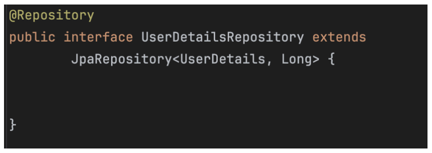
#29 JPA-Part8

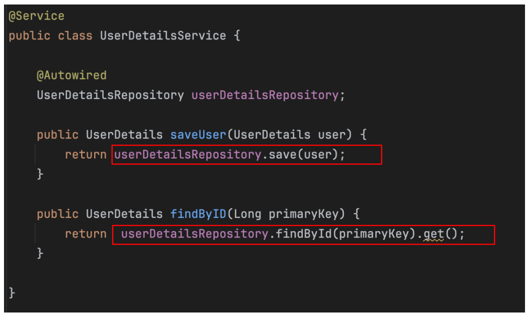
Wednesday, 22 January 2025 7:21 PM

Till now, our *Repository interface* looks like this



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


}
```

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

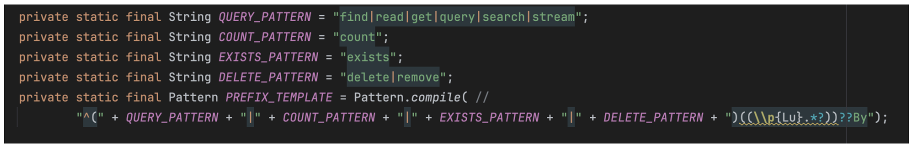


```
@Service
public class UserDetailsService {

    @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**

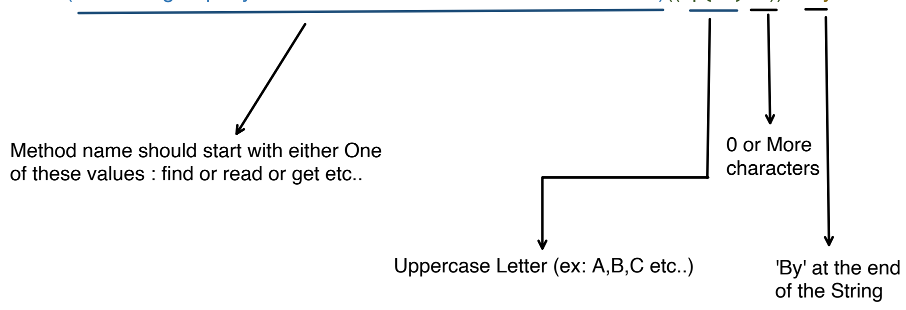
- Automatically generates queries from the methods.
- Need to follow a specific naming convention.
- Derived query used for GET/REMOVE operations but not for INSERT/UPDATE
  - Insert and Update operations is supported though "save()"

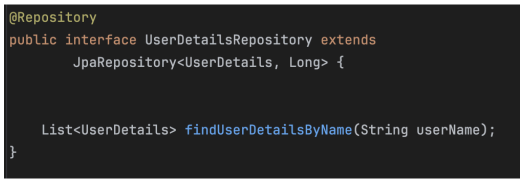
PartTree.java



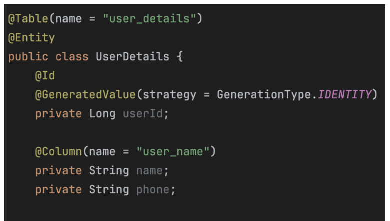
```
private static final String QUERY_PATTERN = "find|read|get|query|search|stream";
private static final String COUNT_PATTERN = "count";
private static final String EXISTS_PATTERN = "exists";
private static final String DELETE_PATTERN = "delete|remove";
private static final Pattern PREFIX_TEMPLATE = Pattern.compile( //
        "^(" + QUERY_PATTERN + "|" + COUNT_PATTERN + "|" + EXISTS_PATTERN + "|" + DELETE_PATTERN + ")((\\p{Lu}.*?))??By");
```

"^(find|read|get|query|search|stream|count|exists|delete|remove)((\\p{Lu}.*?))??By"


Method name should start with either One of these values : find or read or get etc..
0 or More characters
Uppercase Letter (ex: A,B,C etc..)
'By' at the end of the String



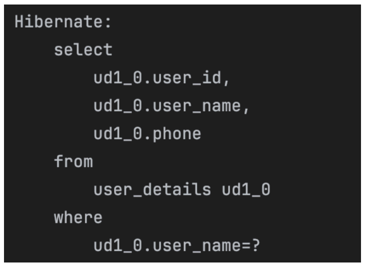
```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {


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



```
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;
```

```
  //getters and setters
}
```

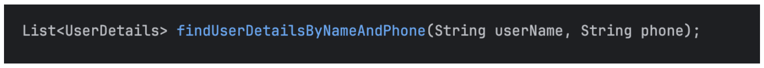
Query in which it get translates too:



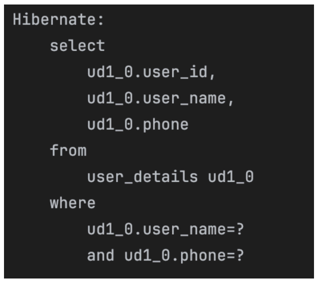
```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
```

Different Use cases:

And:

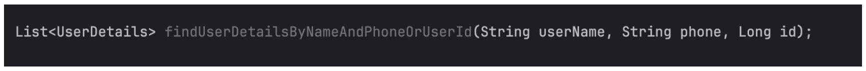


```
List<UserDetails> findUserDetailsByNameAndPhone(String userName, String phone);
```

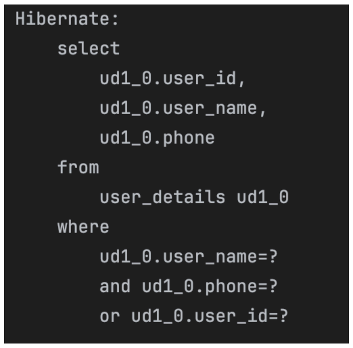


```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
        and ud1_0.phone=?
```

Or:



```
List<UserDetails> findUserDetailsByNameAndPhoneOrUserId(String userName, String phone, Long id);
```



```
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> userName);
```

```
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 userName);
```

```
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 '\'
```

<u>Delete:</u>

- Need to add @Transactional annotation.

