.persist() / crudRepo.save(obj) will just place the object into persistence context/ session cache & it will not insert the record immediately,

obj will be inserted/query will be executed only when we do txn.commit()

|  |  |
| --- | --- |
| without flush  @Test @Transactional @Rollback(value = false) public void saveTest(){  BankAccount accnt= BankAccount.*builder*()  .id(5)  .balance(1)  .firstName("a")  .lastName("b")  .build();   System.***out***.println("Before save");  **bankAccountRepo**.save(accnt);  System.***out***.println("sop after save method");  accnt.setFirstName("kiran abbavaram");// update prev cached obj and re-place in cache  System.***out***.println("method last line");  **bankAccountRepo**.save(accnt); // update prev obj and place in cache- not insert in db  accnt.setLastName("abbare");  accnt.setBalance(1000);  **bankAccountRepo**.save(accnt); }  result  Before save  Hibernate: select bankaccoun0..where bankaccoun0\_.accno=?  sop after save method  method last line  Hibernate: insert into bankaccount (bal, firstname, lastname, accno) values (?, ?, ?, ?)  Hibernate: update bankaccount set bal=?, firstname=?, lastname=? where accno=?  points to note   1. Ee eventhough we wrote bankAccountRepo.save(accnt); 3 times –object was placed & updated in cache and update query fired only 1 time, eventhough we kept 2 times into cache 2. Repo.save() will | with flush  flush yet to read |

JPQL / @Query

Java persistence query language

Select \* from Employee (here we will write query with entity class name instead of original names)

JPQL is for both DQL(select queries) and DML (insert queries)

JPQL is case sensitive for class names and variable names, but it is not case sensitive for sql keywords like into, Like, not, joins..

Since we are writing query , method name can be anything, no need to follow any finder method standards

If the jpql query written in @Query is wrong then app will start successfully, but at runtime while executing the query we will get runtime exception

@Query supports SPEL – where instead of hardcoding table name we can use #{#entityName}

|  |  |
| --- | --- |
| @Query("from Person")  *List*<Person> getAllPersons();   * When we use @Query() as we are giving query directly, we don’t need to follow any finder method standards | – this Person is java entity class name not table name the fields declare here are java entity class properties names, not column names |

Spring Data JPA supports Spring Expression Language (SpEL) expressions in queries defined using the @Query annotation, and Spring Data JPA supports the entityName variable. In a query such as select e from #{#entityName} e, entityName is resolved based on the @Entity annotation. In our case, in UserRepository extends JpaRepository<User, Long>, entityName will resolve to User.

|  |  |
| --- | --- |
| using path params in query | @Query("select count(u) from User u where u.active = ?1")  int findNumberOfUsersByActivity(boolean active); |
| using name params In query | @Query("select u from User u where u.level = :level and u.active = :active")  List<User> findByLevelAndActive(@Param("level") int level, @Param("active") boolean active);  @Query( "select new com.nt.dto.PersonNameOnly(p.name) from #{#entityName} p where salary=:**salary**") *List*<PersonNameOnly> getPerNameWithNamedParamUsingClassProjection(@Param("**salary**") Integer *sal*);  **personRepository**.getPerNameWithNamedParamUsingClassProjection(999).stream().forEach(*e*-> System.***out***.println(*e*.getName())); |
| using native query | @Query(value = "SELECT COUNT(\*) FROM USERS WHERE ACTIVE = ?1", nativeQuery = true) Ⓒ  int findNumberOfUsersByActivityNative(boolean active); |
| using SPEL in query | @Query("select u.username, LENGTH(u.email) as email\_length from #{#entityName} u where u.username like %?1%")  List<Object[]> findByAsArrayAndSort(String text, Sort sort); |

The query language you’ve seen in this example isn’t SQL, it’s the Jakarta Persistence Query Language (JPQL). Although there is syntactically no difference in this trivial example, the Message in the query string doesn’t refer to the database table name but to the persistent class name. For this reason, the Message class name in the query is case-sensitive. If we map the class to a different table, the query will still work.

|  |  |
| --- | --- |
| Here select keyword is not mandatory –   * Read all All columns   @Query("from Person") – this person is entity name not table name *List*<Person> getAllPersons();  *List*<Person> getAllPersons(Sort s, Pageable p); | if u don’t mention select keyword then by default all columns will be fetched   * Here since there is no select keyword by default all columns will be fetched   internal query -- Select \* from student  even for these @Query methods we can pass the paging and sort related interfaces |
| partial columns  public class PersonNameOnly {  private String **name**;  public PersonNameOnly(String *name*) {  this.**name** = *name*;  }  public String getName() {  return **name**;  } } | Generally if u are fetching few columns then you will get as List<Object[]> where each ele in list is an object array that array consists of fetched columns, there is an alternate to fetch only few columns  @Query("select new com.nt.dto.PersonNameOnly( p.name) from Person p") *List*<PersonNameOnly> getNameSalaryOfAllPersons();  public class PersonNameOnly {  private String **name**;   public PersonNameOnly(String *name*) {  this.**name** = *name*;  }   public String getName() {  return **name**;  } } |
| passing parameters to the query | generally to pass inputs to query we should give ?   |  |  | | --- | --- | | select \* from emp where name=?0  this is called positional params  generally not recommended | select \* from emp where name=:firstName  this is named params and this is recommended |   earlier with finder methods, since query is generated by hib, we didn’t pass these arguments  but in real time, we mostly exec query with parameters  ex:- marks between  @Query( "select new com.nt.dto.PersonNameOnly(p.name) from #{#entityName} p where salary=:salary") *List*<PersonNameOnly> getPerNameWithNamedParamUsingClassProjection(@Param("salary") Integer *sal*);  **personRepository**.getPerNameWithNamedParamUsingClassProjection(999).stream().forEach(*e*-> System.***out***.println(*e*.getName())); |
|  |  |
|  |  |

Projections for @query

|  |  |
| --- | --- |
|  | projections means retriving only few columns instead of all columns |
|  | since we are writing query , dev has to tell which constructor to call |
| // Or DTO (Class-Based Projection)  public class PersonNameOnly {  private String **name**;  public PersonNameOnly(String *name*) {  this.**name** = *name*;  }  public String getName() {  return **name**;  } } | // Repository with DTO Projection (requires @Query)- because we have to tell which constructor to invoke  @Query("select new com.nt.dto.PersonNameOnly( p.name) from Person p") // This means we are fetching columns and passing those values into constructor  *List*<PersonNameOnly> getNameSalaryOfAllPersons();  *List*<PersonNameOnly> classBasedProjections = **personRepository**.getNameSalaryOfAllPersons();  The extra code with class based projection is – since we defined cons in class, from @Query method we have to invoke cons with fetched columns  preferred than interface based projections based here we can avoid extra proxy implementation class for that defined interface  internal query :- Hibernate: select p1\_0.name from person p1\_0  so hib is not cheating us- means it fetches only the columns what we ask |
| for @Query interface based proj wont work as that is not applicable | @Query("select p.name from Person p") *List*<Person.*IProductNameOnly*> getNameSalaryOfAllPersonsUsingInterface();   * In above class based projections we are manyally calling parameterized constructor , here as there is no constructor applicable at all   for interfaces – interface based projections are not applicable for custom queries   * Interface based projections is designed only for derived queries(queries for finder methods )   there fore @Query supports only class based projections, bec we are tell which cons to call  qq- in finder method, interface based proj are working, without cons then here also why spr is not able to add her intelligence? |
| Named params + class based proj + using operator | @Query( "select new com.nt.dto.PersonNameOnly(p.name) from #{#entityName} p where p.name like :regex") *List*<PersonNameOnly> getPerNameWithNamedParamsUsingClassProjUSingLikeOp(@Param("regex") String *pattern*  **personRepository**.getPerNameWithNamedParamsUsingClassProjUSingLikeOp("%n").forEach(*e*-> System.***out***.println(*e*.getName())); |

DML Queries Insert/Update/Delete / Modifying queries

For all DML operations @Modifying anno is mandatory bec @Query by default denotes a select query

& method must be annotated with @Transactional or be run from a programmatically managed transaction. (means whoever uses this method must be included in a transaction)

