Master Hibernate and JPA with Spring Boot in 100 Steps

Contents

JPA Introduction

- To turn on JPA statistics
  spring.jpa.properties.hibernate.generate_statistics=true
- To see parameter values passed to sql queries
  logging.level.org.hibernate.type=trace
- JUnit 4 vs 5

  Spring Boot projects with versions >= 2.2.0 use JUnit 5 by default.

Description	JUnit 4	JUnit 5
Test Annotation Changes	`@Before` `@After` `@BeforeClass` `@AfterClass` `@Ignore`	`@BeforeEach` `@AfterEach` `@BeforeAll` `@AfterAll` `@Disabled`
Use `@ExtendWith` instead of `@RunWith`	`@RunWith(SpringJUnit4ClassRunner.class)` `@RunWith(MockitoJUnitRunner.class)`	`@ExtendWith(SpringExtension.class)` `@ExtendWith(MockitoExtension.class)`
Package changes to `org.junit.jupiter`	`org.junit.Test;` `org.junit.Assert.*;`	`org.junit.jupiter.api.Test;` `org.junit.jupiter.api.Assertions.*;`
`@RunWith` is NOT needed with `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`	`@RunWith(SpringRunner.class)` `@SpringBootTest(classes = DemoApplication.class)`	`@SpringBootTest(classes = DemoApplication.class)`

- If you are trying to make a changes in data, you should do it within a transaction. Hence it is a must to use **`@Transactional`** annotation for such operations (method level or class level).
- When we write unit tests, you would want to leave the state of the application as it was before running that test. Use **`@DirtiesContext`** annotation, what it does is after that test is run, spring would automatically reset the data.

Advanced JPA

- Refer – https://github.com/sameerbhilare/jpa-with-hibernate/tree/master/02-jpa-advanced

Entity Manager

- Entity Manager is an interface to Persistent Context.
- It is an interface used to interact with the persistence context.
- Persistent Context –
    - All the entities that are being saved through the entity manager are saved to persistence context.
    - The persistence context keeps track of all the different entities which are changed during a given transaction.
    - And also it keeps track of all the changes that needs to be stored back to the database.
    - A persistence context is a set of entity instances in which, for any persistent entity identity, there is a unique entity instance.
    - Within the persistence context, the entity instances and their lifecycle are managed.
- The `EntityManager API` is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.
- While we are within the scope of a transaction (@Transactional), entity manager keeps track of all the things that were inserted or updated through it.
- So whatever things were modified/inserted through the entity manager, the entity manager would start keeping track of them.
- If you are inside a transaction and you are managing (inserting/updating/deleting) something with the entity manager, then that particular thing continues to be managed by the entity manager until the end of the transaction.
- Example

```
@Transactional
public void playWithEntityManager() {
          Course course1 = new Course("Web Services in 100 Steps");
          em.persist(course1);
          Course course2 = findById(10001L);
          course2.setName("JPA in 50 Steps - Updated");
}
```

  In this example the updated course name will be saved in the database.
  By default the manager tracks everything. If you want to untrack a few entities, you can use EntityManager.detach(). If you want to clear everything out, you can call EntityManager.clear() directly.

EntityManager.flush()

- Synchronize the persistence context to the underlying database.
- The changes up to that point would be sent out (committed) to the database.

EntityManager.detach(Object entity)

- Remove the given entity from the persistence context, causing a managed entity to become detached.
- With this method, the given entity object is not tracked (for updates/deletions) by the Entity manager. So updates to that entity will not be saved in the database.

EntityManager.clear()

- Clear the persistence context, causing ALL managed entities to become detached.
- With this method, ALL the entity objects will not tracked (for updates/deletions) by the Entity manager. So updates those entities will not be saved in the database.

EntityManager.refresh(Object entity)

- Refresh the state of the instance from the database, overwriting changes made to the entity, if any.
- So basically the entity object will be fetched from database (by firing a SELECT query)

EntityManager.createQuery(String qlString)

- To create a JPQL query.
- There are 2 overwritten versions of JPQL createQuery.
    - createQuery(String qlString)
    - createQuery(String qlString, Class<T> resultClass)

JPA Annotations

- Refer section "Annotation Types Summary" of https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

Side Note

When we want to AUTOMATICALLY store inserted date and last updated date for any record, we have a solution for this from Hibernate (not JPA)

- Use @CreatedTimestamp and @UpdateTimestamp for the respective fields.
- Both these annotations are Hibernate specific.
- E.g.

```
@CreatedTimestamp
private LocalDateTime createdDate;

@UpdateTimestamp
private LocalDateTime lastUpdatedDate;
```

JPQL

- JPQL stands for a Java Persistence Query Language.
- Docs – https://docs.oracle.com/html/E13946_04/ejb3_langref.html
- It is a standard from JPA to perform queries against objects and domain classes. So instead of writing SQL queries against database tables and columns, we write JP queries against our objects and their fields.
- These JP queries are internally converted to native SQL queries by the ORM tools like Hibernate.
- Most of what is there in sql is also provided in JPQL keywords and syntax.
- JPQL is **case sensitive** when it comes to the domain class names and its field names but it is **not case sensitive** to when it comes to keywords in the language syntax itself like 'select', 'count', etc.
- There are 2 overwritten versions of JPQL EntityManager.createQuery.
    - createQuery(String qlString)
    - createQuery(String qlString, Class<T> resultClass)
- With named queries (@NamedQuery, @NamedQueries), you can assign a name to a query and use it.
- Example –
  Refer class JPQLTest of https://github.com/sameerbhilare/jpa-with-hibernate/tree/master/02-jpa-advanced

Native Queries

- Sometimes you need to use Native Queries instead of JPQL. With JPA, we can use EntityManager.createNativeQuery(String, qry).
- Also we can use named native query using annotations @NamedNativeQueriey, @NamedNativeQuery.
- There is no way to do bulk update through JPQL, so we have to use native queries in such cases.
- One of the things you'd need to make sure that whenever you are using a native query, you're not making use of persistence context.
  So if you have all these entities data present in your persistence context, then you need to make sure that you would refresh them so that you get the latest stuff from the database.