```
@Transactional
void deleteByName(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=?
Hibernate:
    delete
    from
        user_details
    where
        user_id=?
Hibernate:
    delete
    from
        user_details
    where
        user_id=?
Hibernate:
    delete
    from
        user_details
    where
        user_id=?
```

Paginations and Sorting in Derived Query:

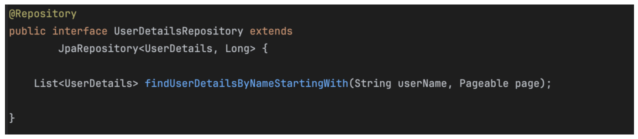
- JPA provides 2 interfaces to support Pagination and Sorting i.e.

$$\begin{tikzcd} & \text{ } \arrow[dl] \arrow[dr] & \\ \text{Pageable} & & \text{Sort} \end{tikzcd}$$

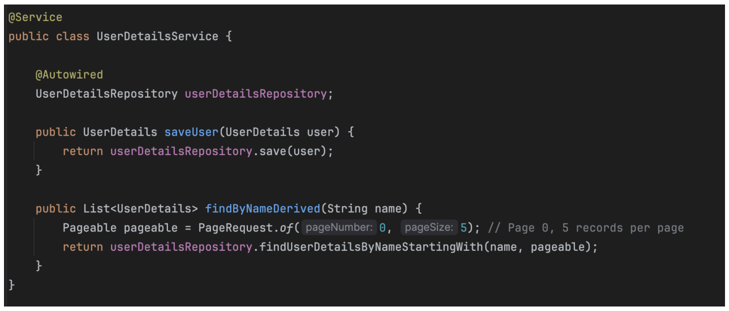
Pageable (org.springframework.data.domain)

→ pageNumber

→ pageSize (no of records per page)

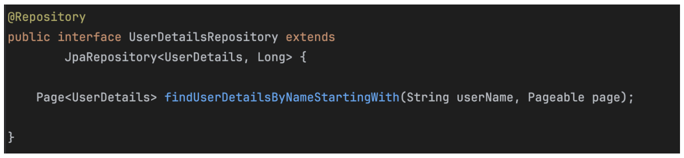


```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    List<UserDetails> findUserDetailsByNameStartingWith(String userName, Pageable page);

}
```

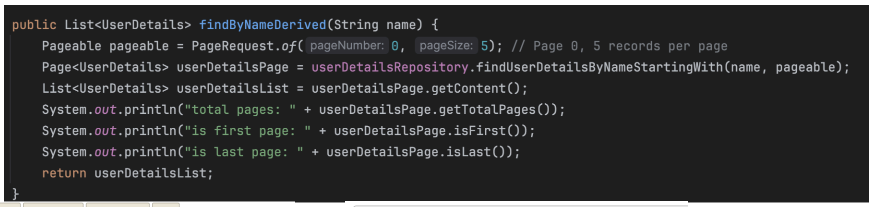


```java
@Service
public class UserDetailsService {

    @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); // Page 0, 5 records per page
        return userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    }
}
```

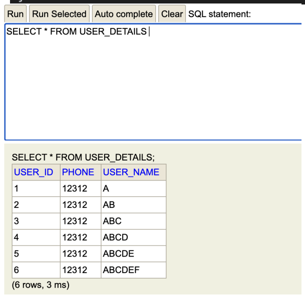
If we need more info about Pages, then we can use "Page" as return type



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

    Page<UserDetails> findUserDetailsByNameStartingWith(String userName, Pageable page);

}
```



```java
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5); // Page 0, 5 records per page
    Page<UserDetails> userDetailsPage = userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    List<UserDetails> userDetailsList = userDetailsPage.getContent();
    System.out.println("total pages: " + userDetailsPage.getTotalPages());
    System.out.println("is first page: " + userDetailsPage.isFirst());
    System.out.println("is last page: " + userDetailsPage.isLast());
    return userDetailsList;
}
```

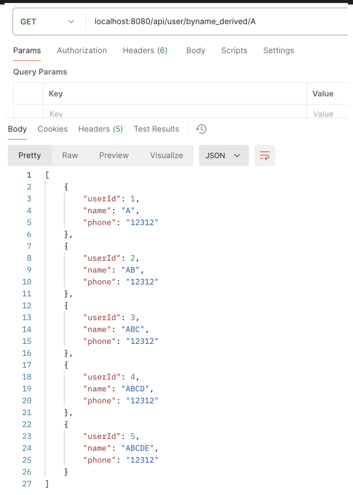


```sql
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

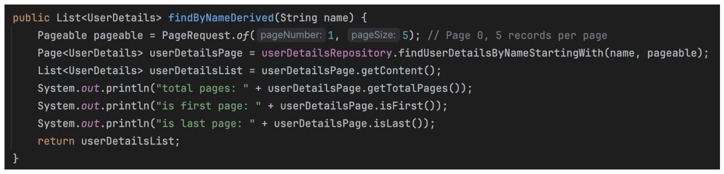
USER_ID	PHONE	USER_NAME
1	12312	A
2	12312	AB
3	12312	ABC
4	12312	ABCD
5	12312	ABCDE
6	12312	ABCDEF

(6 rows, 3 ms)

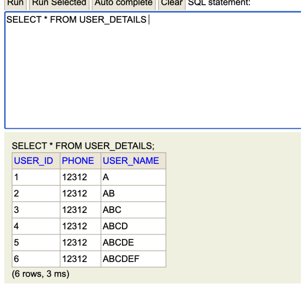


GET localhost:8080/api/user/byname_derived/A

```json
[
    {
        "userId": 1,
        "name": "A",
        "phone": "12312"
    },
    {
        "userId": 2,
        "name": "AB",
        "phone": "12312"
    },
    {
        "userId": 3,
        "name": "ABC",
        "phone": "12312"
    },
    {
        "userId": 4,
        "name": "ABCD",
        "phone": "12312"
    },
    {
        "userId": 5,
        "name": "ABCDE",
        "phone": "12312"
    }
]
```



