jpa-data

▼ Library

▼ Logging

- Logging SLF4J: **Interface**
- LogBack: **implementation**

```
org.springframework.boot:spring-boot-starter-data-jpa:2.6.6
--org.springframework.boot:spring-boot-starter-aop:2.6.6
----org.springframework.boot:spring-boot-starter:2.6.6
------org.springframework.boot:spring-boot-starter-logging:2.6.6
--------org.apache.logging.log4j:log4j-to-slf4j:2.17.2
--------org.slf4j:jul-to-slf4j:1.7.36
```

- p6spy

  In operation, performance test is needed as it may affected

```
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.7'
```

- Test library

  import junit.jupiter

▼ yml

```
spring:
  datasource:
    url: jdbc:h2:tcp://localhost/~/datajpa
    username: sa
    password:
    driver-class-name: org.h2.Driver

  jpa:
    hibernate:
      ddl-auto: create          # Only in developing stage!
    properties:
      hibernate:
        #show_sql: true         # print in console -> use hibernate.SQL
        format_sql: true

logging.level:
    org.hibernate.SQL: debug    # leave as a log
    #org.hibernate.type: trace  # To check parameter # It's better to use p6spy library.
```

- ▼ Spring Boot

  EntityManagerFactory that is made by springboot

  @PersistenceContext

    Springboot container brings JPA’s persistence context, ‘EntityManager’

    ```
    @Repository
    public class MemberJpaRepository {

        @PersistenceContext
        private EntityManager em;
    }
    ```

  Spring boot make EntityManagerFactory and c

- ▼ Basic
  - ▼ Default constructor access modifier

    Set access modifier of the default constructor to ‘protected’

    If the access modifier is ‘private’, the proxy use of ‘hibernate’ might be blocked.

    ```
    protected Member() {  }
    ```

  - ▼ @ToString, toString()

    DO NOT include related entity field. It leads to infinite loop

    ```
    @ToString(of = {..., "team"}) //
    public class Member {
      @ManyToOne(fetch = FetchType.LAZY)
      @JoinColumn(name = "team_id")
      private Team team;

    }
    ```

    ```
    @ToString(of = { ... , "members"})
    public class  Team {
      @OneToMany(mappedBy = "team") // It is recommended to write mappedBy on where foreign key doesn't exist.
      private List<Member> members = new ArrayList<>();

    }
    ```

JSON call has same problem → @JsonIgnore

▼ mappedBy

It is recommended to write mappedBy on where foreign key doesn't exist. (ref.)

▼ Optional<T>

The result might be ‘null’, so it is found in Optional

Then,

```
Optional<Member> find = memberRepository.findById(savedMember.getId());

if(find == null) {
      ...
} else {
  Member findMember = find.get();

  assertThat(findMember.getId()).isEqualTo(member.getId());
  assertThat(findMember.getUsername()).isEqualTo(member.getUsername());
  assertThat(findMember).isEqualTo(member);
}
```

```
```

▼ Update

Update method no need because jpa use dirty check for updates

▼ Errors

Best: Compile error

Good: error occurred when application starts

Worst: Error shown to client

▼ Relation mapping

@xToMany

It is recommended to write ‘mappedBy’ on where foreign key doesn’t exist.

```
public class  Team {
  ...
  @OneToMany(mappedBy = "team")
  private List<Member> members = new ArrayList<>();

  ...
}
```

▼ @Data

only for simple DTO

```
@Data // for simple DTO
public class MemberDto {

  private Long id;
  private String username;
  private String teamName;

  public MemberDto(Long id, String username, String teamName) {
    this.id = id;
    this.username = username;
    this.teamName = teamName;
  }
}
```

▼ Separate repositories

1. core business logic.
2. Tailored to view.

   having DTO and complicating queries.

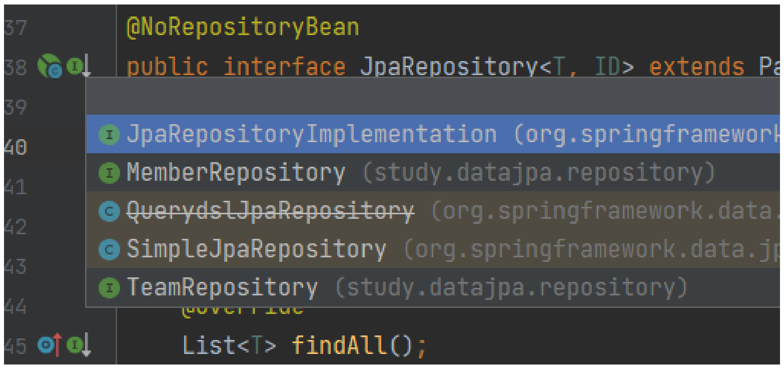
   this repositories will be altered when a view changes.

```
@Transactional
class MemberRepositoryTest {
  @Autowired MemberRepository memberRepository;
  @Autowired MemberQueryRepository memberQueryRepository;
}
```