Modifying queries have the advantage of clearly emphasizing which column they address

|  |  |
| --- | --- |
|  | When u are performing any DML operation  @Modifying Is mandatory |
| delete query using derived methods | @Transactional  int deleteByLevel(int level);  Here as the query is generated by framework then @Modifying isn’t necessary  @Transactional int deleteBySalary(Integer *sal*); |
| delete using JPQL | @Transactional  @Modifying  @Query("delete from User u where u.level = ?1")  int deleteBulkByLevel(int level);  Here @Modifying is needed as the query is generated by us  What is the difference between the deleteByLevel and deleteBulkByLevel methods? The first one runs a query, and it will then remove the returned instances one by one. If there are callback methods that control the lifecycle of each instance (for example, a method to be run when a user is removed), they will be executed. The second method will remove the users in bulk, executing a single JPQL query. No User instance (not even the ones that are already loaded in memory) will execute lifecycle callback methods.  @Query("delete from #{#entityName} p where p.salary = :sal") @Transactional @Modifying int deleteUsingSalaryJPQL(@Param("sal") Integer *allsal*); |
|  |  |

Native queries

It supports all DDL, DQL, DML operations, it supports complex operations , this also supports named parameters

Use native queries if JPQL is becoming more complex

@Query(value= “ss”,nq=true)

Query By example

Study as per book

Inheritance Mapping

Generally, we used to map “1 table for every persistent entity class” (this the default )

|  |  |  |
| --- | --- | --- |
| Single table strategy | table per class (Table per concrete child class) | joined |
| table only for parent class | table only for childs (not for parent) | table for all (parent+child) |
| here all data will be stored in 1 table  disadvantage:- if we store like this always for every record some fields will be null.   |  |  | | --- | --- | | table only for parent | | | no table for childs | no table for childs | | advantage  since both are separate tables no need to filter  we can apply some triggers on each table- no need of joins   |  |  | | --- | --- | | no table for parent | | | table for child | table for child | | problem   1. we have to perform un-neccesary joins to see single rec 2. every time 2 inserts- 1 into parent and into child table |

Why abstract keyword for classes? – bec we don’t want to create object for that class in java

Single Table strategy(i.e.,parent table)

This means single table for entire hierarchy & that to table will be created only for parent class, not for child classes

And data will be distinguished based on discriminator column

This is worst strategy bec we must declare all columns as nullable columns

|  |  |  |
| --- | --- | --- |
|  |  |  |
| @Entity @Inheritance(strategy = *InheritanceType*.***SINGLE\_TABLE***) *//means table will be created only for parent class not for childs* @Data @DiscriminatorColumn(name = "pmode",discriminatorType = *DiscriminatorType*.***STRING***) public abstract class *Payment* {  *//we created this as abstract as we should not create obj for this* @Id  Integer **id**;  Integer **amount**; }  Here this pmode is not a entity property but those will be present in table | @Entity @DiscriminatorValue(value = "bakncheque") public class BankCheq extends *Payment* {  @Column(name = "chequeNumber")  Integer **bankCheqNumber**; }  if strategy is single table,  u cant even annotate child entities with  @Table(name = "card") because child table wont be created na  Without an explicit discriminator value, Hibernate defaults to the fully qualified class name | @Entity @DiscriminatorValue(value = "creditCard") public class CreditCard extends *Payment* {  @Column(name = "cardNumber")  String **cardNumber**; }  if u persist this entity then the above discriminator value will be persisted in that table |

Remember that NOT NULL constraints aren’t allowed in the databse schema,

|  |  |
| --- | --- |
| Main problem with this approach  BankCheq bankcheq=BankCheq.*builder*()  .bankCheqNumber(235)  .build(); bankcheq.setId(3); bankcheq.setAmount(45465); **paymentRepository**.save(bankcheq);  **paymentRepository**.save(creditCard);  if u persist bankcheq then automatically that disciminator value will be saved in that discriminator column  select \* from payment | ex:- if u insert bank cheq child entity credit card details will be null, if u insert credit card entity bankcheque details will be null  so with this approach always we have to encounter null – this is the major problem  and we need to declare all columns as nullable  Imagine that an expiration date for credit cards is required, but the database schema can’t enforce this rule because all columns of the table can be NULL. A simple application programming error can lead to invalid data  so the major problem of this strategy is nullable columns- we must declare all columns as nullable  bec at any time for each row some nulls will come for sure  Another important concern is normalization. We’ve created functional dependencies between non-key columns, violating the third normal form. As always, denormalization for performance reasons can be misleading because it sacrifices long-term stability, maintainability, and the integrity of data for immediate gains  acc to normalization all col must depend on pk, but here its not happenning, 2 diff data in same table  Hence this is called worst strategy as this avoids data integrity and normalization principles |
| @DiscriminatorColumn(name = "pmode",discriminatorType = *DiscriminatorType*.***STRING***)  This means a special column called “pmode” will be created in parent table | denormalized schemas can become a major burden in the long term, and your DBA may not like it at all. |

Why we shouldnot create repository class for each child? Why we should use parent repo only? What happens if we save with child repository?

If both data is stored in same table, for every record 1 field will be null, if u don’t want null/if u don’t want memory wastage, then don’t use this strategy

Qq:- as these childs does not have tables, whats use of marking them as entities

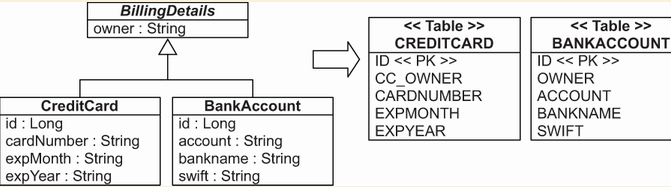
Table per class (Table per concrete sub class with implicit polymorphism, this is best as per me)

Here table will be created for every child concrete class, but no table for parent – all the parent properties will be inherited to each child

No discriminator column, because we already have separate table for each child

All concrete class mappings inherit persistent properties from the superclass (or interface).

From java side it appears as



|  |  |  |
| --- | --- | --- |
| parent | child classes |  |
| @Entity @Inheritance(strategy = *InheritanceType*.***TABLE\_PER\_CLASS***) *//means table will be created for every concrete child class* @Data public abstract class *Payment* {  *//we created this as abstract as we should not create obj for this* @Id  Integer **id**;  Integer **amount**; } | @Table(name = "bankcheck") @Entity public class BankCheq extends *Payment* {  @Column(name = "checknumber")  Integer **bankCheqNumber**; }  here with this approach data will be stored in separate child tables | @Entity @Table(name = "card") public class CreditCard extends *Payment*{  @Column(name = "cardnumber")  String **cardNumber**; } |
| saving data  **paymentRepository**.save(bankcheq);  **paymentRepository**.save(creditCard); |  |  |

Overriding the parent properties

|  |  |
| --- | --- |
| public abstract class BillingDetails {  @Id  @GeneratedValue(generator = “ID\_GENERATOR”)  private Long id;  @NotNull  private String owner;  // . . .  } | @Entity  @AttributeOverride(  name = "owner",  column = @Column(name = "CC\_OWNER", nullable = false))  public class CreditCard extends BillingDetails {  @NotNull  private String cardNumber;  @NotNull  private String expMonth;  @NotNull  private String expYear;  // . . .  } |
|  | using @AttributeOverride anno we can override parent property names |

Joined Strategy

Joined means all the tables parent , child tables are joined

This is most used strategy in real time

Hib people should have renamed this as per class because we are creating a class for every entity – for all parents and childs

Every class in the inheritance hierarachy will have a table (parent, child) , parent entity properties will be in parent table, child properties will be in child table

The main disadvantage is hib needs to join all tables to see full details

|  |  |
| --- | --- |
| Every class in inheritance hierarchy have a table, both the parent and child classes  everytime insert will happen into both parent and child table  fields declare in parent will be inserted into parent table and child fields will be in child table  what happens if we try to save using child repo instead of parent repo?  here all the tables must be in relation with pk fk   * when tables are in parent, child relation seems child doesn’t have to maintain pk col | even all the tables also will be in relation with pk and fk |