```java
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 1,  pageSize: 5); // Page 0, 5 records per page
    Page<UserDetails> userDetailsPage = userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    List<UserDetails> userDetailsList = userDetailsPage.getContent();
    System.out.println("total pages: " + userDetailsPage.getTotalPages());
    System.out.println("is first page: " + userDetailsPage.isFirst());
    System.out.println("is last page: " + userDetailsPage.isLast());
    return userDetailsList;
}
```

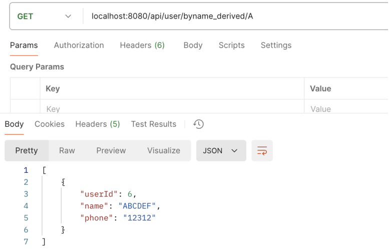


```
SELECT * FROM USER_DETAILS
```

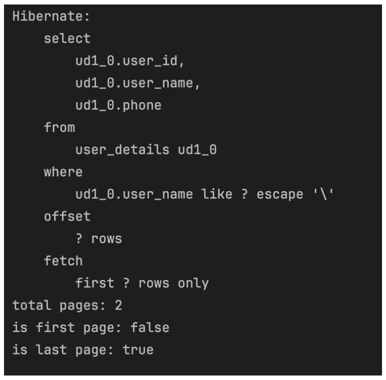
SELECT * FROM USER_DETAILS;

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)

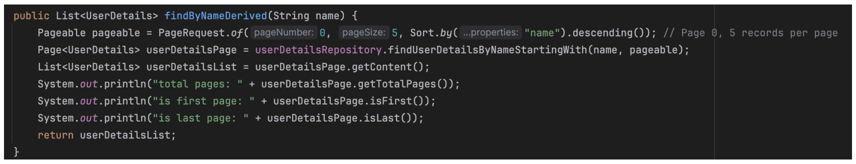


```
[
    {
        "userId": 6,
        "name": "ABCDEF",
        "phone": "12312"
    }
]
```

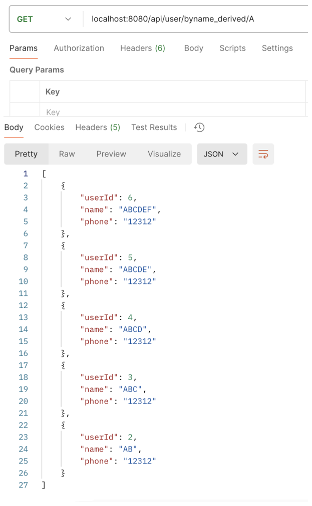


```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    where
        ud1_0.user_name like ? escape '\'
    offset
        ? rows
    fetch
        first ? rows only
total pages: 2
is first page: false
is last page: true
```

Paginations with Sorting:

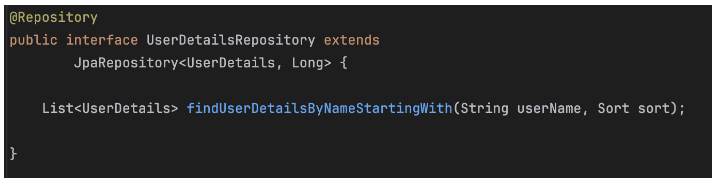


```
public List<UserDetails> findByNameDerived(String name) {
    Pageable pageable = PageRequest.of( pageNumber: 0,  pageSize: 5, Sort.by( ...properties: "name").descending()); // Page 0, 5 records per page
    Page<UserDetails> userDetailsPage = userDetailsRepository.findUserDetailsByNameStartingWith(name, pageable);
    List<UserDetails> userDetailsList = userDetailsPage.getContent();
    System.out.println("total pages: " + userDetailsPage.getTotalPages());
    System.out.println("is first page: " + userDetailsPage.isFirst());
    System.out.println("is last page: " + userDetailsPage.isLast());
    return userDetailsList;
}
```

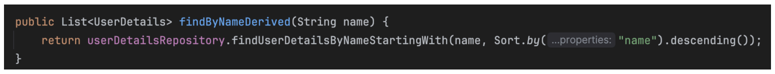


```
[
    {
        "userId": 6,
        "name": "ABCDEF",
        "phone": "12312"
    },
    {
        "userId": 5,
        "name": "ABCDE",
        "phone": "12312"
    },
    {
        "userId": 4,
        "name": "ABCD",
        "phone": "12312"
    },
    {
        "userId": 3,
        "name": "ABC",
        "phone": "12312"
    },
    {
        "userId": 2,
        "name": "AB",
        "phone": "12312"
    }
]
```

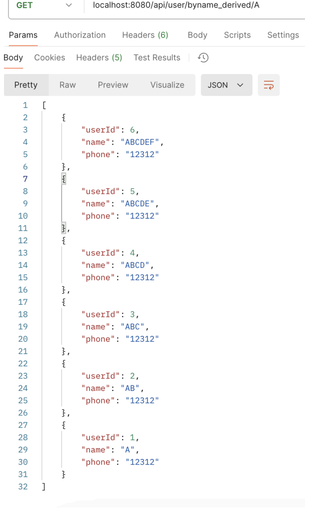
Only Sorting:



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

    List<UserDetails> findUserDetailsByNameStartingWith(String userName, Sort sort);

}
```



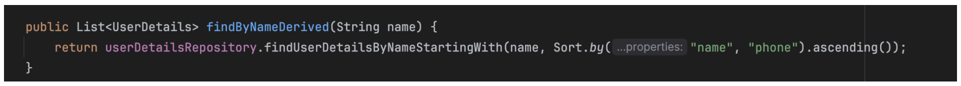
```
public List<UserDetails> findByNameDerived(String name) {
    return userDetailsRepository.findUserDetailsByNameStartingWith(name, Sort.by( ...properties: "name").descending());
}
```



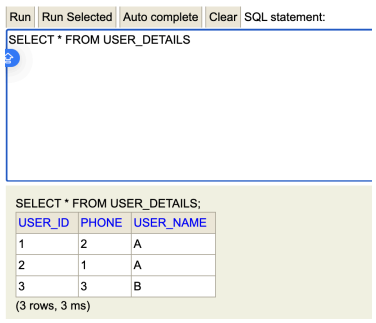
localhost:8080/api/user/byname_derived/A

```
[
    {
        "userId": 6,
        "name": "ABCDEF",
        "phone": "12312"
    },
    {
        "userId": 5,
        "name": "ABCDE",
        "phone": "12312"
    },
    {
        "userId": 4,
        "name": "ABCD",
        "phone": "12312"
    },
    {
        "userId": 3,
        "name": "ABC",
        "phone": "12312"
    },
    {
        "userId": 2,
        "name": "AB",
        "phone": "12312"
    },
    {
        "userId": 1,
        "name": "A",
        "phone": "12312"
    }
]
```