Separate a repository for core business logic from a repository that is tailored to view (using DTO and complicating queries)

▼ Convert into DTO

▼ Check implementer



```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends Pa
```

JpaRepositoryImplementation (org.springframework
MemberRepository (study.datajpa.repository)
~~QuerydslJpaRepository~~ (org.springframework.data.
SimpleJpaRepository (org.springframework.data.jp
TeamRepository (study.datajpa.repository)

```
List<T> findAll();
```

- ▼ Test

  @SpringBootTest : To get spring bean

  @Transactional

  All changes to JPA shall be made within the transaction

  By default, DB is rolled back after testing so there's no query log left.

  @Rollback(false) : To check query log. @transactonal by default

```
@SpringBootTest
@Transactional
@Rollback(false)
class MemberJpaRepositoryTest {

  @Autowired MemberJpaRepository memberJpaRepository;

  @Test
  public void testMember() {
    Member member = new Member("memberA");
    Member savedMember = memberJpaRepository.save(member);

    Member findMember = memberJpaRepository.find(savedMember.getId());

    assertThat(findMember.getId()).isEqualTo(member.getId());
    assertThat(findMember.getUsername()).isEqualTo(member.getUsername());
    assertThat(findMember).isEqualTo(member); // findMember == member;
  }
}
```

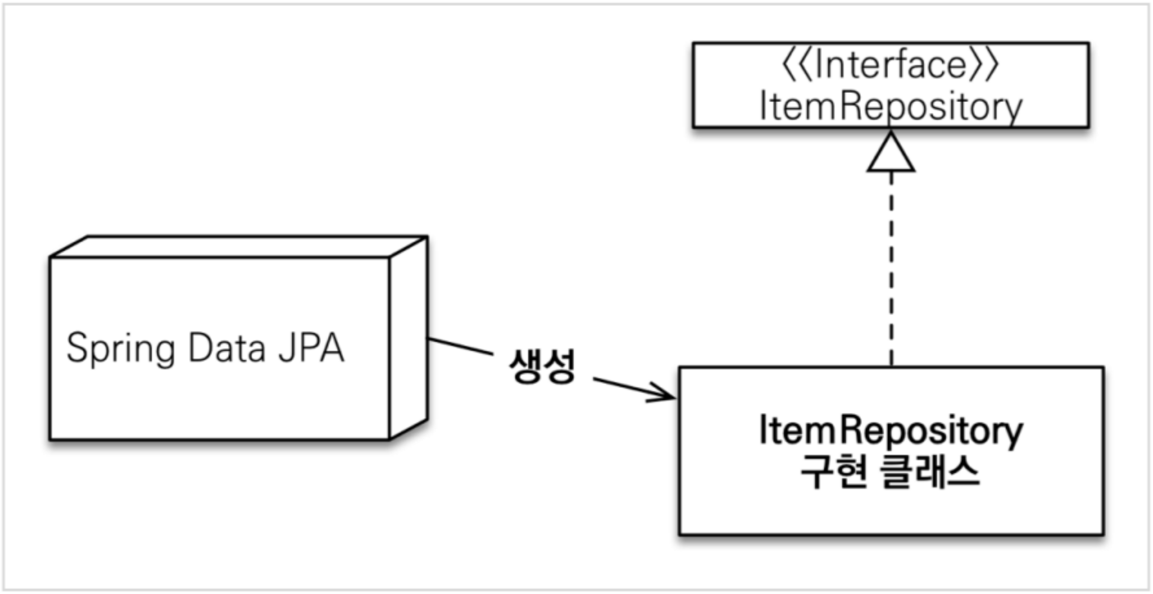
- ▼ Repository Common interface
  - ▼ Setting

    Based on where the '@SpringBootApplication' is applied,

    'Spring data jpa' automatically brings all the things in(or under) the package.

  - ▼ How does interface work?

    'Spring Data Jpa' makes implementation and inject it into the interface.