Criteria Queries

- Here are the steps of using Criteria API queries.
    1. Use Criteria Builder to create a Criteria Query returning the expected result type.
        - E.g.
          ```
          CriteriaBuilder cb = em.getCriteriaBuilder();
          CriteriaQuery<Course> cq = cb.createQuery(Course.class);
          ```
    2. Define roots for tables which are involved in the query.
        - Root defines which tables we are getting the data from.
        - E.g.
          ```
          Root<Course> courseRoot = cq.from(Course.class);
          ```
    3. Define Predicates etc. using Criteria Builder
        - E.g.
          ```
          Predicate like100Steps = cb.like(courseRoot.get("name"), "%100
          Steps");
          ```
    4. Add Predicates etc. to the Criteria Query
        - E.g. `cq.where(like100Steps);`
    5. Build the TypedQuery using the entity manager and criteria query
        - E.g.
          ```
          TypedQuery<Course> query = em.createQuery(cq.select(courseRoot));
          ```
    6. Execute the query.
        - E.g.
          ```
          List<Course> resultList = query.getResultList();
          ```
- Complete Example – Refer CriteriaQueryTest class from https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step57.md

Entity Manager, Transaction and Persistence Context

- Transaction is All or nothing operation.
- With @Tansactional annotation, a transaction is started at the start of the method and committed at the end of the method.
- With JPA, as soon as you define @Tansactional, you would also be creating Persistence Context.
- The persistence context is the place where all the entities which you are operating upon within a transaction are being stored.
- Persistent conduct is created at the start of the transaction and killed as soon as the transaction is ended.
- When we don't use @Tansactional, then each call on EntityManager's methods will act as its own transaction.
- JPA Persistence Context concept = Hibernate Session concept.
- JPA EntityManager class = Hibernate Session class.
- Either in the repository or in the Junit test, we should have @Tansactional.

FAQs – Hibernate and JPA

When does Hibernate send updates to Database?

- Hibernate waits **until the last possible moment** before it would start persisting the changes to the database.
- This is very efficient because if the transaction fails, it would then need to rollback already committed changes.
- If you use EntityManager.flush(), it will immediately save the changes to the database. If the transaction fails, hibernate will rollback the changes which are saved due to EntityManager.flush().

When do we need @Transactional in a Unit Test?

- The important thing for you to understand is if you want to do any change to the database using JPA, you would need a transaction.
- And when we are interacting through a repository, we provide a transaction in the repository (using @Transactional), so we don't need to use @Transactional again on the unit test method.
- However **if we directly use the entity manager in a unit test**, then the unit test has to provide the transaction boundary and that's what we do using the @Transactional annotation on that test method.

Do read only methods need a transaction?

- Without that transaction, you don't have a connection to the database.
- As long as you're firing queries using entity manage, you do not really need a transaction. But the moment you try to use an indirect way (via relationship) to fire a query, then you definitely need a transaction.
- **In eager loading**, we don't separate queries in relationship, so in eager loading case, we don't need transaction for read only methods.
- **In lazy loading** case, if you try to get details of a relationship (e.g. student.getCourse()), naturally hibernate will try to fire select query and it will fail if there is no transaction. So in this case, we need a transaction.

Why do we use @DirtiesContext in a Unit Test?

- When we write unit tests, you would want to leave the state of the application as it was before running that test. Use **`@DirtiesContext`** annotation, what it does is after that test is run, spring would automatically reset (rollback) the data.
- This is useful so that one test should not harm another tests.

Relationships

One To One

- By default, any one to one relationship is **eager fetch**.
- Use mappedBy attribute of @OneToOne annotation on the non-owning side of the relationship. It tells that the field on which @OneToOne annotation is used, is the owner of the relationship. The value of mappedBy attribute is the field name in the non-owning entity. Owner means the table which has the foreign key column in database.
- Complete Example – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step29.md

One To Many and Many To One

- Annotations – @OneToMany, @ManyToOne
- By default, **One to Many** relationship is **lazily fetched**. However on the **Many to One** side of the relationship the fetching is by default **eager fetching**.
- The "Many" side of the relationship will be the owning side of the relationship because in that table we will have foreign key column.
- Hence use mappedBy attribute on the **non-owning** side (@OneToMany) of the relationship. The value of mappedBy attribute is the field name in the non-owning entity.
- Complete Example – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step33.md

Many To Many

- In many to many relationship, we need to create a separate join/mapping table which will have primary keys of the two tables.
- Make one of the two entities the owning side of the relationship. In this case, it does not really matter which side of the relationship is the owning side of the relationship. So use mappedBy attribute on the **non-owning** side of the relationship. The value of mappedBy attribute is the field name in the non-owning entity.
- Customizing the join/mapping table – E.g. use different join table name, different foreign key names in the join table.
    - Use @JoinTable and pass actual database join table name.
    - Use @JoinTable and its attributes joinColumns and inverseJoinColumns, along with @JoinColumn to use different/actual database foreign key names.
- By default, any many to many relationship is **lazily fetched**.
- Complete Example – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step41_End_Of_Relationships.md