- 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 second field is used and so on.



```java
public List<UserDetails> findByNameDerived(String name) {
    return userDetailsRepository.findUserDetailsByNameStartingWith(name, Sort.by( ...properties: "name", "phone").ascending());
}
```

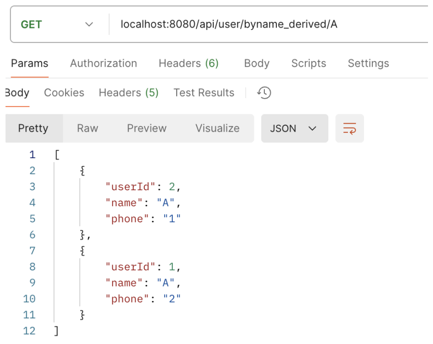


SELECT * FROM USER_DETAILS

SELECT * FROM USER_DETAILS;

USER_ID	PHONE	USER_NAME
1	2	A
2	1	A
3	3	B

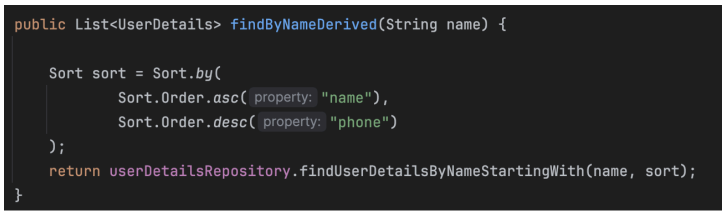
(3 rows, 3 ms)



localhost:8080/api/user/byname_derived/A

```
[
    {
        "userId": 2,
        "name": "A",
        "phone": "1"
    },
    {
        "userId": 1,
        "name": "A",
        "phone": "2"
    }
]
```

- If we need different sorting order for different fields



```java
public List<UserDetails> findByNameDerived(String name) {

    Sort sort = Sort.by(
            Sort.Order.asc( property: "name"),
            Sort.Order.desc( property: "phone")
    );
    return userDetailsRepository.findUserDetailsByNameStartingWith(name, sort);
}
```

<u>Queries which are little complex and can't be handled via Derived Query, we can use:</u>

<u>JPQL:</u>

- Java Persistence Query Language.
- Similar to SQL but works on ***Entity Object*** instead of direct database.
    - Its database independent

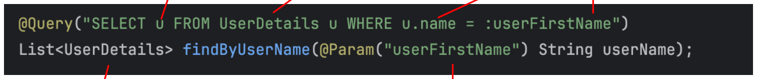
- Works with Entity name and fields and not with table column names.

Syntax:

Entity alias, returns all the fields

This is an entity, not a table name

This is an entity field name, not a column name



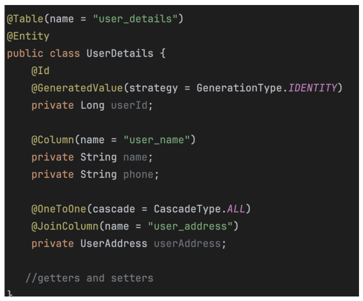
```java
@Query("SELECT u FROM UserDetails u WHERE u.name = :userFirstName")
List<UserDetails> findByUserName(@Param("userFirstName") String userName);
```

Binds, method parameter with named parameter in the query

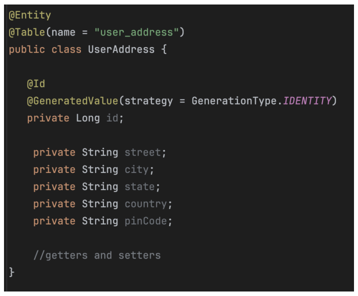
There is no strict rule for Return type:
- you can return List or
-Single object
But, if say there are more than one rows, but in return type, we return Single Object, then JPQL will throw an exception

JPQL query with JOIN

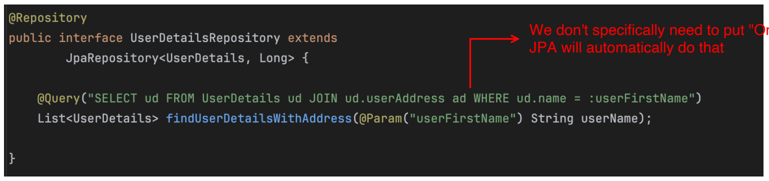
- OneToOne



```java
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_address")
    private UserAddress userAddress;

   //getters and setters
}
```

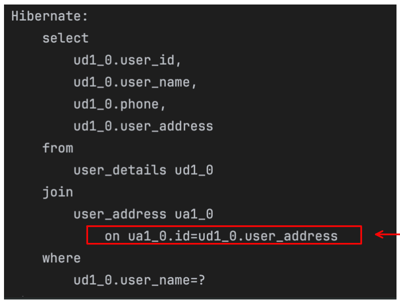


```java
@Entity
@Table(name = "user_address")
public class UserAddress {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

     private String street;
     private String city;
     private String state;
     private String country;
     private String pinCode;

     //getters and setters
}
```

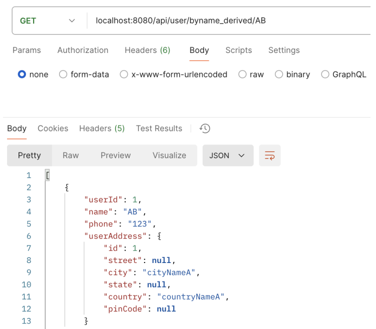


```java
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query("SELECT ud FROM UserDetails ud JOIN ud.userAddress ad WHERE ud.name = :userFirstName")
    List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);

}
```

We don't specifically need to put "On" here, JPA will automatically do that

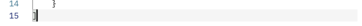


```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ud1_0.user_address
    from
        user_details ud1_0
    join
        user_address ua1_0
            on ua1_0.id=ud1_0.user_address
    where
        ud1_0.user_name=?
```