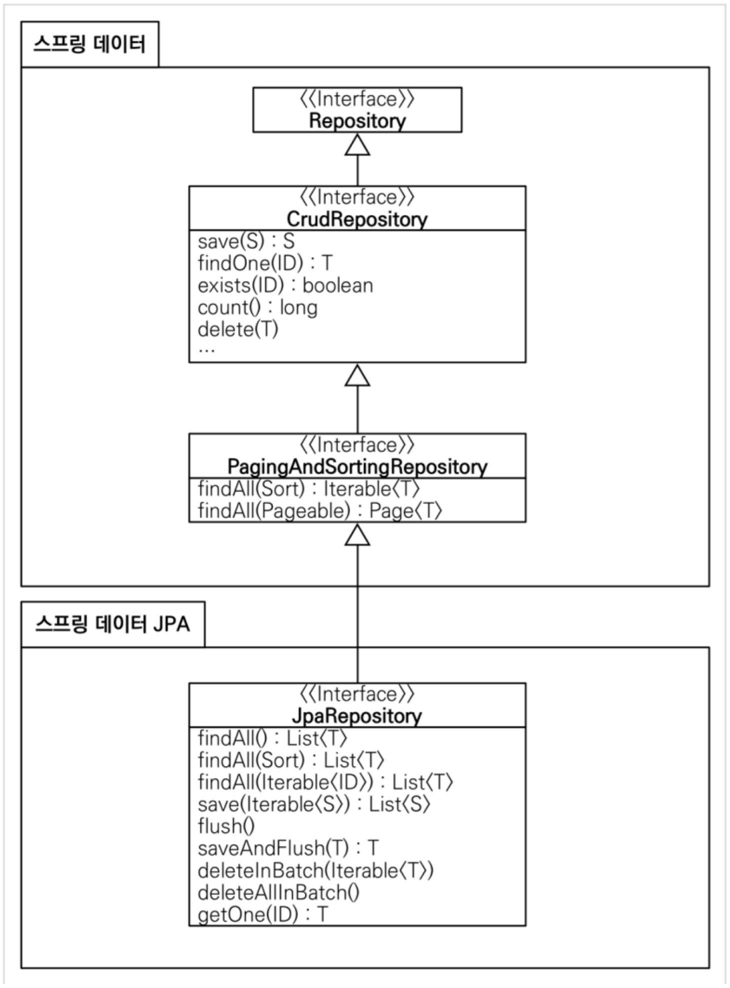


▼ Example code

@Repository is omittable

```
public interface MemberRepository extends JpaRepository<T, Long> {
}
```

▼ Composition



▼ methods examples in Jpa Repository

- findById(ID)

  find an entity.

  EntityManager.find()

- getOne(ID)

  find and entity as proxy

  EntityManager.getReference()

- findAll(...)

  Sort, Pageable conditions can be used as a parameter.

▼ Query method

- ▼ 1. Query creation by method name - for simple and short query

  (ref.)

  Useful for simple and short queries.

  If there are many conditions, use ‘3. @Query’

  Query method name has to be changed too when a field name of an entity changes.

  Error occurs when application starts (Good error)

  No description after ‘By’, select all.

```
public interface MemberRepository extends JpaRepository<Member, Long> {

  List<Member> findByUsernameAndAgeGreaterThan(String username, int age);


  List<Member> findHelloBy();
  List<Member> findTop3HelloBy();
}
```

  - ▼ inquiry (ref.)

    findXxxBy

    readXxxBy

    queryXxxBy

    getXxxBy

  - ▼ count

    countXxxBy: long

  - ▼ exist

existsXxxBy: boolean

- ▼ delete || remove

  deleteXxxBy: long

  removeXxxBy: long

- ▼ distinct

  findDistinct

  findXxxDistinctBy

- ▼ Limit (ref.)

  findFirst

  findFirst3

  findTop

  findTop3

▼ 2. JPA NamedQuery call by method name - impractical

Impractical. Not used in working-level

- Advantage

  find error when loading by parsing.

▼ 3. @Query_ Manual definition in repository interface- for many conditions

Practical, widely used in working-level

Useful when a query contains many conditions.

- Advantage

  find error when loading by parsing.

