#29 JPA-Part8

Wednesday, 22 January 2025 7:21 PM

Till now, our *Repository interface* looks like this

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {


}
```

And in service class, we used to invoke methods which are available i

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public UserDetails findByID(Long primaryKey) {
        return userDetailsRepository.findById(primaryKey).get();
    }

}
```

Then, we have something called: **Derived Query**

- Automatically generates queries from the methods.
- Need to follow a specific naming convention.
- Derived query used for GET/REMOVE operations but not for IN
    - Insert and Update operations is supported though "save

PartTree.java

```java
private static final String QUERY_PATTERN = "find|read|get|query|search|stream";
private static final String COUNT_PATTERN = "count";
private static final String EXISTS_PATTERN = "exists";
private static final String DELETE_PATTERN = "delete|remove";
private static final Pattern PREFIX_TEMPLATE = Pattern.compile( //
        "^(" + QUERY_PATTERN + "|" + COUNT_PATTERN + "|" + EXISTS_PATTERN + "|" +
```

"^(find|read|get|query|search|stream|count|exists|delete|remove)

Method name should start with either One of these values : find or read or get etc..

Uppercase Letter (ex: A,B,C etc..)

```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {


    List<UserDetails> findUserDetailsByName(String userName);
}
```

Query in which it get translates too:

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
```

Different Use cases:

And:

```
List<UserDetails> findUserDetailsByNameAndPhone(String userNam
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
```

```
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
        and ud1_0.phone=?
```

Or:

```
List<UserDetails> findUserDetailsByNameAndPhoneOrUserId(String userName,
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
        and ud1_0.phone=?
        or ud1_0.user_id=?
```

Comparison:

Part.java

```
BETWEEN(2, "IsBetween", "Between"),
IS_NOT_NULL(0, "IsNotNull", "NotNull"),
IS_NULL(0, "IsNull", "Null"),
LESS_THAN("IsLessThan", "LessThan"),
LESS_THAN_EQUAL("IsLessThanEqual", "LessThanEqual"),
GREATER_THAN("IsGreaterThan", "GreaterThan"),
GREATER_THAN_EQUAL("IsGreaterThanEqual", "GreaterThanEqual"),
BEFORE("IsBefore", "Before"),
AFTER("IsAfter", "After"),
NOT_LIKE("IsNotLike", "NotLike"),
LIKE("IsLike", "Like"),
STARTING_WITH("IsStartingWith", "StartingWith", "StartsWith"),
ENDING_WITH("IsEndingWith", "EndingWith", "EndsWith"),
IS_NOT_EMPTY(0, "IsNotEmpty", "NotEmpty"),
IS_EMPTY(0, "IsEmpty", "Empty"),
NOT_CONTAINING("IsNotContaining", "NotContaining", "NotContains"),
CONTAINING("IsContaining", "Containing", "Contains"),
NOT_IN("IsNotIn", "NotIn"),
IN("IsIn", "In"),
NEAR("IsNear", "Near"),
WITHIN("IsWithin", "Within"),
REGEX("MatchesRegex", "Matches", "Regex"),
EXISTS(0, "Exists"),
TRUE(0, "IsTrue", "True"),
FALSE(0, "IsFalse", "False"),
NEGATING_SIMPLE_PROPERTY("IsNot", "Not"),
SIMPLE_PROPERTY("Is", "Equals");
```