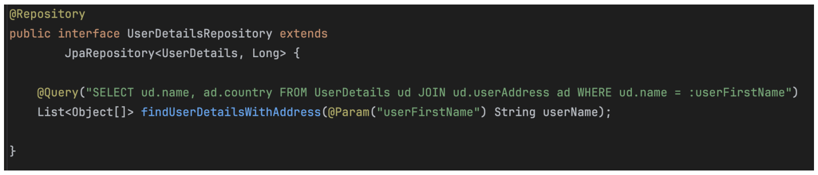


GET localhost:8080/api/user/byname_derived/AB

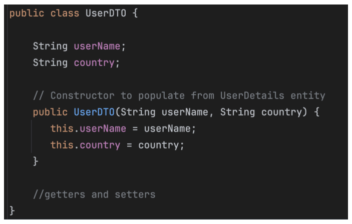
```json
[
    {
        "userId": 1,
        "name": "AB",
        "phone": "123",
        "userAddress": {
            "id": 1,
            "street": null,
            "city": "cityNameA",
            "state": null,
            "country": "countryNameA",
            "pinCode": null
        }
```



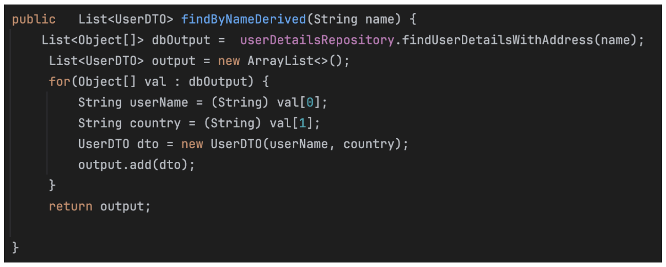
```
    }
}
```



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

    @Query("SELECT ud.name, ad.country FROM UserDetails ud JOIN ud.userAddress ad WHERE ud.name = :userFirstName")
    List<Object[]> findUserDetailsWithAddress(@Param("userFirstName") String userName);

}
```

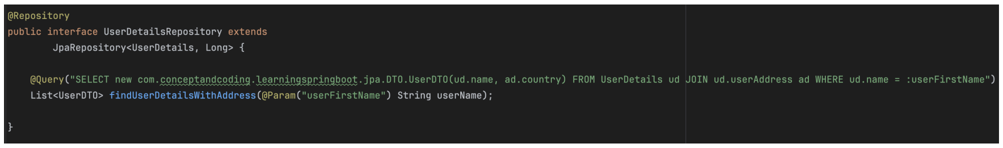


```
public class UserDTO {

    String userName;
    String country;

    // Constructor to populate from UserDetails entity
    public UserDTO(String userName, String country) {
        this.userName = userName;
        this.country = country;
    }

    //getters and setters
}
```



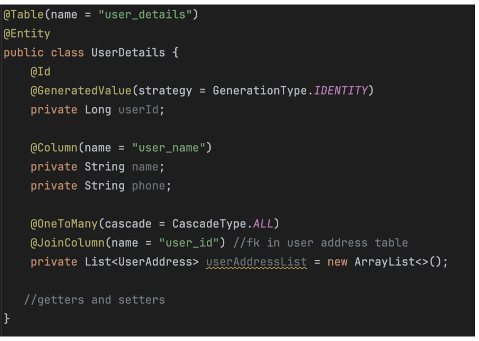
```
public   List<UserDTO> findByNameDerived(String name) {
    List<Object[]> dbOutput =  userDetailsRepository.findUserDetailsWithAddress(name);
     List<UserDTO> output = new ArrayList<>();
     for(Object[] val : dbOutput) {
         String userName = (String) val[0];
         String country = (String) val[1];
         UserDTO dto = new UserDTO(userName, country);
         output.add(dto);
     }
     return output;

}
```

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

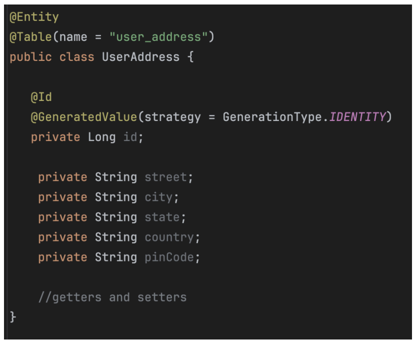


```
@Repository
public interface UserDetailsRepository extends
        JpaRepository<UserDetails, Long> {

    @Query("SELECT new com.conceptandcoding.learningspringboot.jpa.DTO.UserDTO(ud.name, ad.country) FROM UserDetails ud JOIN ud.userAddress ad WHERE ud.name = :userFirstName")
    List<UserDTO> findUserDetailsWithAddress(@Param("userFirstName") String userName);

}
```

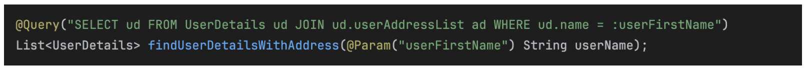
- OneToMany



```
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id") //fk in user address table
    private List<UserAddress> userAddressList = new ArrayList<>();

   //getters and setters
}
```



```
@Entity
@Table(name = "user_address")
public class UserAddress {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

    private String street;
    private String city;
    private String state;
    private String country;
    private String pinCode;

    //getters and setters
}
```



```
@Query("SELECT ud FROM UserDetails ud JOIN ud.userAddressList ad WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);
```

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 occurs.

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.

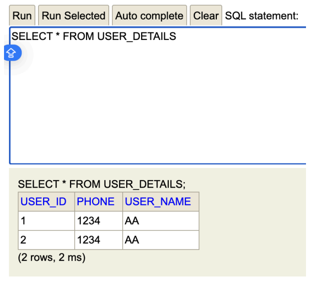
Before going for the solution for this problem, One question might be coming to our mind:

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 PARENT rows and that also have multiple CHILD.

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

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

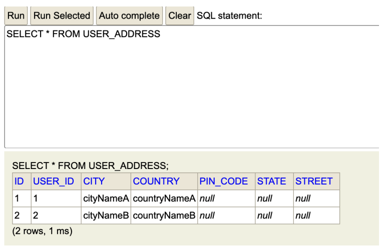