- ▼ Value or DTO searching
  - value

```
// get simple value
  @Query("select m.username from Member m")
  List<String> findUsernameList();
```

- DTO

  Alternative → QueryDSL

  ```
  // get in DTO
    @Query("select new study.datajpa.dto.MemberDto(m.id, m.username, t.name) from Member m join m.team t")
    List<MemberDto> findMemberDto();
  ```

▼ Parameter binding

name base

location base (Impractical)

- Collection parameter binding

  ```
  @Query("select m from Member m where m.username in :names")
    List<Member> findByNames(@Param("names") Collection<String> names);
  ```

  ```
  @Test
    public void findByNamesTest() {
      Member m1 = new Member("AAA", 10);
      Member m2 = new Member("BBB", 20);
      memberRepository.save(m1);
      memberRepository.save(m2);

      List<Member> result = memberRepository.findByNames(Arrays.asList("AAA", "BBB"));

      for (Member member : result) {
        System.out.println("member = " + member);
      }
    }
  ```

▼ Return type

(ref.)

▼ Single value

USE OPTIONAL to find a single row.

- Type

  Return null if there's no data (spring data jpa)

  in case pure jpa, no result exception occurs

  The null checking conditional is a bad code

  ```
  Member findMemberByUsername(String username);
  ```

  ```
  Member findMember = memberRepository.findMemberByUsername("AAA");
  if(findMember != null) { ... } // bad code!!!
  ```

- Optional

  Return Optional.empty if there's no data -> how to deal with it?

  Exception occurs if there are more than one row

  ```
  Optional<Member> findOptionByUsername(String username);
  ```

  ```
  Optional<Member> option = memberRepository.findOptionByUsername("AAA");
  ```

▼ Collection

Return empty collection if there’s no data.

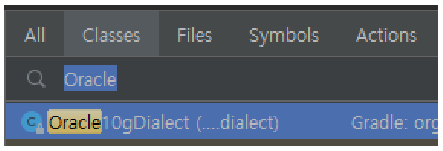
```
List<Member> findListByUsername(String username);
```

```
List<Member> aaa = memberRepository.findListByUsername("AAA");
```

▼ Paging and Sorting

▼ pure JPA

▼ how to find dialect name



```
spring:
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.Oracle10gDialect
        #pakage.className
```

```
public List<Member> findByPage(int age, int offset, int limit) {
  return em.createQuery("select m from Member m where m.age = :age order by m.username desc", Member.class)
      .setParameter("age", age)
      .setFirstResult(offset)
      .setMaxResults(limit)
      .getResultList();
}

public long totalCount(int age) {
  return em.createQuery("select count(m) from Member m where m.age = :age", Long.class)
      .setParameter("age", age)
      .getSingleResult();
}
```

```
public void paging() {
  int age = 10;
  int offset = 0;
  int limit = 3;

  // when
  List<Member> members = memberJpaRepository.findByPage(age, offset, limit);
  long totalCount = memberJpaRepository.totalCount(age);

  // page calculating formula
  /* totalPage == totalCount / size ...
  lastPage ...
  firstPage ...*/

  // then
  assertThat(members.size()).isEqualTo(3);
  assertThat(totalCount).isEqualTo(6);
}
```

- ▼ Spring Data JPA
  - ▼ Paging to slicing and vice versa.

    Without spring jpa, all the codes need to be modified.

    Using data jpa, only thing to do is changing return type.

▼ Page class

NOTE: page start from 0, not 1

- Advantage

No need to make page calculating formula.

Total count is included in Page class

```
// paging and sorting_spring data jpa
Page<Member> findByAge(int age, Pageable pageable);
```

```
// paging and sorting_spring data jpa
@Test
public void paging() {
  memberRepository.save(new Member("member1", 10));
  memberRepository.save(new Member("member2", 10));
  memberRepository.save(new Member("member3", 10));
  memberRepository.save(new Member("member4", 10));
  memberRepository.save(new Member("member5", 10));
  memberRepository.save(new Member("member6", 10));

  int age = 10;
  PageRequest pageRequest = PageRequest.of(0, 3, Sort.Direction.DESC, "username");

  // when
  // Total count is included in Page class
  Page<Member> page = memberRepository.findByAge(age, pageRequest);

  // page calculating formula is not needed

  // invert into dto
  page.map(member-> new MemberDto(member.getId(), member.getUsername(), null));

  // then
  List<Member> content = page.getContent();

  assertThat(content.size()).isEqualTo(3);
  assertThat(page.getTotalElements()).isEqualTo(6);
  assertThat(page.getNumber()).isEqualTo(0);
  assertThat(page.getTotalPages()).isEqualTo(2);
  assertThat(page.isFirst()).isTrue();
  assertThat(page.hasNext()).isTrue();
}
```