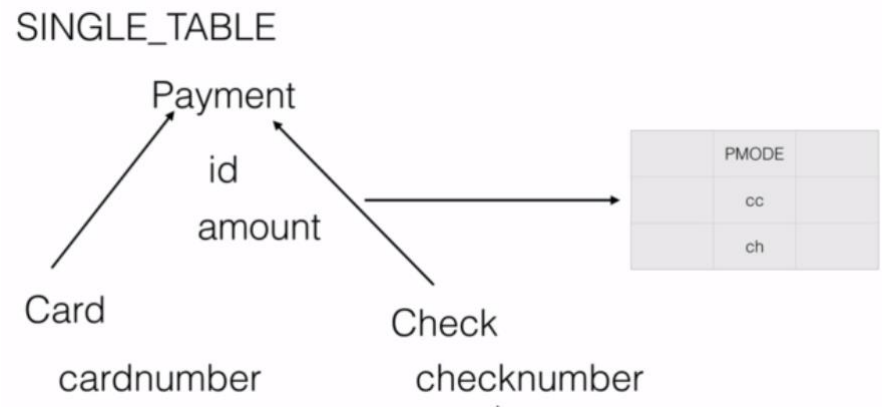
Inheritance Hierarchies with JPA and Hibernate

Introduction

- We use inheritance in Java classes to reuse common fields/methods. So it is common that we use inheritance for Entity classes as well, however the underlying databases do not support inheritance mapping across the database tables. This is known as **sub type problem** in ORM.
- To solve this problem JPA provides inheritance mapping through three types of strategies – SINGLE_TABLE, TABLE_PER_CLASS, JOINED.

JPA Inheritance Strategies

SINGLE_TABLE Strategy (Default)

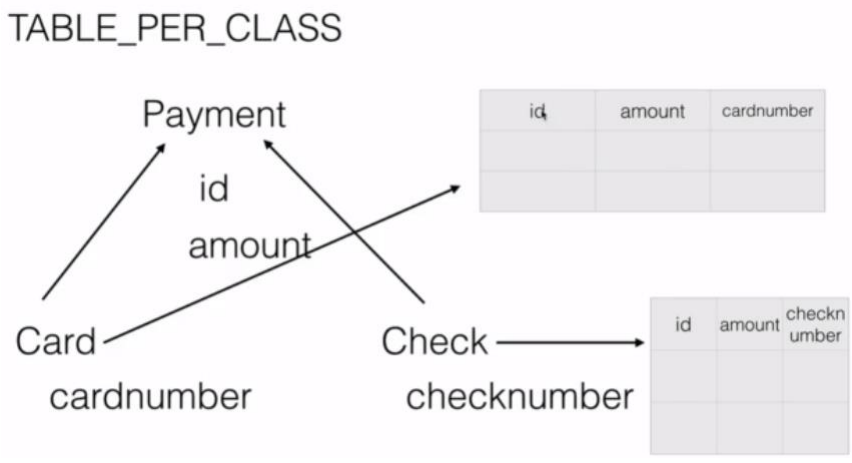


- As the name itself says in this case all the information whether it's card or cheque will go into one single table.
- And since it is going into one single table we need **extra discriminator column** (e.g. pmode in this example) that will differentiate between a card and a cheque.
- So we will provide values for that discriminator column representing a card or cheque so that hibernate will use those values in database to appropriately map both the Entity classes.
- To achieve this, we need to these annotations –
    - On parent class – `@Inheritance, @DiscriminatorColumn`
    - On child class – `@DiscriminatorValue`
- Example 1 – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step45.md
- Example 2 –

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
```

```
@DiscriminatorColumn(name = “pmode”, discriminatorType=
DiscriminatorType.STRING) // pmode column is from database table
public abstract class Payment {
	@Id
	private int id;
	// other common fields apart from the discriminator column
}

// child entity
@Entity
@DiscriminatorValue("cc")
public class CreditCard extends Payment{

	// some fields other than ID column
}

// child entity
@Entity
@DiscriminatorValue("ch")
public class Check extends Payment{

	// some fields other than ID column
}
```

TABLE_PER_CLASS Strategy

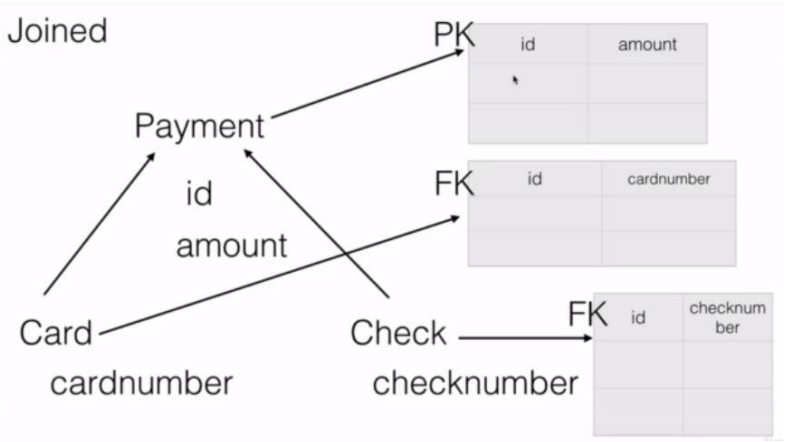


- Here we have **separate tables for child classes** (No table for parent as it is not required).
- So in our example, we will have a table for card as well as a table for check.
- Due to this, the tables of the subclasses will maintain **duplicate columns**.

- Although this approach improves performance, it is not recommended as it duplicates the columns across tables failing the normalization rules.
- To fetch data, JPA internally will use joins on both tables.
- To achieve this, we need `@Inheritance` annotation.
- Example 1 – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step46.md
- Example 2 –

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Payment {
	@Id
	private int id;
	// other fields and getters and setters
}

// child entity
@Entity
@Table(name="credit_card")
public class CreditCard extends Payment{
	// some fields other than ID column
}

// child entity
@Entity
@Table(name="bank_cheque")
public class Check extends Payment{
	// some fields other than ID column
}
```

JOINED Strategy



- This is the most popular and most used inheritance mapping strategies.