```
SELECT * FROM USER_DETAILS
```

SELECT * FROM USER_DETAILS;

USER_ID	PHONE	USER_NAME
1	1234	AA
2	1234	AA

(2 rows, 2 ms)

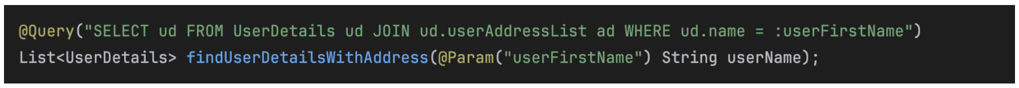


```
SELECT * FROM USER_ADDRESS
```

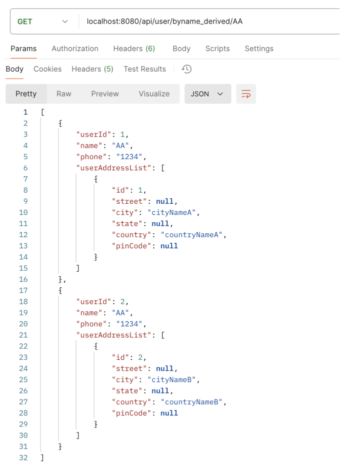
SELECT * FROM USER_ADDRESS;

ID	USER_ID	CITY	COUNTRY	PIN_CODE	STATE	STREET
1	1	cityNameA	countryNameA	*null*	*null*	*null*
2	2	cityNameB	countryNameB	*null*	*null*	*null*

(2 rows, 1 ms)

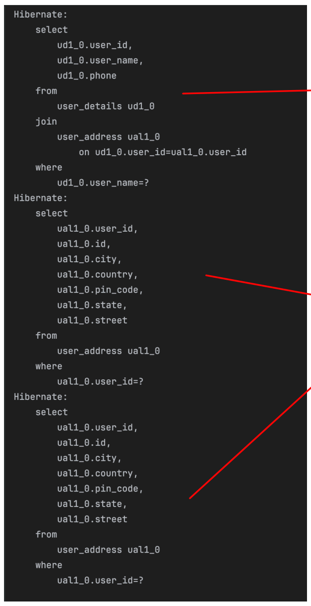


```
@Query("SELECT ud FROM UserDetails ud JOIN ud.userAddressList ad WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);
```



```
localhost:8080/api/user/byname_derived/AA
```

```
[
    {
        "userId": 1,
        "name": "AA",
        "phone": "1234",
        "userAddressList": [
            {
                "id": 1,
                "street": null,
                "city": "cityNameA",
                "state": null,
                "country": "countryNameA",
                "pinCode": null
            }
        ]
    },
    {
        "userId": 2,
        "name": "AA",
        "phone": "1234",
        "userAddressList": [
            {
                "id": 2,
                "street": null,
                "city": "cityNameB",
                "state": null,
                "country": "countryNameB",
                "pinCode": null
            }
        ]
    }
]
```



```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    join
        user_address ual1_0
            on ud1_0.user_id=ual1_0.user_id
    where
        ud1_0.user_name=?
Hibernate:
    select
        ual1_0.user_id,
        ual1_0.id,
        ual1_0.city,
        ual1_0.country,
        ual1_0.pin_code,
        ual1_0.state,
        ual1_0.street
    from
        user_address ual1_0
    where
        ual1_0.user_id=?
Hibernate:
    select
        ual1_0.user_id,
        ual1_0.id,
        ual1_0.city,
        ual1_0.country,
        ual1_0.pin_code,
        ual1_0.state,
        ual1_0.street
    from
        user_address ual1_0
    where
        ual1_0.user_id=?
```

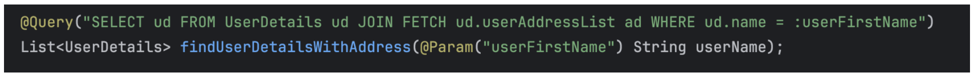
1 query to fetch all users
with Name "AA".
So it will return 2 users.

For each user
Its fetching all its addresses.

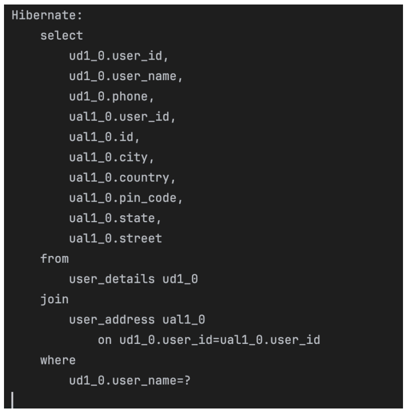
So for 2 users,
2 select query on child table

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

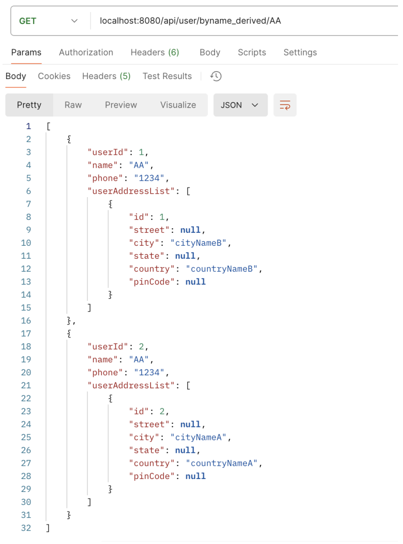
Solution1: using ***JOIN FETCH***
(JPQL)



```java
@Query("SELECT ud FROM UserDetails ud JOIN FETCH ud.userAddressList ad WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetailsWithAddress(@Param("userFirstName") String userName);
```