▼ Slice class

count X

limit + 1

ex) Mobile list

```
// slice
Slice<Member> findAsSliceByAge(int age, Pageable pageable);
```

```
// slice
@Test
public void slicing() {
  memberRepository.save(new Member("member1", 10));
  memberRepository.save(new Member("member2", 10));
  memberRepository.save(new Member("member3", 10));
  memberRepository.save(new Member("member4", 10));
  memberRepository.save(new Member("member5", 10));
  memberRepository.save(new Member("member6", 10));

  int age = 10;
  PageRequest pageRequest = PageRequest.of(0, 3, Sort.Direction.DESC, "username");

  // when
  // Total count is included in Page class
  Slice<Member> page = memberRepository.findAsSliceByAge(age, pageRequest);

  // page calculating formula is not needed

  // invert into dto
  page.map(member-> new MemberDto(member.getId(), member.getUsername(), null));

  // then
  Slice<Member> content = page.getContent();

  assertThat(content.size()).isEqualTo(3);
  // assertThat(page.getTotalElements()).isEqualTo(6);
  assertThat(page.getNumber()).isEqualTo(0);
  // assertThat(page.getTotalPages()).isEqualTo(2);
  assertThat(page.isFirst()).isTrue();
  assertThat(page.hasNext()).isTrue();
}
```

▼ List class

count x

```
List<Member> findAsSliceByAge(int age, Pageable pageable);
```

▼ count query separation

Important in working-level

By separating, count query can optimized

Getting total count is very heavy and it will be much heavier if unnecessary table or conditions are used.

▼ without separating

```
@Query(value = "select m from Member m left join m.team t")
Page<Member> findCountSeparationByAge(int age, Pageable pageable);
```

```
// data
select
    *
from
    ( select
        member0_.member_id as member_id1_0_,
        member0_.age as age2_0_,
        member0_.team_id as team_id4_0_,
        member0_.username as username3_0_
    from
        member member0_
    left outer join
        team team1_
            on member0_.team_id=team1_.team_id
    order by
        member0_.username desc )
where
    rownum <= ?


// count
select
    count(member0_.member_id) as col_0_0_
from
    member member0_
left outer join
    team team1_
        on member0_.team_id=team1_.team_id
```

▼ separating

can use only needed conditions

```
select
    count(member0_.username) as col_0_0_
from
    member member0_
```

▼ Bulk operation (Update in bulk)

In jpa update is conducted by dirty checking.

However sometimes bulk update is needed.

1. Execute bulk operation in the absence of an entity in persistence context.
2. If an entity is unavoidably present in the persistent context, initialize persistence context immediately after bulk operation

▼ @Modifying

Must use this annotation for update query.

or else, following exception occurs

```
org.hibernate.hql.internal.QueryExecutionRequestException: Not supported for
DML operations
```

```
@Modifying // --> jpa's executeUpdate() call
@Query("update Member m " +
    "   set m.age = :age " +
    "   where m.age >= :age")
int bulkAgePlus(@Param("age")int age);
```

NOTE

Bulk operation ignore persistent context and directly dispatch query to the database.

→ Entity in persistent context may differ from entity in database.

Don't forget to update persistent context if read is followed by bulk update.

▼ How to synchronize entity in both persistent context and database

▼ flush and clear

Same transaction, same EntityManager

'em' declared here is the same EntityManager used in two repositories.

```
@Transactional
class MemberRepositoryTest {

  @Autowired MemberRepository memberRepository;
  @Autowired TeamRepository teamRepository;

  /* It is the same EntityManager used in two repositories above. */
  @PersistenceContext EntityManager em;

  @Test
  public void bulkUpdate() {
    // given
    memberRepository.save(new Member("member1", 10));
    memberRepository.save(new Member("member2", 19));
    memberRepository.save(new Member("member3", 20));
    memberRepository.save(new Member("member4", 21));
    memberRepository.save(new Member("member5", 40));

    // when
    int resultCount = memberRepository.bulkAgePlus(20);
    em.flush();
    em.clear();

    List<Member> result = memberRepository.findByUsername("member5");
    Member member5 = result.get(0);
    System.out.println("member5 = " + member5); // 40

    // then
    assertThat(resultCount).isEqualTo(3);
  }
}
```

▼ @Modifying(claerAutomatically = true)

```
@Modifying(clearAutomatically = true) // --> jpa's executeUpdate() call
@Query("update Member m " +
    "   set m.age = :age " +
    "   where m.age >= :age")
int bulkAgePlus(@Param("age")int age);
```

▼ @EntityGraph

Simple way for fetch join of JPQL (LEFT OUTER JOIN)

Simple read → @EntityGraph.