@PrimaryKeyJoinColumn is a when the primary key of one entity is used as a foreign key in another entity's table. (with this annotation we don’t need to declare the fk value in child entity) used in scenarios involving relationships between entities, specifically

In subclasses, we don’t need to specify the join column if the primary key column of the subclass table has (or is supposed to have) the same name as the primary key column of the superclass table. In the following listing, BankCheque will be a subclass of payment

|  |  |  |
| --- | --- | --- |
| payment table | bank cheque table    here eventhough we didn’t declare id In bankcheq table  that col got created as fk column only bec of @PrimaryKeyJoinColumn  **this id fk col is referring to parent table pk col** | card table    here id is not pk , it is just a fk column here referring to pk col in payment table  so this means , we are able to create a table without pk key |
| @Entity @Inheritance(strategy = *InheritanceType*.***JOINED***) *//means table will be created for all parent and childs* @Data public abstract class *Payment* {  *//we created this as abstract as we should not create obj for this* @Id  Integer **id**;  Integer **amount**; }  parent fields will be stored in parent table  child fields will be stored in child table | @Table(name = "bankcheque") @Entity @PrimaryKeyJoinColumn(name = "id") *//This anno is to make pk of table as foreign key of another table* public class BankCheq extends *Payment* {  @Column(name = "checknumber")  Integer **bankCheqNumber**; } | @Entity @Table(name = "card") @PrimaryKeyJoinColumn(name = "id") public class CreditCard extends *Payment*{  @Column(name = "cardnumber")  String **cardNumber**; } |
| main problem here is   1. hib has to insert in 2 rows – parent,child 2. while fetching we have to join | main advantage here is all table are normalized |  |

@Embedded Component mapping

|  |  |
| --- | --- |
|  |  |
| @Embeddable | @Embedded |

Association mappings

|  |  |
| --- | --- |
| 1-1 from both sides 1-1 | 1 person- will have 1 license  1 license belong only to 1 person |
| Many-Many = from both sides 1- many | 1 order will have multiple products  1 product belongs to multiple orders |
| one to many  (left side 1- many and right side one to one) | 1 person will have multiple phone numbers  but 1 phone number belongs to 1 person only |
| Many to one  (left side many-1 and right side ..) | we wont use this mostly in favour of one to many |

|  |  |
| --- | --- |
| uni directional means we can fetch only from 1 side | bi directional means, from both sides we can fetch other and navigate  incase of bi directional we have to keep annotations in both sides  else child cant fetch parent and parent cant fetch child |

In general, all the mapping anno like foreign key col details will be kept in 1 entity and in other entity we just tell mappedBy saying mapping info is already present in other entity, so don’t look for mapping info here – this is the purpose of mappedBy field

@OneToMany(mappedBy = "author")

One to Many & mappedBy

|  |  |
| --- | --- |
| // Author.java (Owning Side)  @Entity  public class Author {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  @OneToMany(mappedBy = "author") // Note: mappedBy is on the inverse side  private List<Book> books;  // ... other fields and getters/setters  } | to define foreign key col- use @JoinColumn(name = "author\_id")  // Book.java (Inverse Side)  @Entity  public class Book {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String title;  @ManyToOne // this says Many books can belong to same author  @JoinColumn(name = "author\_id") // Foreign key column in the BOOK table  private Author author; // This field is referenced by mappedBy in Author.java  // ... other fields and getters/setters  }  which ever the class having foreign key col that must have a repo interface ?? |
| 1. The table which has the primary key is called owning table   Here author table has the primary key so this is called parent table  mappedBy is an attribute which is used to tell hibernate to stop creating additional table for mappings, instead look for mapping information in the other java  entity  If u don’t specify that mappedBy then a separate table will be created with mappings  One Employee have many Laptops, since in Employee tables as we cant have a list type,  Internally hibernate will create a separate table,   |  |  | | --- | --- | | Student\_id | Laptop\_id | | S1 | L1 | | S1 | L2 |   The above states, each student will have many laptops (s1 having 2 laps called l1,l2)  @OneToMany(mappedBy = "author") // here author is a java field name in Book class  private List<Book> books;  class Book {  @ManyToOne // this says Many books can belong to same author  @JoinColumn(name = "author\_id") // Foreign key column in the BOOK table  private Author author  }  To stop this additional table creation, we have to tell hibernate saying, BRO don’t create another table for mappings, we are already having a mapping information in  child java class Book on a java field named author (in child entity we specified fk col)  @OneToMany(mappedBy = "author") //-- here author is the java field name where in child entity Author  Instead in laptop column itself we will create employee , use that , then we have to tell mapped by field  **Why is mappedBy Important?**   * **Database Consistency:** It clearly defines where the foreign key resides, preventing redundant foreign keys and ensuring data integrity. * **Bidirectional Relationships:** mappedBy is essential for bidirectional relationships (where both entities have a reference to each other). It prevents infinite loops during serialization and helps JPA understand the relationship's structure. * mappedBy is always used on top of child field /the inverse side of the @OneToMany or @ManyToMany relationship | 1. The table which has the foreign key is called child table   Here since book table has the foreign key col Book table is called child table   1. We must mention @JoinColumn(name = "author\_id")   this tells author\_id will be the foreign key column acts as a join col between parent table and child table  how to avoid null foreign key values    for 1st 2 records foreign key is null, bec we didn’t set parent to child before saving  to avoid null we have to set parent to child and child to parent  note:-  always u can keep all mapping information like fk col .. in 1 entity  in another entity just tell about where is this mapping info present using mappedBy attribute  mappedBy attribute is used to tell in which entity mapping info is present |

What is cascading in hibernate

Cascading is the process of propagating all the non select operations – update, delete, insert from parent to child

(PERSIST, MERGE, REMOVE, REFRESH, DETACH,ALL).

|  |  |
| --- | --- |
| The advantage with this relationships is  CascadeType.PERSIST   1. if u save parent, then childs also will be saved automatically(cascading effect)   ex:- if u don’t mention cascading, childs will not be saved automatically along with parent | **CascadeType.MERGE:**   * When a parent entity is merged (updated), the associated child entities are also merged. * This is used to synchronize changes from a detached entity back to the persistent context |
| if u delete parent, then childs also will be deleted automatically | first it will delete the child, then once all childs deleted, then parent will be deleted |

|  |  |
| --- | --- |
| Without cascade  public class Hero {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  String **name**;   @OneToMany(mappedBy = "hero")  *Set*<PhoneNumber> **phoneNumbers**; } | @Table(name = "phone\_number") @Entity public class PhoneNumber {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  Integer **number**;   @ManyToOne  @JoinColumn(name = "hero\_id")  *//this @JoinColumn anno says hero\_id will be the fk col which will be mapped to parent table* Hero **hero**;  *//No need to directly hardcode the fk col, we should tell via @JoinCol anno* }    here since there is not cascasde, when I insert into parent, childs are not inserted |
| With cascade  public class Hero {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  String **name**;   @OneToMany(mappedBy = "hero",cascade = *CascadeType*.***ALL***)  *Set*<PhoneNumber> **phoneNumbers**; | @Test  public void oneToManyDemo() {  PhoneNumber p1=PhoneNumber.*builder*().number(8500).build();  PhoneNumber p2=PhoneNumber.*builder*().number(8501).build();  Hero hero= Hero.*builder*()  .name("Ram Charan")  .phoneNumbers(*Set*.*of*(p1,p2))  .build();  **heroRepository**.save(hero); }  here since there is cascade we are seeing records in child table,  but still foreign key column is null, to avoid this fk nulls, we have to map the parent to every child as below  then only we can avoid null |
| mapping bi-directionally  @Table(name = "hero") @Entity public class Hero {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  String **name**;   @OneToMany(mappedBy = "hero",cascade = *CascadeType*.***ALL***)  *Set*<PhoneNumber> **phoneNumbers**;  }  public class PhoneNumber {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  Integer **number**;   @ManyToOne  @JoinColumn(name = "hero\_id")  *//this @JoinColumn anno says hero\_id will be the fk col which will be mapped to parent table* Hero **hero**;  }  if hero is inserted then phonenumbers set also needs to be inserted  then in that case,cascade must be kept on hero classes which is responsible to insert childs | To avoid null value for foreign key column in child table  we should set parent to child else childs foreign key value will be nul  @Test  public void oneToManyDemo() {  PhoneNumber p1=PhoneNumber.*builder*().number(100).build();  PhoneNumber p2=PhoneNumber.*builder*().number(101).build();  Hero hero= Hero.*builder*()  .name("Mega star Chiranjeevi")  .phoneNumbers(*Set*.*of*(p1,p2))  .build();  *//we have to set parent to every child, else fk ref col will be null* p1.setHero(hero);  p2.setHero(hero);  **heroRepository**.save(hero); }   |  |  | | --- | --- | |  | now fk col value also not null | |