```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ual1_0.user_id,
        ual1_0.id,
        ual1_0.city,
        ual1_0.country,
        ual1_0.pin_code,
        ual1_0.state,
        ual1_0.street
    from
        user_details ud1_0
    join
        user_address ual1_0
            on ud1_0.user_id=ual1_0.user_id
    where
        ud1_0.user_name=?
```

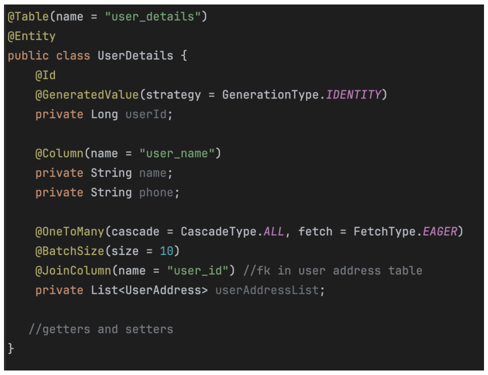


GET localhost:8080/api/user/byname_derived/AA

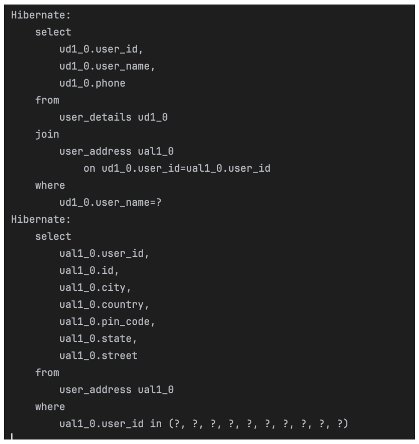
```json
[
    {
        "userId": 1,
        "name": "AA",
        "phone": "1234",
        "userAddressList": [
            {
                "id": 1,
                "street": null,
                "city": "cityNameB",
                "state": null,
                "country": "countryNameB",
                "pinCode": null
            }
        ]
    },
    {
        "userId": 2,
        "name": "AA",
        "phone": "1234",
        "userAddressList": [
            {
                "id": 2,
                "street": null,
                "city": "cityNameA",
                "state": null,
                "country": "countryNameA",
                "pinCode": null
            }
        ]
    }
]
```

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

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



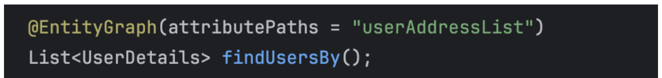
```java
@Table(name = "user_details")
@Entity
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @BatchSize(size = 10)
    @JoinColumn(name = "user_id") //fk in user address table
    private List<UserAddress> userAddressList;

   //getters and setters
}
```



```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone
    from
        user_details ud1_0
    join
        user_address ual1_0
            on ud1_0.user_id=ual1_0.user_id
    where
        ud1_0.user_name=?
Hibernate:
    select
        ual1_0.user_id,
        ual1_0.id,
        ual1_0.city,
        ual1_0.country,
        ual1_0.pin_code,
        ual1_0.state,
        ual1_0.street
    from
        user_address ual1_0
    where
        ual1_0.user_id in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

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.



```java
@EntityGraph(attributePaths = "userAddressList")
List<UserDetails> findUsersBy();
```

How to join Many tables?

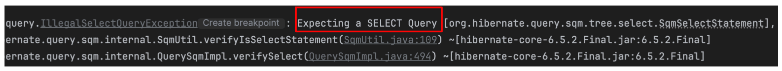
Its almost same as SQL only

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

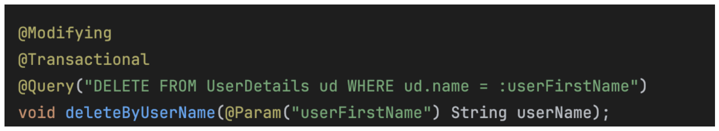
- when @Query annotation used, by-default JPA expects ***SELECT*** query.
- If we try to use "DELETE" or "INSERT" or "UPDATE" query with @Query, JPA will throw error, that:



```
query.IllegalSelectQueryException Create breakpoint : Expecting a SELECT Query [org.hibernate.query.sqm.tree.select.SqmSelectStatement],
ernate.query.sqm.internal.SqmUtil.verifyIsSelectStatement(SqmUtil.java:109) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
ernate.query.sqm.internal.QuerySqmImpl.verifySelect(QuerySqmImpl.java:494) ~[hibernate-core-6.5.2.Final.jar:6.5.2.Final]
```

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

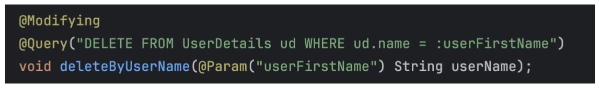
- Since we are trying to update the DB, we also need to use @Transactional annotation.



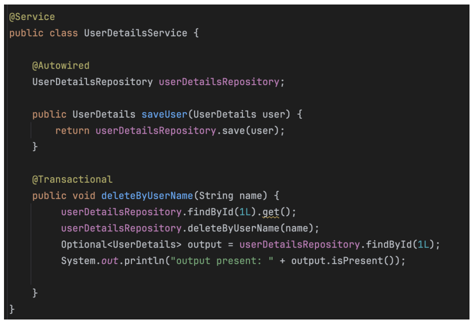
```
@Modifying
@Transactional
@Query("DELETE FROM UserDetails ud WHERE ud.name = :userFirstName")
void deleteByUserName(@Param("userFirstName") String userName);
```

Understanding Usage of Flush and Clear:

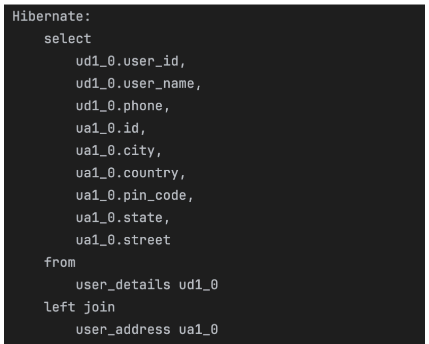
- **As we know, Flush just pushed the persistence context changes to DB but hold the value in persistence context.**
- **Clear, purge the persistence context, and required fresh DB call**



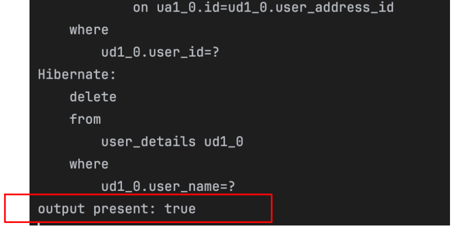
```
@Modifying
@Query("DELETE FROM UserDetails ud WHERE ud.name = :userFirstName")
void deleteByUserName(@Param("userFirstName") String userName);
```



```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

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

    @Transactional
    public void deleteByUserName(String name) {
        userDetailsRepository.findById(1L).get();
        userDetailsRepository.deleteByUserName(name);
        Optional<UserDetails> output = userDetailsRepository.findById(1L);
        System.out.println("output present: " + output.isPresent());

    }
}
```