- Here every class in the inheritance hierarchy will have its own database table. The **parent** class will carry the **common fields** across the child classes and each **child** class will have table which will carry the field which is **specific** to that child. And the parent table is connected to each of the children through the primary key and **foreign key** relationship.
- The advantages is that it is one of the best inheritance mapping strategies which each table storing limited data. It follows normalization. That is the reason each table carries minimal data.
- If there is any disadvantage that is hibernate will have to join these tables to retrieve that data and it loads the data back. But that is expected in enterprise applications.
- To achieve this, we need `@Inheritance` and `@PrimaryKeyJoinColumn` annotations.
- Example 1 – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step47.md
- Example 2 –

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Payment {

	@Id
	private int id;
	// other fields
}

// child table

@Entity
@Table(name="card")
@PrimaryKeyJoinColumn(name="id")
public class CreditCard extends Payment{

	// child specific fields
}

// child table

@Entity
@Table(name="bankcheck")
@PrimaryKeyJoinColumn(name="id")
public class Check extends Payment{

	// child specific fields
}
```

Mapped Super Class Strategy

- Here we don't use @Inheritance, instead we use @MappedSuperClass just to designate parent table for mapping.
- A class annotated with @MappedSuperClass cannot be an Entity.
- It is similar to Table Per Class strategy but here we cannot query on the Super class in JPQL as here the Mapped Super Class cannot be an Entity.
- Example – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step49.md

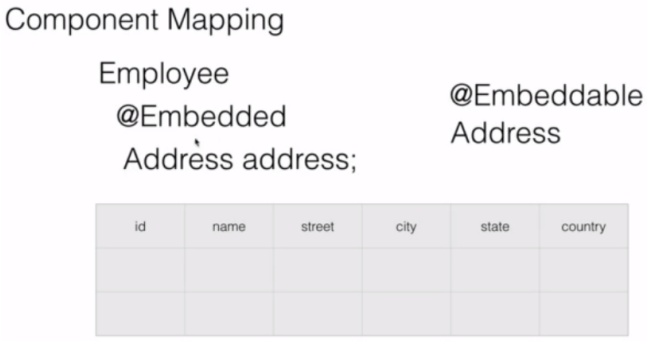
Which Option to Choose?

- Use JOINED strategy if data integrity is must. Note that it involves joins so performance will be slower compared to SINGLE_TABLE.
- Use SINGLE_TABLE strategy performance is must. Note that, it will affect data integrity.
- TABLE_PER_CLASS and Mapped Super Class are not good options as there will be duplicate columns and data.

Component Mapping

Introduction

- We use component mapping when we want to save one class that has a "HAS A" relationship with another class in the same database table.



- Here we mark the Address class with `@Embeddable` annotation from JPA. This tells the JPA implementations like Hibernate that this class is not an entity of its own but it will be embedded in another class.
- Also the Address field inside Employee class will be marked with `@Embedded` annotation. So hibernate will know that when an Employee is saved the embedded class object (Address) should also be saved into the same table.
- Example 1 – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step78.md
- Example 2 –