Complicate query → jqpl with fetch join.

▼ Examples

basic

```
@Override
@EntityGraph(attributePaths = {"team"})
List<Member> findAll();
```

JQPL

```
@EntityGraph(attributePaths = {"team"})
@Query("select m from Member m")
List<Member> findMemberEntityGraph();
```

Query creation by method name

```
@EntityGraph(attributePaths = {"team"})
List<Member> findEntityGraphByUsername(@Param("username") String username);
```

▼ fetchType.LAZY

LAZY type field(team) is made as proxy when the (Member) class is made,

When the field is called, query dispatched and get data.

→ N + 1 problem (I think it should be called 1 + N)

Reading could be not big deal.

It gets on the network when dispatching query and fetching data.

That takes lots of time.

```
public class Member {
  ...

  // proxy로 만들어 놓고 호출 시 가져와서 초기화함
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "team_id")
  private Team team;

  ...
}
```

```
@Test
public void findMemberLazy() {
   // given
   // member1 -> teamA
   // member2 -> teamB
   Team teamA = new Team("teamA");
   Team teamB = new Team("teamB");
   teamRepository.save(teamA);
   teamRepository.save(teamB);

   Member member1 = new Member("member1", 10, teamA);
   Member member2 = new Member("member2", 10, teamB);
   memberRepository.save(member1);
   memberRepository.save(member2);

   em.flush();
   em.clear();

   // when
   List<Member> members = memberRepository.findAll();
   for (Member member : members) {
      System.out.println("member = " + member.getUsername());
      System.out.println(member.getTeam().getClass()); // proxy
      System.out.println("member.teamName = " + member.getTeam().getName());
      System.out.println(member.getTeam().getClass()); // proxy
   }
}
```

▼ Fetch join

N + 1 problem solved.

LAZY type field (Team) is fetched together and it is class not proxy.

```
@Query("select m from Member m left join fetch m.team")
List<Member> findMemberFetchJoin();
```

```
@Test
public void findMemberFetch() {
   ...  // same as 'findMemberLazy()' above

   for (Member member : members) {
      System.out.println("member = " + member.getUsername());
      System.out.println(member.getTeam().getClass()); // class
      System.out.println("member.teamName = " + member.getTeam().getName());
      System.out.println(member.getTeam().getClass()); // class
   }
}
```

▼ @NamedEntityGraph

Impractical

```
@Entity
@NamedEntityGraph(name = "Member.all", attributeNodes = @NamedAttributeNode("team"))
public class Member {
  ...

  @ManyToOne(fetch = FetchType.LAZY) // proxy로 만들어 놓고 호출 시 가져와서 초기화함
  @JoinColumn(name = "team_id")
  private Team team;

  ...
}
```

```
@EntityGraph("Member.all")
List<Member> findEntityGraphByUsername(@Param("username")String username);
```

`

▼ JPA hint (readOnly)

Hint that offered to JPA implementation, hibernate. It’s not a SQL hint

jpa hint to use hibernate methods

- ▼ @QueryHints(value = @QueryHint(name = "org.hibernate.readOnly", value = "true"))

  It doesn’t make snapshot. (I think snapshot is probably second entity for dirty checking)

  Dirty checking takes more memory because it has two entities, original entity and altered entity. Using this hint reading performance can be optimized little bit.

  [NOTE]

  In working-level, It is rarely used for few case that requires a looot of traffic.

  Effect of optimization using this hint is insignificant.

  In many cases, low performance is because of bad query. and if reading performance is really bad, cache like ‘redis’ should be installed.

```
// jpa hint
@QueryHints(value = @QueryHint(name = "org.hibernate.readOnly", value = "true"))
Member findReadOnlyByUsername(String username);
```

```
// JPA hint
@Test
public void queryHint() {
  // given
  Member member1 = new Member("member1", 10);
  memberRepository.save(member1);
  em.flush(); // synchronize database with persistent context
  em.clear(); // empty persistent context

  // when
  // dirty checking
  /*Member findMember = memberRepository.findById(member1.getId()).get();
  findMember.changeUserName("member2");
  em.flush();*/

  Member findMember = memberRepository.findReadOnlyByUsername("member1");
  findMember.changeUserName("member2");
}
```

▼ JPA Lock

Too deep topic, no need to study now.