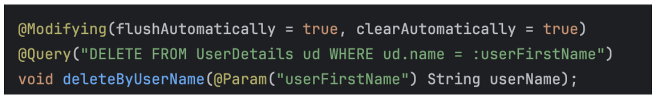


```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_details ud1_0
    left join
        user_address ua1_0
```

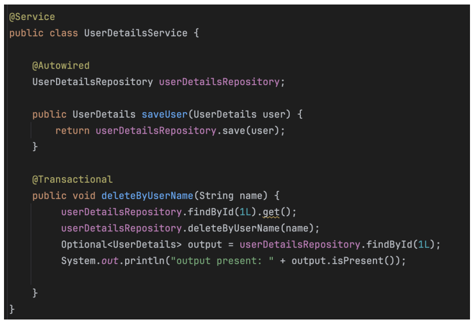


```
            on ua1_0.id=ud1_0.user_address_id
    where
        ud1_0.user_id=?
Hibernate:
    delete
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
output present: true
```

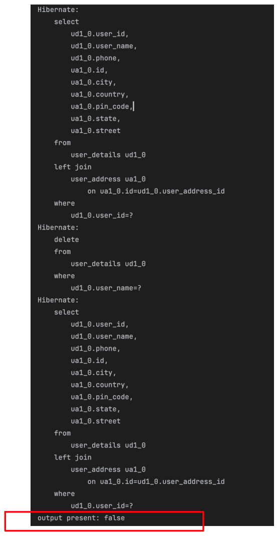
Now using, Flush and Clear



```
@Modifying(flushAutomatically = true, clearAutomatically = true)
@Query("DELETE FROM UserDetails ud WHERE ud.name = :userFirstName")
void deleteByUserName(@Param("userFirstName") String userName);
```



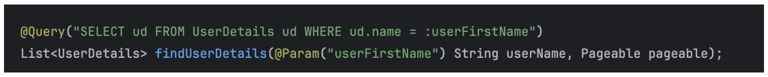
```
@Service
public class UserDetailsService {

    @Autowired
    UserDetailsRepository userDetailsRepository;

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

    @Transactional
    public void deleteByUserName(String name) {
         userDetailsRepository.findById(1L).get();
         userDetailsRepository.deleteByUserName(name);
         Optional<UserDetails> output = userDetailsRepository.findById(1L);
         System.out.println("output present: " + output.isPresent());

    }
}
```



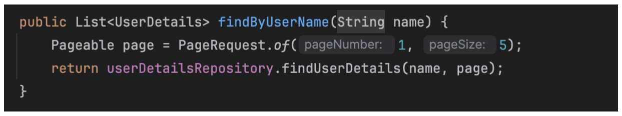
```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_details ud1_0
    left join
        user_address ua1_0
            on ua1_0.id=ud1_0.user_address_id
    where
        ud1_0.user_id=?
Hibernate:
    delete
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ua1_0.id,
        ua1_0.city,
        ua1_0.country,
        ua1_0.pin_code,
        ua1_0.state,
        ua1_0.street
    from
        user_details ud1_0
    left join
        user_address ua1_0
            on ua1_0.id=ud1_0.user_address_id
    where
        ud1_0.user_id=?
output present: false
```

Pagination and Sorting in JPQL

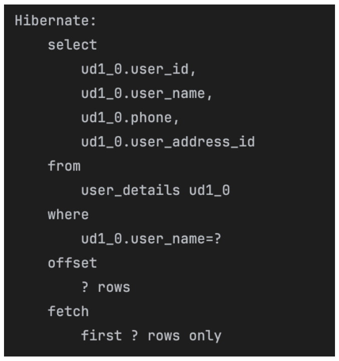
Same like discussed in derived query method



```
@Query("SELECT ud FROM UserDetails ud WHERE ud.name = :userFirstName")
List<UserDetails> findUserDetails(@Param("userFirstName") String userName, Pageable pageable);
```



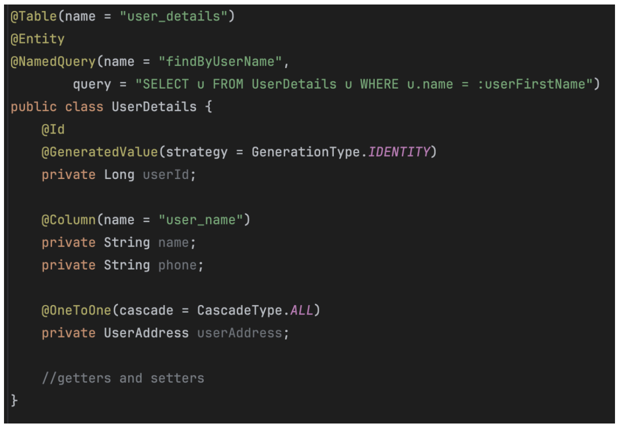
```
public List<UserDetails> findByUserName(String name) {
    Pageable page = PageRequest.of( pageNumber: 1,  pageSize: 5);
    return userDetailsRepository.findUserDetails(name, page);
}
```



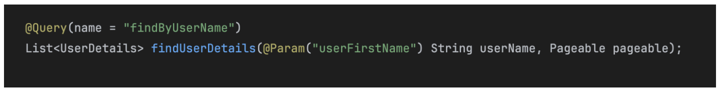
```
Hibernate:
    select
        ud1_0.user_id,
        ud1_0.user_name,
        ud1_0.phone,
        ud1_0.user_address_id
    from
        user_details ud1_0
    where
        ud1_0.user_name=?
    offset
        ? rows
    fetch
        first ? rows only
```

@NamedQuery Annotation

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



```java
@Table(name = "user_details")
@Entity
@NamedQuery(name = "findByUserName",
        query = "SELECT u FROM UserDetails u WHERE u.name = :userFirstName")
public class UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long userId;

    @Column(name = "user_name")
    private String name;
    private String phone;

    @OneToOne(cascade = CascadeType.ALL)
    private UserAddress userAddress;

    //getters and setters
}
```



```java
@Query(name = "findByUserName")
List<UserDetails> findUserDetails(@Param("userFirstName") String userName, Pageable pageable);
```