```
@Entity
public class Employee {
	@Id
	private int id;
	private String name;
	@Embedded
	private Address address;
	// getter and setters
}
@Embeddable
public class Address {
	private String street;
	private String city;
	private String state;
	private String zipcode;
	private String country;
	// getter and setters
}
```

Transaction Management

Introduction

- A transaction is an automic unit of work in which all off it happens or nothing happens.
- Every transaction has four key properties also known as ACID properties – Automicity, Consistency, Isolation and Durability.
- **Automicity** – either all will happen or nothing will happen that is atomic one single unit of work.
- **Consistency** – meaning the database should be left in a consistent state at the end of the transaction.
- **Isolation** – meaning if there are multiple transactions happening, each transaction should work in isolation without impacting each other.
- **Durability** – means once the transaction changes are committed, those changes should stay in the database.

Understanding Dirty, Phantom and Non Repeatable Reads

- These concepts come into picture when there are more than one transaction involved where in one transaction is updating some data and another transaction might read that data.

Dirty Read

- A dirty read occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed.
- This is when transaction 2 reads a value which was modified by transaction 1 before the transaction 1 is completed.

Non Repeatable Reads

- A non-repeatable read is one in which data read twice inside the same transaction cannot be guaranteed to contain the same value.
- So when I am reading the same value twice during the transaction, I get different values because some other transaction has updated that value (after it was read for first time by this transaction).

Phantom Read

- A phantom read occurs when, in the course of a transaction, new rows are added or removed by another transaction to the records being read.
- So at different times, we getting different number of rows in the same transaction because some other transaction inserted new rows or deleted some (after it was read for first time by this transaction).

Isolation Levels

- Four different kinds of isolation levels are –
    - Read Uncommitted
    - Read Committed
    - Repeatable Read
    - Serializable

Read Uncommitted

- Read Uncommitted is the lowest isolation level.
- In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads.
- In this level, transactions are not isolated from each other.

Read Committed

- This isolation level guarantees that any data read is committed at the moment it is read. Thus it does not allows dirty read.
- The transaction holds a read or write lock on the current row, and thus prevent other transactions from reading, updating or deleting it.

Repeatable Read

- This is the most restrictive isolation level.
- The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes.
- Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

Serializable

- This is the Highest isolation level.
- A serializable execution is guaranteed to be serializable. In background, it creates a table lock on the rows which match given constraint (e.g. age between 5 and 55).
- Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

Summary

- The Table is given below clearly depicts the relationship between isolation levels, read phenomena and locks :

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Choosing among Isolation Levels

- If you choose Serializable as isolation level, you will solve above read problems but you will have very poor performance.
- Typically the transaction isolation level which is used by most of the applications is **read committed**. It gives you enough guarantees about the quality of data and also ensures that the performance of the system is good. That would ensure that you are not really locking a lot of stuff. And also the system remains in a decently consistent way.
- It's also possible that even in the same application you can have different isolation levels for different kinds of transactions.

Difference between JPA Transactional and Spring Transactional

- JPA Transactional – javax.transaction.Transactional
- Spring Transactional – org.springframework.transaction.annotation.Transactional
- JPA Transaction allows us to manage transactions over a single database.
- If you want to manage transactions across multiple databases and MQs, use Spring transaction.
- With Spring Transaction, we can define isolation levels.
- Additional resource – https://www.baeldung.com/spring-vs-jta-transactional

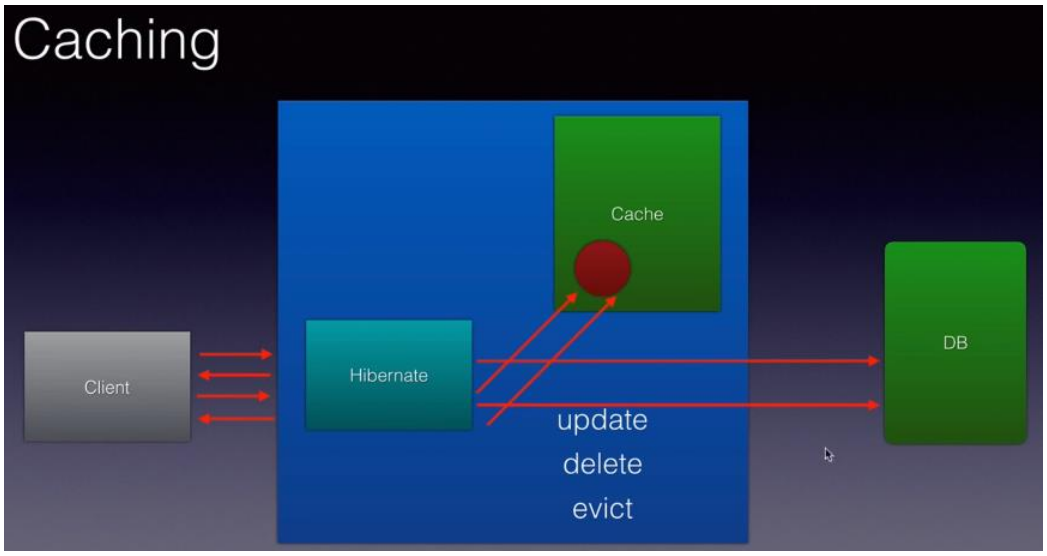
Spring Data JPA and Spring Data REST

Introduction

- **Spring Data** aims to provide simple abstraction to be able to access any kind of data and **Spring Data JPA** is the JPA specific implementation of Spring Data.
- Refer – "Spring Data JPA – Notes" from https://github.com/sameerbhilare/Spring-Data-JPA/blob/main/Resources/Spring%20Data%20JPA%20-%20Notes.pdf
- Repository interfaces –
    - `JpaRepository` extends `PagingAndSortingRepository` which extends `CrudRepository` which extends `Repository`
    - All these are interfaces.
    - `Repository` is a marker interface.
    - `JpaRepository` Javadoc – https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html
- Use Spring Data REST for prototyping only.

Caching with Hibernate

Introduction



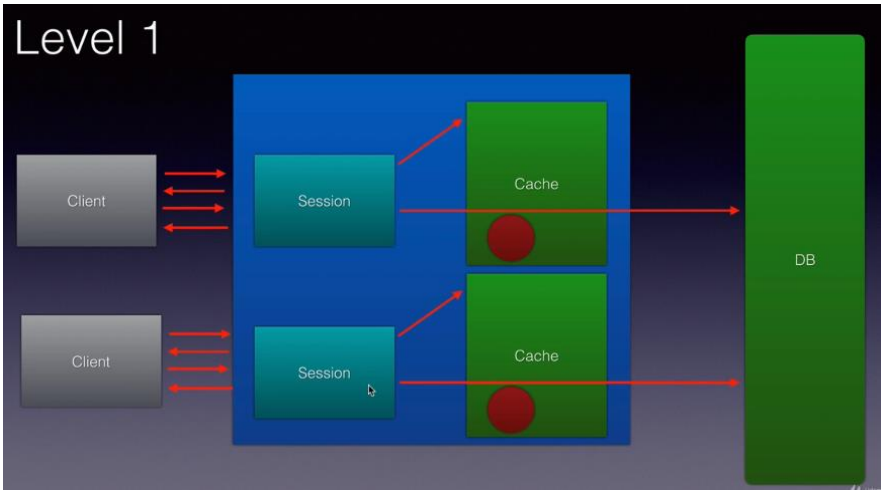
- Every time the client read some data from our application our application or the ORM tool will execute that select statement. So instead of repeating the same read operation multiple times we use caching or a cache.
- Caching is storing the data or an object in a temporary location.
- So the very first time that request comes in, these ORM tools or caching frameworks will read the data, convert them into object and store that object into a temporary memory location or even to the disk. This process is called caching. The next time the request comes in, these ORM frameworks or caching frameworks will first check if that data for that particular request exists in the cache. If it's there, there will be no database select queries executed .No database communication at all, simply pick the object, process it, send it back to the client, improving the performance of the application.
- And this cache can be refreshed or will be refreshed every time that object is updated that record is updated in the database automatically these caching frameworks or even hibernate will refresh the cache, it will read the data, store the object again in the cache which is updated and if it is deleted, it will be deleted from the cache. We can also programmatically evict the cache by using methods called evict on hibernates session object.
- Complete example – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step74-End-Of-Second-Level-Cache.md

Two Levels of Cache

- Hibernates supports caching at two levels –
    - The level 1 cache is at the hibernate **session** level. (added by default)
    - And level 2 cache is at the **session factory** level. (needs additional configuration)

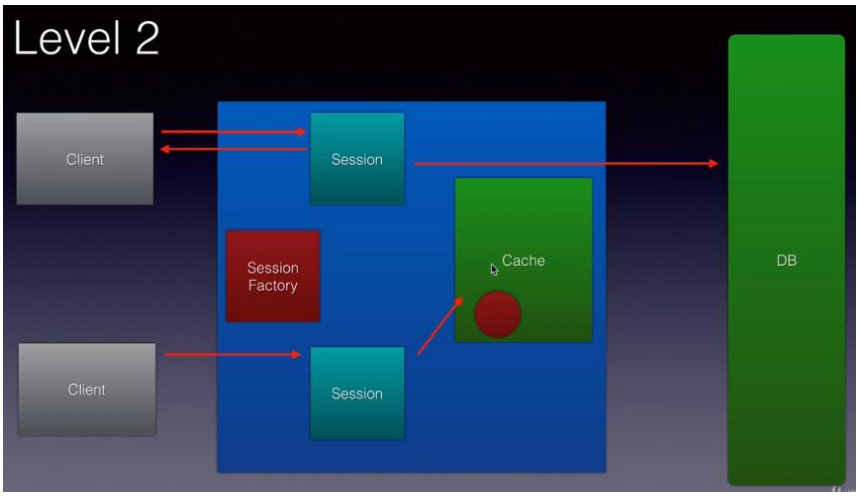
Level 1 Cache (Session Level)

- The first level of cache is within the boundary of a single transaction. It is active by default.



Level 2 Cache (SessionFactory Level)

- The second level cache comes into picture across multiple transactions.
- The second level cache will typically store the common information for all the users of an application. E.g. list of countries, currencies, etc.
- Second level cache needs configuration because we need to tell hibernate which '**frequently not changing or never changing**' data is to be cached. We need to use second level caching frameworks like ehcache.



- Internally if we use Level 1 cache which is free which is always there by default enabled, it happens at the session level.
- A sessionfactory is used to create multiple hibernate sessions. So if we use Level 2 caching, objects will be cached at the session factory level. That is they are shared across hibernate sessions.
- Level 2 cache is very powerful because data here is cached across sessions. But level 2 cache needs some additional work. Hibernate does not have in built support for it. We use caching providers such as **ehcache, Swaram cache, Jboss tree cache, OS cache, Tangosol cache** etc. but **ehcache** is the most popular one and very easy to configure.

Level 1 Cache (Session Level)

- By default, the level one cache will be enabled.
- In order for the hibernate's level 1 caching to work, we need to mark this with @Transactional annotation from spring because spring treats level 1 caching or the spring session is associated with that transaction and for it to work we have to mark it with it @Transactional.
- E.g.