Eager loading and lazyLoading

|  |  |
| --- | --- |
| While doing eager or lazy loading, ensure you have proper hashcode and equals method implementation or don’t implement those methods  in my case due to lombok @Data (which internally provides hashcode and equals implementation, ) I faced many assertion issues while fetching with eager loading  here parent means – the table which holds primary key  child means – the table which have foreign key | |
| @OneToMany(fetch = FetchType.EAGER) | Eager loading means as soon as parent is loaded childs will also be loaded  with eager loading by default both the parent and childs will be loaded at a time  single joins query will be fired to fetch all childs and parent  **@OneToMany(mappedBy = "hero",cascade = *CascadeType*.*ALL*,fetch = *FetchType*.*EAGER*) *Set*<PhoneNumber> phoneNumbers;**  **this is not recommended in realtime as it loads all child data un-neccesarily**  **@Test  public void eagerLoadingDemo(){  System.*out*.println("demo for eager loading");  Hero hero = heroRepository.findById(202).get();  System.*out*.println(hero.getName());  hero.getPhoneNumbers().stream().forEach(System.*out*::println); }** |
| @OneToMany(fetch = FetchType.LAZY)  means loading childs on demand basis  By default all are lazy loading only | **when u are using lazy loading, we must execute in transaction**  childs will be fetched on demand, so when u use getter method on any child, then only it will fetch those child records  Lazy fetching is a performance optimization technique where related entities are not loaded from the database until they are explicitly accessed. This avoids loading unnecessary data upfront, especially for large or complex entity relationships.  why u need to have a txn for lazy loading  If you try to access the lazily fetched data outside of an active transaction, JPA won't have a database connection available, and you'll likely encounter a LazyInitializationException.  **Always access lazily fetched data within a transaction.** This is the most important rule.  @Test @Transactional public void lazyLoadingDemo(){  System.*out*.println("demo for lazy loading");  Hero hero = heroRepository.findById(202).get();  System.*out*.println(hero.getName());  System.*out*.println("about to fetch childs");  hero.getPhoneNumbers().stream().forEach(System.*out*::println); }  log  demo for lazy loading  Hibernate: select h1\_0.id,h1\_0.name from hero h1\_0 where h1\_0.id=?  Mega star Chiranjeevi  about to fetch childs  Hibernate: select pn1\_0.hero\_id,pn1\_0.id,pn1\_0.number from phone\_number pn1\_0 where pn1\_0.hero\_id=?  PhoneNumber(id=53, number=100, hero=Hero{id=202, name='Mega star Chiranjeevi'})  **As per logs, 1st it fetches only parent by exec query from hero table, later when we executed getter methods then it fetched the child rows by firing another query ,**  **in eager loading a big long join query will be executed all data will be fetched in single shot**  **whereas in lazy loading, data will be loaded on demand**   |  |  | | --- | --- | | @Test @Transactional public void lazyLoadingDemo(){  Passport passport = **passportRepository**.findById(2).get();  System.***out***.println(passport);  System.***out***.println("fetching heroine data");  System.***out***.println(passport.getHeroine()); }  @OneToOne(cascade = *CascadeType*.***ALL***,fetch = *FetchType*.***LAZY***) @JoinColumn(name = "heroine\_id")  *//only foreign key can join 2 tables, so this name is fk col name* Heroine **heroine**; | Hibernate: select p1\_0.id,p1\_0.heroine\_id,p1\_0.type,p1\_0.valid\_from,p1\_0.valid\_to from passport p1\_0 where p1\_0.id=?  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  fetching heroine data  Hibernate: select h1\_0.id,h1\_0.age,h1\_0.first\_name,h1\_0.last\_name,p1\_0.id,p1\_0.type,p1\_0.valid\_from,p1\_0.valid\_to from heroine h1\_0 left join passport p1\_0 on h1\_0.id=p1\_0.heroine\_id where h1\_0.id=?  Heroine{id=1, firstName='Rakul', lastName='Preet', age=23} | | here also when we attempted to fetch heroine data, then it went and executed the join query | | |
| loading bi-directionally | when parent is fetched, we can fetch childs automatically  when child is fetched we can fetch a parent automatically |

Dirty checking in hibernate

This check is to perform whether object became dirty (changed) or not after it was loaded

In Hibernate, "dirty checking" is a core mechanism that automatically detects changes made to persistent entities during a transaction. Here's a breakdown of what it is and how it works:

Dirty checking means hib will compare the obj state with original state of an object for persistent entities before commiting ,

It will fully compare all the fields, if any field is changed, and update query will be generated only for those changed fields instead of generating query for all fields, this is a performance improvement

**What it is:**

* Dirty checking is Hibernate's way of determining whether an entity's state has been modified since it was loaded from the database.
* This process allows Hibernate to optimize database updates, ensuring that only the changed fields are updated.

**How it works:**

1. **Snapshotting:**
   * When Hibernate loads an entity from the database into the persistence context (the Session's cache), it takes a snapshot of the entity's original state.
2. **Tracking Changes:**
   * As your application modifies the entity's properties within the Session, Hibernate keeps track of these changes.
3. **Comparison:**
   * Before the transaction is committed (or when the Session is flushed), Hibernate compares the entity's current state with the original snapshot.

**Generating Updates:**

1. If Hibernate detects any differences (i.e., the entity is "dirty"), it generates the necessary SQL UPDATE statements to synchronize the changes with the database.
2. Crucially, Hibernate only updates the columns that have actually changed, which improves performance.

Hibernate internally uses equals() and hashcode() methods to perform dirty checking, so If u change those methods , it may not perform dirty checks properly

Ex:-If u load a list from db through jpa/Hibernate (by default obj will be in session memory), if u perform update only one 1 obj ,, but if u save 2 objects, then still only that changed object alone will be saved, means only 1 insert statement will be executed not 2,

internally since all the loaded objects will be saved in session memory before saving it will compare the state with original snapshot object (called dirty checking),

if the data is changed then only it will try to insert that obj else it will be ignored

|  |  |
| --- | --- |
| wrong simulation / not recommended  Hero hero = **heroRepository**.findById(252).get(); System.***out***.println("parent already loaded , now childs will load lazily on demand as we req"); *//Here in this list i am modifying only few ele, but i am saving full list //Eventhough i save full list, hib will perform dirty check and it will update only particular rec and particular col Set*<PhoneNumber> modifiedList = hero.getPhoneNumbers().stream().map(*e* -> {  if (*e*.getNumber() == 301) {  *e*.setNumber(3011);  }  return *e*; }).collect(Collectors.*toSet*()); System.***out***.println("new list -->"+modifiedList);  here this is not recommended because we are replacing the existing list with new list  internally hascode and equals just compares the ref and it will understand as the entire list is modified and it will generate a big update query for all records in the list |  |