```
List<UserDetails> findUserDetailsByNameIsIn(List<String>
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name in (?)
```

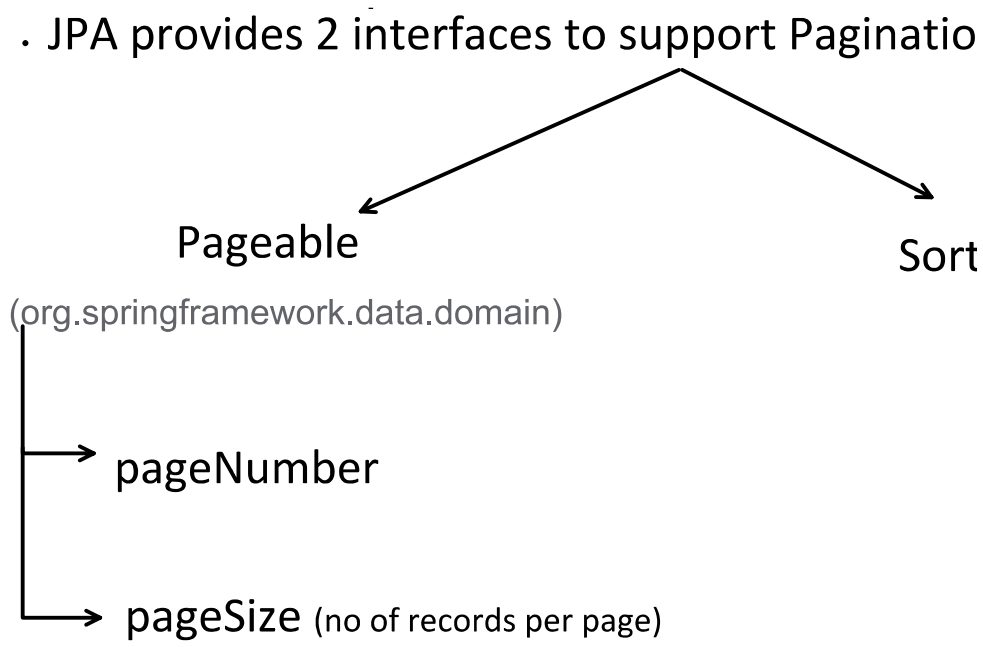
```
List<UserDetails> findUserDetailsByNameLike(String userNa
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name like ? escape '\'
```

Delete:

- Need to add @Transactional annotation.

Paginations and Sorting in Derived Quer




```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    List<UserDetails> findUserDetailsByNameStartingWith(String userName, Pa

}
```

```java
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

    public UserDetails saveUser(UserDetails user) {
        return userDetailsRepository.save(user);
    }

    public List<UserDetails> findByNameDerived(String name) {
        Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5)
        return userDetailsRepository.findUserDetailsByNameStartingWi
    }
}
```

If we need more info about Pages, then we

```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {
```

```
    Page<UserDetails> findUserDetailsByNameStartingWith(

}
```

```
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5
    Page<UserDetails> userDetailsPage = userDetailsRepository.f
    List<UserDetails> userDetailsList = userDetailsPage.getCont
    System.out.println("total pages: " + userDetailsPage.getTot
    System.out.println("is first page: " + userDetailsPage.isFi
    System.out.println("is last page: " + userDetailsPage.isLas
    return userDetailsList;
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	PHONE	USER_NAME
1	12312	A
2	12312	AB
3	12312	ABC
4	12312	ABCD
5	12312	ABCDE
6	12312	ABCDEF

(6 rows, 3 ms)

GET localhost:808

Params Authorization Heade

Query Params

Key

Key

Body Cookies Headers (5) Te

Pretty Raw Preview


```
[
    {
        "userId": 1,
        "name": "A",
        "phone": "123
    },
    {
        "userId": 2,
        "name": "AB",
        "phone": "123
    },
    {
        "userId": 3,
        "name": "ABC"
        "phone": "123
    },
    {
        "userId": 4,
        "name": "ABCD
        "phone": "123
    },
    {
        "userId": 5,
        "name": "ABCD
        "phone": "123
    }
]
```


```
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 1,  pageSize: 5);
    Page<UserDetails> userDetailsPage = userDetailsRepository.find
    List<UserDetails> userDetailsList = userDetailsPage.getContent
    System.out.println("total pages: " + userDetailsPage.getTotalP
    System.out.println("is first page: " + userDetailsPage.isFirst
    System.out.println("is last page: " + userDetailsPage.isLast()
    return userDetailsList;
}
```

Run | Run Selected | Auto complete | Clear | SQL statement:

```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	PHONE	USER_NAME
1	12312	A
2	12312	AB
3	12312	ABC
4	12312	ABCD
5	12312	ABCDE
6	12312	ABCDEF

(6 rows, 3 ms)

GET

Params Authori

Query Params

Key

Key

Body Cookies H

Pretty Raw