```
@Test
@Transactional
public void testCaching() {

 repository.findById(1).get();
 repository.findById(1).get();

}
```

- In above example, if you fire the same select twice, you will see only one select query being fired in the logs (turn on show_sql to true).
- In above example, if you don't use @Transactional, Level 1 caching will not work and you will see 2 select queries in the console logs.
- If we **evict** the loaded object using session.evict(), the object will be removed from the Level 1 cache. And if it is requested again in the same transaction, hibernate will check the object in the level 2 cache if it is configured, otherwise it will fire another select query in the database.

EH cache (Level 2 Cache)

- Fast and lightweight.
- EHCache is second level cache provider, it will cache the objects at the session factory level.
- It supports both in memory as well disk based caching. We can store the objects in the memory in the RAM or it could serialize them to the disk and de-serialize them back as required.
- And it's very powerful because it allows us to configure timeout for a particular object in the cache, the total lifetime of an object in the cache etc. using XML.

Caching Concurrency Strategies

- Based on the use cases and our application we can select four different types of cache concurrency strategies when we cache our JPA entities namely READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL.
- Depending on which we choose, the caching will be impacted the way our objects will be cached our application uses caching will be impacted.

READ_ONLY

- If we choose READ_ONLY which should be chosen only when the entities in the application never change pretty much.
- If our application is a read only application which just takes the data in the database and displays it to the end user. That is when we should be using read only.
- If you use read only strategy on your JPA entity and once the data is ready, if you try to update that record in the database an exception will be thrown. So you should be careful.
- You should use read only when the applications are pretty much reading the data and displaying it to the end user and you are pretty sure that no updates will happen.

NONSTRICT_READ_WRITE

- This is not consistent cache but it's OK for several applications.
- So here the cache will be updated only when the particular transaction completely commits the data to the database.
- So in the meantime if another transaction tries to read that data it will get stale data because the other transaction hasn't committed the data to the database that will not be updated in the cache.
- So if your application is OK for **eventual consistency** then you should be using non-strict read write.

READ_WRITE

- If you need **total consistency** across transactions then you should use a read write strategy.
- This is where soft locks will be used.
- So if a transaction is updating the data which is already cached and the transaction is updating that particular record in that database, a soft lock will be associated with that Cache object within the cache. Now if another transaction comes in and it is trying to read the data it will look at this lock and it will know that another transaction is trying to update this particular object. And it should not read it from the cache and it will go directly against the database and get the latest data.
- So this strategy is more consistent So if your application needs more consistency and if it can't use the non strict read write it then go for the read write strategy.

TRANSACTIONAL

- Rarely used.
- You use this when you are dealing with XA transactions or distributed transactions. So cache changes are done in a distributed XA transaction.
- A change in a cache which could be a commit or a rollback will happen across databases and those changes will impact the cache as well.

Steps to using EH Cache

1. Add ehcache maven dependency. E.g. org.hibernate:hibernate-ehcache

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

2. Enable the caching for our application (in the application.properties).