|  |  |
| --- | --- |
| Generate hashcode and equals based on id, name, phonenumbers  so that During dirty checking if any change among these 3 will be considered  @Table(name = "hero") @Entity public class Hero {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  String **name**;   @OneToMany(mappedBy = "hero",cascade = *CascadeType*.***ALL***,fetch = *FetchType*.***EAGER***)  *Set*<PhoneNumber> **phoneNumbers**;  *//* @Override  public boolean equals(Object *o*) {  if (this == *o*) return true;  if (!(*o* instanceof Hero *hero*)) return false;  return Objects.*equals*(getId(), *hero*.getId()) && Objects.*equals*(getName(), *hero*.getName()) && Objects.*equals*(getPhoneNumbers(), *hero*.getPhoneNumbers());  }   @Override  public int hashCode() {  return Objects.*hash*(getId(), getName(), getPhoneNumbers());  } | @Table(name = "phone\_number") @Entity public class PhoneNumber {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  Integer **number**;   @ManyToOne  @JoinColumn(name = "hero\_id")  *//this @JoinColumn anno says hero\_id will be the fk col which will be mapped to parent table* Hero **hero**;  *//No need to directly hardcode the fk col, we should tell via @JoinCol anno* @Override  public boolean equals(Object *o*) {  if (this == *o*) return true;  if (!(*o* instanceof PhoneNumber *that*)) return false;  return Objects.*equals*(getId(), *that*.getId()) && Objects.*equals*(getNumber(), *that*.getNumber());  }   @Override  public int hashCode() {  return Objects.*hash*(getId(), getNumber());  } |
| @Test public void dirtyCheckingDemo(){  System.***out***.println();  Hero hero = **heroRepository**.findById(252).get();  System.***out***.println("parent already loaded , now childs will load lazily ");  *//Here in this list i am modifying only few ele, but i am saving full list  //Eventhough i save full list, hib will perform dirty check and it will update only particular rec and particular col  Set*<PhoneNumber> originalPhoneNumbers = hero.getPhoneNumbers();  *Set*<PhoneNumber> modifiedList = originalPhoneNumbers.stream().map(*e* -> {  if (*e*.getNumber() == 301) {  *e*.setNumber(3011); //with this original list also will be modified  }  return *e*;  }).collect(Collectors.*toSet*());  hero.setName(hero.getName().toUpperCase());  *//since cascading was enabled, as we save parent all the parent and childs will be saved at once* **heroRepository**.save(hero); }  now hibernate will internally perform dirty checking and it will generate only 1 insert query  as we modified only 1 element in the list | Hibernate: select h1\_0.id,h1\_0.name,pn1\_0.hero\_id,pn1\_0.id,pn1\_0.number from hero h1\_0 left join phone\_number pn1\_0 on h1\_0.id=pn1\_0.hero\_id where h1\_0.id=?  parent already loaded , now childs will load lazily on demand as we req  Hibernate: select h1\_0.id,h1\_0.name,pn1\_0.hero\_id,pn1\_0.id,pn1\_0.number from hero h1\_0 left join phone\_number pn1\_0 on h1\_0.id=pn1\_0.hero\_id where h1\_0.id=?  Hibernate: update hero set name=? where id=?  Hibernate: update phone\_number set hero\_id=?,number=? where id=? |

ManyToMany

Many to many means from both sides one to many here 3 tables will be created – 3rd will be mappings created just to store the mapping info

Internally, it will insert data into 3 tables

Hibernate: insert into company (name,id) values (?,?)

Hibernate: insert into vendor (name,id) values (?,?)

Hibernate: insert into vendor (name,id) values (?,?)

Hibernate: insert into company\_vendors (company\_id,vendor\_id) values (?,?)

Hibernate: insert into company\_vendors (company\_id,vendor\_id) values (?,?)

|  |  |  |
| --- | --- | --- |
|  |  |  |
| @Table(name = "company") @Entity @Getter @Setter public class ProductBasedCompany {   @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;   String **name**;  @Column(name = "emp\_count")  Integer **employeeCount**;  *//here @JoinColumn( name = "company\_id")=join col name & it is foreign key col name* @ManyToMany(cascade = *CascadeType*.***ALL***,  fetch = *FetchType*.***EAGER***)  @JoinTable(name = "company\_vendors",  joinColumns =@JoinColumn( name = "company\_id",referencedColumnName = "id"),  inverseJoinColumns =@JoinColumn(name = "vendor\_id",referencedColumnName = "id")  )  *Set*<Vendor> **vendors**; | @Table(name = "vendor") @Entity public class Vendor {   @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;   String **name**;   *//This vendors is a java field name in prodBasedCompClass  //This says dont create additional table and mapping info already present in another entity on top of vendors field* @ManyToMany(mappedBy = "vendors")  *Set*<ProductBasedCompany> **productBasedCompanies**;  } | @Test *// @Transactional* public void ManyToManyDemo() {  *Set*<Vendor> vendors=*Set*.*of*(  Vendor.*builder*().name("synechron").build(),  Vendor.*builder*().name("wipro").build()  );  ProductBasedCompany company = ProductBasedCompany.*builder*().name("JpMorgan chasea").employeeCount(3500).  vendors(vendors).build();  System.***out***.println("Attempting to save ");  try{  **productBasedCompanyRepo**.save(company);  }catch (InvalidDataAccessApiUsageException *e*){  System.***err***.println(*e*);  throw *e*;  }  System.***out***.println("saved successfully");  } |
| when parent is inserted , to insert childs automatically we need to enable cascading effect |  | The main problem I noticed here is enabling transaction is not inserting any record in database- don’t know why |
| Fetching many to many  @Test public void fetchAllDemo(){  Optional<ProductBasedCompany> citiBank = **productBasedCompanyRepo**.findById(702);  ProductBasedCompany comp = citiBank.get();  System.***out***.println(comp);  System.***out***.println(comp.getVendors()); }  while eager loading a single big joins query will be framed to fetch data from all tables  Hibernate: select pbc1\_0.id,pbc1\_0.emp\_count,pbc1\_0.name,v1\_0.company\_id,v1\_1.id,v1\_1.name from company pbc1\_0 left join company\_vendors v1\_0 on pbc1\_0.id=v1\_0.company\_id left join vendor v1\_1 on v1\_1.id=v1\_0.vendor\_id where pbc1\_0.id=? | here if we fetch parent, childs will be fetched automatically  ProductBasedCompany{id=702, name='JpMorgan chasea', employeeCount=3500}  [Vendor{id=303, name='synechron'}, Vendor{id=302, name='wipro'}] |  |

One to One

There are 2 approaches

* 1. Have same primary col and values in both tables – problem is every license will belongs to a person, but some persons may not have a license

Hence both tables cant have same primary key

* 1. Maintain a foreign key col in child license table and map that foreign key col to the pk col of person table

So that we can map every license to a person, but error is we can map multiple licenses also to a same person to avoid that we can apply unique constraint

But in real time 1-many is only possible – because when licenses expires then we should have mulitple licenses belong to same person , so foreign key is best approach without unique constraint on foreign key column

|  |  |  |
| --- | --- | --- |
| person | license |  |
| @Table(name = "heroine") @Entity public class Heroine {   @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  @Column(name = "first\_name")  String **firstName**;  @Column(name = "last\_name")  String **lastName**;  Integer **age**;  @OneToOne(mappedBy = "heroine")  Passport **passport**; | @Table(name = "passport") @Entity public class Passport {  @Id  @GeneratedValue(strategy = *GenerationType*.***AUTO***)  Integer **id**;  String **type**;   LocalDate **validTo**;  LocalDate **validFrom**;   *//if we save pasport heroin must also be saved* @OneToOne(cascade = *CascadeType*.***ALL***)  @JoinColumn(name = "heroine\_id")  *//only foreign key can join 2 tables, so this name is fk col name* Heroine **heroine**; | @Test public void oneToOneCascadeDemo(){  Heroine heroine= Heroine.*builder*().age(23).firstName("Rakul").lastName("Preet").build();  Passport passport= Passport.*builder*().heroine(heroine)  .type("Ordinary")  .validFrom(LocalDate.*now*())  .validTo(LocalDate.*now*().plusYears(10))  .build();  **passportRepository**.save(passport);  System.***out***.println("records saved successfully"); |