```
[
    {
    }
]
```

```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name like ? escape '\'
    offset
        ? rows
    fetch
        first ? rows only
total pages: 2
is first page: false
is last page: true
```

Paginations with Sorting:

Only Sorting:

- Sort.by accepts multiple fields.
- When multiple fields provided, sorting applied in order.
- first it sort by first field and if there are duplicates then seco

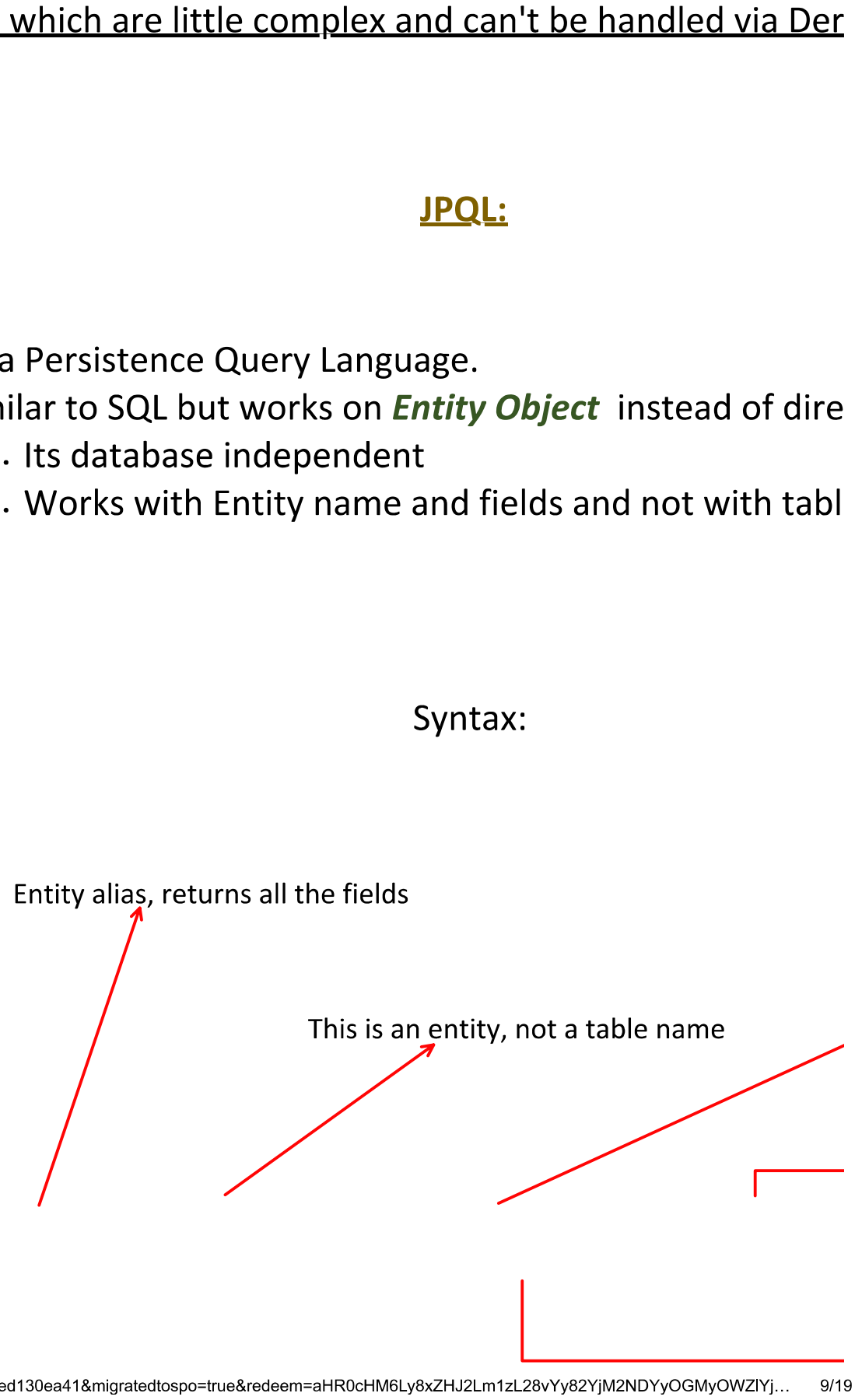
- If we need different sorting order for differen

Queries which are little complex and can't be handled via Der

JPQL:

- Java Persistence Query Language.
- Similar to SQL but works on ***Entity Object*** instead of dire
  - Its database independent
  - Works with Entity name and fields and not with tabl

Syntax:



There is no strict rule for Return type:
- you can return List or
-Single object
But, if say there are more than one rows, but in return type, we return Single Object, then JPQL will throw an exception

JPQL query with JOIN

- OneToOne

We don't specifically
JPA will automaticall

If we don’t, want Object[] to be used, we can also return direct custom DTO

- OneToMany

N+1 Problem and its Solution:

Problem :

Say, 1 User can have Many Addresses.
And our Query is such that, it can fetch more than 1 Users. Then this problem can occ

So, say we have 'N' Users. Then below queries will be hit by JPA:

- 1 query to fetch all the USERS.
- For each User it will fetch ADDRESSES, so for N users, it will fetch N times.

So total number of query hit : N+1.

So we need to find the way, so that only 1 QUERY it hit instead of N+1.

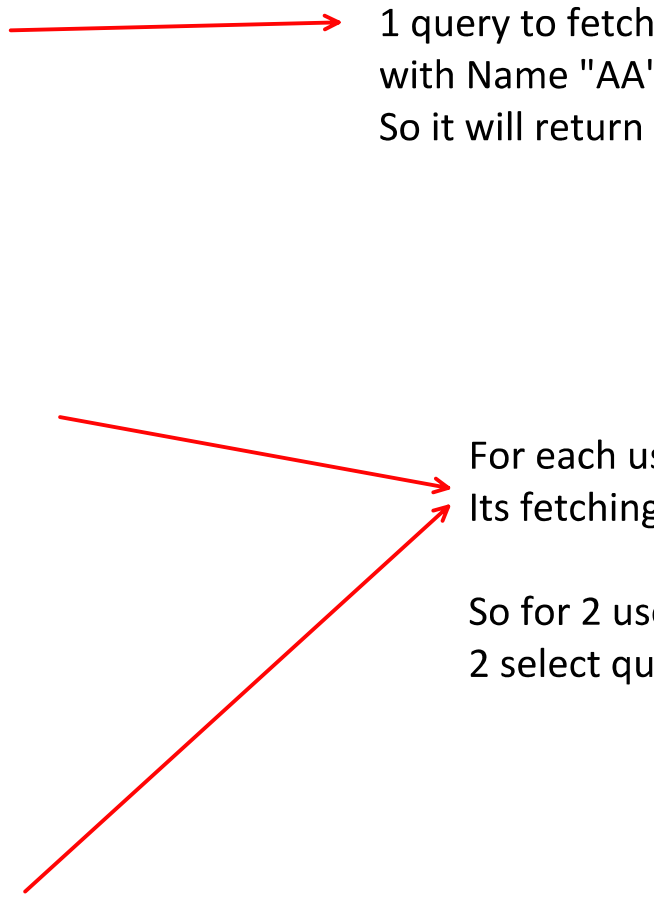
Before going for the solution for this problem, One question might be coming to our mind:

What if, we use EAGER initialization, then can we avoid this issue?

NO because EAGER initialization do not work, when our query tries to fetch multiple PAREN rows and that also have multiple CHILD.