```
#1. enable second level cache
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
#2. specify the caching framework - EhCache
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cach
e.ehcache.EhCacheRegionFactory
spring.cache.ehcache.config=classpath:ehcache.xml
#3. Only cache what I tell to cache. ALL, NONE, ENABLE_SELECTIVE, etc.
spring.jpa.properties.javax.persistence.sharedCache.mode=ENABLE_SELECTIVE
```

Here are available options for **Shared Cache Mode**

- ALL - All entities and entity-related state and data are cached.
- DISABLE_SELECTIVE –Caching is enabled for all entities except those for which Cacheable(false) is specified.
- **ENABLE_SELECTIVE** – Caching is enabled for all entities for Cacheable(true) is specified. Recommended.
- NONE – Caching is disabled for the persistence unit.
- UNSPECIFIED – Caching behavior is undefined: provider-specific defaults may apply.

3. Create the ehcache.xml.

This is where we specify that timeout, the location to store our objects, the temporary location where the object should be cached and all that will go into that.

```
<ehcache>
 <!-- diskStore - path to temp store where objects will be cached by
ehcache.
   This can be property or files system path.
   java.io.tmpdir => This is inbuilt variable in Java which points to
the temporary directory.
  -->
 <diskStore path="java.io.tmpdir"/>

 <defaultCache
  <!-- max elements in cache at a time -->
  maxElementsInMemory="100"

  <!-- false means they should not live forever.
            The cachable object should be destroyed after some time
otherwise we'll run out of memory  -->
  eternal="false"

  <!-- Remove the object fro mthe cahche if it is not accessed for
more than 5 seconds here. -->
  timeToIdleSeconds="5"

  <!-- after 10 seconds the object should be removed from the cache.
No matter what. -->
  timeToLiveSeconds="10"

  <!-- So if we have any memory issues, it will be over flow to disk.
It will be saved on the disk.-->
   overflowToDisk="true"/>
</ehcache>
```

- We will then mark our entities as cacheable. Simply use an annotation on our JPA entities to put them into the cache.

- Finally test both the first level and second level caching.

4. Make Entities Cacheable

- There are two ways to make entity cacheable.
    - Using JPA @Cacheable annotation.
      Refer this example – https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step74-End-Of-Second-Level-Cache.md
    - Using Hibernate @Cache annotation.
- Annotate the entity with @Cache annotation with appropriate caching strategy.
- It's always a good practice and a requirement for our entities to implement Serializable especially when we are using second level cache so that they can be written to the disk.
- As long as is in memory, it will work even without Serializable. But once it starts writing the objects to the disk our objects should be implementing Serializable.
- E.g.

```
@Entity
@Table
@Cache(usage = CacheConcerrencyStrategy.READ_ONLY)
public class Product implements Serializable {

 private static final long serialVersionUID = 1L;
 @Id
 private int id;
 // getters and setters
}
```

5. Testing Cache

- Example

```
@Test
@Transactional
public void testCaching() {
 Session session = entityManager.unwrap(Session.class);
 Product product = repository.findById(1).get(); // 1. DB

 repository.findById(1).get(); // 2. 1st Level cache

 session.evict(product);      // 3. Evicted from 1st level cache

 repository.findById(1).get(); // 4. Fetched from 2nd level cache
//and it will also put it in the first level cache

}
```

- Following the steps 1 to 4, we have second level cache setup and we already have first level cache by default. So when we run above test, only one select query will be fired.

Hibernate and JPA Tips

Hibernate Soft Deletes

- When you delete something from database, that row is completely deleted from the database. This is called **Hard Delete**.
- Sometimes when we delete something, we want to store history of that by adding a column to the database to track whether it active or not, whether it's deleted or not. This is called **Soft Delete**.
- So we want to make sure that whenever we delete an entity, we want to soft delete it b only changing a (custom) column value (e.g. set IS_DELELETED to true).
- Implementing soft delete is Hibernate's solution (not JPA solution)
- **`@SQLDelete`** – is used to fire the query when the entity is deleted (e.g. repo.deleteById())
- **`@Where`** – is used to filter out soft deleted record when firing select queries from other places.
- Complete example – Refer Course.java and CourseRepositoryTest from
- https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step76.md
- Small Snippet from above example –

```
@Entity
@SQLDelete(sql="update course set is_deleted=true where id=?")
@Where(clause="is_deleted = false")
public class Course {