Caching

caching is a mechanism to store frequently accessed data in memory, reducing the number of database round trips and improving application performance. Hibernate provides two main levels of caching:

by default level 1 /session level caching will be enabled, means once txn is opened session cache will be opened automatically

by default all those db interaction outputs will be cached, and once txn completes ses level cache will be deleted automatically

|  |  |
| --- | --- |
|  | when client requests for data, 1st hib will hit the db and get the data and stores in cache before rendering back to user   * 1. When data is updated in db, data will be updated in cache also   2. When data is deleted in db, data will be deleted in cache also |
| here it says every session will have its own cache  every time before it hits db, 1st it will searches in its own associated session cache | for multiple session same cache ,we have providers like – Ehcache, swaram cache, jboss tree cache, os cache |
| First-Level Cache (Session Cache): this is free   * Every hib session will have its own session cache * This is a built-in, mandatory cache associated with a Hibernate Session. * It stores objects within a single session. * When an entity is retrieved, it's stored in the first-level cache. If the same entity is requested again within that session, it's retrieved from the cache. * This cache is not shared between sessions. * When the session closes, the cache is cleared means when transaction is completed, session will be closed, then session level cache will also be cleared   When you use the @Transactional annotation on a service method, Spring will typically:   * Open a Hibernate Session at the beginning of the transaction. * Perform the database operations within the transaction. * Commit the transaction (and close the Session) if successful, or roll back the transaction (and close the Session) if an error occurs.   Therefore, within a transactional method, the session is opened at the start of the method, and closed at the end of the method | Second-Level Cache (SessionFactory Cache):   * This is an optional, pluggable cache shared across multiple sessions within a SessionFactory. * It stores data that can be reused across different sessions, significantly improving performance. * Hibernate supports various second-level cache providers, such as Ehcache and Infinispan. * To use the second-level cache, you need to configure a cache provider and specify which entities should be cached. * this cache stores data that is relevant to the whole application, not just a single session. * Query results can also be stored in the second level cache |

Caching for finder methods

Finder methods defined in repo interfaces will be executed always inside a transaction , even if u wont declare @Transactional jpa will internally open a transaction

 Spring Data JPA repositories are designed to work within a transactional context.

 Even for simple read operations (like those performed by finder methods), a transaction is often started and committed to ensure data consistency.

 Therefore, in practice, a session is very likely to be opened when a finder method is executed.

|  |  |
| --- | --- |
| **Within a Transaction:**   * If a finder method is called within an existing transactional context (e.g., a method annotated with @Transactional), then a Hibernate will use the existing session Session (or, more accurately, an EntityManager) will be associated with that transaction. * This means that if a transaction is already active, the finder method will utilize the currently open session * If there is no existing transaction hib will open a new transaction   means  inside a transactional method if 2 times same finder method is called, then  2nd time it will fetch from session level cache / L1 cache without hitting database  so we can feel the benefits of caching only inside a session, once session is closed eventhough u fire same finder method it will again hit db | **Outside a Transaction:**   * If a finder method is called outside of an active transactional context, Spring Data JPA will typically initiate a transaction (and thus, open a session) for the duration of that method's execution. * So, even if you don't explicitly have a @Transactional annotation, Spring Data JPA often handles the transactional boundaries for repository methods. |
| @Test @Transactional public void sameFinderMethodWillFetchDataFromCache(){  System.***out***.println("hitting and fetching 1st time");  Passport passport = **passportRepository**.findById(2).get();  System.***out***.println(passport);  System.***out***.println("fetching 2nd time from cache- no query will be fired. as inside same txn");  System.***out***.println(**passportRepository**.findById(2).get());   System.***out***.println("fetching 3rd time from cache as inside same txn");  System.***out***.println(**passportRepository**.findById(2).get()); }    since we already annotated with @Transactional all the 3 finder methods will use same txn  Single txn = in single session & 1 ses cache will be created for entire txn  1st time finder method will hit and gets data and stores that data in ses  2nd time same finder methods, as data is already present in same sessionm it wont hit db it simply fetches from ses level cache  Output  hitting and fetching 1st time  Hibernate: select p1\_0.id,h1\_0.id,h1\_0.age,h1\_0.first\_name,h1\_0.last\_name,p1\_0.type,p1\_0.valid\_from,p1\_0.valid\_to from passport p1\_0 left join heroine h1\_0 on h1\_0.id=p1\_0.heroine\_id where p1\_0.id=?  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  fetching 2nd time from cache- no query will be fired. as inside same txn  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  fetching 3rd time from cache as inside same txn  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  note:- after every syout even though we fired finder method, since we are fetching from cache select query is not printed means there is no db exec , | @Test public void seperateTxnSepSessionSepCacheHitEveryTime(){  System.***out***.println("hitting and fetching 1st time");  Passport passport = **passportRepository**.findById(2).get();  System.***out***.println(passport);  System.***out***.println("sep txn-sep cache- as no data in cache- hit db again ");  System.***out***.println(**passportRepository**.findById(2).get());   System.***out***.println("sep txn-sep cache- as no data in cache- hit db again ");  System.***out***.println(**passportRepository**.findById(2).get()); }    since method is not annotated with @Transactional there is no txn by default ,  so every finder method will execute in separate txn  separate txn means separate session for each txn,  so here 1st finder method will be executed in t1 and that data will be stored in s1 session level cache, once 1st finder method exec that session cache will be deleted  again 2nd finder method will be executed in separate txn t2 and new session s2  and new session level cache will be created, in that newly created cache prev session data will not be there na, so again data will be fetched from database  output  hitting and fetching 1st time  Hibernate: select p1\_0.id,h1\_0.id,h1\_0.age,h1\_0.first.. where p1\_0.id=?  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  sep txn-sep cach- no data in cache- hit db again  Hibernate: select p1\_0.id,h1\_0.id,h1\_0.age,h1\_0.first.. where p1\_0.id=?  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  sep txn-sep cach- no data in cache- hit db again  Hibernate: select p1\_0.id,h1\_0.id,h1\_0.age,h1\_0.first.. where p1\_0.id=?  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  as no txn – sep txn for each finder method = sep cache= as no data in cache= every finder() will hit db and gets the data |

Evicting data from session cache

This evict can remove data only from session cache L1 , not from session factory cache L2

…..

|  |  |
| --- | --- |
| ***Evicting Single object from Session cache*** | ***Flushing all data present in ses cache*** |
| *Session* session = **entityManager**.unwrap(*Session*.class); session.evict(passport);  you can "unwrap" the EntityManager to get access to the Hibernate Session.  this line of code retrieves the Hibernate Session object from the JPA EntityManager.  This is done when you need to use Hibernate-specific features that are not available through the standard JPA API.  In simpler terms, it's like having a universal remote (EntityManager) that can control many devices, and then using a special button (unwrap()) to get the original remote (Session) for a specific device (Hibernate) to access its advanced features. | **Clearing the EntityManager:**   * Clearing the EntityManager detaches all managed entities from the persistence context. This effectively removes them from the session cache.   Flushing and clearing the entity manager should be done with care. Unnecessary flushing and clearing can add overhead to your application |
| @Transactional *//since this is transactional, for entire method single txn, so single ses cache will be used* public void evictFromSessionCache(){  System.***out***.println("hitting db 1st time");  Passport passport = **passportRepository**.findById(2).get();   System.***out***.println("exec finder method 2nd time it should get from cache");  System.***out***.println(**passportRepository**.findById(2).get());   System.***out***.println("deleting from cache");  *Session* session = **entityManager**.unwrap(*Session*.class);  session.evict(passport);   System.***out***.println("exec finder method 3rd time, since data is missing from cache it will hit db");  **passportRepository**.findById(2).get(); }  output  hitting db 1st time  Hibernate: select p1\_0.id,h1\_0.id,h1\_0.age,h1\_0. heroine\_id where p1\_0.id=?  exec finder method 2nd time it should get from cache  deleting from cache  exec finder method 3rd time, since data is missing from cache it will hit db  Hibernate: select where p1\_0.id=? | @Test @Transactional *//since this is transactional, for entire method single txn, so single ses cache will be used* public void deleteAllFromCache(){  System.***out***.println("demo for flush the full ses cache, hitting db 1st time");  Passport passport = **passportRepository**.findById(2).get();   System.***out***.println("exec finder method 2nd time it should get from cache");  System.***out***.println(**passportRepository**.findById(2).get());   System.***out***.println("deleting everything from cache");  *Session* session = **entityManager**.unwrap(*Session*.class);  session.evict(passport);  **entityManager**.flush();   System.***out***.println("exec finder method 3rd time, since data is missing from cache it will hit db");  **passportRepository**.findById(2).get(); }  **outupt**  demo for flush the full ses cache, hitting db 1st time  Hibernate: select ..heroine\_id where p1\_0.id=?  exec finder method 2nd time it should get from cache  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  deleting everything from cache  exec finder method 3rd time, since data is missing from cache it will hit db  Hibernate: select ..heroine\_id where p1\_0.id |

Session Factory level cache (L2)

Regardless of whether a second-level cache is enabled, the first-level cache will always be enabled.