In previous video, we tested EAGER with ***"findByID(id)"*** method, in which it make sure that query is fetching only 1 PARENT and that can have many CHILD, that's fine. In that JPA inte draft a JOIN query.

But when Multiple parent with Multiple child get involved, EAGER do not work in just 1 que first fetches all the parent and then for each parent, it fetch all its child.



So, how to solve this, N+1 problem?

Solution1: using ***JOIN FETCH*** (JPQL)

Solution2: using ***@BatchSize(size=10)***

- It wont make only 1 query, but it will reduce it, as it will divide it into batche

Solution3: using ***@EntityGraph(attributePaths="userAddressList")***

- Used over method (helpful in derived methods)
- Tell JPA to fetch all the entries of UserAddress along with user details.

How to join Many tables?

Its almost same as SQL only

Say, we have
Table A has one to many relationship with Table B
Table B has one to many relationship with Table C

```
@Query("SELECT a FROM A a  JOIN a.bList b  JOIN b.cList c WHERE c.someProperty = :someValue
List<A> findAWithBAndC(@Param("someValue") String someValue);
```

@Modifying Annotation

- when @Query annotation used, by-default JPA expects ***SELECT*** query.

- If we try to use "DELETE" or "INSERT" or "UPDATE" query with @Query, JPA will throw error, that:

- @Modifying annotation, is to tell JPA that, expect either "DELETE" or "INSERT" or "UPDATE" query with @Query

- Since we are trying to update the DB, we also need to use @Transactional annotation.

Understanding Usage of Flush and Clear:

- **As we know, Flush just pushed the persistence context changes to DB but hold the value in persistence context.**
- **Clear, purge the persistence context, and required fresh DB call**

Now using, Flush and Clear

Pagination and Sorting in JPQL

Same like discussed in derived query method

@NamedQuery Annotation

- We can name our Query, so that we can reuse it.