  @Id
  @GeneratedValue
  private Long id;

  private boolean isDeleted;
}
```

Gotchas in Soft Delete

- **`@Where`** annotation is not applicable if you are using native queries. There you manually need to add where clause e.g. is_deleted=0
- Whenever we delete an entity the query in the **`@SQLDelete`** gets fired and the column gets updated in the database. However the entity in the cache does not get updated because hibernate does not know at all about what's happening in this query. Solution for this is to use @PreRemove hook. @PreRemove JPA lifecycle method is called whenever a row of this specific entity is deleted.

JPA Lifecycle Methods

- We can track different entity events using hooks provided by JPA.
- @PrePersist – before persist is called for a new entity.
- @PostPersist – after persist is called for a new entity.
- @PreRemove – before an entity is removed.
- @PostRemove – after an entity has been deleted.
- @PreUpdate – before the update operation.
- @PostUpdate – after an entity is updated.
- @PostLoad – after an entity has been loaded.

Using Enums with JPA

- Create enum class and use it in some entity class.
- Use **`@Enumerated`** annotation on the enum field in the entity class. This ensures that the values for this field are properly stored and retrieved.
- By **default**, the value that gets stored in the database is **ordinal** (ordinal means first enum value will have value as number 1 and so on).
- The **problem** with storing ordinal is if you add any new enum values, the ordinals will change because ordinal depends on the position in the enum and hence there will be inconsistency with old and new data.
- The **solution** to this problem is use string as enum type.
  E.g.
  ```
  @Enumerated(EnumType.STRING)
  private ReviewRating rating;
  ```
- Complete example – Refer ReviewRating enum and Review class from https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step79.md

Be cautious with Entity toString() method

- Be very careful about what you are putting in your toString() method of entity class otherwise you might end up with performance issues because if you use relationship columns in your toString method, it will load those entities as well.

When do you use JPA

- Don’t user JPA for batch related operations.
- Not recommended to use JPA when you have a performance intensive application. It would definitely not give you a performance similar to let's say a stored procedure.
- Use it for CRUD related and few joins related operations.
- Use it when you have SQL relational database.

Performance Tuning Tips with Hibernate and JPA

Measure Before Tuning

- Premature optimizations is the root of all evils.
- When you are doing performance tuning, do it only after you measure the performance. So measure the performance (benchmark) and then try to tune it.
- Enable and monitor stats in at least one of the environments.

Indexes

- Any relational database needs to have right indexes.
- Look at Execution Plans and identify the right indexes for your tables.

Use Appropriate Caching

- Understand first level and second level cache.
- Be careful about size of first level cache.
- Use second level cache if possible.
- The second level caching is specific to one instance of the application.
- If you have multiple instances of your application to manage load, use distributed cache.

Eager vs Lazy Fetch

- Use lazy fetch mostly.
- Remember the relationships **ToOne => by default eager fetch. And **ToMany are lazy fetch by default.

Avoid N+1 Query Problem

- **N+1 Query Problem**
  The N+1 query problem happens when the data access framework executed N additional SQL statements to fetch related data that could have been retrieved when executing the primary SQL query.
- One of the solutions to N+1 problem is to use eager fetch for that relationship. But the consequence is it is always fetched.
- So another **solution** is using **Entity Graph**. In this solution, we keep the relationship as lazy fetch only. Create entity graph and pass it as a **hint** to the select query.
  This will perform join and will retrieve all the details and once and won't fire subsequent 'N' different queries.
- Another **solution** is using **Join Fetch**. This will perform join and will fire single select query only.
- Complete Example – Refer class PerformanceTuningTest from https://github.com/sameerbhilare/jpa-with-hibernate/blob/master/02-jpa-advanced/Step86.md

Few more FAQs

How to connect to a different database with Spring Boot?

- Add the pom dependency related to your database.
- Configure application.properties as per your database.
- Refer – https://github.com/sameerbhilare/jpa-with-hibernate#connecting-to-my-sql-and-other-databases

Approach to design great applications with JPA

- The best way to start building applications with GPA is to start with the tables first.
- The approach to build great applications is to start thinking from your relationships, start thinking from your tables and then get down to your entities and then map you are entities with relationships.

Good Practices for developing JPA applications

- Make all internal variables as private.
- Use in-memory database like H2 for unit tests. Use data.sql for initial test data for unit test in /src/test/resources. H2 DB will automatically pickup that sql file and will execute it.

Tips and Tricks

- Refer – https://github.com/sameerbhilare/jpa-with-hibernate
- Spring tool suite (STS) is a very powerful IDE that helps us build spring based projects very easily. It is very similar to eclipse with support for special support for spring based projects.
- `spring-boot-starter-parent` is a **BOM**, which is a special type of pom. BOM stand for **Bill of Materials** within which all the versions of various libraries required for our projects are defined.
- `@RunWith(SpringRunner.class)` – with this we tell Spring Boot to use `SpringRunner` instead of default JUnitRunner class to run our Test classes.
- `@SpringBootTest` - This annotation tells spring boot to search for a class that is marked with spring Boot application and use that class to create a spring application context, a container with all the beans in that application so that we can start testing those beans in that application.
- Use mappedBy attribute of @OneToOne annotation on the non-owning side of the relationship. It tells that the field on which @OneToOne annotation is used, is the owner of the relationship. The value of mappedBy attribute is the field name in the non-owning entity. Owner means the table which has the foreign key column in database.
- OneToOne, ManyToOne => ***ToOne** => by default **eager** fetching.
- OneToMany, ManyToMany => ***ToMany** => by default **lazy** fetching.
- Use Spring Data REST for prototyping only.