**Flow**

so when a finder methods is executing, 1st hib will check in l1 cache,

If object is not found then it will search In L2 cache

If missing in L2 cache also then it will hit Db

The retrieved entity is then stored in both the first-level cache (for the current session) and the second-level cache (for subsequent sessions if entity is marked as Ses fac cacheable with @org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE))

If data is present in second-level cache, it data will be available for all sessions, & when a new session is opened , then that data will be placed in 1st level cache also

This is to bypass the database call and fetch from memory

How to check if data is there in session factory or not?

System.***out***.println("is data in ses fac cache before db hit "+**sessionFactory**.getCache().containsEntity(Passport.class,2));

EH Cache

EH cache supports both in memory and Disk based storage

Very fast and light weight it supports tera bytes of data and supoorts clustering

We can set timeout for each object (time to live in memory before deleting)

Why to use this?? Generally since session level cache will be deleted after transaction completes , Some objects may be needed across multiple sessions,

In that cases keep those long lived objects in the L2 cache ,

Ex:- gold rate, this is same for all objects that to for an entire day, so this must be second level cache,

By mistake if u keep this is l1 cache, if 1,000 people hit db , 1000 sessions will be created and 1,000 times it will be fetched as this is in session cache

So global data which could be same for all sessions those objects must be kept in session factory cache

Updates

hibernate-ehcache is based on Ehcache 2, which is no longer supported - Ehcache 3 is the one supported by the Ehcache team

Ehcache 3 can be easily used instead by using the hibernate-jcache module and have Ehcache 3 (which is a JCache implementor) properly registered with JCache.

the RegionFactory implementation from Ehcache, if/when they provide that

**EHCache memory details**

Ehcache is designed to utilize RAM, among other storage options, for caching data

**Ehcache Storage Tiers:**

* Ehcache allows you to configure different storage tiers, and RAM plays a crucial role.
* **On-Heap Store(not recommended in favor of garbage collection):**
  + This tier directly uses the Java heap memory. This is the fastest storage option, as it provides direct access to the data.
* **Off-Heap Store:**
  + Ehcache can also use off-heap memory, which is still within RAM but outside the Java heap. This helps to reduce garbage collection overhead.
* Therefore, Ehcache very much uses RAM for hot data storage.
* RAM provides very fast read and write access, which is essential for caching.
* The goal of caching is to store frequently accessed data in a location that allows for quick retrieval
*  Strong integration with Java frameworks like Spring and Hibernate.
*  Well-suited for Java-based applications

cache concurrency strategy

|  |  |
| --- | --- |
| Hibernate offers several concurrency strategies, each designed for different data access patterns:   * **READ\_ONLY:** (use this only if ur app is performing only read operations\ For immutable entities. )   + This strategy is used for immutable data that never changes.   + It's the simplest and most performant strategy.   + If an application attempts to update a READ\_ONLY cached entity, an exception will be thrown.   + Ideal for reference data. * **NONSTRICT\_READ\_WRITE(**For entities that are occasionally updated**):**   + This strategy is suitable for data that is read frequently but updated infrequently.   + It doesn't provide strict consistency guarantees, meaning stale data might be read in rare cases.   + It's less strict than READ\_WRITE and can offer better performance in certain scenarios.   means- when u are reading rec-10 from cache, if same data is being updated parallely by some one using transaction, u will not get the latest data, u will get only latest data whatever is in cache, ideally we should not read old data, we should have waited untill txn completes, and let the db data gets synced in cache and then we should read the data from cache till that time we should wait for caching to be updated   * **READ\_WRITE:**   + This strategy provides stricter consistency than NONSTRICT\_READ\_WRITE.   + It uses "soft locks" to prevent concurrent updates from corrupting the cache.   + It's suitable for read-mostly data where occasional updates occur.   + It ensures that concurrent transactions will not read stale data.   **TRANSACTIONAL:** (use this incase of distributed txn or XA transactions)   * + This strategy is designed for transactional cache providers that support full ACID transactions.   + It provides the strongest consistency guarantees.   + It's typically used in conjunction with distributed transactional caches. | When working with Hibernate's second-level cache, choosing the right concurrency strategy is crucial for maintaining data integrity and optimizing performance. Here's a breakdown of the key concepts:  **Understanding Hibernate's Second-Level Cache**   * Hibernate's second-level cache is a shared cache that exists between SessionFactory instances. This cache stores entity data and query results, reducing the need for frequent database access. * Concurrency strategies are essential because they dictate how Hibernate handles concurrent access to cached data, especially when multiple transactions attempt to modify the same entities   **Key Considerations**   * **Data Access Patterns:** The choice of concurrency strategy depends heavily on how your application accesses and modifies data. * **Consistency Requirements:** If data consistency is critical, choose a strategy like READ\_WRITE or TRANSACTIONAL. If some tolerance for stale data is acceptable, NONSTRICT\_READ\_WRITE might be suitable. * **Performance:** READ\_ONLY offers the best performance, while TRANSACTIONAL provides the strongest consistency but can have a performance overhead. * **Cache Provider:** The capabilities of your chosen second-level cache provider (e.g., Ehcache, Hazelcast) can influence the available concurrency strategies. |
| READ\_WRITE    if person A tries to read rec-10 from cache, if another person tries to update the same record in db using a transaction, read in cache will be on hold by implementing soft lock on that cache record, untill txn completes on that record – mean while if another person comes and reads same record from cache, it can notices the lock on that record, instead of waiting it can directly go and read that rec from db after txn complete, but its waste to goto db bec untill txn complete we also cant go and read from db , instead we can wait at cache, once txn completes data from db will be synced here and we can read here itself | READ\_WRITE cont ….  A second transaction may not read the item from the cache while the soft lock is present, and instead simply proceeds to read the item directly from the database, exactly as if a regular cache miss had occurred.  Similarly, the soft lock also prevents this second transaction from storing a stale item to the cache when it returns from its round trip to the database with something that might not quite be the latest version. |

If we annotate only a class, all the data of that class will be stored in L2 cache,

Any possibility to store custom objects in L2 cache

Code

|  |  |
| --- | --- |
| 1. Add properties in application.properties file 2. Add ehcache.xml 3. Make entities cacheable and 4. Make entity serializable to store data In disk | properties  spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true  spring.jpa.properties.hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory  //with above we are telling hibernate to use Ehcache as second level cache /session factory cache  spring.cache.ehcache.config=classpath:ehcache.xml  spring.jpa.properties.javax.persistence.sharedCache.mode=ALL  spring.jpa.properties.javax.persistence.sharedCache.mode=ALL tells your JPA provider to enable caching for all entities by default. This can be beneficial for read-heavy applications, but it's essential to consider memory usage and cache invalidation |
| add jars | implementation 'org.hibernate:hibernate-ehcache' implementation 'org.ehcache:ehcache'  These 2 jars will pull all below jars  +--- org.hibernate:hibernate-ehcache -> 5.6.15.Final  | +--- org.jboss.logging:jboss-logging:3.4.3.Final  | +--- org.hibernate:hibernate-core:5.6.15.Final (\*)  | \--- net.sf.ehcache:ehcache:2.10.6 -> 2.10.9.2  | \--- org.slf4j:slf4j-api:1.7.25 -> 1.7.36  \--- org.ehcache:ehcache -> 3.10.8  \--- javax.cache:cache-api:1.1.0 -> 1.1.1 |
| content for ehcache.xml | <ehcache>  <diskStore *path*="java.io.tmpdir"/>    <defaultCache *maxElementsInMemory*="100" *eternal*="false" *timeToIdleSeconds*="5" *timeToLiveSeconds*="10" *overflowToDisk*="true"/>  </ehcache> |
| annotate to cache specific entity | @Cache(usage = CacheConcurrencyStrategy.READ\_WRITE) |