DO NOT USE lock for the system that has a lot of real-time traffic

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
List<Member> findLockByUsername(@Param("username") String username);
```

```
// sql
select
    member0_.member_id as member_id1_0_,
    member0_.age as age2_0_,
    member0_.team_id as team_id4_0_,
    member0_.username as username3_0_
from
    member member0_
where
    member0_.username=? for update
```

‘for update’ added according to dialect setting.

▼ Implementing a custom repository

▼ custom repository examples

- QueryDSL
- Direct use of JPA (EntityManager)
- Spring JDBC Template
- MyBatis
- Database connection, etc.

▼ Implementing steps

1. Make a interface

   Any repository name

```
public interface MemberRepositoryCustom {
    List<Member> findMemberCustom();
}
```

2. Implement the methods of the new interface

   class name must be [repository interface name] + Impl

```
@RequiredArgsConstructor
public class MemberRepositoryImpl implements MemberRepositoryCustom {

    private final EntityManager em;

    @Override
    public List<Member> findMemberCustom() {
        return em.createQuery("select m from Member m")
                .getResultList();
    }
}
```

3. Extends the new interface.

```
public interface MemberRepository extends JpaRepository<Member, Long>, MemberRepositoryCustom {
  ...
}
```

▼ Auditing

▼ @EnableJpaAuditing

Required to use auditing annotations like

@CreatedDate

@LastModifiedDate

@CreatedBy

@LastModifiedBy

▼ auditorPrivider()

Get and inject the ID for

@CreatedBy

@LastModifiedBy

▼ @EntityListeners(AuditingEntityListener.class)

To notify Event-based operating

- Global setting (xml)

refer to text book page 51

```
@EnableJpaAuditing
@SpringBootApplication
public class DataJpaApplication {

   public static void main(String[] args) {
        SpringApplication.run(DataJpaApplication.class, args);
    }

    @Bean
    public AuditorAware<String> auditorProvider() {
        return () -> Optional.of(UUID.randomUUID().toString());
    }
}
```

Mostly, time is required

```
@EntityListeners(AuditingEntityListener.class)
@MappedSuperclass
@Getter
public class BaseTimeEntity {

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createDate;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

}
```

However, sometimes these data are not needed.

```
@EntityListeners(AuditingEntityListener.class) // To notify Event-based operating
@MappedSuperclass
@Getter
public class BaseEntity extends BaseTimeEntity{

    @CreatedBy
    @Column(updatable = false)
    private String createdBy;

    @LastModifiedBy
    private String lastModifiedBy;
}
```

- ▼ Web extension
  - ▼ Domain class converter

    NOT RECOMMENDED

    find and bind entity object based on ‘entity id’ from http parameter

    ```
    // http://localhost:8080/members/1
    @GetMapping("/members2/{id}")
    public String findMember2(@PathVariable("id") Member member) {
        return member.getUsername();
    }
    ```

- ▼ Paging and sorting

  (provided by spring data jpa)

  they are applicable in spring mvc

  page: starts from 0

  size: the number of data in one page

  - ▼ example

    ```
    @GetMapping("/members")
    public Page<Member> list(Pageable pageable) {
        // http://localhost:8080/members?page=0
        // http://localhost:8080/members?page=1&size=3
        // http://localhost:8080/members?page=1&size=3&sort=id,desc
        // http://localhost:8080/members?page=1&size=3&sort=id,desc&sort=username
        Page<Member> page = memberRepository.findAll(pageable);
        return page;
    }
    ```

  - ▼ configuration
    - ▼ global

      ```
      spring.data.web.pageable.default-page-size=20
      spring.data.web.pageable.max-page-size=2000
      ```

      ```
      data:
        web:
          pageable:
            default-page-size: 10
            max-page-size: 2000
      ```

    - ▼ separate

      @PageableDefualt( )

      ```
      @GetMapping("/members")
      public Page<Member> list(@PageableDefault(size = 5) Pageable pageable) {
          Page<Member> page = memberRepository.findAll(pageable);
          return page;
      }
      ```

- ▼ @Qualifier

  check textbook if needed

- ▼ convert into DTO

  ```
  @GetMapping("/members")
  public Page<MemberDto> list(@PageableDefault(size = 5) Pageable pageable) {
      return memberRepository.findAll(pageable)
              .map(member -> new MemberDto(member.getId(), member.getUsername(), null));
  }
  ```

  DTO can see entity. However an entity must not see DTO.

  ```
  @GetMapping("/members")
  public Page<MemberDto> list(@PageableDefault(size = 5) Pageable pageable) {
      return memberRepository.findAll(pageable)
              .map(member -> new MemberDto(member));
  }
  ```

  method reference

  (alt + enter on ‘new MemberDto’ above)

  ```
  GetMapping("/members")
  public Page<MemberDto> list(@PageableDefault(size = 5) Pageable pageable) {
      return memberRepository.findAll(pageable)
              .map(MemberDto::new);
  }
  ```

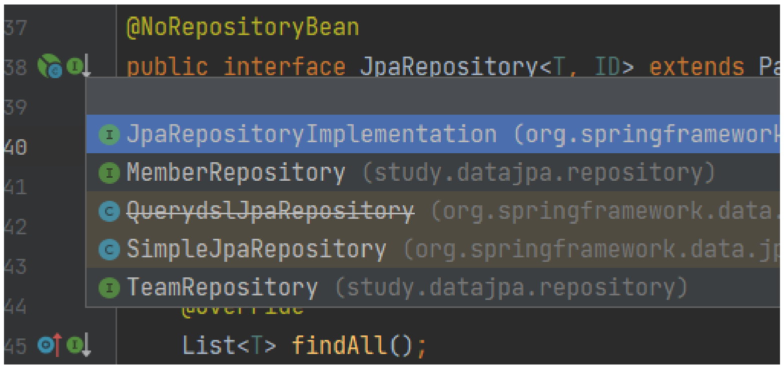
- ▼ Set page from 1

  DO NOT CHAGE (recommended)

  if you really wan’t to change, check the textbook page 55 and lecture

▼ Spring data jpa implementer analysis

- ▼ Check implementer



```
@NoRepositoryBean
public interface JpaRepository<T, ID> extends Pa
```

JpaRepositoryImplementation (org.springframework
MemberRepository (study.datajpa.repository)
~~QuerydslJpaRepository~~ (org.springframework.data.
SimpleJpaRepository (org.springframework.data.jp
TeamRepository (study.datajpa.repository)

```
    List<T> findAll();
```

- ▼ SimpleJpaRepository
  - ▼ @Repository

    Invert subtechnology(jpa, jdbc ...) to abstract exception of spring.

    Exceptions differs from technologies like jdbc, jpa.

    this annotation change those exceptions to the one that is dealt with spring framework.

    which means, when exception returns to service or controller layer, those are spring exception not jdbc or jpa exception.

    In other word, if substructure changes (like jdbc to jpa), mechanism to handle exception is same.

  - ▼ @Transactional

    If transaction doesn’t exist, it is handled here.

    If transaction exists, sub logic succeed in sub logic.

    reaonly=true in class level → because there are many read methods

    individual annotation for create and update

  - ▼ save()

    new entity → persist()

    existence → merger()

▼ The way to distinguish new entity

- JPA ID strategy

    1. @Id + @GeneratedValue

       save() call without id

    2. @Id only

       save() call with id already present.

       → mege() call

       → find from dabatbase. recognize as new entity if not exists (inefficient)

<u>USE @GeneratedValue for ID</u>

If can’t, implement Persistable and modify isNew()

```
@Entity
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Item extends BaseTimeEntity implements Persistable<String>  {

  @Id
  private String id;

  public Item(String id ) {
    this.id = id;
  }

  @Override
  public String getId() {
    return null;
  }

  @Override
  public boolean isNew() {
    return getCreateDate() == null;
  }
}
```

@CreateDate is inserted when persist() call

```
@Test
  public void save() {
    Item item = new Item("A");
    itemRepository.save(item);
  }
```

```
@Transactional
  @Override
  public <S extends T> S save(S entity) {

    Assert.notNull(entity, "Entity must not be null.");
    if (entityInformation.isNew(entity)) {
      em.persist(entity); // @CreateDate is inserted
      return entity;
    } else {
      return em.merge(entity);
    }
  }
```

▼ Other funtions

USE QueryDSL

- ▼ Specifications

  DO NOT USE → USE QueryDSL

- ▼ Query by Example

  DO NOT USE → USE QueryDSL

- ▼ Projections

  Used sometimes → Use QueryDSL

- ▼ Native Query

  DO NOT USE → Use QueryDSL

  When finding as DTO, use

  1. projection with native query
  2. jdbc template
  3. myBatis