|  |  |
| --- | --- |
| @Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)  if u annotate this on entity, all objects of that entity will be cached and kept into session factory cache L2 | if u uncomment this @cache anno on entity  then entities of this type will not be cached  spring.jpa.properties.javax.persistence.sharedCache.mode=ALL  if All then all entities will be cached |
| import org.hibernate.annotations.Cache;  import org.hibernate.annotations.CacheConcurrencyStrategy;  @Entity  @Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)  public class MyEntity {  @Id  private Long id;  // ... other fields  }  problem  simply annotating an entity with @Cache will make *all* instances of that entity type eligible for caching. If you need fine-grained control and want to cache only specific objects of an entity,  solution  import org.hibernate.query.Query;  //...  Query<MyEntity> query = session.createQuery("from MyEntity where someCondition = :value", MyEntity.class);  query.setParameter("value", someValue);  query.setCacheable(true);  List<MyEntity> results = query.getResultList(); | .  yet to practice |
| @Test @Transactional public void deleteFromSessFactCache() {  *Session* sess = **entityManager**.unwrap(*Session*.class);  System.***out***.println("is data in ses fac cache before db hit " + **sessionFactory**.getCache().containsEntity(Passport.class, 2));   System.***out***.println("hitting db 1st time to store in ses,sf cache");  Passport passport = **passportRepository**.findById(2).get();  System.***out***.println("is data in ses fac cache after db query " + **sessionFactory**.getCache().containsEntity(Passport.class, 2));  System.***out***.println("firing finder() 2nd time to fetch from ses cache \n" + passport);  **passportRepository**.findById(2);   System.***out***.println("clearing from ses cache ");  sess.evict(passport);  System.***out***.println("is data in ses fac cache after evicting from ses cache-- " + **sessionFactory**.getCache().containsEntity(Passport.class, 2));   System.***out***.println("cleared from ses cache, but still it should fetch from ses fac cache");  Passport passport1 = **passportRepository**.findById(2).get();  System.***out***.println(passport1);   System.***out***.println("clearing from ses cache ");  sess.evict(passport);  System.***out***.println("is data in ses fac cache after evicting from ses cache-- " + **sessionFactory**.getCache().containsEntity(Passport.class, 2));   System.***out***.println("cleared from ses cache, but still it should fetch from ses fac cache");  Passport passport2 = **passportRepository**.findById(2).get();  System.***out***.println(passport2);  } | Note: -  in some cases - eventhough obj is present in ses fac cache and we evicted from ses cache, query is again fired to fetch from db(but it shouldnot have hit the db as data is present in ses fac cache )  as per gemini, it is saying hib will interact with db to check if the data present in ses fac cache is stale or not  Console output  is data in ses fac cache before db hit false  hitting db 1st time to store in ses,sf cache  Hibernate: select .. where passport0\_.id=?  is data in ses fac cache after db query true  firing finder() 2nd time to fetch from ses cache  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  clearing from ses cache  is data in ses fac cache after evicting from ses cache-- true  cleared from ses cache, but still it should fetch from ses fac cache  Hibernate: select .. where passport0\_.id=?  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  clearing from ses cache  is data in ses fac cache after evicting from ses cache-- true  cleared from ses cache, but still it should fetch from ses fac cache  Passport{id=2, type='Ordinary', validTo=2035-03-03, validFrom=2025-03-03}  analysis  1st time when we evicted from ses cache, eventhough data is present in ses fac cache, db query is still executed to check whether data in ses fac cache is stale or not  2nd time when we evicted from ses cache, db query is not executed, it fetched from ses fac cache as it is missing from ses cache, this time stale data comparision not happened  Essentially, Hibernate optimizes the second retrieval because it considers the data to be already validated. |

For very specific and complex caching requirements, you might consider implementing your own application-level caching using a separate cache provider (like Redis or Caffeine).

Transactions

Every txn will have 4 key properties

Atomic - This is an atomic unit of work, means do all or do nothing,( This property ensures that a transaction is treated as a single, indivisible unit of work.)

 If any part of the transaction fails, the entire transaction is rolled back, and the database is returned to its previous state.

 This "all or nothing" principle prevents partial updates that could lead to data corruption.

Ex:- if we apply transaction on a method, then entire method is treated as single unit of work- means either all those statements will be executed or nothing will be executed

Consistency –

Isolation – even if many tranasctions are running in parallel, all txns must run in isolation,

It ensures that transactions are executed independently, as if they were running in isolation.

This prevents interference between transactions, ensuring that the results of one transaction are not affected by another.

Durability –means once a transaction is committed changes are permanent and will survive even system failures, such as power outages or crashes.



 Committed transactions are recorded in persistent storage, ensuring that the data is not lost.

 This property is typically achieved through techniques like transaction logs and database backups.

Components 🡪 Transacation manager (JPA) will interact with 🡪 Resource Manager 🡪 DB

|  |  |
| --- | --- |
| @Test @Transactional public void txnDemo(){   int transferAmount=500;  BankAccount megStarAccount = **bankAccountRepo**.findById(1).get();  megStarAccount.setBalance(megStarAccount.getBalance()-transferAmount);  **bankAccountRepo**.save(megStarAccount);  System.***out***.println(10/0);  BankAccount ramCharanAccount = **bankAccountRepo**.findById(2).get();  ramCharanAccount.setBalance(megStarAccount.getBalance()+transferAmount);  **bankAccountRepo**.save(ramCharanAccount); } | transaction means – all or nothing - all lines are considered as 1 unit of work  we wantedly throwing exception after 1 transaction,  since we annotated with @Transactional, even though 1st got saved, as second save is failed, prev success txn also will be rollbacked ,  assumption:- during a txn all the data will be written to database with some flags which other cant read, but those will be written to database , but those records can be readable only when the data is committed |

Every interaction with the database should occur within transaction boundaries, even if we’re only reading data, so we start a new transaction. Any potential failure appearing from now on will not affect the previously committed transaction.

That’s why even finder methods always will run in a transaction,

**Read-Only Transactions:**

Even for read-only operations, using a transaction (and marking it as read-only) can provide performance benefits. For example, Hibernate and the database can optimize certain operations when they know that the transaction won't be modifying any data.

**Preventing "Dirty Reads" and "Phantom Reads":**

Transactions, with appropriate isolation levels, prevent issues like "dirty reads" (reading uncommitted data) and "phantom reads" (seeing new rows inserted by another transaction).

Problem of Repetable Reads? - within a transaction, if you perform the same fetch operation multiple times, you'll get the same results bec data was cached

To avoid the problem of repeatable reads /reading data from cache – execute the fetch operations also in separate transaction

it's generally best practice to use transactions for data integrity and consistency.

FAQ

* 1. How to connect to 2 databases – in hib create 2 hib.cg.xml , create 2 sess factory beans , and create 2 session object 1 from each session
  2. Switching between jpa and hibernate:-

To obtain a SessionFactory from an EntityManagerFactory, you’ll have to unwrap the first one from the second one.

|  |  |
| --- | --- |
| private static SessionFactory getSessionFactory  (EntityManagerFactory entityManagerFactory) {  return entityManagerFactory.unwrap(SessionFactory.class);  }  *Session* session = **entityManager**.unwrap(*Session*.class); session.evict(passport); | in some cases some functionalities will only be available with hibernate, in that cases we should use hibernate instead of JPA,  ex:- if we want to evict an object from session cache, this evict is available in hibernate alone ,  so in this case we must use hib by using unwrap |

* 1. Why session .save() / crudRepository.save() will not immediately run the query () why it waits till transaction completion?

session .save(obj) / crudRepository.save(obj) will push that object from transient state to persistent state and that obj will be inserted only when we comitted

* after ses.persist()/reposave() also developer can modify that object, since hib has to track that obj untill he commits the transaction – it will monitor that obj, if changes happens again it needs to run update query – so that case 2 queries will be fired, to avoid it- it will run single query at last
  1. sdf